Transformation Rules for Locally Stratified Constraint Logic Programs

Fabio Fioravanti[1], Alberto Pettorossi[2], Maurizio Proietti[3]

(1) Dipartimento di Informatica, Università dell'Aquila, L'Aquila, Italy
fioravan@univaq.it
(2) DISP, University of Tor Vergata, Roma, Italy
adp@iasi.rm.cnr.it
(3) IASI-CNR, Roma, Italy
proietti@iasi.rm.cnr.it

Abstract We propose a set of transformation rules for constraint logic programs with negation. We assume that every program is locally stratified and, thus, it has a unique perfect model. We give sufficient conditions which ensure that the proposed set of transformation rules preserves the perfect model of the programs. Our rules extend in some respects the rules for logic programs and constraint logic programs already considered in the literature and, in particular, they include a rule for unfolding a clause with respect to a negative literal.

1 Introduction

Program transformation is a very powerful methodology for developing correct and efficient programs from formal specifications. This methodology is particularly convenient in the case of declarative programming languages, where programs are formulas and program transformations can be viewed as replacements of formulas by new, equivalent formulas.

The main advantage of using the program transformation methodology for program development is that it allows us to address the correctness and the efficiency issues at separate stages. Often little effort is required for encoding formal specifications (written by using equational or logical formalisms) as declarative programs (written as functional or logic programs). These programs are correct by construction, but they are often computationally inefficient. Here is where program transformation comes into play: from a correct (and possibly inefficient) initial program version we can derive a correct and efficient program version by means of a sequence of program transformations that preserve correctness. We say that a program transformation preserves correctness, or it is *correct*, if the semantics of the initial program is equal to the semantics of the derived program.

A very popular approach followed when applying the program transformation methodology, is the one based on *transformation rules* and *strategies* [9]: the rules are elementary transformations that preserve the program semantics and the strategies are (possibly nondeterministic) procedures that guide the application of transformation rules with the objective of deriving efficient programs. Thus, a

program transformation is realized by a sequence $P_0, \ldots, P_n$ of programs, called a *transformation sequence*, where, for $i = 0, \ldots, n-1$, P_{k+1} is derived from P_k by applying a transformation rule according to a given transformation strategy. A transformation sequence is said to be *correct* if the programs $P_0, \ldots, P_n$ have the same semantics.

Various sets of program transformation rules have been proposed in the literature for several declarative programming languages, such as, functional [9,39], logic [44], constraint [7,11,27], and functional-logic languages [1]. In this paper we consider a constraint logic programming language with negation [19,28] and we study the correctness of a set of transformation rules that extends the sets which were already considered for constraint logic programming languages. We will not deal here with transformation strategies, but we will show through some examples (see Section 5) that the transformation rules can be applied in a rather systematic (yet not fully automatic) way.

We assume that constraint logic programs are *locally stratified* [4,35]. This assumption simplifies our treatment because the semantics of a locally stratified program is determined by its unique *perfect model* which is equal to its unique *stable model*, which is also its unique, total *well-founded model* [4,35]. (The definitions of locally stratified programs, perfect models, and other notions used in this paper are recalled in Section 2.)

The set of transformation rules we consider in this paper includes the *unfolding* and *folding* rules (see, for instance, [7,11,16,17,23,27,29,31,37,38,40,42,43,44]). In order to understand how these rules work, let us first consider propositional programs. The *definition* of an atom a in a program is the set of clauses that have a as head. The atom a is also called the *definiendum*. The disjunction of the bodies of the clauses that constitute the definition of a, is called the *definiens*. Basically, the application of the unfolding rule consists in replacing an atom occurring in the body of a clause by its definiens and then applying, if necessary, some suitable boolean laws to obtain clauses. For instance, given the following programs P_1 and P_2:

$$\begin{array}{lll} P_1\colon & p \leftarrow q \wedge r & \qquad P_2\colon \; p \leftarrow \neg a \wedge r \\ & q \leftarrow \neg a & \qquad\quad\;\; p \leftarrow b \wedge r \\ & q \leftarrow b & \qquad\quad\;\; q \leftarrow \neg a \\ & & \qquad\quad\;\; q \leftarrow b \end{array}$$

we have that by unfolding the first clause of program P_1 we get program P_2.

Folding is the inverse of unfolding and consists in replacing an occurrence of a definiens by the corresponding occurrence of the definiendum (before this replacement we may apply suitable boolean laws). For instance, by folding the first two clauses of P_2 using the definition of q, we get program P_1. An important feature of the folding rule is that the definition used for folding may occur in a previous program in the transformation sequence. The formal definitions of the unfolding and folding transformation rules for constraint logic programs will be given in Section 3. The usefulness of the program transformation approach based on the unfolding and folding rules, is now very well recognized in the scientific community as indicated by a large number of papers (see [29] for a survey).

A relevant property we will prove in this paper is that the unfolding of a clause w.r.t. an atom occurring in a negative literal, also called *negative unfolding*, preserves the perfect model of a locally stratified program. This property is interesting, because negative unfolding is useful for program transformation, but it may *not* preserve the perfect models (nor the stable models, nor the well-founded model) if the programs are not locally stratified. For instance, let us consider the following programs P_1 and P_2:

$$\begin{array}{lll} P_1\colon & p \leftarrow \neg q & \qquad\qquad P_2\colon \; p \leftarrow p \\ & q \leftarrow \neg p & \qquad\qquad\quad\; q \leftarrow \neg p \end{array}$$

Program P_2 can be obtained by unfolding the first clause of P_1 (i.e., by first replacing q by the body $\neg p$ of the clause defining q, and then replacing $\neg\neg p$ by p). Program P_1 has two perfect models: $\{p\}$ and $\{q\}$, while program P_2 has the unique perfect model $\{q\}$.

In this paper we consider the following transformation rules (see Section 3): definition introduction and definition elimination (for introducing and eliminating definitions of predicates), positive and negative unfolding, positive and negative folding (that is, unfolding and folding w.r.t. a positive and a negative occurrence of an atom, respectively), and also rules for applying boolean laws and rules for manipulating constraints.

Similarly to other sets of transformation rules presented in the literature (see, for instance, [1,7,9,11,27,39,44]), a transformation sequence constructed by arbitrary applications of the transformation rules presented in this paper, may be incorrect. As customary, we will ensure the correctness of transformation sequences only if they satisfy suitable properties: we will call them *admissible* sequences (see Section 4). Although our transformation rules are extensions or adaptations of transformation rules already considered for stratified logic programs or logic programs, in general, for our correctness proof we cannot rely on already known results. Indeed, the definition of an admissible transformation sequence depends on the interaction among the rules and, in particular, correctness may not be preserved if we modify even one rule only.

To see that known results do not extend in a straightforward way when adding negative unfolding to a set of transformation rules, let us consider the transformation sequences constructed by first (1) unfolding all clauses of a definition δ and then (2) folding some of the resulting clauses by using the definition δ itself. If at Step (1) we use positive unfolding only, then the perfect model semantics is preserved [37,42], while this semantics may not be preserved if we use negative unfolding, as indicated by the following example.

Example 1. Let us consider the transformation sequence P_0, P_1, P_2, where:

$$\begin{array}{lll} P_0\colon \; p(X) \leftarrow \neg q(X) & P_1\colon p(X) \leftarrow X<0 \wedge \neg q(X) & P_2\colon \; p(X) \leftarrow X<0 \wedge p(X) \\ \quad\; q(X) \leftarrow X \geq 0 & \quad\; q(X) \leftarrow X \geq 0 & \quad\; q(X) \leftarrow X \geq 0 \\ \quad\; q(X) \leftarrow q(X) & \quad\; q(X) \leftarrow q(X) & \quad\; q(X) \leftarrow q(X) \end{array}$$

Program P_1 is derived by unfolding the first clause of P_0 w.r.t. the negative literal $\neg q(X)$ (that is, by replacing the definiendum $q(X)$ by its definiens $X \geq 0 \vee q(X)$, and then applying De Morgan's law). Program P_2 is derived by folding the first

clause of P_1 using the definition $p(X) \leftarrow \neg q(X)$ in P_0. We have that, for any $a < 0$, the atom $p(a)$ belongs to the perfect model of P_0, while $p(a)$ does not belong to the perfect model of P_2.

The main result of this paper (see Theorem 3 in Section 4) shows the correctness of a transformation sequence constructed by first (1) unfolding all clauses of a (non-recursive) definition δ w.r.t. a *positive* literal, then (2) unfolding zero or more clauses w.r.t. a *negative* literal, and finally (3) folding some of the resulting clauses by using the definition δ. The correctness of such transformation sequences cannot be established by the correctness results presented in [37,42].

The paper is structured as follows. In Section 2 we present the basic definitions of locally stratified constraint logic programs and perfect models. In Section 3 we present our set of transformation rules and in Section 4 we give sufficient conditions on transformation sequences that ensure the preservation of perfect models. In Section 5 we present some examples of program derivation using our transformation rules. In all these examples the negative unfolding rule plays a crucial role. Finally, in Section 6 we discuss related work and future research.

2 Preliminaries

In this section we recall the syntax and semantics of constraint logic programs with negation. In particular, we will give the definitions of locally stratified programs and perfect models. For notions not defined here the reader may refer to [2,4,19,20,26].

2.1 Syntax of Constraint Logic Programs

We consider a first order language $\mathcal{L}$ generated by an infinite set *Vars* of *variables*, a set *Funct* of *function symbols* with arity, and a set *Pred* of *predicate symbols* (or predicates, for short) with arity. We assume that *Pred* is the union of two disjoint sets: (i) the set $Pred_c$ of *constraint* predicate symbols, including the *equality* symbol $=$, and (ii) the set $Pred_u$ of *user defined* predicate symbols.

A *term* of $\mathcal{L}$ is either a variable or an expression of the form $f(t_1, \ldots, t_n)$, where f is an n-ary function symbol and $t_1, \ldots, t_n$ are terms. An *atomic formula* is an expression of the form $p(t_1, \ldots, t_n)$ where p is an n-ary predicate symbol and $t_1, \ldots, t_n$ are terms. A *formula* of $\mathcal{L}$ is either an atomic formula or a formula constructed from atomic formulas by means of connectives ($\neg$, $\wedge$, $\vee$, $\rightarrow$, $\leftarrow$, $\leftrightarrow$) and quantifiers ($\exists$, $\forall$).

Let e be a term, or a formula, or a set of terms or formulas. The set of variables occurring in e is denoted by $vars(e)$. Given a formula φ, the set of the *free variables* occurring in φ is denoted by $FV(\varphi)$. A term or a formula is *ground* iff it does not contain variables. Given a set $X = \{X_1, \ldots, X_n\}$ of n variables, by $\forall X\, \varphi$ we denote the formula $\forall X_1 \ldots \forall X_n\, \varphi$. By $\forall(\varphi)$ we denote the *universal closure* of φ, that is, the formula $\forall X\, \varphi$, where $FV(\varphi) = X$. Analogous notations will be adopted for the existential quantifier $\exists$.

A *primitive constraint* is an atomic formula $p(t_1, \ldots, t_n)$ where p is a predicate symbol in $Pred_c$. The set $\mathcal{C}$ of *constraints* is the smallest set of formulas of $\mathcal{L}$ that contains all primitive constraints and is closed w.r.t. negation, conjunction, and existential quantification. This closure assumption simplifies our treatment, but as we will indicate at the end of this section, we can do without it.

An *atom* is an atomic formula $p(t_1, \ldots, t_n)$ where p is an element of $Pred_u$ and $t_1, \ldots, t_n$ are terms. A *literal* is either an atom A, also called *positive literal*, or a negated atom $\neg A$, also called *negative literal*. Given any literal L, by $\overline{L}$ we denote: (i) $\neg A$, if L is the atom A, and (ii) A, if L is the negated atom $\neg A$. A *goal* is a (possibly empty) conjunction of literals (here we depart from the terminology used in [2,26], where a goal is defined as the negation of a conjunction of literals). A *constrained literal* is the conjunction of a constraint and a literal. A *constrained goal* is the conjunction of a constraint and a goal.

A *clause* γ is a formula of the form $H \leftarrow c \wedge G$, where: (i) H is an atom, called the *head* of γ and denoted $hd(\gamma)$, and (ii) $c \wedge G$ is a constrained goal, called the *body* of γ and denoted $bd(\gamma)$. A conjunction of constraints and/or literals may be empty (in which case it is equivalent to *true*). A clause of the form $H \leftarrow c$, where c is a constraint and the goal part of the body is the empty conjunction of literals, is called a *constrained fact*. A clause of the form $H \leftarrow$, whose body is the empty conjunction, is called a *fact*.

A *constraint logic program* (or *program*, for short) is a finite set of clauses. A *definite clause* is a clause whose body has no occurrences of negative literals. A *definite program* is a finite set of definite clauses.

Given two atoms $p(t_1, \ldots, t_n)$ and $p(u_1, \ldots, u_n)$, we denote by $p(t_1, \ldots, t_n) = p(u_1, \ldots, u_n)$ the constraint: $t_1 = u_1 \wedge \ldots \wedge t_n = u_n$. For the notion of *substitution* and for the application of a substitution to a term we refer to [2,26]. Given a formula φ and a substitution $\{X_1/t_1, \ldots, X_n/t_n\}$ we denote by $\varphi\{X_1/t_1, \ldots, X_n/t_n\}$ the result of simultaneously replacing in φ all free occurrences of $X_1, \ldots, X_n$ by $t_1, \ldots, t_n$.

We say that a predicate p *immediately depends on* a predicate q in a program P iff there exists in P a clause of the form $p(\ldots) \leftarrow B$ and q occurs in B. We say that p *depends on* q in P iff there exists a sequence $p_1, \ldots, p_n$, with $n > 1$, of predicates such that: (i) $p_1 = p$, (ii) $p_n = q$, and (iii) for $i = 1, \ldots, n-1$, p_i immediately depends on p_{i+1}. Given a user defined predicate p and a program P, the *definition of* p *in* P, denoted $Def(p, P)$, is the set of clauses γ in P such that p is the predicate symbol of $hd(\gamma)$.

A *variable renaming* is a bijective mapping from *Vars* to *Vars*. The application of a variable renaming ρ to a formula φ returns the formula $\rho(\varphi)$, which is said to be a *variant* of φ, obtained by replacing each (bound or free) variable occurrence X in φ by the variable $\rho(X)$. A variant of a set $\{\varphi_1, \ldots, \varphi_n\}$ of formulas is the set $\{\rho(\varphi_1), \ldots, \rho(\varphi_n)\}$, also denoted $\rho(\{\varphi_1, \ldots, \varphi_n\})$. During program transformation we will feel free to silently apply variable renamings to clauses and to sets of clauses because, as the reader may verify, they preserve program semantics (see Section 2.2). Moreover, we will feel free to change the names of the bound variables occurring in constraints, as usually done in predicate calculus.

2.2 Semantics of Constraint Logic Programs

In this section we present the definition of the semantics of constraint logic programs with negation. This definition extends similar definitions given in the literature for definite constraint logic programs [19] and logic programs with negation [4,35].

We proceed as follows: (i) we define an *interpretation for the constraints*, following the approach used in first order logic (see, for instance, [2]), (ii) we introduce the notion of $\mathcal{D}$*-model*, that is, a model for constraint logic programs which is parametric w.r.t. the interpretation $\mathcal{D}$ for the constraints, (iii) we introduce the notion of *locally stratified program*, and finally, (iv) we define the *perfect* $\mathcal{D}$*-model* (also called *perfect model*, for short) of locally stratified programs.

An *interpretation* $\mathcal{D}$ *for the constraints* consists of: (1) a non-empty set D, called *carrier*, (2) an assignment of a function $f_{\mathcal{D}}$: $D^n \to D$ to each n-ary function symbol f in *Funct*, and (3) an assignment of a relation $p_{\mathcal{D}}$ over D^n to each n-ary predicate symbol in $Pred_c$. In particular, $\mathcal{D}$ assigns the set $\{\langle d, d\rangle \mid d \in D\}$ to the equality symbol $=$.

We assume that D is a set of ground terms. This is not restrictive because we may add suitable 0-ary function symbols to $\mathcal{L}$.

Given a formula φ whose predicate symbols belong to $Pred_c$, we consider the satisfaction relation $\mathcal{D} \models \varphi$, which is defined as usual in first order predicate calculus (see, for instance, [2]). A constraint c is said to be *satisfiable* iff its existential closure is satisfiable, that is, $\mathcal{D} \models \exists(c)$. If $\mathcal{D} \not\models \exists(c)$, then c is said to be *unsatisfiable* in $\mathcal{D}$.

Given an interpretation $\mathcal{D}$ for the constraints, a $\mathcal{D}$*-interpretation* I assigns a relation over D^n to each n-ary user defined predicate symbol in $Pred_u$, that is, I can be identified with a subset of the set $\mathcal{B}_{\mathcal{D}}$ of ground atoms defined as follows: $\mathcal{B}_{\mathcal{D}} = \{p(d_1, \ldots, d_n) \mid p$ is a predicate symbol in $Pred_u$ and $(d_1, \ldots, d_n) \in D^n\}$. A *valuation* is a function v: $Vars \to D$. We extend the domain of a valuation v to terms, constraints, literals, and clauses as we now indicate. Given a term t, we inductively define the term $v(t)$ as follows: (i) if t is a variable X then $v(t) = v(X)$, and (ii) if t is $f(t_1, \ldots, t_n)$ then $v(t) = f_{\mathcal{D}}(v(t_1), \ldots, v(t_n))$. Given a constraint c, $v(c)$ is the constraint obtained by replacing every free variable $X \in FV(c)$ by the ground term $v(X)$. Notice that $v(c)$ is a closed formula which may be not ground. Given a literal L, (i) if L is the atom $p(t_1, \ldots, t_n)$, then $v(L)$ is the ground atom $p(v(t_1), \ldots, v(t_n))$, and (ii) if L is the negated atom $\neg A$, then $v(L)$ is the ground, negated atom $\neg v(A)$. Given a clause γ: $H \leftarrow c \wedge L_1 \wedge \ldots \wedge L_m$, $v(\gamma)$ is the clause $v(H) \leftarrow v(c) \wedge v(L_1) \wedge \ldots \wedge v(L_m)$.

Let I be a $\mathcal{D}$-interpretation and v a valuation. Given a literal L, we say that $v(L)$ *is true in* I iff either (i) L is an atom and $v(L) \in I$, or (ii) L is a negated atom $\neg A$ and $v(A) \notin I$. We say that the literal $v(L)$ *is false in* I iff it is not true in I. Given a clause γ: $H \leftarrow c \wedge L_1 \wedge \ldots \wedge L_m$, $v(\gamma)$ *is true in* I iff either (i) $v(H)$ is true in I, or (ii) $\mathcal{D} \not\models v(c)$, or (iii) there exists $i \in \{1, \ldots, m\}$ such that $v(L_i)$ is false in I.

A $\mathcal{D}$-interpretation I is a $\mathcal{D}$*-model* of a program P iff for every clause γ in P and for every valuation v, we have that $v(\gamma)$ is true in I. It can be shown that

every definite constraint logic program P has a *least* $\mathcal{D}$-model w.r.t. set inclusion (see, for instance [20]).

Unfortunately, constraint logic programs which are not definite may fail to have a least $\mathcal{D}$-model. For example, the program consisting of the clause $p \leftarrow \neg q$ has the two minimal (not least) models $\{p\}$ and $\{q\}$. This fact has motivated the introduction of the set of *locally stratified* programs [4,35]. For every locally stratified program one can associate a unique (minimal, but not least, w.r.t. set inclusion) model, called *perfect model*, as follows.

A *local stratification* is a function σ: $\mathcal{B}_\mathcal{D} \rightarrow W$, where W is the set of countable ordinals. If $A \in \mathcal{B}_\mathcal{D}$ and $\sigma(A)$ is the ordinal α, we say that the *stratum* of A is α. Given a clause γ in a program P, a valuation v, and a local stratification σ, we say that a clause $v(\gamma)$ of the form: $H \leftarrow c \wedge L_1 \wedge \ldots \wedge L_m$ is *locally stratified* w.r.t. σ iff *either* $\mathcal{D} \models \neg c$ *or*, for $i = 1, \ldots, m$, if L_i is an atom A then $\sigma(H) \geq \sigma(A)$ else if L_i is a negated atom $\neg A$ then $\sigma(H) > \sigma(A)$. Given a local stratification σ, we say that program P is *locally stratified w.r.t.* σ, or σ is a *local stratification for* P, iff for every clause γ in P and for every valuation v, the clause $v(\gamma)$ is locally stratified w.r.t. σ. A program P is *locally stratified* iff there exists a local stratification σ such that P is *locally stratified w.r.t.* σ. For instance, let us consider the following program *Even*:

$even(0) \leftarrow$
$even(X) \leftarrow X = Y + 1 \ \wedge \ \neg even(Y)$

where the interpretation for the constraints is as follows: (1) the carrier is the set of the natural numbers, and (2) the addition function is assigned to the function symbol $+$. The program *Even* is locally stratified w.r.t. the stratification function σ such that for every natural number n, $\sigma(even(n)) = n$.

