A Folding Algorithm for Eliminating Existential Variables from Constraint Logic Programs

Valerio Senni[1], Alberto Pettorossi[1], and Maurizio Proietti[2]

[1] DISP, University of Rome Tor Vergata, Via del Politecnico 1, I-00133 Rome, Italy
{senni,pettorossi}@disp.uniroma2.it
[2] IASI-CNR, Viale Manzoni 30, I-00185 Rome, Italy
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. We show that an implementation of our folding algorithm performs well in practice.

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 [1]. 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. For instance, the formula $\forall N\,(N>0 \rightarrow p(N))$, specifying "the predicate $p(N)$ holds for every positive number N", can be written by using the following two clauses:

$prop \leftarrow \neg q \qquad\qquad q \leftarrow N > 0 \wedge \neg p(N)$

where N is an existential variable. 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 [2,3]. In particular, in [3] it has been shown that by eliminating the existential variables from a constraint logic program defining a nullary predicate, like *prop* above, one may obtain a propositional program and, thus, decide whether or not that predicate holds.

The transformation techniques for the elimination of the existential variables make use of the *unfolding* and *folding* rules which have been first proposed in the

context of functional programming by Burstall and Darlington [4], and then extended to logic programming [5,6] and to constraint logic programming [7,8,9,10]. In the techniques for eliminating existential variables a particularly relevant role is played by the folding rule, which 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 the variables occurring in $K\vartheta$ are a subset of the variables occurring 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. 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 (however, the techniques we will present are valid without relevant changes also when the inequations are over the real numbers).

For instance, let us consider the 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 1–3 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 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 can easily be checked by unfolding η w.r.t. $s(2X_1+1, a, f(X_3))$.) In general, there may be zero or more triples $\langle e, \vartheta, R\rangle$ that satisfy the conditions for the applicability of the folding rule. For this reason, our folding algorithm is nondeterministic and in different runs 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 present an experimental evaluation of its performance. Finally, in Section 6 we discuss 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 ra-

tional numbers. As already mentioned, the results we will present in this paper are valid also when the constraints are conjunctions of linear inequations over the real numbers. For notions not defined here the reader may refer to [1,11].

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 $\{+, \cdot\} \cup \mathbb{Q} \subseteq \mathit{Fun}$ (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, in general, may be considered as finite trees), we assume that $\mathcal{L}$ is a typed language [11] with two basic types: `rat`, which is the type of rational numbers, and `tree`, which is the type of finite trees. We also consider types constructed from basic types by using the type constructors $\times$ and $\rightarrow$. A variable $X \in \mathit{Var}$ has either type `rat` or type `tree`. We denote by $\mathit{Var}_{\mathtt{rat}}$ and $\mathit{Var}_{\mathtt{tree}}$ 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 $\tau_1, \ldots, \tau_n, \tau_{n+1} \in \{\mathtt{rat}, \mathtt{tree}\}$. In particular, the predicate symbols $\geq$ and $>$ have type $\mathtt{rat} \times \mathtt{rat}$, the function symbols $+$ and $\cdot$ have type $\mathtt{rat} \times \mathtt{rat} \rightarrow \mathtt{rat}$, and the rational numbers have type `rat`. The function symbols in $\{+, \cdot\} \cup \mathbb{Q}$ are the only symbols whose type is $\tau_1 \times \cdots \times \tau_n \rightarrow \mathtt{rat}$, 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 $a_1 X_1 + \ldots + a_n X_n + a_{n+1}$, where $a_1, \ldots, a_{n+1}$ are rational numbers and $X_1, \ldots, X_n$ are variables in $\mathit{Var}_{\mathtt{rat}}$ (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 $\mathit{Var}_{\mathtt{tree}}$ or a term of the form $f(u_1, \ldots, u_n)$, where f is a function symbol of type $\tau_1 \times \cdots \times \tau_n \rightarrow \mathtt{tree}$, 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). We denote by $\mathit{LIN}_{\mathbb{Q}}$ the set of all constraints.

