Proving Properties of Constraint Logic Programs by Eliminating Existential Variables

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

(1) DISP, University of Roma Tor Vergata, Via del Politecnico 1, I-00133 Roma, Italy
pettorossi@disp.uniroma2.it, senni@disp.uniroma2.it
(2) IASI-CNR, Viale Manzoni 30, I-00185 Roma, Italy
proietti@iasi.rm.cnr.it

Abstract. We propose a method for proving first order properties of constraint logic programs which manipulate finite lists of real numbers. Constraints are linear equations and inequations over reals. Our method consists in converting any given first order formula into a stratified constraint logic program and then applying a suitable unfold/fold transformation strategy that preserves the perfect model. Our strategy is based on the elimination of existential variables, that is, variables which occur in the body of a clause and not in its head. Since, in general, the first order properties of the class of programs we consider are undecidable, our strategy is necessarily incomplete. However, experiments show that it is powerful enough to prove several non-trivial program properties.

1 Introduction

It has been long recognized that program transformation can be used as a means of proving program properties. In particular, it has been shown that unfold/fold transformations introduced in [4,20] can be used to prove several kinds of program properties, such as equivalences of functions defined by recursive equation programs [5,9], equivalences of predicates defined by logic programs [14], first order properties of predicates defined by stratified logic programs [15], and temporal properties of concurrent systems [7,19]. In this paper we consider stratified logic programs *with constraints* and we propose a method based on unfold/fold transformations to prove first order properties of these programs.

The main reason that motivates our method is that transformation techniques may serve as a way of eliminating *existential variables* (that is, variables which occur in the body of a clause and not in its head) and the consequent quantifier elimination can be exploited to prove first order formulas. Quantifier elimination is a well established technique for theorem proving in first order logic [18] and one of its applications is Tarski's decision procedure for the theory of the field of reals. However, no quantifier elimination method has been developed so far to prove formulas within theories defined by constraint logic programs, where the constraints are themselves formulas of the theory of reals.

Consider, for instance, the following constraint logic program which defines the membership relation for finite lists of reals:

$Member$: $member(X, [Y|L]) \leftarrow X = Y$
$member(X, [Y|L]) \leftarrow member(X, L)$

Suppose we want to show that every finite list of reals has an upper bound, i.e.,

φ: $\forall L\, \exists U\, \forall X\, (member(X, L) \rightarrow X \leq U)$

Tarski's quantifier elimination method cannot help in this case, because the membership relation is not defined in the language of the theory of reals. The transformational technique we propose in this paper, proves the formula φ in two steps. In the first step we transform φ into clause form by applying a variant of the Lloyd-Topor transformation [11], thereby deriving the following clauses:

$Prop_1$: 1. $prop \leftarrow \neg p$
2. $p \leftarrow list(L) \wedge \neg q(L)$
3. $q(L) \leftarrow list(L) \wedge \neg r(L, U)$
4. $r(L, U) \leftarrow X > U \wedge list(L) \wedge member(X, L)$

where $list(L)$ holds iff L is a finite list of reals. The predicate $prop$ is equivalent to φ in the sense that $M(Member) \models \varphi$ iff $M(Member \cup Prop_1) \models prop$, where $M(P)$ denotes the perfect model of a stratified constraint logic program P. In the second step, we eliminate the existential variables by extending to constraint logic programs the techniques presented in [16] in the case of definite logic programs. For instance, the existential variable X occurring in the body of the above clause 4, is eliminated by applying the unfolding and folding rules and transforming that clause into the following two clauses: $r([X|L], U) \leftarrow X > U \wedge list(L)$ and $r([X|L], U) \leftarrow r(L, U)$. By iterating the transformation process, we eliminate all existential variables and we derive the following program which defines the predicate $prop$:

$Prop_2$: 1. $prop \leftarrow \neg p$
$2'$. $p \leftarrow p_1$
$3'$. $p_1 \leftarrow p_1$

Now, $Prop_2$ is a propositional program and has a *finite* perfect model, which is $\{prop\}$. Since all transformations we have made can be shown to preserve the perfect model, we have that $M(Member) \models \varphi$ iff $M(Prop_2) \models prop$ and, thus, we have completed the proof of φ.

The main contribution of this paper is the proposal of a proof method for showing that a closed first order formula φ holds in the perfect model of a stratified constraint logic program P, that is, $M(P) \models \varphi$. Our proof method is based on program transformations which eliminate existential variables.

The paper is organized as follows. In Section 2 we consider a class of constraint logic programs, called *lr-programs* (*lr* stands for lists of reals), which is Turing complete and for which our proof method is fully automatic. Those programs manipulate finite lists of reals with constraints which are linear equations and inequations over reals. In Section 3 we present the transformation strategy which defines our proof method and we prove its soundness. Due to the undecidability of the first order properties of *lr*-programs, our proof method is

necessarily incomplete. Some experimental results obtained by using a prototype implementation are presented in Section 5. Finally, in Section 6 we discuss related work in the field of program transformation and theorem proving.

2 Constraint Logic Programs over Lists of Reals

We assume that the reals are defined by the usual structure $\mathcal{R} = \langle R, 0, 1, +, \cdot, \leq \rangle$. In order to specify programs and formulas, we use a *typed* first order language [11] with two types: (i) *real*, denoting the set of reals, and (ii) *list of reals* (or *list*, for short), denoting the set of finite lists of reals.

We assume that every element of R is a constant of type *real*. A term p of type *real* is defined as follows:

$p ::= a \mid X \mid p_1 + p_2 \mid a \cdot X$

where a is a real number and X is a variable of type *real*. We also write aX, instead of $a \cdot X$. A term of type *real* will also be called a *linear polynomial*. An *atomic constraint* is a formula of the form: $p_1 = p_2$, or $p_1 < p_2$, or $p_1 \leq p_2$, where p_1 and p_2 are linear polynomials. We also write $p_1 > p_2$ and $p_1 \geq p_2$, instead of $p_2 < p_1$ and $p_2 \leq p_1$, respectively. A *constraint* is a finite conjunction of atomic constraints. A *first order formula over reals* is a first order formula constructed out of atomic constraints by using the usual connectives and quantifiers (i.e., $\neg, \wedge, \vee, \rightarrow, \exists, \forall$). By $F_{\mathcal{R}}$ we will denote the set of first order formulas over reals. A term l of type *list* is defined as follows:

$l ::= L \mid [\,] \mid [p \,|\, l\,]$

where L is a variable of type *list* and p is a linear polynomial. A term of type *list* will also be called a *list*. An *atom* is a formula of the form $r(t_1, \ldots, t_n)$ where r is an n-ary predicate symbol (with $n \geq 0$ and $r \notin \{=, <, \leq\}$) and, for $i = 1, \ldots, n$, t_i is either a linear polynomial or a list. An atom is *linear* if each variable occurs in it at most once. A *literal* is either an atom (i.e., a positive literal) or a negated atom (i.e., a negative literal). A *clause* C is a formula of the form: $A \leftarrow c \wedge L_1 \wedge \ldots \wedge L_m$, where: (i) A is an atom, (ii) c is a constraint, and (iii) $L_1, \ldots, L_m$ are literals. A is called the *head* of the clause, denoted $hd(C)$, and $c \wedge L_1 \wedge \ldots \wedge L_m$ is called the *body* of the clause, denoted $bd(C)$.