The perfect model of a program P which is locally stratified w.r.t. a stratification function σ is the least $\mathcal{D}$-model of P w.r.t. a suitable ordering based on σ, as specified by the following definition. This ordering is, in general, different from set inclusion.

Definition 1. (*Perfect Model*) [35]. *Let P be a locally stratified program, let σ be any local stratification for P, and let I, J be $\mathcal{D}$-interpretations. We say that I is* preferable *to J, and we write $I \prec J$ iff for every $A_1 \in I - J$ there exists $A_2 \in J - I$ such that $\sigma(A_1) > \sigma(A_2)$. A $\mathcal{D}$-model M of P is called a* perfect $\mathcal{D}$-model (*or a* perfect model, *for short*) *iff for every $\mathcal{D}$-model N of P different from M, we have that $M \prec N$.*

It can be shown that the perfect model of a locally stratified program always exists and does not depend on the choice of the local stratification function σ, as stated by the following theorem.

Theorem 1. [35] *Every locally stratified program P has a unique perfect model $M(P)$.*

By Theorem 1, $M(P)$ is the least $\mathcal{D}$-model of P w.r.t. the $\prec$ ordering. For instance, the perfect model of the program consisting of the clause $p \leftarrow \neg q$ is

$\{p\}$ because $\sigma(p) > \sigma(q)$ and, thus, the $\mathcal{D}$-model $\{p\}$ is preferable to the $\mathcal{D}$-model $\{q\}$ (i.e., $\{p\} \prec \{q\}$). Similarly, it can be verified that the perfect model of the program *Even* is $M(Even) = \{even(n) \mid n$ is an even non-negative integer$\}$. In Section 4 we will provide a method for constructing the perfect model of a locally stratified program based on the notion of *proof tree*.

Let us conclude this section by showing that the assumption that the set $\mathcal{C}$ of constraints is closed w.r.t. negation, conjunction, and existential quantification is not really needed. Indeed, given a locally stratified clause $H \leftarrow c \wedge G$, where the constraint c is written by using negation, or conjunction, or existential quantification, we can replace $H \leftarrow c \wedge G$ by an equivalent set of locally stratified clauses. For instance, if c is $\exists X\, d$ then we can replace $H \leftarrow c \wedge G$ by the two clauses:

$H \leftarrow newp(Y_1, \ldots, Y_n) \wedge G$
$newp(Y_1, \ldots, Y_n) \leftarrow d$

where *newp* is a new, user defined predicate and $\{Y_1, \ldots, Y_n\} = FV(\exists X\, d)$. Analogous replacements can be applied in the case where a constraint is written by using negation or conjunction.

3 The Transformation Rules

In this section we present a set of rules for transforming locally stratified constraint logic programs. We postpone to Section 6 the detailed comparison of our set of transformation rules with other sets of rules which were proposed in the literature for transforming logic programs and constraint logic programs. The application of our transformation rules is illustrated by simple examples. More complex examples will be given in Section 5.

The transformation rules are used to construct a *transformation sequence*, that is, a sequence $P_0, \ldots, P_n$ of programs. We assume that P_0 is locally stratified w.r.t. a fixed local stratification function σ: $\mathcal{B}_{\mathcal{D}} \rightarrow W$, and we will say that $P_0, \ldots, P_n$ is constructed *using* σ. We also assume that we are given a set $Pred_{int} \subseteq Pred_u$ of *predicates of interest*.

A transformation sequence $P_0, \ldots, P_n$ is constructed as follows. Suppose that we have constructed a transformation sequence $P_0, \ldots, P_k$, for $0 \leq k \leq n-1$, the next program P_{k+1} in the transformation sequence is derived from program P_k by the application of a transformation rule among R1–R10 defined below.

Our first rule is the *definition introduction* rule, which is applied for introducing a new predicate definition. Notice that by this rule we can introduce a new predicate defined by m (≥ 1) non-recursive clauses.

R1. Definition Introduction. Let us consider m (≥ 1) clauses of the form:

$\delta_1 : newp(X_1, \ldots, X_h) \leftarrow c_1 \wedge G_1$
$\ldots$
$\delta_m : newp(X_1, \ldots, X_h) \leftarrow c_m \wedge G_m$

where:

(i) *newp* is a predicate symbol not occurring in $\{P_0, \ldots, P_k\}$,
(ii) $X_1, \ldots, X_h$ are distinct variables occurring in $FV(\{c_1 \wedge G_1, \ldots, c_m \wedge G_m\})$,
(iii) every predicate symbol occurring in $\{G_1, \ldots, G_m\}$ also occurs in P_0, and
(iv) for every ground substitution ϑ with domain $\{X_1, \ldots, X_h\}$,
$\sigma(newp(X_1, \ldots, X_h)\vartheta)$ is the least ordinal α such that, for every valuation v and for every $i = 1, \ldots, m$,
either (iv.1) $\mathcal{D} \models \neg v(c_i\vartheta)$ *or* (iv.2) for every literal L occurring in $v(G_i\vartheta)$, if L is an atom A then $\alpha \geq \sigma(A)$ else if L is a negated atom $\neg A$ then $\alpha > \sigma(A)$.
By *definition introduction* (or *definition*, for short) from program P_k we derive the program $P_{k+1} = P_k \cup \{\delta_1, \ldots, \delta_m\}$. For $k \geq 0$, $Defs_k$ denotes the set of clauses introduced by the definition rule during the transformation sequence $P_0, \ldots, P_k$. In particular, $Defs_0 = \emptyset$.

Condition (iv), which is needed to ensure that σ is a local stratification for each program in the transformation sequence $P_0, \ldots, P_{k+1}$ (see Proposition 1), is not actually restrictive, because *newp* is a predicate symbol *not* occurring in P_0 and, thus, we can always choose the local stratification σ for P_0 so that Condition (iv) holds. As a consequence of Condition (iv), $\sigma(newp(X_1, \ldots, X_h)\vartheta)$ is the least upper bound of $S_p \cup S_n$ w.r.t. $<$ where:

$$S_p = \{\sigma(A) \mid 1 \leq i \leq m,\ v \text{ is a valuation, } A \text{ occurs in } v(G_i\vartheta),\ \mathcal{D} \models v(c_i\vartheta)\}, \text{ and}$$
$$S_n = \{\sigma(A)+1 \mid 1 \leq i \leq m,\ v \text{ is a valuation, } \neg A \text{ occurs in } v(G_i\vartheta),\ \mathcal{D} \models v(c_i\vartheta)\}.$$

In particular, if for $i = 1, \ldots, m$, $\mathcal{D} \models \neg\exists(c_i\vartheta)$, then $S_p \cup S_n = \emptyset$ and we have that $\sigma(newp(X_1, \ldots, X_h)\vartheta) = 0$.

The *definition elimination* rule is the inverse of the definition introduction rule. It can be used to discard from a given program the definitions of predicates which are not of interest.

R2. Definition Elimination. Let p be a predicate such that no predicate of the set $Pred_{int}$ of the predicates of interest depends on p in P_k. By *eliminating* the definition of p, from program P_k we derive the new program $P_{k+1} = P_k - Def(p, P_k)$.

The *unfolding* rule consists in: (i) replacing an atom $p(t_1, \ldots, t_m)$ occurring in the body of a clause, by a suitable instance of the disjunction of the bodies of the clauses which are the definition of p, and (ii) applying suitable boolean laws for deriving clauses. The suitable instance of Step (i) is computed by adding a constraint of the form $p(t_1, \ldots, t_m) = K$ for each head K of a clause in $Def(p, P_k)$. There are two unfolding rules: (1) the positive unfolding rule, and (2) the negative unfolding rule, corresponding to the case where $p(t_1, \ldots, t_m)$ occurs positively and negatively, respectively, in the body of the clause to be unfolded. In order to perform Step (ii), in the case of positive unfolding we apply the distributivity law, and in the case of negative unfolding we apply De Morgan's, distributivity, and double negation elimination laws.

R3. Positive Unfolding. Let $\gamma: H \leftarrow c \wedge G_L \wedge A \wedge G_R$ be a clause in program P_k and let P'_k be a variant of P_k without common variables with γ. Let

$\gamma_1 :\ K_1 \leftarrow c_1 \wedge B_1$

$\ldots$

$\gamma_m :\ K_m \leftarrow c_m \wedge B_m$

where $m \geq 0$ and $B_1, \ldots, B_m$ are conjunction of literals, be all clauses of program P'_k such that, for $i = 1, \ldots, m$, $\mathcal{D} \models \exists(c \wedge A = K_i \wedge c_i)$.
By *unfolding clause* γ *w.r.t. the atom* A we derive the clauses

$\eta_1 :\ H \leftarrow c \wedge A = K_1 \wedge c_1 \wedge G_L \wedge B_1 \wedge G_R$

$\ldots$

$\eta_m :\ H \leftarrow c \wedge A = K_m \wedge c_m \wedge G_L \wedge B_m \wedge G_R$

and from program P_k we derive the program $P_{k+1} = (P_k - \{\gamma\}) \cup \{\eta_1, \ldots, \eta_m\}$.

Notice that if $m = 0$ then, by positive unfolding, clause γ is deleted from P_k.

Example 2. Let P_k be the following program:

1. $p(X) \leftarrow X \geq 1 \wedge q(X)$
2. $q(Y) \leftarrow Y = 0$
3. $q(Y) \leftarrow Y = Z + 1 \wedge q(Z)$

where we assume that the interpretation for the constraints is given by the structure $\mathcal{R}$ of the real numbers. Let us unfold clause 1 w.r.t. the atom $q(X)$. The constraint $X \geq 1 \wedge X = Y \wedge Y = 0$ constructed from the constraints of clauses 1 and 2 is unsatisfiable, that is, $\mathcal{R} \models \neg\exists X \exists Y (X \geq 1 \wedge X = Y \wedge Y = 0)$, while the constraint $X \geq 1 \wedge X = Y \wedge Y = Z + 1$ constructed from the constraints of clauses 1 and 3, is satisfiable. Thus, we derive the following program P_{k+1}:

1u. $p(X) \leftarrow X \geq 1 \wedge X = Y \wedge Y = Z + 1 \wedge q(Z)$
2. $q(Y) \leftarrow Y = 0$
3. $q(Y) \leftarrow Y = Z + 1 \wedge q(Z)$

R4. Negative Unfolding. Let $\gamma :\ H \leftarrow c \wedge G_L \wedge \neg A \wedge G_R$ be a clause in program P_k and let P'_k be a variant of P_k without common variables with γ. Let

$\gamma_1 :\ K_1 \leftarrow c_1 \wedge B_1$

$\ldots$

$\gamma_m :\ K_m \leftarrow c_m \wedge B_m$

where $m \geq 0$ and $B_1, \ldots, B_m$ are conjunction of literals, be all clauses of program P'_k such that, for $i = 1, \ldots, m$, $\mathcal{D} \models \exists(c \wedge A = K_i \wedge c_i)$. Suppose that, for $i = 1, \ldots, m$, there exist an idempotent substitution $\vartheta_i = \{X_{i1}/t_{i1}, \ldots, X_{in}/t_{in}\}$ and a constraint d_i such that the following conditions hold:

(i) $\mathcal{D} \models \forall(c \rightarrow ((A = K_i \wedge c_i) \leftrightarrow (X_{i1} = t_{i1} \wedge \ldots \wedge X_{in} = t_{in} \wedge d_i)))$,

(ii) $\{X_{i1}, \ldots, X_{in}\} \subseteq V_i$, where $V_i = FV(\gamma_i)$, and

(iii) $FV(d_i \wedge B_i\vartheta_i) \subseteq FV(c \wedge A)$.

Then, from the formula

$\psi_0 :\ c \wedge G_L \wedge \neg(\exists V_1\, (A = K_1 \wedge c_1 \wedge B_1) \vee \ldots \vee \exists V_m\, (A = K_m \wedge c_m \wedge B_m)) \wedge G_R$

we get an equivalent disjunction of constrained goals by performing the following steps. In these steps we silently apply the associativity of $\wedge$ and $\vee$.

Step 1. (*Eliminate* $\exists$) Since Conditions (i), (ii), and (iii) hold, we derive from ψ_0 the following equivalent formula:

$\psi_1 :\ c \wedge G_L \wedge \neg((d_1 \wedge B_1\vartheta_1) \vee \ldots \vee (d_m \wedge B_m\vartheta_m)) \wedge G_R$

Step 2. (*Push* $\neg$ *inside*) We apply to ψ_1 as long as possible the following rewritings of formulas, where d is a constraint, At is an atom, G, G_1, G_2 are goals, and D is a disjunction of constrained literals:

$$\begin{array}{lll} \neg((d \wedge G) \vee D) & \longrightarrow & \neg(d \wedge G) \wedge \neg D \\ \neg(d \wedge G) & \longrightarrow & \neg d \vee (d \wedge \neg G) \\ \neg(G_1 \wedge G_2) & \longrightarrow & \neg G_1 \vee \neg G_2 \\ \neg\neg At & \longrightarrow & At \end{array}$$

Thus, from ψ_1 we derive the following equivalent formula:

$$\begin{array}{ll} \psi_2 :\ c \wedge G_L \wedge (\neg d_1 & \vee\ (d_1 \wedge (\overline{L_{11}\vartheta_1} \quad \vee \ldots \vee\ \overline{L_{1p}\vartheta_1}))) \\ \quad \wedge & \ldots \\ \quad \wedge (\neg d_m & \vee\ (d_m \wedge (\overline{L_{m1}\vartheta_m} \vee \ldots \vee \overline{L_{mq}\vartheta_m}))) \\ \quad \wedge G_R & \end{array}$$

where $L_{11} \wedge \ldots \wedge L_{1p}$ is B_1, ..., and $L_{m1} \wedge \ldots \wedge L_{mq}$ is B_m.

Step 3. (*Push* $\vee$ *outside*) We apply to ψ_2 as long as possible the following rewriting of formulas, where φ_1, φ_2, and φ_3 are formulas:

$\varphi_1 \wedge (\varphi_2 \vee \varphi_3) \longrightarrow (\varphi_1 \wedge \varphi_2) \vee (\varphi_1 \wedge \varphi_3)$

and then we move constraints to the left of literals by applying the commutativity of $\wedge$. Thus, from ψ_2 we get an equivalent formula of the form:

$\psi_3 :\ (c \wedge e_1 \wedge G_L \wedge Q_1 \wedge G_R) \vee \ldots \vee (c \wedge e_r \wedge G_L \wedge Q_r \wedge G_R)$

where $e_1, \ldots, e_r$ are constraints and $Q_1, \ldots, Q_r$ are goals.

Step 4. (*Remove unsatisfiable disjuncts*) We remove from ψ_3 every disjunct $(c \wedge e_j \wedge G_L \wedge Q_j \wedge G_R)$, with $1 \leq j \leq r$, such that $\mathcal{D} \models \neg\exists(c \wedge e_j)$, thereby deriving an equivalent disjunction of constrained goals of the form:

$\psi_4 :\ (c \wedge e_1 \wedge G_L \wedge Q_1 \wedge G_R) \vee \ldots \vee (c \wedge e_s \wedge G_L \wedge Q_s \wedge G_R)$

By *unfolding clause* γ *w.r.t. the negative literal* $\neg A$ we derive the clauses

$$\begin{array}{l} \eta_1 :\ H \leftarrow c \wedge e_1 \wedge G_L \wedge Q_1 \wedge G_R \\ \quad \ldots \\ \eta_s :\ H \leftarrow c \wedge e_s \wedge G_L \wedge Q_s \wedge G_R \end{array}$$

and from program P_k we derive the program $P_{k+1} = (P_k - \{\gamma\}) \cup \{\eta_1, \ldots, \eta_s\}$.

Notice that: (i) if $m = 0$, that is, if we unfold clause γ w.r.t. a negative literal $\neg A$ such that the constraint $c \wedge A = K_i \wedge c_i$ is satisfiable for no clause $K_i \leftarrow c_i \wedge B_i$ in P'_k, then we get the new program P_{k+1} by deleting $\neg A$ from the body of clause γ, and (ii) if we unfold clause γ w.r.t. a negative literal $\neg A$ such that for some clause $K_i \leftarrow c_i \wedge B_i$ in P'_k, $\mathcal{D} \models \forall(c \rightarrow \exists V_i\, (A = K_i \wedge c_i))$ and B_i is the empty conjunction, then we derive the new program P_{k+1} by deleting clause γ from P_k.

An application of the negative unfolding rule is illustrated by the following example.

Example 3. Suppose that the following clause belongs to program P_k:

$\gamma:\ h(X) \leftarrow X \geq 0 \wedge \neg p(X)$

and let

$p(Y) \leftarrow Y = Z+1 \wedge Z \geq 0 \wedge q(Z)$
$p(Y) \leftarrow Y = Z-1 \wedge Z \geq 1 \wedge q(Z) \wedge \neg r(Z)$

be the definition of p in P_k. Suppose also that the constraints are interpreted in the structure $\mathcal{R}$ of the real numbers. Now let us unfold clause γ w.r.t. $\neg p(X)$. We start off from the formula:

$$\psi_0:\ X \geq 0 \wedge \neg(\ \exists Y\, \exists Z\, (X = Y \wedge Y = Z+1 \wedge Z \geq 0 \wedge q(Z)) \vee \exists Y\, \exists Z\, (X = Y \wedge Y = Z-1 \wedge Z \geq 1 \wedge q(Z) \wedge \neg r(Z)))$$

Then we perform the four steps indicated in rule R4 as follows.
Step 1. Since we have that:

$$\mathcal{R} \models \forall X\, \forall Y\, \forall Z\, (X \geq 0 \rightarrow (\ (X = Y \wedge Y = Z+1 \wedge Z \geq 0) \leftrightarrow (Y = X \wedge Z = X-1 \wedge X \geq 1)))$$

and

$$\mathcal{R} \models \forall X\, \forall Y\, \forall Z\, (X \geq 0 \rightarrow (\ (X = Y \wedge Y = Z-1 \wedge Z \geq 1) \leftrightarrow (Y = X \wedge Z = X+1)))$$

we derive the formula:

$$\psi_1:\ X \geq 0 \wedge \neg((X \geq 1 \wedge q(X-1)) \vee (q(X+1) \wedge \neg r(X+1)))$$

Steps 2 and 3. By applying the rewritings indicated in rule R4 we derive the following formula:

$$\begin{array}{l}\psi_3:\ (X \geq 0 \wedge \neg X \geq 1 \wedge \neg q(X+1)) \vee \\ \quad (X \geq 0 \wedge \neg X \geq 1 \wedge r(X+1)) \vee \\ \quad (X \geq 0 \wedge X \geq 1 \wedge \neg q(X-1) \wedge \neg q(X+1)) \vee \\ \quad (X \geq 0 \wedge X \geq 1 \wedge \neg q(X-1) \wedge r(X+1))\end{array}$$

Step 4. Since all constraints in the formula derived at the end of Steps 2 and 3 are satisfiable, no disjunct is removed.

Thus, by unfolding $h(X) \leftarrow X \geq 0 \wedge \neg p(X)$ w.r.t. $\neg p(X)$ we derive the following clauses:

$h(X) \leftarrow X \geq 0 \wedge \neg X \geq 1 \wedge \neg q(X+1)$
$h(X) \leftarrow X \geq 0 \wedge \neg X \geq 1 \wedge r(X+1)$
$h(X) \leftarrow X \geq 0 \wedge X \geq 1 \wedge \neg q(X-1) \wedge \neg q(X+1)$
$h(X) \leftarrow X \geq 0 \wedge X \geq 1 \wedge \neg q(X-1) \wedge r(X+1)$

The validity of Conditions (i), (ii), and (iii) in the negative unfolding rule allows us to eliminate the existential quantifiers as indicated at Step 1. If these conditions do not hold and nonetheless we eliminate the existential quantifiers, then negative unfolding may be incorrect, as illustrated by the following example.

Example 4. Let us consider the following programs P_0 and P_1, where P_1 is obtained by negative unfolding from P_0, but Conditions (i)–(iii) do not hold:

$$\begin{array}{lll} P_0: & p \leftarrow \neg q & \qquad P_1: \ p \leftarrow \neg r(X) \\ & q \leftarrow r(X) & \qquad\quad\ \ q \leftarrow r(X) \\ & r(X) \leftarrow X = 0 & \qquad\quad\ \ r(X) \leftarrow X = 0 \end{array}$$

We have that: $p \notin M(P_0)$ while $p \in M(P_1)$. (We assume that the carrier of the interpretation for the constraints contains at least one element different from 0.)

