Journal of Automated Reasoning (2005) 34: 325–363
DOI: 10.1007/s10817-005-9022-x

Mechanically Proving Termination Using Polynomial Interpretations*

EVELYNE CONTEJEAN[1], CLAUDE MARCHÉ[1], ANA PAULA TOMÁS[2] and XAVIER URBAIN[3]
[1]*PCRI-LRI (CNRS UMR 8623) & INRIA Futurs Bât. 490, Université Paris-Sud, Centre d'Orsay, 91405 Orsay, Cedex, France. e-mail: {contejea, marche}@lri.fr*
[2]*DCC-FC & LIACC, University of Porto, R. do Campo Alegre 823, 4150-180 Porto, Portugal. e-mail: apt@ncc.up.pt*
[3]*CEDRIC, IIE, Conservatoire National des Arts et Métiers, 18 allée Jean Rostand, 91025 Evry, Cedex, France.*

Abstract. For a long time, term orderings defined by polynomial interpretations were scarcely used in 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 regard 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 some experimental results that show how useful polynomial orderings actually are in practice.

Key words: term rewriting, termination, polynomial interpretations.

1. Introduction

For decades, the use of the standard Manna–Ness criterion [38] (that is each rule decreases w.r.t. a well-founded ordering) dominated among the different known methods aimed at proving termination of term rewriting systems. The orderings required by this criterion must have strong properties, such as strict monotonicity; they are usually distinguished in two classes: *syntactical* orderings and *semantical* orderings. Syntactical orderings rely on a precedence on symbols that is extended to terms, while semantical ones make use of an interpretation of terms. Among the latter, term orderings defined by polynomial interpretations have been defined in 1979 in a pioneer paper of Lankford [35].

* 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".

There are several reasons that made polynomial interpretations less popular than syntactical methods like RPO [16]. From a theoretical point of view, the combination of polynomials with the Manna–Ness criterion puts strong restrictions on the class of relations the obtained ordering can contain: first, the length of derivations has a double exponential bound [28]; second, the computed function necessarily belongs to a restricted complexity class [7]. For instance, the termination of the famous Ackermann–Peter function can easily be proven by using RPO, while it is impossible to obtain a suitable polynomial interpretation. From a practical point of view, precedence-based orderings are easier to implement, automatic search is decidable, whereas the search for suitable polynomials is necessarily incomplete.

But recently, considerable progress was achieved on automated termination proof, in particular by the use of the dependency pairs method and its termination criteria [1], its applications to incremental/hierarchical termination proofs [23, 56], and to termination under specific strategies such as innermost termination [1, 23] or context-sensitive rewriting [25].

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 in part similar to Knuth–Bendix orderings [24, 27], but not to Recursive Path orderings. In fact, the latter are always strictly monotonic; hence transformations have been proposed such as *argument filtering* [1] or more generally *recursive program scheme* [13, 31]. Thus, some additional transformation steps have to take place during a proof discovery process using RPO-like orderings. However, to add 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 in polynomial interpretations-based orderings led us to design a new implementation of them inside the C*i*ME rewrite tool [11]. We describe hereafter the theoretical basis of this implementation, which is able to find polynomial orderings for termination proofs with *full* and *efficient* automation. New improvements include a technique of translation (Section 3.4) and an original method for solving non-linear Diophantine constraints (Section 4.2).

A key issue in finding polynomial orderings is to solve some nonlinear constraints over natural numbers. In order to improve efficiency, these constraints are linearized thanks to the introduction of abstraction variables, and the problem of minimising the number of such variables arises. Quite surprising, this problem is a generalization of the well-known problem of computation of *addition chains* [4–6, 8, 14, 19, 20, 44–46, 50, 51, 57], which arises naturally

when one wants to compute a polynomial expression while minimizing the number of multiplications.

This paper is organised as follows. In Section 2, we first discuss the currently known termination criteria that are suitable for automation, and 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. In Section 5 we present 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 [2, 18, 53] for details and to Arts & Giesl [1, 23] regarding dependency pairs.

As it is now suitable for the dependency pairs approach where both strict and nonstrict comparisons of terms occur, we need both strict orderings and quasi-orderings.

Formally, a *term ordering* is a pair $(\succeq, >)$ 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, that is, reflexive and transitive; (2) $>$ is a strict ordering, i.e., irreflexive and transitive; and (3) $> \cdot \succeq = >$ or $\succeq \cdot > = >$.

A term ordering is said to be *well-founded* if there is no infinite strictly decreasing sequence $t_1 > t_2 > \cdots$ and *stable* if both $>$ and $\succeq$ are stable under substitutions, that is for any terms t_1 and t_2 and for any substitution σ, if $t_1 > t_2$, then $t_1\sigma > t_2\sigma$, 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 $t, u, v_1, \ldots, v_{i-1}, v_{i+1}, \ldots, v_n$, $t\,\mathcal{R}\,u$ implies

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

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

We shall point out that our notion of weak reduction ordering is a particular case of the very general notion of *weak reduction pair* of Kusakari & al. [33], which requires (1') $\succeq$ being any monotonic and stable relation (but not necessarily reflexive nor transitive), (2') $>$ being well-founded and stable, and (3') $>\cdot\succeq \subseteq >$ or $\succeq\cdot> \subseteq >$. However it is easy to see that if $\succeq$ is reflexive, (3') implies (3); in other words, any weak reduction pair made of orderings is a weak reduction ordering in our setting.

In order to prove termination of a given TRS R, several possible criteria exist. The simplest one is the *standard* Manna-Ness criterion [39]: if there exists a *strict* reduction ordering $(\succeq, >)$ such that $l > r$ 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 $(\succeq, >)$ such that

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

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, 42].

Regarding innermost termination, there are improvements of dependency pairs criteria [1] where the underlying ordering is no longer 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, $s' > t'$ whenever $s > t$, $s =_E s'$ 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. For all material regarding polynomials we use Lang's notations and refer to his book [34]. 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 summarize 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 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

$$t_1 \succeq_\phi t_2 \text{ iff } \phi(t_1) \geq_D \phi(t_2)$$
$$t_1 >_\phi t_2 \text{ iff } \phi(t_1) >_D \phi(t_2)\,.$$

LEMMA 3.2. *$(\succeq_\phi, >_\phi)$ is a term ordering on ground terms. It is well-founded if $>_D$ is well-founded.*

Proof. From $>_D = \geq_D - \leq_D$, it is easy to get $>_D \cdot \geq_D = \geq_D \cdot >_D = >_D$, and from that it is easy to get that $(\succeq_\phi, >_\phi)$ is a term ordering. If $(\succeq_\phi, >_\phi)$ was not well-founded, there would be an infinite decreasing sequence $t_1 >_\phi t_2 >_\phi t_3 >_\phi \cdots$ 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 generalize this construction to nonground 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 that leads to an almost equivalent definition.