A *constraint logic program over lists of reals*, or simply a *program*, is a set of clauses. A program is *stratified* if no predicate depends negatively on itself [2]. Given a term or a formula f, $vars(f)$ denotes the set of variables occurring in f. Given a clause C, a variable V is said to be an *existential variable* of C if $V \in vars(bd(C)) - vars(hd(C))$.

The *definition* of a predicate p in a program P, denoted by $Def(p, P)$, is the set of the clauses of P whose head predicate is p. The *extended definition* of p in P, denoted by $Def^*(p, P)$, is the union of the definition of p and the definitions of all predicates in P on which p depends (positively or negatively). A program is *propositional* if every predicate occurring in the program is 0-ary. Obviously, if P is a propositional program then, for every predicate p, $M(P) \models p$ is decidable.

Definition 1 (*lr*-program). Let X denote a variable of type *real*, L a variable of type *list*, p a linear polynomial, r_1 and r_2 two predicate symbols, and c a constraint. An *lr-clause* is a clause defined as follows:

$$\begin{array}{lll} \textit{head term}: & h & ::= \; X \mid [\,] \mid [X|L] \\ \textit{body term}: & b & ::= \; p \mid L \\ \textit{lr-clause}: & C & ::= \; r_1(h_1,\ldots,h_k) \leftarrow c \\ & & \quad \mid \; r_1(h_1,\ldots,h_k) \leftarrow c \wedge r_2(b_1,\ldots,b_m) \\ & & \quad \mid \; r_1(h_1,\ldots,h_k) \leftarrow c \wedge \neg r_2(b_1,\ldots,b_m) \end{array}$$

where: (i) $vars(p) \neq \emptyset$, (ii) $r_1(h_1,\ldots,h_k)$ is a linear atom, and (iii) clause C has no existential variables. An *lr-program* is a finite set of *lr*-clauses. □

We assume that the following *lr*-clauses belong to every *lr*-program (but we will omit them when writing *lr*-programs):

$$\textit{list}([\,]) \leftarrow$$
$$\textit{list}([X|L]) \leftarrow \textit{list}(L)$$

The specific syntactic form of *lr*-programs is required for the automation of the transformation strategy we will introduce in Section 3. Here is an *lr*-program:

P_1: $\textit{sumlist}([\,], Y) \leftarrow Y = 0$
$\textit{sumlist}([X|L], Y) \leftarrow \textit{sumlist}(L,\, Y - X)$
$\textit{haspositive}([X|L]) \leftarrow X > 0$
$\textit{haspositive}([X|L]) \leftarrow \textit{haspositive}(L)$

The following definition introduces the class of programs and formulas which can be given in input to our proof method.

Definition 2 (Admissible Pair). Let P be an *lr*-program and φ a closed first order formula with no other connectives and quantifiers besides $\neg, \wedge$, and $\exists$. We say that $\langle P, \varphi\rangle$ is an *admissible pair* if: (i) every predicate symbol occurring in φ and different from $\leq, <, =$, also occurs in P, (ii) every predicate of arity n (>0) occurring in P and different from $\leq, <, =$, has at least one argument of type *list*, and (iii) for every proper subformula σ of φ, if σ is of the form $\neg\psi$, then either σ is a formula in $F_{\mathcal{R}}$ or σ has a free occurrence of a variable of type *list*. □

Conditions (ii) and (iii) of Definition 2 are needed to guarantee the soundness of our proof method (see Theorem 3).

Example 1. Let us consider the above program P_1 defining the predicates *sumlist* and *haspositive*, and the formula

$$\pi: \quad \forall L\, \forall Y\, ((\textit{sumlist}(L, Y) \wedge Y > 0) \rightarrow \textit{haspositive}(L))$$

which expresses the fact that if the sum of the elements of a list is positive then the list has at least one positive member. This formula can be rewritten as:

$$\pi_1: \quad \neg\exists L\, \exists Y\, (\textit{sumlist}(L, Y) \wedge Y > 0 \wedge \neg \textit{haspositive}(L))$$

The pair $\langle P_1, \pi_1\rangle$ is admissible. Indeed, the only proper subformula of π_1 of the form $\neg\psi$ is $\neg\textit{haspositive}(L)$ and the free variable L is of type *list*. □

In order to define the semantics of our logic programs we consider $\mathcal{LR}$-interpretations where: (i) the type *real* is mapped to the set of reals, (ii) the type *list* is mapped to the set of lists of reals, and (iii) the symbols $+$, $\cdot$, $=$, $<$, $\leq$, $[\,]$, and $[\_|\_]$ are mapped to the usual corresponding operations and relations on reals and lists of reals. The semantics of a stratified logic program P is assumed to be its *perfect $\mathcal{LR}$-model* $M(P)$, which is defined similarly to the perfect model of a stratified logic program [2,12,17] by considering $\mathcal{LR}$-interpretations, instead of Herbrand interpretations. Note that for every formula $\varphi \in F_{\mathcal{R}}$, we have that $\mathcal{R} \models \varphi$ iff for any $\mathcal{LR}$-interpretation $\mathcal{I}$, $\mathcal{I} \models \varphi$.

Now we present a transformation, called *Clause Form Transformation*, that allows us to derive stratified logic programs starting from formulas, called *statements*, of the form: $A \leftarrow \beta$, where A is an atom and β is a typed first order formula. Our transformation is a variant of the transformation proposed by Lloyd and Topor in [11]. When applying the Clause Form Transformation, we will use the following well known property which guarantees that existential quantification and negation can always be eliminated from first order formulas on reals.

Lemma 1 (Variable Elimination). *For any formula $\varphi \in F_{\mathcal{R}}$ there exist n (≥ 0) constraints $c_1, \ldots, c_n$ such that:* (i) $\mathcal{R} \models \forall(\varphi \leftrightarrow (c_1 \vee \ldots \vee c_n))$, *and* (ii) *every variable in vars$(c_1 \vee \ldots \vee c_n)$ occurs free in φ.*

In what follows we write $C[\gamma]$ to denote a formula where the subformula γ occurs as an *outermost conjunct*, that is, $C[\gamma] = \gamma_1 \wedge \gamma \wedge \gamma_2$ for some (possibly empty) conjunctions γ_1 and γ_2.

Clause Form Transformation.
Input: A statement S whose body has no other connectives and quantifiers besides $\neg, \wedge$, and $\exists$. *Output*: A set of clauses denoted $CFT(S)$.

(*Step* A) Starting from S, repeatedly apply the following rules A.1–A.5 until a set of clauses is generated.

(A.1) If $\gamma \in F_{\mathcal{R}}$ and γ is *not* a constraint, then replace $A \leftarrow C[\gamma]$ by the n statements $A \leftarrow C[c_1], \ldots, A \leftarrow C[c_n]$, where $c_1 \vee \ldots \vee c_n$, with $n \geq 0$, is a disjunction of constraints which is equivalent to γ. (The existence of such a disjunction is guaranteed by Lemma 1 above.)

(A.2) If $\gamma \notin F_{\mathcal{R}}$ then replace $A \leftarrow C[\neg\neg\gamma]$ by $A \leftarrow C[\gamma]$.