The reason why the negative unfolding step of Example 4 is incorrect is that the clause $q \leftarrow r(X)$ is, as usual, implicitly universally quantified at the front, and $\forall X\,(q \leftarrow r(X))$ is logically equivalent to $q \leftarrow \exists X\, r(X)$. Now, a correct negative unfolding rule should replace the clause $p \leftarrow \neg q$ in program P_0 by $p \leftarrow \neg\exists X\, r(X)$, while in program P_1 we have derived the clause $p \leftarrow \neg r(X)$ which, by making the quantification explicit at the front of the body, can be written as $p \leftarrow \exists X\, \neg r(X)$.

The *folding* rule consists in replacing instances of the bodies of the clauses that are the definition of a predicate by the corresponding head. As for unfolding, we have a positive folding and a negative folding rule, depending on whether folding is applied to positive or negative occurrences of (conjunctions of) literals. Notice that by the positive folding rule we may replace $m\,(\geq 1)$ clauses by one clause only.

R5. Positive Folding. Let $\gamma_1, \ldots, \gamma_m$, with $m \geq 1$, be clauses in P_k and let Defs'_k be a variant of Defs_k without common variables with $\gamma_1, \ldots, \gamma_m$. Let the definition of a predicate in Defs'_k consist of the clauses

$\delta_1:\ K \leftarrow d_1 \wedge B_1$

$\ldots$

$\delta_m:\ K \leftarrow d_m \wedge B_m$

where, for $i = 1, \ldots, m$, B_i is a non-empty conjunction of literals. Suppose that there exists a substitution ϑ such that, for $i = 1, \ldots, m$, clause γ_i is of the form $H \leftarrow c \wedge d_i\vartheta \wedge G_L \wedge B_i\vartheta \wedge G_R$ and, for every variable X in the set $FV(d_i \wedge B_i) - FV(K)$, the following conditions hold: (i) $X\vartheta$ is a variable not occurring in $\{H, c, G_L, G_R\}$, and (ii) $X\vartheta$ does not occur in the term $Y\vartheta$, for any variable Y occurring in $d_i \wedge B_i$ and different from X.
By *folding clauses* $\gamma_1, \ldots, \gamma_m$ *using clauses* $\delta_1, \ldots, \delta_m$ we derive the clause η: $H \leftarrow c \wedge G_L \wedge K\vartheta \wedge G_R$ and from program P_k we derive the program $P_{k+1} = (P_k - \{\gamma_1, \ldots, \gamma_m\}) \cup \{\eta\}$.

The following example illustrates an application of rule R5.

Example 5. Suppose that the following clauses belong to P_k:

γ_1: $h(X) \leftarrow X \geq 1 \wedge Y = X - 1 \wedge p(Y, 1)$
γ_2: $h(X) \leftarrow X \geq 1 \wedge Y = X + 1 \wedge \neg q(Y)$

and suppose that the following clauses constitute the definition of a predicate *new* in Defs_k:

δ_1: $\mathit{new}(Z, C) \leftarrow V = Z - C \wedge p(V, C)$
δ_2: $\mathit{new}(Z, C) \leftarrow V = Z + C \wedge \neg q(V)$

For $\vartheta = \{V/Y, Z/X, C/1\}$, we have that $\gamma_1 = h(X) \leftarrow X \geq 1 \wedge (V = Z - C \wedge p(V, C))\vartheta$ and $\gamma_2 = h(X) \leftarrow X \geq 1 \wedge (V = Z + C \wedge \neg q(V))\vartheta$, and the substitution ϑ satisfies Conditions (i) and (ii) of the positive folding rule. By folding clauses γ_1 and γ_2 using clauses δ_1 and δ_2 we derive:

η: $h(X) \leftarrow X \geq 1 \wedge new(Z, 1)$

R6. Negative Folding. Let γ be a clause in P_k and let $Defs'_k$ be a variant of $Defs_k$ without common variables with γ. Suppose that there exists a predicate in $Defs'_k$ whose definition consists of a single clause $\delta : K \leftarrow d \wedge A$, where A is an atom. Suppose also that there exists a substitution ϑ such that clause γ is of the form: $H \leftarrow c \wedge d\vartheta \wedge G_L \wedge \neg A\vartheta \wedge G_R$ and $FV(K) = FV(d \wedge A)$.

By *folding clause* γ *using clause* δ we derive the clause η: $H \leftarrow c \wedge d\vartheta \wedge G_L \wedge \neg K\vartheta \wedge G_R$ and from program P_k we derive the program $P_{k+1} = (P_k - \{\gamma\}) \cup \{\eta\}$.

The following is an example of application of the negative folding rule.

Example 6. Let the following clause belong to P_k:

γ: $h(X) \leftarrow X \geq 0 \wedge q(X) \wedge \neg r(X, 0)$

and let *new* be a predicate whose definition in $Defs_k$ consists of the clause:

δ: $new(X, C) \leftarrow X \geq C \wedge r(X, C)$

By folding γ using δ we derive:

η: $h(X) \leftarrow X \geq 0 \wedge q(X) \wedge \neg new(X, 0)$

The positive and negative folding rule are not fully symmetric for the following three reasons.

(1) By positive folding we can fold several clauses at a time by using *several* clauses whose body may contain several literals, while by negative folding we can fold a *single* clause at a time by using a single clause whose body contains precisely one atom. This is motivated by the fact that a conjunction of more than one literal cannot occur inside negation in the body of a clause.

(2) By positive folding, for $i = 1, \ldots, m$, the constraint $d\vartheta_i$ occurring in the body of clause γ_i is removed, while by negative folding the constraint $d\vartheta$ occurring in the body of clause γ is not removed. Indeed, the removal of the constraint $d\vartheta$ would be incorrect. For instance, let us consider the program P_k of Example 6 above and let us assume that γ is the only clause defining the predicate h. Let us also assume that the predicates q and r are defined by the following two clauses: $q(X) \leftarrow X < 0$ and $r(X, 0) \leftarrow X < 0$. We have that $h(-1) \notin M(P_k)$. Suppose that we apply the negative folding rule to clause γ and we remove the constraint $X \geq 0$, thereby deriving the clause $h(X) \leftarrow q(X) \wedge \neg new(X, 0)$, instead of clause η. Then we obtain a program whose perfect model has the atom $h(-1)$.

(3) The conditions on the variables occurring in the clauses used for folding are less restrictive in the case of positive folding (see Conditions (i) and (ii) of R5) than in the case of negative folding (see the condition $FV(K) = FV(d \wedge A)$). Notice that a negative folding rule where the condition $FV(K) = FV(d \wedge A)$ is replaced by Conditions (i) and (ii) of R5 would be incorrect, in general. To see this, let us consider the following example which may be viewed as the inverse derivation of Example 4.

Example 7. Let us consider the following programs P_0, P_1, and P_2, where P_1 is obtained from P_0 by definition introduction, and P_2 is obtained from P_1 by

incorrectly folding $p \leftarrow \neg r(X)$ using $q \leftarrow r(Y)$. Notice that $FV(q) \neq FV(r(X))$ but Conditions (i) and (ii) are satisfied by the substitution $\{Y/X\}$.

$$
\begin{array}{lll}
P_0\colon\ p \leftarrow \neg r(X) & P_1\colon\ p \leftarrow \neg r(X) & P_2\colon\ p \leftarrow \neg q \\
\quad\ r(X) \leftarrow X{=}0 & \quad\ r(X) \leftarrow X{=}0 & \quad\ r(X) \leftarrow X{=}0 \\
 & \quad\ q \leftarrow r(Y) & \quad\ q \leftarrow r(Y)
\end{array}
$$

We have that: $p \in M(P_0)$ while $p \notin M(P_2)$. (We assume that the carrier of the interpretation for the constraints contains at least one element different from 0.)

If we consider the folding and unfolding rules outside the context of a transformation sequence, either rule can be viewed as the inverse of the other. However, given a transformation sequence $P_0, \ldots, P_n$, it may be the case that from a program P_k in that sequence we derive program P_{k+1} by folding, and from program P_{k+1} we *cannot* derive by unfolding a program P_{k+2} which is equal to P_k. This is due to the fact that in the transformation sequence $P_0, \ldots, P_k, P_{k+1}$, in order to fold some clauses in program P_k, we may use clauses in $Defs_k$ which are neither in P_k nor in P_{k+1}, while for unfolding program P_{k+1} we can only use clauses which belong to P_{k+1}. Thus, according to the terminology introduced in [29], we say that folding is, in general, not *reversible*. This fact is shown by the following example.

Example 8. Let us consider the transformation sequence:

$$
\begin{array}{llll}
P_0\colon\ p \leftarrow q & P_1\colon\ p \leftarrow q & P_2\colon\ p \leftarrow q & P_3\colon\ p \leftarrow r \\
\quad\ q \leftarrow & \quad\ q \leftarrow & \quad\ q \leftarrow & \quad\ q \leftarrow \\
 & \quad\ r \leftarrow q & \quad\ r \leftarrow & \quad\ r \leftarrow
\end{array}
$$

where P_1 is derived from P_0 by introducing the definition $r \leftarrow q$, P_2 is derived from P_1 by unfolding the clause $r \leftarrow q$, and P_3 is derived from P_2 by folding the clause $p \leftarrow q$ using the definition $r \leftarrow q$. We have that from program P_3 we cannot derive a program equal to P_2 by applying the positive unfolding rule.

Similarly, the unfolding rules are not reversible in general. In fact, if we derive a program P_{k+1} by unfolding a clause in a program P_k and we have that $Defs_k = \emptyset$, then we cannot apply the folding rule and derive a program P_{k+2} which is equal to P_k, simply because no clause in $Defs_k$ is available for folding.

The following *replacement* rule can be applied to replace a set of clauses with a new set of clauses by using laws based on equivalences between formulas. In particular, we consider: (i) boolean laws, (ii) equivalences that can be proved in the chosen interpretation $\mathcal{D}$ for the constraints, and (iii) properties of the equality predicate.

R7. Replacement Based on Laws. Let us consider the following rewritings $\Gamma_1 \Rightarrow \Gamma_2$ between sets of clauses (we use $\Gamma_1 \Leftrightarrow \Gamma_2$ as a shorthand for the two rewritings $\Gamma_1 \Rightarrow \Gamma_2$ and $\Gamma_2 \Rightarrow \Gamma_1$). Each rewriting is called a *law*.

Boolean Laws

(1) $\{H \leftarrow c \wedge A \wedge \neg A \wedge G\} \quad \Leftrightarrow \emptyset$

(2) $\{H \leftarrow c \wedge H \wedge G\} \quad \Leftrightarrow \emptyset$

(3) $\{H \leftarrow c \wedge G_1 \wedge A_1 \wedge A_2 \wedge G_2\} \quad \Leftrightarrow \{H \leftarrow c \wedge G_1 \wedge A_2 \wedge A_1 \wedge G_2\}$

(4) $\{H \leftarrow c \wedge A \wedge A \wedge G\} \quad \Rightarrow \{H \leftarrow c \wedge A \wedge G\}$

(5) $\{H \leftarrow c \wedge G_1, \; H \leftarrow c \wedge d \wedge G_1 \wedge G_2\} \quad \Leftrightarrow \{H \leftarrow c \wedge G_1\}$

(6) $\{H \leftarrow c \wedge A \wedge G, \; H \leftarrow c \wedge \neg A \wedge G\} \quad \Rightarrow \{H \leftarrow c \wedge G\}$

Laws of Constraints

(7) $\{H \leftarrow c \wedge G\} \Leftrightarrow \emptyset$
if the constraint c is unsatisfiable, that is, $\mathcal{D} \models \neg\exists(c)$

(8) $\{H \leftarrow c_1 \wedge G\} \Leftrightarrow \{H \leftarrow c_2 \wedge G\}$
if $\mathcal{D} \models \forall (\exists Y\, c_1 \leftrightarrow \exists Z\, c_2)$, where:
(i) $Y = FV(c_1) - FV(\{H, G\})$, and
(ii) $Z = FV(c_2) - FV(\{H, G\})$

(9) $\{H \leftarrow c \wedge G\} \Leftrightarrow \{H \leftarrow c_1 \wedge G, \; H \leftarrow c_2 \wedge G\}$
if $\mathcal{D} \models \forall (c \leftrightarrow (c_1 \vee c_2))$

Laws of Equality

(10) $\{(H \leftarrow c \wedge G)\{X/t\}\} \Leftrightarrow \{H \leftarrow X = t \wedge c \wedge G\}$
if the variable X does not occur in the term t and t is free for X in c.

Let Γ_1 and Γ_2 be sets of clauses such that: (i) $\Gamma_1 \Rightarrow \Gamma_2$, and (ii) Γ_2 is locally stratified w.r.t. the fixed local stratification function σ. By *replacement* from Γ_1 we derive Γ_2 and from program P_k we derive the program $P_{k+1} = (P_k - \Gamma_1) \cup \Gamma_2$.

Condition (ii) on Γ_2 is needed because a replacement based on laws (1), (2), (5), and (7), used from right to left, may not preserve local stratification. For instance, the first law may be used to introduce a clause of the form $p \leftarrow p \wedge \neg p$, which is not locally stratified. We will see at the end of Section 4 that if we add the reverse versions of the boolean laws (4) or (6), then the correctness result stated in Theorem 3 does not hold.

The following definition is needed for stating rule R8 below. The set of *useless predicates* in a program P is the maximal set U of predicate symbols occurring in P such that a predicate p is in U iff every clause γ in $Def(p, P)$ is of the form $H \leftarrow c \wedge G_1 \wedge q(\ldots) \wedge G_2$ for some q in U. For example, in the following program:

$p(X) \leftarrow q(X) \wedge \neg r(X)$
$q(X) \leftarrow p(X)$
$r(X) \leftarrow X > 0$

p and q are useless predicates, while r is not useless.

R8. Deletion of Useless Predicates. If p is a useless predicate in P_k, then from program P_k we derive the program $P_{k+1} = P_k - Def(p, P_k)$.

Neither of the rules R2 and R8 subsumes the other. Indeed, on one hand the definition of a predicate p on which no predicate of interest depends, can be deleted by rule R2 even if p is not useless. On the other hand, the definition of a useless predicate p can be deleted by rule R8 even if a predicate of interest depends on p.

The *constraint addition* rule R9 which we present below, can be applied to add to the body of a clause a constraint which is implied by that body. Conversely, the *constraint deletion* rule R10, also presented below, can be applied to delete from the body of a clause a constraint which is implied by the rest of the body. Notice that these implications should hold in the perfect model of program P_k, while the applicability conditions of rule R7 (see, in particular, the replacements based on laws 7–9) are independent of P_k. Thus, for checking the applicability conditions of rules R9 and R10 we may need a program analysis based, for instance, on abstract interpretation [10].

R9. Constraint Addition. Let $\gamma_1 : \ H \leftarrow c \wedge G$ be a clause in P_k and let d be a constraint such that $M(P_k) \models \forall((c \wedge G) \rightarrow \exists X\, d)$, where $X = FV(d) - FV(\gamma_1)$. By *constraint addition* from clause γ_1 we derive the clause $\gamma_2 : \ H \leftarrow c \wedge d \wedge G$ and from program P_k we derive the program $P_{k+1} = (P_k - \{\gamma_1\}) \cup \{\gamma_2\}$.

The following example shows an application of the constraint addition rule that cannot be realized by applying laws of constraints according to rule R7.

Example 9. Let us consider the following program P_k:

1. $nat(0) \leftarrow$
2. $nat(N) \leftarrow N = M+1 \wedge nat(M)$

Since $M(P_k) \models \forall M\, (nat(M) \rightarrow M \geq 0)$, we can add the constraint $M \geq 0$ to the body of clause 2. This constraint addition improves the termination of the program when using a top-down strategy.

R10. Constraint Deletion. Let $\gamma_1 : \ H \leftarrow c \wedge d \wedge G$ be a clause in P_k and let d be a constraint such that $M(P_k) \models \forall((c \wedge G) \rightarrow \exists X\, d)$, where $X = FV(d) - FV(H \leftarrow c \wedge G)$. Suppose that the clause $\gamma_2 : \ H \leftarrow c \wedge G$ is locally stratified w.r.t. the fixed σ. By *constraint deletion* from clause γ_1 we derive clause γ_2 and from program P_k we derive the program $P_{k+1} = (P_k - \{\gamma_1\}) \cup \{\gamma_2\}$.

We assume that γ_2 is locally stratified w.r.t. σ because otherwise, the constraint deletion rule may not preserve local stratification. For instance, let us consider the following program P:

$p(X) \leftarrow$
$p(X) \leftarrow X \neq X \wedge \neg p(X)$

P is locally stratified because for all elements d in the carrier of the interpretation $\mathcal{D}$ for the constraints, we have that $\mathcal{D} \models d = d$. We also have that $M(P) \models \forall X\, (\neg p(X) \rightarrow X \neq X)$. However, if we delete the constraint $X \neq X$ from the second clause of P we derive the clause $p(X) \leftarrow \neg p(X)$ which is not locally stratified w.r.t. any local stratification function.

4 Preservation of Perfect Models

In this section we present some sufficient conditions which ensure that a transformation sequence constructed by applying the transformation rules listed in Section 3, preserves the perfect model semantics.

We will prove our correctness theorem for *admissible* transformation sequences, that is, transformation sequences constructed by applying the rules according to suitable restrictions. The reader who is familiar with the program transformation methodology, will realize that most transformation strategies can, indeed, be realized by means of admissible transformation sequences. In particular, all examples of Section 5 are worked out by using this kind of transformation sequences.

We proceed as follows. (i) First we show that the transformation rules preserve local stratification. (ii) Then we introduce the notion of an *admissible* transformation sequence. (iii) Next we introduce the notion of a *proof tree* for a ground atom A and a program P and we show that $A \in M(P)$ iff there exists a proof tree for A and P. Thus, the notion of proof tree provides the operational counterpart of the perfect model semantics. (iv) Then, we prove that given any admissible transformation sequence $P_0, \ldots, P_n$, any set $Pred_{int}$ of predicates of interest, and any ground atom A whose predicate is in $Pred_{int}$, we have that for $k = 0, \ldots, n$, there exists a proof tree for A and P_k iff there exists a proof tree for A and $P_0 \cup Defs_n$. (v) Finally, by using the property of proof trees considered at Point (iii), we conclude that an admissible transformation sequence preserves the perfect model semantics (see Theorem 3).

Let us start off by showing that the transformation rules preserve the local stratification function σ which was fixed for the initial program P_0 at the beginning of the construction of the transformation sequence.

Proposition 1. [Preservation of Local Stratification]. *Let P_0 be a locally stratified program, let $\sigma : \mathcal{B}_{\mathcal{D}} \to W$ be a local stratification function for P_0, and let $P_0, \ldots, P_n$ be a transformation sequence using σ. Then the programs $P_0, \ldots, P_n$, and $P_0 \cup Defs_n$ are locally stratified w.r.t. σ.*

The proof of Proposition 1 is given in Appendix A.

An admissible transformation sequence is a transformation sequence that satisfies two conditions: (1) every clause used for positive folding is unfolded w.r.t. a positive literal, and (2) the definition elimination rule cannot be applied before any other transformation rule. An admissible transformation sequence is formally defined as follows.

Definition 2. [Admissible Transformation Sequence] *A transformation sequence $P_0, \ldots, P_n$ is said to be* admissible *iff the following two conditions hold:*
(1) *for $k = 0, \ldots, n-1$, if P_{k+1} is derived from P_k by applying the positive folding rule to clauses $\gamma_1, \ldots, \gamma_m$ using clauses $\delta_1, \ldots, \delta_m$, then for $i = 1, \ldots, m$ there exists j, with $0 < j < n$, such that $\delta_i \in P_j$ and program P_{j+1} is derived from P_j by positive unfolding of clause δ_i, and*

(2) *if for some* $m<n$, P_{m+1} *is derived from* P_m *by the definition elimination rule then for all* $k = m, \ldots, n-1$, P_{k+1} *is derived from* P_k *by applying the definition elimination rule.*

When proving our correctness theorem (see Theorem 3 below), we will find it convenient to consider transformation sequences which are admissible and satisfy some extra suitable properties. This motivates the following notion of *ordered* transformation sequences.