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 $n \geq 0$. Similarly to the case of constraints, 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 e is a term or a formula then by $Vars_{\mathtt{rat}}(e)$ and $Vars_{\mathtt{tree}}(e)$ we denote, respectively, the set of variables of type `rat` and of type `tree` occurring in e. By $Vars(e)$ we denote the set of all variables occurring in e, that is, $Vars_{\mathtt{rat}}(e) \cup Vars_{\mathtt{tree}}(e)$. Similar notation will also be used for sets of terms or sets of formulas. Given a clause γ: $H \leftarrow c \wedge G$, by $EVars(\gamma)$ we denote the set of the *existential variables* of γ, that is, $Vars(c \wedge G) - Vars(H)$. The *constraint-local* variables of γ are the variables in the set $Vars(c) - Vars(\{H, G\})$. 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 [11] with the following extra condition: for 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 $\mathcal{L}_{\mathtt{rat}}$ denote the sublanguage of $\mathcal{L}$ given by the set $Var_{\mathtt{rat}}$ of variables, the set $\{+, \cdot\} \cup \mathbb{Q}$ of function symbols, and the set $\{\geq, >\}$ of predicate symbols. We denote by $\mathcal{Q}$ the interpretation which assigns to every function symbol or predicate symbol of $\mathcal{L}_{\mathtt{rat}}$ the usual function or relation on $\mathbb{Q}$. For a formula φ of $\mathcal{L}_{\mathtt{rat}}$ (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 $\mathcal{L}_{\mathtt{rat}}$, that is, for each φ of $\mathcal{L}_{\mathtt{rat}}$, $I \models \varphi$ 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 [1]. 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 we have that every such program P has a *perfect* $\mathcal{Q}$-model [1,7,10], 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 $\{X_1, \ldots, X_n\} = Vars(C)$ 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 a set C of constraints in $LIN_{\mathbb{Q}}$ as input 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 or the Khachiyan algorithms [12]. We assume that we are also given a function *project* such that for every constraint $c \in LIN_{\mathbb{Q}}$ and for every finite set of variables $X \subseteq Var_{\mathtt{rat}}$, $\mathcal{Q} \models \forall X\, ((\exists Y\, c) \leftrightarrow project(c, X))$, where $Y = Vars(c) - X$ and $Vars(project(c, X)) \subseteq X$. The *project* function can be implemented, for instance, by using the Fourier-Motzkin variable elimination algorithm or the algorithm presented in [13].

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) $Vars_{\mathtt{rat}}(H) \cap Vars_{\mathtt{rat}}(G) = \emptyset$, and (iv) γ has no constraint-local variables. It is always possible to transform any clause γ_1 into a clause γ_2 in normal form such that γ_1 and γ_2 have the same $\mathcal{Q}$-models. (In particular, the constraint-local variables of any given clause can be eliminated by applying the *project* function.) The clause γ_2 is called *a normal form of* γ_1. Without loss of

generality, when presenting the folding rule and the corresponding algorithm for its application, we will assume that the clauses are in normal form.

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 variable renaming 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 refer to this theory as the $AC_\wedge$ *theory* [14].

Proposition 1. (i) $\cong$ *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 rules considered in the literature [7,8,9,10]. 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 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 $EVars(\delta)$, 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) $Vars(K\vartheta) \subseteq Vars(H)$. 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 clause δ again or other clauses. In Theorem 1 below we establish the correctness of the folding rule w.r.t. the perfect model semantics. That correctness follows immediately from [6].

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*, *folding*. For a detailed presentation of the definition and unfolding rules we refer to [10]. 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 1.

Theorem 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* $\delta \in Defs_k$, *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* $P_0 \cup Defs_n$ *and* P_n *are stratified and* $M(P_0 \cup Defs_n) = M(P_n)$.

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 1. The objective of our folding algorithm is to find a constraint e, a substitution ϑ, and a goal R such that $\gamma \cong H \leftarrow e \wedge d\vartheta \wedge B\vartheta \wedge R$ holds (see Point (1) of Definition 1), and also Points (2) and (3) of Definition 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 new 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 $\alpha\,\beta$, that is, the composition of the substitutions α and β. The output of the folding algorithm is either the clause $\eta\colon H \leftarrow e \wedge K\vartheta \wedge R$, or **fail** if folding is not possible. Since Definition 1 does not determine e, ϑ, and R in a unique way, our folding algorithm is nondeterministic and, as already said, in different runs it may compute different output 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, 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 $EVars(\delta)$, the following conditions hold: (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) $Vars_{\mathtt{tree}}(K\alpha) \subseteq Vars(H)$. If such α and R do not exist, then **fail**.

Consider a set $Bnds$ of bindings initialized to the singleton $\{B/G\}$. Consider also the following rewrite rules (i)–(x). When the left hand side of a rule is written as $Bnds_1 \cup Bnds_2 \implies \ldots$ then we assume that $Bnds_1 \cap Bnds_2 = \emptyset$.

(i) $\{(L_1 \wedge B_1)\,/\,(G_1 \wedge L_2 \wedge G_2)\} \cup Bnds \implies \{L_1/L_2,\ B_1/(G_1 \wedge G_2)\} \cup Bnds$ where: (1) L_1 and L_2 are 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 goals (possibly, the empty conjunction *true*);

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

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

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

(v) $\{a(s_1, \ldots, s_n)/X\} \cup Bnds \implies$ **fail**, if X is a variable;

(vi) $\{X/s\} \cup Bnds \implies$ **fail**, if X is a variable 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) $\{X/s,\ true/G_2\} \cup Bnds \Longrightarrow$ **fail**, if $X \in EVars(\delta)$ and $s \in Vars(G_2)$;

(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 ground term of type `tree`, 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) $\{B/G\} \Longrightarrow^* \{true/R\} \cup \alpha$ (where $true/R \notin \alpha$), (c2) no α' exists such that $\alpha \Longrightarrow \alpha'$, and (c3) α is different from **fail** (that is, α is a maximally rewritten, non-failing set of bindings such that (c1) holds) 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 [15]. 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 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`×`tree`×`tree`, `rat`×`tree`×`rat`, `tree`, and `rat`×`tree`×`tree`, respectively, and the function f is of type `tree`→`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:

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

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 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 in the head of clause δ.