The idea is the following: we should not interpret a nonground 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

$$f \succeq_{D,X} g \text{ iff for all } \rho \in X \to D, f(\rho) \geq_D g(\rho)$$
$$f >_{D,X} g \text{ iff for all } \rho \in X \to D, f(\rho) >_D g(\rho)\,.$$

We point out that $>_{D,X}$ is *not* $\succeq_{D,X} - \preceq_{D,X}$, and in some sense, that is why term orderings as ordering pairs are needed. 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 $(x, a) \notin >_{D,X}$.

Now, automation of such an ordering relies only 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 that 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

$$t_1 \succeq_\phi t_2 \text{ iff } \phi(t_1) \succeq_{D,X} \phi(t_2)$$
$$t_1 >_\phi t_2 \text{ iff } \phi(t_1) >_{D,X} \phi(t_2)\,.$$

LEMMA 3.4. *$(\succeq_\phi, >_\phi)$ is a term ordering on nonground terms. It is well-founded if $>_D$ is well-founded.*

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 ρ of variables, we would have an infinite decreasing sequence $f_1(\rho) >_D f_2(\rho) >_D f_3(\rho) >_D \ldots$ of elements of D, leading to a contradiction. □

EXAMPLE 3.5. 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 $\rho(x)$ and $\rho(y)$ for x and y:

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

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, $f(f(a,x),y) \not\succeq_\phi f(x,a)$ because when $\rho(x) = \rho(y) = 0$, both terms are interpreted as 4.

DEFINITION 3.6. 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 $\rho \in X \to D$,

$$\phi(f(t_1, \ldots, t_n))(\rho) = [\![f]\!]_\phi(\phi(t_1)(\rho), \ldots, \phi(t_n)(\rho))$$
$$\phi(x)(\rho) = \rho(x).$$

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

LEMMA 3.7. *Let ϕ be any homomorphic interpretation. For any substitution σ and any valuation ρ,* let us denote $\phi(\sigma, \rho)$ *the valuation mapping any variable x to $\phi(x\sigma)(\rho)$.* Then for any term t, $\phi(t\sigma)(\rho) = \phi(t)\phi(\sigma,\rho)$.

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

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

and if t is a variable x, $\phi(x)\phi(\sigma,\rho) = \phi(\sigma,\rho)(x) = \phi(x\sigma)(\rho)$. □

LEMMA 3.8. *If ϕ is an homomorphic interpretation, then $(\succeq_\phi, >_\phi)$ is stable.*

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

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

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

$$\begin{aligned}\phi(t_1\sigma)(\rho) &= \phi(t_1)\phi(\sigma,\rho)\\ &>_D \phi(t_2)\phi(\sigma,\rho) \qquad \text{by (1)}\\ &= \phi(t_2\sigma)(\rho).\end{aligned}$$

Hence $t_1\sigma >_\phi t_2\sigma$. the same proof holds for $\succeq_\phi$. □

LEMMA 3.9. *For any symbol f of arity n, and $1 \le i \le n$, 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 monotonic (resp. strictly) nondecreasing in x, then $\succeq_\phi$ (resp. $>_\phi$) is monotonic with reference to i-th argument of f.*

Proof. Straightforward. □

EXAMPLE 3.10. (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 $f(a, a) >_\phi a$ (since $\phi(f(a, a)) = 1 > 0 = \phi(a)$), but $f(f(a, a), a) \not>_\phi f(a, a)$ (since $\phi(f(f(a, a), a)) = 1 \times 0 + 1 = 1 = \phi(f(a, a))$).

3.2. INTERPRETATIONS OVER INTEGERS

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 3.11. For a given $\mu \in \mathbb{Z}$, let $D_\mu = \{x \in \mathbb{Z} \mid x \ge \mu\}$. 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 ϕ defined by functions $[\![f]\!]_\phi$, $f \in \mathcal{F}$, is called a *polynomial* interpretation if for all f, $[\![f]\!]_\phi$ 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, 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 $t_1 >_\phi t_2$.

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

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

We are now ready to transform each of these properties into a positiveness-property.

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

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

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

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

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.

Regarding Item (4), it is sufficient to check that for each equation $t \simeq u$ of theory E, we have $t \succeq_\phi u$ and $u \succeq_\phi t$, that is $\phi(t) - \phi(u) = 0$. For the case of AC, Ben Cherifa and Lescanne [3] 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 3.12. Here is a rewrite system for an endomorphism on a monoid [3].

$$\begin{aligned} (x \times y) \times z &\to x \times (y \times z) \\ f(x) \times f(y) &\to f(x \times y) \\ f(x) \times (f(y) \times z) &\to f(x \times y) \times z. \end{aligned}$$

In order to prove termination of this system using the standard Manna–Ness criterion, the following interpretation is proposed [3], with $\mu = 1$:

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

Let us check all needed conditions:

1. First, 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. Second, to check that $[[f]]$ strictly increases in its argument, we check that

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

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

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

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

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

is nonnegative when $x, y \geq 1$.
3. And third, 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

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

hence,

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

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

$$[\![f(x) \times f(y)]\!] = (2x)(2y) + 2x$$
$$[\![f(x \times y)]\!] = 2(xy + x).$$

Hence

$$[\![f(x) \times f(y)]\!] - [\![f(x \times y)]\!] - 1 = [4xy + 2x] - [2(xy + x)] - 1$$
$$= 2xy - 1$$

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

$$[\![f(x) \times (f(y) \times z)]\!] = 2x(2yz + 2y) + 2x$$
$$[\![f(x \times y) \times z]\!] = 2(xy + x)z + 2(xy + x).$$

Hence

$$[\![f(x) \times (f(y) \times z)]\!] - f(x \times y) \times z - 1$$
$$= [2x(2yz + 2y) + 2x] - [2(xy + x)z + 2(xy + x)] - 1$$
$$= 4xyz + 4xy + 2x - 2xyz - 2xz - 2xy - 2x - 1$$
$$= 2xyz + 2xy - 2xz - 1$$

which, after proper factorization $2xz(y - 1) + (2xy - 1)$, 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 factorization 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 remark first that this problem is undecidable in general since Hilbert's Tenth Problem can be reduced to it [40, 41].

Testing positiveness, in the context of automated termination proof, has been studied by several authors [3, 22, 36, 48, 49, 52]. 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 [54] 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 [29] made a comparison between some of these methods while proposing a new one: the absolute positiveness method. They proved it to be strictly more powerful than methods of Ben Cherifa & Lescanne [3] and Steinbach [52], and to be equivalent to Giesl's method [22] (which computes successive derivatives).

The methods of Rouyer [48, 49] and Lescanne [36] are not comparable with the absolute positiveness method. On a few examples, they succeed in proving positiveness of nonabsolutely 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 3.13. 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 [29]) is the following.

LEMMA 3.14. *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 Giesl's method [22].