(A.3) If $\gamma \wedge \delta \notin F_{\mathcal{R}}$ then replace the statement $A \leftarrow C[\neg(\gamma \wedge \delta)]$ by the two statements $A \leftarrow C[\neg newp(V_1, \ldots, V_k)]$ and $newp(V_1, \ldots, V_k) \leftarrow \gamma \wedge \delta$, where *newp* is a new predicate and $V_1, \ldots, V_k$ are the variables which occur free in $\gamma \wedge \delta$.

(A.4) If $\gamma \notin F_{\mathcal{R}}$ then replace the statement $A \leftarrow C[\neg\exists V\, \gamma]$ by the two statements $A \leftarrow C[\neg newp(V_1, \ldots, V_k)]$ and $newp(V_1, \ldots, V_k) \leftarrow \gamma$, where *newp* is a new predicate and $V_1, \ldots, V_k$ are the variables which occur free in $\exists V\, \gamma$.

(A.5) If $\gamma \notin F_{\mathcal{R}}$ then replace $A \leftarrow C[\exists V\, \gamma]$ by $A \leftarrow C[\gamma\{V/V_1\}]$, where V_1 is a new variable.

(*Step* B) For every clause $A \leftarrow c \wedge G$ such that $L_1, \ldots, L_k$ are the variables of type *list* occurring in G, replace $A \leftarrow c \wedge G$ by $A \leftarrow c \wedge list(L_1) \wedge \ldots \wedge list(L_k) \wedge G$.

Example 2. The set $CFT(prop_1 \leftarrow \pi_1)$, where π_1 is the formula given in Example 1, consists of the following two clauses:

D_2 : $prop_1 \leftarrow \neg new_1$
D_1 : $new_1 \leftarrow Y > 0 \wedge list(L) \wedge sumlist(L, Y) \wedge \neg haspositive(L)$

(The subscripts of the names of these clauses follow the bottom-up order in which they will be processed by the UF_{lr} strategy we will introduce below.) □

By construction, we have that if $\langle P, \varphi \rangle$ is an admissible pair and *prop* is a new predicate symbol, then $P \cup CFT(prop \leftarrow \varphi)$ is a stratified program. The Clause Form Transformation is correct with respect to the perfect $\mathcal{LR}$-model semantics, as stated by the following theorem.

Theorem 1 (Correctness of CFT). *Let $\langle P, \varphi \rangle$ be an admissible pair. Then, $M(P) \models \varphi$ iff $M(P \cup CFT(prop \leftarrow \varphi)) \models prop$.*

In general, a clause in $CFT(prop \leftarrow \varphi)$ is *not* an *lr*-clause because, indeed, existential variables may occur in its body. The clauses of $CFT(prop \leftarrow \varphi)$ are called *typed-definitions*. They are defined as follows.

Definition 3 (Typed-Definition, Hierarchy). A *typed-definition* is a clause of the form: $r(V_1, \ldots, V_n) \leftarrow c \wedge list(L_1) \wedge \ldots \wedge list(L_k) \wedge G$ where: (i) $V_1, \ldots, V_n$ are distinct variables of type *real* or *list*, and (ii) $L_1, \ldots, L_k$ are the variables of type *list* that occur in G. A sequence $\langle D_1, \ldots, D_n \rangle$ of typed-definitions is said to be a *hierarchy* if for $i = 1, \ldots, n$, the predicate of $hd(D_i)$ does not occur in $\{bd(D_1), \ldots, bd(D_i)\}$. □

One can show that given a closed formula φ, the set $CFT(prop \leftarrow \varphi)$ of clauses can be ordered as a hierarchy $\langle D_1, \ldots, D_n \rangle$ of typed-definitions such that $Def(prop, \{D_1, \ldots, D_n\}) = \{D_k, D_{k+1}, \ldots, D_n\}$, for some k with $1 \leq k \leq n$.

3 The Unfold/Fold Proof Method

In this section we present the transformation strategy, called UF_{lr} (Unfold/Fold strategy for *lr*-programs), which defines our proof method for proving properties of *lr*-programs. Our strategy applies in an automatic way the transformation rules for stratified constraint logic programs presented in [8]. In particular, the UF_{lr} strategy makes use of the definition introduction, (positive and negative) unfolding, (positive) folding, and constraint replacement rules. (These rules extend the ones proposed in [6,12] where the unfolding of a clause with respect to a negative literal is not permitted.)

Given an admissible pair $\langle P, \varphi \rangle$, let us consider the stratified program $P \cup CFT(prop \leftarrow \varphi)$. The goal of our UF_{lr} strategy is to derive a program *TransfP*

such that $Def^*(prop, TransfP)$ is propositional and, thus, $M(TransfP) \models prop$ is decidable. We observe that, in order to achieve this goal, it is enough that the derived program *TransfP* is an *lr*-program, as stated by the following lemma, which follows directly from Definition 1.

Lemma 2. *Let P be an lr-program and p be a predicate occurring in P. If p is 0-ary then $Def^*(p, P)$ is a propositional program.*

As already said, the clauses in $CFT(prop \leftarrow \varphi)$ form a hierarchy $\langle D_1, \ldots, D_n \rangle$ of typed-definitions. The UF_{lr} strategy consists in transforming, for $i = 1, \ldots, n$, clause D_i into a set of *lr*-clauses. The transformation of D_i is performed by applying the following three substrategies, in this order: (i) *unfold*, which unfolds D_i with respect to the positive and negative literals occurring in its body, thereby deriving a set *Cs* of clauses, (ii) *replace-constraints*, which replaces the constraints appearing in the clauses of *Cs* by equivalent ones, thereby deriving a new set *Es* of clauses, and (iii) *define-fold*, which introduces a set *NewDefs* of new typed-definitions (which are not necessarily *lr*-clauses) and folds all clauses in *Es*, thereby deriving a set *Fs* of *lr*-clauses. Then each new definition in *NewDefs* is transformed by applying the above three substrategies, and the whole UF_{lr} strategy terminates when no new definitions are introduced. The substrategies *unfold*, *replace-constraints*, and *define-fold* will be described in detail below.

The UF_{lr} Transformation Strategy.
Input: An *lr*-program P and a hierarchy $\langle D_1, \ldots, D_n \rangle$ of typed-definitions.
Output: A set *Defs* of typed-definitions including $D_1, \ldots, D_n$, and an *lr*-program *TransfP* such that $M(P \cup Defs) = M(TransfP)$.

TransfP := *P*; *Defs* := $\{D_1, \ldots, D_n\}$;
for $i = 1, \ldots, n$ **do** *InDefs* := $\{D_i\}$;
  while *InDefs* $\neq \emptyset$ **do**
  unfold(*InDefs*, *TransfP*, *Cs*);
  replace-constraints(*Cs*, *Es*);
  define-fold(*Es*, *Defs*, *NewDefs*, *Fs*);
  TransfP := *TransfP* $\cup$ *Fs*; *Defs* := *Defs* $\cup$ *NewDefs*; *InDefs* := *NewDefs*;
  end-while;
  eval-props: for each predicate p such that $Def^*(p, TransfP)$ is propositional,
  if $M(TransfP) \models p$ **then** *TransfP* := (*TransfP* − *Def*(*p*, *TransfP*)) $\cup$ $\{p \leftarrow\}$
    else *TransfP* := (*TransfP* − *Def*(*p*, *TransfP*))
end-for

Our assumption that $\langle D_1, \ldots, D_n \rangle$ is a hierarchy ensures that, when transforming clause D_i, for $i = 1, \ldots, n$, we only need the clauses obtained after the transformation of $D_1, \ldots, D_{i-1}$. These clauses are those of the current value of *TransfP*.

