RAPPORT DE RECHERCHE

L R I

MECHANICALLY PROVING TERMINATION USING POLYNOMIAL INTERPRETATIONS

CONTEJEAN E / MARCHE C / URBAIN X TOMAS A P

Unité Mixte de Recherche 8623
CNRS-Université Paris Sud – LRI

01/2004

Rapport de Recherche N° 1382

CNRS – Université de Paris Sud
Centre d'Orsay
LABORATOIRE DE RECHERCHE EN INFORMATIQUE
Bâtiment 490
91405 ORSAY Cedex (France)

Mechanically proving termination using polynomial interpretations*

Evelyne Contejean Claude Marché
Xavier Urbain
PCRI — LRI (CNRS UMR 8623) & INRIA Futurs
Bât. 490, Université Paris-Sud, Centre d'Orsay
91405 Orsay Cedex, France

Ana Paula Tomás
Departamento de Ciência de Computadores
Faculdade de Ciências da Universidade do Porto
R. do Campo Alegre, 823
4150-180 Porto, Portugal

January 12, 2004

Abstract

For a long time, term orderings defined by polynomial interpretations have been considered far too restrictive to be used for computer-aided termination proof of TRSs. But recently, the introduction of the dependency pairs approach achieved considerable progress w.r.t. automated termination proof, in particular by requiring from the underlying ordering much weaker properties than the classical approach. As a consequence, the noticeable power of a combination dependency pairs/polynomial orderings yielded a regain of interest for these interpretations.

We describe criteria on polynomial interpretations for them to define weakly monotonic orderings. From these criteria, we obtain new techniques both for mechanically checking termination using a given polynomial interpretation, and for finding such interpretations with full automation. With regards to automated search, we propose an original method for solving Diophantine constraints.

We implemented these techniques into the C*i*ME rewrite tool, and we provide experiments that show how useful polynomial orderings actually are in practice.

1 Introduction

For decades, the use of the standard Manna-Ness criterion (that is each rule decreases w.r.t. a well-founded ordering) dominated amongst the different known methods aimed at proving termination of term rewriting systems. The orderings required by this criterion must have strong properties like strict monotonicity; they are usually distinguished in two classes: *syntactical* orderings and *semantical* orderings. Syntactical orderings rely on a precedence on symbols which is extended to terms while semantical ones make use of an interpretation of

*This research was supported in part by the EWG CCL II, the cooperation CNRS-ICCTI, projects 4312, 5518 and 6777, and the "ATIP C*i*ME du département STIC du CNRS"

terms. Amongst the latter, term orderings defined by polynomial interpretations have been defined in 1979 in a landmark paper of Lankford [28]. The combination of polynomials with the Manna-Ness criterion puts strong restrictions on the class of relations the obtained ordering can contain [9], and for a long time they have been considered much too weak to be used in the practice of proving termination of TRSs.

But recently, considerable progress was achieved on automated termination proof, in particular by use of the dependency pairs method and its termination criteria [1], its applications to incremental/hierarchical termination proofs [2, 41], and to termination under specific strategies such as innermost termination [1, 2] or context-sensitive rewriting [22].

These new techniques demand much weaker properties on the underlying ordering 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 w.r.t. termination proof with the Manna-Ness criterion observed a regain of interest. This is the case for polynomial orderings.

It has been noticed that the situation is partly similar with Knuth-Bendix orderings [21, 24], but not with Recursive Path orderings. In fact, the latter are always strictly monotonic and that is why transformations have been proposed such as *argument filtering* or more generally *recursive program scheme*. Such transformations are in general resources consuming during the proof discovery process. However, adding argument filtering to polynomial orderings is pointless since any ordering defined by an argument filtering and a set of polynomial interpretations can be also defined directly by some other set of interpretations.

This new interest for polynomial interpretations-based orderings conducted us to design a new implementation of these inside the C*i*ME rewrite tool [12]. We combined existing techniques with new improvements, and we describe hereafter the theoretical basis of this implementation, which is able to find polynomial orderings for termination proofs with *full* and *efficient* automation.

A key issue in finding polynomial orderings is to solve some non-linear constraints over natural numbers. In order to improve efficiency, these constraints are linearised thanks to the introduction of abstraction variables, and the problem of minimizing the number of such variables arises. Quite surprisingly, this problem is a generalisation of the well-known problem of computation of *addition chains* [5, 6, 7, 8, 13, 17, 18, 34, 35, 36, 38, 39, 42], which arises naturally when one wants to compute a polynomial expression while minimising the number of multiplications.

This paper is organised as follows. In Section 2, we firstly discuss about the currently known termination criteria which are suitable for automation, and about which properties of term orderings are needed for such criteria. In Section 3, we recall how orderings by polynomial interpretations are defined, and we show that, given a TRS R and a polynomial interpretation, every verification needed to check termination of R reduces to check positiveness of polynomial expressions. Then, we recall known techniques for checking positiveness of such expressions, and give new results about μ-translation of polynomial interpretations. In Section 4, we consider the problem of finding suitable polynomial interpretations with full automation, and present our new method for solving Diophantine constraints arising in such a search. Finally, in Section 5 we comment a few results from a selection of experiments conducted with the C*i*ME system.

2 Termination criteria

We assume the reader familiar with basic notions of term rewriting and termination, especially with the dependency pairs approach; we refer to surveys [3, 16] for details and to Arts & Giesl [1, 2] regarding dependency pairs.

As it is now suitable for dependency pairs approach where both strict and large comparisons of terms occur, we need both strict and large term orderings. We use a slightly modified variant of *ordering pair* or *reduction pair* [27]. For simplicity, since no confusion is possible, we still call such pairs *term orderings*.

Formally, a *term ordering* is a pair $(\succeq, \succ)$ of relations over the set $T(\mathcal{F}, X)$ of terms over signature $\mathcal{F}$ and variables X, such that: 1) $\succeq$ is a quasi-ordering, i.e. reflexive and transitive; 2) $\succ$ is a strict ordering, i.e. irreflexive and transitive, included in $\succeq - \preceq$. Our conditions fulfil the requirements of the very general definition of Kusakari *et al.* [27]. In particular, 1) and 2) imply conditions $\succ \cdot \succeq \subseteq \succ$ and $\succeq \cdot \succ \subseteq \succ$ they require.

A term ordering is said to be *well-founded* if there is no infinite strictly decreasing sequence $t_1 \succ t_2 \succ \cdots$; *stable* if both $\succ$ and $\succeq$ are stable under substitutions, that is for any terms t_1 and t_2 and for any substitution σ, if $t_1 \succ t_2$ then $t_1\sigma \succ t_2\sigma$, and if $t_1 \succeq t_2$ then $t_1\sigma \succeq t_2\sigma$.

For a given symbol f of the signature, of arity $n \geq 1$, we say that a relation $\mathcal{R}$ is monotonic with reference to the i-th argument of f, $1 \leq i \leq n$, if for any terms $t, u, v_1, \ldots, v_{i-1}, v_{i+1}, \ldots, v_n$, $t \mathrel{\mathcal{R}} u$ implies

$$f(v_1, \ldots, v_{i-1}, t, v_{i+1}, \ldots, v_n) \mathrel{\mathcal{R}} f(v_1, \ldots, v_{i-1}, u, v_{i+1}, \ldots, v_n)$$

A term ordering $(\succeq, \succ)$ is *weakly monotonic* if $\succeq$ is monotonic with respect to all arguments of all function symbols; it is *strictly monotonic* if $\succ$ is also monotonic with respect to all arguments of all function symbols. A term ordering $(\succeq, \succ)$ is called a *weak (resp. strict) reduction ordering* if it is well-founded, stable and weakly (resp. strictly) monotonic.

To prove termination of a given TRS R, several possible criteria exist. The simplest one is the *standard* Manna-Ness criterion: if there exists a *strict* reduction ordering $(\succeq, \succ)$ such that $l \succ r$ for each rule $l \to r \in R$, then R is terminating. There are a few variants of *dependency pairs* criteria, the simplest one being: if there exists a *weak* reduction ordering $(\succeq, \succ)$ such that

- for each rule $l \to r \in R$, $l \succeq r$;
- for each dependency pair $\langle u, v \rangle$ of R, $u \succ v$;

then R is terminating. Hence, unlike the standard criterion, the underlying ordering is not required to be strictly monotonic. In fact, a common requirement of dependency pairs criteria consists in relying on the use of a weak reduction ordering, even for criteria based on *estimated dependency graphs* [1, 33].

Regarding innermost termination, there are improvements of dependency pairs criteria [1] where the underlying ordering is not anymore asked to be weakly monotonic with reference to *all* arguments of all symbols in the signature, but to *some* of them only. Thus, defining such orderings makes an issue.

Finally, another usual concern in the practice of termination is rewriting modulo an equational theory E, like commutativity (C) or associativity and commutativity (AC). In such a case, the underlying ordering must be *compatible* with the aforementioned theory, that is $s' \succ t'$ whenever $s \succ t$, $s =_E s'$ and $t =_E t'$, and similarly for $\succeq$.

3 Term orderings defined by polynomial interpretations

We will now focus on term orderings defined by polynomial interpretations. In Section 3.1, we define orderings based on arbitrary interpretations and we show which conditions they must satisfy to guarantee, on the generated ordering, the properties listed in the previous section. In Section 3.2, we focus on polynomial interpretations, and we show that all these conditions can be reduced to positiveness of some polynomials. Then, in Section 3.3, we summarise the known methods for checking positiveness. Those are still complex and in Section 3.4 we propose a new technique of *translation* of interpretations, which eventually allows to reduce checkings of conditions required on polynomials to very simple tests on positiveness of their coefficients.