EXAMPLE 3.15. Let us consider the last rule in Lankford's example, Section 3.12. We have to check whether

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

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

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

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 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 3.16. *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 nonground terms, we have to take care of valuations of variables: indeed, there is a one-to-one correspondence T between valuations in D_0 and valuations in D_μ, defined by $T(I)(x) = I(x) + \mu$, and we have for any nonground term t, by structural induction:

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

Hence we have, for any nonground 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. □

The consequence is double. First, 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 more efficient to compute the μ-translation of interpretations *once and for all*: later on, each time we will want to check $t_1 \succeq_\phi t_2$, we will compute $\phi_0(t_1) - \phi_0(t_2)$ and check positivity of its coefficients, thus avoiding any additional and expensive computation of μ-translations. So all further computations will be done with these new interpretations, and checking positiveness will be done simply by examining positiveness of coefficients only.

EXAMPLE 3.17. Again with Lankford's example, we can compute the translations of interpretations of f and $\times$. We define the new interpretations as

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

Second, it shows that if we want to *search for* a polynomial interpretation automatically, fixing $\mu = 0$ is enough. This important fact will be used in the next section. There is a potential drawback though, coefficients of μ-translated polynomials could be larger, but in general we can hope that there are other solutions with smaller coefficients: for the example above, an automatic search indeed found that $[\![f]\!](x) = x + 1$ is suitable too.

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

LEMMA 3.18. *A polynomial* 0*-interpretation* $P(x_1, \ldots, x_n)$ *with nonnegative coefficients is always nondecreasing 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* $k > 0$.

Proof. If $P(x_1, \ldots, x_n) = ax_i^k + Q(x_1, \ldots, x_n)$ with $a > 0$, $k > 0$ and Q has nonnegative coefficients, then it is clearly increasing in x_i. Conversely, if $P(x_1, \ldots, x_n)$ is increasing in x_i, then $P(0, \ldots, 0, x_i, 0, \ldots, 0)$ is also increasing in x_i, so there must be a monomial in only x_i with a positive coefficient. □

EXAMPLE 3.19. With Lankford's example, the new interpretations given for $\mu = 0$ generate a strictly monotonic ordering, since the coefficient of x in $[\![f]\!]_{\phi_0}$ is 2 and the 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 [52]. Such a bound being fixed, we show that searching for interpretations reduces to solving Diophantine constraints. However, this problem is still undecidable [40, 41].

Thus, in Section 4.2, we discuss 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 tailored to solve Diophantine constraints. Then, in Section 4.4, we address practical problems arising in implementation, that is, the excessively 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 [52]. 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:

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

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.

First, 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 4.1. Back to Lankford's example, if we want to automatically find suitable polynomial interpretations, we may try parametric simple interpretations

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

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

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

The last three, after normalization, become

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

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

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

4.2. SOLVING DIOPHANTINE CONSTRAINTS: GENERAL IDEA

The general idea is, first, 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 [10, 30]. The usual way of solving such constraints is a generalization of the well-known Davis–Putnam procedure for deciding satisfiability of propositional formulas, which are formulas 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 that, given a store and a constraint, performs some logical deductions to produce a smaller store. The second part is a nondeterministic branching which explores all possible values of variables, with various heuristics.

When specialized to solving constraints in an integer domain $[0, B]$, it is handy to have a store which memorizes only the minimum and the maximum values of variables, leading to the main algorithm given in 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 $x.\min < x.\max$.

```
solve(C : constraint, B : integer) =
  let S = {x.min ← 0, x.max ← B | x ∈ Var(C)} in
  branch(S, C)

branch(S : store, C : constraint) =
  let S = propagate(S, C) in
  if for each x in S, x.min = x.max then
    if isSolution(S, C) then  output "solution found : " s
    else throw NoSolution
  else
  let x = chooseVar(S) in
  try
  let S1 = {S with x.max ← S.x.min} in branch(S1, C)
  catch NoSolution →
  let S2 = {S with x.min ← S.x.min + 1} in branch(S2, C)
```

Figure 1. Main algorithm for solving Diophantine constrains.

Finally, *isSolution* is a procedure that, 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 in 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. In constraint programming in general, and for finite domain constraints in particular, the 'first-fail principle' is known to be good: the idea is to try to fail as quickly as possible, in order to cut branches. In C*i*ME, we implemented 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 is chosen, and again among all possible remaining variables, the variable which occurs most often in the constraints is chosen. This means that when setting the chosen variable to its minimum value, we add as much new constraints as possible, in particular we first try to set variables to 0 as much as possible. We have experimented variants of this heuristic, but this one appears to be the best. We will not discuss further the possible implementation of *chooseVar*.

EXAMPLE 4.2. 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 $xy = 15$, $x \geq \lceil \frac{15}{y} \rceil \geq \lceil \frac{15}{12} \rceil = 2$ (we denote $\lceil a \rceil$ the smallest integer 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 $xy = 15$, 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 4.3. The constraints $\{2x + y \leq 12, xy = 15\}$ will be transformed into finite domain constraints:

$$x \in [0, (12 - y)/2] \quad x \in [15/y, 15/y]$$
$$y \in [0, 12 - 2 \times x] \quad y \in [15/x, 15/x]$$

This approach is quite well-known indeed for *linear* Diophantine constraints [10] which is a major case in applications of constraints logic programming. However, the case of *nonlinear* Diophantine constraints seems not to have been studied. So we designed a specialized 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 4.4. 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

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

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

DEFINITION 4.5. For a valuation $\sigma : X \to \mathbb{N}$, we define a function mapping positive polynomial expressions to integers by

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

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 $x \leq e_2$. A 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 4.6. The finite domain translation of a Diophantine constraint $P \geq 0$, $P \leq 0$ or $P = 0$, is a set of n constraints, n being the number of occurrences of each variable in P. Each of these constraints are computed as follows: for each occurrence of a variable x in a monomial M of P, say $M = \pm a x^k Q$ where $a > 0$ and x does not occur in Q, we rewrite the constraint into one of the three forms

$$ax^k Q + R \geq 0 \tag{2}$$

$$ax^k Q + R \leq 0 \tag{3}$$

$$ax^k Q + R = 0 \tag{4}$$

A constraint of form equation (2) is translated into

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

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. A constraint of form (3) is translated into

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

A constraint of form (4) is translated into

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

The next proposition shows that our translation is sound. We point out that we require the Diophantine constraints we start from to contain no 'trivial' constraint, that is, no constraint of the form $c \geq 0$ (or $\leq$ or $=$), where c is a constant. Such constraints are either trivially true and may be removed, or false and the whole set is unsatisfiable.

