A. Pettorossi, M. Proietti

PERFECT MODEL CHECKING VIA UNFOLD/FOLD TRANSFORMATIONS

R. 513 Novembre 1999

Alberto Pettorossi  – Dipartimento di Informatica, Sistemi e Produzione, Università di Roma Tor Vergata, Via di Tor Vergata, I-00133 Roma, Italy, and Istituto di Analisi dei Sistemi ed Informatica del CNR, Viale Manzoni 30, I-00185 Roma, Italy.
Email: adp@iasi rm.cn r.it. URL: http://www.iasi rm.cn r.it/ adp.

Maurizio Proietti  – Istituto di Analisi dei Sistemi ed Informatica del CNR, Viale Manzoni 30, I-00185 Roma, Italy. Email: proietti@iasi rm.cn r.it.
URL: http://www.iasi rm.cn r.it/proietti.

This work has been partially supported by MURST Progetto Cofinanziato ‘Tecniche Formali per la Specifica, l’Analisi, la Verifica, la Sintesi e la Trasformazione di Sistemi Software’ (Italy) and Progetto Coordinato CNR ‘Verifica, Analisi e Trasformazione dei Programmi Logici’ (Italy).

ISSN: 1128–3378
Abstract

We show how program transformation rules and strategies may be used for proving the satisfiability of first order formulas in some classes of models. In particular, we propose a technique for showing that a closed first order formula $\varphi$ holds in the perfect model $M(P)$ of a logic program $P$ with locally stratified negation. This property is denoted by $M(P) \models \varphi$.

For this purpose we consider a new version of the unfold/fold transformation rules and we show that this version preserves the perfect model semantics. Our proof method, called unfold/fold proof method, shows $M(P) = \varphi$ by: (i) introducing a new predicate symbol $f$ and constructing a conjunction $F(f, \varphi)$ of clauses such that $M(P) = \varphi$ iff $M(P \land F(f, \varphi)) = f$, and then (ii) transforming the program $P \land F(f, \varphi)$ into a new program of the form $Q \land f$, for some conjunction $Q$ of clauses.

We also present a strategy for applying our unfold/fold rules in a semi-automatic way. Our strategy may or may not terminate, depending on the initial program $P$ and formula $\varphi$. However, we identify some classes of programs and formulas for which our strategy always terminates and thus, our strategy is a decision procedure for checking whether or not $M(P) = \varphi$ holds for any given program $P$ and formula $\varphi$ in those classes.

Key words: Program Transformation, Mechanical Theorem Proving, Logic Programming
1. Introduction

One of the main motivations of this paper is to better understand the relationship between unfold/fold program transformation \[5, 28\] and theorem proving. It is usually recognized that folding steps during program transformation correspond to applications of inductive hypotheses during proofs by induction, and goal replacements correspond to lemma applications.

Some transformational techniques for proving equivalences of functional and logic programs have been already presented in \[9, 17\] and \[21, 26\], respectively. In this paper we extend these techniques by introducing a method for proving that a closed first order formula \(\varphi\) holds in the perfect model \[23\] of a locally stratified logic program \(P\). This property is denoted by 

\[ M(P) \models \varphi. \]

Perfect models are the usual intended semantics for logic programs with locally stratified negation, and for those programs all major approaches to the semantics of negation coincide. Indeed, a locally stratified program has a unique perfect model which is equal to its unique stable model, and also equal to its total well-founded model (see \[1\] for a survey on the semantics of negation in logic programming).

Our proof method for showing that \(M(P) \models \varphi\) holds, consists of two steps:

\[\text{Step 1:}\]
we use a variant of the Lloyd-Topor transformation \[19\] for transforming a statement of the form: \(f \leftarrow \varphi\) where \(f\) is a new predicate symbol, into a conjunction \(F(f, \varphi)\) of clauses, possibly with negated atoms in their bodies, such that \(P \land F(f, \varphi)\) is locally stratified and \(M(P) \models \varphi\) iff \(M(P \land F(f, \varphi)) \models f\), and

\[\text{Step 2:}\]
we show that \(M(P \land F(f, \varphi)) \models f\) holds by applying transformation rules which preserve perfect models, and deriving from \(P \land F(f, \varphi)\) a new program of the form: \(Q \land f\).

Since \(f\) is false in the perfect model of any program which has no clause with head \(f\), our method can also be used to show that \(M(P) \not\models \varphi\), that is, \(M(P) \models \neg \varphi\), by deriving from \(P \land F(f, \varphi)\) a new program \(R\), such that \(f\) does not occur in the head of any clause in \(R\).

We illustrate our transformational proof method using the following example where we prove a property of a given transition system.

**Example.** [Semaphore] Consider the following program \(P\), where \(x\) and \(y\) are variables, and \(s \ldots s0\) with \(n\) occurrences of the successor function \(s\), denotes the natural number \(n\):

1. \(down(sx) \leftarrow \neg down(x)\)
2. \(up(0,0)\)
3. \(up(sx,0) \leftarrow down(x)\)
4. \(up(sx, sy) \leftarrow up(sx, y), x>y\)

Program \(P\) describes a semaphore which as time progresses, alternates between the states \(up\) and \(down\). When the semaphore goes \(up\) for the \(n\)-th time, it stays \(up\) for a period of \(2n\) time-units, and when it goes \(down\), it stays \(down\) for one time-unit only (see the following table).

<table>
<thead>
<tr>
<th>(up)</th>
<th>0,0</th>
<th>2,0</th>
<th>2,1</th>
<th>4,0</th>
<th>4,1</th>
<th>4,2</th>
<th>4,3</th>
<th>6,0</th>
<th>...</th>
<th>6,5</th>
<th>8,0</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>(down)</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>7</td>
<td>9</td>
<td>10</td>
<td>...</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>...</td>
</tr>
<tr>
<td>(time)</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>...</td>
</tr>
</tbody>
</table>

We want to prove the following Property (A) of the semaphore:

\[\forall x, y \ (x>y, \ up(x, y) \rightarrow up(ssx, 0))\]
which states that if the semaphore is up then it will be up again in the future. We start from
the statement: \( f \leftarrow (\forall x, y (x > y, \text{up}(x, y)) \rightarrow \text{up}(s s x, 0)) \). By applying our LT transformation
(see Section 3) which is a variant of the Lloyd-Topor transformation, we get the clauses:

5. \( f \leftarrow \neg g \)
6. \( g \leftarrow x > y, \text{up}(x, y), \neg \text{up}(s s x, 0) \)

This concludes Step 1 of our proof method.

Step 2 is realized by applying the following transformation rules: definition introduction, positive and negative unfolding, folding, tautology, clause deletion, and goal replacement. These
rules are extensions of the familiar rules presented in [27, 28] and, under the conditions of Section
4 below, they preserve perfect models.

In this Semaphore Example we introduce the definitions: (i) \( h \leftarrow x > y, \text{up}(x, y), \neg \text{up}(s s s x, 0) \)
and (ii) \( k(n) \leftarrow x > z, \text{plus}(y, n, z), \text{up}(s x, y), \neg \text{up}(s s s s x, 0) \). By positive and negative unfolding,
folding, and goal replacement, we get the following program:

\[
Q: f \leftarrow \neg g, \quad g \leftarrow h, \quad h \leftarrow k(s0), \quad k(n) \leftarrow k(sn)
\]

The details of the derivation are given in 3.