4.2 Constraint Matching

Given two clauses in normal form γ: $H \leftarrow c \wedge G$ and δ: $K \leftarrow d \wedge B$, if the goal matching procedure **GM** returns the substitution α and the goal R, then we can construct 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 in input these two clauses, which, for reasons of simplicity, we now rename as γ': $H \leftarrow c \wedge B' \wedge R$ and δ': $K' \leftarrow d' \wedge B'$, respectively, and returns 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) $Vars(K'\beta) \subseteq Vars(H)$, and (4) $Vars(e) \subseteq Vars(\{H, R\})$. If such e and β do not exist, then the procedure **CM** returns **fail**.

Now, let $\widetilde{e}$ denote the constraint $project(c, X)$, where $X = Vars(c) - Vars(B')$ (see Section 2 for the definition of the *project* function). Lemma 1 below shows that, for any substitution β, if there exists a constraint e satisfying Conditions (1)–(4) above, then we can always take e to be the constraint $\widetilde{e}$. Thus, by Lemma 1 the procedure **CM** should only search for a substitution β such that $\mathcal{Q} \models \forall(c \leftrightarrow (\widetilde{e} \wedge d'\beta))$.

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

Now we introduce some notions and we state some properties (see Lemma 2 and Theorem 2) which will be exploited by the constraint matching procedure **CM** for reducing the equivalence between c and $\widetilde{e} \wedge d'\beta$, for a suitable β, to a set of equivalences between the atomic constraints occurring in c and $\widetilde{e} \wedge d'\beta$.

A conjunction $a_1 \wedge \ldots \wedge a_m$ of (not necessarily distinct) atomic constraints is said to be *redundant* if there exists i, with $0 \leq i \leq m$, such that $\mathcal{Q} \models \forall((a_1 \wedge \ldots \wedge a_{i-1} \wedge a_{i+1} \wedge \ldots \wedge a_m) \rightarrow a_i)$. In this case we also 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 can assume that any given constraint c is of the form $p_1\, \rho_1\, 0 \wedge \ldots \wedge p_m\, \rho_m\, 0$, where $m \geq 0$ and $\rho_1, \ldots, \rho_m \in \{\geq, >\}$. We define the *interior* of c, denoted $interior(c)$, 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 $interior(c)$ 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 and $interior(c_2) : X - Y > 0 \wedge Y > 0 \wedge X > 0$ is redundant). The following Lemma 2 characterizes the equivalence of two constraints when one of them is admissible.

Lemma 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 order to see that admissibility is a needed hypothesis for Lemma 2, let us consider the non-admissible 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 which has the properties stated in Lemma 2.

Lemma 2 will be used to show that if there exists a substitution β such that $\mathcal{Q} \models \forall(c \leftrightarrow (\widetilde{e} \wedge d'\beta))$, where c is an admissible constraint and $\widetilde{e}$ is defined as in Lemma 1, then **CM** computes such a substitution β. Indeed, given the constraint c, of the form $a_1 \wedge \ldots \wedge a_m$, and the constraint $\widetilde{e} \wedge d'$, of the form $b_1 \wedge \ldots \wedge b_n$, **CM** computes: (1) an injection μ from $\{1, \ldots, m\}$ to $\{1, \ldots, n\}$, and (2) a substitution 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 $\mathcal{Q} \models \forall(c \rightarrow b_j\beta)$.

In order to compute β satisfying the property of Point (2.i), we make use of the following Property $P1$: given the satisfiable, non-redundant 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 replace $p > 0$ and $q > 0$ by $p \geq 0$ and $q \geq 0$, respectively.

Finally, in order to compute β satisfying the property of Point (2.ii), we make use of the following Theorem 2 which is a generalization of the above Property $P1$ and it is an extension of Farkas' Lemma to the case of systems of weak and strict inequalities [12].

Theorem 2. *Suppose that* $p_1\, \rho_1\, 0, \ldots, p_m\, \rho_m\, 0$, $p_{m+1}\, \rho_{m+1}\, 0$ *are atomic constraints such that, for* $i = 1, \ldots, m+1$, $\rho_i \in \{\geq, >\}$ *and* $\mathcal{Q} \models \exists(p_1\, \rho_1\, 0 \wedge \ldots \wedge p_m\, \rho_m\, 0)$. *Then* $\mathcal{Q} \models \forall(p_1\, \rho_1\, 0 \wedge \ldots \wedge p_m\, \rho_m\, 0 \rightarrow p_{m+1}\, \rho_{m+1}\, 0)$ *iff there exist* $k_1 \geq 0, \ldots, k_{m+1} \geq 0$ *such that: (i)* $\mathcal{Q} \models \forall\,(k_1p_1 + \cdots + k_mp_m + k_{m+1} = p_{m+1})$, *and (ii) if* ρ_{m+1} *is* $>$ *then* $(\sum_{i \in I} k_i) > 0$, *where* $I = \{i \mid 1 \leq i \leq m+1,\ \rho_i \text{ is } >\}$.