PROPOSITION 4.7. *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 so-lutions 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 4.5, the truth of $\sigma(c)$ implies the truth of $\sigma(d)$. 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 4.5. □

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 in 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 $\sum(x.\max - x.\min)$ decreases, or it remains unchanged and the size of the *active* set of constraints decreases. Functions *minVal* and *maxVal* are defined by

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

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

PROPOSITION 4.8. *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

$$minVal(f, s) \leq eval(f, \sigma) \leq maxVal(f, s) \tag{5}$$

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.

```
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 ;
       let C = dependOn(passive, x) in
       active ← active ∪ C ; passive ← passive − C
  end while
  return s
```

Figure 2. The propagation algorithm.

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 $s.x.\text{min}$. Assume $e_1 = \sqrt[n]{(f_1 - f_2)/f_3}$. Then

$$minVal(e_1, s) = \left\lceil \sqrt[n]{\left\lceil \frac{minVal(f_1, s) - maxVal(f_2, s)}{maxVal(f_3, s)} \right\rceil} \right\rceil \tag{6}$$

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 equation (5)

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

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 equation (6), and again because $\sigma(x)$ is an integer, we get $\sigma(x) \geq minVal(e_1, s)$.

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

4.4. FURTHER OPTIMIZATIONS

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 be solved increases quickly with the number of rules of the TRS. Hence, dealing with actual systems brings to the fore the need for more optimizations.

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

4.4.1. *Simplifications*

In Proposition 4.7, 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 primitive monomials and the c_i are the coefficients, we have the following simplification rules:

$$\sum c_i m_i = 0 \Rightarrow \text{allNull}(c_0, m_1, \ldots, m_k) \text{ if all } c_i \geq 0$$

$$\sum c_i m_i \geq 0 \Rightarrow \mathit{true} \text{ if all } c_i \geq 0$$

$$\sum c_i m_i = 0 \Rightarrow \text{allNull}(c0, m_1, \ldots, m_k) \text{ if all } c_i \leq 0$$

$$\sum c_i m_i \geq 0 \Rightarrow \text{allNull}(c0, m_1, \ldots, m_k) \text{ if all } c_i \leq 0$$

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

PROPOSITION 4.9. *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 [10], 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 called *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 $[0, B]$. For instance, $\{xy = 4, x + y = 4\}$ has a solution in $[0, 2]$ whereas $\{z = 4, z = xy, x + y = 4\}$ has not. Hence, when performing abstractions, a maximal bound has to be computed for each variable (after computing the initial store, in procedure *solve* of Figure 1). The transformation rules are

Square abstraction

$$S, C \Rightarrow S \cup \{z.\min = 0, z.\max = (x.\max)^2\}, C[x^2/z] \cup \{z = x^2\}$$

Product abstraction

$$S, C \Rightarrow S \cup \{z.\min = 0, z.\max = x.\max \times y.\max\}, C[xy/z] \cup \{z = xy\}$$

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 4.10. *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 there is a solution σ' of D in S such that $\sigma'(x) = \sigma(x)$ for each variable x of C.*

Proof. The if part is trivial. For the only if part we proceed by induction on the number of abstraction performed. If none this is trivial since DC. If the proposition is true for S, D, and we perform an additional abstraction, say

$$S, D \Rightarrow S \cup \{z.\min = 0, z.\max = x.\max \times y.\max\}, D[xy/z] \cup \{z = xy\}$$

(the proof is similar with a square abstraction). If σ is a solution of C, then by induction there is a σ' solution of D such that $\sigma'(x) = \sigma(x)$. We pose $\sigma''(z) = \sigma'(x) \times \sigma'(y)$ and $\sigma''(v) = \sigma'(v)$ for $v \neq z$. Then σ'' is clearly a solution of $D[xy/z] \cup \{z = xy\}$ in $S \cup \{z.\min = 0, z.\max = x.\max \times y.\max\}$. □

4.4.3. *Minimisation of the number of introduced variables*

Until now, we have not discussed strategies for choosing which product or square to abstract, and in which order. Ideally, 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* [32], 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 [45]. Its complexity is bounded with $2\log(n)$,★ but it is not optimal: for x^{15} it requires six 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 $\log(n)$ and $2\log(n)$, but in general, the only way to compute it is by using a nondeterministic '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 nonoptimal algorithm of our own. It involves a heuristic way to decide which square or product to abstract, and never backtracks 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 $x^\alpha y^\beta$.

The complete algorithm, displayed in Figure 3, consists in applying square abstraction or product abstraction iteratively, always choosing an abstraction of maximal weight. If a product and a square have the same (maximal) weight, the

★ $\log(n)$ denotes the logarithm of n in base 2, rounded by floor.

```
abstraction(C : constraint, B : integer) =
  let S = {x.min ← 0, x.max ← B | x ∈ Var(C)} in
  while there are squares or products to abstract do
    look for a square or product of maximal weight:
    if it is a square x^2 then
      let z be a fresh variable in
      S ← S ∪ {z.min ← 0, z.max ← (x.max)^2}
      C ← C[x^2/z] ∪ {z = x^2}
    else
      it is a product xy,
      let z be a fresh variable in
      S ← S ∪ {z.min ← 0, z.max ← x.max × y.max}
      C ← C[xy/z] ∪ {z = xy}
  end while
  return S, C
```

Figure 3. The complete algorithm for square and product abstraction.

square abstraction will be preferred (a property we will use in the following complexity analysis). Note that in the case of one power of one variable, this algorithm is equivalent to the dichotomic one above. Technically, our implementation involves a variant, obtained by stopping the abstraction whenever the maximal weight is not at least two. 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 [46] gave the following estimation of the length of the shortest chain:

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

where n is the maximal coefficient, p the number of variables, q the number of monomials, $H = pq\log(n+1)$, 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 have been able to establish a bound that is also logarithmic in n and linear in the number of nonzero coefficients.

To study this complexity, we care neither about the original polynomials nor about, the coefficients of monomials; we just have to assume we have a set of power products (i.e., monomial with coefficient 1). Then if we have q such monomials over p variables, we build a matrix $q \times p$ of exponents, each row corresponding to a monomial, and each column to a variable. Such a matrix uniquely determines the set of monomials, up to permutation of variables names, but their order is not significant for the complexity of the algorithm.

During the proof of the next theorem, we express the algorithm in terms of matrix transformation.

THEOREM 4.11. *On input of several monomials identified with their matrix M, the complexity $\mathcal{C}(M)$ of our algorithm is bounded by*

$$\mathcal{C}(M) \leq \sum_{1\leq j\leq p} C(M^j) - q$$

where M^j is the jth column of M and

$$C(A) = \sum_{a\in A, 1\leq a} \log(a) + \left(\sum_{a\in A, 1\leq a} 1\right)(1 + \log(\max_{a\in A, 1\leq a} a))$$

when A is a column.

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

Let us first notice that the number of rows is invariant during the process of abstraction, and if A is a 0/1 vector, $C(A)$ is equal to the number of nonzero components in A, and if $A \leq B$ component by component, then $C(A) \leq C(B)$.