Definition 3. [Ordered Transformation Sequence] *A transformation sequence* $P_0, \ldots, P_n$ *is said to be* ordered *iff it is of the form:*

$P_0, \ldots, P_i, \ldots, P_j, \ldots, P_m, \ldots, P_n$

where:
(1) *the sequence* $P_0, \ldots, P_i$, *with* $i \geq 0$, *is constructed by applying* i *times the definition introduction rule, that is,* $P_i = P_0 \cup Defs_i$;
(2) *the sequence* $P_i, \ldots, P_j$ *is constructed by unfolding w.r.t. a positive literal each clause in* $Defs_i$ *which is used for applications of the folding rule in* $P_j, \ldots, P_m$;
(3) *the sequence* $P_j, \ldots, P_m$, *with* $j \leq m$, *is constructed by applying any rule, except the definition introduction and definition elimination rules; and*
(4) *the sequence* $P_m, \ldots, P_n$, *with* $m \leq n$, *is constructed by applying the definition elimination rule.*

Notice that in an ordered transformation sequence we have that $Defs_i = Defs_n$. Every ordered transformation sequence is admissible, because of Points (2) and (4) of Definition 3. Conversely, by the following Proposition 2, in our correctness proofs we will assume, without loss of generality, that any admissible transformation sequence is ordered.

Proposition 2. *For every admissible transformation sequence* $P_0, \ldots, P_n$, *there exists an ordered transformation sequence* $Q_0, \ldots, Q_r$ *(with* r *possibly different from* n*), such that: (i)* $P_0 = Q_0$, *(ii)* $P_n = Q_r$, *and (iii) the set of definitions introduced during* $P_0, \ldots, P_n$ *is equal to the set of definitions introduced during* $Q_0, \ldots, Q_r$.

The easy proof of Proposition 2 is omitted for reasons of space. It is based on the fact that the applications of some transformation rules can be suitably rearranged without changing the initial and final programs in a transformation sequence.

Now we present the operational counterpart of the perfect model semantics, that is, the notion of a proof tree. A proof tree for a ground atom A and a locally stratified program P is constructed by transfinite induction as indicated in the following definition.

Definition 4. [Proof Tree] *Let* A *be a ground atom,* P *be a locally stratified program, and* σ *be any local stratification for* P. *Let* $PT_{<A}$ *be the set of proof trees for ground atoms* B *and* P *with* $\sigma(B) < \sigma(A)$. *A* proof tree *for* A *and* P *is a finite tree* T *of goals such that: (i) the root of* T *is* A, *(ii) a node* N *of* T *has*

children $L_1, \ldots, L_r$ iff N is a ground atom B and there exists a clause $\gamma \in P$ and a valuation v such that $v(\gamma)$ is $B \leftarrow c \wedge L_1 \wedge \ldots \wedge L_r$ and $\mathcal{D} \models c$, and (iii) every leaf of T is either the empty conjunction true or a negated ground atom $\neg B$ such that there is no proof tree for B and P in $PT_{<A}$.

The following theorem establishes that the operational semantics based on proof trees is equivalent to the perfect model semantics.

Theorem 2. [Proof Trees and Perfect Models] *Let P be a locally stratified program. For all ground atoms A, there exists a proof tree for A and P iff $A \in M(P)$.*

Our proofs of correctness use induction w.r.t. suitable *well-founded measures* over proof trees, ground atoms, and ground goals (see, in particular, the proofs of Propositions 3 and 5 in Appendices B and C). We now introduce these measures.

Let T be a proof tree for a ground atom A and a locally stratified program P. By $size(T)$ we denote the number of atoms occurring at non-leaf nodes of T. For any ground atom A, locally stratified program P, and local stratification σ for P, we define the following measure:

$\mu(A, P) = min_{lex}\{\langle \sigma(A), size(T)\rangle \mid T \text{ is a proof tree for } A \text{ and } P\}$

where min_{lex} denotes the minimum w.r.t. the lexicographic ordering $<_{lex}$ over $W \times N$, where W is the set of countable ordinals and N is the set of natural numbers. $\mu(A, P)$ is undefined if there is no proof tree for A and P. The measure μ is extended from ground atoms to ground literals as follows. Given a ground literal L, we define:

$\mu(L, P) =$ *if* L is an atom A *then* $\mu(A, P)$
else if L is a negated atom $\neg A$ *then* $\langle \sigma(A), 0\rangle$

Now we extend μ to ground goals. First, we introduce the binary, associative operation $\oplus : (W \times N)^2 \to (W \times N)$ defined as follows:

$\langle \alpha_1, m_1\rangle \oplus \langle \alpha_2, m_2\rangle = \langle max(\alpha_1, \alpha_2),\ m_1 + m_2\rangle$

Then, given a ground goal $L_1 \wedge \ldots \wedge L_n$, we define:

$\mu(L_1 \wedge \ldots \wedge L_n,\ P) = \mu(L_1, P) \oplus \ldots \oplus \mu(L_n, P)$

The measure μ is well-founded in the sense that there is no infinite sequence of ground goals $G_1, G_2, \ldots$ such that $\mu(G_1, P) > \mu(G_2, P) > \ldots$

In order to show that an ordered transformation sequence $P_0, \ldots, P_i, \ldots, P_j, \ldots, P_m, \ldots, P_n$ (where the meaning of the subscripts is the one of Definition 3) preserves the perfect model semantics, we will use Theorem 2 and we will show that, for $k = 0, \ldots, n$, given any ground atom A whose predicate belongs to the set $Pred_{int}$ of predicates of interest, there exists a proof tree for A and P_k iff there exists a proof tree for A and $P_0 \cup Defs_n$. Since $P_i = P_0 \cup Defs_n$, it is sufficient to show the following properties, for any ground atom A:

(P1) there exists a proof tree for A and P_i iff there exists a proof tree for A and P_j,

(P2) there exists a proof tree for A and P_j iff there exists a proof tree for A and P_m, and

(P3) if the predicate of A is in $Pred_{int}$, then there exists a proof tree for A and P_m iff there exists a proof tree for A and P_n.

Property P1 is proved by the following proposition.

Proposition 3. *Let P_0 be a locally stratified program and let $P_0, \ldots, P_i, \ldots, P_j, \ldots, P_m, \ldots, P_n$ be an ordered transformation sequence. Then there exists a proof tree for a ground atom A and P_i iff there exists a proof tree for A and P_j.*

The proof of Proposition 3 is given in Appendix B. It is a proof by induction on $\sigma(A)$ and on the size of the proof tree for A.

In order to prove the only-if part of Property P2, we will show a stronger invariant property based on the following consistency notion.

Definition 5. [P_j-consistency] *Let $P_0, \ldots, P_i, \ldots, P_j, \ldots, P_m, \ldots, P_n$ be an ordered transformation sequence, P_k be a program in this sequence, and A be a ground atom. We say that a proof tree T for A and P_k is* P_j-consistent *iff for every ground atom B and ground literals $L_1, \ldots, L_r$, if B is the father of $L_1, \ldots, L_r$ in T, then $\mu(B, P_j) > \mu(L_1 \wedge \ldots \wedge L_r, P_j)$.*

The invariant property is as follows: for every program P_k in the sequence $P_j, \ldots, P_m$, if there exists a P_j-consistent proof tree for A and P_j, then there exists a P_j-consistent proof tree for A and P_k.

It is important that P_j-consistency refers to the program P_j obtained by applying the positive unfolding rule to each clause that belongs to $Defs_i$ and is used in $P_j, \ldots, P_m$ for a folding step. Indeed, if the positive unfolding rule is not applied to a clause in $Defs_i$, and this clause is then used (possibly, together with other clauses) in a folding step, then the preservation of P_j-consistent proof trees may not be ensured and the transformation sequence may not be correct. This is shown by Example 1 of the Introduction where we assume that the first clause $p(X) \leftarrow \neg q(X)$ of P_0 has been added by the definition introduction rule in a previous step.

We have the following.

Proposition 4. *If there exists a proof tree for a ground atom A and program P_j then there exists a P_j-consistent proof tree for A and P_j.*

Proof. Let T be a proof tree for A and P_j such that $\langle \sigma(A), size(T) \rangle$ is minimal w.r.t. $<_{lex}$. Then T is P_j-consistent. □

Notice that in the proof of Proposition 4 we state the existence of a P_j-consistent proof tree for a ground atom A and program P_j without providing an effective method for constructing this proof tree. In fact, it should be noticed that no effective method can be given for constructing the minimal proof tree for a given atom and program, because the existence of such a proof tree is not decidable and not even semi-decidable.

By Proposition 4, in order to prove Property P2 it is enough to show the following Proposition 5.

Proposition 5. *Let P_0 be a locally stratified program and let $P_0, \ldots, P_i, \ldots, P_j, \ldots, P_m, \ldots, P_n$ be an ordered transformation sequence. Then, for every ground atom A we have that:*

(Soundness) *if there exists a proof tree for A and P_m, then there exists a proof tree for A and P_j, and*

(Completeness) *if there exists a P_j-consistent proof tree for A and P_j, then there exists a P_j-consistent proof tree for A and P_m.*

The proof of Proposition 5 is given in Appendix C.

In order to prove Property P3, it is enough to prove the following Proposition 6, which is a straightforward consequence of the fact that the existence of a proof tree for a ground atom with predicate p is determined only by the existence of proof trees for atoms with predicates on which p depends.

Proposition 6. *Let P be a locally stratified program and let $Pred_{int}$ be a set of predicates of interest. Suppose that program Q is derived from program P by eliminating the definition of a predicate q such that no predicate in $Pred_{int}$ depends on q. Then, for every ground atom A whose predicate is in $Pred_{int}$, there exists a proof tree for A and P iff there exists a proof tree for A and Q.*

Now, as a consequence of Propositions 1–6, and Theorem 2, we get the following theorem which ensures that an admissible transformation sequence preserves the perfect model semantics.

Theorem 3. [Correctness of Admissible Transformation Sequences] *Let P_0 be a locally stratified program and let $P_0, \ldots, P_n$ be an admissible transformation sequence. Let $Pred_{int}$ be the set of predicates of interest. Then $P_0 \cup Defs_n$ and P_n are locally stratified and for every ground atom A whose predicate belongs to $Pred_{int}$, $A \in M(P_0 \cup Defs_n)$ iff $A \in M(P_n)$.*

This theorem does not hold if we add to the boolean laws listed in rule R7 of Section 3 the inverse of law (4), as shown by the following example.

Example 10. Let us consider the following transformation sequence:

P_0: $p \leftarrow q \wedge q$ $\quad$ P_1: $p \leftarrow q$ $\quad$ P_2: $p \leftarrow q \wedge q$ $\quad$ P_3: $p \leftarrow p$
$\quad\quad q \leftarrow$ $\quad\quad\quad\quad q \leftarrow$ $\quad\quad\quad\quad q \leftarrow$ $\quad\quad\quad\quad q \leftarrow$

We assume that the clause for p in P_0 is added to P_0 by the definition introduction rule, so that it can be used for folding. Program P_1 is derived from P_0 by unfolding, program P_2 is derived from P_1 by replacement based on the reverse of law (4), and finally, program P_3 is derived by folding the first clause of P_2 using the first clause of P_0. We have that $p \in M(P_0)$, while $p \notin M(P_3)$.

Analogously, the reader may verify that Theorem 3 does not hold if we add to the boolean laws of rule R7 the inverse of law (6).

5 Examples of Use of the Transformation Rules

In this section we show some program derivations realized by applying the transformation rules of Section 3. These program derivations are examples of the following three techniques: (1) the *determinization* technique, which is used for deriving a deterministic program from a nondeterministic one [14,33], (2) the *program synthesis* technique, which is used for deriving a program from a first order logic specification (see, for instance, [18,41] and [6] in this book for a recent survey), and (3) the *program specialization* technique, which is used for deriving a specialized program from a given program and a given portion of its input data (see, for instance, [21] and [24] for a recent survey).

Although we will *not* provide in this paper any automatic transformation strategy, the reader may realize that in the examples we will present, there is a systematic way of performing the program derivations. In particular, we perform all derivations according to the repeated application of the following sequence of steps: (i) first, we consider some predicate definitions in the initial program or we introduce some new predicate definitions, (ii) then we unfold these definitions by applying the positive and, possibly, the negative unfolding rules, (iii) then we manipulate the derived clauses by applying the rules of replacement, constraint addition, and constraint deletion, and (iv) finally, we apply the folding rules. The final programs are derived by applying the definition elimination rule, and keeping only those clauses that are needed for computing the predicates of interest.

5.1 Determinization: Comparing Even and Odd Occurrences of a List

Let us consider the problem of checking whether or not, for any given list L of numbers, the following property $r(L)$ holds: every number occurring in L in an even position is greater or equal than every number occurring in L in an odd position. The locally stratified program *EvenOdd* shown below, solves the given problem by introducing a new predicate $p(L)$ which holds iff there is a pair $\langle X, Y\rangle$ of numbers such that X occurs in the the list L in an even position, Y occurs in L in an odd position, and $X < Y$. Thus, for any list L, the property $r(L)$ holds iff $p(L)$ does not hold.

EvenOdd:

1. $r(L) \leftarrow list(L) \wedge \neg p(L)$
2. $p(L) \leftarrow I \geq 1 \wedge J \geq 1 \wedge X < Y \wedge$
   $occurs(X, I, L) \wedge even(I) \wedge occurs(Y, J, L) \wedge \neg even(J)$
3. $even(X) \leftarrow X = 0$
4. $even(X+1) \leftarrow X \geq 0 \wedge \neg even(X)$
5. $occurs(X, I, [H|T]) \leftarrow I = 1 \wedge X = H$
6. $occurs(X, I+1, [H|T]) \leftarrow I \geq 1 \wedge occurs(X, I, T)$
7. $list([\,]) \leftarrow$
8. $list([H|T]) \leftarrow list(T)$

In this program $occurs(X, I, L)$ holds iff X is the I-th element (with $I \geq 1$) of the list L starting from the left. When executed by using SLDNF resolution, this *EvenOdd* program may generate, in a nondeterministic way, all possible pairs $\langle X, Y \rangle$, occurring in even and odd positions, respectively. This program has an $O(n^2)$ time complexity in the worst case, where n is the length of the input list.

We want to derive a more efficient *definite* program that can be executed in a *deterministic* way, in the sense that for every constrained goal $c \wedge A \wedge G$ derived from a given ground query by LD-resolution [3] there exists at most one clause $H \leftarrow d \wedge K$ such that $c \wedge A = H \wedge d$ is satisfiable.

To give a sufficient condition for determinism we need the following notion. We say that a variable X is a *local variable* of a clause γ iff $X \in FV(bd(\gamma)) - FV(hd(\gamma))$. The determinism of a program P can be ensured by the following syntactic conditions: (i) no clause in P has local variables and (ii) any two clauses $H_1 \leftarrow c_1 \wedge G_1$ and $H_2 \leftarrow c_2 \wedge G_2$ in P are *mutually exclusive*, that is, the constraint $H_1 = H_2 \wedge c_1 \wedge c_2$ is unsatisfiable.

Our derivation consists of two transformation sequences. The first sequence starts from the program made out of clauses 2–8 and derives a deterministic, definite program Q for predicate p. The second sequence starts from $Q \cup \{1\}$ and derives a deterministic, definite program $EvenOdd_{det}$ for predicate r.

Let us show the construction of the first transformation sequence. Since clause 2 has local variables, we want to transform it into a set of clauses that have no local variables and are mutually exclusive, and thus, they will constitute a deterministic, definite program. We start off by applying the positive unfolding rule to clause 2, followed by applications of the replacement rule based on laws of constraints and equality. We derive:

9. $p([A|L]) \leftarrow J \geq 1 \wedge Y < A \wedge occurs(Y, J, L) \wedge even(J+1)$
10. $p([A|L]) \leftarrow I \geq 1 \wedge J \geq 1 \wedge X < Y \wedge occurs(X, I, L) \wedge$
$even(I+1) \wedge occurs(Y, J, L) \wedge \neg even(J+1)$

Now, by applications of the positive unfolding rule, negative unfolding, and replacement rules, we derive the following clauses for p:

11. $p([A, B|L]) \leftarrow B < A$
12. $p([A, B|L]) \leftarrow B \geq A \wedge I \geq 1 \wedge X < A \wedge occurs(X, I, L) \wedge even(I)$
13. $p([A, B|L]) \leftarrow B \geq A \wedge I \geq 1 \wedge B < X \wedge occurs(X, I, L) \wedge \neg even(I)$
14. $p([A, B|L]) \leftarrow B \geq A \wedge I \geq 1 \wedge J \geq 1 \wedge X < Y \wedge occurs(X, I, L) \wedge even(I) \wedge$
$occurs(Y, J, L) \wedge \neg even(J)$

Notice that the three clauses 12, 13, and 14, are not mutually exclusive. In order to derive a deterministic program for p, we introduce the following new definition:

15. $new1(A, B, L) \leftarrow I \geq 1 \wedge X < A \wedge occurs(X, I, L) \wedge even(I)$
16. $new1(A, B, L) \leftarrow I \geq 1 \wedge B < X \wedge occurs(X, I, L) \wedge \neg even(I)$
17. $new1(A, B, L) \leftarrow I \geq 1 \wedge J \geq 1 \wedge X < Y \wedge occurs(X, I, L) \wedge even(I) \wedge$
$occurs(Y, J, L) \wedge \neg even(J)$

and we fold clauses 12, 13, and 14 by using the definition of $new1$, that is, clauses 15, 16, and 17. We derive:

18. $p([A, B|L]) \leftarrow B \geq A \wedge new1(A, B, L)$

Clauses 11 and 18 have no local variables and are mutually exclusive. We are left with the problem of deriving a deterministic program for the newly introduced predicate $new1$.

By applying the positive unfolding, negative unfolding, and replacement rules, from clauses 15, 16, and 17, we get:

19. $new1(A, B, [C|L]) \leftarrow B < C$
20. $new1(A, B, [C|L]) \leftarrow I \geq 1 \wedge B < X \wedge occurs(X, I, L) \wedge even(I)$
21. $new1(A, B, [C|L]) \leftarrow I \geq 1 \wedge X < A \wedge occurs(X, I, L) \wedge \neg even(I)$
22. $new1(A, B, [C|L]) \leftarrow I \geq 1 \wedge X < C \wedge occurs(X, I, L) \wedge \neg even(I)$
23. $new1(A, B, [C|L]) \leftarrow I \geq 1 \wedge J \geq 1 \wedge X < Y \wedge occurs(X, I, L) \wedge$
$\neg even(I) \wedge occurs(Y, J, L) \wedge even(J)$

In order to derive mutually exclusive clauses without local variables we first apply the replacement rule and derive sets of clauses corresponding to mutually exclusive cases, and then we fold each of these sets of clauses. We use the replacement rule based on law (5) and law (9) which is justified by the equivalence: $\forall X \forall Y(true \leftrightarrow X \geq Y \vee X < Y)$. We get:

24. $new1(A, B, [C|L]) \leftarrow B < C$
25. $new1(A, B, [C|L]) \leftarrow B \geq C \wedge A \geq C \wedge I \geq 1 \wedge B < X \wedge$
$occurs(X, I, L) \wedge even(I)$
26. $new1(A, B, [C|L]) \leftarrow B \geq C \wedge A \geq C \wedge I \geq 1 \wedge X < A \wedge$
$occurs(X, I, L) \wedge \neg even(I)$
27. $new1(A, B, [C|L]) \leftarrow B \geq C \wedge A \geq C \wedge I \geq 1 \wedge J \geq 1 \wedge X < Y \wedge$
$occurs(X, I, L) \wedge \neg even(I) \wedge$
$occurs(Y, J, L) \wedge even(J)$
28. $new1(A, B, [C|L]) \leftarrow B \geq C \wedge A < C \wedge I \geq 1 \wedge B < X \wedge$
$occurs(X, I, L) \wedge even(I)$
29. $new1(A, B, [C|L]) \leftarrow B \geq C \wedge A < C \wedge I \geq 1 \wedge X < C \wedge$
$occurs(X, I, L) \wedge \neg even(I)$
30. $new1(A, B, [C|L]) \leftarrow B \geq C \wedge A < C \wedge I \geq 1 \wedge J \geq 1 \wedge X < Y \wedge$
$occurs(X, I, L) \wedge \neg even(I) \wedge$
$occurs(Y, J, L) \wedge even(J)$

The three sets of clauses: {24}, {25, 26, 27}, and {28, 29, 30} correspond to the mutually exclusive cases: $(B < C)$, $(B \geq C \wedge A \geq C)$, and $(B \geq C \wedge A < C)$, respectively. Now, in order to fold each set {25, 26, 27} and {28, 29, 30} and derive mutually exclusive clauses without local variables, we introduce the following new definition:

31. $new2(A, B, L) \leftarrow I \geq 1 \wedge B < X \wedge occurs(X, I, L) \wedge even(I)$
32. $new2(A, B, L) \leftarrow I \geq 1 \wedge X < A \wedge occurs(X, I, L) \wedge \neg even(I)$
33. $new2(A, B, L) \leftarrow I \geq 1 \wedge J \geq 1 \wedge X < Y \wedge occurs(X, I, L) \wedge \neg even(I) \wedge$
$occurs(Y, J, L) \wedge even(J)$

By folding clauses 25, 26, 27 and 28, 29, 30 using clauses 31, 32, and 33, for predicate $new1$ we get the following mutually exclusive clauses without local variables:

34. $new1(A, B, [C|L]) \leftarrow B < C$
35. $new1(A, B, [C|L]) \leftarrow B \geq C \wedge A \geq C \wedge new2(A, B, L)$
36. $new1(A, B, [C|L]) \leftarrow B \geq C \wedge A < C \wedge new2(C, B, L)$

Unfortunately, the clauses for the new predicate $new2$ have local variables and are not mutually exclusive. Thus, we continue our derivation and, by applying the positive unfolding, negative unfolding, replacement, and folding rules, from clauses 31, 32, and 33 we derive the following clauses (this derivation is similar to the derivation that lead from {15, 16, 17} to {34, 35, 36} and we omit it):

37. $new2(A, B, [C|L]) \leftarrow C < A$
38. $new2(A, B, [C|L]) \leftarrow C \geq A \wedge B \geq C \wedge new1(A, C, L)$
39. $new2(A, B, [C|L]) \leftarrow C \geq A \wedge B < C \wedge new1(A, B, L)$

The set of clauses derived so far starting from the initial clause 2, that is, {11, 18, 34, 35, 36, 37, 38, 39} constitutes a deterministic program for p, call it Q.