The following *unfold* substrategy transforms a set *InDefs* of typed-definitions by first applying the unfolding rule with respect to each positive literal in the

body of a clause and then applying the unfolding rule with respect to each negative literal in the body of a clause. In the sequel, we will assume that the conjunction operator $\wedge$ is associative, commutative, idempotent, and with neutral element *true*. In particular, the order of the conjuncts will *not* be significant.

The *unfold* Substrategy.
Input: An *lr*-program *Prog* and a set *InDefs* of typed-definitions.
Output: A set *Cs* of clauses.

Initially, no literal in the body of a clause of *InDefs* is marked as 'unfolded'.

Positive Unfolding: **while** there exists a clause C in *InDefs* of the form $H \leftarrow c \wedge G_L \wedge A \wedge G_R$, where A is an atom which is not marked as 'unfolded' **do**
Let C_1: $K_1 \leftarrow c_1 \wedge B_1$, ..., C_m: $K_m \leftarrow c_m \wedge B_m$ be all clauses of program *Prog* (where we assume $vars(Prog) \cap vars(C) = \emptyset$) such that, for $i = 1, \ldots, m$, (i) there exists a most general unifier ϑ_i of A and K_i, and (ii) the constraint $(c \wedge c_i)\vartheta_i$ is satisfiable. Let U be the following set of clauses:

$$U = \{(H \leftarrow c \wedge c_1 \wedge G_L \wedge B_1 \wedge G_R)\vartheta_1, \ldots, (H \leftarrow c \wedge c_m \wedge G_L \wedge B_m \wedge G_R)\vartheta_m\}$$

Let W be the set of clauses derived from U by removing all clauses of the form

$$H \leftarrow c \wedge G_L \wedge A \wedge \neg A \wedge G_R$$

Inherit the markings of the literals in the body of the clauses of W from those of C, and mark as 'unfolded' the literals $B_1\vartheta_1, \ldots, B_m\vartheta_m$;
$InDefs := (InDefs - \{C\}) \cup W$;
end-while;

Negative Unfolding: **while** there exists a clause C in *InDefs* of the form $H \leftarrow c \wedge G_L \wedge \neg A \wedge G_R$, where $\neg A$ is a literal which is not marked as 'unfolded' **do**
Let C_1: $K_1 \leftarrow c_1 \wedge B_1, \ldots, C_m$: $K_m \leftarrow c_m \wedge B_m$ be all clauses of program *Prog* (where we assume that $vars(Prog) \cap vars(C) = \emptyset$) such that, for $i = 1, \ldots, m$, there exists a most general unifier ϑ_i of A and K_i. By our assumptions on *Prog* and on the initial value of *InDefs*, and as a result of the previous *Positive Unfolding* phase, we have that, for $i = 1, \ldots, m$, B_i is either the empty conjunction *true* or a literal and $A = K_i\vartheta_i$. Let U be the following set of statements:

$$U = \{H \leftarrow c \wedge d_1\vartheta_1 \wedge \ldots \wedge d_m\vartheta_m \wedge G_L \wedge L_1\vartheta_1 \wedge \ldots \wedge L_m\vartheta_m \wedge G_R \mid$$
(i) for $i = 1, \ldots, m$, either ($d_i = c_i$ and $L_i = \neg B_i$) or ($d_i = \neg c_i$ and $L_i = true$), and (ii) $c \wedge d_1\vartheta_1 \wedge \ldots \wedge d_m\vartheta_m$ is satisfiable$\}$

Let W be the set of clauses derived from U by applying as long as possible the following rules:

- remove $H \leftarrow c \wedge G_L \wedge \neg true \wedge G_R$ and $H \leftarrow c \wedge G_L \wedge A \wedge \neg A \wedge G_R$
- replace $\neg\neg A$ by A, $\neg(p_1 \leq p_2)$ by $p_2 < p_1$, and $\neg(p_1 < p_2)$ by $p_2 \leq p_1$
- replace $H \leftarrow c_1 \wedge \neg(p_1 = p_2) \wedge c_2 \wedge G$ by $H \leftarrow c_1 \wedge p_1 < p_2 \wedge c_2 \wedge G$
  $H \leftarrow c_1 \wedge p_2 < p_1 \wedge c_2 \wedge G$

Inherit the markings of the literals in the body of the clauses of W from those of C, and mark as 'unfolded' the literals $L_1\vartheta_1, \ldots, L_m\vartheta_m$;
$InDefs := (InDefs - \{C\}) \cup W$;

end-while;
$Cs := InDefs$.

Negative Unfolding is best explained through an example. Let us consider a program consisting of the clauses C: $H \leftarrow c \wedge \neg A$, $A \leftarrow c_1 \wedge B_1$, and $A \leftarrow c_2 \wedge B_2$. The negative unfolding of C w.r.t. $\neg A$ gives us the following four clauses:

$H \leftarrow c \wedge \neg c_1 \wedge \neg c_2$
$H \leftarrow c \wedge c_1 \wedge \neg c_2 \wedge \neg B_1$
$H \leftarrow c \wedge \neg c_1 \wedge c_2 \wedge \neg B_2$
$H \leftarrow c \wedge c_1 \wedge c_2 \wedge \neg B_1 \wedge \neg B_2$

whose conjunction is equivalent to $H \leftarrow c \wedge \neg((c_1 \wedge B_1) \vee (c_2 \wedge B_2))$.

Example 3. Let us consider the program-property pair $\langle P_1, \pi_1 \rangle$ of Example 1. In order to prove that $M(P_1) \models \pi_1$, we apply the UF_{lr} strategy starting from the hierarchy $\langle D_1, D_2 \rangle$ of typed-definitions of Example 2. During the first execution of the body of the for-loop of that strategy, the *unfold* substrategy is applied, as we now indicate, by using as input the program P_1 and the set $\{D_1\}$ of clauses. We have the following positive and negative unfolding steps.

Positive Unfolding. By unfolding clause D_1 w.r.t. $list(L)$ and then unfolding the resulting clauses w.r.t. $sumlist(L, Y)$, we get:

C_1: $new_1 \leftarrow Y > 0 \wedge list(L) \wedge sumlist(L, Y - X) \wedge \neg haspositive([X|L])$

Negative Unfolding. By unfolding clause C_1 w.r.t. $\neg haspositive([X|L])$, we get:

C_2: $new_1 \leftarrow Y > 0 \wedge X \leq 0 \wedge list(L) \wedge sumlist(L, Y - X) \wedge \neg haspositive(L)$ $\square$

The correctness of the *unfold* substrategy follows from the fact that the positive and negative unfoldings are performed according to the rules presented in [8]. The termination of that substrategy is due to the fact that the number of literals which are not marked as 'unfolded' and which occur in the body of a clause, decreases when that clause is unfolded. Thus, we have the following result.