If the input of the algorithm is a set of q monomials of the form x_{j_i}, $1 \leq i \leq q, 1 \leq j_i \leq p$, that is, M is a 0/1 matrix, which is exactly one 1 per row, there is no abstraction to perform, hence $\mathcal{C}(M) = 0$.

$$\sum_{1\leq j\leq p} C(M^j) - q = q - q \geq 0 = \mathcal{C}(M)$$

Otherwise, some abstractions have to be done.

1. Let us assume that the first step is a square abstraction over a variable of column of exponents A: by reordering the variables, without loss of generality, the matrix of exponents is of the form $[A; M]$. After the square abstraction, it is of the form

$$M' = \left[\left\lfloor \frac{A}{2} \right\rfloor; A \bmod 2; M\right]$$

and we have

$$C\left(\left\lfloor \frac{A}{2} \right\rfloor\right) = \sum_{a\in A, 2\leq a} (\log(a) - 1) + \left(\sum_{a\in A, 2\leq a} 1\right)\left(1 + \log\left(\max_{a\in A, 2\leq a} a\right) - 1\right)$$

$$C(A \bmod 2) = \sum_{a\in A, a \bmod 2 = 1} 1$$

$$
\begin{aligned}
&\mathcal{C}([A;M]) = 1 + \mathcal{C}(M') \\
&\leq 1 + C\left(\lfloor \frac{A}{2} \rfloor\right) + C(A \bmod 2) + \sum\nolimits_{1 \leq j} C(M^j) - q \quad \text{(by induction)} \\
&= 1 + \sum_{a \in A, 2 \leq a} (\log(a) - 1) + \left(\sum_{a \in A, 2 \leq a} 1\right)\left(1 + \log\left(\max_{a \in A, 2 \leq a} a\right) - 1\right) \\
&\quad + \sum_{a \in A,\, a \bmod 2 = 1} 1 + \sum_{1 \leq j} C(M^j) - q \\
&= 1 - \sum_{a \in A, 2 \leq a} 1 + \sum_{a \in A, 2 \leq a} \log(a) \\
&\quad + \left(\sum_{a \in A, 2 \leq a} 1\right)\left(1 + \log\left(\max_{a \in A, 2 \leq a} a\right)\right) - \sum_{a \in A, 2 \leq a} 1 \\
&\quad + \sum_{a \in A,\, a \bmod 2 = 1} 1 + \sum_{1 \leq j} C(M^j) - q \\
&= 1 - \sum_{a \in A, 2 \leq a} 1 + \sum_{a \in A, 1 \leq a} \log(a) + \left(\sum_{a \in A, 1 \leq a} 1\right)\left(1 + \log\left(\max_{a \in A, 2 \leq a} a\right)\right) \\
&\quad - \left(\sum_{a \in A, 1 = a} 1\right)\left(1 + \log\left(\max_{a \in A, 2 \leq a} a\right)\right) - \sum_{a \in A, 2 \leq a} 1 + \sum_{a \in A,\, a \bmod 2 = 1} 1 \\
&\quad + \sum_{1 \leq j} C(M^j) - q
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{C}([A;M]) = 1 + \mathcal{C}(M') \\
&\leq 1 - \sum_{a \in A, 2 \leq a} 1 + \sum_{a \in A, 1 \leq a} \log(a) + \left(\sum_{a \in A, 1 \leq a} 1\right)\left(1 + \log\left(\max_{a \in A, 2 \leq a} a\right)\right) \\
&\quad - \sum_{a \in A, 1 = a} 1 - \sum_{a \in A, 2 \leq a} 1 + \sum_{a \in A,\, a \bmod 2 = 1} 1 \\
&\quad + \sum_{1 \leq j} C(M^j) - q \\
&\leq 1 - \sum_{a \in A, 2 \leq a} 1 + \sum_{a \in A, 1 \leq a} \log(a) + \left(\sum_{a \in A, 1 \leq a} 1\right)\left(1 + \log\left(\max_{a \in A, 2 \leq a} a\right)\right) \\
&\quad + \sum_{1 \leq j} C(M^j) - q \\
&= 1 - \sum_{a \in A, 2 \leq a} 1 + C(A) + \sum_{1 \leq j} C(M^j) - q
\end{aligned}
$$

Since a square abstraction has been performed, A contains at least an element greater than or equal to 2, hence,

$$\mathcal{C}([A;M]) \leq C(A) + \sum_{1\leq j} C(M^j) - q$$

2. Let us assume that the first step of the algorithm is a product abstraction over some variables and that A and B are their columns of exponents, respectively. By reordering the variables and then the monomials, the matrix of exponents may be given the form

$$\begin{bmatrix} A^+ & B^- & \\ A^= & B^= & M \\ A^- & B^+ & \end{bmatrix}$$

where $A^+ > B^-$, $A^= = B^=$ and $A^- < B^+$ component by component, the first two columns being A and B. Hence,

$$\min(A,B) = \begin{bmatrix} B^- \\ A^= \\ A^- \end{bmatrix} = \begin{bmatrix} B^- \\ B^= \\ A^- \end{bmatrix}$$

$$A - \min(A,B) = \begin{bmatrix} A^+ - B^- \\ 0 \\ 0 \end{bmatrix} \quad B - \min(A,B) = \begin{bmatrix} 0 \\ 0 \\ B^+ - A^- \end{bmatrix}$$

$$\begin{aligned}
\mathcal{C}[A;B;M] &= 1 + \mathcal{C}[A - \min(A,B); B - \min(A,B); \min(A,B); M] \\
&\leq 1 + C(A - \min(A,B)) + C(B - \min(A,B)) \\
&\quad + C(\min(A,B)) + \sum_{j\geq 1} C(M^j) - q \qquad \text{(by induction)} \\
&= 1 + C(A^+ - B^-) + C(B^+ - A^-) \\
&\quad + \sum_{a\in A^-, 1\leq a} \log a + \left(\sum_{a\in A^-, 1\leq a} 1 \right) \left(1 + \log \left(\max_{a\in \min(A,B), 1\leq a} a \right) \right) \\
&\quad + \sum_{a\in A^=, 1\leq a} \log a + \left(\sum_{a\in A^=, 1\leq a} 1 \right) \left(1 + \log \left(\max_{a\in \min(A,B), 1\leq a} a \right) \right) \\
&\quad + \sum_{b\in B^-, 1\leq b} \log b + \left(\sum_{b\in B^-, 1\leq b} 1 \right) \left(1 + \log \left(\max_{b\in \min(A,B), 1\leq b} b \right) \right) \\
&\quad + \sum_{j\geq 1} C(M^j) - q
\end{aligned}$$