Now we construct the second transformation sequence starting from $Q \cup \{1\}$ for deriving a deterministic, *definite* program for r. We start off by considering clause 1 which defines r and, by positive unfolding, negative unfolding, and replacement we derive:

40. $r([\,]) \leftarrow$
41. $r([A]) \leftarrow$
42. $r([A, B|L]) \leftarrow list(L) \wedge B \geq A \wedge \neg new1(A, B, L)$

By introducing the following definition:

43. $new3(A, B, L) \leftarrow list(L) \wedge B \geq A \wedge \neg new1(A, B, L)$

and then folding clause 42 using clause 43, we derive the following definite clauses:

44. $r([\,]) \leftarrow$
45. $r([A]) \leftarrow$
46. $r([A, B|L]) \leftarrow B \geq A \wedge new3(A, B, L)$

Now, we want to transform clause 43 into a set of definite clauses. By positive unfolding, negative unfolding, and replacement, from clause 43 we derive:

47. $new3(A, B, [\,]) \leftarrow B \geq A$
48. $new3(A, B, [C|L]) \leftarrow B \geq C \wedge A < C \wedge list(L) \wedge B \geq C \wedge \neg new2(C, B, L)$
49. $new3(A, B, [C|L]) \leftarrow B \geq C \wedge A \geq C \wedge list(L) \wedge B \geq A \wedge \neg new2(A, B, L)$

In order to transform clauses 48 and 49 into definite clauses, we introduce the following definition:

50. $new4(A, B, L) \leftarrow list(L) \wedge B \geq A \wedge \neg new2(A, B, L)$

and we fold clauses 48 and 49 using clause 50. We get:

51. $new3(A, B, [\,]) \leftarrow B \geq A$
52. $new3(A, B, [C|L]) \leftarrow B \geq C \wedge A < C \wedge new4(C, B, L)$
53. $new3(A, B, [C|L]) \leftarrow B \geq C \wedge A \geq C \wedge new4(A, B, L)$

Now we are left with the task of transforming clause 50 into a set of definite clauses. By applying the positive unfolding, negative unfolding, replacement, and folding rules, we derive:

54. $new4(A, B, [\,]) \leftarrow B \geq A$
55. $new4(A, B, [C|L]) \leftarrow B < C \wedge C \geq A \wedge new3(A, B, L)$
56. $new4(A, B, [C|L]) \leftarrow B \geq C \wedge C \geq A \wedge new3(A, C, L)$

Finally, by eliminating the definitions of the predicates on which r does not depend, we get, as desired, the following final program which is a deterministic, definite program.

EvenOdd$_{det}$*:*

44. $r([\,]) \leftarrow$
45. $r([A]) \leftarrow$
46. $r([A, B|L]) \leftarrow B \geq A \wedge new3(A, B, L)$
51. $new3(A, B, [\,]) \leftarrow B \geq A$
52. $new3(A, B, [C|L]) \leftarrow B \geq C \wedge A < C \wedge new4(C, B, L)$
53. $new3(A, B, [C|L]) \leftarrow B \geq C \wedge A \geq C \wedge new4(A, B, L)$
54. $new4(A, B, [\,]) \leftarrow B \geq A$
55. $new4(A, B, [C|L]) \leftarrow B < C \wedge C \geq A \wedge new3(A, B, L)$
56. $new4(A, B, [C|L]) \leftarrow B \geq C \wedge C \geq A \wedge new3(A, C, L)$

Given a list of numbers L of length n, the *EvenOdd*$_{det}$ program checks that $r(L)$ holds by performing at most $2n$ comparisons between numbers occurring in L. Program *EvenOdd*$_{det}$ works by traversing the input list L only once (without backtracking) and storing, for every initial portion L_1 of the input list L, the maximum number A occurring in an odd position of L_1 and the minimum number B occurring in an even position of L_1 (see the first two arguments of the predicates $new3$ and $new4$). When looking at the first element C of the portion of the input list still to be visited (i.e., the third argument of $new3$ or $new4$), the following two cases are possible: either (Case 1) the element C occurs in an odd position of the input list L, i.e., a call of the form $new3(A, B, [C|L_2])$ is executed, or (Case 2) the element C occurs in an even position of the input list L, i.e., a call of the form $new4(A, B, [C|L_2])$ is executed. In Case (1) program *EvenOdd*$_{det}$ checks that $B \geq C$ holds and then updates the value of the maximum number occurring in an odd position with the maximum between A and C. In Case (2) program *EvenOdd*$_{det}$ checks that $C \geq A$ holds and then updates the value of the minimum number occurring in an even position with the minimum between B and C.

5.2 Program Synthesis: The N-queens Problem

The N-queens problem has been often considered in the literature for presenting various programming techniques, such as recursion and backtracking. We consider it here as an example of the program synthesis technique, as it has been done in [41]. Our derivation is different from the one presented in [41], because the derivation in [41] makes use of the unfold/fold transformation rules for definite programs together with an *ad hoc* transformation rule (called *negation technique*) for transforming general programs (with negation) into definite programs. In contrast, we use unfold/fold transformation rules for general programs, and in particular, our negative unfolding rule of Section 3.

The N-queens problem can be informally specified as follows. We are required to place $N(\geq 0)$ queens on an $N \times N$ chess board, so that no two queens attack each other, that is, they do not lie on the same row, column, or diagonal. A board configuration with this property is said to be *safe*. By using the fact that no two queens should lie on the same row, we represent an $N \times N$ chess board as a list L of N positive integers: the k-th element on L represents the column of the queen on row k.

In order to give a formal specification of the N-queens problem we follow the approach presented in [32], which is based on first order logic. We introduce the following constraint logic program:

P: $nat(0) \leftarrow$
$nat(N) \leftarrow N = M+1 \wedge M \geq 0 \wedge nat(M)$
$nat\_list([\,]) \leftarrow$
$nat\_list([H|T]) \leftarrow nat(H) \wedge nat\_list(T)$
$length([\,], 0) \leftarrow$
$length([H|T], N) \leftarrow N = M+1 \wedge M \geq 0 \wedge length(T, M)$
$member(X, [H|T]) \leftarrow X = H$
$member(X, [H|T]) \leftarrow member(X, T)$
$in\_range(X, M, N) \leftarrow X = N \wedge M \leq N$
$in\_range(X, M, N) \leftarrow N = K+1 \wedge M \leq K \wedge in\_range(X, M, K)$
$occurs(X, I, [H|T]) \leftarrow I = 1 \wedge X = H$
$occurs(X, I+1, [H|T]) \leftarrow I \geq 1 \wedge occurs(X, I, T)$

and the following first order formula:

$$\begin{aligned}
\varphi(N, L) : \; & nat(N) \wedge nat_list(L) \wedge & (1)\\
& length(L, N) \wedge \forall X\,(member(X, L) \rightarrow in_range(X, 1, N)) \wedge & (2)\\
& \forall A,B,K,M\,((1 \leq K \wedge K \leq M \wedge occurs(A, K, L) \wedge occurs(B,M,L)) & (3)\\
& \qquad \rightarrow (A \neq B \wedge A - B \neq M - K \wedge B - A \neq M - K)) & (4)
\end{aligned}$$

In the above program and formula $in\_range(X, M, N)$ holds iff $X \in \{M, M+1, \ldots, N\}$ and $N \geq 0$. The other predicates have been defined in previous programs or do not require explanation. Now we define the relation $queens(N, L)$ where N is a nonnegative integer and L is a list of positive integers, as follows:

$$queens(N, L) \;\text{ iff }\; M(P) \models \varphi(N, L)$$

Line (2) of the formula $\varphi(N, L)$ above specifies a chess board as a list of N integers each of which is in the range $[1, \ldots, N]$. If $N = 0$ the list is empty. Lines (3) and (4) of $\varphi(N, L)$ specify the safety property of board configurations. Now, we would like to derive a constraint logic program R which computes the relation $queens(N, L)$, that is, R should define a predicate $queens(N, L)$ such that:

$$(\pi) \qquad M(R) \models queens(N, L) \;\text{ iff }\; M(P) \models \varphi(N, L)$$

Following the approach presented in [32], we start from the formula (called a *statement*) $queens(N, L) \leftarrow \varphi(N, L)$ and, by applying a variant of the *Lloyd-Topor transformation* [26], we derive the following stratified logic program:

F : 1. $queens(N,L) \leftarrow nat(N) \wedge nat\_list(L) \wedge length(L,N) \wedge$
$\neg aux1(L,N) \wedge \neg aux2(L)$
2. $aux1(L,N) \leftarrow member(X,L) \wedge \neg in\_range(X,1,N)$
3. $aux2(L) \leftarrow 1 \leq K \wedge K \leq M \wedge$
$\neg(A \neq B \wedge A - B \neq M - K \wedge B - A \neq M - K) \wedge$
$occurs(A,K,L) \wedge occurs(B,M,L)$

This variant of the Lloyd-Topor transformation is a fully automatic transformation, but it cannot be performed by using our transformation rules, because it operates on first order formulas. It can be shown that this variant of the Lloyd-Topor transformation preserves the perfect model semantics and, thus, we have that: $M(P \cup F) \models queens(N,L)$ iff $M(P) \models \varphi(N,L)$.

The derived program $P \cup F$ is not very satisfactory from a computational point of view because, when using SLDNF resolution with the left-to-right selection rule, it may not terminate for calls of the form $queens(n,L)$ where n is a nonnegative integer and L is a variable. Thus, the process of program synthesis proceeds by applying the transformation rules listed in Section 3, thereby transforming program $P \cup F$ into a program R such that: (i) Property (π) holds, (ii) R is a definite program, and (iii) R terminates for all calls of the form $queens(n,L)$, where n is any nonnegative integer and L is a variable. Actually, the derivation of the final program R is performed by constructing two transformation sequences: (i) a first one, which starts from the initial program P, introduces clauses 2 and 3 by definition introduction, and ends with a program Q, and (ii) a second one, which starts from program Q, introduces clause 1 by definition introduction, and ends with program R.

We will illustrate the application of the transformation rules for deriving program R without discussing in detail how this derivation can be performed in an automatic way using a particular strategy. As already mentioned, the design of suitable transformation strategies for the automation of program derivations for constraint logic programs, is beyond the scope of the present paper.

The program transformation process starts off from program $P \cup \{2,3\}$ by transforming clauses 2 and 3 into a set of clauses without local variables, so that they can be subsequently used for unfolding clause 1 w.r.t. $\neg aux1(L,N)$ and $\neg aux2(L)$ (see the negative unfolding rule R4).

By positive unfolding, replacement, and positive folding, from clause 2 we derive:

4. $aux1([H|T],N) \leftarrow \neg in\_range(H,1,N)$
5. $aux1([H|T],N) \leftarrow aux1(T,N)$

Similarly, by positive unfolding, replacement, and positive folding, from clause 3 we derive:

6. $aux2([A|T]) \leftarrow M \geq 1 \wedge \neg(A \neq B \wedge A - B \neq M \wedge B - A \neq M) \wedge$
$occurs(B,M,T)$
7. $aux2([A|T]) \leftarrow aux2(T)$

In order to eliminate the local variables B and M occurring in clause 6, by the definition introduction rule we introduce the following new clause, whose body is a generalization of the body of clause 6:

8. $new1(A,T,J) \leftarrow M \geq 1 \wedge \neg(A \neq B \wedge A - B \neq M + J \wedge B - A \neq M + J) \wedge$
$occurs(B,M,T)$

By replacement and positive folding, from clause 6 we derive:

6f. $aux2([A|T]) \leftarrow new1(A,T,0)$

Now, by positive unfolding, replacement, and positive folding, from clause 8 we derive:

9. $new1(A,[B|T],K) \leftarrow \neg(A \neq B \wedge A - B \neq K + 1 \wedge B - A \neq K + 1)$
10. $new1(A,[B|T],K) \leftarrow new1(A,T,K+1)$

The program, call it Q, derived so far is $P \cup \{4, 5, 6f, 7, 9, 10\}$, and clauses 4, 5, 6f, 7, 9, and 10 have no local variables.

Now we construct a new transformation sequence which takes Q as initial program. We start off by applying the definition introduction rule and adding clause 1 to program Q. Our objective is to transform clause 1 into a set of definite clauses. We first apply the definition rule and we introduce the following clause, whose body is a generalization of the body of clause 1:

11. $new2(N,L,K) \leftarrow nat(M) \wedge nat\_list(L) \wedge length(L,M) \wedge$
$\neg aux1(L,N) \wedge \neg aux2(L) \wedge N = M + K$

By replacement and positive folding, from clause 11 we derive:

1f. $queens(N,L) \leftarrow new2(N,L,0)$

By positive and negative unfolding, replacement, constraint addition, and positive folding, from clause 11 we derive:

12. $new2(N,[\,],K) \leftarrow N = K$
13. $new2(N,[H|T],K) \leftarrow N \geq K + 1 \wedge new2(N,T,K+1) \wedge$
$nat(H) \wedge nat\_list(T) \wedge in\_range(H,1,N) \wedge$
$\neg new1(H,T,0)$

In order to derive a definite program we introduce a new predicate $new3$ defined by the following clause:

14. $new3(A,T,N,M) \leftarrow nat(A) \wedge nat\_list(T) \wedge in\_range(A,1,N) \wedge$
$\neg new1(A,T,M)$

We fold clause 13 using clause 14 and we derive the following definite clause:

13f. $new2(N,[H|T],K) \leftarrow N \geq K + 1 \wedge new2(N,T,K+1) \wedge new3(H,T,N,0)$

By positive and negative unfolding, replacement, and positive folding, from clause 14 we derive the following definite clauses:

15. $new3(A,[\,],N,M) \leftarrow in\_range(A,1,N) \wedge nat(A)$
16. $new3(A,[B|T],N,M) \leftarrow A \neq B \wedge A - B \neq M + 1 \wedge B - A \neq M + 1 \wedge$
$nat(B) \wedge new3(A,T,N,M+1)$

Finally, by assuming that the set of predicates of interest is the singleton $\{queens\}$, by definition elimination we derive the following program:

R: 1f. $queens(N, L) \leftarrow new2(N, L, 0)$
12. $new2(N, [\,], K) \leftarrow N = K$
13f. $new2(N, [H|T], K) \leftarrow N \geq K+1 \wedge new2(N, T, K+1) \wedge new3(H, T, N, 0)$
15. $new3(A, [\,], N, M) \leftarrow in\_range(A, 1, N) \wedge nat(A)$
16. $new3(A, [B|T], N, M) \leftarrow A \neq B \wedge A - B \neq M+1 \wedge B - A \neq M+1 \wedge$
$nat(B) \wedge new3(A, T, N, M+1)$

together with the clauses for the predicates *in_range* and *nat*.

Program R is a definite program and, by Theorem 3, we have that $M(R) \models queens(N, L)$ iff $M(P \cup F \cup Defs) \models queens(N, L)$, where $F \cup Defs$ is the set of all clauses introduced by the definition introduction rule during the transformation sequences from P to R. Since *queens* does not depend on *Defs* in $P \cup F \cup Defs$, we have that $M(R) \models queens(N, L)$ iff $M(P \cup F) \models queens(N, L)$ and, thus, Property (π) holds. Moreover, it can be shown that R terminates for all calls of the form $queens(n, L)$, where n is any nonnegative integer and L is a variable.

Notice that program R computes a solution of the N-queens problem in a clever way: each time a queen is placed on the board, program R checks that it does not attack any other queen already placed on the board.

5.3 Program Specialization: Derivation of Counter Machines from Constrained Regular Expressions

Given a set $\mathcal{N}$ of variables ranging over natural numbers, a set $\mathcal{C}$ of constraints over natural numbers, and a set K of identifiers, we define a *constrained regular expression* e over the alphabet $\{a, b\}$ as follows:

$e ::= \; a \mid b \mid e_1 \cdot e_2 \mid e_1 + e_2 \mid e\hat{}N \mid not(e) \mid k$

where $N \in \mathcal{N}$ and $k \in K$. An identifier $k \in K$ is defined by a *definition* of the form $k \equiv (c : e)$, where $c \in \mathcal{C}$ and e is a constrained regular expression. For instance, the set $\{a^m b^n \mid m = n \geq 0\}$ of strings in $\{a, b\}^*$ is denoted by the identifier k which is defined by the following definition:

$k \equiv (M = N : (a\hat{}M \cdot b\hat{}N))$.

Obviously, constrained regular expressions may denote languages which are *not* regular.

Given a string S and a constrained regular expression e, the following locally stratified program P checks whether or not S belongs to the language denoted by e. We assume that constraints are definable as conjunctions of equalities and disequalities over natural numbers.

P : $string([\,]) \leftarrow$
$string([a|S]) \leftarrow string(S)$
$string([b|S]) \leftarrow string(S)$
$symbol(a) \leftarrow$
$symbol(b) \leftarrow$
$app([\,], L, L) \leftarrow$
$app([A|X], Y, [A|Z]) \leftarrow app(X, Y, Z)$
$in\_language([A], A) \leftarrow symbol(A)$

$in\_language(S, (E1 \cdot E2)) \leftarrow app(S1, S2, S) \wedge$
$\quad in\_language(S1, E1) \wedge in\_language(S2, E2)$
$in\_language(S, E1{+}E2) \leftarrow in\_language(S, E1)$
$in\_language(S, E1{+}E2) \leftarrow in\_language(S, E2)$
$in\_language(S, not(E)) \leftarrow \neg\, in\_language(S, E)$
$in\_language([\,], E\hat{}I) \leftarrow I{=}0$
$in\_language(S, E\hat{}I) \leftarrow I{=}J{+}1 \wedge J{\geq}0 \wedge app(S1, S2, S) \wedge$
$\quad in\_language(S1, E) \wedge in\_language(S2, E\hat{}J)$
$in\_language(S, K) \leftarrow (K \equiv (C : E)) \wedge solve(C) \wedge in\_language(S, E)$
$solve(X{=}Y) \leftarrow X{=}Y$
$solve(X{\geq}Y) \leftarrow X{\geq}Y$
$solve(C_1 \wedge C_2) \leftarrow solve(C_1) \wedge solve(C_2)$

For example, in order to check whether a string S does *not* belong to the language denoted by k, where k is defined by the following definition: $k \equiv (M = N : (a\hat{}M \cdot b\hat{}N))$, we add to program P the clause:

$$(k \equiv (M{=}N : (a\hat{}M \cdot b\hat{}N))) \leftarrow$$

and we evaluate a query of the form:

$$string(S) \wedge in\_language(S, not(k))$$

Now, if we want to specialize program P w.r.t. this query, we introduce the new definition:

1. $new1(S) \leftarrow string(S) \wedge in\_language(S, not(k))$

By unfolding clause 1 we get:

2. $new1(S) \leftarrow string(S) \wedge \neg\, in\_language(S, k)$

We cannot perform the negative unfolding of clause 2 w.r.t. $\neg\, in\_language(S, k)$ because of the local variables in the clauses for $in\_language(S, k)$. In order to derive a predicate which is equivalent to $in\_language(S, k)$ and is defined by clauses without local variables, we introduce the following clause:

3. $new2(S) \leftarrow in\_language(S, k)$

By unfolding clause 3 we get:

4. $new2(S) \leftarrow M{=}N \wedge app(S1, S2, S) \wedge$
$\quad in\_language(S1, a\hat{}M) \wedge in\_language(S2, b\hat{}N)$

We generalize clause 4 and we introduce the following clause 5:

5. $new3(S, I) \leftarrow M{=}N{+}I \wedge app(S1, S2, S) \wedge$
$\quad in\_language(S1, a\hat{}M) \wedge in\_language(S2, b\hat{}N)$

By unfolding clause 5, performing replacements based on laws of constraints, and folding, we get:

6. $new3(S, N) \leftarrow in\_language(S, b\hat{}N)$
7. $new3([a|S], N) \leftarrow new3(S, N{+}1)$

In order to fold clause 6 we introduce the following definition:

8. $new4(S, N) \leftarrow in\_language(S, b\hat{}N)$

By unfolding clause 8, performing some replacements based on laws of constraints, and folding, we get:

9. $new4([\,],0) \leftarrow$
10. $new4([b|S],N) \leftarrow N \geq 1 \wedge new4(S,N-1)$

By negative folding of clause 2 and positive folding of clauses 4 and 6 we get the following program:

2f. $new1(S) \leftarrow string(S) \wedge \neg\, new2(S)$
4f. $new2(S) \leftarrow new3(S,0)$
6f. $new3(S,N) \leftarrow new4(S,N)$
7. $new3([a|S],N) \leftarrow new3(S,N+1)$
9. $new4([\,],0) \leftarrow$
10. $new4([b|S],N) \leftarrow N \geq 1 \wedge new4(S,N-1)$

Now from clause 2f, by positive and negative unfoldings, replacements based on laws of constraints, and folding, we get:

11. $new1([a|S]) \leftarrow string(S) \wedge \neg\, new3(S,1)$
12. $new1([b|S]) \leftarrow string(S)$

In order to fold clause 11 we introduce the following definition:

13. $new5(S,N) \leftarrow string(S) \wedge \neg\, new3(S,N)$

By positive and negative unfolding and folding we get:

14. $new5([\,],N) \leftarrow$
15. $new5([a|S],N) \leftarrow new5(S,N+1)$
16. $new5([a|S],N) \leftarrow string(S) \wedge \neg\, N \geq 1$
17. $new5([b|S],N) \leftarrow string(S) \wedge \neg\, new4(S,N-1)$

In order to fold clause 17 we introduce the following definition:

18. $new6(S,N) \leftarrow string(S) \wedge \neg\, new4(S,N)$

Now, starting from clause 18, by positive and negative unfolding, replacements based on laws of constraints, folding, and elimination of the predicates on which $new1$ does not depend, we get the following final, specialized program:

P_{spec} : 11f. $new1([a|S]) \leftarrow new5(S,1)$
12. $new1([b|S]) \leftarrow string(S)$
14. $new5([\,],N) \leftarrow$
15. $new5([a|S],N) \leftarrow new5(S,N+1)$
16. $new5([b|S],0) \leftarrow string(S)$
17f. $new5([b|S],N) \leftarrow new6(S,N-1)$
19. $new6([\,],N) \leftarrow N \neq 0$
20. $new6([a|S],N) \leftarrow string(S)$
21. $new6([b|S],0) \leftarrow string(S)$
22. $new6([b|S],N) \leftarrow new6(S,N-1)$

This specialized program corresponds to a one-counter machine (that is, a push-down automaton where the stack alphabet contains one letter only [5]) and it takes $O(n)$ time to test that a string of length n does *not* belong to the language $\{a^m{\cdot}b^n \mid m = n \geq 0\}$.