As we will see below, the constraint matching procedure **CM** may generate *bilinear* polynomials (see rules (i)–(iii)), that is, non-linear polynomials of a particular form, which we now define. Let p be a polynomial and $\langle P_1, P_2\rangle$ be a partition of a (proper or not) superset of $Vars(p)$. The polynomial p is said to be *bilinear in the partition* $\langle P_1, P_2\rangle$ if the monomials of p are of the form: *either* (i) $k\,XY$, where k is a rational number, $X \in P_1$, and $Y \in P_2$, *or* (ii) $k\,X$, where k is a rational number and X is a variable, *or* (iii) k, where k is a rational number. Let us consider a polynomial p which is bilinear in the partition $\langle P_1, P_2\rangle$ where $P_2 = \{Y_1, \ldots, Y_m\}$. The *normal form* of p, denoted $nf(p)$, *w.r.t. a given ordering* $Y_1, \ldots, Y_m$ *of the variables in* P_2, is a bilinear polynomial which is derived by: (i) computing the bilinear polynomial $p_1Y_1 + \cdots + p_mY_m + p_{m+1}$ such that $\mathcal{Q} \models \forall\,(p_1Y_1 + \cdots + p_mY_m + p_{m+1} = p)$, and (ii) erasing from that bilinear polynomial every summand p_iY_i such that $\mathcal{Q} \models \forall\,(p_i = 0)$.

Constraint Matching Procedure: CM

Input: two clauses in normal form $\gamma' \colon H \leftarrow c \wedge B' \wedge R$ and $\delta' \colon K' \leftarrow d' \wedge B'$. *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) $Vars(K'\beta) \subseteq Vars(H)$, and (4) $Vars(e) \subseteq Vars(\{H, R\})$. If such e and β do not exist, then **fail**.

IF c is unsatisfiable THEN return an arbitrary ground, unsatisfiable constraint e and a substitution β of the form $\{U_1/a_1, \ldots, U_s/a_s\}$, where $\{U_1, \ldots,$

$U_s\} = \mathit{Vars}_{\mathtt{rat}}(K')$ and $a_1, \ldots, a_s$ are arbitrary rational numbers ELSE, if c is satisfiable, we proceed as follows.
Let X be the set $\mathit{Vars}(c) - \mathit{Vars}(B')$, Y be the set $\mathit{Vars}(d') - \mathit{Vars}(B')$, and Z be the set $\mathit{Vars}_{\mathtt{rat}}(B')$. Let e be the constraint $\mathit{project}(c, X)$. Without loss of generality, we may assume that: (i) c is a constraint of the form $p_1\,\rho_1\,0 \wedge \ldots \wedge p_m\,\rho_m\,0$, where for $i = 1, \ldots, m$, p_i is a linear polynomial and $\rho_i \in \{\geq, >\}$, and (ii) $e \wedge d'$ is a constraint of the form $q_1\,\pi_1\,0 \wedge \ldots \wedge q_n\,\pi_n\,0$, where for $j = 1, \ldots, n$, q_i is a linear polynomial and $\pi_i \in \{\geq, >\}$.

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) f_1, g_1, f_2, and g_2 are constraints, (2) S_1 and S_2 are sets of constraints, and (3) σ_1 and σ_2 are substitutions. In the rewrite rules (i)–(v) below, whenever S_1 is written as $A \cup B$, we assume that $A \cap B = \emptyset$.

(i) $\langle p\,\rho\,0 \wedge f \leftrightarrow g_1 \wedge q\,\rho\,0 \wedge g_2,\ S,\ \sigma\rangle \Longrightarrow$
$\langle f \leftrightarrow g_1 \wedge g_2,\ \{\mathit{nf}(Vp - q) = 0, V > 0\} \cup S,\ \sigma\rangle$
where V is a new variable and $\rho \in \{\geq, >\}$;

(ii) $\langle \mathit{true} \leftrightarrow q \geq 0 \wedge g,\ S,\ \sigma\rangle \Longrightarrow$
$\langle \mathit{true} \leftrightarrow g,\ \{\mathit{nf}(V_1 p_1 + \ldots + V_m p_m + V_{m+1} - q) = 0,$
$V_1 \geq 0, \ldots, V_{m+1} \geq 0\} \cup S,\ \sigma\rangle$
where $V_1, \ldots, V_{m+1}$ are new variables;