It is easy to obtain that

$$\mathcal{C}[A;B;M] \leq 1 + C(A) + C(B) + \sum_{j\geq 1} C(M^j) - q$$

since

$$\begin{aligned} C(A^+ - B^-) + \sum_{a\in A^-, 1\leq a} \log\, a + \left(\sum_{a\in A^-, 1\leq a} 1\right)\left(1 + \log\left(\max_{a\in \min(A,B), 1\leq a} a\right)\right) \\ + \sum_{a\in A^=, 1\leq a} \log\, a + \left(\sum_{a\in A^=, 1\leq a} 1\right)\left(1 + \log\left(\max_{a\in \min(A,B), 1\leq a} a\right)\right) \\ \leq C(A) \end{aligned}$$

and

$$\begin{aligned} C(B^+ - A^-) + \sum_{b\in B^-, 1\leq b} \log\, b + \left(\sum_{b\in B^-, 1\leq b} 1\right)\left(1 + \log\left(\max_{b\in \min(A,B), 1\leq b} b\right)\right) \\ \leq C(B) \end{aligned}$$

We shall prove the actual strictness of the inequality between $\mathcal{C}[A;B;M]$ and $1 + C(A) + C(B) + \sum_{j\geq 1} C(M^j) - q$, since the two inequalities cannot be exact equalities simultaneously. Let us assume that we have indeed exact equalities; this implies that

$$\forall (a,b) \in [A^+; B^-]\ \log(a-b) = \log(a)$$

$$\forall (a,b) \in [A^-; B^+]\ \log(b-a) = \log(b)$$

$$\forall a \in A^= a = 0$$

that is,

$$A^+ > 2B^-, A^= = B^= = 0, B^+ > 2A^-,$$

componentwise, and

$$\begin{aligned} &\left(\sum_{a\in A^-, 1\leq a} 1\right)\left(1 + \log\left(\max_{a\in \min(A,B), 1\leq a} a\right)\right) \\ &= \left(\sum_{a\in A^-, 1\leq a} 1\right)\left(1 + \log\left(\max_{a\in A, 1\leq a} a\right)\right)\left(\sum_{b\in B^-, 1\leq b} 1\right)\left(1 + \log\left(\max_{a\in \min(A,B), 1\leq a} a\right)\right) \\ &= \left(\sum_{b\in B^-, 1\leq b} 1\right)\left(1 + \log\left(\max_{b\in B, 1\leq b} b\right)\right) \end{aligned}$$

Since $A^= = 0$ and we have performed a product abstraction, it is impossible that both A^- and B^- are equal to 0.

a) Let us assume first that $A^- = 0$ and $B^- \neq 0$. The square abstraction over A has not been selected; this means that

$$\sum_{a\in A}\left\lfloor\frac{a}{2}\right\rfloor < \sum_{b\in B^-} b + \sum_{b\in B^=} b + \sum_{a\in A^-} a = \sum_{b\in B^-} b$$

But since $A^+ > 2B^-$, we can deduce that

$$\sum_{b\in B^-} b \leq \sum_{a\in A}\left\lfloor\frac{a}{2}\right\rfloor,$$

which leads to

$$\sum_{a\in A}\left\lfloor\frac{a}{2}\right\rfloor < \sum_{a\in A}\left\lfloor\frac{a}{2}\right\rfloor,$$

a contradiction.

b) The case when $A^- \neq 0$ and $B^- = 0$ is identical.

c) Let us assume now that $A^- \neq 0$ and $B^- \neq 0$. This implies that $(\sum_{a\in A^-,1\leq a} 1) \neq 0$ and $(\sum_{b\in B^-,1\leq b} 1) \neq 0$, hence

$$\log\Big(\max_{a\in\min(A,B),1\leq a} a\Big) = \log\Big(\max_{a\in A,1\leq a} a\Big)\ \log\Big(\max_{a\in\min(A,B),1\leq a} a\Big)$$

$$= \log\Big(\max_{b\in B,1\leq b} b\Big)$$

Let us assume without loss of generality that the maximum of A and B is reached on $a_0 \in A$ (i.e., $a_0 = \max\{\max A, \max B\}$). When a is in $\min(A,B) = A^- \cup B^-$, $2a$ is less than a_0, hence

$$\log\Big(\max_{a\in\min(A,B),1\leq a} a\Big) \leq log(a_0) - 1$$

which contradicts the first one of the above equalities.
The case when

$$\mathcal{C}[A;B;M] = 1 + C(A) + C(B) + \sum_{j\geq 1} C(M^j) - q$$

has been eliminated, hence

$$\mathcal{C}[A;B;M] \leq C(A) + C(B) + \sum_{j\geq 1} C(M^j) - q$$

and we are done. □

Notice that in the case of a single monomial, the bound can be improved:

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

where $|l|$ denotes the length of l, since the subcase 2c) cannot occur (see [12] for more details).

5. Implementation and Experiments

Our method for automatically finding polynomial interpretations that are suitable for proving termination of a given TRS has been implemented in the C*i*ME rewrite tool. We deal with strong termination only, but we point out that the constraints solving part of C*i*ME has also been used by other systems such as Muterm [37, 38] for context-sensitive rewriting and Cariboo [21]. The AProVE [24] tool also has an implementation of polynomial interpretations based on the algorithm described in this paper.

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),
                  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 by 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 [56] 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. [26, 42] are incompatible with this incremental approach, which indeed requires to prove CE-termination.

We detail now 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 [47]: 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 [15]: 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 [11] on how to select this combination with C*i*ME).

Figure 4 summarizes results on these examples, in particular regarding the variable abstraction strategy. Times are obtained on a Pentium III 933 MHz 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 regard 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 nontrivial dependency graph components, hence five sets of Diophantine constraints to solve, and the numbers are given for each of them. The remaining columns give 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 solving thousands of constraints, over hundreds of

TRS	Dioph.		FD (0)		FD (1)		FD (2)	
	cons	var	cons	var	cons	var	cons	var
Rosu *et al.* [47]	170	45	3896	45	3409	854	2353	328
(time)				9.9s		4.7s		2.9s
Deplagne [15]	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 4. Some experimental results.

		FD (0)	FD (1)	FD (2)
solved TRSs	number	304	315	314
	percentage	53.0%	54.9%	54.7%
	average time	1.44s	0.89s	0.47s
unsolved TRSs	number	270	259	260
	percentage	47.0%	45.1%	45.3%
	average time	26.47s	27.02s	26.77s

Figure 5. Some experimental results on 574 TRSs from the TPDB.

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 too 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.