3.1 Orderings defined by interpretations

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

Definition 3.1 *Let ϕ be a function which maps each ground term $t \in T(\mathcal{F})$ to an element of D. The term ordering $(\succeq_\phi, \succ_\phi)$ generated by ϕ is defined by*

$$\begin{array}{lll} t_1 \succeq_\phi t_2 & \textit{iff} & \phi(t_1) \geq_D \phi(t_2) \\ t_1 \succ_\phi t_2 & \textit{iff} & \phi(t_1) >_D \phi(t_2) \end{array}$$

Lemma 3.2 *If $>_D$ is well-founded then $(\succeq_\phi, \succ_\phi)$ is a well-founded term ordering on ground terms.*

Proof. If $(\succeq_\phi, \succ_\phi)$ was not well-founded, there would be an infinite decreasing sequence $t_1 \succ_\phi t_2 \succ_\phi t_3 \succ_\phi \cdots$ that is, by definition, $\phi(t_1) >_D \phi(t_2) >_D \phi(t_3) >_D \cdots$ Hence, $>_D$ would not be well-founded. □

Now, we want 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 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

$$\begin{array}{lll} f \geq_{D,X} g & \text{iff} & \text{for all } I \in X \to D, f(I) \geq_D g(I) \\ f >_{D,X} g & \text{iff} & \text{for all } I \in X \to D, f(I) >_D g(I) \end{array}$$

We should point out that $>_{D,X}$ is *not* $\geq_{D,X} - \leq_{D,X}$, and in some sense, that is why term orderings as ordering pairs are needed.

Now, automation of such an ordering only relies on the automation of this ordering on functions. We shall see in Section 3.3 how to automate that ordering in the special case of D being a set of integers.

Definition 3.3 *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 ordering generated by ϕ is defined by*

$$\begin{array}{lll} t_1 \succeq_\phi t_2 & \textit{iff} & \phi(t_1) \geq_{D,X} \phi(t_2) \\ t_1 \succ_\phi t_2 & \textit{iff} & \phi(t_1) >_{D,X} \phi(t_2) \end{array}$$

Example. Let D be the set $\mathbb{N}$ of natural numbers and let $\geq_D$ be the standard ordering $\geq$ on $\mathbb{N}$. Let us consider signature $\mathcal{F} = \{a, f\}$ where a is a constant and f is of arity 2.

An interpretation ϕ can map a term like $f(f(a,x),y)$ into, say, $2^x + 2y + 3$. That means precisely that given any nonnegative integer values $I(x)$ and $I(y)$ for x and y:

$$\phi(f(f(a,x),y))(I) = 2^{I(x)} + 2I(y) + 3$$

Moreover, if we interpret $f(x,a)$ into $x + 4$, then we have $f(f(a,x),y) \succeq_\phi f(x,a)$ since $2^n + 2m + 3 \geq n + 4$ for all $n, m \in \mathbb{N}$ (because $2^n \geq n + 1$). On the other hand, $f(f(a,x),y) \not\succ_\phi f(x,a)$ because when $I(x) = I(y) = 0$ both terms are interpreted as 4.

Lemma 3.4 *If $>_D$ is well-founded then $(\succeq_\phi, \succ_\phi)$ is well-founded on non-ground terms.*

Proof. Proof is similar to that of previous lemma, but additionally we have to show that $>_{D,X}$ is well-founded itself. If it was not, there would be an infinite decreasing sequence $f_1 >_{D,X} f_2 >_{D,X} f_3 >_{D,X} \ldots$, but then for an arbitrary interpretation I of variables, we would have an infinite decreasing sequence $f_1(I) >_D f_2(I) >_D f_3(I) >_D \ldots$ of elements of D, leading to a contradiction. $\Box$

Definition 3.5 *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 $I \in X \to D$,*

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

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

Lemma 3.6 *Let ϕ be any homomorphic interpretation. For any substitution σ and any valuation I, let us denote $\phi(\sigma, I)$ the valuation mapping any variable x to $\phi(x\sigma)(I)$. Then for any term t, $\phi(t\sigma)(I) = \phi(t)\phi(\sigma, I)$.*

Proof. By structural induction on t. If $t = f(t_1,\ldots,t_n)$ then

$$\begin{array}{rcl} \phi(t\sigma)(I) & = & \phi(f(t_1\sigma,\ldots,t_n\sigma))(I) \\ & = & [\![f]\!]_\phi(\phi(t_1\sigma)(I),\ldots,\phi(t_n\sigma)(I)) \\ & = & [\![f]\!]_\phi(\phi(t_1)\phi(\sigma,I),\ldots,\phi(t_n)\phi(\sigma,I)) \text{ by induction} \\ & = & \phi(f(t_1,\ldots,t_n))\phi(\sigma,I) \end{array}$$

and if t is a variable x, $\phi(x)\phi(\sigma,I) = \phi(\sigma,I)(x) = \phi(x\sigma)(I)$ $\Box$

Lemma 3.7 *If ϕ is an homomorphic interpretation, then $(\geq_\phi, >_\phi)$ is stable.*

Proof. Let t_1 and t_2 be two terms such that $t_1 \succ_\phi t_2$, that is by definition $\phi(t_1) >_{D,X} \phi(t_2)$, i.e.

$$\text{for all } I \in X \to D,\ \phi(t_1)(I) >_D \phi(t_2)(I) \quad (1)$$

Let σ be any substitution. Then for all $I \in X \to D$,

$$\begin{array}{rcll} \phi(t_1\sigma)(I) & = & \phi(t_1)\phi(\sigma, I) & \\ & >_D & \phi(t_2)\phi(\sigma, I) & \text{by (1)} \\ & = & \phi(t_2\sigma)(I) & \end{array}$$

hence $t_1\sigma \succ_\phi t_2\sigma$. the same proof holds for $\succeq_\phi$. $\square$

Lemma 3.8 *For any symbol f of arity n, and $1 \leq i \leq n$, if for all $d_1, \ldots, d_{i-1}, d_{i+1}, \ldots, d_n$ in D, $[\![f]\!](d_1, \ldots, d_{i-1}, x, d_{i+1}, \ldots, d_n)$ is weakly (resp. strictly) increasing in x then $\geq_\phi$ (resp. $>_\phi$) is monotonic with respect to i-th argument of f.*

Proof. Straightforward.$\square$

Example. (Continued) Let us consider $[\![f]\!](x, y) = xy + 1$ and $[\![a]\!] = 0$. The generated ordering is weakly but not strictly monotonic, since $[\![f]\!]$ is not strictly increasing in x when $y = 0$. For instance $f(a, a) \succ_\phi a$ (since $\phi(f(a, a)) = 1 > 0 = \phi(a)$), but $f(f(a, a), a) \not\succ_\phi f(a, a)$ (since $\phi(f(f(a, a), a)) = 1 \times 0 + 1 = 1 = \phi(f(a, a))$).

3.2 Interpretations over integers

So as 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 3.9 *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 precise the value of μ.*

An arithmetic homomorphic interpretation is called a polynomial *interpretation if for all $f \in \mathcal{F}$, $[\![f]\!]$ is a polynomial function.*

To check whether a given polynomial interpretation is suitable for proving termination of a given TRS using any of the criteria mentioned in Section 2, we 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 $t_1 \succ_\phi t_2$.

Finally, for the case of rewriting modulo a theory E, we have to be able to check that

4. $\succeq$ and $\succ$ are compatible with E.

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 weakly increasing in its i-th argument if and only if polynomial

$$\begin{array}{rcl} Q(X_1,\ldots,X_n) & = & P(X_1,\ldots,X_{i-1},X_i+1,X_{i+1},\ldots,X_n) \\ & & -\,P(X_1,\ldots,X_{i-1},X_i,X_{i+1},\ldots,X_n) \end{array}$$

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

$$\begin{array}{rcl} Q(X_1,\ldots,X_n) & = & P(X_1,\ldots,X_{i-1},X_i+1,X_{i+1},\ldots,X_n) \\ & & -\,P(X_1,\ldots,X_{i-1},X_i,X_{i+1},\ldots,X_n)-1 \end{array}$$

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 $t_1 \succ_\phi t_2$ if and only if polynomial P_1-P_2-1 is nonnegative on D_μ^n.

Regarding Item (4), it is sufficient to check that for each equation $t \simeq u$ of theory E, we have $t \succeq u$ and $u \succeq t$, that is $\phi(t)-\phi(u)=0$. For the case of AC, Ben Cherifa and Lescanne [4] showed a simplified sufficient condition: a polynomial interpretation $[\![f]\!](x,y)$ generates an ordering compatible with associativity and commutativity of f if and only if it has the form $axy+b(x+y)+c$ where $b^2=b+ac$.

Example. Here is one of the examples given by Lankford [28]. It is a rewrite system for an endomorphism on a monoid.

$$\begin{array}{rcl} (x\times y)\times z & \to & x\times(y\times z) \\ f(x)\times f(y) & \to & f(x\times y) \\ f(x)\times(f(y)\times z) & \to & f(x\times y)\times z \end{array}$$

To prove termination of this system using the standard Manna-Ness criterion, Lankford proposes the following interpretation, with $\mu=1$:

$$\begin{array}{rcl} [\![f]\!](x) & = & 2x \text{ and} \\ [\![\times]\!](x,y) & = & xy+x \end{array}$$

Let us check all needed conditions:

1. Firstly, to check that $[\![f]\!]$ effectively maps D_1 into D_1 we check that $[\![f]\!](x)-1 = 2x-1$ is nonnegative when $x\geq 1$. Similarly, for $[\![\times]\!]$, we check that $[\![\times]\!](x,y)-1 = xy+x-1$ is nonnegative when $x,y\geq 1$.