Now, since \( k \) is a useless predicate being defined by the recursive clause \( k(n) \leftarrow k(sn) \) only,
without base case, we may apply the clause deletion rule (see rule R6 in Section 4) and we get
the program

\[
R: f \leftarrow \neg g, \quad g \leftarrow h, \quad h \leftarrow k(s0)
\]

By unfolding clause \( h \leftarrow k(s0) \) w.r.t. \( k(s0) \) we delete that clause, because the definition of the
predicate \( k \) in program \( R \) is empty. Then, by unfolding \( g \leftarrow h \) w.r.t. \( h \) we also delete that clause,
because the definition of \( h \) has been deleted in the previous unfolding step. Thus, the derived
program is the clause \( f \leftarrow \neg g \) only. Finally, by unfolding \( f \leftarrow \neg g \) w.r.t. \( \neg g \) we get \( f \), because
the definition of \( g \) in the derived program is empty. This completes our transformational proof
of Property (A) we wanted to show. \( \Box \)

Notice that our proof of Property (A) is based on the crucial step of deleting the clause
\( k(n) \leftarrow k(sn) \), and this step is correct because in the perfect model of program \( R \) there are no
atoms with predicate \( k \).

The tight correspondence between program transformation and theorem proving can also be
exploited to turn program transformation strategies into proof strategies. In particular, at Step
2 of our proof method, in order to direct the transformation rules, we may use the so-called UFS
strategy which is an enhancement of a strategy introduced in the case of definite logic programs
for deriving programs without existential variables [22]. When our UFS strategy terminates, it
derives from \( P \wedge F(f, \varphi) \) either (i) a new program of the form \( Q \wedge f \), in which case \( M(P) = \varphi \),
or (ii) a program where no clause has head predicate \( f \), in which case \( M(P) \neq \varphi \). We will show
that the UFS strategy indeed terminates for some classes of properties and some classes of logic
programs with locally stratified negation, and thus, it can be used for deciding whether or not
\( M(P) = \varphi \) holds when \( \varphi \) and \( P \) belong to those classes.

The plan of the paper is as follows. In Section 3 we present Step 1 of our proof method by
introducing a variant of the Lloyd-Topor transformation technique and we show that it preserves
the perfect model semantics. The original Lloyd and Topor’s correctness result w.r.t. the com-
pletion semantics [19] is often not suitable for proving properties of perfect models. In Section
4 we present the transformation rules, including the Definition rule, the Positive and Negative
Unfolding rule (that is, unfolding of positive and negative literals), the Folding rule, the Tautology rule, the Clause Deletion rule, and the Goal Replacement rule. We show that all these rules preserve the perfect model semantics, while previous sets of rules proposed by Seki [27] include the Definition, Positive Unfolding and Folding rules only.

In Section 5 we present our UFS strategy for locally stratified programs, and in Section 6 we define some classes of programs and programs for which a deterministic version of the UFS strategy terminates. We show that these classes are non trivial by proving that both the equivalence problem of tree automata [12] and the decision problem for the clausal fragment of the weak monadic second order theory with n successors [24] can be reduced to the problem of deciding whether or not $M(P) \models \varphi$ holds for suitable programs and formulas in those classes. Finally, in Section 7 we compare our transformational proof method to other theorem proving methods, such as, (i) methods based on the Clark completion [7, 19], (ii) methods based on resolution augmented by the negation as (finite or infinite) failure rule [1, 23], (iii) methods based on induction, and (iv) the proof by consistency method [15, 16, 20].

2. Preliminaries

In this section we recall some basic definitions concerning locally stratified logic programs and perfect models. For notions not defined here the reader may refer to [1, 19, 23].

The formulas and programs we consider are constructed by using a fixed first-order language $\mathcal{L}$. Programs are conjunctions of clauses of the form: $H \leftarrow L_1, \ldots, L_n$, where $H$ is an atom and for every $i = 1, \ldots, n$, $L_i$ is a literal, that is, an atom (i.e., positive literal) or a negated atom (i.e., negative literal). A goal is a conjunction of literals. The empty conjunction is true and the empty disjunction is false. Conjunction between literals, goals, and clauses is denoted by comma and also by $\land$. Conjunction is associative, commutative, and its neutral element is true. Thus, the order of literals in a goal is immaterial. For $n = 0$, the clause $H \leftarrow L_1, \ldots, L_n$ is said to be a unit clause and it is written as $H \leftarrow \text{true}$ or simply $H$. A clause (or a program) is said to be definite iff no negated atom occurs in it. We may freely rename the variables occurring in clauses. The renaming of all variables of a clause using new variables is called renaming apart [19]. The head and the body of a clause $C$ are denoted by $hd(C)$ and $bd(C)$, respectively. The predicate symbol of the atom $hd(C)$ is called the head predicate of $C$.

Given a term $t$ we denote by $vars(t)$ the set of all variables occurring in $t$. Similar notations will be used for the variables occurring in formulas. Given a clause $C$, a variable in $bd(C)$ is said to be existential iff it belongs to $vars(bd(C)) - vars(hd(C))$. Given a formula $\varphi$ we denote by $freesv(\varphi)$ the set of all free variables of $\varphi$. A clause $C$ is said to be ground iff no variable occurs in it. A literal is said to be propositional iff its predicate symbol is nullary, that is, it has arity 0. A goal (or a clause, or a program) is propositional iff all its literals are propositional. A formula is said to be function-free iff no function symbols occur in it.

The definition of a predicate $p$ in a program $P$, denoted by $Def(p, P)$, is the conjunction of all clauses of $P$ whose head predicate is $p$. We say that $p$ is defined in $P$ iff $Def(p, P)$ is not empty. We say that a predicate $p$ depends on a predicate $q$ in $P$ iff either there exists in $P$ a clause of the form: $p(\ldots) \leftarrow B$ such that $q$ occurs in the goal $B$ or there exists in $P$ a predicate $r$ such that $p$ depends on $r$ in $P$ and $r$ depends on $q$ in $P$. The extended definition of a predicate $p$ in a program $P$, denoted by $Def^*(p, P)$, is the conjunction of the definition of $p$ and the definitions of all the predicates on which $p$ depends in $P$.

The set of the useless predicates of a program $P$ is the maximal set $U$ of predicates of $P$ such that a predicate $p$ is in $U$ iff the body of each clause of $Def(p, P)$ has a positive literal
whose predicate is in \( U \). For instance, \( p \) and \( q \) are useless and \( r \) is not useless in the program 
\[(p \leftarrow q, r) \land (q \leftarrow p) \land (r \leftarrow)\].