Lemma 3. *Let Prog be an lr-program and let InDefs be a set of typed-definitions such that the head predicates of the clauses of InDefs do not occur in Prog. Then, given the inputs Prog and InDefs, the unfold substrategy terminates and returns a set Cs of clauses such that* $M(Prog \cup InDefs) = M(Prog \cup Cs)$.

The *replace-constraints* substrategy derives from a set Cs of clauses a new set Es of clauses by applying equivalences between existentially quantified disjunctions of constraints. We use the following two rules: *project* and *clause split*.

Given a clause $H \leftarrow c \wedge G$, the *project* rule eliminates all variables that occur in c and do not occur elsewhere in the clause. Thus, *project* returns a new clause $H \leftarrow d \wedge G$ such that $\mathcal{R} \models \forall((\exists X_1 \ldots \exists X_k\, c) \leftrightarrow d)$, where: (i) $\{X_1, \ldots, X_k\} = vars(c) - vars(\{H, G\})$, and (ii) $vars(d) \subseteq vars(c) - \{X_1, \ldots, X_k\}$. In our prototype theorem prover (see Section 5), the *project* rule is implemented by using a variant of the Fourier-Motzkin Elimination algorithm [1].

The *clause split* rule replaces a clause C by two clauses C_1 and C_2 such that, for $i = 1, 2$, the number of occurrences of existential variables in C_i is less

than the number of occurrences of existential variables in C. The clause split rule applies the following property, which expresses the fact that $\langle \mathcal{R}, \leq \rangle$ is a linear order: $\mathcal{R} \models \forall X \forall Y\, (X < Y \vee Y \leq X)$. For instance, a clause of the form $H \leftarrow Z \leq X \wedge Z \leq Y \wedge G$, where Z is an existential variable occurring in the conjunction G of literals and X and Y are not existential variables, is replaced by the two clauses $H \leftarrow Z \leq X \wedge X < Y \wedge G$ and $H \leftarrow Z \leq Y \wedge Y \leq X \wedge G$. The decrease of the number of occurrences of existential variables guarantees that we can apply the clause split rule a finite number of times only.

The *replace-constraints* **Substrategy.**
Input: A set Cs of clauses. *Output*: A set Es of clauses.

• *Introduce Equations.* (A) From Cs we derive a new set R_1 of clauses by applying as long as possible the following two rules, where p denotes a linear polynomial which is not a variable, and Z denotes a fresh new variable:

(R.1) $H \leftarrow c \wedge G_L \wedge r(\ldots, p, \ldots) \wedge G_R$ is replaced by
$H \leftarrow c \wedge Z = p \wedge G_L \wedge r(\ldots, Z, \ldots) \wedge G_R$

(R.2) $H \leftarrow c \wedge G_L \wedge \neg r(\ldots, p, \ldots) \wedge G_R$ is replaced by
$H \leftarrow c \wedge Z = p \wedge G_L \wedge \neg r(\ldots, Z, \ldots) \wedge G_R$

(B) From R_1 we derive a new set R_2 of clauses by applying to every clause C in R_1 the following rule. Let C be of the form $H \leftarrow c \wedge G$. Suppose that $\mathcal{R} \models \forall\, (c \leftrightarrow (X_1 = p_1 \wedge X_n = p_n \wedge d))$, where: (i) $X_1, \ldots, X_n$ are existential variables of C, (ii) $vars(X_1 = p_1 \wedge \ldots \wedge X_n = p_n \wedge d) \subseteq vars(c)$, (iii) $\{X_1, \ldots, X_n\} \cap vars(\{p_1, \ldots, p_n, d\}) = \emptyset$. Then we replace C by $H \leftarrow X_1 = p_1 \wedge \ldots \wedge X_n = p_n \wedge d \wedge G$.

• *Project.* We derive a new set R_3 of clauses by applying to every clause in R_2 the *project* rule.

• *Clause Split.* From R_3 we derive a new set R_4 of clauses by applying as long as possible the following rule. Let C be a clause of the form $H \leftarrow c_1 \wedge c_2 \wedge c \wedge G$ (modulo commutativity of $\wedge$). Let E be the set of existential variables of C. Let $X \in E$ and let d_1 and d_2 be two inequations such that $\mathcal{R} \models \forall\, ((c_1 \wedge c_2) \leftrightarrow (d_1 \wedge d_2))$. Suppose that: (i) $d_1 \wedge d_2$ is of one of the following six forms:

$X \leq p_1 \wedge X \leq p_2$	$X \leq p_1 \wedge X < p_2$	$X < p_1 \wedge X < p_2$
$p_1 \leq X \wedge p_2 \leq X$	$p_1 \leq X \wedge p_2 < X$	$p_1 < X \wedge p_2 < X$

and (ii) $(vars(p_1) \cup vars(p_2)) \cap E = \emptyset$.
Then C is replaced by the following two clauses: C_1: $H \leftarrow d_1 \wedge p_1 < p_2 \wedge c \wedge G$ and C_2: $H \leftarrow d_2 \wedge p_2 \leq p_1 \wedge c \wedge G$, and then each clause in $\{C_1, C_2\}$ with an unsatisfiable constraint in its body is removed.

• *Eliminate Equations.* From R_4 we derive the new set Es of clauses by applying to every clause C in R_4 the following rule. If C is of the form $H \leftarrow X_1 = p_1 \wedge \ldots \wedge X_n = p_n \wedge d \wedge G$ where $\{X_1, \ldots, X_n\} \cap vars(\{p_1, \ldots p_n, d\}) = \emptyset$, then C is replaced by $(H \leftarrow d \wedge G)\{X_1/p_1, \ldots, X_n/p_n\}$.

The transformation described at Point (A) of *Introduce Equations* allows us to treat all polynomials occurring in the body of a clause in a uniform way as arguments of constraints. The transformation described at Point (B) of *Introduce Equations* identifies those existential variables which can be eliminated during the final *Eliminate Equations* transformation. That elimination is performed by substituting, for $i=1,\ldots,n$, the variable X_i by the polynomial p_i.

Example 4. By applying the *replace-constraints* substrategy, clause C_2 of Example 3 is transformed as follows. By introducing equations we get:

C_3: $new_1 \leftarrow Y>0 \wedge X\leq 0 \wedge Z=Y-X \wedge list(L) \wedge sumlist(L,Z) \wedge \neg haspositive(L)$

Then, by applying the *project* transformation, we get:

C_4: $new_1 \leftarrow Z>0 \wedge list(L) \wedge sumlist(L,Z) \wedge \neg haspositive(L)$ □

The correctness of the *replace-constraints* substrategy is a straightforward consequence of the fact that the *Introduce Equations*, *Project*, *Clause Split*, and *Eliminate Equations* transformations are performed by using the rule of *replacement based on laws* presented in [8]. The termination of *Introduce Equations* and *Eliminate Equations* is obvious. The termination of *Project* is based on the termination of the specific algorithm used for variable elimination (e.g., Fourier-Motzkin algorithm). As already mentioned, the termination of *Clause Split* is due to the fact that at each application of this transformation the number of occurrences of existential variables decreases. Thus, we get the following lemma.

