A Folding Rule for Eliminating Existential Variables from Constraint Logic Programs[*]

Valerio Senni
Department of Informatics, Systems, and Production
University of Rome Tor Vergata
Via del Politecnico 1, 00133 Rome, Italy
senni@disp.uniroma2.it

Alberto Pettorossi
Department of Informatics, Systems, and Production
University of Rome Tor Vergata
Via del Politecnico 1, 00133 Rome, Italy
pettorossi@disp.uniroma2.it

Maurizio Proietti
IASI-CNR
Viale Manzoni 30, 00185 Rome, Italy
maurizio.proietti@iasi.cnr.it

Abstract. The existential variables of a clause in a constraint logic program are the variables which occur in the body of the clause and not in its head. The elimination of these variables is a transformation technique which is often used for improving program efficiency and verifying program properties. We consider a folding transformation rule which ensures the elimination of existential variables and we propose an algorithm for applying this rule in the case where the constraints are linear inequations over rational or real numbers. The algorithm combines techniques for matching terms modulo equational theories and techniques for solving systems of linear inequations. Through some examples we show that an implementation of our folding algorithm has a good performance in practice.

Address for correspondence: Valerio Senni, DISP, University of Rome "Tor Vergata", Via del Politecnico 1, 00133 Rome, Italy
[*]This paper is a revised version of [17].

Keywords: Program transformation, folding rule, variable elimination, constraint logic programming

1. Introduction

Constraint logic programming is a very expressive language for writing programs in a declarative way and for specifying and verifying properties of software systems [9]. When writing programs in a declarative style or writing specifications, one often uses *existential variables*, that is, variables which occur in the body of a clause and not in its head. However, the use of existential variables may give rise to inefficient or even nonterminating computations (and this may happen when an existential variable denotes an intermediate data structure or when an existential variable ranges over an infinite set). For this reason some transformation techniques have been proposed for eliminating those variables from logic programs and constraint logic programs [13, 14]. These techniques make use of the *unfolding* and *folding* rules which have been first proposed in the context of functional programming by Burstall and Darlington [5], and then extended to logic programming [19, 20] and to constraint logic programming [3, 7, 8, 11].

For instance, let us consider the problem of checking whether or not a list L of rational numbers has a prefix P such that the sum of all elements of P is at least M. A constraint logic program that solves this problem is the following:

1. $\mathit{prefixsum}(L,M) \leftarrow N \geq M \ \wedge\ \mathit{app}(P,S,L) \ \wedge\ \mathit{sum}(P,N)$
2. $\mathit{app}([\,],Y,Y) \leftarrow$
3. $\mathit{app}([H|X],Y,[H|Z]) \leftarrow \mathit{app}(X,Y,Z)$
4. $\mathit{sum}([\,],0) \leftarrow$
5. $\mathit{sum}([H|X],N) \leftarrow N = H+R \ \wedge\ \mathit{sum}(X,R)$

When answering queries which are instances of the atom $\mathit{prefixsum}(L,M)$, the program computes values for the variables P, S, and N, which are the existential variables of clause 1 and are not needed in the final answer. We can eliminate these existential variables and improve the efficiency of the program, by applying the unfolding and folding rules as follows. From clause 1, by applying the unfolding rule several times, we derive:

6. $\mathit{prefixsum}(L,M) \leftarrow 0 \geq M$
7. $\mathit{prefixsum}([H|T],M) \leftarrow N \geq M \ \wedge\ N = H+R \ \wedge\ \mathit{app}(P,S,T) \ \wedge\ \mathit{sum}(P,R)$

Now we fold clause 7 by using clause 1 and we derive:

8. $\mathit{prefixsum}([H|T],M) \leftarrow \mathit{prefixsum}(T,M-H)$

For this folding step we have used the fact that, in our theory of constraints, clause 7 is equivalent to the clause $\mathit{prefixsum}([H|T],M) \leftarrow R \geq M-H \wedge \mathit{app}(P,S,T) \wedge \mathit{sum}(P,R)$, whose body is an instance of the body of clause 1. The final program, consisting of clauses 6 and 8, has no existential variables and, thus, does not construct unnecessary intermediate values for computing the relation $\mathit{prefixsum}$.

As shown in the above example, the folding rule plays a particularly relevant role in the techniques for eliminating existential variables. (In particular, it would have been impossible to eliminate all existential variables from the clauses defining $\mathit{prefixsum}$ by using the unfolding rule only.) For that reason in this paper we focus our attention on the folding rule, which in the general case can be defined as follows.

Let (i) H and K be atoms, (ii) c and d be constraints, and (iii) G and B be goals (that is, conjunctions of literals). Given two clauses γ: $H \leftarrow c \wedge G$ and δ: $K \leftarrow d \wedge B$, if there exist a constraint e, a substitution ϑ, and a goal R such that $H \leftarrow c \wedge G$ is equivalent (w.r.t. a given theory of constraints) to $H \leftarrow e \wedge (d \wedge B)\vartheta \wedge R$, then γ is folded into the clause η: $H \leftarrow e \wedge K\vartheta \wedge R$. In order to use the folding rule to eliminate existential variables we also require that every variable occurring in $K\vartheta$ also occurs in H.

In the literature no algorithm is provided to determine whether or not, given a theory of constraints, the suitable e, ϑ, and R which are required for folding, do exist [3, 7, 8, 11]. In this paper we propose an algorithm based on linear algebra and term rewriting techniques for computing e, ϑ, and R, if they exist, in the case when the constraints are linear inequations over the rational numbers. The techniques we will present are valid without relevant changes also when the inequations are over the real numbers. As an example of application of the folding algorithm, let us consider the following clauses:

γ: $p(X_1, X_2, X_3) \leftarrow X_1 < 1 \wedge X_1 \geq Z_1 + 1 \wedge Z_2 > 0 \wedge q(Z_1, f(X_3), Z_2) \wedge r(X_2)$
δ: $s(Y_1, Y_2, Y_3) \leftarrow W_1 < 0 \wedge Y_1 - 3 \geq 2W_1 \wedge W_2 > 0 \wedge q(W_1, Y_3, W_2)$

and suppose that we want to fold γ using δ for eliminating the existential variables Z_1 and Z_2 occurring in γ. Our folding algorithm **FA** computes (see Examples 4.1–4.4 in Section 4): (i) the constraint e: $X_1 < 1$, (ii) the substitution ϑ: $\{Y_1/2X_1 + 1,\ Y_2/a, Y_3/f(X_3), W_1/Z_1, W_2/Z_2\}$, where a is an arbitrary new constant, and (iii) the goal R: $r(X_2)$, and the clause derived by folding γ using δ is:

η: $p(X_1, X_2, X_3) \leftarrow X_1 < 1 \wedge s(2X_1 + 1, a, f(X_3)) \wedge r(X_2)$

which has no existential variables. (The correctness of this folding step can easily be checked by unfolding η w.r.t. $s(2X_1 + 1, a, f(X_3))$.) In general, a triple $\langle e, \vartheta, R\rangle$ that satisfies the conditions for the applicability of the folding rule may not exist or may not be unique. For this reason our folding algorithm is nondeterministic and, in different executions, it may compute different folded clauses.

The paper is organized as follows. In Section 2 we introduce some basic definitions concerning constraint logic programs. In Section 3 we present the folding rule which we use for eliminating existential variables. In Section 4 we describe our algorithm for applying the folding rule and we prove the soundness and completeness of this algorithm with respect to the declarative specification of the rule. In Section 5 we analyze the complexity of our folding algorithm. We also describe an implementation of that algorithm and we evaluate its performance by presenting some experimental results. Finally, in Section 6 we discuss the related work and we suggest some directions for future investigations.

2. Preliminary Definitions

In this section we recall some basic definitions concerning constraint logic programs, where the constraints are conjunctions of linear inequations over the rational numbers. As already mentioned, the results we will present in this paper are valid without relevant changes also when the constraints are conjunctions of linear inequations over the real numbers. For notions not defined here the reader may refer to [9, 10].

Let us consider a first order language $\mathcal{L}$ given by a set Var of variables, a set Fun of function symbols, and a set $Pred$ of predicate symbols. Let $+$ denote addition, $\cdot$ denote multiplication, and $\mathbb{Q}$ denote the set of rational numbers. We assume that $\{+, \cdot\} \cup \mathbb{Q} \subseteq Fun$ (in particular, every rational number is assumed to be a 0-ary function symbol). We also assume that the predicate symbols $\geq$ and $>$ denoting inequality and strict inequality, respectively, belong to $Pred$.

In order to distinguish terms representing rational numbers from other terms (which may be viewed as finite trees), we assume that $\mathcal{L}$ is a typed language [10] with two basic types: `rat`, which is the type of the rational numbers, and `tree`, which is the type of the finite trees. We also consider types constructed from basic types by the usual type constructors $\times$ and $\rightarrow$. A variable $X \in \mathit{Var}$ has either type `rat` or type `tree`. We denote by $\mathit{Var}_{\texttt{rat}}$ and $\mathit{Var}_{\texttt{tree}}$ the set of variables of type `rat` and `tree`, respectively. A predicate symbol of arity n and a function symbol of arity n in $\mathcal{L}$ have types of the form $\tau_1 \times \cdots \times \tau_n$ and $\tau_1 \times \cdots \times \tau_n \rightarrow \tau_{n+1}$, respectively, for some types $\tau_1, \ldots, \tau_n, \tau_{n+1} \in \{\texttt{rat}, \texttt{tree}\}$. In particular, the predicate symbols $\geq$ and $>$ have type $\texttt{rat}\times\texttt{rat}$, the function symbols $+$ and $\cdot$ have type $\texttt{rat}\times\texttt{rat} \rightarrow \texttt{rat}$, and the rational numbers have type `rat`. The function symbols in $\{+, \cdot\} \cup \mathbb{Q}$ are the only symbols whose type is $\tau_1 \times \cdots \times \tau_n \rightarrow \texttt{rat}$, for some types $\tau_1, \ldots, \tau_n$, with $n \geq 0$.

A *term* u is either a *term of type* `rat` or a *term of type* `tree`. A term p of type `rat` is a *linear polynomial* of the form $a_1X_1 + \ldots + a_nX_n + a_{n+1}$, where $a_1, \ldots, a_{n+1}$ are rational numbers and $X_1, \ldots, X_n$ are variables in $\mathit{Var}_{\texttt{rat}}$ (a *monomial* of the form aX stands for the term $a \cdot X$). A term t of type `tree` is either a variable X in $\mathit{Var}_{\texttt{tree}}$ or a term of the form $f(u_1, \ldots, u_n)$, where f is a function symbol of type $\tau_1 \times \cdots \times \tau_n \rightarrow \texttt{tree}$, and $u_1, \ldots, u_n$ are terms of type $\tau_1, \ldots, \tau_n$, respectively.

An *atomic constraint* is a linear inequation of the form $p_1 \geq p_2$ or $p_1 > p_2$. A *constraint* is a conjunction $c_1 \wedge \ldots \wedge c_n$, where $c_1, \ldots, c_n$ are atomic constraints. When $n = 0$ we write $c_1 \wedge \ldots \wedge c_n$ as *true*. A constraint of the form $p_1 \geq p_2 \wedge p_2 \geq p_1$ is abbreviated as the equation $p_1 = p_2$ (which, thus, is not an atomic constraint).