By \( \text{ground}(P) \) we denote the conjunction of all clauses in \( \mathcal{L} \) which are ground instances of clauses of \( P \), and by \( B_\mathcal{L} \) we denote the Herbrand Base of \( \mathcal{L} \), that is, the set of all ground atoms in \( \mathcal{L} \). A stratification \( \sigma \) is a total function from \( B_\mathcal{L} \) to the set \( W \) of countable ordinals. Given a ground literal \( L \) which is the atom \( A \) or the negated atom \( \neg A \), we say that \( L \) is in stratum \( \alpha \) iff \( \sigma(A) = \alpha \).

A ground clause \( H \leftarrow L_1, \ldots, L_n \) is locally stratified w.r.t. a stratification \( \sigma \) iff for every \( i = 1, \ldots, n \), if \( L_i \) is an atom then \( \sigma(H) \geq \sigma(L_i) \), and if \( L_i \) is a negated atom then \( \sigma(H) > \sigma(L_i) \). We say that the program \( P \) is locally stratified iff there exists a stratification \( \sigma \) such that every clause in \( \text{ground}(P) \) is locally stratified w.r.t. \( \sigma \).

Let \( P_\alpha \) be the conjunction of the clauses in \( \text{ground}(P) \) whose head is in the stratum \( \alpha \). We may assume without loss of generality, that the stratum of every ground literal is greater than 0, so that \( P_0 \) may be assumed to be the empty conjunction of clauses. The perfect model \( M(P) \) of a program \( P \) which is locally stratified w.r.t. a stratification \( \sigma \), is the subset \( \bigcup_{\alpha \in W} M_\alpha \) of \( B_\mathcal{L} \), where for every ordinal \( \alpha \) in \( W \), the set \( M_\alpha \) is defined as follows:

1. (1) \( M_0 \) is the empty set,
2. (2) for every successor ordinal \( \alpha + 1 \), \( M_{\alpha + 1} \) is the least Herbrand model of the program derived from \( P_{\alpha + 1} \) as follows: (i) every literal \( L \) in stratum \( \tau \) \((\leq \alpha)\) in the body of a clause in \( P_{\alpha + 1} \) is deleted iff \( M_\tau \models L \), and (ii) every clause \( C \) in \( P_{\alpha + 1} \) is deleted iff in \( \text{bd}(C) \) there exists a literal \( L \) in stratum \( \tau \) \((\leq \alpha)\) such that \( M_\tau \models L \), and
3. (3) for every limit ordinal \( \lambda \), \( M_\lambda \) is \( \bigcup_{\alpha < \lambda} M_\alpha \).

### 3. Translating First-Order Formulas into Logic Programs

In this section we present a method that given a locally stratified program \( P \) and a closed first order formula \( \varphi \) in \( \mathcal{L} \), introduces a new predicate \( f \) and constructs a locally stratified program \( P \land F(f, \varphi) \) such that \( M(P) \models \varphi \) iff \( M(P \land F(f, \varphi)) \models f \). This method is based upon a variant of a transformation proposed by Lloyd and Topor [19].

In order to construct the locally stratified program \( F(f, \varphi) \) for a given closed first-order formula \( \varphi \), we need to consider a class of formulas, called statements [19], of the form \( A \leftarrow \beta \) where \( A \) is an atom and \( \beta \), called the body of the statement, is a (possibly open) first-order logic formula. We write \( C[\gamma] \) to denote a first-order formula where the subformula \( \gamma \) occurs as a conjunct ‘at top level’, that is, \( C[\gamma] = \rho_1 \land \ldots \land \rho_r \land \gamma \land \sigma_1 \land \ldots \land \sigma_s \) for some first-order formulas \( \rho_1, \ldots, \rho_r, \sigma_1, \ldots, \sigma_s \), and some \( r \geq 0 \) and \( s \geq 0 \). When we say that the formula \( C[\gamma] \) is transformed into the formula \( C[\delta] \), we mean that \( C[\delta] \) is obtained from \( C[\gamma] \) by replacing the top level conjunct \( \gamma \) by the new top level conjunct \( \delta \).

Given any conjunction of statements the following LT transformation, similar to the one proposed by Lloyd and Topor and reported in [19], terminates and it produces a conjunction of clauses which is a locally stratified program.

**The LT transformation from conjunctions of statements to locally stratified programs.**

Given a conjunction of statements, perform the following transformations.
(A) Eliminate from the body of every statement all occurrences of logical constants, connectives, and quantifiers other than \( \text{true}, \neg, \land, \text{and } \exists \). For every statement \( st \), rename the bound variables of \( st \) so that none of them occurs in \( \text{freevars}(st) \) and all of them are distinct.
(B) Apply as long as possible the following rules:

(B.1) \( A \leftarrow C[\neg \text{true}] \) is deleted

(B.2) \( A \leftarrow C[\neg \varphi] \) is transformed into \( A \leftarrow C[\varphi] \)

(B.3) \( A \leftarrow C[\neg (\varphi \land \psi)] \) is transformed into \( A \leftarrow C[\neg \text{newp}(y_1, \ldots, y_k)] \land \text{newp}(y_1, \ldots, y_k) \leftarrow \varphi \land \psi \)

where \( \text{newp} \) is a new predicate symbol and \( \{y_1, \ldots, y_k\} = \text{freevars}(\varphi \land \psi) \). We assume that \( \varphi \neq \text{true} \) and \( \psi \neq \text{true} \).

(B.4) \( A \leftarrow C[\exists x \varphi] \) is transformed into \( A \leftarrow C[\neg \text{newp}(y_1, \ldots, y_k)] \land \text{newp}(y_1, \ldots, y_k) \leftarrow \varphi \)

where \( \text{newp} \) is a new predicate symbol and \( \{y_1, \ldots, y_k\} = \text{freevars}(\exists x \varphi) \).

(B.5) \( A \leftarrow C[\exists x \varphi] \) is transformed into \( A \leftarrow C[\varphi] \)

Given a locally stratified program \( P \) and a closed first-order formula \( \varphi \), we denote by \( F(f, \varphi) \) the conjunction of the clauses derived by applying the LT transformation to the statement \( f \leftarrow \varphi \), where \( f \) is a new predicate symbol occurring neither in \( \varphi \) nor in \( P \). We assume that the new predicates introduced during the construction of \( F(f, \varphi) \) do not occur in \( P \).

The reader may verify that in the Semaphore Example of Section 1, clauses 5 and 6 have been derived by applying the LT transformation starting from the following statement:

\[ f \leftarrow \forall x, y \ (x > y, \ \text{up}(x, y) \rightarrow \text{up}(ssx, 0)) \]

The following result states that the LT transformation is correct w.r.t. the perfect model semantics, and thus, Step 1 of our proof method is sound. Recall that, instead, the Lloyd-Topor transformation as described in [19], is known to be correct w.r.t. the completion semantics only.

**Theorem 1. [Correctness of LT Transformation w.r.t. Perfect Models]** Let \( P \) be a locally stratified program, \( \varphi \) be a closed first-order formula, and \( f \) be a predicate symbol occurring neither in \( \varphi \) nor in \( P \). If \( F(f, \varphi) \) is obtained from \( f \leftarrow \varphi \) by the LT transformation, then (i) \( P \land F(f, \varphi) \) is a locally stratified program, and (ii) \( M(P) \models \varphi \) iff \( M(P \land F(f, \varphi)) \models f \).