2. Secondly, to check that $[\![f]\!]$ strictly increases in its argument, we check that

$$\begin{array}{rcl} [\![f]\!](x+1)-[\![f]\!](x)-1 & = & 2(x+1)-2x-1 \\ & = & 1 \end{array}$$

is nonnegative when $x \geq 1$. To check that $[\![\times]\!]$ strictly increases in its first argument, we check that

$$\begin{array}{rl} & [\![\times]\!](x+1,y) - [\![\times]\!](x,y) - 1 \\ = & ((x+1)y + (x+1)) - (xy + x) - 1 \\ = & xy + y + x + 1 - xy - x - 1 \\ = & y \end{array}$$

is nonnegative when $x, y \geq 1$. Finally, to check that $[\![\times]\!]$ strictly increases in its second argument, we check that

$$\begin{array}{rcl} [\![\times]\!](x,y+1) - [\![\times]\!](x,y) - 1 & = & (x(y+1) + x) - (xy + x) - 1 \\ & = & xy + x + x - xy - x - 1 \\ & = & x - 1 \end{array}$$

is nonnegative when $x, y \geq 1$.

3. And thirdly, we have to check that for each rule, the left-hand side is strictly greater than the right-hand side. For the first rule we have

$$\begin{array}{rcl} [\![(x \times y) \times z]\!] & = & (xy + x)z + (xy + x) \\ [\![x \times (y \times z)]\!] & = & x(yz + y) + x \end{array}$$

hence

$$\begin{array}{rl} & [\![(x \times y) \times z]\!] - [\![x \times (y \times z)]\!] - 1 \\ = & [(xy + x)z + (xy + x)] - [x(yz + y) + x] - 1 \\ = & xyz + xz + xy + x - xyz - xy - x - 1 \\ = & xz - 1 \end{array}$$

is nonnegative when $x, y, z \geq 1$. For the second rule we have

$$\begin{array}{rcl} [\![f(x) \times f(y)]\!] & = & (2x)(2y) + 2x \\ [\![f(x \times y)]\!] & = & 2(xy + x) \end{array}$$

hence

$$\begin{array}{rcl} [\![f(x) \times f(y)]\!] - [\![f(x \times y)]\!] - 1 & = & [4xy + 2x] - [2(xy + x)] - 1 \\ & = & 2xy - 1 \end{array}$$

is nonnegative when $x, y \geq 1$. For the third rule we have

$$\begin{array}{rcl} [\![f(x) \times (f(y) \times z)]\!] & = & 2x(2yz + 2y) + 2x \\ [\![f(x \times y) \times z]\!] & = & 2(xy + x)z + 2(xy + x) \end{array}$$

hence

$$\begin{array}{rl} & [\![f(x) \times (f(y) \times z)]\!] - [\![f(x \times y) \times z]\!] - 1 \\ = & [2x(2yz + 2y) + 2x] - [2(xy + x)z + 2(xy + x)] - 1 \\ = & 4xyz + 4xy + 2x - 2xyz - 2xz - 2xy - 2x - 1 \\ = & 2xyz + 2xy - 2xz - 1 \end{array}$$

which, after proper factorisation $2xz(y-1) + (2xy-1)$, is easily proven nonnegative when $x, y, z \geq 1$.

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.

However, as illustrated by this last check where a smart factorisation was required to somehow dominate the negative coefficients, checking whether a polynomial is nonnegative or not is far from being a simple task. We shall address this problem in the next section.

3.3 Testing positiveness of polynomial functions

We focus now on the *positiveness problem* of polynomial functions that is: given a polynomial $P \in \mathbb{Z}[X_1, \ldots, X_n]$, prove that $P(x_1, \ldots, x_n) \geq 0$ for any value $x_i \geq \mu$. We shall remark firstly that this problem is undecidable in general since Hilbert's Tenth Problem can be reduced to it [32].

Testing positiveness, in the context of automated termination proof, has been studied by several authors [4, 20, 40]. 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 TRSs.

Recently, Hong & Jakuš [25] made a comparison between all these methods while proposing new ones. Very briefly: they showed that all previous methods were *at most* equivalent to a condition using *μ-absolute positiveness*.

Definition 3.10 *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.

Note that this is not exactly the definition of Hong & Jakuš: they regard *strict* positiveness, which can be obtained by considering polynomial $Q - 1$ instead of Q in our definition. A straightforward sufficient condition for positiveness (adapted from Hong & Jakuš [25]) is the following:

Lemma 3.11 *If P is μ-absolutely positive, then it is nonnegative for all values in D_μ of its variables.*

Proof. If P has n variables, let $k_1, \ldots, k_n$ be arbitrary integers greater or equal to μ, then

$$P(k_1, \ldots, k_n) = Q(k_1 - \mu, \ldots, k_n - \mu)$$

is nonnegative since it is an expression without any negative operations. □

This seems to be a nice and simple check to perform. However, computing the "translated" polynomial Q above can be quite costly in general, as shown in the example below. More precisely, Hong & Jakuš showed that the algorithmic complexity of computing translation is the same as for a method by Giesl [20] which computes successives derivatives.

Example. Let us consider the last rule in Lankford's example, Section 3.2. We have to check whether

$$P(x, y, z) = 2xyz + 2xy - 2xz - 1 \geq 0$$

for each $x, y, z \geq 1$. We compute

$$\begin{array}{rl} & P(x+1, y+1, z+1) \\ = & 2(x+1)(y+1)(z+1) + 2(x+1)(y+1) - 2(x+1)(z+1) - 1 \\ = & 2xyz + 2xy + 2yz + 2xz + 2x + 2y + 2z + 2 \\ & \quad + 2xy + 2x + 2y + 2 - 2xz - 2x - 2z - 2 - 1 \\ = & 2xyz + 4xy + 2yz + 2x + 4y + 1 \end{array}$$

which has nonnegative coefficients only. Nevertheless, since this method is at least as powerful as the former ones while being quite simple, it will be our choice. But in fact, we are going to improve it a bit in our context, as we shall see in next subsection.

3.4 Computing μ-translation in advance

We will start from the easy remark that if we have $\mu = 0$, the previous positiveness test based on absolute positiveness becomes completely trivial. Hence taking $\mu = 0$ as often as possible seems to be a good choice.

A natural question is: "Is it *always* possible to choose $\mu = 0$?" The answer is "Yes" indeed, and the proof is surprisingly easy.

Proposition 3.12 *Let* $(\succeq_\phi, \succ_\phi)$ *be a term ordering defined by polynomial interpretations with a given* μ*. Then this ordering can also be defined by some polynomial interpretations with* $\mu = 0$*.*

Proof. Assuming given a polynomial μ-interpretation ϕ, let us define interpretation ϕ_0 by

$$[\![f]\!]_{\phi_0}(x_1, \ldots, x_n) = [\![f]\!]_\phi(x_1 + \mu, \ldots, x_n + \mu) - \mu$$

where ϕ_0 is the newly defined 0-interpretation on terms. By an easy structural induction, we have for any ground term t

$$\phi_0(t) = \phi(t) - \mu$$

hence we have immediately, for any ground terms t_1 and t_2, $t_1 \succeq_{\phi_0} t_2$ iff $t_1 \succeq_\phi t_2$, and the same for $\succ_{\phi_0}$. On non-ground terms, we have to take care of valuations of variables: indeed, there is a one-to-one correspondance T between valuations in D_0 and valuations in D_μ, defined by $T(I)(x) = I(x) + \mu$, and we have for any non-ground term t, by structural induction:

$$\phi_0(t)(I) = \phi(t)(T(I)) - \mu$$

Hence we have, for any non-ground terms t_1 and t_2, $t_1 \succeq_{\phi_0} t_2$ iff $t_1 \succeq_\phi t_2$, and the same for $\succ_{\phi_0}$. So both interpretations define the same ordering.$\square$

The consequence is double. Firstly, it shows that if we want to *search for* a polynomial interpretation automatically, fixing $\mu = 0$ is enough. This fact will be used in the next section. Secondly and regarding implementation, in order to avoid the computation cost of μ-translation of some polynomial P each time we want to check its positiveness, it is much efficient to compute the μ-translation of interpretations *once and for all*.

Example. Again with Lankford's example, we can compute the translations of interpretations of f and $\times$. We define the new interpretations as

$$\begin{array}{rcl} [\![f]\!]_{\phi_0}(x) & = & [\![f]\!]_\phi(x+1) - 1 = 2(x+1) - 1 = 2x + 1 \\ [\![\times]\!]_{\phi_0}(x, y) & = & [\![\times]\!]_\phi(x+1, y+1) - 1 \\ & = & (x+1)(y+1) + (x+1) - 1 = xy + 2x + y + 1 \end{array}$$

All further computations will be done with these new interpretations, and checking positiveness will be done simply by examining positiveness of coefficients only.

Finally, we end this section by giving a simplified criterion for weak or strict monotonicity of the generated ordering, when $\mu = 0$.

Lemma 3.13 *A polynomial* 0*-interpretation* $P(x_1, \ldots, x_n)$ *with nonnegative coefficients is always weakly increasing in each of its arguments. It is strictly increasing in its* i*-th argument if and only if the coefficient of* x_i *is positive.*

Proof. Straightforward.□

Example. With Lankford's example, the new interpretations given for $\mu = 0$ generate a strictly monotonic ordering, since coefficient of x in $[\![f]\!]_{\phi_0}$ is 2 and coefficients of x and y in $[\![\times]\!]_{\phi_0}$ are 2 and 1 respectively.

4 Automated search for polynomial interpretations

This section is devoted to methods for *searching* suitable polynomial interpretations for proving termination of a given TRS. As shown in the previous section, we may look for 0-interpretations only, without any loss of generality. And for such interpretations, reducing positiveness of a polynomial to positiveness of each of its coefficients is a correct (yet incomplete) method which is at least as powerful as other methods known in the literature.