An *atom* is of the form $r(u_1, \ldots, u_n)$, where r is a predicate symbol, not in $\{\geq, >\}$, of type $\tau_1 \times \ldots \times \tau_n$ and $u_1, \ldots, u_n$ are terms of type $\tau_1, \ldots, \tau_n$, respectively. A *literal* is either an atom (called a *positive literal*) or a negated atom (called a *negative literal*). A *goal* is a conjunction $L_1 \wedge \ldots \wedge L_n$ of literals, with $n \geq 0$. The conjunction of 0 literals is denoted by *true*. A *constrained goal* is a conjunction $c \wedge G$, where c is a constraint and G is a goal. A *clause* is of the form $H \leftarrow c \wedge G$, where H is an atom and $c \wedge G$ is a constrained goal. A *constraint logic program* is a set of clauses. A *formula* of the language $\mathcal{L}$ is constructed as usual in first order logic from the symbols of $\mathcal{L}$ by using the logical connectives $\wedge$, $\vee$, $\neg$, $\rightarrow$, $\leftarrow$, $\leftrightarrow$, and the quantifiers $\exists$, $\forall$.

If f is a term or a formula then by $\mathit{Vars}_{\texttt{rat}}(f)$ and $\mathit{Vars}_{\texttt{tree}}(f)$ we denote, respectively, the set of variables of type `rat` and of type `tree` occurring in f. By $\mathit{Vars}(f)$ we denote the set of all variables occurring in f, that is, $\mathit{Vars}_{\texttt{rat}}(f) \cup \mathit{Vars}_{\texttt{tree}}(f)$. A similar notation will also be used for the variables occurring in sets of terms and sets of formulas. Given a clause γ: $H \leftarrow c \wedge G$, by $\mathit{EVars}(\gamma)$ we denote the set of the *existential variables* of γ, which is defined to be $\mathit{Vars}(c \wedge G) - \mathit{Vars}(H)$. The *constraint-local* variables of γ are the variables in the set $\mathit{Vars}(c) - \mathit{Vars}(\{H, G\})$. Given a set $X = \{X_1, \ldots, X_n\}$ of variables and a formula φ, by $\forall X\, \varphi$ we denote the formula $\forall X_1 \ldots \forall X_n\, \varphi$ and by $\exists X\, \varphi$ we denote the formula $\exists X_1 \ldots \exists X_n\, \varphi$. By $\forall(\varphi)$ and $\exists(\varphi)$ we denote the *universal closure* and the *existential closure* of φ, respectively. In what follows we will use the notion of *substitution* as defined in [10] with the following extra condition on types: given any substitution $\{X_1/t_1, \ldots, X_n/t_n\}$, for $i = 1, \ldots, n$, the type of X_i is equal to the type of t_i.

Let $\mathcal{L}_{\texttt{rat}}$ denote the sublanguage of $\mathcal{L}$ given by the set $\mathit{Var}_{\texttt{rat}}$ of variables, the set $\{+, \cdot\} \cup \mathbb{Q}$ of function symbols, and the set $\{\geq, >\}$ of predicate symbols. Throughout the paper we will denote by $\mathcal{Q}$ the interpretation which assigns to every symbol in $\{+, \cdot\} \cup \mathbb{Q} \cup \{\geq, >\}$ the expected function or relation on $\mathbb{Q}$. For a formula φ of $\mathcal{L}_{\texttt{rat}}$ (and, in particular, for a constraint), the satisfaction relation $\mathcal{Q} \models \varphi$ is defined as usual in first order logic. A *$\mathcal{Q}$-interpretation* is an interpretation I for the typed language $\mathcal{L}$ which agrees with $\mathcal{Q}$ for each formula φ of $\mathcal{L}_{\texttt{rat}}$, that is, for each φ of $\mathcal{L}_{\texttt{rat}}$, $I \models \varphi$ iff $\mathcal{Q} \models \varphi$. The definition of a $\mathcal{Q}$-interpretation for typed languages is a straightforward extension of the one for untyped languages [9]. We say that a $\mathcal{Q}$-interpretation I is a *$\mathcal{Q}$-model* of a program P if for every clause $\gamma \in P$

we have that $I \models \forall(\gamma)$. Similarly to the case of logic programs, we can define *stratified* constraint logic programs and in [8, 9, 11] it is shown that every such program P has a *perfect* $\mathcal{Q}$-model, denoted by $M(P)$.

A *solution* of a set C of constraints is a ground substitution σ of the form $\{X_1/a_1, \ldots, X_n/a_n\}$, where $\{X_1, \ldots, X_n\} = \mathit{Vars}(C)$ and $a_1, \ldots, a_n \in \mathbb{Q}$, such that $\mathcal{Q} \models c\,\sigma$ for every $c \in C$. A set of constraints is said to be *satisfiable* if it has a solution.

We assume that we are given a function solve that takes as input a set C of constraints and returns a solution σ of C, if C is satisfiable, and **fail** otherwise. The function solve can be implemented, for instance, by using the Fourier-Motzkin algorithm or the Khachiyan algorithm [16]. We assume that we are also given a function $\mathit{project}$ such that for every constraint c and for every finite set of variables $X \subseteq \mathit{Var}_{\mathtt{rat}}$, $\mathcal{Q} \models \forall X\, ((\exists Y\, c) \leftrightarrow \mathit{project}(c, X))$, where $Y = \mathit{Vars}(c) - X$ and $\mathit{Vars}(\mathit{project}(c, X)) \subseteq X$. The $\mathit{project}$ function can be implemented, for instance, by using the Fourier-Motzkin algorithm or the algorithm presented in [22].

A clause $\gamma\colon H \leftarrow c \wedge G$ is said to be in *normal form* if (i) every term of type `rat` occurring in G is a variable, (ii) each variable of type `rat` occurs at most once in G, (iii) $\mathit{Vars}_{\mathtt{rat}}(H) \cap \mathit{Vars}_{\mathtt{rat}}(G) = \emptyset$, and (iv) γ has no constraint-local variables. It is always possible to transform any clause γ_1 into a clause γ_2 such that γ_2 has the same $\mathcal{Q}$-models as γ_1 and γ_2 is in normal form. Clause γ_2 is called *a normal form of* γ_1. In particular, from a clause γ_1, we can compute a clause γ_1' that satisfies conditions (i)–(iii) by introducing a new variable and a corresponding equation for each outermost occurrence of a term of type `rat` in G. Clause γ_1' is computed in linear time w.r.t. the size of γ_1. By applying the *project* function, we can eliminate the constraint-local variables from γ_1' and obtain a clause γ_2 that satisfies also condition (iv). In the worst case, the application of the *project* function takes exponential time in the number of variables to be eliminated [22]. Without loss of generality, when presenting the folding rule and the algorithm for its application, we will assume that the clauses are in normal form.

Definition 2.1. Given two clauses γ_1 and γ_2, we write $\gamma_1 \cong \gamma_2$ if there exist a normal form $H \leftarrow c_1 \wedge B_1$ of γ_1, a normal form $H \leftarrow c_2 \wedge B_2$ of γ_2, and a renaming substitution ρ such that: (1) $H = H\rho$, (2) $B_1 =_{AC} B_2\rho$, and (3) $\mathcal{Q} \models \forall\, (c_1 \leftrightarrow c_2\rho)$, where $=_{AC}$ denotes equality modulo the equational theory of associativity and commutativity of conjunction. We will refer to this theory as the *AC*$_\wedge$ *theory* [1].

Proposition 2.1. (i) The relation $\cong$ is an equivalence relation. (ii) If $\gamma_1 \cong \gamma_2$ then, for every $\mathcal{Q}$-interpretation I, $I \models \gamma_1$ iff $I \models \gamma_2$. (iii) If γ_2 is a normal form of γ_1 then $\gamma_1 \cong \gamma_2$.

3. The Folding Rule

In this section we introduce our folding transformation rule which is a variant of the folding rules considered in the literature [3, 7, 8, 11, 19, 20]. In particular, by using our variant of the folding rule we may replace a constrained goal occurring in the body of a clause where some existential variables occur, by an atom which has no existential variables in the folded clause.

Definition 3.1. (Folding Rule)
Let γ: $H \leftarrow c \wedge G$ and δ: $K \leftarrow d \wedge B$ be clauses in normal form without variables in common. Suppose also that there exist a constraint e, a substitution ϑ, and a goal R such that: (1) $\gamma \;\cong\; H \leftarrow e \wedge d\vartheta \wedge B\vartheta \wedge R$; (2) for every variable X in $\mathit{EVars}(\delta)$, the following conditions hold: (2.1) $X\vartheta$ is a variable not occurring

in $\{H, e, R\}$, and (2.2) $X\vartheta$ does not occur in the term $Y\vartheta$, for every variable Y occurring in $d \wedge B$ and different from X; (3) $\mathit{Vars}(K\vartheta) \subseteq \mathit{Vars}(H)$. By *folding clause* γ *using clause* δ we derive the clause η: $H \leftarrow e \wedge K\vartheta \wedge R$.

Condition (3) ensures that no existential variable of η occurs in $K\vartheta$. However, in e or R some existential variables may still occur. These variables may be eliminated by further folding steps using again clause δ or other clauses. In Theorem 3.1 below we will establish the correctness of the folding rule w.r.t. the perfect model semantics. This correctness result follows immediately from [19].

In order to state Theorem 3.1 we need the following notion. A *transformation sequence* is a sequence $P_0, \ldots, P_n$ of programs such that, for $k = 0, \ldots, n-1$, program P_{k+1} is derived from program P_k by an application of one of the following transformation rules: *definition*, *unfolding* (w.r.t. *positive* literals), and *folding*. For a detailed presentation of the definition and unfolding rules for constraint logic programs we refer to [8]. An application of the folding rule is defined as follows. For $k = 0, \ldots, n$, by Defs_k we denote the set of clauses introduced by the definition rule during the construction of $P_0, \ldots, P_k$. Program P_{k+1} is derived from program P_k by an application of the folding rule if $P_{k+1} = (P_k - \{\gamma\}) \cup \{\eta\}$, where γ is a clause in P_k, δ is a clause in Defs_k, and η is the clause derived by folding γ using δ as indicated in Definition 3.1.

Theorem 3.1. Let P_0 be a stratified program and let $P_0, \ldots, P_n$ be a transformation sequence. Suppose that, for $k = 0, \ldots, n-1$, if P_{k+1} is derived from P_k by folding clause γ using clause $\delta \in \mathit{Defs}_k$, then there exists j, with $0 < j < n$, such that $\delta \in P_j$ and P_{j+1} is derived from P_j by unfolding δ w.r.t. a positive literal in its body. Then $P_0 \cup \mathit{Defs}_n$ and P_n are stratified and $M(P_0 \cup \mathit{Defs}_n) = M(P_n)$.

4. An Algorithm for Applying the Folding Rule

Now we will present an algorithm for determining whether or not a clause $\gamma : H \leftarrow c \wedge G$ can be folded using a clause $\delta : K \leftarrow d \wedge B$, according to Definition 3.1. The objective of our folding algorithm is to find a constraint e, a substitution ϑ, and a goal R such that Point (1) (that is, $\gamma \cong\ H \leftarrow e \wedge d\vartheta \wedge B\vartheta \wedge R$), Point (2), and Point (3) of Definition 3.1 hold. Our algorithm computes e, ϑ, and R, if they exist, by applying two procedures: (i) the *goal matching procedure*, called **GM**, which matches the goal G against B and returns a substitution α and a goal R such that $G =_{AC} B\alpha \wedge R$, and (ii) the *constraint matching procedure*, called **CM**, which matches the constraint c against $d\,\alpha$ and returns a substitution β and a constraint e such that c is equivalent to $e \wedge d\,\alpha\,\beta$ in the theory of constraints. The substitution ϑ to be found is the composition, denoted $\alpha\,\beta$, of the substitutions α and β. The output of the folding algorithm is either the clause η: $H \leftarrow e \wedge K\vartheta \wedge R$, if folding is possible, or **fail**, if folding is not possible. Since Definition 3.1 does not uniquely determine e, ϑ, and R, our folding algorithm is nondeterministic and, as already mentioned, in different executions it may compute different folded clauses.