Lemma 4. *For any program Prog and set Cs ⊆ Prog of clauses, the replace-constraints substrategy with input Cs terminates and returns a set Es of clauses such that* $M(Prog) = M((Prog - Cs) \cup Es)$.

The *define-fold* substrategy eliminates all existential variables in the clauses of the set *Es* obtained after the *unfold* and *replace-constraints* substrategies. This elimination is done by folding all clauses in *Es* that contain existential variables. In order to make these folding steps we use the typed-definitions in *Defs* and, if necessary, we introduce new typed-definitions which we add to the set *NewDefs*.

The *define-fold* **Substrategy.**
Input: A set *Es* of clauses and a set *Defs* of typed-definitions.
Output: A set *NewDefs* of typed-definitions and a set *Fs* of *lr*-clauses.

Initially, both *NewDefs* and *Fs* are empty.
for each clause C: $H \leftarrow c \wedge G$ in *Es* **do**
if C is an *lr*-clause **then** $Fs := Fs \cup \{C\}$ **else**

- *Define.* Let E be the set of existential variables of C. We consider a clause *NewD* of the form $newp(V_1,\ldots,V_m) \leftarrow d \wedge B$ constructed as follows:
(1) let c be of the form $c_1 \wedge c_2$, where $vars(c_1) \cap E = \emptyset$ and for every atomic constraint a occurring in c_2, $vars(a) \cap E \neq \emptyset$; let $d \wedge B$ be the most general (modulo variants) conjunction of constraints and literals such that there exists a substitution ϑ with the following properties: (i) $(d \wedge B)\vartheta = c_2 \wedge G$, and (ii) for

each binding V/p in ϑ, V is a variable not occurring in C, $vars(p) \neq \emptyset$, and $vars(p) \cap E = \emptyset$;
(2) *newp* is a new predicate symbol;
(3) $\{V_1, \ldots, V_m\} = vars(d \wedge B) - E$.
NewD is added to *NewDefs*, unless in *Defs* there exists a typed-definition D which is equal to *NewD*, modulo the name of the head predicate, the names of variables, equivalence of constraints, and the order and multiplicity of literals in the body. If such a clause D belongs to *Defs* and no other clause in *Defs* has the same head predicate as D, then we assume that $NewD = D$.

- *Fold.* Clause C is folded using clause *NewD* as follows:
$Fs := Fs \cup \{H \leftarrow c_1 \wedge newp(V_1, \ldots, V_m)\vartheta\}$.
end-for

Example 5. Let us consider the clause C_4 derived at the end of Example 4. The *Define* phase produces a typed-definition which is a variant of the typed-definition D_1 introduced at the beginning of the application of the strategy (see Example 2). Thus, C_4 is folded using clause D_1, and we get the clause:

C_5: $new_1 \leftarrow new_1$

Let us now describe how the proof of $M(P_1) \models \pi_1$ proceeds. The program *TransfP* derived so far consists of clause C_5 together with the clauses defining the predicates *list*, *sumlist*, and *haspositive*. Thus, $Def^*(new_1, TransfP)$ consists of clause C_5 only, which is propositional and, by *eval-props*, we remove C_5 from *TransfP* because $M(TransfP) \not\models new_1$. The strategy continues by considering the typed definition D_2 (see Example 2). By unfolding D_2 with respect to $\neg new_1$ we get the final program *TransfP*, which consists of the clause $prop_1 \leftarrow$ together with the clauses for *list*, *sumlist*, and *haspositive*. Thus, $M(TransfP) \models prop_1$ and, therefore, $M(P_1) \models \pi_1$. □

The proof of correctness for the *define-fold* substrategy is more complex than the proofs for the other substrategies. The correctness results for the unfold/fold transformations presented in [8] guarantee the correctness of a folding transformation if each typed-definition used for folding is unfolded w.r.t. a positive literal during the application of the UF_{lr} transformation strategy. The fulfillment of this condition is ensured by the following two facts: (1) by the definition of an admissible pair and by the definition of the Clause Form Transformation, each typed-definition has at least one positive literal in its body (indeed, by Condition (iii) of Definition 2 each negative literal in the body of a typed-definition has at least one variable of type *list* and, therefore, the body of the typed-definition has at least one *list* atom), and (2) in the *Positive Unfolding* phase of the *unfold* substrategy, each typed-definition is unfolded w.r.t. all positive literals.

Note that the set *Fs* of clauses derived by the *define-fold* substrategy is a set of *lr*-clauses. Indeed, by the *unfold* and *replace-constraints* substrategies, we derive a set *Es* of clauses of the form $r(h_1, \ldots, h_k) \leftarrow c \wedge G$, where $h_1, \ldots, h_k$ are head terms (see Definition 1). By folding we derive clauses of the form

$r(h_1, \ldots, h_k) \leftarrow c_1 \wedge newp(V_1, \ldots, V_m)\vartheta$

where $vars(c_1 \wedge newp(V_1, \ldots, V_m)\vartheta) \subseteq vars(r(h_1, \ldots, h_k))$, and for $i = 1, \ldots, m$, $vars(V_i\vartheta) \neq \emptyset$ (by the conditions at Points (1)–(3) of the *Define* phase). Hence, all clauses in *Fs* are *lr*-clauses.

The termination of the *define-fold* substrategy is obvious, as each clause is folded at most once. Thus, we have the following result.

Lemma 5. *During the UF_{lr} strategy, if the define-fold substrategy takes as inputs the set Es of clauses and the set Defs of typed-definitions, then this substrategy terminates and returns a set NewDefs of typed-definitions and a set Fs of lr-clauses such that $M(TransfP \cup Es \cup NewDefs) = M(TransfP \cup Fs \cup NewDefs)$.*

By using Lemmata 3, 4, and 5 we get the following correctness result for the UF_{lr} strategy.

Theorem 2. *Let P be an lr-program and $\langle D_1, \ldots, D_n \rangle$ a hierarchy of typed-definitions. Suppose that the UF_{lr} strategy with inputs P and $\langle D_1, \ldots, D_n \rangle$ terminates and returns a set Defs of typed-definitions and a program TransfP. Then:* (i) *TransfP is an lr-program and* (ii) $M(P \cup Defs) = M(TransfP)$.

Now, we are able to prove the soundness of the unfold/fold proof method.