We tried these different abstraction methods on a large collection of TRSs from the Termination Problems Data Base,★ Figure 5 summarizes the results. We proceeded as follows: we stopped unfinished computations after one minute and we forced C*i*ME to search for simple polynomials with a bound on coefficients of 3. These are not the most efficient parameters with reference to the time spent on proof search; some problems indeed require linear interpretations only. However, we are concerned here with the efficiency of constraint solving only and not by the finest tuning of the tool for a given base of termination problems. Several remarks can be made. First, the differences are quite small, because the majority of TRSs in the TPDB are either quite simple or so hard that indeed no polynomial interpretation exist for them. Second, we see that fewer problems are solved when using policy FD(0), hence using policy FD(1) or FD(2) provides more termination power in practice. With respect to the number of problems solved, FD(1) or FD(2) are almost equivalent, indeed FD(1) solved three problems that FD(2) did not, and FD(2) solved two problems that FD(1) did not. This can be understood from the well-known fact that in constraint solving on finite domains, making more variable abstraction is better for finding solution, but of course making variable abstraction can take time: indeed,

★ http://www.lri.fr/~marche/wst2004-competition/tpdb.html.

it can be noticed that FD(2) answers significantly more quickly. Finally, all heuristics take approximately the same average time in failing cases, but this is probably simply a statistical effect of having a one-minute time limit.

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.

An interesting extension of this work would be the use of exponential interpretations, as proposed by Lescanne [36]. However, there is a major problem: the criterion proposed by Lescanne to ensure positiveness of exponential functions is very different from the absolute positiveness criterion and there is no clear way to derive a method for automated search for such interpretations. In other words, an extension to Hong & Jakuš's work [29] to exponential functions should be done first.

In an early work on program proofs [43, 54], Turing remarked that complexity measures for termination could be by transfinite ordinal number. Actually, ordinal notations have been considered in fairly recent work on termination [9, 17] but never used in automated tools. An extension of our technique to ordinal interpretations is then another interesting future work.

Acknowledgements

We thank Pierre Lescanne for his fruitful remarks, which greatly improved our description of the state of the art, and the anonymous referees for their detailed remarks on the first version of this paper.

References

1. Arts, T. and Giesl, J.: Termination of term rewriting using dependency pairs, *Theor. Comp. Sci.* **236** (2000), 133-178.
2. Baader, F. and Nipkow, T.: *Term Rewriting and All That*, Cambridge University Press, 1998.
3. Ben Cherifa, A. and Lescanne, P.: Termination of rewriting systems by polynomial interpretations and its implementation, *Sci. Comput. Program.* **9** (1987), 137–159.
4. Bergeron, F., Berstel, J. and Brlek, S.: Efficient computation of addition chains, *Journal de théorie des nombres de Bordeaux* **6** (1994), 21–38.
5. Bergeron, F., Berstel, J., Brlek, S. and Duboc, C.: Addition chains using continued fractions, *J. Algorithms* **10**(3) (1989), 403–412.
6. Bleichenbacher, D. and Flammenkamp, A.: An Efficient Algorithm for Computing Shortest Addition Chains, http://wwwhomes.uni-bielefeld.de/achim/addition_chain.html, 1997.
7. Bonfante, G., Cichon, A., Marion, J.-Y. and Touzet, H.: Algorithms with polynomial interpretation termination proof, *J. Funct. Program.* **11**(1) (2001), 33–53.

8. Bos, J. and Coster, M.: Addition Chain Heuristics, in G. Brassard (ed.), *Advances in Cryptology – Proc. Crypto '89*, Vol. 435 *of Lecture Notes in Computer Science*, Santa Barbara, California, USA, pp. 400–407, 1989.
9. Cichon, E. A. and Touzet:, H.: An Ordinal Calculus for Proving Termination in Term Rewriting, in H. Kirchner (ed.), *Proceedings 21st International Colloquium on Trees in Algebra and Programming*, Vol. 1059 of *Lecture Notes in Computer Science*, Linköping (Sweden), pp. 226–240, 1996.
10. Codognet, P. and Diaz, D.: Compiling constraints in clp(FD), *J. Log. Program.* **27**(3) (1996), 185–226.
11. Contejean, E., Marché, C. Monate, B. and Urbain, X.: Proving Termination of Rewriting with C*i*ME, in A. Rubio (ed.), *Extended Abstracts of the 6th International Workshop on Termination, WST'03*, 2003, pp. 71-73, http://cime.lri.fr.
12. Contejean, E., Marché, C., Tomás, A.-P. and Urbain, X.: Mechanically proving termination using polynomial interpretations, Research Report 1382, LRI, 2004.
13. Courcelle, B.: Recursive Applicative Program Schemes, in J. van Leeuwen (ed.), *Handbook of Theoretical Computer Science*, Vol. B., North-Holland, Chapt. 9, 1990, pp. 459–492.
14. de Rooij, P.: Efficient exponentiation using precomputation and vector addition chains, in A. D. Santis (ed.), *Advances in Cryptology – EUROCRYPT '94*, Vol. 950 of *Lecture Notes in Computer Science*, Perugia, Italy, 1995, pp. 389–399.
15. Deplagne, É.: Sequent Calculus Viewed Modulo, in Catherine Pilière (ed.), *Proceedings of the Fifth ESSLLI Student Session*, University of Birmingham, 2000, pp. 66–76.
16. Dershowitz, N.: Orderings for term rewriting systems, *Theor. Comp. Sci.* **17**(3) (1982), 279–301.
17. Dershowitz, N.: Trees, Ordinals, and Termination, in M. C. Gaudel and J.-P. Jouannaud (eds.), *4th International Joint Conference on Theory and Practice of Software Development*, Vol. 668 of *Lecture Notes in Computer Science*, Orsay, France, pp. 243–250.
18. Dershowitz, N. and Jouannaud, J.-P.: Rewrite Systems, in J. van Leeuwen (ed.), *Handbook of Theoretical Computer Science*, Vol. B., North-Holland, 1990, pp. 243–320.
19. Dobkin, D. and Lipton, R. J.: Addition chain methods for the evaluation of specific polynomials, *SIAM J. Comput.* **9**(1) (1980), 121–125.
20. Downey, P. J., Leong, B. L. and Sethi, R.: Computing sequences with addition chains, *SIAM J. Comput.* **10**(3) (1981), 638–646.
21. Fissore, O., Gnaedig, I. and Kirchner, H.: CARIBOO: An Induction Based Proof Tool for Termination with Strategies, in *Proc. Fourth International Conference on Principles and Practice of Declarative Programming (PPDP)*, Pittsburgh, USA, 2002, pp. 62–73, http://www.loria.fr/~fissore/CARIBOO.
22. Giesl, J.: Generating Polynomial Orderings for Termination Proofs, in J. Hsiang (ed.), *6th International Conference on Rewriting Techniques and Applications*, Vol. 914 of *Lecture Notes in Computer Science*, Kaiserslautern, Germany, 1995, pp. 426–431.
23. Giesl, J., Arts, T. and Ohlebusch, E.: Modular termination proofs for rewriting using dependency Pairs, *J. Symb. Comput.* **34**(2) (2002), 21–58.
24. Giesl, J., Thiemann, R., Schneider-Kamp, P. and Falke, S.: AProVE: A System for Proving Termination, in A. Rubio (ed.), *Extended Abstracts of the 6th International Workshop on Termination, WST'03*, 2003, pp. 68–70, http://www-i2.informatik.rwth-aachen.de/AProVE.
25. Gramlich, B. and Lucas, S.: Simple Termination of Context-Sensitive Rewriting, in B. Fischer and E. Visser (eds.), in *Proc. 3rd ACM Sigplan Workshop on Rule-Based Programming, RULE'02*, Pittsburgh, USA, 2002, pp. 29–41.
26. Hirokawa, N. and Middeldorp, A.: Approximating Dependency Graphs without using Tree Automata Techniques, in A. Rubio (ed.), *Extended Abstracts of the 6th International Workshop on Termination, WST'03*, 2003, pp. 8–10, Technical Report DSIC II/15/03, Universidad Politécnica de Valencia, Spain.