4.1. Goal Matching

Let us now present the goal matching procedure **GM**. This procedure uses the notion of binding which is defined as follows: a *binding* is a pair of the form e_1/e_2, where e_1 and e_2 are either both goals or both terms. Thus, the notion of *set of bindings* is a generalization of the notion of substitution.

Goal Matching Procedure: GM

Input: two clauses in normal form without variables in common $\gamma\colon H \leftarrow c \wedge G$ and $\delta\colon K \leftarrow d \wedge B$.
Output: a substitution α and a goal R such that: (1) $G =_{AC} B\alpha \wedge R$; (2) for every variable X in $EVars(\delta)$, (2.1) $X\alpha$ is a variable not occurring in $\{H, R\}$, and (2.2) $X\alpha$ does not occur in the term $Y\alpha$, for every variable Y occurring in $d \wedge B$ and different from X; (3) $Vars_{\texttt{tree}}(K\alpha) \subseteq Vars(H)$. If such α and R do not exist, then **fail**.

Consider a set $Bnds$ of bindings initialized to the singleton $\{(B \wedge T)/G\}$, where T is a new symbol denoting a variable ranging over goals. Consider also the rewrite rules (i)–(x) listed below. In the left hand sides of these rules, whenever we write $S \cup Bnds$, for any set S of bindings, we assume that $S \cap Bnds = \emptyset$.

(i) $\{(L_1 \wedge B_1 \wedge T) \,/\, (G_1 \wedge L_2 \wedge G_2)\} \cup Bnds \Longrightarrow \{L_1/L_2,\ (B_1 \wedge T)/(G_1 \wedge G_2)\} \cup Bnds$
where: (1) L_1 and L_2 are either both positive or both negative literals and have the same predicate symbol with the same arity, and (2) B_1, G_1, and G_2 are (possibly empty) conjunctions of literals;

(ii) $\{\neg A_1/\neg A_2\} \cup Bnds \Longrightarrow \{A_1/A_2\} \cup Bnds$;

(iii) $\{a(s_1, \ldots, s_n)/a(t_1, \ldots, t_n)\} \cup Bnds \Longrightarrow \{s_1/t_1, \ldots, s_n/t_n\} \cup Bnds$;

(iv) $\{a(s_1, \ldots, s_m)/b(t_1, \ldots, t_n)\} \cup Bnds \Longrightarrow$ **fail**, if a is different from b or $m \neq n$;

(v) $\{a(s_1, \ldots, s_n)/X\} \cup Bnds \Longrightarrow$ **fail**, if $X \in Vars(\gamma)$;

(vi) $\{X/s\} \cup Bnds \Longrightarrow$ **fail**, if $X \in Vars(\delta)$ and $X/t \in Bnds$ for some t syntactically different from s;

(vii) $\{X/s\} \cup Bnds \Longrightarrow$ **fail**, if $X \in EVars(\delta)$ and one of the following three conditions holds: (1) s is not a variable, or (2) $s \in Vars(H)$, or (3) there exists $Y \in Vars(d \wedge B)$ different from X such that (3.1) $Y/t \in Bnds$, for some term t, and (3.2) $s \in Vars(t)$;

(viii) $\{X/s,\ T/G_1\} \cup Bnds \Longrightarrow$ **fail**, if $X \in EVars(\delta)$ and $s \in Vars(G_1)$;

(ix) $\{X/s\} \cup Bnds \Longrightarrow$ **fail**, if $X \in Vars_{\texttt{tree}}(K)$ and $Vars(s) \not\subseteq Vars(H)$;

(x) $Bnds \Longrightarrow \{X/s\} \cup Bnds$, where s is an arbitrary term of type `tree` such that $Vars(s) \subseteq Vars(H)$, if $X \in Vars_{\texttt{tree}}(K) - Vars(B)$ and there is no term t such that $X/t \in Bnds$.

IF there exist a set of bindings α (which, by construction, is a substitution) and a goal R such that: (c1) $\{(B \wedge T)/G\} \Longrightarrow^* \alpha \cup \{T/R\}$ (where $T/R \notin \alpha$) and (c2) no $Bnds$ exists such that $\alpha \cup \{T/R\} \Longrightarrow Bnds$ (that is, informally, $\alpha \cup \{T/R\}$ is a maximally rewritten, non-failing set of bindings derived from the singleton $\{(B \wedge T)/G\}$)
THEN return α and R ELSE return **fail**.

Rule (i) associates each literal in B with a literal in G in a nondeterministic way. Rules (ii)–(vi) are a specialization to our case of the usual rules for matching [21]. Rules (vii)–(x) ensure that any pair $\langle \alpha, R \rangle$ computed by **GM** satisfies Conditions (2) and (3) of the folding rule, or if no such pair exists, then **GM** returns **fail**.

Example 4.1. Let us apply the procedure **GM** to the clauses γ and δ presented in the Introduction, where the predicates p, q, r, and s are of type `rat`$\times$`tree`$\times$`tree`, `rat`$\times$`tree`$\times$`rat`, `tree`, and `rat`$\times$`tree`$\times$`tree`, respectively, and the function f is of type `tree`$\rightarrow$`tree`. The clauses γ and δ are in normal form and have no variables in common. The procedure **GM** performs the following rewritings, where the arrow $\overset{r}{\Longrightarrow}$ denotes an application of the rewrite rule r:

$$\{q(W_1, Y_3, W_2) \wedge T/(q(Z_1, f(X_3), Z_2) \wedge r(X_2))\}$$
$$\overset{\text{i}}{\Longrightarrow} \{q(W_1, Y_3, W_2)/q(Z_1, f(X_3), Z_2),\ \ T/r(X_2)\}$$
$$\overset{\text{iii}}{\Longrightarrow} \{W_1/Z_1,\ \ Y_3/f(X_3),\ \ W_2/Z_2,\ \ T/r(X_2)\}$$
$$\overset{\text{x}}{\Longrightarrow} \{W_1/Z_1,\ \ Y_3/f(X_3),\ \ W_2/Z_2,\ \ Y_2/a,\ \ T/r(X_2)\}$$

In the final set of bindings, the term a is an arbitrary constant of type `tree`. The output of **GM** is the substitution α: $\{W_1/Z_1,\ Y_3/f(X_3),\ W_2/Z_2,\ Y_2/a\}$ and the goal R: $r(X_2)$.

The goal matching procedure **GM** is *sound* in the sense that if **GM** returns a substitution α and a goal R, then α and R satisfy the output conditions of **GM**. The goal matching procedure is also *complete* in the sense that if there exist a substitution α and a goal R that satisfy the output conditions of **GM**, then **GM** does not return **fail**. The termination of the goal matching procedure can be shown via an argument based on the multiset ordering of the size of the bindings. Indeed, each of the rules (i)–(ix) replaces a binding by a finite number of smaller bindings, and rule (x) can be applied at most once for each variable occurring in the head of clause δ. A detailed proof of the soundness, completeness, and termination of **GM** can be found in [18].

4.2. Constraint Matching

Let us assume that given two clauses in normal form $\gamma:\ H \leftarrow c \wedge G$ and $\delta:\ K \leftarrow d \wedge B$, the goal matching procedure **GM** returns the substitution α and the goal R. By using α and R, we construct the two clauses in normal form: $H \leftarrow c \wedge B\alpha \wedge R$ and $K\alpha \leftarrow d\alpha \wedge B\alpha$ such that $G =_{AC} B\alpha \wedge R$. The constraint matching procedure **CM** takes as input these two clauses we have constructed. For reasons of simplicity, we rename them as γ': $H \leftarrow c \wedge B' \wedge R$ and δ': $K' \leftarrow d' \wedge B'$, respectively. The procedure **CM** returns as output a constraint e and a substitution β such that: (1) $\gamma' \cong H \leftarrow e \wedge d'\beta \wedge B' \wedge R$, (2) $B'\beta = B'$, (3) $\mathit{Vars}(K'\beta) \subseteq \mathit{Vars}(H)$, and (4) $\mathit{Vars}(e) \subseteq \mathit{Vars}(\{H, R\})$. If such e and β do not exist, then the procedure **CM** returns **fail**.

Let $\widetilde{e}$ denote the constraint $\mathit{project}(c, X)$, where $X = \mathit{Vars}(c) - \mathit{Vars}(B')$ (the definition of the $\mathit{project}$ function is given in Section 2). By Lemma 4.1 below, the procedure **CM** does not lose any solution if it returns as constraint e the value of $\widetilde{e}$, and then compute a substitution β such that $\mathcal{Q} \models \forall(c \leftrightarrow (\widetilde{e} \wedge d'\beta))$, $B'\beta = B'$, and $\mathit{Vars}(K'\beta) \subseteq \mathit{Vars}(H)$ hold.

Lemma 4.1. Let $\gamma':\ H \leftarrow c \wedge B' \wedge R$ and $\delta':\ K' \leftarrow d' \wedge B'$ be the input clauses to the constraint matching procedure. For every substitution β, there exists a constraint e such that the following four conditions hold: (1) $\gamma' \cong H \leftarrow e \wedge d'\beta \wedge B' \wedge R$, (2) $B'\beta = B'$, (3) $\mathit{Vars}(K'\beta) \subseteq \mathit{Vars}(H)$, and (4) $\mathit{Vars}(e) \subseteq \mathit{Vars}(\{H, R\})$ iff $\mathcal{Q} \models \forall(c \leftrightarrow (\widetilde{e} \wedge d'\beta))$ and Conditions (2) and (3) hold.

The following example illustrates the fact that if the procedure **CM** returns for the constraint e the value of $\widetilde{e}$, then **CM** may compute the substitution β by solving a set of constraints over the set $\mathbb{Q}$ of the rational numbers.

Example 4.2. Let us consider again the clauses γ and δ of the Introduction. Let α and $r(X_2)$ be the substitution and the goal computed by applying the procedure **GM** to γ and δ as shown in the above Example 4.1. Let us then consider the following clauses $\gamma' : H \leftarrow c \wedge B' \wedge R$ and $\delta' : K' \leftarrow d' \wedge B'$ which are equal to γ and $\delta\alpha$, respectively:

γ': $p(X_1, X_2, X_3) \leftarrow X_1 < 1 \ \wedge\ X_1 \geq Z_1 + 1 \ \wedge\ Z_2 > 0 \ \wedge\ q(Z_1, f(X_3), Z_2) \ \wedge\ r(X_2)$

δ': $s(Y_1, a, f(X_3)) \leftarrow Z_1 < 0 \wedge Y_1 - 3 \geq 2Z_1 \wedge Z_2 > 0 \wedge q(Z_1, f(X_3), Z_2)$

Thus, the constraint c is $X_1 < 1 \ \wedge\ X_1 \geq Z_1 + 1 \ \wedge\ Z_2 > 0$ and the goal B' is $q(Z_1, f(X_3), Z_2)$. Those two clauses γ' and δ' are the input to the procedure **CM**. The constraint $\widetilde{e}$ returned by the procedure **CM** is $\mathit{project}((X_1 < 1 \ \wedge\ X_1 \geq Z_1 + 1 \ \wedge\ Z_2 > 0), \{X_1\})$, which is equivalent to $X_1 < 1$.