Theorem 3 (Soundness of the Unfold/Fold Proof Method). *Let $\langle P, \varphi \rangle$ be an admissible pair and let $\langle D_1, \ldots, D_n \rangle$ be the hierarchy of typed-definitions obtained from $prop \leftarrow \varphi$ by the Clause Form Transformation. If the UF_{lr} strategy with inputs P and $\langle D_1, \ldots, D_n \rangle$ terminates and returns a program TransfP, then:*

$$M(P) \models \varphi \quad \textit{iff} \quad (prop \leftarrow) \ \in TransfP$$

Proof. By Theorem 1 and Point (ii) of Theorem 2, we have that $M(P) \models \varphi$ iff $M(TransfP) \models prop$. By Point (i) of Theorem 2 and Lemma 2 we have that $Def^*(prop, TransfP)$ is propositional. Since the last step of the UF_{lr} strategy is an application of the *eval-props* transformation, we have that $Def^*(prop, TransfP)$ is either the singleton $\{prop \leftarrow\}$, if $M(TransfP) \models prop$, or the empty set, if $M(TransfP) \not\models prop$. □

4 A Complete Example

As an example of application of our transformation strategy for proving properties of constraint logic programs we consider the *lr*-program *Member* and the property φ given in the Introduction. The formula φ is rewritten as follows:

$\varphi_1 : \ \neg\exists L\, \neg\exists U\, \neg\exists X\, (X > U \wedge member(X, L))$

The pair $\langle Member, \varphi_1 \rangle$ is admissible. By applying the Clause Form Transformation starting from the statement $prop \leftarrow \varphi_1$, we get the following clauses:

D_4: $prop \leftarrow \neg p$
D_3: $p \leftarrow list(L) \wedge \neg q(L)$
D_2: $q(L) \leftarrow list(L) \wedge \neg r(L, U)$
D_1: $r(L, U) \leftarrow X > U \wedge list(L) \wedge member(X, L)$

where $\langle D_1, D_2, D_3, D_4\rangle$ is a hierarchy of typed-definitions. Note that the three nested negations in φ_1 generate the three atoms p, $q(L)$, and $r(L,U)$ with their typed-definitions D_3, D_2, and D_1, respectively. The arguments of p, q, and r are the free variables of the corresponding subformulas of φ_1. For instance, $r(L,U)$ corresponds to the subformula $\exists X\,(X > U \wedge \mathit{member}(X,L))$ which has L and U as free variables. Now we apply the UF_{lr} strategy starting from the program *Member* and the hierarchy $\langle D_1, D_2, D_3, D_4\rangle$.

• *Execution of the for-loop with* $i = 1$. We have: $\mathit{InDefs} = \{D_1\}$. By unfolding clause D_1 w.r.t. the atoms $\mathit{list}(L)$ and $\mathit{member}(X,L)$ we get:

1.1 $r([X|T],U) \leftarrow X > U \wedge \mathit{list}(T)$
1.2 $r([X|T],U) \leftarrow Y > U \wedge \mathit{list}(T) \wedge \mathit{member}(Y,T)$

No replacement of constraints is performed. Then, by folding clause 1.2 using D_1, we get:

1.3 $r([X|T],U) \leftarrow r(T,U)$

After the *define-fold* substrategy the set *Fs* of clauses is $\{1.1, 1.3\}$, and at this point the program *TransfP* is *Member* $\cup\, \{1.1, 1.3\}$. No new definitions are introduced and, thus, $\mathit{InDefs} = \emptyset$ and the while-loop terminates. *eval-props* is not performed because the predicate r is not propositional.

• *Execution of the for-loop with* $i = 2$. We have: $\mathit{InDefs} = \{D_2\}$. We unfold clause D_2 w.r.t. $\mathit{list}(L)$ and $\neg r(L,U)$, we get:

2.1 $q([\,]) \leftarrow$
2.2 $q([X|T]) \leftarrow X \leq U \wedge \mathit{list}(T) \wedge \neg r(T,U)$

No replacement of constraints is performed. Then we introduce the new definition:

2.3 $q_1(X,T) \leftarrow X \leq U \wedge \mathit{list}(T) \wedge \neg r(T,U)$

and we fold clause 2.2 using clause 2.3. We get:

2.4 $q([X|T]) \leftarrow q_1(X,T)$

Since $\mathit{NewDefs} = \mathit{InDefs} = \{2.3\}$ we execute again the body of the while-loop. By unfolding clause 2.3 w.r.t. $\mathit{list}(T)$ and $\neg r(T,U)$, we get:

2.5 $q_1(X,[\,]) \leftarrow$
2.6 $q_1(X,[Y|T]) \leftarrow X \leq U \wedge Y \leq U \wedge \mathit{list}(T) \wedge \neg r(T,U)$

By applying *replace-constraints*, clause 2.6 generates the following two clauses:

2.6.1 $q_1(X,[Y|T]) \leftarrow X > Y \wedge X \leq U \wedge \mathit{list}(T) \wedge \neg r(T,U)$
2.6.2 $q_1(X,[Y|T]) \leftarrow X \leq Y \wedge Y \leq U \wedge \mathit{list}(T) \wedge \neg r(T,U)$

By folding clauses 2.6.1 and 2.6.2 using clause 2.3, we get:

2.7 $q_1(X,[Y|T]) \leftarrow X > Y \wedge q_1(X,T)$
2.8 $q_1(X,[Y|T]) \leftarrow X \leq Y \wedge q_1(Y,T)$

At this point the program *TransfP* is *Member* $\cup \{1.1, 1.3, 2.1, 2.4, 2.5, 2.7, 2.8\}$. No new definitions are introduced and, thus, the while-loop terminates. *eval-props* is not performed because the predicates q and q_1 are not propositional.

• *Execution of the for-loop with* $i = 3$. We have: $\mathit{InDefs} = \{D_3\}$. By unfolding clause D_3 w.r.t. $\mathit{list}(L)$ and $\neg q(L)$, we get:

3.1 $p \leftarrow \mathit{list}(T) \wedge \neg q_1(X,T)$

No replacement of constraints is performed. The following new definition:

3.2 $p_1 \leftarrow \mathit{list}(T) \wedge \neg q_1(X,T)$

is introduced. Then by folding clause 3.1 using clause 3.2, we get:

3.3 $p \leftarrow p_1$

Since $\mathit{NewDefs} = \mathit{InDefs} = \{3.2\}$ we execute again the body of the while-loop. By unfolding clause 3.2 w.r.t. $\mathit{list}(T)$ and $\neg q_1(X,T)$, we get:

3.4 $p_1 \leftarrow X > Y \wedge \mathit{list}(T) \wedge \neg q_1(X,T)$

3.5 $p_1 \leftarrow X \leq Y \wedge \mathit{list}(T) \wedge \neg q_1(Y,T)$

Since the variable Y occurring in the constraints $X > Y$ and $X \leq Y$ is existential, we apply the *project* rule to clauses 3.4 and 3.5 and we get the following clause:

3.6 $p_1 \leftarrow \mathit{list}(T) \wedge \neg q_1(X,T)$

This clause can be folded using clause 3.2, thereby deriving the following clause:

3.7 $p_1 \leftarrow p_1$

Clauses 3.3 and 3.7 are added to *TransfP*. Since the predicates p and p_1 are both propositional, we execute *eval-props*. We have that: (i) $M(\mathit{TransfP}) \not\models p_1$ and (ii) $M(\mathit{TransfP}) \not\models p$. Thus, clauses 3.3 and 3.7 are removed from *TransfP*. Hence, $\mathit{TransfP} = \mathit{Member} \cup \{1.1, 1.3, 2.1, 2.4, 2.5, 2.7, 2.8\}$.

• *Execution of the for-loop with* $i = 4$. We have: $\mathit{InDefs} = \{D_4\}$. By unfolding clause D_4 w.r.t. $\neg p$, we get the clause:

4. $\mathit{prop} \leftarrow$

This clause shows that, as expected, property φ holds for any finite list of reals.