The first step, done in Section 4.1, is to fix a bound on the degree of polynomials we search for. On that respect, we follow Steinbach classification [40]. Such a bound being fixed, we show that searching for interpretations reduces to solving Diophantine constraints. However, this problem is still undecidable [32].

Thus, in Section 4.2, we discuss on partial methods for solving such constraints. As in the first step, we need to fix a bound on the values of variables we search for: to make the problem decidable, we are reduced to solving constraints over a finite domain. We show in Section 4.3 how known methods for such constraints can be instantiated into solving Diophantine constraints. Then, in Section 4.4, we address practical problems arising in implementation, that is the too high algorithmic complexity.

4.1 Parametric polynomial interpretations

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 degree of the polynomials we search for.

Steinbach classified restricted forms of multivariate polynomials [40]. The *linear* class contains polynomials of degree 1 at most, that is

$$P(x_1, \ldots, x_n) = a_1 x_1 + \cdots + a_n x_n + c$$

The *simple* class contains polynomials of at most degree 1 in each variable:

$$P(x_1, \ldots, x_n) = \sum_{i_j \in \{0,1\}} a_{i_1,\ldots,i_n} x_1^{i_1} \cdots x_n^{i_n}$$

The *simple-mixed* class contains polynomials whose monomials consist of either a single variable with degree 2, or of several variables of at most degree 1:

$$P(x_1, \ldots, x_n) = \sum_{i_j \in \{0,1\}} a_{i_1,\ldots,i_n} x_1^{i_1} \cdots x_n^{i_n} + \sum_{1 \leq i \leq n} b_i x_i^2$$

This terminology is used 'as is' in our implementation, to select a given class. We added the *quadratic* class for polynomials of degree 2, an extension to the *simple-mixed* class:

$$P(x_1, \ldots, x_n) = \sum_{i_j \in \{0,1,2\}} a_{i_1,\ldots,i_n} x_1^{i_1} \cdots x_n^{i_n}$$

This classification is slightly overridden when commutative or associative-commutative symbols are involved (those are the only equational theories supported in our implementation). For AC symbols, we always choose a parametric interpretation of the form $axy + b(x+y)+c$ where $b^2 = b + ac$. For a commutative but not associative symbol f, we should use a polynomial $[\![f]\!]$ such that $[\![f]\!](x, y) = [\![f]\!](y, x)$, hence the special form of linear polynomials $[\![f]\!](x, y) = a(x + y) + b$, simple polynomials $[\![f]\!](x, y) = axy + b(x + y) + c$, and simple-mixed or quadratic polynomials $[\![f]\!](x, y) = a(x^2 + y^2) + bxy + c(x + y) + d$.

Once a class of polynomials is chosen, we have a finite number of variables. Now we have to translate, into constraints on these variables, each of the conditions that ensure suitability of the defined ordering $(\succeq, \succ)$. We detail this process below. In the following, $P \geq 0$ means each coefficient of P is nonnegative, and $P = 0$ means each coefficient is null.

Firstly, we shall point out that the requirement for $\succeq$ monotonic, $\succ$ and $\succeq$ stable, and $\succ$ well-founded is satisfied as soon as all coefficients are nonnegative. The reduction of the remaining conditions to constraints is as follows:

- $\succ$ monotonic w.r.t. ith arg. of f, reduces to

  $$a_i \geq 1$$

  if a_i is the coefficient of x_i in $[\![f]\!]$.

- $\succeq$ and $\succ$ compatible with an equational theory E reduces to

  $$[\![t]\!] - [\![u]\!] = 0$$

  for each equation $t \simeq u$ of E.

- For the special case of commutativity: $\succeq$ and $\succ$ C-compatible w.r.t. f reduces to nothing if a symmetric parametric interpretation is chosen as above.

- For the AC case, $\succeq$ and $\succ$ AC-compatible w.r.t. f, reduces to

  $$b^2 = b + ac$$

  if the parametric interpretation of f is $[\![f]\!](x, y) = axy + b(x + y) + c$.

- $t_1 \succeq t_2$ reduces to

  $$[\![t_1]\!] - [\![t_2]\!] \geq 0$$

- $t_1 \succ t_2$ reduces to
$$[\![t_1]\!] - [\![t_2]\!] - 1 \geq 0$$

Hence, at this step, proving termination of a given TRS has been reduced to the problem of solving a set of Diophantine constraints on the coefficients introduced in the parametric interpretations.

Example. Back to Lankford's example, if we want to automatically find suitable polynomial interpretations, we may try parametric simple interpretations

$$\begin{array}{rcl} [\![f]\!](x) & = & ax+b \\ [\![\times]\!](x,y) & = & cxy+dx+ey+f \end{array}$$

Thus, proving termination of the system reduces to solving the following constraints (the first three coming from strict monotonicity conditions):

$$\begin{array}{l} a > 0 \qquad d > 0 \qquad e > 0 \\ c(cxy+dx+ey+f)z + d(cxy+dx+ey+f) + ez + f \\ \qquad\qquad > \quad cx(cyz+dy+ez+f) + dx + e(cyz+dy+ez+f) + f \\ c(ax+b)(ay+b) + d(ax+b) + e(ay+b) + f \\ \qquad\qquad > \quad a(cxy+dx+ey+f) + b \\ c(ax+b)(c(ay+b)z + d(ay+b) + ez + f) + d(ax+b) + \\ e(c(ay+b)z + d(ay+b) + ez + f) + f \\ \qquad\qquad > \quad c(a(cxy+dx+ey+f)+b)z + \\ \qquad\qquad\qquad d(a(cxy+dx+ey+f)+b) + ez + f \end{array}$$

The last three, after normalisation, become:

$$\begin{array}{rcl} (cd-ce)xz + (d^2-cf-d)x + (e-e^2+fc)z + (df-ef-1) & \geq & 0 \\ (ca^2-ca)xy + (cab)x + (cab)y + (cb^2-fa+f+eb+db-b-1) & \geq & 0 \\ (c^2a^2-c^2a)xyz + (dca^2-dca)xy + (eca-dca+c^2ba)xz & & \\ +(fca-d^2a+dcba+da)x + (e^2-fca+2ecb-e+c^2b^2-cb)z & & \\ +(c^2ba)yz + (dcba)y + (fe-fda+fcb+edb+dcb^2-1) & \geq & 0 \end{array}$$

Hence, by the simple criterion of absolute positiveness, proving termination reduces to solving:

$$\begin{array}{rclcrcl} a-1 & \geq & 0 & \qquad & cb^2-fa+f+eb+db-b-1 & \geq & 0 \\ d-1 & \geq & 0 & & c^2a^2-c^2a & \geq & 0 \\ e-1 & \geq & 0 & & dca^2-dca & \geq & 0 \\ cd-ce & \geq & 0 & & eca-dca+c^2ba & \geq & 0 \\ d^2-cf-d & \geq & 0 & & c^2ba & \geq & 0 \\ e-e^2+fc & \geq & 0 & & fca-d^2a+dcba+da & \geq & 0 \\ df-ef-1 & \geq & 0 & & e^2-fca+2ecb-e+c^2b^2-cb & \geq & 0 \\ ca^2-ca & \geq & 0 & & dcba & \geq & 0 \\ cab & \geq & 0 & & fe-fda+fcb+edb+dcb^2-1 & \geq & 0 \end{array}$$

$solve(C : \text{constraint}, B : \text{integer}) =$
  let $S = \{x.\text{min} \leftarrow 0, x.\text{max} \leftarrow B \mid x \in \text{Var}(C)\}$ in
  $branch(S, C)$

$branch(S : \text{store}, C : \text{constraint}) =$
  let $S = propagate(S, C)$ in
  if for each x in $S, x.\text{min} = x.\text{max}$ then
    if $isSolution(S, C)$ then output "solution found : " s
    else throw NoSolution
  else
  let $x = chooseVar(S)$ in
  try
  let $S_1 = \{S \text{ with } x.\text{max} \leftarrow S.x.\text{min}\}$ in $branch(S_1, C)$
  catch NoSolution $\rightarrow$
  let $S_2 = \{S \text{ with } x.\text{min} \leftarrow S.x.\text{min} + 1\}$ in $branch(S_2, C)$

Figure 1: Main algorithm for solving Diophantine constraints

4.2 Solving Diophantine constraints: general idea

The general idea is, firstly, 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 [11, 26]. The usual way of solving such constraints is a generalisation of the well-known Davis-Putnam procedure for deciding satisfiability of propositional formulae, which are formulae where variables lie in the finite domain $\{true, false\}$. The general shape of the solving algorithm is made of two parts, working on a data structure called *store* which tells which values are possible for each variable. The first part is the constraint propagation procedure which, given a store and a constraint, performs some logical deductions to produce a smaller store. The second part is a non-deterministic branching which explores all possible values of variables, with various heuristics.

When specialised to solving constraints in an integer domain $[0, B]$, it is handy to have a store which memorises only the minimum and the maximum values of variables, leading to the main algorithm given Figure 1. In that algorithm, *propagate* is the constraint propagation procedure: it is supposed to perform arbitrary correct deductions from a given store and given constraints: $propagate(s, C)$ returns a store s', included in s (that is for each variable x, $s'.x.min \geq s.x.min$ and $s'.x.max \leq s.x.max$) such that any solution of C inside s is also inside s'. Note that it may also throw NoSolution if it deduces that no solution exists. Procedure $chooseVar(s)$ chooses a variable on which a reasoning by cases will be done. It should return any variable x such that $x.min < x.max$. Finally, *isSolution* is a procedure which, given a store where each variable is associated to a single value, tells whether this store is a solution or not. Notice that the algorithm given Figure 1 uses a particular branching strategy: once a variable x is chosen, two cases are considered, the first one occurs when x is equal to its minimum possible value, the second one occurs when it is greater than that. Any other branching strategy is possible, such as domain bisection, but the one chosen here gives better results in practice.