Now we will compute a substitution β such that: (i) $\mathcal{Q} \models \forall (c \leftrightarrow (\widetilde{e} \wedge d'\beta))$ holds, and (ii) Conditions (2) and (3) as stated in Lemma 4.1, hold. These three conditions are as follows:

$$\mathcal{Q} \models \forall\, (X_1 < 1 \wedge X_1 \geq Z_1 + 1 \wedge Z_2 > 0 \ \leftrightarrow\ X_1 < 1 \wedge (Z_1 < 0 \wedge Y_1 - 3 \geq 2Z_1 \wedge Z_2 > 0)\beta) \qquad (f.0)$$

$$q(Z_1, f(X_3), Z_2)\beta = q(Z_1, f(X_3), Z_2) \qquad \text{(that is, } Z_1\beta = Z_1,\ X_3\beta = X_3,\ Z_2\beta = Z_2\text{)} \qquad (2)$$

$$\mathit{Vars}(s(Y_1, a, f(X_3))\beta) \subseteq \{X_1, X_2, X_3\} \qquad (3)$$

We have that Equivalence $(f.0)$ holds if the following equivalences $(f.1)$, $(f.2)$, and $(f.3)$, and implication $(f.4)$ hold:

$$\mathcal{Q} \models \forall\, (X_1 < 1 \leftrightarrow X_1 < 1) \qquad (f.1)$$

$$\mathcal{Q} \models \forall\, (X_1 \geq Z_1 + 1 \ \leftrightarrow\ (Y_1 - 3 \geq 2Z_1)\beta) \qquad (f.2)$$

$$\mathcal{Q} \models \forall\, (Z_2 > 0 \ \leftrightarrow\ (Z_2 > 0)\beta) \qquad (f.3)$$

$$\mathcal{Q} \models \forall\, (X_1 < 1 \wedge X_1 \geq Z_1 + 1 \wedge Z_2 > 0 \ \rightarrow\ (Z_1 < 0)\beta) \qquad (f.4)$$

Equivalence $(f.1)$ trivially holds. Equivalence $(f.2)$ can be reduced to an equation over the rational numbers because Equivalence $(f.2)$ holds if there exists a rational number $k > 0$ such that

$$\mathcal{Q} \models \forall\, (k(X_1 - Z_1 - 1) = (Y_1 - 3 - 2Z_1)\beta)$$

holds. By Condition (2), the substitution β is the identity on Z_1 and, hence, the equation $k(X_1 - Z_1 - 1) = (Y_1 - 3 - 2Z_1)\beta$ holds for any β such that

$$Y_1\beta = (2 - k)Z_1 + kX_1 + 3 - k$$

Now we determine the value of the parameter k and, hence, the substitution β, as follows. Since by Condition (3) $\mathit{Vars}(s(Y_1, a, f(X_3))\beta) \subseteq \{X_1, X_2, X_3\}$ we get that, for every value of Z_1, $(2 - k)Z_1 = 0$. Thus, $k = 2$ and, by replacing k by 2 in the equation above, we get the new equation $Y_1\beta = 2X_1 + 1$. This equation is satisfied if the binding $Y_1/(2X_1 + 1)$ belongs to β. Finally, we have that Equivalences $(f.3)$ and $(f.4)$ hold for $\beta = \{Y_1/(2X_1 + 1)\}$. We will see that, indeed, the substitution β we have obtained is the one returned by the constraint matching procedure **CM** we will introduce ibelow.

The crucial steps in Example 4.2 have been the following two: (i) the reduction of Equivalence $(f.0)$ to a set of equivalences between *atomic* constraints (see $(f.1)$–$(f.3)$) or implications with *atomic* conclusions (see $(f.4)$), and (ii) the reduction of one of these equivalences, namely $(f.2)$, to an equation over the rational numbers, via the introduction of the auxiliary rational parameter k.

Now we introduce some notions and we state some properties (see Lemma 4.2 and Theorem 4.1) which will be exploited by the constraint matching procedure **CM** for performing in the general case those two reduction steps. Indeed, the procedure **CM** consists of a set of rewrite rules which reduce the equivalence between c and $\widetilde{e} \wedge d'\beta$ to a set of equations and inequations over the rational numbers, via the introduction of suitable auxiliary parameters. The properties we now state also provide sufficient conditions which guarantee the construction of the desired substitution β, if there exists one.

A conjunction $a_1 \wedge \ldots \wedge a_m$ of (not necessarily distinct) atomic constraints $a_1, \ldots, a_m$ is said to be *redundant* if $\mathcal{Q} \models \forall((a_1 \wedge \ldots \wedge a_{i-1} \wedge a_{i+1} \wedge \ldots \wedge a_m) \rightarrow a_i)$ for some $i \in \{1, \ldots, m\}$. In this case we say that a_i is redundant in $a_1 \wedge \ldots \wedge a_m$. Thus, the empty conjunction *true* is non-redundant and an atomic constraint a is redundant iff $\mathcal{Q} \models \forall(a)$. Given a redundant constraint c, we can always derive a non-redundant constraint c' which is equivalent to c, that is, $\mathcal{Q} \models \forall(c \leftrightarrow c')$, by repeatedly eliminating from the constraint at hand an atomic constraint which is redundant in that constraint.

Without loss of generality, we may assume that any given constraint c is of the form $p_1 \rhd_1 0 \wedge \ldots \wedge p_m \rhd_m 0$, with $m \geq 0$ and $\rhd_1, \ldots, \rhd_m \in \{\geq, >\}$. We define the $\mathit{interior}$ of c, denoted $\mathit{interior}(c)$, to be the constraint $p_1 > 0 \wedge \ldots \wedge p_m > 0$.

A constraint c is said to be *admissible* if both c and $\mathit{interior}(c)$ are satisfiable and non-redundant. For instance, the constraint $c_1 : X - Y \geq 0 \wedge Y \geq 0$ is admissible, while the constraint $c_2 : X - Y \geq 0 \wedge Y \geq 0 \wedge X > 0$ is not admissible (indeed, c_2 is non-redundant, but $\mathit{interior}(c_2) : X - Y > 0 \wedge Y > 0 \wedge X > 0$ is redundant). The following Lemma 4.2 characterizes the equivalence between two constraints whenever one of them is admissible.

Lemma 4.2. Let us consider an admissible constraint a of the form $a_1 \wedge \ldots \wedge a_m$ and a constraint b of the form $b_1 \wedge \ldots \wedge b_n$, where $a_1, \ldots, a_m, b_1, \ldots, b_n$ are atomic constraints (in particular, they are not equalities). We have that $\mathcal{Q} \models \forall\,(a \leftrightarrow b)$ holds iff there exists an injection $\mu : \{1, \ldots, m\} \rightarrow \{1, \ldots, n\}$ such that for $i = 1, \ldots, m$, $\mathcal{Q} \models \forall\,(a_i \leftrightarrow b_{\mu(i)})$ and for $j = 1, \ldots, n$, if $j \notin \{\mu(i) \mid 1 \leq i \leq m\}$, then $\mathcal{Q} \models \forall\,(a \rightarrow b_j)$.

In Lemma 4.2 we have required that the constraint a be admissible. This is a needed hypothesis as the following example shows. Let us consider the non-admissible constraint $c_2 : X - Y \geq 0 \wedge Y \geq 0 \wedge X > 0$ and the constraint $c_3 : X - Y \geq 0 \wedge Y \geq 0 \wedge X + Y > 0$. We have that $\mathcal{Q} \models \forall(c_2 \leftrightarrow c_3)$ and yet there is no injection μ which has the properties stated in Lemma 4.2.

Given the clauses $\gamma' : H \leftarrow c \wedge B' \wedge R$ and $\delta' : K' \leftarrow d' \wedge B'$ such that: (i) c is an admissible constraint of the form $a_1 \wedge \ldots \wedge a_m$, and (ii) $\widetilde{e} \wedge d'$ is a constraint of the form $b_1 \wedge \ldots \wedge b_n$, where $\widetilde{e}$ is $\mathit{project}(c, \mathit{Vars}(c) - \mathit{Vars}(B'))$, the constraint matching procedure **CM** may exploit Lemma 4.2 and compute a substitution β which satisfies $\mathcal{Q} \models \forall(c \leftrightarrow (\widetilde{e} \wedge d'\beta))$ and Conditions (2) and (3) of Lemma 4.1, according to the following algorithm: first (1) **CM** computes an injection μ from $\{1, \ldots, m\}$ to $\{1, \ldots, n\}$, (see rule (i) in the procedure **CM** below) and then (2) it computes β such that:

(2.i) for $i = 1, \ldots, m$, $\mathcal{Q} \models \forall(a_i \leftrightarrow b_{\mu(i)}\beta)$, and

(2.ii) for $j = 1, \ldots, n$, if $j \notin \{\mu(i) \mid 1 \leq i \leq m\}$, then $\mathcal{Q} \models \forall(c \rightarrow b_j\beta)$

(see rules (ii)–(v) in the procedure **CM** below).

By Lemma 4.2, one can show that if the constraint c is admissible, the above algorithm for computing the substitution β which satisfies $\mathcal{Q} \models \forall(c \leftrightarrow (\widetilde{e} \wedge d'\beta))$ and Conditions (2) and (3) of Lemma 4.1 is *complete* in the sense that it computes such a substitution β if there exists one. Note that, if the constraint c is non-admissible then it can be the case that there is no injection μ which satisfies the conditions provided in Lemma 4.2 and yet clause γ can be folded using δ, according to Definition 3.1. In this case, the procedure **CM** fails.

In order to compute β satisfying Point (2.i) above, the procedure **CM** makes use of the following *Property* $P1$: given the satisfiable, non-redundant atomic constraints $p > 0$ and $q > 0$, we have that

$\mathcal{Q} \models \forall(p>0 \leftrightarrow q>0)$ holds iff there exists a rational number $k>0$ such that $\mathcal{Q} \models \forall(kp-q=0)$ holds. Property $P1$ holds also if we consider $\forall(p\geq 0 \leftrightarrow q\geq 0)$, instead of $\forall(p>0 \leftrightarrow q>0)$.

In order to compute β satisfying Point (2.ii) above, the procedure **CM** makes use of the following Theorem 4.1 which is a generalization of the above Property $P1$ and it is an extension of Farkas' Lemma to the case of systems of weak ($\geq$) and strict ($>$) inequalities [16], rather than weak inequalities only.

Theorem 4.1. Suppose that $p_1 \rhd_1 0, \ldots, p_m \rhd_m 0$, $p_{m+1} \rhd_{m+1} 0$ are atomic constraints such that, for $i=1,\ldots,m+1$, $\rhd_i \in \{\geq,>\}$ and $\mathcal{Q} \models \exists(p_1 \rhd_1 0 \wedge \ldots \wedge p_m \rhd_m 0)$. Then $\mathcal{Q} \models \forall(p_1 \rhd_1 0 \wedge \ldots \wedge p_m \rhd_m 0 \rightarrow p_{m+1} \rhd_{m+1} 0)$ iff there exist $k_1 \geq 0, \ldots, k_{m+1} \geq 0$ such that: (i) $\mathcal{Q} \models \forall\,(k_1 p_1 + \cdots + k_m p_m + k_{m+1} = p_{m+1})$, and (ii) if $\rhd_{m+1}$ is $>$ then $(\sum_{i\in I} k_i)>0$, where $I=\{i \mid 1\leq i\leq m+1,\ \rhd_i \text{ is } >\}$.

As we will see, the constraint matching procedure **CM** may construct *bilinear* polynomials (see rules (i)–(iii)), which defined as follows. Let p be a polynomial and $\langle P_1, P_2\rangle$ be a partition of a (proper or not) superset of $\mathit{Vars}(p)$. The polynomial p is said to be *bilinear in the partition* $\langle P_1, P_2\rangle$ if there exists a polynomial q such that $\mathcal{Q} \models \forall\,(p=q)$ and q is a sum of monomials, each of which is of the form: *either* (i) $k\,VU$, where k is a rational number, $V \in P_1$, and $U \in P_2$, *or* (ii) $k\,U$, where k is a rational number and $U \in P_1 \cup P_2$, *or* (iii) k, where k is a rational number.