5 Experimental Results

We have implemented our proof method by using the MAP transformation system [13] running under SICStus Prolog on a 900MHz Power PC. Constraint satisfaction and entailment were performed using the clp(r) module of SICStus. Our prototype has automatically proved the properties listed in the following table, where the predicates *member*, *sumlist*, and *haspositive* are defined as shown in Sections 1 and 2, and the other predicates have the following meanings: (i) $\mathit{ord}(L)$ holds iff L is a list of the form $[a_1, \ldots, a_n]$ and for $i = 1, \ldots, n-1$, $a_i \leq a_{i+1}$, (ii) $\mathit{sumzip}(L, M, N)$ holds iff L, M, and N are lists of the form $[a_1, \ldots, a_n]$, $[b_1, \ldots, b_n]$, and $[a_1 + b_1, \ldots, a_n + b_n]$, respectively, and (iii) $\mathit{leqlist}(L, M)$ holds iff L and M are lists of the form $[a_1, \ldots, a_n]$ and $[b_1, \ldots, b_n]$, respectively, and for $i = 1, \ldots, n$, $a_i \leq b_i$. We do not write here the *lr*-programs which define the predicates $\mathit{ord}(L)$, $\mathit{sumzip}(L, M, N)$, and $\mathit{leqlist}(L, M)$.

Property	Time
$\forall L\, \exists M\, \forall Y\, (\mathit{member}(Y, L) \rightarrow Y \leq M)$	140 ms
$\forall L\, \forall Y\, ((\mathit{sumlist}(L, Y) \wedge Y > 0) \rightarrow \mathit{haspositive}(L))$	170 ms
$\forall L\, \forall Y\, ((\mathit{sumlist}(L, Y) \wedge Y > 0) \rightarrow \exists X(\mathit{member}(X, L) \wedge X > 0))$	160 ms
$\forall L\, \forall M\, \forall N\, ((\mathit{ord}(L) \wedge \mathit{ord}(M) \wedge \mathit{sumzip}(L, M, N)) \rightarrow \mathit{ord}(N))$	160 ms
$\forall L\, \forall M\, ((\mathit{leqlist}(L, M) \wedge \mathit{sumlist}(L, X) \wedge \mathit{sumlist}(M, Y)) \rightarrow X \leq Y)$	50 ms

6 Related Work and Conclusions

We have presented a method for proving first order properties of constraint logic programs based on unfold/fold program transformations, and we have shown that the ability of unfold/fold transformations to eliminate existential variables [16] can be turned into a useful theorem proving method. We have provided a fully automatic strategy for the class of *lr*-programs, which are programs acting on reals and finite lists of reals, with constraints as linear equations and inequations over reals. The choice of lists is actually a simplifying assumption we have made and we believe that the extension of our method to any finitely generated data structure is quite straightforward. However, the use of constraints over the reals is an essential feature of our method, because quantifier elimination from constraints is a crucial subprocedure of our transformation strategy.

The first order properties of *lr*-programs are undecidable (and not even semi-decidable), because one can encode every partial recursive function as an *lr*-program without list arguments. As a consequence our proof method is necessarily incomplete. We have implemented the proof method based of program transformation and we have proved some simple, yet non-trivial, properties. As the experiments show, the performance of our method is encouraging.

Our method is an extension of the method presented in [15] which considers logic programs without constraints. The addition of constraints is a very relevant feature, because it provides more expressive power and, as already mentioned, we may use special purpose theorem provers for checking constraint satisfaction and for quantifier elimination. Our method can also be viewed as an extension of other techniques based on unfold/fold transformations for proving equivalences of predicates [14,19], and indeed, our method can deal with a class of first order formulas which properly includes equivalences.

Some papers have proposed transformational techniques to prove propositional temporal properties of finite and/or infinite state systems (see, for instance, [7,10,19]). Since propositional temporal logic can be encoded in first order logic, in principle these techniques can be viewed as instances of the unfold/fold proof method presented here.

However, it should be noted that the techniques described in [7,10,19] have their own peculiarities because they are tailored to the specific problem of verifying concurrent systems.

Finally, we think that a direct comparison of the power of our proof method with that of traditional theorem provers is somewhat inappropriate. The techniques used in those provers are very effective and are the result of a well established line of research (see, for instance, [3] for a survey on the automation of mathematical induction). However, our approach has its novelty and is based on principles which have not been explored in the field of theorem proving. In particular, the idea of making inductive proofs by unfold/fold transformations for eliminating quantifiers, has not yet been investigated within the theorem proving community.

7 Acknowledgments

We would like to thank the anonymous referees for their helpful comments and suggestions.

References

1. K. R. Apt. *Principles of Constraint Programming*. Cambridge Univ. Press, 2003.
2. K. R. Apt and R. N. Bol. Logic programming and negation: A survey. *Journal of Logic Programming*, 19, 20:9–71, 1994.
3. A. Bundy. The automation of proof by mathematical induction. In *Handbook of Automated Reasoning*, volume I, pages 845–911. North Holland, 2001.
4. R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, January 1977.
5. B. Courcelle. Equivalences and transformations of regular systems – applications to recursive program schemes and grammars. *Theor. Comp. Sci.*, 42:1–122, 1986.
6. S. Etalle and M. Gabbrielli. Transformations of CLP modules. *Theoretical Computer Science*, 166:101–146, 1996.
7. F. Fioravanti, A. Pettorossi, and M. Proietti. Verifying CTL properties of infinite state systems by specializing constraint logic programs. In *Proceedings VCL'01, Florence, Italy*, pages 85–96. University of Southampton, UK, 2001.
8. F. Fioravanti, A. Pettorossi, and M. Proietti. Transformation rules for locally stratified constraint logic programs. In *Program Development in Computational Logic*, LNCS 3049, pages 292–340. Springer, 2004.
9. L. Kott. The McCarthy's induction principle: 'oldy' but 'goody'. *Calcolo*, 19(1):59–69, 1982.
10. 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*, LNCS 1817, pages 63–82. Springer, 1999.
11. J. W. Lloyd. *Foundations of Logic Programming*. Springer, 1987. 2nd Edition.
12. M. J. Maher. A transformation system for deductive database modules with perfect model semantics. *Theoretical Computer Science*, 110:377–403, 1993.
13. The MAP System. `http://www.iasi.cnr.it/~proietti/system.html`.
14. 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.
15. A. Pettorossi and M. Proietti. Perfect model checking via unfold/fold transformations. In *Proc. CL 2000, London, UK*, LNAI 1861, pp. 613–628. Springer, 2000.
16. M. Proietti and A. Pettorossi. Unfolding-definition-folding, in this order, for avoiding unnecessary variables in logic programs. *Theor. Comp. Sci.*, 142(1):89–124, 1995.
17. 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.
18. M. O. Rabin. Decidable theories. In Jon Barwise, editor, *Handbook of Mathematical Logic*, pages 595–629. North-Holland, 1977.
19. A. Roychoudhury, K. Narayan Kumar, C. R. Ramakrishnan, I. V. Ramakrishnan, and S. A. Smolka. Verification of parameterized systems using logic program transformations. In *Proc. TACAS 2000*, LNCS 1785, pp. 172–187. Springer, 2000.
20. H. Tamaki and T. Sato. Unfold/fold transformation of logic programs. In S.-Å. Tärnlund, ed., *Proceedings of ICLP '84*, pages 127–138, Uppsala, Sweden, 1984.