(iii) $\langle \mathit{true} \leftrightarrow q > 0 \wedge g,\ S,\ \sigma\rangle \Longrightarrow$
$\langle \mathit{true} \leftrightarrow g,\ \{\mathit{nf}(V_1 p_1 + \ldots + V_m p_m + V_{m+1} - q) = 0,$
$V_1 \geq 0, \ldots, V_{m+1} \geq 0,\ (\sum_{i \in I} V_i) > 0\} \cup S,\ \sigma\rangle$
where $V_1, \ldots, V_{m+1}$ are new variables and $I = \{i \mid 1 \leq i \leq m+1,\ \rho_i \text{ is } >\}$;

(iv) $\langle f \leftrightarrow g,\ \{p\,U + q = 0\} \cup S,\ \sigma\rangle \Longrightarrow \langle f \leftrightarrow g,\ \{p = 0, q = 0\} \cup S,\ \sigma\rangle$
if $U \in X \cup Z$;

(v) $\langle f \leftrightarrow g,\ \{a\,U + q = 0\} \cup S,\ \sigma\rangle \Longrightarrow$
$\langle f \leftrightarrow (g\{U/-\frac{q}{a}\}),\ \{\mathit{nf}(p\{U/-\frac{q}{a}\})\rho\,0 \mid p\,\rho\,0 \in S\},\ \sigma\{U/-\frac{q}{a}\}\rangle$
if $U \in Y$, $\mathit{Vars}(q) \cap \mathit{Vars}(R) = \emptyset$, and $a \in (\mathbb{Q} - \{0\})$;

IF there exist a set C of constraints and a substitution σ_Y such that: (c1) $\langle c \leftrightarrow e \wedge d',\ \emptyset,\ \emptyset\rangle \Longrightarrow^* \langle \mathit{true} \leftrightarrow \mathit{true},\ C,\ \sigma_Y\rangle$, (c2) there is no triple T such that $\langle \mathit{true} \leftrightarrow \mathit{true},\ C,\ \sigma_Y\rangle \Longrightarrow T$, (c3) for every constraint $f \in C$, we have that $\mathit{Vars}(f) \subseteq W$, 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 $\langle \mathit{true} \leftrightarrow \mathit{true},\ C,\ \sigma_Y\rangle$, and (c4) C is satisfiable and $\mathit{solve}(C) = \sigma_W$,
THEN construct a substitution σ_G of the form $\{U_1/a_1, \ldots,\ U_s/a_s\}$, where $\{U_1, \ldots, U_s\} = \mathit{Vars}_{\mathtt{rat}}(K'\sigma_Y\sigma_W) - \mathit{Vars}(H)$ and $a_1, \ldots, a_s$ are arbitrary rational numbers, and return the constraint e and the substitution $\beta = \sigma_Y\sigma_W\sigma_G$
ELSE return **fail**.

Note that in order to apply rules (iv) and (v), $p\,U$ and $a\,U$, respectively, should be the leftmost monomials. The procedure **CM** is nondeterministic (see rule (i)). By induction on the number of rule applications, we can show that the polynomials occurring in the second components of the triples 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. The normal forms of the bilinear polynomials which occur in the rewrite rules are all computed w.r.t. the fixed variable ordering $Z_1, \ldots, Z_h$, $Y_1, \ldots, Y_k$, $X_1, \ldots, X_l$, where $\{Z_1, \ldots, Z_h\} = Z$, $\{Y_1, \ldots, Y_k\} = Y$, and $\{X_1, \ldots, X_l\} = X$.

The termination of the procedure **CM** is a consequence of the following facts: (1) each application of rules (i), (ii), and (iii) reduces the number of atomic constraints occurring in the first component of 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 at hand, does not introduce any new variables, and replaces an equation occurring in the second component of the triple at hand by two smaller equations; (3) each application of rule (v) does not modify the number of atomic constraints in the first component of the triple at hand and eliminates all occurrences of a variable. Thus, the termination of **CM** can be proved by a lexicographic combination of two linear orderings and a multiset ordering.

Example 2. 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 1. Let us also consider the clauses γ' and δ', where γ' is γ and δ' is $\delta\alpha$, that is,

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

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 as follows:

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

Now **CM** performs the following rewritings, where: (i) all polynomials are bilinear in $\langle \{V_1, \ldots, V_7\}, \{X_1, Y_1, Z_1, Z_2\} \rangle$, (ii) their normal forms are computed w.r.t. the variable ordering Z_1, Z_2, Y_1, X_1, and (iii) $\stackrel{r}{\Longrightarrow}{}^k$ denotes k applications of rule r. (We have underlined the constraints that are rewritten by an 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 $p \geq 0$.)