6 Related Work and Conclusions

During the last two decades various sets of unfold/fold transformation rules have been proposed for different classes of logic programs. The authors who first introduced the unfold/fold rules for logic programs were Tamaki and Sato in their seminal paper [44]. That paper presents a set of rules for transforming definite logic programs and it also presents the proof that those rules are correct w.r.t. the least Herbrand model semantics. Most of the subsequent papers in the field have followed Tamaki and Sato's approach in that: (i) the various sets of rules which have been published can be seen as extensions or variants of Tamaki and Sato's rules, and (ii) the techniques used for proving the correctness of the rules are similar to those used by Tamaki and Sato (the reader may look at the references given later in this section, and also at [29] for a survey). In the present paper we ourselves have followed Tamaki and Sato's approach, but we have considered the more complex framework of locally stratified constraint logic programs with the perfect model semantics.

Among the rules we have presented, the following ones were initially introduced in [44] (in the case of definite logic programs): (R1) definition introduction, restricted to one clause only (that is, with $m=1$), (R3) positive unfolding, (R5) positive folding, restricted to one clause only (that is, with $m=1$). Our rules of replacement, deletion of useless predicates, constraint addition, and constraint deletion (that is, rules R7, R8, R9, and R10, respectively) are extensions to the case of constraint logic programs with negation of the *goal replacement* and *clause addition/deletion* rules presented in [44]. In comparing the rules in [44] and the corresponding rules we have proposed, let us highlight also the following important difference. The goal replacement and clause addition/deletion of [44] are very general, but their applicability conditions are based on properties of the least Herbrand model and properties of the proof trees (such as goal equivalence or clause implication) which, in general, are very difficult to prove. On the contrary, (i) the applicability conditions of our replacement rule require the verification of (usually decidable) properties of the constraints, (ii) the property of being a useless predicate is decidable, because it refers to predicate symbols only (and not to the value of their arguments), and (iii) the applicability conditions for constraint addition and constraint deletion can be verified in most cases by program analysis techniques based on abstract interpretation [10].

For the correctness theorem (see Theorem 3) relative to admissible transformation sequences we have followed Tamaki and Sato's approach, and as in [44], the correctness is ensured by assuming the validity of some suitable conditions on the construction of the transformation sequences.

Let us now relate our work here to that of other authors who have extended in several ways the work by Tamaki and Sato and, in particular, those who have extended it to the cases of: (i) general logic programs, and (ii) constraint logic programs.

Tamaki and Sato's unfolding and folding rules have been extended to general logic programs (without constraints) by Seki. He proved his extended rules correct w.r.t. various semantics, including the perfect model semantics [42,43].

Building upon previous work for definite logic programs reported in [17,22,38], paper [37] extended Seki's folding rule by allowing: (i) *multiple* folding, that is, one can fold m (≥ 1) clauses at a time using a definition consisting of m clauses, and (ii) *recursive* folding, that is, the definition used for folding can contain recursive clauses.

Multiple folding can be performed by applying our rule R5, but recursive folding *cannot*. Indeed, by rule R5 we can fold using a definition introduced by rule R1, and this rule does *not* allow the introduction of recursive clauses. Thus, in this respect the folding rule presented in this paper is less powerful than the folding rule considered in [37]. On the other hand, the set of rules presented here is more powerful than the one in [37] because it includes negative unfolding (R4) and negative folding (R6). These two rules are very useful in practice, and both are needed for the program derivation examples we have given in Section 5. They are also needed in the many examples of program verification presented in [13]. For reasons of simplicity, we have presented our non-recursive version of the positive folding rule because it has much simpler applicability conditions. In particular, the notion of admissible transformation sequence is much simpler for non-recursive folding. We leave for future research the problem of studying the correctness of a set of transformation rules which includes positive and negative unfolding, as well as recursive positive folding and recursive negative folding.

Negative unfolding and negative folding were also considered in our previous work [32]. The present paper extends the transformation rules presented in [32] by adapting them to a logic language with constraints. Moreover, in [32] we did not present the proof of correctness of the transformation rules and we only showed some applications of our transformation rules to theorem proving and program synthesis.

In [40] Sato proposed a set of transformation rules for *first order programs*, that is, for a logic language that extends general logic programs by allowing arbitrary first order formulas in the bodies of the clauses. However, the semantics considered in [40] is based on a three valued logic with the three truth values *true*, *false*, and *undefined* (corresponding to non terminating computations). Thus, the results presented in [40] cannot be directly compared with ours. In particular, for instance, the rule for eliminating useless predicates (R8) does not preserve the three valued semantics proposed in [40], because this rule may transform a program that does not terminate for a given query, into a program that terminates for that query. Moreover, the conditions for the applicability of the folding rule given in [40] are based on the chosen three valued logic and cannot be compared with those presented in this paper.

Various other sets of transformation rules for general logic programs (including several variants of the goal replacement rule) have been proved correct w.r.t. other semantics, such as, the operational semantics based on SLDNF resolution [16,42], Clark's completion [16], and Kunen's and Fitting's three valued extensions of Clark's completion [8]. We will not enter into a detailed comparison with these works here. It will suffice to say that these works are not directly comparable with ours because of the different set of rules (in particular, none of

these works considers the negative unfolding rule) and the different semantics considered.

The unfold/fold transformation rules have also been extended to constraint logic programs in [7,11,12,27]. Papers [7,11] deal with definite programs, while [27] considers locally stratified programs and proves that, with suitable restrictions, the unfolding and folding rules preserve the perfect model semantics. Our correctness result presented here extends that in [27] because: (i) the rules of [27] include neither negative unfolding nor negative folding, and (ii) the folding rule of [27] is *reversible*, that is, it can only be applied for folding a set of clauses in a program P by using a set of clauses that occur in P. As already mentioned in Section 3, our folding rule is not reversible, because we may fold clauses in program P_k of a transformation sequence by using definitions occurring in $Defs_k$, but possibly not in P_k. Reversibility is a very strong limitation, because it does not allow the derivation of recursive clauses from non-recursive clauses. In particular, the derivations presented in our examples of Section 5 could not be performed by using the reversible folding rule of [27].

Finally, [12] proposes a set of transformation rules for locally stratified constraint logic programs tailored to a specific task, namely, program specialization and its application to the verification of infinite state reactive systems. Due to their specific application, the transformation rules of [12] are much more restricted than the ones presented here. In particular, by using the rules of [12]: (i) we can only introduce *constrained atomic definitions*, that is, definitions that consist of single clauses whose body is a constrained atom, (ii) we can unfold clauses w.r.t. a negated atom only if that atom succeeds or fails in one step, and (iii) we can apply the positive and negative folding rules by using constrained atomic definitions only.

We envisage several lines for further development of the work presented in this paper. As a first step forward, one could design strategies for automating the application of the transformation rules proposed here. In our examples of Section 5 we have demonstrated that some strategies already considered in the literature for the case of definite programs, can be extended to general constraint logic programs. This extension can be done, in particular, for the following strategies: (i) the elimination of local variables [34], (ii) the derivation of deterministic programs [33], and (iii) the rule-based program specialization [24].

It has been pointed out by recent studies that there is a strict relationship between program transformation and various other methodologies for program development and software verification (see, for instance, [13,15,25,30,31,36]). Thus, strategies for the automatic application of transformation rules can be exploited in the design of automatic techniques in these related fields and, in particular, in program synthesis and theorem proving. We believe that transformation methodologies for logic and constraint languages can form the basis of a very powerful framework for machine assisted software development.

Acknowledgements

We would like to thank Maurice Bruynooghe and Kung-Kiu Lau for inviting us to contribute to this volume. We would like also to acknowledge the very stimulating conversations we have had over the years with the members of the LOPSTR community since the beginning of the series of the LOPSTR workshops. Finally, we express our thanks to the anonymous referees for their helpful comments and suggestions.

7 Appendices

7.1 Appendix A

In this Appendix A we will use the fact that, given any two atoms A and B, and any valuation v, if $\sigma(v(A)) \geq \sigma(v(B))$ then for every substitution ϑ, $\sigma(v(A\vartheta)) \geq \sigma(v(B\vartheta))$. The same holds with $>$, instead of $\geq$.

Proof of Proposition 1. [Preservation of Local Stratification]. We will prove that, for $k = 0, \ldots, n$, P_k is locally stratified w.r.t. σ by induction on k.
Base case ($k = 0$). By hypothesis P_0 is locally stratified w.r.t. σ.
Induction step. We assume that P_k is locally stratified w.r.t. σ and we show that P_{k+1} is locally stratified w.r.t. σ. We proceed by cases depending on the transformation rule which is applied to derive P_{k+1} from P_k.
Case 1. Program P_{k+1} is derived by definition introduction (rule R1). We have that $P_{k+1} = P_k \cup \{\delta_1, \ldots, \delta_m\}$, where P_k is locally stratified w.r.t. σ by the inductive hypothesis and $\{\delta_1, \ldots, \delta_m\}$ is locally stratified w.r.t. σ by Condition (iv) of R1. Thus, P_{k+1} is locally stratified w.r.t. σ.

Case 2. Program P_{k+1} is derived by definition elimination (rule R2). Then P_{k+1} is locally stratified w.r.t. σ because $P_{k+1} \subseteq P_k$.

Case 3. Program P_{k+1} is derived by positive unfolding (rule R3). We have that $P_{k+1} = (P_k - \{\gamma\}) \cup \{\eta_1, \ldots, \eta_m\}$, where γ is a clause in P_k of the form $H \leftarrow c \wedge G_L \wedge A \wedge G_R$ and clauses $\eta_1, \ldots, \eta_m$ are derived by unfolding γ w.r.t. A. Since, by the induction hypothesis, $(P_k - \{\gamma\})$ is locally stratified w.r.t. σ, it remains to show that, for every valuation v, for $i = 1, \ldots, m$, clause $v(\eta_i)$ is locally stratified w.r.t. σ. Take any valuation v. For $i = 1, \ldots, m$, there exists a clause γ_i in a variant of P_k of the form $K_i \leftarrow c_i \wedge B_i$ such that η_i is of the form $H \leftarrow c \wedge A = K_i \wedge c_i \wedge G_L \wedge B_i \wedge G_R$. By the inductive hypothesis, $v(H \leftarrow c \wedge G_L \wedge A \wedge G_R)$ and $v(K_i \leftarrow c_i \wedge B_i)$ are locally stratified w.r.t. σ. We consider two cases: (a) $\mathcal{D} \models \neg v(c \wedge A = K_i \wedge c_i)$ and (b) $\mathcal{D} \models v(c \wedge A = K_i \wedge c_i)$. In Case (a), $v(\eta_i)$ is locally stratified w.r.t. σ by definition. In Case (b), we have that: (i) $\mathcal{D} \models v(c)$, (ii) $\mathcal{D} \models v(A) = v(K_i)$, and (iii) $\mathcal{D} \models v(c_i)$. Let us consider a literal $v(L)$ occurring in the body of $v(\eta_i)$. If $v(L)$ is an atom occurring positively in $v(G_L \wedge G_R)$ then $\sigma(v(H)) \geq \sigma(v(L))$ because $v(H \leftarrow c \wedge G_L \wedge A \wedge G_R)$ is locally stratified w.r.t. σ and $\mathcal{D} \models v(c)$. Similarly, if $v(L)$ is a negated atom occurring in $v(G_L \wedge G_R)$ then $\sigma(v(H)) > \sigma(\overline{v(L)})$. If $v(L)$ is an atom occurring positively in $v(B_i)$ then $\sigma(v(H)) \geq \sigma(v(L))$. Indeed:

$$
\begin{array}{rl}
\sigma(v(H)) \geq \sigma(v(A)) & \text{(because } v(H \leftarrow c \wedge G_L \wedge A \wedge G_R) \text{ is locally stratified} \\
& \quad \text{w.r.t. } \sigma \text{ and } \mathcal{D} \models v(c)) \\
= \sigma(v(K_i)) & \text{(because } v(A) = v(K_i)) \\
\geq \sigma(v(L)) & \text{(because } v(K_i \leftarrow c_i \wedge B_i) \text{ is locally stratified w.r.t. } \sigma \\
& \quad \text{and } \mathcal{D} \models v(c_i))
\end{array}
$$

Similarly, if $v(L)$ is a negated atom occurring in $v(B)$ then $\sigma(v(H)) > \sigma(\overline{v(L)})$. Thus, the clause $v(\eta_i)$ is locally stratified w.r.t. σ.

Case 4. Program P_{k+1} is derived by negative unfolding (rule R4). As in Case 3, we have that $P_{k+1} = (P_k - \{\gamma\}) \cup \{\eta_1, \ldots, \eta_s\}$, where γ is a clause in P_k of the form $H \leftarrow c \wedge G_L \wedge \neg A \wedge G_R$ and clauses $\eta_1, \ldots, \eta_s$ are derived by negative unfolding γ w.r.t. $\neg A$. Since, by the induction hypothesis, $(P_k - \{\gamma\})$ is locally stratified w.r.t. σ, it remains to show that, for every valuation v, for $j = 1, \ldots, s$, clause $v(\eta_j)$ is locally stratified w.r.t. σ. Take any valuation v. Let $K_1 \leftarrow c_1 \wedge B_1, \ldots, K_m \leftarrow c_m \wedge B_m$ be the clauses in a variant of P_k such that, for $i = 1, \ldots, m$, $\mathcal{D} \models \exists(c \wedge A = K_i \wedge c_i)$. Then, we have that, for $j = 1, \ldots, s$, the clause $v(\eta_j)$ is of the form $v(H \leftarrow c \wedge e_j \wedge G_L \wedge Q_j \wedge G_R)$, where $v(Q_j)$ is a conjunction of literals. By the applicability conditions of the negative unfolding rule and by construction (see Steps 1–4 of R4), we have that there exist m substitutions $\vartheta_1, \ldots, \vartheta_m$ such that the following two properties hold:

(P.1) for every literal $v(L)$ occurring in $v(Q_j)$ there exists a (positive or negative) literal $v(M)$ occurring in $v(B_i\vartheta_i)$ for some $i \in \{1, \ldots, m\}$, such that $v(L)$ is $\overline{v(M)}$, and

(P.2) if $v(L)$ occurs in $v(Q_j)$ and $v(L)$ is $\overline{v(M)}$ with $v(M)$ occurring in $v(B_i\vartheta_i)$ for some $i \in \{1, \ldots, m\}$, then $\mathcal{D} \models v((c \wedge e_j) \rightarrow (A = K_i\vartheta_i \wedge c_i\vartheta_i))$.

We will show that $v(\eta_j)$ is locally stratified w.r.t. σ. By the inductive hypothesis, we have that $v(H \leftarrow c \wedge G_L \wedge \neg A \wedge G_R)$ and $v(K_i\vartheta_i \leftarrow c_i\vartheta_i \wedge B_i\vartheta_i)$ are locally stratified w.r.t. σ.
We consider two cases: (a) $\mathcal{D} \models \neg v(c \wedge e_j)$ and (b) $\mathcal{D} \models v(c \wedge e_j)$. In Case (a), $v(\eta_j)$ is locally stratified w.r.t. σ by definition. In Case (b), take any literal $v(L)$ occurring in $v(Q_j)$. By Properties (P.1) and (P.2), $v(L)$ is $\overline{v(M)}$ for some $v(M)$ occurring in $v(B_i)$. We also have that: (i) $\mathcal{D} \models v(A) = v(K_i\vartheta_i)$ and (ii) $\mathcal{D} \models v(c_i\vartheta_i)$. Moreover $\mathcal{D} \models v(c)$, because we are in Case (b). Now, if $v(M)$ is a positive literal occurring in $v(B_i)$ we have:

$$
\begin{array}{rl}
\sigma(v(H)) > \sigma(v(A)) & \text{(because } v(H \leftarrow c \wedge G_L \wedge \neg A \wedge G_R) \text{ is locally stratified} \\
& \quad \text{w.r.t. } \sigma \text{ and } \mathcal{D} \models v(c)) \\
= \sigma(v(K_i\vartheta_i)) & \text{(because } v(A) = v(K_i\vartheta_i)) \\
(\dagger) \quad \geq \sigma(v(M)) & \text{(because } v(K_i\vartheta_i \leftarrow c_i\vartheta_i \wedge B_i\vartheta_i) \text{ is locally stratified} \\
& \quad \text{w.r.t. } \sigma \text{ and } \mathcal{D} \models v(c_i\vartheta_i)).
\end{array}
$$

Thus, we get: $\sigma(v(H)) > \sigma(v(M))$, and we conclude that $v(\eta_j)$ is locally stratified w.r.t. σ. Similarly, if $v(M)$ is a negative literal occurring in $v(B_i\vartheta_i)$, we also get: $\sigma(v(H)) > \sigma(\overline{v(M)})$. (In particular, if $v(M)$ is a negative literal, at Point (†) above, we have $\sigma(v(K_i\vartheta_i)) > \sigma(\overline{v(M)})$.) Thus, we also conclude that $v(\eta_j)$ is locally stratified w.r.t. σ.