It is straightforward to see that this algorithm will end in finite time whatever the implementations of *propagate* and *chooseVar* might be. But of course, the efficiency highly depends on clever implementations of these two subroutines: for example, if *propagate* does not deduce anything and simply returns the store given as argument, then the algorithm will explore exhaustively the set $[0, B]^n$ (n number of variables) of possible solutions. We will not discuss further on possible implementation of *chooseVar*. In C*i*ME, *chooseVar* implements the following heuristic: a variable with the smallest min is chosen, amongst all variables with the minimal min, a variable with the largest value of (max − min) is chosen, and again amongst all possible remaining variables, the variable which occurs most often in the constraints is chosen.

Example. Let's consider the set of Diophantine constraints

$$2x + y \leq 12\ ,\ xy = 15$$

and assume we want to solve it for x, y in interval $[0, 100]$. We first build the initial store

x	0	100
y	0	100

We may then propagate the constraints: from $2x + y \leq 12$, we deduce $y \leq 12 - 2x \leq 12$. From $2x + y \leq 12$ again, we deduce $x \leq \lfloor\frac{12-y}{2}\rfloor \leq \lfloor\frac{12}{2}\rfloor = 6$ (we denote $\lfloor a\rfloor$ the greatest integer less than or equal to a). Hence the store becomes

x	0	6
y	0	12

Furthermore, from $x.y = 15$, $x \geq \lceil\frac{15}{y}\rceil \geq \lceil\frac{15}{12}\rceil = 2$ (we denote $\lceil a\rceil$ the smallest greater than or equal to a), and $y \geq \lceil\frac{15}{x}\rceil \geq \lceil\frac{15}{6}\rceil = 3$ hence

x	2	6
y	3	12

Considering again $2x + y \leq 12$, we get $x \leq \lfloor\frac{12-3}{2}\rfloor = 4$ and $y \leq 12 - 2 \times 2 = 8$, and again from $x.y = 15$, we have $y \geq \lceil\frac{15}{x}\rceil \geq \lceil\frac{15}{4}\rceil = 4$ hence we get

x	2	4
y	4	8

and we cannot deduce more on intervals for x and y, so this last store is the result of propagation of the constraints on the initial store. We have then to reason by cases. We choose one of the variables, say x, and branch into two cases: whether x is equal to its minimum value in the store or not. If we fix $x = 2$ we get the store

x	2	2
y	4	8

and propagation of $\lceil\frac{15}{x}\rceil \leq y \leq \lfloor\frac{15}{x}\rfloor$ leads to an inconsistent store

x	2	2
y	8	7

Hence exception NoSolution may be thrown. Backtracking to the last branching point, we know that $x \neq 2$ and get the store

x	3	4
y	4	8

which, after propagation, becomes

x	3	4
y	4	5

Choosing then the value $x = 3$ and propagating the constraints leads to the solution $x = 3$, $y = 5$.

4.3 Translating Diophantine constraints into finite domain constraints

The next goal is to make constraint propagation efficient. The main idea, coming from constraint logic programming and implicitly used in the previous example, is to transform the constraints into so-called *finite domain constraints* of the form $x \in [e_1, e_2]$ where e_1 and e_2 are expressions. Constraint propagation will then be done by computing minimal value of e_1 and maximal value of e_2 and comparing them with minimal and maximal values of x.
Example. The constraints $\{2x + y \leq 12, xy = 15\}$ will be transformed into finite domain constraints:

$$\begin{array}{rclcrcl} x & \in & [0, (12 - y)/2] & \quad & x & \in & [15/y, 15/y] \\ y & \in & [0, 12 - 2 \times x] & \quad & y & \in & [15/x, 15/x] \end{array}$$

This approach is quite well-known indeed for *linear* Diophantine constraints [11] which is a major case in applications of constraints logic programming. However, the case of *non-linear* Diophantine constraints seems not to to have been studied. So we designed a specialised variant of finite domain constraints to fit our needs, which we describe now. A problem arising when putting Diophantine constraints into a form $x \in [e_1, e_2]$ is that we need to use divisions (as in the example above) and/or n-th roots. Thus, we have to keep in mind the semantics of such expressions and neither divide by zero nor take the root of a negative number.

Definition 4.1 *A* finite domain Diophantine constraint *is a formula of the form* $x \in [e_1, e_2]$ *where* e_1 *and* e_2 *are* finite domain Diophantine expressions, *of the form*

$$\sqrt[n]{(f - f)/f}$$

where n *denotes a positive integer, and* f *denotes* positive polynomial expressions *as defined by the grammar*

$$f \quad ::= \quad n \mid x \mid f + f \mid f \times f$$

where n *denotes a nonnegative integer constant and* x *a variable.*

For readability, we simply write e for $e - 0$, $\sqrt[1]{e}$ and $e/1$.

We define what is a solution of a set of such finite domain constraints, taking care of not dividing by zero and not evaluating the root of a negative number:

Definition 4.2 *For a valuation $\sigma : X \to \mathbb{N}$, we define a function mapping positive polynomial expressions to integers by*

$$\begin{array}{rcl} eval(n,\sigma) & = & n \\ eval(x,\sigma) & = & \sigma(x) \\ eval(e_1+e_2,\sigma) & = & eval(e_1,\sigma)+eval(e_2,\sigma) \\ eval(e_1\times e_2,\sigma) & = & eval(e_1,\sigma)\times eval(e_2,\sigma) \end{array}$$

We say that σ is a solution of a set C of finite domain Diophantine constraints when it is a solution of each constraint in C. It is a solution of a constraint $x \in [e_1, e_2]$ if it satisfies $x \geq e_1$ and $x \leq e_2$. Valuation σ satisfies $x \geq \sqrt[n]{(f_1 - f_2)/f_3}$ when $\sigma(x)^n \times eval(f_3,\sigma) + eval(f_2,\sigma) \geq eval(f_1,\sigma)$, and it satisfies $x \leq \sqrt[n]{(f_1 - f_2)/f_3}$ when $\sigma(x)^n \times eval(f_3,\sigma) + eval(f_2,\sigma) \leq eval(f_1,\sigma)$.

We are now ready to define our translation from Diophantine constraints into finite domain constraints.

Definition 4.3 *The finite domain translation of a Diophantine constraint $P \geq 0$ is a set of n constraints, n being the number of occurrences of each variable in P, computed as follows: for each occurrence of a variable x in P, one may write without loss of generality $P = ax^kQ + R$, where Q is a power product where x does not occur and $a > 0$ (if $a < 0$, consider constraints $-P \leq 0$ or $-P = 0$ respectively, and consider other cases below), and generate*

$$x \in \left[\sqrt[k]{(R_{neg} - R_{pos})/aQ}, B\right]$$

where B is the bound of solutions to search for, and $R = R_{pos} - R_{neg}$ where R_{pos} and R_{neg} have only positive coefficients.

Similarly, the translation of $ax^kQ + R \leq 0$ is

$$x \in \left[0, \sqrt[k]{(R_{neg} - R_{pos})/aQ}\right]$$

and the translation of $ax^kQ + R = 0$ is

$$x \in \left[\sqrt[k]{(R_{neg} - R_{pos})/aQ}, \sqrt[k]{(R_{neg} - R_{pos})/aQ}\right]$$

The reason why we group together monomials with the same sign will be made clear later. The next proposition shows that our translation is sound. There is a small point here: we require the Diophantine constraints we start from to contain no “trivial” constraints, i.e. no constraint of the form $c \geq 0$ (or $\leq$ or $=$) where c is a constant. These are either trivially true and may be removed, or false and the whole set of constraints is unsatisfiable.

Proposition 4.4 *If C is a set of Diophantine constraints containing no trivial constraint, then its set of solutions in $[0, B]$ is exactly the same as the set of solutions in $[0, B]$ of its translation D into finite domain constraints.*

Proof. If σ satisfies C, since any constraint d of D comes from a translation of some constraint c in C, and from Definition 4.2, the truth of $\sigma(c)$ implies the truth of $\sigma(d)$.

```
propagate(s : store, C : constraint) : store =
  let active = C and passive = ∅ in
  while active ≠ ∅ do
      let c = choose(active) in
      active ← active − {c} ; passive ← {c} ∪ passive
      assuming c has the form x ∈ [e1, e2] :
      let m1 = try max(s.x.min, minVal(e1, s))
               catch ArithError → s.x.min    in
      let m2 = try min(s.x.max, maxVal(e2, s))
               catch ArithError → s.x. max    in
      if m1 > m2 then throw NoSolution else
      if m1 > s.x.min or m2 < s.x.max then
        (* new deduction made *)
        (s ← {s with x.min ← m1, x.max ← m2}, C)
        let C = dependOn(passive, x) in
        active ← active ∪ C ; passive ← passive − C
  end while
  return s
```

Figure 2: The propagation algorithm

Conversely, if σ is a solution of D, then for any constraint c of C, c generated at least one constraint d in D (because c is not trivial), and again, the truth of $\sigma(d)$ implies the truth of $\sigma(c)$ by Definition 4.2.□

We can go back now to the algorithm for solving finite domain constraints. We want to design an implementation of the propagation procedure that is specific to the form of constraint we have. The proposed algorithm is given Figure 2, where the auxiliary procedure *choose*(C) returns an arbitrary element of C, and *dependOn*(C, x) returns the subset of C where x occurs. The while loop always terminates because at each iteration either the store size $\sum(x.\text{max} - x.\text{min})$ decreases, or it remains unchanged and the size of the $active$ set of constraints decreases. Functions minVal and maxVal are defined by

$$
\begin{array}{rcl}
minVal(n, s) & = & maxVal(n, s) = n \\
minVal(x, s) & = & s.x.min \\
maxVal(x, s) & = & s.x.max \\
minVal(e_1 + e_2, s) & = & minVal(e_1, s) + minVal(e_2, s) \\
maxVal(e_1 + e_2, s) & = & maxVal(e_1, s) + maxVal(e_2, s) \\
minVal(e_1 \times e_2, s) & = & minVal(e_1, s) \times minVal(e_2, s) \\
maxVal(e_1 \times e_2, s) & = & maxVal(e_1, s) \times maxVal(e_2, s) \\
minVal(e_1 - e_2, s) & = & minVal(e_1, s) - maxVal(e_2, s) \\
maxVal(e_1 - e_2, s) & = & maxVal(e_1, s) - minVal(e_2, s) \\
minVal(e_1 / e_2, s) & = & \lceil minVal(e_1, s) / maxVal(e_2, s)) \rceil \\
maxVal(e_1 / e_2, s) & = & \lfloor maxVal(e_1, s) / minVal(e_2, s)) \rfloor \\
minVal(\sqrt[n]{e}, s) & = & \left\lceil \sqrt[n]{minVal(e, s)} \right\rceil
\end{array}
$$

$$maxVal(\sqrt[n]{e}, s) \quad = \quad \left\lfloor \sqrt[n]{maxVal(e, s)} \right\rfloor$$

where any division by zero or root of a negative number throws exception ArithError.