$$
\begin{array}{l}
\langle (\underline{1 - X_1 > 0} \wedge X_1 - Z_1 - 1 \geq 0 \wedge Z_2 > 0) \leftrightarrow \\
\quad (\underline{1 - X_1 > 0} \wedge -Z_1 > 0 \wedge Y_1 - 3 - 2Z_1 \geq 0 \wedge Z_2 > 0),\ \ \emptyset,\ \ \emptyset \rangle \\
\stackrel{\text{i}}{\Longrightarrow} \langle (\underline{X_1 - Z_1 - 1 \geq 0} \wedge Z_2 > 0) \leftrightarrow (-Z_1 > 0 \wedge \underline{Y_1 - 3 - 2Z_1 \geq 0} \wedge Z_2 > 0), \\
\quad \{(1 - V_1)X_1 + V_1 - 1 = 0, V_1 > 0\},\ \ \emptyset \rangle \\
\stackrel{\text{i}}{\Longrightarrow} \langle \underline{Z_2 > 0} \leftrightarrow (-Z_1 > 0 \wedge \underline{Z_2 > 0}), \\
\quad \{(1 - V_1)X_1 + V_1 - 1 = 0, V_1 > 0,\ (2 - V_2)Z_1 - Y_1 + V_2X_1 - V_2 + 3 = 0, V_2 > 0\},\ \emptyset \rangle \\
\stackrel{\text{i}}{\Longrightarrow} \langle \mathit{true} \leftrightarrow \underline{-Z_1 > 0}, \\
\quad \{(1 - V_1)\underline{X_1 + V_1} - 1 = 0, V_1 > 0,\ (2 - V_2)Z_1 - Y_1 + V_2X_1 - V_2 + 3 = 0, V_2 > 0, \\
\quad\ (V_3 - 1)Z_2 = 0, V_3 > 0\},\ \ \emptyset \rangle \\
\stackrel{\text{iii}}{\Longrightarrow} \langle \mathit{true} \leftrightarrow \mathit{true}, \\
\quad \{\underline{(1 - V_1)X_1 + V_1 - 1 = 0}, V_1 > 0,\ \underline{(2 - V_2)Z_1 - Y_1 + V_2X_1 - V_2 + 3 = 0}, V_2 > 0, \\
\quad\ \underline{(V_3 - 1)Z_2 = 0}, V_3 > 0,\ \underline{(1 - V_5)Z_1 + V_6Z_2 + (V_5 - V_4)X_1 + V_4 - V_5 + V_7 = 0}, \\
\quad\ \underline{V_4 \geq 0, V_5 \geq 0, V_6 \geq 0, V_7 \geq 0},\ V_4 + V_6 + V_7 > 0\},\ \ \emptyset \rangle
\end{array}
$$

$\stackrel{\text{iv}}{\Longrightarrow}{}^{6}\langle true \leftrightarrow true,$
$\{1-V_1=0, V_1-1=0, V_1>0,\ 2-V_2=0, \underline{-Y_1+V_2X_1-V_2+3=0}, V_2>0,$
$V_3-1=0, V_3>0,\ 1-V_5=0, V_6=0, V_5-V_4=0, V_4-V_5+V_7=0,$
$V_4\geq 0, V_5\geq 0, V_6\geq 0, V_7\geq 0,\ V_4+V_6+V_7>0\},\quad \emptyset\rangle$
$\stackrel{\text{v}}{\Longrightarrow}\langle true \leftrightarrow true,$
$\{1-V_1=0, V_1-1=0, V_1>0,\ 2-V_2=0, V_2>0,$
$V_3-1=0, V_3>0,\ 1-V_5=0, V_6=0, V_5-V_4=0, V_7-V_5+V_4=0,$
$V_4\geq 0, V_5\geq 0, V_6\geq 0, V_7\geq 0,\ V_4+V_6+V_7>0\},\quad \{Y_1/V_2X_1-V_2+3\}\rangle$

Let C be the second component of the final triple of the above sequence of rewritings. We have that C is satisfiable and has a unique solution given by the following substitution: $\sigma_W = solve(C) = \{V_1/1, V_2/2, V_3/1, V_4/1, V_5/1, V_6/0, V_7/0\}$. The substitution σ_Y computed in the third component of the final triple of the above sequence of rewritings is $\{Y_1/V_2X_1-V_2+3\}$. Since $Vars_{\mathtt{rat}}(s(Y_1, a, f(X_3))\sigma_Y\sigma_W) - Vars(H) = \{X_1, X_3\} - \{X_1, X_2, X_3\} = \emptyset$, we have that σ_G is the identity substitution. Thus, the output of the procedure **CM** is the constraint $e = X_1 < 1$ and the substitution $\beta = \sigma_Y\sigma_W\sigma_G = \{Y_1/2X_1+1\} \cup \sigma_W$.

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

The following theorem states that the folding algorithm **FA** terminates (Point 1), it is sound (Point 2), and, if the constraint c is admissible, then **FA** is complete (Point 3). The proof of this result can be found in [16].

Theorem 3 (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 1;* (3) *if it is possible to fold γ using δ according to Definition 1 and the constraint occurring in γ is either unsatisfiable or admissible, then* **FA** *does not return* **fail**.