Proof. (i) Since \( f \) is a new predicate symbol and rules B.1-B.5 do not introduce cycles in the dependency graph, we have that the dependency graph of \( F(f, \varphi) \) has no cycles. Since no predicate in \( P \) depends on the predicates in \( F(f, \varphi) \) and \( P \) is locally stratified, we have that \( (P \land F(f, \varphi)) \) is locally stratified.

(ii) We have that \( M(P \land F(f, \varphi)) \models \text{comp}(P \land F(f, \varphi)) \) where \( \text{comp}(P) \) denotes the Clark completion of program \( P \) [19]. By construction we have that \( \text{comp}(P \land F(f, \varphi)) \models f \leftrightarrow \varphi \) and thus, \( M(P \land F(f, \varphi)) \models f \leftrightarrow \varphi \). To conclude the proof it remains to show that \( M(P) \models \varphi \) holds iff \( M(P \land F(f, \varphi)) = \varphi \). This follows from the fact that no predicate defined in \( P \) and no predicate occurring in \( \varphi \) depends on a predicate defined in \( F(f, \varphi) \). \( \square \)

4. Transformation Rules

In this section we present the rules for transforming logic programs and we provide a sufficient condition which ensures that perfect models are preserved during program transformation.

A transformation sequence is a sequence of programs \( P_0, \ldots, P_n \), where, for \( 0 \leq k \leq n - 1 \), program \( P_{k+1} \) is derived from program \( P_k \) by the application of a transformation rule as indicated below.
For the application of the transformation rules we divide the predicate symbols of the language into two categories: basic predicates and non-basic predicates. Atoms, literals, and goals which have occurrences of basic predicates only, are called basic atoms, basic literals, and basic goals, respectively. We assume that each basic atom is in a strictly smaller stratum w.r.t. any non-basic atom.

Our partition of the set of predicates into basic or non-basic predicates is arbitrary and it is required only for proving the Correctness Theorem below. The actual partition one may want to consider in practice, depends on the given initial program $P_0$. For instance, if one is interested in proving properties of a program on the domain of integer numbers, one may assume that addition, multiplication, and ordering relations are defined by basic predicates.

For $0 \leq k \leq n$, we also consider the conjunction $Defs_k$ of definitions, constructed as follows:

1. $Defs_0$ is the conjunction of every clause $C$ in program $P_0$ of the form $p(x_1, \ldots, x_m) \leftarrow L_1, \ldots, L_n$, with $n > 0$ such that: (i) $x_1, \ldots, x_m$ are distinct variables (possibly all variables) occurring in the goal $L_1, \ldots, L_n$, (ii) at least one literal among $L_1, \ldots, L_n$ is a non-basic positive literal, (iii) no predicate symbol occurring in the goal $L_1, \ldots, L_n$ depends on $p$ in $P_0$, and (iv) $C = \text{Def}(p, P_0)$;

2. For $k > 0$, $Defs_k$ is the conjunction of the clauses in $Defs_0$ and those introduced by the definition rule (see rule R1) during the transformation sequence $P_0, \ldots, P_k$.

**R1. Definition Rule.** We get program $P_{k+1}$ by adding to program $P_k$ a clause $C$ of the form:

$\text{newp}(x_1, \ldots, x_m) \leftarrow L_1, \ldots, L_n$, with $n > 0$, such that: (i) $x_1, \ldots, x_m$ are distinct variables occurring in $L_1, \ldots, L_n$, (ii) at least one literal among $L_1, \ldots, L_n$ is a non-basic positive literal, and (iii) the predicate symbol $\text{newp}$ is a non-basic predicate which occurs neither in $P_0, \ldots, P_k$, nor in $L_1, \ldots, L_n$.

**R2. Positive Unfolding Rule.** Let $C$ be a renamed apart clause in $P_k$ of the form $H \leftarrow G_1, A, G_2$, where $A$ is an atom, and $G_1$ and $G_2$ are (possibly empty) goals. Suppose that:

1. $D_1, \ldots, D_m$, with $m \geq 0$, are all clauses of program $P_k$, such that $A$ is unifiable with $\text{hd}(D_1), \ldots, \text{hd}(D_m)$, with most general unifiers $\theta_1, \ldots, \theta_m$, respectively, and

2. $C_i$ is the clause $(H \leftarrow G_1, \text{bd}(D_i), G_2)\theta_i$, for $i = 1, \ldots, m$.

By unfolding $C$ w.r.t. $A$ we derive from program $P_k$ the new program $P_{k+1}$ by replacing $C$ by $C_1, \ldots, C_m$.

In particular, if $m = 0$, that is, if we unfold a clause $C$ in program $P_k$ w.r.t. an atom which is not unifiable with the head of any clause in $P_k$, then we derive from program $P_k$ the new program $P_{k+1}$ by deleting clause $C$.

**R3. Negative Unfolding Rule.** Let $C$ be a renamed apart clause in $P_k$ of the form $H \leftarrow G_1, \neg A, G_2$, where $A$ is an atom, and $G_1$ and $G_2$ are (possibly empty) goals. Let $D_1, \ldots, D_m$, with $m \geq 0$, be all clauses of program $P_k$, such that $A$ is unifiable with $\text{hd}(D_1), \ldots, \text{hd}(D_m)$, with most general unifiers $\theta_1, \ldots, \theta_m$, respectively. Assume that:

1. $A = \text{hd}(D_1)\theta_1 = \cdots = \text{hd}(D_m)\theta_m$, that is, for $i = 1, \ldots, m$, $A$ is an instance of $\text{hd}(D_i)$,

2. for $i = 1, \ldots, m$, $D_i$ has no existential variables, and

3. from $G_1, \neg (\text{bd}(D_1)\theta_1 \lor \cdots \lor \text{bd}(D_m)\theta_m)$, $G_2$ we get an equivalent disjunction $Q_1 \lor \cdots \lor Q_r$ of goals, with $r \geq 0$, by first pushing $\neg$ inside and then pushing $\lor$ outside.
By unfolding $C$ w.r.t. $\neg A$ we derive from program $P_k$ the new program $P_{k+1}$ by replacing $C$ by $C_1, \ldots, C_r$, where $C_i$ is the clause $H \leftarrow Q_i$, for $i = 1, \ldots, r$.

In particular: (i) if $m = 0$, that is, if we unfold a clause $C$ w.r.t. a negative literal $\neg A$ such that $A$ is not unified with the head of any clause in $P_k$, then we get the new program $P_{k+1}$ by deleting $\neg A$ from the body of clause $C$, and (ii) if for some $i \in 1, \ldots, m$, $bd(D_i) = true$, that is, if we unfold a clause $C$ w.r.t. a negative literal $\neg A$ such that $A$ is an instance of the head of a unit clause in $P_k$, then we derive from program $P_k$ the new program $P_{k+1}$ by deleting clause $C$.