Given a polynomial p which is bilinear in the partition $\langle P_1, P_2\rangle$, where $P_2 = \{U_1,\ldots,U_m\}$, a *normal form* of p, denoted $\mathit{nf}(p)$, *w.r.t. a given linear order* $U_1,\ldots,U_m$ *of the variables in* P_2, is *any* polynomial which is derived from p by: (i) computing a polynomial of the form $r_1U_1 + \cdots + r_mU_m + r_{m+1}$ such that: (i.1) $\mathcal{Q} \models \forall\,(p = r_1U_1+\cdots+r_mU_m+r_{m+1})$, and (i.2) $r_1,\ldots,r_{m+1}$ are linear polynomials whose variables are in P_1, and (ii) erasing from that polynomial every summand r_iU_i such that $\mathcal{Q} \models \forall\,(r_i=0)$.

In what follows, we will extend our terminology and we will call a constraint any conjunction $c_1 \wedge \ldots \wedge c_n$ of formulas, where for $i=1,\ldots,n$, c_i is of the form $p \geq 0$ or $p>0$ and p is a bilinear polynomial.

Constraint Matching Procedure: CM

Input: two clauses in normal form, possibly with variables in common, $\gamma' : H \leftarrow c \wedge B' \wedge R$ and $\delta' : K' \leftarrow d' \wedge B'$.
Output: a constraint e and a substitution β such that: (1) $\gamma' \cong H \leftarrow e \wedge d'\beta \wedge B' \wedge R$, (2) $B'\beta = B'$, (3) $\mathit{Vars}(K'\beta) \subseteq \mathit{Vars}(H)$, and (4) $\mathit{Vars}(e) \subseteq \mathit{Vars}(\{H,R\})$. If such e and β do not exist, then **fail**.

IF c is unsatisfiable THEN return an arbitrary unsatisfiable constraint e such that $\mathit{Vars}(e) \subseteq \mathit{Vars}(\{H,R\})$ and a substitution β of the form $\{U_1/a_1,\ldots,U_s/a_s\}$, where $\{U_1,\ldots,U_s\} = \mathit{Vars}_{\texttt{rat}}(K')$ and $a_1,\ldots,a_s$ are arbitrary terms of type `rat` such that, for $i=1,\ldots,s$, $\mathit{Vars}(a_i) \subseteq \mathit{Vars}(H)$
ELSE proceed as follows.

Let X be the set $\mathit{Vars}(c) - \mathit{Vars}_{\texttt{rat}}(B')$, Y be the set $\mathit{Vars}(d') - \mathit{Vars}_{\texttt{rat}}(B')$, and Z be the set $\mathit{Vars}_{\texttt{rat}}(B')$. Let e be the constraint $\mathit{project}(c,X)$. Without loss of generality, we may assume that:

– c is a constraint of the form $p_1 \rhd_1 0 \wedge \ldots \wedge p_m \rhd_m 0$, where for $i=1,\ldots,m$, p_i is a linear polynomial and $\rhd_i \in \{\geq,>\}$, and

– $e \wedge d'$ is a constraint of the form $q_1 \rhd_1 0 \wedge \ldots \wedge q_n \rhd_n 0$, where for $j=1,\ldots,n$, q_i is a linear polynomial and $\rhd_i \in \{\geq,>\}$.

Let us consider the following rewrite rules (i)–(v) which are all of the form:

$$\langle f_1 \leftrightarrow g_1,\ S_1,\ \sigma_1\rangle \Longrightarrow \langle f_2 \leftrightarrow g_2,\ S_2,\ \sigma_2\rangle$$

where: (1.1) f_1 and f_2 are constraints, (1.2) g_1 and g_2 are conjunctions of constraints of the form $q \rhd 0$, where q is a bilinear polynomial and $\rhd \in \{\geq, >\}$, (2) S_1 and S_2 are sets of constraints of the form $q \rhd 0$, where q is a bilinear polynomial and $\rhd \in \{\geq, >\}$, and (3) σ_1 and σ_2 are substitutions. Recall that an equation between polynomials of the form $p_1 = p_2$ stands for the two inequations $p_1 \geq p_2$ and $p_2 \geq p_1$. The polynomials occurring in g_1, g_2, S_1, and S_2 are all bilinear in the partition $\langle W, X \cup Y \cup Z\rangle$, where W is the set of the new variables introduced during the application of the rewrite rules (i)–(v). The normal forms of those bilinear polynomials are all defined w.r.t. any fixed variable ordering of the form: $Z_1, \ldots, Z_h$, $Y_1, \ldots, Y_k$, $X_1, \ldots, X_\ell$, where $\{Z_1, \ldots, Z_h\} = Z$, $\{Y_1, \ldots, Y_k\} = Y$, and $\{X_1, \ldots, X_\ell\} = X$. In the rewrite rules (iv) and (v), where S_1 is written as $A \cup S$, we assume that $A \cap S = \emptyset$.

(i) $\langle p \rhd 0 \wedge f \leftrightarrow g_1 \wedge q \rhd 0 \wedge g_2,\ S,\ \sigma\rangle \Longrightarrow \langle f \leftrightarrow g_1 \wedge g_2,\ \{nf(Vp - q) = 0, V > 0\} \cup S,\ \sigma\rangle$

where V is a new variable and either both occurrences of $\rhd$ are $\geq$ or both occurrences of $\rhd$ are $>$;

(ii) $\langle true \leftrightarrow q \geq 0 \wedge g,\ S,\ \sigma\rangle \Longrightarrow$

$\langle true \leftrightarrow g,\ \{nf(V_1 p_1 + \ldots + V_m p_m + V_{m+1} - q) = 0, V_1 \geq 0, \ldots, V_{m+1} \geq 0\} \cup S,\ \sigma\rangle$

where $V_1, \ldots, V_{m+1}$ are new variables and the constraint c in clause γ' is $p_1 \rhd_1 0 \wedge \ldots \wedge p_m \rhd_m 0$;

(iii) $\langle true \leftrightarrow q > 0 \wedge g,\ S,\ \sigma\rangle \Longrightarrow$

$\langle true \leftrightarrow g,\ \{nf(V_1 p_1 + \ldots + V_m p_m + V_{m+1} - q) = 0,$

$V_1 \geq 0, \ldots, V_{m+1} \geq 0,\ (\sum_{i \in I} V_i) > 0\} \cup S,\ \sigma\rangle$

where $V_1, \ldots, V_{m+1}$ are new variables, $I = \{i \mid 1 \leq i \leq m+1,\ \rhd_i \text{ is } >\}$, and the constraint c in clause γ' is $p_1 \rhd_1 0 \wedge \ldots \wedge p_m \rhd_m 0$;

(iv) $\langle f \leftrightarrow g,\ \{pU + q = 0\} \cup S,\ \sigma\rangle \Longrightarrow \langle f \leftrightarrow g,\ \{p = 0, q = 0\} \cup S,\ \sigma\rangle$

if $U \in X \cup Z$;

(v) $\langle f \leftrightarrow g,\ \{aU + q = 0\} \cup S,\ \sigma\rangle \Longrightarrow$

$\langle f \leftrightarrow (g\{U/-\frac{q}{a}\}),\ \{nf(p\{U/-\frac{q}{a}\}) \rhd 0 \mid p \rhd 0 \in S\},\ \sigma\{U/-\frac{q}{a}\}\rangle$

if $U \in Y$, $Vars(q) \cap Vars(R) = \emptyset$, $a \in (\mathbb{Q} - \{0\})$, and $\rhd \in \{\geq, >\}$;

IF there exist a set C of atomic constraints and a substitution σ_Y such that: (c1) $\langle c \leftrightarrow e \wedge d',\ \emptyset,\ \emptyset\rangle \Longrightarrow^* \langle true \leftrightarrow true,\ C,\ \sigma_Y\rangle$, (c2) for every $f \in C$, we have that f is of the form $p \rhd 0$, where p is a linear polynomial and $\rhd \in \{\geq, >\}$, and $Vars(f) \subseteq W$, where W is the set of the new variables introduced during the rewriting steps from $\langle c \leftrightarrow e \wedge d',\ \emptyset,\ \emptyset\rangle$ to $\langle true \leftrightarrow true,\ C,\ \sigma_Y\rangle$, and (c3) C is satisfiable and $solve(C) = \sigma_W$,
THEN construct a ground substitution σ_G of the form $\{U_1/a_1, \ldots,\ U_s/a_s\}$, where $\{U_1, \ldots, U_s\} = Vars_{\texttt{rat}}(K'\sigma_Y\sigma_W) - Vars(H)$ and $a_1, \ldots, a_s$ are arbitrary terms of type `rat` such that, for $i = 1, \ldots, s$, $Vars(a_i) \subseteq Vars(H)$, and return the constraint e and the substitution $\beta = \varphi_Y\, \sigma_G$, where φ_Y is the substitution $\sigma_Y \sigma_W$ restricted to the set Y,
ELSE return **fail**.

Note that the procedure **CM** is nondeterministic (in particular, rule (i) associates an atomic constraint in c with an atomic constraint in $e \wedge d'$ in a nondeterministic way). Note also that in order to apply rules (iv) and (v), $p\,U$ and $a\,U$ should be the leftmost monomials in the bilinear polynomials $p\,U + q$ and $a\,U + q$, respectively.

The procedure **CM** is *sound* in the sense that if it returns the constraint e and the substitution β, then e and β satisfy the output Conditions (1)–(4) of **CM**. Now we sketch the proof of this soundness property. A detailed proof is given in [18]. By Lemma 4.1 it is enough to show that, for $e = \mathit{project}(c, X)$, $\mathcal{Q} \models \forall\,(c \leftrightarrow e \wedge d'\beta)$ and the output Conditions (2) and (3) hold. By the definition of the sets X, Y, Z, and W of variables we may assume, without loss of generality, that $X\beta = X$, $Z\beta = Z$, and $Z \cap \mathit{Vars}(Y\beta) = \emptyset$, and $W\beta = W$, that is, the substitution β is a mapping from Y to terms with variables not in Z (for a proof of these facts, see [18]). Hence, it is enough to show that the substitution β is such that $\mathcal{Q} \models \forall\,(c \leftrightarrow (e \wedge d')\beta)$ (note that β is applied also to the constraint e) and Conditions (2) and (3) hold.

The procedure **CM** starts from the initial triple $\langle c \leftrightarrow e \wedge d', \emptyset, \emptyset\rangle$ and nondeterministically constructs a sequence of triples by applying the rewrite rules (i)–(v) until Conditions (c1)–(c3) are satisfied. If no such sequence exists, **CM** returns **fail**. We will say that a substitution β *satisfies* a triple $\langle f \leftrightarrow g,\ S,\ \sigma\rangle$ if there exists a value for the variables in the set W such that $\mathcal{Q} \models \forall X\,\forall Z\,(f \leftrightarrow g\beta)$, $\mathcal{Q} \models \forall X\,\forall Z\,(S\beta)$, and, for every variable $U \in Y$, $\mathcal{Q} \models \forall(U\sigma\beta = U\beta)$ (note that a variable of the set W may occur either in the constraint g, or in the set S, or in the substitution σ).

Now we show that each rewrite rule which constructs from an old triple $\langle f_1 \leftrightarrow g_1,\ S_1,\ \sigma_1\rangle$ a new triple $\langle f_2 \leftrightarrow g_2,\ S_2,\ \sigma_2\rangle$, is sound in the sense that, for all substitutions β, if β satisfies the triple $\langle f_2 \leftrightarrow g_2, S_2,\ \sigma_2\rangle$ then β satisfies also the triple $\langle f_1 \leftrightarrow g_1,\ S_1,\ \sigma_1\rangle$. Moreover, if β satisfies the initial triple $\langle c \leftrightarrow e \wedge d', \emptyset, \emptyset\rangle$ then β is a correct output substitution.