Example 3. Let us consider clause γ: $p(X_1,X_2,X_3) \leftarrow X_1 < 1 \wedge X_1 \geq Z_1+1 \wedge Z_2 > 0 \wedge q(Z_1, f(X_3), Z_2) \wedge r(X_2)$ and clause δ: $s(Y_1, Y_2, Y_3) \leftarrow W_1 < 0 \wedge Y_1-3 \geq 2W_1 \wedge W_2 > 0 \wedge q(W_1, Y_3, W_2)$ of the Introduction. Let the substitution α: $\{W_1/Z_1, Y_3/f(X_3), W_2/Z_2, Y_2/a\}$ and the goal R: $r(X_2)$ be the result of

applying the procedure **GM** to γ and δ as shown in Example 1, and let the constraint $e : X_1 < 1$ and the substitution $\beta : \{Y_1/2X_1{+}1\} \cup \sigma_W$ be the result of applying the procedure **CM** to γ and $\delta\alpha$ as shown in Example 2. 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 Algorithm and Experimental Results

Let us first analyze the time complexity of our folding algorithm **FA** by assuming that: (i) each rule application during the goal matching procedure **GM** and the constraint matching procedure **CM** takes constant time, and (ii) each computation of the functions *nf*, *solve*, and *project* takes constant time. In these hypotheses our **FA** algorithm is in NP (w.r.t. the number of occurrences of symbols in the input clauses). To show this result, it is sufficient to show that both the goal matching procedure **GM** and the constraint matching procedure **CM** are in NP.

We have that **GM** is in NP w.r.t. the number of occurrences of symbols in the two goals B and G appearing in the input clauses. Indeed, rule (i) of **GM** chooses a mapping from the set of the occurrences of the literals of B to the set of occurrences of the literals of G and each application of any other rule of **GM** consumes at least one symbol of the input clauses.

We have that also **CM** is in NP w.r.t. the number N of occurrences of symbols in the initial triple $\langle c \leftrightarrow e \wedge d',\ \emptyset,\ \emptyset \rangle$. Indeed, rule (i) of **CM** chooses a mapping from the set of occurrences of the atomic constraints in c to the set of occurrences of the atomic constraints in $e \wedge d'$. Moreover, the length of any sequence of applications of the other rules of **CM** is polynomial in N as we now show. First, we may assume that the applications of rules (iv) and (v) are done after the applications of rules (i), (ii), and (iii). Since each application of rules (i), (ii), and (iii) reduces the number of constraints occurring in the first component of the triple at hand, we may have at most N applications of these three rules. Moreover, each application of rules (i), (ii), and (iii) introduces at most $m{+}1$ new variables, with $m{+}1 \leq N$. Hence, at most N^2 new variables are introduced. Rule (iv) can be applied at most M times, where M is the number of variable occurrences in the second component of the triple at hand. Finally, each application of rule (v) eliminates all occurrences of one variable in Y, which is a subset of the variables occurring in the input triple and, therefore, this rule can be applied at most N times. Moreover, for each application of rule (v), the cardinality of the second component of the triple at hand does not change and the number of variable occurrences in each constraint in that component is bounded by the cardinality of $X \cup Y \cup Z$ (which is at most N). Thus, M is bounded by a polynomial of the value of N.

A more detailed time complexity analysis of our folding algorithm **FA** where we do *not* assume that the functions *nf*, *solve*, and *project* are computed in constant time, is as follows. (i) *nf* takes polynomial time in the size of its argument, (ii) *solve* takes polynomial time in the number of variables of its argument by using Khachiyan's method [12], and (iii) *project* takes $O(2^v)$ time, where

$v = |Vars(c) \cap Vars(B')|$ (see [13] for the complexity of variable elimination from linear constraints). Since the *project* function is applied only once at the beginning of the procedure **CM**, we get that the computation of our **FA** algorithm requires nondeterministic polynomial time plus $O(2^v)$ time.

Note that since matching modulo the equational theory $\mathrm{AC}_\wedge$ is NP-complete [14,17], one cannot hope for a folding algorithm whose asymptotic time complexity is significantly better than our **FA** algorithm.

In the following Table 1 we report some experimental results for our algorithm **FA**, implemented in SICStus Prolog 3.12, on a Pentium IV 3GHz. We have considered the example $D0$ of the Introduction, the four examples $D1$–$D4$ for which folding can be done in one way only (*Number of Foldings* $=1$), and the four examples $N1$–$N4$ for which folding can be done in more than one way (*Number of Foldings* >1).

The *Number of Variables* row indicates the number of variables in clause γ (to be folded) plus the number of variables in clause δ (used for folding). The *Time* row indicates the seconds required for finding the folded clause (or the first folded clause, in examples $N1$–$N4$). The *Total-Time* row indicates the seconds required for finding all folded clauses. (Note that even when there exists one folded clause only, *Total-Time* is greater than *Time* because, after the folded clause has been found, **FA** checks that no other folded clauses can be computed.)

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 $r \leftarrow C>1 \wedge 3>C \wedge D>1 \wedge 3>D \wedge q(C) \wedge q(D)$. Similar clauses (with more variables) have been used in the other examples.

Our algorithm **FA** performs reasonably well in practice. However, when the number of variables (and, in particular, the number of variables are of type `rat`) increases, the performance rapidly deteriorates.