Case 5. Program P_{k+1} is derived by positive folding (rule R5). For reasons of simplicity, we assume that we fold one clause only, that is, $m = 1$ in rule R5. The general case where $m \geq 1$ is analogous. We have that $P_{k+1} = (P_k - \{\gamma\}) \cup \{\eta\}$, where η is a clause of the form $H \leftarrow c \wedge G_L \wedge K\vartheta \wedge G_R$ derived by positive folding of clause γ of the form $H \leftarrow c \wedge d\vartheta \wedge G_L \wedge B\vartheta \wedge G_R$ using a clause δ of the form $K \leftarrow d \wedge B$ introduced by rule R1. We have to show that, for every valuation v, $v(H \leftarrow c \wedge G_L \wedge K\vartheta \wedge G_R)$ is locally stratified w.r.t. σ. By the inductive hypothesis, we have that: (i) for every valuation v, $v(\gamma)$ is locally stratified w.r.t. σ, and (ii) for every valuation v, $v(\delta)$ is locally stratified w.r.t. σ. Take any valuation v. There are two cases: (a) $\mathcal{D} \models \neg v(c)$ and (b) $\mathcal{D} \models v(c)$. In Case (a), $v(\eta)$ is locally stratified w.r.t. σ by definition. In Case (b), take any literal $v(L)$ occurring in $v(B\vartheta)$. Now, *either* (b1) $v(L)$ is a positive literal, *or* (b2) $v(L)$ is a negative literal. In Case (b1) there are two subcases: (b1.1) $\mathcal{D} \models \neg v(d\vartheta)$, and (b1.2) $\mathcal{D} \models v(d\vartheta)$. In Case (b1.1) by Condition (iv) of rule R1, $\sigma(v(K\vartheta)) = 0$ and thus, $\sigma(v(H)) \geq \sigma(v(K\vartheta))$. Hence, $v(\eta)$ is locally stratified w.r.t. σ. In Case (b1.2), we have that $\mathcal{D} \models v(c \wedge d\vartheta)$ and, by the inductive hypothesis, $\sigma(v(H)) \geq \sigma(v(L\vartheta))$. Thus, $\sigma(v(H)) \geq \sigma(v(K\vartheta))$, because by Condition (iv) of rule R1, $\sigma(v(K\vartheta))$ is the smallest ordinal α such that $\alpha \geq \sigma(v(L\vartheta))$. Thus, $v(\eta)$ is locally stratified w.r.t. σ.
Case (b2), when $v(L)$ is a negative literal occurring in $v(B\vartheta)$, has a proof similar to the one of Case (b1), except that $\sigma(v(H)) > \sigma(v(L\vartheta))$, instead of $\sigma(v(H)) \geq \sigma(v(L\vartheta))$.

Case 6. Program P_{k+1} is derived by negative folding (rule R6). We have that $P_{k+1} = (P_k - \{\gamma\}) \cup \{\eta\}$, where η is a clause of the form $H \leftarrow c \wedge d\vartheta \wedge G_L \wedge \neg K\vartheta \wedge G_R$ derived by negative folding of clause γ of the form $H \leftarrow c \wedge d\vartheta \wedge G_L \wedge \neg A\vartheta \wedge G_R$ using a clause δ of the form $K \leftarrow d \wedge A$ introduced by rule R1. We have to show that, for every valuation v, $v(\eta)$ is locally stratified w.r.t. σ. By the inductive hypothesis, we have that: (i) for every valuation v, $v(H \leftarrow c \wedge d\vartheta \wedge G_L \wedge \neg A\vartheta \wedge G_R)$ is locally stratified w.r.t. σ, and (ii) for every valuation v, $v(K \leftarrow d \wedge A)$ is locally stratified w.r.t. σ. Take any valuation v. There are two cases: (a) $\mathcal{D} \models \neg v(c \wedge d\vartheta)$, and (b) $\mathcal{D} \models v(c \wedge d\vartheta)$. In Case (a), $v(\eta)$ is locally stratified w.r.t. σ by definition. In Case (b), by the inductive hypothesis, we have only to show that $\sigma(v(H)) > \sigma(v(K\vartheta))$. Since $\mathcal{D} \models v(c \wedge d\vartheta)$, by the inductive hypothesis we have that $\sigma(v(H)) > \sigma(v(A\vartheta))$. By Condition (iv) of the rule R1, we have that $\sigma(v(H)) > \sigma(v(K\vartheta))$. Hence, $v(\eta)$ is locally stratified w.r.t. σ.

Case 7. Program P_{k+1} is derived by replacement (rule R7). We have that $P_{k+1} = (P_k - \Gamma_1) \cup \Gamma_2$, where $(P_k - \Gamma_1)$ is locally stratified w.r.t. σ by the inductive hypothesis and Γ_2 is locally stratified w.r.t. σ by the applicability conditions of rule R7. Thus, P_{k+1} is locally stratified w.r.t. σ.

Case 8. Program P_{k+1} is derived by deletion of useless clauses (rule R8). P_{k+1} is locally stratified w.r.t. σ by the inductive hypothesis because $P_{k+1} \subseteq P_k$.

Case 9. Program P_{k+1} is derived by constraint addition (rule R9). We have that $P_{k+1} = (P_k - \{\gamma_1\}) \cup \{\gamma_2\}$, where $\gamma_2 : H \leftarrow c \wedge d \wedge G$ is the clause in P_{k+1} derived by constraint addition from the clause $\gamma_1 : H \leftarrow c \wedge G$ in P_k. For every valuation v, $v(H \leftarrow c \wedge d \wedge G)$ is locally stratified w.r.t. σ because: (i) by the induction

hypothesis $v(H \leftarrow c \wedge G)$ is locally stratified w.r.t. σ and (ii) if $\mathcal{D} \models v(c \wedge d)$ then $\mathcal{D} \models v(c)$. Since, by the inductive hypothesis, $(P_k - \{\gamma_1\})$ is locally stratified w.r.t. σ, also P_{k+1} is locally stratified w.r.t. σ.

Case 10. Program P_{k+1} is derived by constraint deletion (rule R10). We have that $P_{k+1} = (P_k - \{\gamma_1\}) \cup \{\gamma_2\}$, where γ_2: $H \leftarrow c \wedge G$ is the clause in P_{k+1} derived by constraint deletion from clause γ_1: $H \leftarrow c \wedge d \wedge G$ in P_k. By the applicability conditions of R10, γ is locally stratified w.r.t. σ. Since, by the inductive hypothesis, $(P_k - \{\gamma_1\})$ is locally stratified w.r.t. σ, also P_{k+1} is locally stratified w.r.t. σ.

Finally, $P_0 \cup \mathit{Defs}_n$ is locally stratified w.r.t. σ by the hypothesis that P_0 is locally stratified w.r.t. σ and by Condition (iv) of rule R1. □

7.2 Appendix B

In the proofs of Appendices B and C we use the following notions. Given a clause γ: $H \leftarrow c \wedge L_1 \wedge \ldots \wedge L_m$ and a valuation v such that $\mathcal{D} \models v(c)$, we denote by γ_v the clause $v(H \leftarrow L_1 \wedge \ldots \wedge L_m)$. We define $\mathit{ground}(\gamma) = \{\gamma_v \mid v$ is a valuation and $\mathcal{D} \models v(c)\}$. Given a set Γ of clauses, we define $\mathit{ground}(\Gamma) = \bigcup_{\gamma \in \Gamma} \mathit{ground}(\gamma)$.

Proof of Proposition 3. Recall that $P_0, \ldots, P_i$ is constructed by $i\,(\geq 0)$ applications of the definition rule, that is, $P_i = P_0 \cup \mathit{Defs}_i$, and $P_i, \ldots, P_j$ is constructed by applying once the positive unfolding rule to each clause in Defs_i. Let σ be the fixed stratification function considered at the beginning of the construction of the transformation sequence. By Proposition 1, each program in the sequence $P_i, \ldots, P_j$ is locally stratified w.r.t. σ.

Let us consider a ground atom A. By complete induction on the ordinal $\sigma(A)$ we prove that, for $k = i, \ldots, j-1$, there exists a proof tree for A and P_k iff there exists a proof tree for A and P_{k+1}. The inductive hypothesis is:

(I1) for every ground atom A', if $\sigma(A') < \sigma(A)$ then there exists a proof tree for A' and P_k iff there exists a proof tree for A' and P_{k+1}.

(*If Part*) We consider a proof tree U for A and P_{k+1}, and we show that we can construct a proof tree T for A and P_k. We proceed by complete induction on $\mathit{size}(U)$. The inductive hypothesis is:

(I2) given any proof tree U_1 for a ground atom A_1 and P_{k+1}, if $\mathit{size}(U_1) < \mathit{size}(U)$ then there exists a proof tree T_1 for A_1 and P_k.

Let γ be a clause of P_{k+1} and let γ_v: $A \leftarrow L_1 \wedge \ldots \wedge L_r$ be the clause in $\mathit{ground}(\gamma)$ used at the root of U. Thus, $L_1, \ldots, L_r$ are the children of A in U. For $h = 1, \ldots, r$, if L_h is an atom then the subtree U_h of U rooted at L_h is a proof tree for L_h and P_{k+1}. Since $\mathit{size}(U_h) < \mathit{size}(U)$, by the inductive hypothesis (I2) there exists a proof tree T_h for L_h and P_k. For $h = 1, \ldots, r$, if L_h is a negated atom $\neg A_h$ then, by the definition of proof tree, there exists no proof tree for A_h and P_{k+1}. Since σ is a local stratification for P_{k+1}, we have that $\sigma(A_h) < \sigma(A)$ and, by the inductive hypothesis (I1) there exists no proof tree for A_h and P_k.

Now, we proceed by cases.

Case 1. $\gamma \in P_k$. We construct T as follows. The root of T is A. We use γ_v: $A \leftarrow L_1 \wedge \ldots \wedge L_r$ to construct the children of A. If $r = 0$ then *true* is the only child of A in T, and T is a proof tree for A and P_k. Otherwise $r \geq 1$ and, for $h = 1, \ldots, r$, if L_h is an atom A_h then T_h is the subtree of T at A_h, and if L_h is a negated atom then L_h is a leaf of T. By construction we have that T is a proof tree for A and P_k.
Case 2. $\gamma \notin P_k$ and $\gamma \in P_{k+1}$ because γ is derived by positive unfolding. Thus, there exist: a clause α in P_k of the form $H \leftarrow c \wedge G_L \wedge A_S \wedge G_R$ and a variant β of a clause in P_k of the form $K \leftarrow d \wedge B$ such that clause γ is of the form $H \leftarrow c \wedge A_S = K \wedge d \wedge G_L \wedge B \wedge G_R$. Thus, (i) $v(H) = A$, (ii) $\mathcal{D} \models v(c \wedge A_S = K \wedge d)$, and (iii) $v(G_L \wedge B \wedge G_R) = L_1, \ldots, L_r$. By (ii) we have that $\alpha_v \in \mathit{ground}(P_k)$ and $\beta_v \in \mathit{ground}(P_k)$. (Notice that, since β is a variant of a clause in P_k, then $\beta_v \in \mathit{ground}(P_k)$.)
We construct T as follows. The root of T is A. We use α_v to construct the children of A and then we use β_v to construct the children of A_S. The leaves of the tree constructed in this way are $L_1, \ldots, L_r$. If $r = 0$ then *true* is the only leaf of T, and T is a proof tree for A and P_k. Otherwise $r \geq 1$ and, for $h = 1, \ldots, r$, if L_h is an atom then T_h is the subtree of T rooted at L_h, and if L_h is a negated atom then L_h is a leaf of T. By construction we have that T is a proof tree for A and P_k.

(*Only-if Part*) We consider a proof tree T for a ground atom A and program P_k, for $k = i, \ldots j-1$, and we show that we can construct a proof tree U for A and P_{k+1}. We proceed by complete induction on $\mathit{size}(T)$. The inductive hypothesis is:

(I3) given any proof tree T_1 for a ground atom A_1 and P_k, if $\mathit{size}(T_1) < \mathit{size}(T)$ then there exists a proof tree U_1 for A_1 and P_{k+1}.

Let γ be a clause of P_k and let γ_v: $A \leftarrow L_1 \wedge \ldots \wedge L_r$ be the clause in $\mathit{ground}(\gamma)$ used at the root of T. Now we proceed by cases.
Case 1. $\gamma \in P_{k+1}$. We construct the proof tree U for A and P_{k+1} as follows. We use γ_v to construct the children $L_1, \ldots, L_r$ of the root A. If $r = 0$ then *true* is the only child of A in U, and U is a proof tree for A and P_{k+1}. Otherwise, $r \geq 1$ and, for $h = 1, \ldots, r$, if L_h is an atom, we consider the subtree T_h of T rooted at L_h. We have that T_h is a proof tree for L_h and P_k with $\mathit{size}(T_h) < \mathit{size}(T)$ and, therefore, by the inductive hypothesis (I3), there exists a proof tree U_h for L_h and P_{k+1}. For $h = 1, \ldots, r$, if L_h is a negated atom $\neg A_h$, then $\sigma(A) > \sigma(A_h)$ because σ is a stratification function for P_k. Thus, by the inductive hypothesis (I1) we have that there is no proof tree for A_h and P_{k+1}. The construction of U continues as follows. For $h = 1, \ldots, r$, if L_h is an atom then we use U_h as a subtree of U rooted at L_h and, if L_h is a negated atom, then L_h is a leaf of U. Thus, by construction we have that U is a proof tree for A and P_{k+1}.
Case 2. $\gamma \in P_k$ and $\gamma \notin P_{k+1}$ because γ has been unfolded w.r.t. an atom in its body. Let us assume that γ is of the form $H \leftarrow c \wedge G_L \wedge A_S \wedge G_R$ and γ has been unfolded w.r.t. A_S. We have that: (i) $v(H) = A$, (ii) $\mathcal{D} \models v(c)$, and (iii) the ground literals $L_1, \ldots, L_r$ such that $L_1 \wedge \ldots \wedge L_r = v(G_L \wedge A_S \wedge G_R)$ are the children of A in T. Let β: $K \leftarrow d \wedge B$ be the clause in P_k which has been used for

constructing the children of $v(A_S)$ in T. Thus, there exists a valuation v' such that: (iv) $v(A_S) = v'(K)$, (v) $\mathcal{D} \models v'(d)$, and (vi) the literals in $v'(B)$ are the children of $v(A_S)$ in T. Without loss of generality we may assume that γ and β have no variables in common and $v = v'$. Thus, the ground literals $M_1, \ldots, M_s$ such that $M_1 \wedge \ldots \wedge M_s = v(G_L \wedge B \wedge G_R)$ are descendants of A in T. For $h = 1, \ldots, s$, if M_h is an atom, let us consider the subtree T_h of T rooted at M_h. We have that T_h is a proof tree for M_h and P_k with $size(T_h) < size(T)$ and, therefore, by the inductive hypothesis (I3), there exists a proof tree U_h for M_h and P_{k+1}. For $h = 1, \ldots, s$, if M_h is a negated atom $\neg A_h$ then M_h is a leaf of T and there exists no proof tree for A_h and P_k. Since σ is a stratification function for P_k, we have that $\sigma(A) > \sigma(A_h)$ and thus, by the inductive hypothesis (I1), there exists no proof tree for A_h and P_{k+1}.

Now let us consider the clause $\eta : H \leftarrow c \wedge A_S = K \wedge d \wedge G_L \wedge B \wedge G_R$. η is one of the clauses derived by unfolding γ because $\beta \in P_k$ and, by (ii), (iv), (v) and the assumption that $v = v'$, we have that $\mathcal{D} \models v(c \wedge A_S = K \wedge d)$ and hence $\mathcal{D} \models \exists(c \wedge A_S = K \wedge d)$. Thus, we construct a proof tree U for A and P_{k+1} as follows. Since $A = v(H)$ and $M_1 \wedge \ldots \wedge M_s = v(G_L \wedge B \wedge G_R)$, we can use η_v: $v(H \leftarrow G_L \wedge B \wedge G_R)$ to construct the children $M_1, \ldots, M_s$ of A in U. If $s = 0$ then *true* is the only child of A in U, and U is a proof tree for A and P_{k+1}. Otherwise, $s \geq 1$ and, for $h = 1, \ldots, s$, if M_h is an atom then U_h is the proof tree rooted at M_h in U. If M_h is a negated atom then M_h is a leaf of U. The proof tree U is the proof tree for A and P_{k+1} to be constructed. □

7.3 Appendix C

Proof of Proposition 5. Recall that the transformation sequence $P_0, \ldots, P_i, \ldots, P_j, \ldots, P_m$ is constructed as follows (see Definition 3):
(1) the sequence $P_0, \ldots, P_i$, with $i \geq 0$, is constructed by applying i times the definition introduction rule, that is, $P_i = P_0 \cup \mathit{Defs}_i$;
(2) the sequence $P_i, \ldots, P_j$ is constructed by applying once the positive unfolding rule to each clause in Defs_i which is used for applications of the folding rule in $P_j, \ldots, P_m$;
(3) the sequence $P_j, \ldots, P_m$, with $j \leq m$, is constructed by applying any rule, except the definition introduction and definition elimination rules.
Let σ be the fixed stratification function considered at the beginning of the construction of the transformation sequence. By Proposition 1, each program in the sequence $P_0 \cup \mathit{Defs}_i, \ldots, P_j, \ldots, P_m$ is locally stratified w.r.t. σ.

We will prove by induction on k that, for $k = j, \ldots, m$,
(*Soundness*) if there exists a proof tree for a ground atom A and P_k then there exists a proof tree for A and P_j, and
(*Completeness*) if there exists a P_j-consistent proof tree for a ground atom A and P_j then there exists a P_j-consistent proof tree for A and P_k.
The base case ($k = j$) is trivial.
For proving the induction step, consider any k in $\{j, \ldots, m{-}1\}$. We assume that the soundness and completeness properties hold for that k, and we prove that they hold for $k{+}1$. For the soundness property it is enough to prove that:

- if there exists a proof tree for a ground atom A and P_{k+1} then there exists a proof tree for A and P_k,

and for the completeness property it is enough to prove that:

- if there exists a P_j-consistent proof tree for a ground atom A and P_k then there exists a P_j-consistent proof tree for A and P_{k+1}.

We proceed by complete induction on the ordinal $\sigma(A)$ associated with the ground atom A. The inductive hypotheses are:

(IS) for every ground atom A' such that $\sigma(A') < \sigma(A)$, if there exists a proof tree for A' and P_{k+1} then there exists a proof tree for A' and P_k, and

(IC) for every ground atom A' such that $\sigma(A') < \sigma(A)$, if there exists a P_j-consistent proof tree for A' and P_k then there exists a P_j-consistent proof tree for A' and P_{k+1}.

By the inductive hypotheses on soundness and completeness for k, (IS), (IC), and Proposition 4, we have that:

(ISC) for every ground atom A' such that $\sigma(A') < \sigma(A)$, there exists a proof tree for A' and P_k iff there exists a proof tree for A' and P_{k+1}.

Now we give the proofs for the soundness and the completeness properties.

Proof of Soundness. Given a proof tree U for A and P_{k+1} we have to prove that there exists a proof tree T for A and P_k. The proof is by complete induction on $size(T)$. The inductive hypothesis is:

(Isize) Given any proof tree U' for a ground atom A' and P_{k+1}, if $size(U') < size(U)$ then there exists a proof tree T' for A' and P_k.

Let γ be a clause in P_{k+1} and v be a valuation. Let $\gamma_v \in ground(\gamma)$ be the ground clause of the form $A \leftarrow L_1 \wedge \ldots \wedge L_r$ used at the root of U. We proceed by considering the following cases: *either* (Case 1) γ belongs to P_k *or* (Case 2) γ does not belong to P_k and it has been derived from some clauses in P_k by applying a transformation rule among R3, R4, R5, R6, R7, R9, R10. (Recall that R1 and R2 are not applied in $P_j, \ldots, P_m$, and by R8 we delete clauses.)

The proof of Case 1 and the proofs of Case 2 for rules R3, R4, R9, and R10 are left to the reader. Now we present the proofs of Case 2 for rules R5, R6, and R7.

Case 2, rule R5. Clause γ is derived by positive folding. Let γ be derived by folding clauses $\gamma_1, \ldots, \gamma_m$ in P_k using clauses $\delta_1, \ldots, \delta_m$ where, for $i = 1, \ldots, m$, clause δ_i is of the form $K \leftarrow d_i \wedge B_i$ and clause γ_i is of the form $H \leftarrow c \wedge d_i\vartheta \wedge G_L \wedge B_i\vartheta \wedge G_R$, for a substitution ϑ satisfying Conditions (i) and (ii) given in (R5). Thus, γ is of the form: $H \leftarrow c \wedge G_L \wedge K\vartheta \wedge G_R$ and we have that: (a) $v(H) = A$, (b) $\mathcal{D} \models v(c)$, and (c) $v(G_L \wedge K\vartheta \wedge G_R) = L_1 \wedge \ldots \wedge L_r$. Since program P_{k+1} is locally stratified w.r.t. σ, by the inductive hypotheses (ISC) and (Isize) we have that: for $h = 1, \ldots, r$, if L_h is an atom then there exists a proof tree T_h for L_h and P_k, and if L_h is a negated atom $\neg A_h$ then there is no proof tree for A_h and P_k. The atom $v(K\vartheta)$ is one of the literals $L_1, \ldots, L_r$, say L_f, and thus, there exists a proof tree for $v(K\vartheta)$ and P_k. By the inductive hypothesis

(Soundness) for P_k and Proposition 3, there exists a proof tree for $v(K\vartheta)$ and P_i. Since $P_i = P_0 \cup \mathit{Defs}_n$ and $\delta_1, \ldots, \delta_m$ are all clauses in (a variant of) $P_0 \cup \mathit{Defs}_n$ which have the same predicate symbol as K, there exists $\delta_p \in \delta_1, \ldots, \delta_m$ such that δ_p is of the form $K \leftarrow d_p \wedge B_p$ and δ_p is used to construct the children of $v(K\vartheta)$ in the proof tree for $v(K\vartheta)$ and P_i. By Conditions (i) and (ii) on ϑ given in (R5), we have that: (d) $\mathcal{D} \models v(d_p\vartheta)$ and (e) $v(B_p\vartheta) = M_1 \wedge \ldots \wedge M_s$. By the definition of proof tree, for $h = 1, \ldots, s$, if M_h is an atom then there exists a proof tree for M_h and P_i, else if M_h is a negated atom $\neg E_h$ then there is no proof tree for E_h and P_i. By Propositions 3 and 4 and the inductive hypotheses (Soundness and Completeness) we have that, for $h = 1, \ldots, s$, if M_h is an atom then there exists a proof tree $\widehat{T_h}$ for M_h and P_k, else if M_h is a negated atom $\neg E_h$ then there is no proof tree for E_h and P_k.