27. Hirokawa, N. and Middeldorp, A.: Tsukuba Termination Tool, in R. Nieuwenhuis (ed.), *14th International Conference on Rewriting Techniques and Applications*, Vol. 2706 of *Lecture Notes in Computer Science*, Valencia, Spain, 2003, pp. 311–320.
28. Hofbauer, D. and Lautermann, C.: Termination Proofs and the Length of Derivations, in N. Dershowitz (ed.), *Rewriting Techniques and Applications*, Vol. 355 of *Lecture Notes in Computer Science*, Chapel Hill, USA, 1989, pp. 167–177.
29. Hong, H. and Jakuš, D.: Testing positiveness of polynomials, *J. Autom. Reason.* **21**(1) (1998), 23–38.
30. Jaffar, J. and Lassez, J.-L.: Constraint Logic Programming, in *Proceedings of the 14th ACM Conference on Principles of Programming Languages*, Munich, Germany, 1987, pp. 111–119.
31. Klop, J. W.: Term rewriting systems, in S. Abramsky, D. Gabbay, and T. Maibaum (eds.), *Handbook of Logic in Computer Science*, Vol. 2., Clarendon, 1992, pp. 1–116.
32. Knuth, D. E.: *The art of computer programming*, 2nd edn, Vol. 2., Addison-Wesley, 1981.
33. Kusakari, K., Nakamura, M. and Toyama, Y.: Argument Filtering Transformation, in G. Nadathur (ed.), *Principles and Practice of Declarative Programming, International Conference PPDP'99*, Vol. 1702 of *Lecture Notes in Computer Science*, Paris, 1999, pp. 47–61.
34. Lang, S.: *Algebra*, 3rd edn, Addison-Wesley, 1993.
35. Lankford, D. S.: On proving term rewriting systems are Noetherian, Technical Report MTP-3, Mathematics Department, Louisiana Tech. Univ., 1979. Available at http://perso.ens-lyon.fr/pierre.lescanne/not_accessible.html.
36. Lescanne, P.: Elementary interpretations in proofs of termination, *Form. Asp. Comput.* **7** (1995), 77–90.
37. Lucas, S.: Termination of (Canonical) Context-Sensitive Rewriting, in S. Tison (ed.), *13th International Conference on Rewriting Techniques and Applications*, Vol. 2378 of *Lecture Notes in Computer Science*, Copenhagen, Denmark, 2002, pp. 296–310.
38. Lucas, S.: 2003, Mu-term, A Tool for Proving Termination of Rewriting with Replacement Restrictions, 2002, Available at http://www.dsic.upv.es/~slucas/csr/termination/muterm/.
39. Manna, Z. and Ness, S.: On the termination of Markov algorithms, in *Proc. Third Hawaii International Conference on Systems Sciences*, Honolulu, HI, 1970, pp. 789–792. http://perso.ens-lyon.fr/pierre.lescanne/not_accessible.html.
40. Matiyasevich, Y. V.: Enumerable sets are diophantine, *Sov. Math. (Dokladi)* **11**(2) (1970), 354–357.
41. Matiyasevich, Y. V.: *Hilbert's Tenth Problem*, MIT Press, 1993.
42. Middeldorp, A.: Approximating Dependency Graphs using Tree Automata Techniques, in R. Goré, A. Leitsch, and T. Nipkow (eds.), *International Joint Conference on Automated Reasoning*, Vol. 2083 of *Lecture Notes in Artificial Intelligence*, Siena, Italy, 2001, pp. 593–610.
43. Morris, F. L. and Jones, C. B.: An early program proof by Alan Turing, *Ann. Hist. Comput.* **6**(2) (1984), 139–143.
44. Olivos, J.: On vectorial addition chains, *J. Algorithms* **2**(1) (1981), 13–21.
45. Papadimitriou, C. and Knuth, D.: Duality in addition chains, *Bulletin of the EACTS* **13** (1981), 2–3.
46. Pippenger, N.: On the evaluation of powers and monomials, *SIAM J. Comput.* **9**(2) (1980), 230–250.
47. Rosu, G. and Viswanathan, M.: Testing Extended Regular Language Membership Incrementally by Rewriting, in R. Nieuwenhuis (ed.), *14th International Conference on Rewriting Techniques and Applications*, Vol. 2706 of *Lecture Notes in Computer Science*, 2003, Valencia, Spain.

48. Rouyer, J.: 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.
49. Rouyer, J.: 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) (1991), 157–169.
50. Sauerbrey, J. and Dietel, A.: Resource Requirements for the Application of Addition Chains in Modulo Exponentiation, in R. Rueppel (ed.), *Advances in Cryptology – EUROCRYPT '92*, Vol. 658 of *Lecture Notes in Computer Science*, Balatonfüred, Hungary, 1993, pp. 174–182.
51. Schonhage, A.: A lower bound for the length of addition chains, *Theor. Comp. Sci.* **1**(1) (1975), 1–12.
52. Steinbach, J.: Proving Polynomials Positive, in R. Shyamasundar (ed.), *Foundations of Software Technology and Theoretical Computer Science*, Vol. 652 of *Lecture Notes in Computer Science*, New Delhi, India, 1992, pp. 191–202.
53. Terese, Bezem, M., Klop, J.W. and de Vrijer, R. (eds.) (2003) *Term Rewriting Systems*, Vol. 55 of *Cambridge Tracts in Theoretical Computer Science*, Cambridge University Press, 1992, http://www.cs.vu.nl/~terese/.
54. Tarski, A.: *A Decision Method for Elementary Algebra and Geometry*, University of California Press, Berkeley, CA, 1951.
55. Turing, A. M.: Checking a large routine, in *Report of a Conference on High Speed Automatic Calculing Machines*, Cambridge, 1949, pp. 67–69.
56. Urbain, X.: Modular and incremental automated termination proofs, *J. Autom. Reason.* **32**, (2004) 315–355.
57. Zantema, H.: Minimizing Sums of Addition Chains, *J. Algorithms* **12** (1991), 281–307.