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 (in seconds)	0.01	0.01	0.08	3.03	306	0.02	0.08	0.23	1.09
Total-Time (in 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 various examples.

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 [2] and for proving program properties [3]. This technique makes use of the definition, unfolding, and folding rules [5,6,7,8,9,10]. In this paper we have considered constraint logic programs, where the constraints are linear inequations over the rational (or real) numbers, and we have focused

on the problem of automating the 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 [3,7,8,9,10], are specified in a declarative way and no algorithm is 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, [10]) 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 above mentioned program transformation techniques [2,3] which improve program efficiency or prove program properties by eliminating existential variables.

We have proved the termination and the soundness of our 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. The class of admissible constraints is quite large. We have also 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 similar to the problem of matching two terms modulo an associative, commutative (AC, for short) equational theory [18,19]. 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.

The *constraint matching* procedure solves a generalized form of the matching problem, modulo the equational theory $\mathcal{Q}$ of linear inequations over the rational numbers. That problem can be seen as a *restricted unification* problem [20]. In [20] it is described how to obtain, under certain conditions, 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 $\mathcal{Q}$ of constraints a solution is provided in the literature neither for the restricted unification problem nor for the unrestricted one. Moreover, one cannot apply the so called *combination methods* either [21]. 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 $\mathcal{Q}$ because some applicability conditions are not satisfied and, in particular, $\mathcal{Q}$ is neither *collapse-free* nor *regular* [21].

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 [22].

Acknowledgements

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

References

1. Jaffar, J., Maher, M.: Constraint logic programming: A survey. Journal of Logic Programming 19/20, 503–581 (1994)
2. Proietti, M., Pettorossi, A.: Unfolding-definition-folding, in this order, for avoiding unnecessary variables in logic programs. Theo. Comp. Sci. 142(1), 89–124 (1995)
3. Pettorossi, A., Proietti, M., Senni, V.: Proving properties of constraint logic programs by eliminating existential variables. In Etalle, S., Truszczynski, M. (eds.) ICLP 2006. LNCS, vol. 4079, pp. 179–195. Springer (2006)
4. Burstall, R.M., Darlington, J.: A transformation system for developing recursive programs. Journal of the ACM 24(1), 44–67 (1977)
5. Tamaki, H., Sato, T.: Unfold/fold transformation of logic programs. In: Tärnlund, S.Å. (ed.) Proc. ICLP '84, pp. 127–138. Uppsala University, Uppsala, Sweden (1984)
6. H. Seki. Unfold/fold transformation of stratified programs. *Theoretical Computer Science*, 86:107–139, 1991.
7. Maher, M.J.: A transformation system for deductive database modules with perfect model semantics. Theoretical Computer Science 110, 377–403 (1993)
8. Etalle, S., Gabbrielli, M.: Transformations of CLP modules. Theoretical Computer Science 166, 101–146 (1996)
9. Bensaou, N., Guessarian, I.: Transforming constraint logic programs. Theoretical Computer Science 206, 81–125 (1998)
10. Fioravanti, F., Pettorossi, A., Proietti, M.: Transformation rules for locally stratified constraint logic programs. In Lau, K.K., Bruynooghe, M. (eds.) Program Development in Computational Logic. LNCS, vol. 3049, pp. 292–340. Springer (2004)
11. Lloyd, J.W.: Foundations of Logic Programming. Second Edition. Springer (1987)
12. Schrijver, A.: Theory of Linear and Integer Programming. J. Wiley & Sons (1986)
13. Weispfenning, V.: The complexity of linear problems in fields. J. Symb. Comput. 5(1-2), 3–27 (1988)
14. Baader, F., Snyder, W.: Unification theory. In Robinson, A., Voronkov, A. (eds.) Handbook of Automated Reasoning. Vol. I, pp. 445–532. Elsevier Science (2001)
15. Terese: Term Rewriting Systems. Cambridge University Press (2003)
16. Senni, V.: Transformation Techniques for Constraint Logic Programs with Application to Protocol Verification. PhD thesis, University of Rome "Tor Vergata", Rome, Italy (2008)
17. Benanav, D., Kapur, D., Narendran, P.: Complexity of matching problems. Journal of Symbolic Computation 3(1-2), 203–216 (1987)
18. Livesey, M., Siekmann, J.: Unification of A+C Terms (Bags) and A+C+I Terms (Sets). TR 3/76, Institut für Informatik I, Universität Karlsruhe (1976)
19. Stickel, M.E.: A unification algorithm for associative-commutative functions. J. ACM 28(3), 423–434 (1981)

20. Bürckert, H.J.: Some relationships between unification, restricted unification, and matching. Proc. CADE '86. LNCS, vol. 230, pp. 514–524. Springer (1986)
21. Ringeissen, C.: Matching in a class of combined non-disjoint theories. In Baader, F. (ed.) CADE 2003. LNCS, vol. 2741, pp. 212–227. Springer (2003)
22. The MAP transformation system. http://www.iasi.cnr.it/∼proietti/system.html