Now we construct the proof tree T for A and P_k as follows. By (a), (b), and (d), we have that $v(H) = A$ and $\mathcal{D} \models v(c \wedge d_p\vartheta)$. Thus, we construct the children of A in T by using the clause γ_p: $H \leftarrow c \wedge d_p\vartheta \wedge G_L \wedge B_p\vartheta \wedge G_R$. Since $v(G_L \wedge B_p\vartheta \wedge G_R) = L_1 \wedge \ldots \wedge L_{f-1} \wedge M_1 \wedge \ldots \wedge M_s \wedge L_{f+1} \wedge \ldots \wedge L_r$, the children of A in T are: $L_1, \ldots, L_{f-1}, M_1, \ldots, M_s, L_{f+1}, \ldots, L_r$. By the applicability conditions of the positive folding rule, we have that $s > 0$ and A has a child different from the empty conjunction *true*. The children of A are constructed as follows. For $h = 1, \ldots, r$, if L_h is an atom then T_h is the subtree of T rooted in L_h, else if L_h is a negated atom then L_h is a leaf of T. For $h = 1, \ldots, s$, if M_h is an atom then $\widehat{T_h}$ is the subtree of T rooted in M_h, else if M_h is a negated atom then M_h is a leaf of T.

Case 2, rule R6. Clause γ is derived by negative folding. Let γ be derived by folding a clause α in P_k of the form $H \leftarrow c \wedge G_L \wedge \neg A_F\vartheta \wedge G_R$ by using a clause $\delta \in \mathit{Defs}_i$ of the form $K \leftarrow d \wedge A_F$. Thus, γ is of the form $H \leftarrow c \wedge G_L \wedge \neg K\vartheta \wedge G_R$.

Let γ_v be of the form $A \leftarrow L_1 \wedge \ldots \wedge L_{f-1} \wedge \neg v(K\vartheta) \wedge L_{f+1} \wedge \ldots \wedge L_r$, that is, $v(H) = A$ and $\mathcal{D} \models v(c)$. By the conditions on the applicability of rule R6, we also have that $\mathcal{D} \models v(d\vartheta)$. Since program P_{k+1} is locally stratified w.r.t. σ, we have that $\sigma(v(K\vartheta)) < \sigma(A)$. By the definition of proof tree, there is no proof tree for $v(K\vartheta)$ and P_{k+1}. Thus, by hypothesis (ISC) there exists no proof tree for $v(K\vartheta)$ and P_k. By the inductive hypothesis (Completeness) and Propositions 3 and 4, there exists no proof tree for $v(K\vartheta)$ and $P_0 \cup \mathit{Defs}_i$ and thus, since $K \leftarrow d \wedge A_F$ is the only clause defining the head predicate of K and $\mathcal{D} \models v(d\vartheta)$, there is no proof tree for $v(A_F\vartheta)$ and $P_0 \cup \mathit{Defs}_i$. By Proposition 3 and the inductive hypothesis (Soundness), there exists no proof tree for $v(A_F\vartheta)$ and P_k. Since $\mathcal{D} \models v(c)$ there exists a clause α_v in $\mathit{ground}(\alpha)$ of the form $A \leftarrow L_1 \wedge \ldots \wedge L_{f-1} \wedge \neg v(A_F\vartheta) \wedge L_{f+1} \wedge \ldots \wedge L_r$. We begin the construction of T by using α_v at the root. For all $h = 1, \ldots, f-1, f+1, \ldots, r$ such that L_h is an atom and U_h is the subtree of U rooted in L_h, we have that $\mathit{size}(U_h) < \mathit{size}(U)$. By hypothesis (Isize) there exists a proof tree T_h for L_h and P_k which we use as a subtree of T rooted in L_h. For all $h = 1, \ldots, f-1, f+1, \ldots, r$ such that L_h is a negated atom $\neg A_h$ we have that $\sigma(A_h) < \sigma(A)$, because program P_{k+1} is locally stratified w.r.t. σ. Moreover, there is no proof tree for A_h in P_{k+1}, because U is a proof tree. By hypothesis (ISC) we have that there is no proof tree for A_h in

P_k. Thus, for all $h = 1, \ldots, f-1, f+1, \ldots, r$ such that L_h is a negated atom we take L_h to be a leaf of T.

*Case*2, rule R7. Clause γ is derived by replacement. We only consider the case where P_{k+1} is derived from program P_k by applying the replacement rule based on law (8). The other cases are left to the reader. Suppose that a clause η: $H \leftarrow c_1 \wedge G$ in P_k is replaced by clause γ: $H \leftarrow c_2 \wedge G$ and $\mathcal{D} \models \forall\, (\exists Y\, c_1 \leftrightarrow \exists Z\, c_2)$, where: (i) $Y = FV(c_1) - FV(\{H, G\})$ and (ii) $Z = FV(c_2) - FV(\{H, G\})$. Thus, $\mathit{ground}(\gamma) = \mathit{ground}(\eta)$ and we can construct a proof tree for the ground atom A and P_k by using a clause in $\mathit{ground}(\eta)$, instead of a clause in $\mathit{ground}(\gamma)$.

Proof of Completeness. Given a P_j-consistent proof tree for A and P_k, we prove that there exists a P_j-consistent proof tree for A and P_{k+1}. The proof is by well-founded induction on $\mu(A, P_j)$. The inductive hypothesis is:

(Iμ) for every ground atom A' such that $\mu(A', P_j) < \mu(A, P_j)$, if there exists a P_j-consistent proof tree T' for A' and P_k then there exists a P_j-consistent proof tree U' for A' and P_{k+1}.

Let γ be a clause in P_k and v be a valuation such that $\gamma_v \in \mathit{ground}(\gamma)$ is the ground clause of the form $H \leftarrow L_1 \wedge \ldots \wedge L_r$ used at the root of T.

The proof proceeds by considering the following cases: *either* γ belongs to P_{k+1} *or* γ does not belong to P_{k+1} because it has been replaced (together with other clauses in P_k) with new clauses derived by an application of a transformation rule among R3, R4, R5, R6, R7, R8, R9, R10 (recall that R1 and R2 are not applied in $P_j, \ldots, P_m$). We present only the case where P_{k+1} is derived from P_k by positive folding (rule R5). The other cases are similar and are left to the reader.

Suppose that P_{k+1} is derived from P_k by folding clauses $\gamma_1, \ldots, \gamma_m$ in P_k using clauses $\delta_1, \ldots, \delta_m$ in (a variant of) Defs_k, and let γ be γ_p, with $1 \leq p \leq m$. Suppose also that, for $i = 1, \ldots, m$, clause δ_i is of the form $K \leftarrow d_i \wedge B_i$ and clause γ_i is of the form $H \leftarrow c \wedge d_i\vartheta \wedge G_L \wedge B_i\vartheta \wedge G_R$, for a substitution ϑ satisfying Conditions (i) and (ii) given in (R5). The clause η derived by folding $\gamma_1, \ldots, \gamma_m$ using $\delta_1, \ldots, \delta_m$ is of the form: $H \leftarrow c \wedge G_L \wedge K\vartheta \wedge G_R$. Since we use γ_v at the root of T, we have that: (a) $v(H) = A$, (b) $\mathcal{D} \models v(c \wedge d_p\vartheta)$, and (c) $v(G_L \wedge B_p\vartheta \wedge G_R) = L_1 \wedge \ldots \wedge L_r$, that is, for some $f1$, $f2$, $v(G_L) = L_1 \wedge \ldots \wedge L_{f1}$, $v(B_p\vartheta) = L_{f1+1} \wedge \ldots \wedge L_{f2}$, and $v(G_R) = L_{f2+1} \wedge \ldots \wedge L_r$. By Proposition 4 and the inductive hypotheses (Soundness and Completeness), for $h = f1+1, \ldots, f2$, if L_h is an atom then there exists a proof tree for L_h and P_j, and if L_h is a negated atom $\neg A_h$ then there is no a proof tree for A_h and P_j. By Proposition 3, by the fact that (by ii) $\mathcal{D} \models v(d_p\vartheta)$, and by the fact that $\delta_p \in P_i$ (recall that $\mathit{Defs}_k \subseteq P_i$), we have that there exists a proof tree for $v(K\vartheta)$ and P_j. Moreover, since $K \leftarrow d_p \wedge B_p$ has been unfolded w.r.t. a positive literal, we have that:

(†) $\quad \mu(v(B_p\vartheta), P_j) \geq \mu(v(K\vartheta), P_j)$

By Proposition 4 and the inductive hypothesis (Completeness), there exists a proof tree for $v(K\vartheta)$ and P_k. Since T is P_j-consistent we have that, for $h = 1, \ldots, r$, $\mu(A, P_j) > \mu(L_h, P_j)$. Moreover, we have that:

$$
\begin{aligned}
\mu(A, P_j) &> \mu(v(G_L \wedge B_p\vartheta \wedge G_R), P_j) && \text{(because } T \text{ is } P_j\text{-consistent)} \\
&= \mu(v(G_L), P_j) \oplus \mu(v(B_p\vartheta), P_j) \oplus \mu(v(G_R), P_j) && \text{(by definition of } \mu) \\
&\geq \mu(v(G_L), P_j) \oplus \mu(v(K\vartheta), P_j) \oplus \mu(v(G_R), P_j) && \text{(by (†))} \\
&\geq \mu(v(K\vartheta), P_j) && \text{(by definition of } \mu)
\end{aligned}
$$

By the inductive hypotheses (Iμ) and (IS), for $h = 1, \ldots, f1, f2{+}1, \ldots, r$, if L_h is an atom then there exists a P_j-consistent proof tree U_h for L_h and P_{k+1}, and if L_h is a negated atom $\neg A_h$ then there is no a proof tree for A_h and P_{k+1}. Moreover, by the inductive hypothesis (Iμ), there exists a P_j-consistent proof tree $\widehat{U}$ for $v(K\vartheta)$ and P_{k+1}.

Now we construct a P_j-consistent proof tree U for A and P_{k+1} as follows. By (a) and (b) we have that $v(H) = A$ and $\mathcal{D} \models v(c)$. Thus, we construct the children of A in U by using the clause η: $H \leftarrow c \wedge G_L \wedge K\vartheta \wedge G_R$. Since $v(G_L \wedge K\vartheta \wedge G_R) = L_1 \wedge \ldots \wedge L_{f1} \wedge v(K\vartheta) \wedge L_{f2+1} \wedge \ldots \wedge L_r$, the children of A in U are: $L_1, \ldots, L_{f1}, v(K\vartheta), L_{f2+1}, \ldots, L_r$. The construction of U continues as follows. For $h = 1, \ldots, f1, f2+1, \ldots, r$, if L_h is an atom then U_h is the P_j-consistent subtree of U rooted in L_h, else if L_h is a negated atom then L_h is a leaf of U. Finally, the subtree of U rooted in $v(K\vartheta)$ is the P_j-consistent proof tree $\widehat{U}$.

The proof tree U is indeed P_j-consistent because: (i) for $h = 1, \ldots, f1$, $f2{+}1, \ldots, r$, $\mu(A, P_j) > \mu(L_h, P_j)$, (ii) $\mu(A, P_j) \geq \mu(v(K\vartheta), P_j)$, and (iii) every subtree rooted in one of the literals $L_1, \ldots, L_{f1}, v(K\vartheta), L_{f2+1}, \ldots, L_r$ is P_j-consistent. □

References

1. M. Alpuente, M. Falaschi, G. Moreno, and G. Vidal. A transformation system for lazy functional logic programs. In A. Middeldorp and T. Sato, editors, *Proceedings of the 4th Fuji International Symposium on Functional and Logic Programming, FLOPS'99*, Lecture Notes in Computer Science 631, pages 147–162. Springer-Verlag, 1999.
2. K. R. Apt. Introduction to logic programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 493–576. Elsevier, 1990.
3. K. R. Apt. *From Logic Programming to Prolog.* Prentice Hall, London, UK, 1997.
4. K. R. Apt and R. N. Bol. Logic programming and negation: A survey. *Journal of Logic Programming*, 19, 20:9–71, 1994.
5. J.-M. Autebert, J. Berstel, and L. Boasson. Context-free languages and pushdown automata. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume 1, pages 111–174. Springer, Berlin, 1997.
6. D. Basin, Y. Deville, P. Flener, A. Hamfelt, and J.F. Nilsson. Synthesis of programs in computational logic. In M. Bruynooghe and K.-K. Lau, editors, *Program Development in Computational Logic*. Springer, 2004.
7. N. Bensaou and I. Guessarian. Transforming constraint logic programs. *Theoretical Computer Science*, 206:81–125, 1998.

8. A. Bossi, N. Cocco, and S. Etalle. Transforming normal programs by replacement. In A. Pettorossi, editor, *Proceedings 3rd International Workshop on Meta-Programming in Logic, Meta '92, Uppsala, Sweden*, Lecture Notes in Computer Science 649, pages 265–279, Berlin, 1992. Springer-Verlag.
9. R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, January 1977.
10. M. Garcia de la Banda, M. Hermenegildo, M. Bruynooghe, V. Dumortier, G. Janssens, and W. Simoens. Global analysis of constraint logic programs. *ACM Transactions on Programming Languages and Systems*, 18(5):564–614, 1996.
11. S. Etalle and M. Gabbrielli. Transformations of CLP modules. *Theoretical Computer Science*, 166:101–146, 1996.
12. F. Fioravanti. *Transformation of Constraint Logic Programs for Software Specialization and Verification*. PhD thesis, Università di Roma "La Sapienza", Italy, 2002.
13. F. Fioravanti, A. Pettorossi, and M. Proietti. Verifying CTL properties of infinite state systems by specializing constraint logic programs. In *Proceedings of the ACM Sigplan Workshop on Verification and Computational Logic VCL'01, Florence (Italy)*, Technical Report DSSE-TR-2001-3, pages 85–96. University of Southampton, UK, 2001.
14. F. Fioravanti, A. Pettorossi, and M. Proietti. Specialization with clause splitting for deriving deterministic constraint logic programs. In *Proceedings of the IEEE International Conference on Systems, Man and Cybernetics, Hammamet (Tunisia)*. IEEE Computer Society Press, 2002.
15. L. Fribourg and H. Olsén. Proving safety properties of infinite state systems by compilation into Presburger arithmetic. In *CONCUR '97*, Lecture Notes in Computer Science 1243, pages 96–107. Springer-Verlag, 1997.
16. P. A. Gardner and J. C. Shepherdson. Unfold/fold transformations of logic programs. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic, Essays in Honor of Alan Robinson*, pages 565–583. MIT, 1991.
17. M. Gergatsoulis and M. Katzouraki. Unfold/fold transformations for definite clause programs. In M. Hermenegildo and J. Penjam, editors, *Proceedings Sixth International Symposium on Programming Language Implementation and Logic Programming (PLILP '94)*, Lecture Notes in Computer Science 844, pages 340–354. Springer-Verlag, 1994.
18. C. J. Hogger. Derivation of logic programs. *Journal of the ACM*, 28(2):372–392, 1981.
19. J. Jaffar and M. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19/20:503–581, 1994.
20. J. Jaffar, M. Maher, K. Marriott, and P. Stuckey. The semantics of constraint logic programming. *Journal of Logic Programming*, 37:1–46, 1998.
21. N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.
22. T. Kanamori and H. Fujita. Unfold/fold transformation of logic programs with counters. Technical Report 179, ICOT, Tokyo, Japan, 1986.
23. T. Kanamori and K. Horiuchi. Construction of logic programs based on generalized unfold/fold rules. In *Proceedings of the Fourth International Conference on Logic Programming*, pages 744–768. The MIT Press, 1987.
24. M. Leuschel and M. Bruynooghe. Logic program specialisation through partial deduction: Control issues. *Theory and Practice of Logic Programming*, 2(4&5):461–515, 2002.

25. M. Leuschel and T. Massart. Infinite state model checking by abstract interpretation and program specialization. In A. Bossi, editor, *Proceedings of LOPSTR '99, Venice, Italy*, Lecture Notes in Computer Science 1817, pages 63–82. Springer, 1999.
26. J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, Berlin, 1987. Second Edition.
27. M. J. Maher. A transformation system for deductive database modules with perfect model semantics. *Theoretical Computer Science*, 110:377–403, 1993.
28. K. Marriott and P. Stuckey. *Programming with Constraints: An Introduction.* The MIT Press, 1998.
29. A. Pettorossi and M. Proietti. Transformation of logic programs: Foundations and techniques. *Journal of Logic Programming*, 19,20:261–320, 1994.
30. A. Pettorossi and M. Proietti. Synthesis and transformation of logic programs using unfold/fold proofs. *Journal of Logic Programming*, 41(2&3):197–230, 1999.
31. A. Pettorossi and M. Proietti. Perfect model checking via unfold/fold transformations. In J. W. Lloyd, editor, *First International Conference on Computational Logic, CL 2000, London, UK, 24-28 July, 2000*, Lecture Notes in Artificial Intelligence 1861, pages 613–628. Springer, 2000.
32. A. Pettorossi and M. Proietti. Program Derivation = Rules + Strategies. In A. Kakas and F. Sadri, editors, *Computational Logic: Logic Programming and Beyond* (*Essays in honour of Bob Kowalski, Part I*), Lecture Notes in Computer Science 2407, pages 273–309. Springer, 2002.
33. A. Pettorossi, M. Proietti, and S. Renault. Reducing nondeterminism while specializing logic programs. In *Proc. 24-th ACM Symposium on Principles of Programming Languages, Paris, France*, pages 414–427. ACM Press, 1997.
34. M. Proietti and A. Pettorossi. Unfolding-definition-folding, in this order, for avoiding unnecessary variables in logic programs. *Theoretical Computer Science*, 142(1):89–124, 1995.
35. T. C. Przymusinski. On the declarative semantics of stratified deductive databases and logic programs. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 193–216. Morgan Kaufmann, 1987.
36. A. Roychoudhury, K. Narayan Kumar, C. R. Ramakrishnan, I. V. Ramakrishnan, and S. A. Smolka. Verification of parameterized systems using logic program transformations. In *Proceedings of the Sixth International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2000, Berlin, Germany*, Lecture Notes in Computer Science 1785, pages 172–187. Springer, 2000.
37. A. Roychoudhury, K. Narayan Kumar, C. R. Ramakrishnan, and I.V. Ramakrishnan. Beyond Tamaki-Sato style unfold/fold transformations for normal logic programs. *International Journal on Foundations of Computer Science*, 13(3):387–403, 2002.
38. A. Roychoudhury, K. Narayan Kumar, C.R. Ramakrishnan, and I.V. Ramakrishnan. A parameterized unfold/fold transformation framework for definite logic programs. In *Proceedings of Principles and Practice of Declarative Programming (PPDP)*, Lecture Notes in Computer Science 1702, pages 396–413. Springer-Verlag, 1999.
39. D. Sands. Total correctness by local improvement in the transformation of functional programs. *ACM Toplas*, 18(2):175–234, 1996.
40. T. Sato. An equivalence preserving first order unfold/fold transformation system. *Theoretical Computer Science*, 105:57–84, 1992.

41. T. Sato and H. Tamaki. Transformational logic program synthesis. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 195–201. ICOT, 1984.
42. H. Seki. Unfold/fold transformation of stratified programs. *Theoretical Computer Science*, 86:107–139, 1991.
43. H. Seki. Unfold/fold transformation of general logic programs for well-founded semantics. *Journal of Logic Programming*, 16(1&2):5–23, 1993.
44. H. Tamaki and T. Sato. Unfold/fold transformation of logic programs. In S.-Å. Tärnlund, editor, *Proceedings of the Second International Conference on Logic Programming*, pages 127–138, Uppsala, Sweden, 1984. Uppsala University.