R4. Folding Rule. Let $D$ be a renamed apart definition in $Defs_k$ and $C$ be a clause in $P_k$ of the form $H \leftarrow G_1, B, G_2$, where $B$, $G_1$, and $G_2$ are (possibly empty) goals. Suppose that for some substitution $\theta$: (i) $B = bd(D)\theta$, and (ii) for every variable $x$ in the set $vars(D) - vars(bd(D))$, we have that $x\theta$ is a variable which occurs neither in $\{H, G_1, G_2\}$ nor in the term $y\theta$, for any variable $y$ occurring in $bd(D)$ and different from $x$.

By folding clause $C$ w.r.t. $B$ using clause $D$ we derive clause $E : H \leftarrow G_1, bd(D)\theta, G_2$ and we get from program $P_k$ the new program $P_{k+1}$ by replacing $C$ by $E$.

R5. Tautology Rule. We get the new program $P_{k+1}$ by replacing in $P_k$ a conjunction of clauses by the corresponding equivalent conjunction of clauses, according to the following equivalences, where $G$ and $R$ denote (possibly empty) goals, and $H$ and $A$ denote atoms:

(1) $(H \leftarrow A, \neg A, G) \leftrightarrow true$
(2) $(H \leftarrow H, G) \leftrightarrow true$
(3) $(H \leftarrow G, H \leftarrow G, R) \leftrightarrow (H \leftarrow G)$
(4) $(H \leftarrow A, G, R, H \leftarrow \neg A, G) \leftrightarrow (H \leftarrow G, R, H \leftarrow \neg A, G)$

R6. Clause Deletion Rule. We get the new program $P_{k+1}$ by removing from $P_k$ the definitions of the useless predicates of $P_k$.

R7. Goal Replacement Rule. Let $C$ be a renamed apart clause in $P_k$ of the form $H \leftarrow G_1, Q, G_2$, where $Q$, $G_1$, and $G_2$ are (possibly empty) goals. Suppose that, for some goal $R$, $M(P_k) \models \forall x_1 \ldots x_u (\exists y_1 \ldots y_v Q \leftarrow \exists z_1 \ldots z_w R)$

where: (i) $\{y_1, \ldots, y_v\} = vars(Q) - vars(H, G_1, G_2)$, (ii) $\{z_1, \ldots, z_w\} = vars(R) - vars(H, G_1, G_2)$, and (iii) $\{x_1, \ldots, x_u\} = vars(Q, R) - \{y_1, \ldots, y_v, z_1, \ldots, z_w\}$. Suppose also that $Q$ and $R$ are basic goals and $H$ is a non-basic atom. Then from program $P_k$ we derive the new program $P_{k+1}$ by replacing $C$ by the clause $H \leftarrow G_1, R, G_2$.

Theorem 2. [Correctness of the Transformation Rules] Let $P_0, \ldots, P_n$, be a transformation sequence such that the following holds: if for some $k$ with $1 \leq k < n$, we have applied rule R4 for folding clause $C$ in $P_k$ using clause $D$ in $P_0 \land Defs_k$, then there exists $i$, with $0 \leq i < k$ such that $D$ occurs in $P_i$ and $P_{i+1}$ is derived from $P_i$ by positive unfolding of $D$ w.r.t. a non-basic atom. Then we have that $M(P_0 \land Defs_n) = M(P_n)$.

Proof. It is an extension of the proofs in [27, 28] and it is based on the fact that if we establish a recursion via a folding step, thereby generating from program $P_i$ the new program $P_{i+1}$, then the size (according to a suitable measure) of the derivations of ground atoms w.r.t. $P_i$, is not smaller than the size of the corresponding derivations w.r.t. $P_{i+1}$. This property is indeed ensured by the positive unfolding of the clause used for the folding step leading from $P_i$ to $P_{i+1}$.

Notice that the statement obtained from Theorem 2 by replacing 'positive unfolding' by 'negative unfolding' is not a theorem, as shown by the following example. Let $P_0$ be the program $(p \leftarrow \neg q(x)) \land (q(x) \leftarrow q(x)) \land (q(x) \leftarrow r)$. By negative unfolding w.r.t. $\neg q(x)$ we get program
10.

\[ P_1: (p \leftarrow \neg q(x), \neg r) \land (q(x) \leftarrow q(x)) \land (q(x) \leftarrow r). \] Then by folding we get program \( P_2: (p \leftarrow p, \neg r) \land (q(x) \leftarrow q(x)) \land (q(x) \leftarrow r). \) We have that \( M(P_0) \models p, \) while \( M(P_2) \models \neg p. \)

5. A Strategy for Unfold/Fold Proofs

In order to verify whether or not \( M(P) \models \varphi \) holds, Step 2 of our unfold/fold proof method requires the construction of a transformation sequence from program \( P \land F(f, \varphi) \) to a new program, say \( T \), such that either (i) \( \text{Def}(f, T) \) is \( f \), and in this case we infer that \( M(P) \models \varphi \), or (ii) \( T \) is a program where \( \text{Def}(f, T) \) is the empty conjunction, and in this case we infer that \( M(P) \models \varphi \). To construct this transformation sequence we need a strategy for guiding the application of the transformation rules, and indeed, in this section we present such a strategy, called UFS (short for Unfold/Fold proof Strategy). The UFS strategy is an extension of the strategy introduced in the case of definite logic programs for eliminating existential variables [22]. The basic idea is that by eliminating existential variables, from \( P \land F(f, \varphi) \) we may derive a program, say \( S \), such that \( \text{Def}^*(f, S) \) is a propositional program. Then, we can transform \( S \) by using the clause deletion, unfolding, tautology, and goal replacement rules, into a program \( T \) satisfying either (i) or (ii) above.

Obviously, since in general \( M(P) \models \varphi \) is an undecidable property, our UFS strategy may fail to produce a propositional program \( S \) starting from an arbitrary program \( P \land F(f, \varphi) \). However, in the next section we will show that in some cases our strategy is guaranteed to terminate and it produces a propositional program \( S \), and hence, the desired program \( T \). Thus, our UFS strategy is a decision procedure for the problem of verifying whether or not \( M(P) \models \varphi \) holds, when \( P \) and \( \varphi \) belong to suitable classes of programs and formulas, respectively. Moreover, even when our strategy derives a program which is not propositional, it may still be the case that we are able to infer whether or not \( M(P) \models \varphi \) holds, and indeed, this happened in the Semaphore Example given in the Introduction.

For specifying the UFS strategy, we need the following definition (see also [19]).

**Definition 1.** A level mapping of a program \( P \) is a mapping from the set of predicate symbols occurring in \( P \) to the set of natural numbers. The level of predicate \( p \) is the value of \( p \) under this mapping.

By the definition of the LT transformation there exists a level mapping of \( P \land F(f, \varphi) \) such that: (i) the level of every predicate defined in \( P \) is 0, (ii) the level of every predicate defined in \( F(f, \varphi) \) is greater than 0, (iii) for each clause \( p(\ldots) \leftarrow B \) in \( F(f, \varphi) \) the level of every predicate symbol in \( B \) is strictly smaller than the level of \( p \), and (iv) the predicate \( f \) has the highest level, say \( K \). For instance, in the Semaphore Example we may choose the level mapping \( m \) defined by the following equations: \( m(\text{down}) = m(\text{up}) = m(\succ) = 0, \ m(g) = 1, \) and \( m(f) = 2 \) (thus, \( K = 2 \)).