Let us now consider each of the rewrite rules (i)–(v) and let us show that this rule is sound.

Let us start from rule (i). When applying this rule, for each atomic constraint $p \rhd 0$ in f_1 **CM** selects an atomic constraint $q \rhd 0$ in f_2. Thus, by a sequence of applications of rule (i) starting from the initial triple $\langle c \leftrightarrow e \wedge d', \emptyset, \emptyset\rangle$, **CM** constructs an injective mapping from the atomic constraints in c to the atomic constraints in $e \wedge d'$. If such an injective mapping does not exist, **CM** returns **fail**. Rule (i) deletes the selected atomic constraints $p \rhd 0$ and $q \rhd 0$ and adds to the second component of the triple the equation $\mathit{nf}(Vp - q) = 0$ and the constraint $V > 0$. The soundness of rule (i) follows from Property $P1$, which ensures that $\mathcal{Q} \models \forall\,(p \rhd 0 \leftrightarrow (q \rhd 0)\beta)$ iff there exists a rational number $V > 0$ such that $\mathcal{Q} \models \forall\,(\mathit{nf}(Vp - q\beta) = 0)$.

Rules (ii) and (iii) are applied when the first component of the triple at hand is of the form $\mathit{true} \leftrightarrow g$, that is, none of the atomic constraints in g belongs to the image of the injection computed by rule (i). Every application of rules (ii) and (iii) deletes an atomic constraint $q \rhd 0$ from g and adds to the second component of the triple the equation $\mathit{nf}(V_1 p_1 + \ldots + V_m p_m + V_{m+1} - q) = 0$ and a set $\{V_1 \geq 0, \ldots, V_{m+1} \geq 0\}$ of constraints (with an additional constraint of the form $(\sum_{i \in I} V_i) > 0$ in case of rule (iii)). The soundness of rules (ii) and (iii) follows from the fact that c is a constraint of the form $p_1 \rhd_1 0 \wedge \ldots \wedge p_m \rhd_m 0$ and, by Theorem 4.1, we have that $\mathcal{Q} \models \forall\,(c \rightarrow (q \rhd 0)\beta)$ iff there exist rational numbers $V_1 \geq 0, \ldots, V_{m+1} \geq 0$ such that $\mathcal{Q} \models \forall\,(\mathit{nf}(V_1 p_1 + \ldots + V_m p_m + V_{m+1} - q\beta) = 0)$ (with the additional constraint $(\sum_{i \in I} V_i) > 0$ in case of rule (iii)).

The soundness of rules (iv) and (v) is based on the following *Property $P2$*: $\mathcal{Q} \models \forall((pU + q = 0) \leftrightarrow (p = 0 \wedge q = 0) \vee (p \neq 0 \wedge U = -\frac{q}{p}))$.

Rule (iv) replaces an equation $pU+q=0$, where $U \in X \cup Z$, by the two equations $p=0$ and $q=0$. The soundness of this rule follows from the fact that, for any value of the variables $V_1, \ldots, V_r \in W$, $\mathcal{Q} \models \forall T\,((pU+q)\beta=0)$ iff $\mathcal{Q} \models \forall T\,(p=0)$ and $\mathcal{Q} \models \forall T\,(q\beta=0)$, where $T = \mathit{Vars}((pU)\beta, q\beta, p) - W$. This equivalence follows from Property $P2$, by observing that: (1) $(pU)\beta = pU$ because $U \in X \cup Z$ and $pU+q$ is bilinear in $\langle W, X \cup Y \cup Z\rangle$ and, therefore, $\mathit{Vars}(p) \subseteq W$, and (2) the case where $\mathcal{Q} \models \forall T\,(U = -\frac{q\beta}{p})$ is impossible because, for any β, $U \notin \mathit{Vars}(q\beta)$ (indeed: (2.1) since $pU+q$ is in normal form, we have that $U \notin \mathit{Vars}(q)$, (2.2) since $Z \cap \mathit{Vars}(Y\beta) = \emptyset$, if $U \in Z$ then we have that $U \notin \mathit{Vars}(q\beta)$, and (2.3) since by the variable ordering we use for computing normal forms we have that no variable in the set Y occurs in $pU+q$ to the right of a variable in the set X, if $U \in X$ then we have that $Y \cap \mathit{Vars}(q) = \emptyset$ and, thus, $q\beta = q$).

Rule (v) deletes an equation $aU+q=0$, where $U \in Y$, $\mathit{Vars}(q) \cap \mathit{Vars}(R) = \emptyset$, and $a \in \mathbb{Q} - \{0\}$, and applies the substitution $\{U/-\frac{q}{a}\}$ to all components of the triple at hand. (Note that U does not occur in f.) The soundness of this rule follows from the fact that, for any value of the variables $V_1, \ldots, V_r \in W$, $\mathcal{Q} \models \forall T\,((aU+q)\beta=0)$ iff $\mathcal{Q} \models \forall T\,(U\beta = -\frac{q\beta}{a})$, where $T = \mathit{Vars}(U\beta, q\beta) - W$. This equivalence follows from Property $P2$, because $a \in \mathbb{Q} - \{0\}$. (Note that the condition $\mathit{Vars}(q) \cap \mathit{Vars}(R) = \emptyset$ is required to satisfy the output Condition (3) of **CM**.)

If the rewriting process terminates and from the initial triple $\langle c \leftrightarrow e \wedge d', \emptyset, \emptyset\rangle$ we derive, by a sequence of applications of rules (i)–(v), a new triple $\langle \mathit{true} \leftrightarrow \mathit{true}, C, \sigma_Y\rangle$ such that Conditions (c1)–(c3) listed at the end of the procedure hold, then no rule can be applied to the triple $\langle \mathit{true} \leftrightarrow \mathit{true}, C, \sigma_Y\rangle$ and, hence, in the set C there is no occurrence of a variable in $X \cup Y \cup Z$. Moreover, C is a set of constraints on the variables in the set W. Since by Condition (c3) the set of constraints in C is satisfiable and since β is defined as $\varphi_Y \sigma_G$, where φ_Y is the restriction of the substitution $\sigma_Y \sigma_W$ to the set Y of variables, we have that the substitution β satisfies the triple $\langle \mathit{true} \leftrightarrow \mathit{true}, C, \sigma_Y\rangle$. Therefore, by the soundness of the rewrite rules shown above, we get that the substitution β computed by the procedure **CM** satisfies also the initial triple $\langle c \leftrightarrow e \wedge d', \emptyset, \emptyset\rangle$ and, thus, it is a correct output substitution.

As already mentioned, by using Lemma 4.2, it can be shown that if c is an admissible constraint, the procedure **CM** is also *complete*, in the sense that if there exist a constraint e and a substitution β that satisfy the output conditions of **CM**, then **CM** does not return **fail** (see [18] for a detailed proof).

The termination of the constraint matching procedure is a consequence of the following facts: (1) each application of rules (i), (ii), and (iii) reduces the number of atomic constraints occurring in g in the triple $\langle f \leftrightarrow g,\ S,\ \sigma\rangle$ at hand; (2) each application of rule (iv) does not modify the first component of the triple $\langle f \leftrightarrow g,\ S,\ \sigma\rangle$ at hand, does not introduce any new variables, and reduces the number of occurrences in S of the variables in the set $X \cup Z$; (3) each application of rule (v) does not modify the number of atomic constraints in the first component of the triple $\langle f \leftrightarrow g,\ S,\ \sigma\rangle$ at hand and eliminates all occurrences in S of a variable in the set Y. Thus, the termination of **CM** can be proved by a suitable lexicographic ordering on the number of the atomic constraints and variables. The details of the termination proof can be found in [18].

The following example illustrates an execution of the procedure **CM**.

Example 4.3. Let us consider again the clauses γ and δ of the Introduction and let α be the substitution computed by applying the procedure **GM** to γ and δ as shown in Example 4.1. Let us also consider the clauses γ' and δ', where γ' is γ and δ' is $\delta\alpha$, that is,

γ': $p(X_1, X_2, X_3) \leftarrow X_1 < 1 \wedge X_1 \geq Z_1 + 1 \wedge Z_2 > 0 \wedge q(Z_1, f(X_3), Z_2) \wedge r(X_2)$

δ': $s(Y_1, a, f(X_3)) \leftarrow Z_1 < 0 \wedge Y_1 - 3 \geq 2Z_1 \wedge Z_2 > 0 \wedge q(Z_1, f(X_3), Z_2)$

Now we apply the procedure **CM** to clauses γ' and δ'. The constraint $X_1 < 1 \wedge X_1 \geq Z_1 + 1 \wedge Z_2 > 0$ occurring in γ' is satisfiable. The procedure **CM** starts off by computing the constraint e. We get:

$$e = \mathit{project}(X_1 < 1 \wedge X_1 \geq Z_1 + 1 \wedge Z_2 > 0, \{X_1\}) = X_1 < 1$$

Then **CM** performs a sequence of rewritings which we list below, where: (i) all polynomials are bilinear in the partition $\langle \{V_1, \ldots, V_7\}, \{X_1, Y_1, Z_1, Z_2\} \rangle$, (ii) their normal forms are computed w.r.t. the variable ordering Z_1, Z_2, Y_1, X_1, and (iii) $\overset{\mathrm{r}}{\Longrightarrow}^k$ denotes k applications of rule r. (In the following sequence of rewritings we have underlined the constraints that are rewritten by the application of a rule. Note also that the atomic constraints occurring in the initial triple are the ones in γ' and δ', rewritten into the form $p > 0$ or $p \geq 0$.)

$$\langle (\underline{1-X_1>0} \wedge X_1-Z_1-1 \geq 0 \wedge Z_2>0) \leftrightarrow (\underline{1-X_1>0} \wedge -Z_1>0 \wedge Y_1-3-2Z_1 \geq 0 \wedge Z_2>0),\ \emptyset,\ \emptyset \rangle$$

$$\begin{aligned}
\overset{\mathrm{i}}{\Longrightarrow}\ & \langle (\underline{X_1-Z_1-1 \geq 0} \wedge Z_2>0) \leftrightarrow (-Z_1>0 \ \wedge\ \underline{Y_1-3-2Z_1 \geq 0} \wedge Z_2>0),\\
& \{(1-V_1)X_1+V_1-1=0, V_1>0\},\ \emptyset \rangle\\
\overset{\mathrm{i}}{\Longrightarrow}\ & \langle \underline{Z_2>0} \leftrightarrow (-Z_1>0 \wedge \underline{Z_2>0}),\\
& \{(1-V_1)X_1+V_1-1=0, V_1>0,\ (2-V_2)Z_1-Y_1+V_2X_1-V_2+3=0, V_2>0\},\ \emptyset \rangle\\
\overset{\mathrm{i}}{\Longrightarrow}\ & \langle \mathit{true} \leftrightarrow \underline{-Z_1>0},\\
& \{(1-V_1)X_1+V_1-1=0, V_1>0,\ (2-V_2)Z_1-Y_1+V_2X_1-V_2+3=0, V_2>0,\\
& \ (V_3-1)Z_2=0, V_3>0\},\ \emptyset \rangle\\
\overset{\mathrm{iii}}{\Longrightarrow}\ & \langle \mathit{true} \leftrightarrow \mathit{true},\\
& \{\underline{(1-V_1)X_1+V_1-1=0}, V_1>0,\ \underline{(2-V_2)Z_1-Y_1+V_2X_1-V_2+3=0}, V_2>0,\\
& \ \underline{(V_3-1)Z_2=0}, V_3>0,\ \underline{(1-V_5)Z_1+V_6Z_2+(V_5-V_4)X_1+V_4-V_5+V_7=0},\\
& \ V_4 \geq 0, V_5 \geq 0, V_6 \geq 0, V_7 \geq 0,\ V_4+V_6+V_7>0\},\ \emptyset \rangle\\
\overset{\mathrm{iv}}{\Longrightarrow}^6 & \langle \mathit{true} \leftrightarrow \mathit{true},\\
& \{1-V_1=0, V_1-1=0, V_1>0,\ 2-V_2=0, \underline{-Y_1+V_2X_1-V_2+3=0}, V_2>0,\\
& \ V_3-1=0, V_3>0,\ 1-V_5=0, V_6=0, V_5-V_4=0, V_4-V_5+V_7=0,\\
& \ V_4 \geq 0, V_5 \geq 0, V_6 \geq 0, V_7 \geq 0,\ V_4+V_6+V_7>0\},\ \emptyset \rangle\\
\overset{\mathrm{v}}{\Longrightarrow}\ & \langle \mathit{true} \leftrightarrow \mathit{true},\\
& \{1-V_1=0, V_1-1=0, V_1>0,\ 2-V_2=0, V_2>0, && (\dagger)\\
& \ V_3-1=0, V_3>0,\ 1-V_5=0, V_6=0, V_5-V_4=0, V_4-V_5+V_7=0, && (\dagger)\\
& \ V_4 \geq 0, V_5 \geq 0, V_6 \geq 0, V_7 \geq 0,\ V_4+V_6+V_7>0\}, && (\dagger)\\
& \{Y_1/V_2X_1-V_2+3\}\rangle && (\dagger\dagger)
\end{aligned}$$