Proposition 4.5 *The propagation algorithm is correct, that is whenever σ is a solution of C included in some store s, then it is also included in store propagate(s, C).*

Proof. It suffices to show that this property is an invariant of the while loop. Assume σ is a solution of C included in a store s, that is for all x, $s.x.\min \leq \sigma(x) \leq s.x.\max$. Then by an easy structural induction, for any finite domain positive polynomial expression f we have

$$minVal(f, s) \leq eval(f, \sigma) \leq maxVal(f, s) \tag{2}$$

This is true indeed because only nonnegative expressions are involved in f, hence the minimum value of a product is the product of the minimum values of its arguments, and the same for the maximum.

Assume a new deduction is made by propagation of some constraint $x \in [e_1, e_2]$. Assume the new deduction is made on e_1, that is $minVal(e_1, s)$ is defined and greater than $s.x.$min. Assume $e_1 = \sqrt[n]{(f_1 - f_2)/f_3}$, then

$$minVal(e_1, s) = \left\lceil \sqrt[n]{\left\lceil \frac{minVal(f_1, s) - maxVal(f_2, s)}{maxVal(f_3, s)} \right\rceil} \right\rceil \tag{3}$$

where no undefined operations exists in this formula, that is $maxVal(f_3, s)$ is positive and the fraction is nonnegative. Since σ is a solution, we have

$$\sigma(x)^n \times eval(f_3, \sigma) + eval(f_2, \sigma) \geq eval(f_1, \sigma)$$

hence by (2)

$$\sigma(x)^n \times maxVal(f_3, s) + maxVal(f_2, s) \geq minVal(f_1, s)$$

So, since $maxVal(f_3, s)$ is positive

$$\sigma(x)^n \geq (minVal(f_1, s)maxVal(f_2, s))/maxVal(f_3, s)$$

that is

$$\sigma(x)^n \geq \lceil (minVal(f_1, s) - maxVal(f_2, s))/maxVal(f_3, s) \rceil$$

because $\sigma(x)^n$ is an integer. Since the right-hand side is nonnegative, and root functions are increasing

$$\sigma(x) \geq \sqrt[n]{\lceil (minVal(f_1, s)maxVal(f_2, s))/maxVal(f_3, s) \rceil}$$

and by (3), and again because $\sigma(x)$ is an integer, we get $\sigma(x) \geq minVal(e_1, s)$.

We prove similarly that $\sigma(x) \leq maxVal(e_2, s)$, and proceed similarly when the new deduction is made on e_2.□

4.4 Further optimisations

The algorithm provided in the previous section is reasonably efficient, at least much more efficient than the trivial algorithm which explores all possible valuations. However, as noticed in the example of Section 4.1, the size of the constraints to solve increases quickly with the number of rules of the TRS. Practice brings to the fore the need for more optimisations.

We first give some straightforward simplification rules, then we explore an improvement which amounts to abstracting squares and products in order to make each constraint more "atomic" and to share products as much as possible. Eventually we analyse the complexity of performing these sharings.

4.4.1 Simplifications

In Proposition 4.4, we have already seen that one should handle constraints where no variable occurs before performing any translation into finite domain constraints. In fact, more simplification rules can be applied before the translation: assume we write a polynomial $\sum c_i m_i$ where the m_i are power products and the c_i are the coefficients, we have the following simplification rules:

$$\begin{array}{lcl}
\sum c_i m_i = 0 & \Rightarrow & \text{allNull}(c_0, m_1, \ldots, m_k) \text{ if all } c_i \geq 0 \\
\sum c_i m_i \geq 0 & \Rightarrow & \textit{true} \text{ if all } c_i \geq 0 \\
\sum c_i m_i = 0 & \Rightarrow & \text{allNull}(c0, m_1, \ldots, m_k) \text{ if all } c_i \leq 0 \\
\sum c_i m_i \geq 0 & \Rightarrow & \text{allNull}(c0, m_1, \ldots, m_k) \text{ if all } c_i \leq 0
\end{array}$$

where allNull$(c_0, m1 \ldots, m_k)$ is either *false* if constant coefficient c_0 is not 0, or the set of constraints $\{m_1 = 0, \ldots, m_k = 0\}$ if $c_0 = 0$.

Proposition 4.6 *These transformations preserve the set of solutions.*

Proof. Straightforward.□

4.4.2 Abstracting squares and products of Diophantine constraints

As noticed, for finite domain constraints in general, by Codognet & Diaz [11], the efficiency of the propagation procedure can be significantly improved by making the constraints as small as possible, so as to get a small number of constraints given by *dependOn*. One way to achieve this is to introduce an operation of *abstraction*: to introduce fresh variables to denote subexpressions. For example, for solving the constraint

$$x^7 - x^4 + x^3 - 5 \geq 0$$

one may introduce $y = x^2$ to transform the constraint into $\{xy^3 - y^2 - xy - 5 \geq 0, y = x^2\}$, then furthermore $z = y^2$ and $t = xy$ to get $\{tz - z - t - 5 \geq 0, y = x^2, z = y^2, t = xy\}$. On this form, one may further note that this introduction of variables makes some sharing of common subexpressions. Such abstractions could be made on any subexpressions, but we made the choice of performing them only for abstractions of squares and products of variables, so as to share multiplications only (we discuss further this choice in Section 5).

However, these abstractions introduce a small difficulty: they do not preserve the set of solutions in a given interval $[0, B]$ because, for instance, if we introduce a variable $z = xy$, then the value of z for a given solution may be greater than B. Hence the abstractions need to be done relatively to the initial store. In practice these operations must be done after computing the initial store, in procedure *solve* of Figure 1. The transformation rules are

$$\begin{array}{lcl} S, C & \Rightarrow & S \cup \{z.\text{min} = 0, z.\text{max} = (x.\text{max})^2\}, C[x^2/z] \cup \{z = x^2\} \\ S, C & \Rightarrow & S \cup \{z.\text{min} = 0, z.\text{max} = x.\text{max} \times y.\text{max}\}, C[xy/z] \cup \{z = xy\} \end{array}$$

where z is a fresh variable and $C[e/z]$ denotes replacement of e by z in C. Replacement of x^2 by z amounts to replacing any power x^{2n} by z^n and x^{2n+1} by xz^n, and replacement of xy by z amounts to replacing any $x^{n+k}y^n$ by $x^k z^n$ and any $x^n y^{n+k}$ by $y^k z^n$.

Proposition 4.7 *These transformations preserve the set of solutions in the following sense: given a bound B and a set of constraints C, given S, D obtained by any number of abstractions starting from $(\{x.min = 0, x.max = B \mid x \in C\}, C)$, a valuation σ in $[0, B]$ is a solution of C if and only if it can be extended (giving values to fresh variables) into a solution of D in S.*

Proof. Straightforward, as soon as the maximal bounds of introduced variables are computed as the maximal bounds of the variables they abstract, as in transformation rules above.□

4.4.3 Minimisation of the number of introduced variables

For the moment we did not give any strategy for choosing which product or which square to abstract, in which order. One would like to proceed in such a way that a minimal number of extra variables is introduced. However, finding such a minimal way is equivalent to solving the famous problem of *addition chains* already mentioned in Section 1, where the length of a chain corresponds to the number of introduced variables.

The simplest case is when one wants to compute a power of a single variable with the minimum number of multiplications. A well-known efficient algorithm is the dichotomic one: $x^{2n} = (x^n)^2$, $x^{2n+1} = x \times (x^n)^2$, also called recursive scheme in [35]. Its complexity is bounded with $2\log_2(n)$, but it is not optimal: for x^{15} it requires 6 multiplications whereas it is possible to proceed with only 5. The classical complexity results on addition chains for a single integer n state that the length of the shortest chain is between $\log_2(n)$ and $2\log_2(n)$, but in general, the only way to compute it is by using a non-deterministic "branch and bound" algorithm, and no closed formula on n giving the optimal number of multiplications is known. Moreover, we are in the very general case, with several variables and several monomials, thus the problem is even more complicated.

Since no optimal deterministic algorithm was known, we decided to use a non-optimal algorithm of our own. It involves a heuristic way to decide which square or product to abstract, and never bracktracks so that the computation would be done in a short time.

Our algorithm proceeds as follows: it computes the *weight* of each possible square and product i.e., the number of multiplications that will be saved if the abstraction is performed:

- the weight of x^2 is the sum of $\lfloor \alpha/2 \rfloor$ for each occurrence of x^α;
- the weight of xy is the sum of $\min(\alpha, \beta)$ for each occurrence of $x^\alpha y^\beta$.