The UFS strategy, as we will present it now, requires three substrategies: (1) UNFOLD, (2) TAUTOLEG & REPLACE, and (3) DEFINE & FOLD. Below they will be made algorithmic for the particular classes of programs and formulas for which the UFS strategy terminates.

**The Unfold/Fold Proof Strategy UFS.**

**Input:** (i) \( P \land F(f, \varphi) \), where \( P \) is a locally stratified program and \( \varphi \) is a closed first order formula, and (ii) a set \( \text{Laws} \) of equivalences of the form: \( M(P) \models \forall x_1 \ldots x_u (\exists y_1 \ldots y_s Q \leftrightarrow \exists z_1 \ldots z_w R) \), where \( Q \) and \( R \) are basic goals.
Output: A program $T$ such that: (i) $M(T) \models f$ iff $M(P \land F(f, \varphi)) \models f$, and (ii) $Def(f, T)$ is either $f$ or the empty conjunction.

Let $P \land F(f, \varphi)$ be $P \land P_1 \land \ldots \land P_K$, where for $i = 1, \ldots, K$, program $P_i$ is the conjunction of the clauses whose head predicate has level $i$.

$T := P$;
for $i = 1, \ldots, K$ do
Let $Pos$ be the conjunction of the clauses of $P_i$ whose body has at least one non-basic positive literal, and let $Neg$ be the conjunction of the clauses of $P_i$ which are not in $Pos$.
Let $Defs$ be $Pos$ and $Out$ be the empty conjunction.
while $Pos$ is not the empty conjunction do
(1) UNFOLD($T$, $Pos$, $U$): From program $T \land Pos$ we derive $T \land U$ by a finite sequence of applications of the positive or negative unfolding rules to the clauses in $Pos$. We require that: [Progression] the positive unfolding rule is applied at least once to each clause in $Pos$.
(2) TAUTOLOGY & REPLACE($T$, $Laws$, $U$, $R$): From program $T \land U$ we derive $T \land R$ by a finite sequence of applications of the tautology and goal replacement rules to the clauses in $U$, using the equivalences in the set $Laws$.
(3) DEFINE & FOLD($T$, $R$, $Defs$, $OutClauses$, $NewDefs$): From program $T \land R$ we derive $T \land OutClauses \land NewDefs$ by: (i) a finite sequence of applications of the definition rule by which we introduce the (possibly empty) conjunction $NewDefs$ of clauses, followed by (ii) a finite sequence of applications of the folding rule to the clauses in $R$, using clauses occurring in $Defs \land NewDefs$. We assume that the definition and folding steps are such that the following conditions are satisfied:
(3.1) [Positive definitions] The body of each clause in $NewDefs$ has at least one non-basic positive literal.
(3.2) [No existential variables] Each clause in $OutClauses$ which has been derived by folding has no existential variables.
(3.3) [Full folding] Each predicate in the body of each clause in $OutClauses$ which has been derived by folding, occurs in $Defs \land NewDefs$.
$Out := Out \land OutClauses$; $Pos := NewDefs$; $Defs := Defs \land NewDefs$
end while;
Delete from $Out$ the definitions of useless predicates, thereby deriving $Out'$; $T := T \land Out' \land Neg$;
Initialize $D$ to the conjunction of all definitions of nullary predicates in $T$;
Initialize $Q$ to the conjunction of all definitions of non-nullary predicates in $T$;
while $D$ is a not a conjunction of unit clauses do
UNFOLD($Q$, $D$, $U$): From program $Q \land D$ we derive $Q \land U$ by a (possibly empty) sequence of applications of the positive or negative unfolding rules to the clauses in $D$. (Here the Progression requirement need not be satisfied.)
TAUTOLOGY & REPLACE($Q$, $Laws$, $U$, $R$); $D := R$
end while;
Unfold the clauses of $Q$ w.r.t. their propositional literals, thereby deriving $Q'$; $T := Q' \land D$
end for

Our UFS strategy proceeds by iterating over levels (from level 1 to level $K$) a sequence of transformations on the program $T$, which is initialized to $P$. For $i = 1, \ldots, K$, program $P_i$
which is the conjunction of the clauses defining the predicates with level \( i \), is processed by the
UFS strategy, and the \textbf{while} loop (\( \dagger \)) generates from program \( T \land P^i \) (which is \( T \land \neg \varphi \land \text{Pos} \))
the new program \( T \). The objective of the \textbf{while} loop (\( \dagger \)) is: (i) to ensure that positive unfolding
steps w.r.t. non-basic atoms are performed before folding, and (ii) to avoid existential variables
via definition and folding steps. Then useless predicates are deleted. Finally, the \textbf{while} loop
(\( \ddagger \)) performs unfolding, tautology, and goal replacement steps on the definitions of nullary
predicates so to reduce each of them, if possible, either to the empty definition (in which case the
(corresponding predicate is \textit{false}) or to a unit clause (in which case the corresponding predicate is \textit{true}).
The truth values of the propositional predicates are then propagated by the unfolding
steps (\( \ddagger \)). When the last program level \( K \) has been processed, we get for the predicate \( f \) either
the empty definition or the clause \( f \). Thus, we may establish whether or not \( M(P \land F(f, \varphi)) \models f \)
holds.

The reader may check that our introductory Semaphore Example has indeed been worked out
using the UFS proof strategy.

The soundness of our proof strategy follows from the fact that the transformation rules are
used in such a way that the hypothesis of the Correctness Theorem of Section 4 holds.

Obviously, the UFS strategy may not terminate, because in general \( M(P \land F(f, \varphi)) \models f \) is
undecidable, and indeed: (i) during the execution of the \textbf{while} loop (\( \dagger \)) we may introduce by
applying the definition rule, an infinite number of new clauses, and thus, \textit{Pos} never becomes the
empty conjunction, and (ii) the \textbf{while} loop (\( \ddagger \)) may not terminate for nullary predicates which
depend on non-nullary predicates.

6. Decision Procedures Using the Unfold/Fold Proof Strategy

In this section we present two classes of formulas, called \textit{tree-typed formulas} and \textit{tree-typed
clausal formulas}, and two classes of programs, called \textit{MR programs} and \textit{DL programs}, for which
a deterministic version of the UFS strategy, called \textit{dUFS} strategy, terminates. Thus, the \textit{dUFS}
strategy is a decision procedure for establishing whether or not \( M(P) \models \varphi \) holds, when \( \varphi \) and
\( P \) belong to the given classes.

In this section we assume that all predicates are non-basic. The tree-typed formulas are
function-free formulas whose variables range over sets of trees denoted by \textit{tree programs} which
we now define.

\textbf{Definition 2.} [\textbf{Tree Programs}] A \textit{tree program} is a conjunction of tree clauses. A \textit{tree clause}
is a clause of the form: \( r_0(t(x_1, \ldots , x_n)) \leftarrow r_1(x_1), \ldots , r_n(x_n) \), with \( n \geq 0 \), where \( t \) is a function
symbol and \( x_1, \ldots , x_n \) are distinct variables. A \textit{tree predicate} is a predicate defined by a tree
program. A \textit{tree atom} is an atom with a tree predicate.