Let C be the set of constraints occurring in the lines marked by $(\dagger)$. We have that C is satisfiable and has a unique solution given by the following substitution:

$$\sigma_W = \mathit{solve}(C) = \{V_1/1, V_2/2, V_3/1, V_4/1, V_5/1, V_6/0, V_7/0\}$$

The substitution σ_Y computed in the line marked by $(\dagger\dagger)$ is $\{Y_1/V_2X_1 - V_2 + 3\}$. Hence, the substitution φ_Y, which is defined as $\sigma_Y\sigma_W$ restricted to $\{Y_1\}$, is $\{Y_1/2X_1 + 1\}$. Since we have that $\mathit{Vars}_{\mathtt{rat}}(s(Y_1, a, f(X_3))\sigma_Y\sigma_W) - \mathit{Vars}(H) = \{X_1, X_3\} - \{X_1, X_2, X_3\} = \emptyset$, the substitution σ_G is the identity. Thus, the output of the procedure **CM** is the constraint $e = X_1 < 1$ and the substitution $\beta = \varphi_Y\sigma_G = \{Y_1/2X_1 + 1\}$.

4.3. The Folding Algorithm

Now we are ready to present our folding algorithm.

Folding Algorithm: FA

Input: two clauses in normal form without variables in common $\gamma\colon H \leftarrow c \wedge G$ and $\delta\colon K \leftarrow d \wedge B$.
Output: the clause $\eta\colon H \leftarrow e \wedge K\vartheta \wedge R$, if it is possible to fold γ using δ according to Definition 3.1, and **fail**, otherwise.

IF there exist a substitution α and a goal R which are the output of an execution of the procedure **GM** when clauses γ and δ are given as input to **GM**
AND there exist a constraint e and a substitution β which are the output of an execution of the procedure **CM** when clauses $\gamma'\colon H \leftarrow c \wedge B\alpha \wedge R$ and $\delta'\colon K\alpha \leftarrow d\alpha \wedge B\alpha$ are given as input to **CM**
THEN return the clause $\eta\colon H \leftarrow e \wedge K\alpha\beta \wedge R$ ELSE return **fail**.

The following theorem, whose proof is given in [18], states that (1) the folding algorithm **FA** terminates, (2) **FA** is sound, and, (3) if the constraint c is admissible, then **FA** is complete.

Theorem 4.2. (Termination, Soundness, and Completeness of FA)
Let the input of the algorithm **FA** be two clauses γ and δ in normal form without variables in common. Then: (1) **FA** terminates; (2) if **FA** returns a clause η, then η can be derived by folding γ using δ according to Definition 3.1; (3) if it is possible to fold γ using δ according to Definition 3.1 and the constraint occurring in γ is either unsatisfiable or admissible, then **FA** does not return **fail**.

Example 4.4. Let us consider the clause

$$\gamma\colon p(X_1, X_2, X_3) \leftarrow X_1 < 1 \wedge X_1 \geq Z_1 + 1 \wedge Z_2 > 0 \wedge q(Z_1, f(X_3), Z_2) \wedge r(X_2)$$

and the clause

$$\delta\colon s(Y_1, Y_2, Y_3) \leftarrow W_1 < 0 \wedge Y_1 - 3 \geq 2W_1 \wedge W_2 > 0 \wedge q(W_1, Y_3, W_2)$$

of the Introduction. Let the substitution $\alpha : \{W_1/Z_1,\ Y_3/f(X_3),\ W_2/Z_2,\ Y_2/a\}$ and the goal $R : r(X_2)$ be the result of applying the procedure **GM** to γ and δ as shown in Example 4.1, and let the constraint $e : X_1 < 1$ and the substitution $\beta : \{Y_1/2X_1 + 1\}$ be the result of applying the procedure **CM** to γ and $\delta\alpha$ as shown in Example 4.3. Then, the output of the folding algorithm **FA** is the clause $\eta : p(X_1, X_2, X_3) \leftarrow e \wedge s(Y_1, Y_2, Y_3)\alpha\beta \wedge R$, that is:

$$\eta : p(X_1, X_2, X_3) \leftarrow X_1 < 1 \wedge s(2X_1 + 1, a, f(X_3)) \wedge r(X_2).$$

5. Complexity of the Folding Algorithm and Experimental Results

For any clause γ, let $size(\gamma)$ denote be the number of occurrences of symbols in γ. A similar notation will also be used for constraints, terms, and sets of constraints or terms. We evaluate the time complexity of our folding algorithm **FA** w.r.t. $size(\gamma)+size(\delta)$, where γ and δ are the clauses given as input to **FA**. First we consider the complexity of the basic functions *nf*, *solve*, and *project*: (i) for any bilinear polynomial p, the computation of $nf(p)$ takes polynomial time w.r.t. $size(p)$, (ii) for any set C of constraints, the computation of $solve(C)$ takes polynomial time w.r.t. $size(C)$ by using Khachiyan's method [16], and

(iii) for any constraint c and set X of variables, the computation of $project(c, X)$ takes $2^{O(|X|)}$, where $|X|$ denotes the cardinality of X (see [22] for the complexity of variable elimination from linear constraints). We will see in the following analysis that, due to the time complexity of computing the *project* function, any nondeterministic execution of the folding algorithm in the worst case takes $2^{O(size(\gamma)+size(\delta))}$ time. Before making this analysis, let us observe that the function $project$ is applied to a subset X of the variables occurring in γ (in particular, with reference to the procedure **CM**, $X = Vars(c) \cap Vars(B')$) and it is often the case that $|X|$ is much smaller than $size(\gamma)+size(\delta)$. Thus, in order to analyze this particular case, we assume that the value of $|X|$ is fixed and the time complexity of the function *project* is a constant value. In this hypothesis our algorithm **FA** is in NP (w.r.t. $size(\gamma)+size(\delta)$). To show this result, now we prove that both the goal matching procedure **GM** and the constraint matching procedure **CM** are in NP.

First we consider the procedure **GM**. Let s be a sequence of applications of the rewrite rules (i)–(x) of **GM** starting from the initial set $\{(B \wedge T)/G\}$ of bindings, where B and G are the goals occurring in the body of δ and γ, respectively. First, we note that each application of one of the rules (i)–(ix) reduces at least by one the number of occurrences of symbols. Rule (x) can be applied at most M times, where M is the number of variables occurring in the head of clause δ. Thus, the length of the sequence s is linear in $size(\gamma)+size(\delta)$. Finally, by a single application of a rule, any set of bindings can be rewritten into at most K different new sets of bindings, where K is the number of occurrences of literals in G (see, in particular, rule (i) which is nondeterministic). Thus, **GM** is in NP w.r.t. $size(\gamma)+size(\delta)$.

Now we show that also **CM** is in NP. Let $\langle c \leftrightarrow e \wedge d',\ \emptyset,\ \emptyset\rangle$ be the initial triple and let N be $size(\{c, e \wedge d'\})$. We have the following property: for every maximal sequence s_1 of rewritings of the form $D \Longrightarrow \cdots \Longrightarrow E$ constructed by applications of the rewrite rules (i)–(v) of **CM**, there exists a sequence s_2 of the form $D \Longrightarrow \cdots \Longrightarrow E$ such that: (1) s_1 and s_2 have equal length, (2) in s_2 every application of rules (i), (ii), and (iii) occurs before all applications of rules (iv) and (v), and (3) rules (iv) and (v) are applied in the following order, starting from the triple of the form $\langle f_1 \leftrightarrow g_1, S_1, \sigma_1\rangle$ which is obtained after the applications of the rules (i), (ii), and (iii): (3.1) first, rule (iv) is applied as long as possible for eliminating all occurrences of the variables $Z_1, \ldots, Z_h$ from S_1, thereby deriving a new set S_2 of constraints, (3.2) then, rule (v) is applied as long as possible for eliminating all occurrences of the variables $Y_1, \ldots, Y_k$ from S_2, thereby deriving a new set S_3 of constraints, and (3.3) finally, rule (iv) is applied as long as possible for eliminating all occurrences of the variables $X_1, \ldots X_\ell$ from S_3, thereby deriving a set S_4 of constraints. Thus, S_4 is a set of constraints whose variables are all in W. Note that Conditions (3.1), (3.2), and (3.3) on the order of application of rules (iv) and (v) can be imposed because the normal forms of the bilinear polynomials occurring in the second component of every triple are computed w.r.t. the fixed variable ordering $Z_1, \ldots, Z_h, Y_1, \ldots, Y_k, X_1, \ldots X_\ell$.

Thus, for the time complexity analysis of **CM** we may restrict ourselves to sequences of rewritings constructed like the sequence s_2 above, that is, sequences which satisfy Conditions (2), (3.1), (3.2), and (3.3). First, note that each application of rules (i), (ii), and (iii) reduces the number of constraints occurring in the first component of the triple at hand. Hence, we may have at most N applications of the rules (i), (ii), and (iii). Moreover, each application of rules (i), (ii), and (iii) introduces at most $m+1$ new variables, where $m+1 \in O(N)$. Hence, during the applications of rules (i), (ii), and (iii), the number of new variables introduced is $O(N^2)$, that is, $|W| \in O(N^2)$. We also have that each application of rules (i), (ii), and (iii) adds at most $m+3$ constraints to the second component of the triple. Thus, after the application of rules (i), (ii), and (iii) we get a set S_1 of constraints such that $|S_1| \in O(N^2)$. Then, in the sequence s_2 rule (iv) is applied at most M_1 times, where M_1 is the number of occurrences in S_1

Example	$D0$	$D1$	$D2$	$D3$	$D4$	$N1$	$N2$	$N3$	$N4$
Number of Foldings	1	1	1	1	1	2	4	4	16
Number of Variables	10	4	8	12	16	4	8	12	16
Time (seconds)	0.01	0.01	0.08	3.03	306	0.02	0.08	0.23	1.09
Total-Time (seconds)	0.02	0.02	0.14	4.89	431	0.03	49	1016	11025

Table 1. Execution times of the folding algorithm **FA** for the examples $D0$, $D1$–$D4$, and $N1$–$N4$.

of variables in Z. Now, since all bilinear polynomials are in normal form, we have that $M_1 \leq |S_1| \times |Z|$ and $M_1 \in O(N^3)$. We also have that $|S_2|$ is equal to $|S_1| + z$, where z is the number of occurrences in S_1 of variables in Z. Since $|S_1| \in O(N^2)$, we get that $|S_2| \in O(N^2)$. Then, every application of rule (v) eliminates all occurrences of a variable in Y and, therefore, rule (v) is applied M_2 times with $M_2 = |Y| \leq N$. Note that after all applications of rule (v), we get the set S_3 of constraints whose cardinality is $|S_2| - |Y|$, and thus, $|S_3| \in O(N^2)$. Finally, rule (iv) is applied at most M_3 times, where M_3 is the number of occurrences in S_3 of variables in X. We have that $M_3 \leq |S_3| \times |X|$ and thus, $M_3 \in O(N^3)$. Therefore, the total number of applications of rules (i)–(v) in the sequence s_2 is $O(N^3)$. Since each rule application takes polynomial time w.r.t. N, we get a polynomial time cost of the **CM** procedure w.r.t. N. Now, in order to conclude that **CM** is in NP w.r.t. N we have to examine the nondeterminism of the **CM** procedure. We have that by a single application of a rule, any triple can be rewritten into at most $O(N^2)$ different new triples. Indeed, (1) by an application of rule (i), any triple can be rewritten into at most n different new triples, where n is the number of atomic constraints in $e \wedge d'$, and $n \leq N$, (2) rules (ii) and (iii) are deterministic, and (3) rules (iv) and (v) can be applied by selecting an equation in the second component of the triple at hand in at most $O(N^2)$ ways. Thus, **CM** is in NP w.r.t. N. Since $N \leq size(\gamma) + size(\delta)$, we get that **CM** is in NP w.r.t. $size(\gamma) + size(\delta)$.