Then, the algorithm always chooses an abstraction of maximal weight. Note that in the case of one power of one variable, this algorithm is equivalent to the dichotomic one above.

Technically, if a product and a square have the same (maximal) weight, the square abstraction will be preferred, a property we will use in the following complexity analysis.

Our implementation involves a variant, obtained by stopping the abstraction whenever the maximal weight is not at least 2. In other words, no abstraction of a product is performed if it does not make some sharing. We will discuss this variant in Section 5.

4.4.4 Complexity analysis

Pippenger [36] gave the following estimation of the shortest chain:

$$L(p,q,n) = \min(p,q)\log_2 n + \frac{H}{\log_2 H} U\left(\sqrt{\frac{\log_2 \log_2 H}{\log_2 H}}\right) + O(\max(p,q))$$

where n is the maximal coefficient, p the number of variables, q the number of monomials, $H = pq\log_2(n+1)$, and $U(x)$ is of the form $2^{O(x)}$. Roughly speaking, this bound depends linearly in p and q, and logarithmically in n. Regarding our algorithm, we were not able to establish a non-trivial complexity bound in the general case, however we give a complexity bound for the case of one monomial, showing that our algorithm complexity is linear in p and logarithmic in n.

Theorem 4.8 *In the case of a single monomial identified with its tuple of exponents l , the complexity $C(l)$ of our algorithm is bounded by*

$$C(l) \leq \log(\max(l)) + \sum_{z \in l} \log(z) + |l| - 1$$

where $|l|$ denotes the length of l, and $\log(x)$ is the logarithm of x in base 2, rounded by floor.

Proof. By induction on l (the sum of elements for example).

If the input of the algorithm is a monomial of the form x, that is $l = (1)$, there is no abstraction to perform, hence $C(1) = 0 \leq \log 1 + \log 1 + 1 - 1$. Otherwise, some abstractions have to be done:

1. Let us assume that the first step is a square abstraction over a variable of exponent a: by reordering the variables, without loss of generality, the tuple of exponents is of the form $(a, b_2, \ldots, b_n)$. Since a square abstraction has been chosen,

   $$\forall i, i \geq 2 \Longrightarrow \left\lfloor \frac{a}{2} \right\rfloor \geq \min\{a, b_i\}$$

   hence

   $$\forall i, i \geq 2 \Longrightarrow \left\lfloor \frac{a}{2} \right\rfloor \geq b_i$$

   and in particular a is the maximum of $(a, b_2, \ldots, b_n)$.

   (a) If a is even, then $a = 2a'$ with $a' > 0$, $(a, b_2, \ldots, b_n)$ is transformed into $(a', b_2, \ldots, b_n)$ and the maximum of $(a', b_2, \ldots, b_n)$ is a':

   $$\begin{array}{lll} C(a, b_2, \ldots, b_n) = 1 + C(a', b_2, \ldots, b_n) & & \\ \quad \leq \; 1 + \log a' + \log a' + \sum_{i \geq 2} \log b_i + n - 1 & & \text{(IH)} \\ \quad = \; 1 + (\log a - 1) + (\log a - 1) + \sum_{i \geq 2} \log b_i + n - 1 & & \\ \quad \leq \; \log a + \log a + \sum_{i \geq 2} \log b_i + n - 1 & & \end{array}$$

(b) If a is odd, then $a = 2a' + 1$ with $a' > 0$, $(a, b_2, \ldots, b_n)$ is transformed into $(a', 1, b_2, \ldots, b_n)$ and the maximum of $(a', 1, b_2, \ldots, b_n)$ is a':

$$\begin{array}{rll} C(a, b_2, \ldots, b_n) = 1 + C(a', 1, b_2, \ldots, b_n) & & \\ \leq & 1 + \log a' + \log a' + \sum_{i \geq 2} \log b_i + (n+1) - 1 & \text{(IH)} \\ = & 1 + (\log a - 1) + (\log a - 1) + \sum_{i \geq 2} \log b_i + (n+1) - 1 & \\ = & \log a + \log a + \sum_{i \geq 2} \log b_i + n - 1 & \end{array}$$

2. Let us assume that the first step of the algorithm is a product abstraction over some variables of respective exponents a and b: by reordering the variables, without loss of generality, the tuple of exponents is of the form $(a, b, c_3, \ldots, c_n)$ where $a \geq b$.

   (a) If $a = b$, then $(a, b, c_3, \ldots, c_n)$ is transformed into $(b, c_3, \ldots, c_n)$. Let $m = \max(a, b, c_3, \ldots, c_n)$ and $m' = \max(b, c_3, \ldots, c_n)$:

$$\begin{array}{rll} C(a, b, c_3, \ldots, c_n) = 1 + C(b, c_3, \ldots, b_n) & & \\ \leq & 1 + \log m' + \log b + \sum_{i \geq 3} \log c_i + (n-1) - 1 & \text{(IH)} \\ \leq & \log m + \log a + \log b + \sum_{i \geq 3} \log c_i + n - 1 & \end{array}$$

   (b) If $a > b$, $(a, b, c_3, \ldots, c_n)$ is transformed into $(b, a - b, c_3, \ldots, c_n)$. Let $m = \max(a, b, c_3, \ldots, c_n)$ and $m' = \max(a - b, b, c_3, \ldots, c_n)$. Since a maximal weighted product abstraction has been chosen, this means in particular that

$$\min(a, b) = b > \left\lfloor \frac{a}{2} \right\rfloor$$

   hence $2b \geq a$, hence $2(a - b) \leq a$, hence $1 + \log(a - b) \leq \log a$, hence:

$$\begin{array}{rll} C(a, b, c_3, \ldots, c_n) = 1 + C(a - b, b, c_3, \ldots, c_n) & & \\ \leq & 1 + \log m' + \log(a - b) + \log b + \sum_{i \geq 3} \log c_i + n - 1 & \text{(IH)} \\ \leq & \log m + \log a + \log b + \sum_{i \geq 3} \log c_i + n - 1 & \end{array}$$

□

5 Implementation and experiments

Our method for automatically finding polynomial interpretations suitable for termination of a given TRS has been implemented in the C*i*ME rewrite tool. We deal with strong termination only, but we shall point out that the constraints solving part of C*i*ME has also been used by other systems like MUTERM [30, 31] for context-sensitive rewriting and CARIBOO [19].

For the search for polynomial interpretations, the user may select the class of polynomial interpretations to look for, giving both the form (linear, simple, simple-mixed, quadratic) and the bound on coefficients. Here is a sample session for proving termination of Lankford's example:

```
CiME> let F = signature ". : infix binary ; f : 1";
F : signature = <signature>
CiME> let X = vars "x y z";
X : variable_set = <variable set>
CiME> let R = TRS F X " (x.y).z -> x.(y.z) ;
 f(x).f(y) -> f(x.y) ; f(x).(f(y).z) -> f(x.y).z  ;";
R : (F,X) TRS = { (x . y) . z -> x . (y . z),
```

TRS	Dioph.		FD (0)		FD (1)		FD (2)	
	cons	var	cons	var	cons	var	cons	var
Rosu *et al.* [37]	170	45	3896	45	3409	854	2353	328
(time)				9.9s		4.7s		2.9s
Deplagne [14]	12	4	15	4	25	9	17	5
	158	19	1665	19	976	164	968	160
	1085	26	11360	26	4638	668	4422	560
	92	17	304	17	389	96	317	60
	186	41	935	41	1159	301	833	138
(time)				192.9s		9.8s		7.4s

Figure 3: Some experimental results

```
                  f(x) . f(y) -> f(x . y),
                  f(x) . (f(y) . z) -> f(x . y) . z }
CiME> polyinterpkind {("simple",2)};
CiME> termination R;
Trying to solve the following constraints:
{ (V_0 . V_1) . V_2 > V_0 . (V_1 . V_2) ;
  f(V_0) . (f(V_1) . V_2) > f(V_0 . V_1) . V_2 ;
  f(V_0) . f(V_1) > f(V_0 . V_1) }
Search parameters: simple polynomials, coefficient bound is 2.
Solution found for these constraints:
[.](X0,X1) = X1*X0 + X1 + 2*X0 + 1;  [f](X0) = X0 + 1;
Termination proof found.
```

This proof was made using the standard termination criterion, the user may nevertheless ask for various dependency pairs criteria: with or without dependency graphs, and with or without *marks* (i.e. *tuple symbols*). It is moreover possible to use an automatic decomposition of TRSs into *modules* for performing termination in several parts [41] following an incremental and modular fashion. Our implementation of dependency graphs considers the approach of Arts & Giesl [1] only, in particular because the (better) estimated graphs of Middeldorp *et al.* [23, 33] are incompatible with this incremental approach, which indeed requires to prove CE-termination.

A large collection of examples is available in the C*i*ME distribution, which demonstrate the practical power of the combination of polynomial interpretations and dependency pairs criteria. This combination is also able to prove termination of a significant part of the examples of the Termination Problem Data Base (`http://www.lsi.upc.es/~albert/tpdb.html`). We detail here some experiments with C*i*ME on two examples recently published in the literature, the termination of which was presented as a difficult task. The first one was presented in 2003 by Rosu & Viswanathan [37]: a TRS of 33 rules for regular language membership, with a termination proof found by hand. C*i*ME is able to find a proof automatically, using the standard criterion, simple-mixed polynomials, with bound 2 for coefficients. The second one was proposed in 2000 by Deplagne [14]: a TRS of 53 rules for sequent calculus modulo, without any termination proof. C*i*ME is able to find a proof automatically (indeed the only proof known to date), using the dependency graph criterion combined with the modular approach, without marks, with simple polynomials and bound 3 for coefficients (see [12] on how to select this combination with C*i*ME).