\textbf{Definition 3.} [\textbf{Tree-typed Formulas}] A \textit{tree-typed formula} over a program \( P \) is a first-order
formula \( \varphi \), defined as follows:

\[ \varphi ::= p(x_1, \ldots , x_n) \mid \neg \varphi \mid \varphi_1 \land \varphi_2 \mid \varphi_1 \lor \varphi_2 \mid \forall x \,(r(x) \rightarrow \varphi) \mid \exists x \,(r(x) \land \varphi) \]

where: (i) \( x_1, \ldots , x_n \), with \( n \geq 0 \), are distinct variables, (ii) all predicates occurring in \( \varphi \) are
defined in \( P \), and (iii) \textit{Def} \( ^{t}(r, P) \) is a tree program.

\textbf{Example.} [\textbf{Even-Odd Paths}] Let us consider the following \textit{Even-Odd} program:
1. \( \text{bin(leaf)} \)

2. \( \text{bin}(t(x,y)) \leftarrow \text{bin}(x), \text{bin}(y) \)

3. \( \text{even(leaf)} \)

4. \( \text{even}(t(x,y)) \leftarrow \neg \text{even}(x) \)

5. \( \text{even}(t(x,y)) \leftarrow \neg \text{even}(y) \)

6. \( \text{odd}(t(x,y)) \leftarrow \neg \text{odd}(x), \neg \text{odd}(y) \)

Clauses 1 and 2 are tree clauses. The formula \( \varphi: \forall x (\text{bin}(x) \rightarrow \text{even}(x) \lor \text{odd}(x)) \) is a tree-typed formula over Even-Odd. Informally, \( \varphi \) means that in every binary tree there exists a path of even length or all paths have odd length. Thus, we expect that \( M(P) \models \varphi \) holds, as we will formally prove below.

The Tree-typed LT Transformation.

In order to construct the locally stratified program \( F(f, \varphi) \) for a given tree-typed formula \( \varphi \), we apply the so-called tree-typed LT transformation, which is a variant of the LT transformation (see Section 3) obtained by replacing rules (B.3) and (B.4) by the following ones:

(B.3)* \( A \leftarrow C[\neg(\varphi \land \psi)] \) is transformed into \( A \leftarrow C[\neg \newp(y_1, \ldots, y_k)] \land \newp(y_1, \ldots, y_k) \leftarrow \rho \land \varphi \land \psi \)

where \( \newp \) is a new predicate symbol, \( \{y_1, \ldots, y_k\} = \text{frees}(\varphi \land \psi) \), and \( \rho \) is the conjunction of all tree atoms \( r(y) \) occurring as conjuncts at top level in \( C[\neg(\varphi \land \psi)] \) and such that \( y \in \{y_1, \ldots, y_k\} \).

(B.4)* \( A \leftarrow C[\exists x (r(x) \land \varphi)] \) is transformed into \( A \leftarrow C[\neg \newp(y_1, \ldots, y_k)] \land \newp(y_1, \ldots, y_k) \leftarrow \rho \land r(x) \land \varphi \)

where \( \newp \) is a new predicate symbol, \( \{y_1, \ldots, y_k\} = \text{frees}(\exists x (r(x) \land \varphi)) \), and \( \rho \) is the conjunction of all tree atoms \( r(y) \) occurring as conjuncts at top level in \( C[\exists x (r(x) \land \varphi)] \) and such that \( y \in \{y_1, \ldots, y_k\} \).

Example. [Even-Odd Paths. Continued] For the tree-typed formula \( \varphi: \forall x (\text{bin}(x) \rightarrow \text{even}(x) \lor \text{odd}(x)) \), the corresponding \( F(f, \varphi) \) obtained by tree-typed LT transformation is the conjunction of the following two clauses:

7. \( f \leftarrow \neg g \)

8. \( g \leftarrow \text{bin}(x), \neg \text{even}(x), \neg \text{odd}(x) \)

The tree-typed LT transformation is correct w.r.t the perfect model semantics as stated by the following theorem.

Theorem 3. [Correctness of the tree-typed LT Transformation w.r.t. Perfect Models]

For every locally stratified logic program \( P \) and closed tree-typed formula \( \varphi \) over \( P \), if \( F(f, \varphi) \) is obtained from \( \varphi \) by the tree-typed LT transformation, then

(i) \( P \land F(f, \varphi) \) is a locally stratified program and \( M(P) \models \varphi \) iff \( M(P \land F(f, \varphi)) \models f \), and

(ii) for each clause \( C \) in \( F(f, \varphi) \) we have that: \( \text{(ii.1)} \) \( C \) is of the form \( \newp(x_1, \ldots, x_m) \leftarrow r_1(x_1), \ldots, r_n(x_n), G \), where \( r_1(x_1), \ldots, r_n(x_n) \) are tree atoms, \( \text{(ii.2)} \) \( \text{vars}(C) = \{x_1, \ldots, x_n\} \),

\( \text{(ii.3)} \) \( G \) is a function-free goal, and \( \text{(ii.4)} \) \( C = \text{Def}(\newp, P \land F(f, \varphi)) \).

Proof. See Appendix 2.

Definition 4. [Monadic Regular Programs] A monadic regular program is a conjunction of monadic regular clauses. A monadic regular clause (or an MR clause, for short) is a locally stratified clause of the form:

\[ p_0(t(x_1, \ldots, x_n)) \leftarrow p_1(y_1), \ldots, p_k(y_k), \neg p_{k+1}(y_{k+1}), \ldots, \neg p_m(y_m) \]

with \( n, m \geq 0 \), where \( t \) is a function symbol, \( x_1, \ldots, x_n \) are distinct variables, and \( y_1, \ldots, y_m \) are (not necessarily distinct) variables occurring in \( \{x_1, \ldots, x_n\} \).
Tree programs are MR programs.

We now describe the deterministic version of the UFS strategy, which we call dUFS.

The Unfold/Fold Proof Strategy dUFS.

The dUFS strategy is obtained from the UFS strategy defined in Section 5, by replacing the
substrategies: (1) UNFOLD, (2) TAUTOLOGY & REPLACE, and (3) DEFINE & FOLD by the
following ones, respectively:

(1d) BREADTH-FIRST UNFOLD\((T, \mathit{Pos}, U)\): From program \(T \land \mathit{Pos}\) we derive \(T \land U\) by: (i) one application of positive unfolding w.r.t. each positive literal occurring in the body of a
clause in \(\mathit{Pos}\), followed by (ii) one application of negative unfolding w.r.t. each negative literal
occurring in the body of a clause in \(\mathit{Pos}\).

(2d) TAUTOLOGY\((T, U, R)\): From program \(T \land U\) we derive \(T \land R\) by a sequence of applications
of the tautology rule, constructed by rewriting as long as possible an instance of the left hand
side of an equivalence of Rule R5 by the corresponding instance of the right hand side.