Note that since matching modulo the equational theory AC$_\wedge$ is NP-complete [2], there is no folding algorithm whose asymptotic time complexity is significantly better than our algorithm **FA**, in the case when $|X|$ is fixed.

Finally, if we do *not* assume that $|X|$ is fixed, since $|X| < size(\gamma) + size(\delta)$ and $project(c, X)$ is computed (at the beginning of the **CM** procedure) at most once for each execution of the algorithm **FA**, we get that, as already mentioned, for any given pair of input clauses, each execution of **FA** takes $2^{O(size(\gamma)+size(\delta))}$ time.

In Table 1 we report some experimental results concerning our algorithm **FA**, implemented in SICStus Prolog 3.12, on a Pentium IV 3GHz. Each column of Table 1 refers to a particular example: column $D0$ refers to the example of the Introduction, columns $D1$–$D4$ refer to four examples for which folding can be done in one way only (*Number of Foldings* $= 1$), and four columns $N1$–$N4$ refer to four examples for which folding can be done in more than one way (*Number of Foldings* $=$ 2, or 4, or 16).

The row named *Number of Variables* indicates the number of variables occurring in clause γ (which is the clause to be folded) plus the number of variables occurring in clause δ (which is the clause used for folding). The row named *Time* shows the seconds required for finding the folded clause (or the first folded clause, in examples $N1$–$N4$, where more than one folding is possible). The row named *Total-Time* shows the seconds required for finding all folded clauses. Note that even when one folding only is possible, we have that *Total-Time* is greater than *Time* because, after the folded clause has been found, **FA** checks whether or not one more folded clause can be found.

In Example $D1$ clause γ is $p(A) \leftarrow A < 1 \wedge A \geq B+1 \wedge q(B)$ and clause δ is $r(C) \leftarrow D < 0 \wedge C-3 \geq 2D \wedge q(D)$. In Example $N1$ clause γ is $p \leftarrow A > 1 \wedge 3 > A \wedge B > 1 \wedge 3 > B \wedge q(A) \wedge q(B)$ and clause δ is $r \leftarrow C > 1 \wedge 3 > C \wedge D > 1 \wedge 3 > D \wedge q(C) \wedge q(D)$. In the other examples $D2$–$D4$ and $N2$–$N4$ we have considered clauses with more variables (and also more constraints and literals) according to the values shown in the row named *Number of Variables*.

From our experimental results we may conclude that the algorithm **FA** performs reasonably well in practice, but when the number of variables (and, in particular, the number of variables of type `rat`) increases, its performance rapidly deteriorates.

6. Related Work and Conclusions

The elimination of existential variables from logic programs and constraint logic programs is a program transformation technique which has been proposed for improving program performance [14] and for proving program properties [13]. This technique makes use of the definition, unfolding, and folding rules [3, 7, 8, 11, 20]. In this paper we have considered constraint logic programs, where the constraints are linear inequations over the rational (or real) numbers, and we have studied the problem of the automatic application of the folding rule. Indeed, the applicability conditions of the many folding rules for transforming constraint logic programs which have been proposed in the literature [3, 7, 8, 11, 13], are specified in a declarative way and no algorithm has been given to determine whether or not, given a clause γ to be folded by using a clause δ, one can actually perform that folding step. The problem of checking the applicability conditions of the folding rule is not trivial (see, for instance, the example presented in the Introduction).

In this paper we have considered a folding rule which is a variant of the rules proposed in the literature, and we have given an algorithm, called **FA**, for checking its applicability conditions. To the best of our knowledge, ours is the first algorithmic presentation of the folding rule. The applicability conditions of our rule consist of the usual conditions (see, for instance, [8]) together with the extra condition that, after folding, the existential variables should be eliminated. Thus, our algorithm **FA** is an important step forward for the full automation of the program transformation techniques [13, 14] for improving program efficiency or proving program properties by eliminating existential variables.

We have proved the termination and the soundness of our folding algorithm **FA**. We have also proved that if the constraint appearing in the clause γ to be folded is *admissible*, then **FA** is complete, that is, it does not return **fail** whenever folding is possible. Finally, we have implemented the folding algorithm and our experimental results show that it performs reasonably well in practice.

Our algorithm **FA** consists of two procedures: (i) the *goal matching* procedure, and (ii) the *constraint matching* procedure. The *goal matching* procedure solves a problem which is similar to the problem of matching two terms modulo an associative, commutative equational theory, also called *AC theory* [2]. However, in our case we have the extra conditions that: (i.1) the matching substitution should be consistent with the types (either rational numbers or trees), and (i.2) after folding, the existential variables should be eliminated. Thus, we could not directly use the AC-matching algorithms available in the literature [6].

The *constraint matching* procedure solves a generalized form of the matching problem, modulo the equational theory, called $LIN_{\mathbb{Q}}$, of linear inequations over the rational numbers. That problem can be seen as a *restricted unification* problem [4]. In [4] it is described how to obtain, if certain conditions hold, an algorithm for solving a restricted unification problem from an algorithm that solves the corresponding unrestricted unification problem. To the best of our knowledge, for the theory $LIN_{\mathbb{Q}}$ of constraints an algorithm is provided neither for the restricted unification problem nor for the unrestricted one. Moreover, one cannot apply the so called *combination methods* [15]. These methods consist in constructing a matching algorithm for a given theory which is the combination of simpler theories, starting from the matching algorithms for those simpler theories. Unfortunately, as we said, we cannot use these combination methods for the theory $LIN_{\mathbb{Q}}$ because some applicability conditions are not satisfied and, in particular, $LIN_{\mathbb{Q}}$ is neither *collapse-free* nor *regular* [15].

In the future we plan to adapt our folding algorithm **FA** to other constraint domains such as the linear inequations over the integers. We will also perform a more extensive experimentation of our folding algorithm using the MAP program transformation system for constraint logic programs [12].

Acknowledgements

We thank the anonymous referees for helpful suggestions. We also thank John Gallagher for comments on a draft of this paper.

References

[1] Baader, F., Snyder, W.: Unification Theory, in: *Handbook of Automated Reasoning* (A. Robinson, A. Voronkov, Eds.), vol. I, Elsevier Science, 2001, 445–532.

[2] Benanav, D., Kapur, D., Narendran, P.: Complexity of matching problems, *Journal of Symbolic Computation*, **3**(1-2), 1987, 203–216.

[3] Bensaou, N., Guessarian, I.: Transforming Constraint Logic Programs, *Theoretical Computer Science*, **206**, 1998, 81–125.

[4] Bürckert, H.-J.: Some Relationships between Unification, Restricted Unification, and Matching, *Proceedings of the 8th International Conference on Automated Deduction*, 230, Springer-Verlag, London, UK, 1986.

[5] Burstall, R. M., Darlington, J.: A Transformation System for Developing Recursive Programs, *Journal of the ACM*, **24**(1), January 1977, 44–67.

[6] Eker, S. M.: Improving the efficiency of AC matching and unification, RR-2104, INRIA Lorraine & CRIN, Villers-les-Nancy, France, 1993.

[7] Etalle, S., Gabbrielli, M.: Transformations of CLP Modules, *Theoretical Computer Science*, **166**, 1996, 101–146.

[8] Fioravanti, F., Pettorossi, A., Proietti, M.: Transformation Rules for Locally Stratified Constraint Logic Programs, *Program Development in Computational Logic* (K.-K. Lau, M. Bruynooghe, Eds.), Lecture Notes in Computer Science 3049, Springer, 2004.

[9] Jaffar, J., Maher, M.: Constraint Logic Programming: A Survey, *Journal of Logic Programming*, **19/20**, 1994, 503–581.

[10] Lloyd, J. W.: *Foundations of Logic Programming*, Springer-Verlag, Berlin, 1987, Second Edition.

[11] Maher, M. J.: A Transformation System for Deductive Database Modules with Perfect Model Semantics, *Theoretical Computer Science*, **110**, 1993, 377–403.

[12] The MAP Transformation System, 1995–2008, Available from http://www.iasi.cnr.it/∼proietti/system.html.

[13] Pettorossi, A., Proietti, M., Senni, V.: Proving Properties of Constraint Logic Programs by Eliminating Existential Variables, in: *Proceedings of the 22nd International Conference on Logic Programming, ICLP'06* (S. Etalle, M. Truszczyński, Eds.), Lecture Notes in Computer Science 4079, Springer, 2006, 179–195.

[14] Proietti, M., Pettorossi, A.: Unfolding-Definition-Folding, in this Order, for Avoiding Unnecessary Variables in Logic Programs, *Theoretical Computer Science*, **142**(1), 1995, 89–124.

[15] Ringeissen, C.: Matching in a Class of Combined Non-disjoint Theories, *Proceedings of the 19th International Conference on Automated Deduction, CADE-19* (F. Baader, Ed.), Lecture Notes in Computer Science 2741, Springer, 2003.

[16] Schrijver, A.: *Theory of Linear and Integer Programming,* John Wiley & Sons, 1986.

[17] Senni, V., Pettorossi, A., Proietti, M.: A Folding Algorithm for Eliminating Existential Variables from Constraint Logic Programs, in: *Proceedings of the 24th International Conference on Logic Programming, ICLP'08* (M. Garcia de la Banda, E. Pontelli, Eds.), Lecture Notes in Computer Science 5366, Springer, 2008, 284–300.

[18] V. Senni, A. Pettorossi, and M. Proietti. A folding rule for eliminating existential variables from constraint logic programs. Technical Report 08-03, IASI-CNR, Rome, Italy, 2008. Available from: `http://www.iasi.cnr.it/~proietti/reports.html`.

[19] H. Seki. Unfold/fold transformation of stratified programs. *Theoretical Computer Science*, 86:107–139, 1991.

[20] Tamaki, H., Sato, T.: Unfold/Fold Transformation of Logic Programs, *Proceedings of the Second International Conference on Logic Programming, ICLP'84* (S.-Å. Tärnlund, Ed.), Uppsala University, Uppsala, Sweden, 1984.

[21] Terese: *Term Rewriting Systems,* Cambridge University Press, 2003.

[22] Weispfenning, V.: The complexity of linear problems in fields, *Journal of Symbolic Computation*, **5**(1-2), 1988, 3–27.