Figure 3 summarises results on these examples, in particular regarding the variable abstraction strategy. Times are obtained on a Pentium III 933MHz processor. Note that Rosu's example is also solved in less than a second with dependency pairs criterion and modular approach, with much fewer constraints; we give the results for the standard criterion because they are more informative with regards to the abstraction policy. The second column of the table gives the number of Diophantine constraints to solve and the number of variables. With the second example there are five non-trivial dependency graph components, hence five set of Diophantine constraints to solve, and the numbers are given for each of them. The remaining columns gives the number of finite domain constraints generated, as well as the number of variables, with reference to three different ways of conducting abstractions. For Column FD(0), no square or product abstraction is made before the translation. For Column FD(1), all squares and products are abstracted. Finally for For Column FD(2), only squares and products occurring at least twice are abstracted.

These results lead to the following conclusions. First of all, one should notice that our method allows to solve thousands of constraints, over hundreds of variables, in few seconds. Concerning the abstraction strategy, policy FD(2) leads obviously to the best results, the numbers of the fifth column are always better indeed than the ones of the third and fourth. The conclusion is clearly that abstraction of squares and products is useful, but only for those that occur several times: this shows how important in practice is the idea of sharing behind these abstractions. In Section 4.2, we mentioned that one may also consider the possibility of abstracting additions, and not only products. We indeed made some experiments, but they were not very successful: the number of generated variables is already quite large with product abstraction, and adding some more seems to costly. However, this may be also because we were not able to find a convenient strategy for those abstractions: performing those efficiently is again a problem as difficult as addition chain problems.

6 Conclusion and future work

By combining several criteria and transformations (testing positiveness using absolute positiveness, using μ-translation, solving Diophantine constraints by translation into finite domain constraints, sharing squares and products in a smart way) we obtained an efficient method for proving termination using polynomial interpretations. A work which is still to be done is a complete analysis of theoretical complexity of our abstraction algorithm, or even finding a better one.

An interesting extension of this work would be the use of exponential interpretations, as proposed by Lescanne [29]. However, there is a major problem: the criterion proposed by Lescanne to ensure positiveness of exponential function is quite ad-hoc, which makes it hard to transform into a method for automatic search for such interpretations. In other words, an extension to Hong & Jakuš's work [25] to exponential functions is a required first step. One may use afterwards finite domain contraints solving techniques to solve the underlying constraints, certainly without major difficulty.

Another interesting extension would be to ordinal interpretations [10, 15], which are probably not as popular as they should be.

References

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

[2] T. Arts, J. Giesl, and E. Ohlebusch. Modular termination proofs for rewriting using dependency pairs. *Journal of Symbolic Computation*, 34(2):21–58, 2002.

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

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

[5] F. Bergeron, J. Berstel, and S. Brlek. Efficient computation of addition chains. *Journal de théorie des nombres de Bordeaux*, 6:21–38, 1994.

[6] F. Bergeron, J. Berstel, S. Brlek, and C. Duboc. Addition chains using continued fractions. *Journal of Algorithms*, 10(3):403–412, 1989.

[7] D. Bleichenbacher and A. Flammenkamp. An efficient algorithm for computing shortest addition chains. `http://wwwhomes.uni-bielefeld.de/achim/addition_chain.html`, 2003.

[8] J. Bos and M. Coster. Addition chain heuristics. In *Advances in Cryptology – Proc. Crypto '89*, volume 435 of *Lecture Notes in Computer Science*, pages 400–407. Springer-Verlag, 1989.

[9] E. Cichon. Bounds on derivation lenghts from termination proofs. Research Report CSD-TR-622, Department of Computer Science, Royal Holloway and Bedford New College, University of London, June 1990.

[10] E. A. Cichon and H. Touzet. An ordinal calculus for proving termination in term rewriting. In H. Kirchner, editor, *Proceedings 21st International Colloquium on Trees in Algebra and Programming*, volume 1059 of *Lecture Notes in Computer Science*, pages 226–240, Linköping (Sweden), Apr. 1996. Springer-Verlag.

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

[12] E. Contejean, C. Marché, B. Monate, and X. Urbain. Proving Termination of Rewriting with CiME. In A. Rubio, editor, *Extended Abstracts of the 6th International Workshop on Termination, WST'03*, pages 71–73, June 2003. `http://cime.lri.fr/`.

[13] P. de Rooij. Efficient exponentiation using precomputation and vector addition chains. In A. D. Santis, editor, *Advances in Cryptology – EUROCRYPT '94*, volume 950 of *Lecture Notes in Computer Science*, pages 9–12. Springer-Verlag, 1994.

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

[15] N. Dershowitz. Trees, ordinals, and termination. In M. C. Gaudel and J.-P. Jouannaud, editors, *4th International Joint Conference on Theory and Practice of Software Development*, volume 668 of *Lecture Notes in Computer Science*, pages 243–250, Orsay, France, Apr. 1993. Springer-Verlag.

[16] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 243–320. North-Holland, 1990.

[17] D. Dobkin and R. J. Lipton. Addition chain methods for the evaluation of specific polynomials. *SIAM Journal on Computing*, 9(1):121–125, 1980.

[18] P. J. Downey, B. L. Leong, and R. Sethi. Computing sequences with addition chains. *SIAM Journal on Computing*, 10(3):638–646, 1981.

[19] O. Fissore, I. Gnaedig, and H. Kirchner. CARIBOO : An Induction Based Proof Tool for Termination with Strategies. In *Proceedings of the Fourth International Conference on Principles and Practice of Declarative Programming (PPDP)*, Pittsburgh, USA, October 2002. ACM Press. `http://www.loria.fr/~fissore/CARIBOO`.

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

[21] J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke. Aprove: A system for proving termination. In A. Rubio, editor, *Extended Abstracts of the 6th International Workshop on Termination, WST'03*, June 2003. `http://www-i2.informatik.rwth-aachen.de/AProVE`.

[22] B. Gramlich and S. Lucas. Simple termination of context-sensitive rewriting. In B. Fischer and E. Visser, editors, *Proc. of 3rd ACM Sigplan Workshop on Rule-Based Programming, RULE'02*, pages 29–41, Pittsburgh, USA, Oct. 2002. ACM Press.

[23] N. Hirokawa and A. Middeldorp. Approximating Dependency Graphs without using Tree Automata Techniques. In A. Rubio, editor, *Extended Abstracts of the 6th International Workshop on Termination, WST'03*, pages 8–10, June 2003. Technical Report DSIC II/15/03, Universidad Politécnica de Valencia, Spain.

[24] N. Hirokawa and A. Middeldorp. Tsukuba termination tool. In R. Nieuwenhuis, editor, *14th International Conference on Rewriting Techniques and Applications*, volume 2706 of *Lecture Notes in Computer Science*, pages 311–320, Valencia, Spain, June 2003. Springer-Verlag.

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

[26] 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.

[27] 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.

[28] D. S. Lankford. On proving term rewriting systems are Noetherian. Technical Report MTP-3, Mathematics Department, Louisiana Tech. Univ., 1979. Available at `http://www.ens-lyon.fr/~plescann/not_accessible.html`.

[29] P. Lescanne. Termination of rewrite systems by elementary interpretations. In H. Kirchner and G. Levi, editors, *3th International Conference on Algebraic and Logic Programming*, volume 632 of *Lecture Notes in Computer Science*, pages 21–36, Volterra, Italy, Sept. 1992. Springer-Verlag.

[30] S. Lucas. Termination of (canonical) context-sensitive rewriting. In S. Tison, editor, *13th International Conference on Rewriting Techniques and Applications*, volume 2378 of *Lecture Notes in Computer Science*, pages 296–310, Copenhagen, Denmark, July 2002. Springer-Verlag.

[31] S. Lucas. Mu-term, a tool for proving termination of rewriting with replacement restrictions, 2003. Available at `http://www.dsic.upv.es/~slucas/csr/termination/muterm/`.

[32] J. V. Matijasevic. Enumerable sets are diophantine. *Soviet Mathematics (Dokladi)*, 11(2):354–357, 1970.

[33] A. Middeldorp. Approximating dependency graphs using tree automata techniques. In R. Goré, A. Leitsch, and T. Nipkow, editors, *International Joint Conference on Automated Reasoning*, volume 2083 of *Lecture Notes in Artificial Intelligence*, pages 593–610, Siena, Italy, june 2001. Springer-Verlag.

[34] J. Olivos. On vectorial addition chains. *Journal of Algorithms*, 2(1):13–21, Mar. 1981.

[35] C. Papadimitriou and D. Knuth. Duality in Addition Chains. *Bulletin of the EACTS*, 13:2–3, 1981.

[36] N. Pippenger. On the evaluation of powers and monomials. *SIAM Journal of Computing*, 9(2):230–250, 1980.

[37] G. Rosu and M. Viswanathan. Testing extended regular language membership incrementally by rewriting. In R. Nieuwenhuis, editor, *14th International Conference on Rewriting Techniques and Applications*, volume 2706 of *Lecture Notes in Computer Science*, Valencia, Spain, June 2003. Springer-Verlag.

[38] J. Sauerbrey and A. Dietel. Resource requirements for the application of addition chains in modulo exponentiation. In R. Rueppel, editor, *Advances in Cryptology - EUROCRYPT '92*, volume 658 of *Lecture Notes in Computer Science*, pages 174–182. Springer-Verlag, 1993.

[39] A. Schonhage. A lower bound for the length of addition chains. *Theoretical Computer Science*, 1(1):1–12, June 1975.

[40] 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.

[41] X. Urbain. Modular and incremental automated termination proofs. *Journal of Automated Reasoning*, To appear.

[42] H. Zantema. Minimizing sums of addition chains. *Journal of Algorithms*, 12:281–307, 1991.

Contents