(3d) BLOCK-DEFINE & FOLD\((T, R, \mathit{Defs}, \mathit{OutClauses}, \mathit{NewDefs})\): From program \(T \land R\) we
derive program \(T \land \mathit{OutClauses} \land \mathit{NewDefs}\) as follows:
for each non-unit clause \(C\) in \(R\)
(i) we partition \(lhd(C)\) into subconjunctions, called blocks, such that \(lhd(C) = B_1 \land \ldots \land B_m\),
and two literals occur in the same subconjunction \(B_i\), for some \(i, 1 \leq i \leq m\), iff they share a
variable,
(ii) for \(i = 1, \ldots , m\), we apply the definition rule and we add to \(\mathit{NewDefs}\) a clause of the form
\(\mathit{Newp} \leftarrow B_i\), where \(\mathit{vars}(\mathit{Newp}) = \mathit{vars}(B_i) \cap \mathit{vars}(lhd(C))\), unless a variant clause already
occurs in \(\mathit{Defs}\), modulo the head predicate symbol, and
(iii) we fold \(C\) w.r.t. \(B_1\), and we fold the resulting clause w.r.t. \(B_2\), and so on, until we fold
w.r.t. \(B_m\). □

We have that:
- the BREADTH-FIRST UNFOLD substrategy fulfills the Progression requirement,
- in the TAUTOLOGY substrategy we have omitted the Laws argument because goal replacements
are not performed, and
- the BLOCK-DEFINE & FOLD substrategy fulfills the conditions (3.1), (3.2), and (3.3) of the
DEFINE & FOLD substrategy of Section 5. Indeed, Condition (3.1) is fulfilled, because each
variable occurring in the body of a clause generated by the strategy dUFS also occurs in a tree
atom, and therefore, each block has at least one non-basic positive literal. Moreover, by Points
(ii) and (iii) in (3d) above, also Conditions (3.2) and (3.3) are fulfilled.

Example. [Even-Odd Paths. Continued] The proof that \(\forall x (\mathit{bin}(x) \rightarrow \mathit{even}(x) \lor \mathit{odd}(x))\) holds
in the perfect model of the Even-Odd program is as follows. We start from program \(P\) made out
of clauses from 1 to 6, program \(P^1\) made out of clause 8, and program \(P^2\) made out of clause 7.
We start from level 1. Initially \(\mathit{Pos} = \text{clause 8}\). By breadth-first unfolding, from clause 8 we get
the following clauses:
9. \(g \leftarrow \mathit{bin}(x), \mathit{bin}(y), \mathit{even}(x), \mathit{even}(y), \mathit{odd}(x)\)
10. \(g \leftarrow \mathit{bin}(x), \mathit{bin}(y), \mathit{even}(x), \mathit{even}(y), \mathit{odd}(y)\)

By the tautology rule R5.3 we delete clause 10 (it is subsumed by clause 9). Then we apply the
BLOCK-DEFINE & FOLD substrategy by introducing the following two definitions:
11. \(h_{co} \leftarrow \mathit{bin}(x), \mathit{even}(x), \mathit{odd}(x)\)
12. \(h_c \leftarrow \mathit{bin}(y), \mathit{even}(y)\)
and by folding clauses 9 we get:

\[ 9.f \quad g \leftarrow h_{eo}, h_e \]

Now \( \text{Pos} = \text{clause 11} \land \text{clause 12} \) and we have to perform once more the body of the \textsc{while} loop (\( \dagger \)) of the dUFS strategy. By unfolding clauses 11 and 12 we get:

\[ 11.1 \quad h_{eo} \leftarrow \text{bin}(x), \text{bin}(y), \neg \text{even}(x), \neg \text{odd}(x), \neg \text{odd}(y) \]
\[ 11.2 \quad h_{eo} \leftarrow \text{bin}(x), \text{bin}(y), \neg \text{even}(y), \neg \text{odd}(x), \neg \text{odd}(y) \]
\[ 12.1 \quad h_e \]
\[ 12.2 \quad h_e \leftarrow \text{bin}(x), \text{bin}(y), \neg \text{even}(x) \]
\[ 12.3 \quad h_e \leftarrow \text{bin}(x), \text{bin}(y), \neg \text{even}(y) \]

By the tautology rule R5.3 we delete clause 11.2 (it is subsumed by clause 11.1) and we also delete clauses 12.2 and 12.3 (they are subsumed by clause 12.1). Then we apply the \textsc{block-define} \& \textsc{fold} substrategy by introducing the following definition:

\[ 13. \quad h_o \leftarrow \text{bin}(x), \neg \text{odd}(x) \]
and by folding clause 11.1, we get:

\[ 11.1f \quad h_{eo} \leftarrow g, h_o \]

Now \( \text{Pos} = \text{clause 13} \). By unfolding clause 13 we get:

\[ 13.1 \quad h_o \]
\[ 13.2 \quad h_o \leftarrow \text{bin}(x), \text{bin}(y), \neg \text{odd}(x), \neg \text{odd}(y) \]

By the tautology rule R5.3 we delete clause 13.2 (it is subsumed by clause 13.1). No application of the \textsc{block-define} \& \textsc{fold} substrategy is required. Now \( \text{Pos} \) is empty, and \( \text{Out} \) is the conjunction of the following clauses:

\[ 9.f \quad g \leftarrow h_{eo}, h_e \]
\[ 11.1f \quad h_{eo} \leftarrow g, h_o \]
\[ 12.1 \quad h_e \]
\[ 13.1 \quad h_o \]

We then delete clauses 9.f and 11.1f because they are the definitions of the predicates \( g \) and \( h_{eo} \) which are useless in \( \text{Out} \). The \textsc{while} loop (\( \dagger \)) does not change \( D \). Since \( Q = Q' \), we have that \( T \) is made out of the clauses of \( P \) together with the clauses 12.1 and 13.1.

We can now start processing level 2, that is, program \( P^2 \). Since \( \text{Pos} \) is the empty conjunction, the body of the \textsc{while} loop (\( \dagger \)) is never executed. \( \text{Neg} \) is clause 7, and before the execution of the \textsc{while} loop (\( \dagger \)), program \( T \) is made out of the clauses in \( P \) together with the following clauses:

\[ 7. \quad f \leftarrow \neg g \]
\[ 12.1 \quad h_e \]
\[ 13.1 \quad h_o \]

After the \textsc{while} loop (\( \dagger \)) and the statement (\( \dagger \dagger \)) we get the new program \( T \) made out of the clauses in \( P \) and the clause:

\[ 7.1 \quad f \]

with clauses 12.1 and 13.1. Now, since \( M(T) = f \), we have that \( M(P) = \varphi \). \( \Box \)

Obviously the \textsc{breadth-first unfold}, \textsc{tautology}, and \textsc{block-define} \& \textsc{fold} substrategies terminate. Also the dUFS proof strategy terminates for tree-typed formulas and MR programs, as stated by the following theorem.

\textbf{Theorem 4. [Termination of the dUFS Proof Strategy for MR Programs]} Let \( P \) be an MR program and \( \varphi \) be a tree-typed formula over \( P \). Then the dUFS proof strategy with input program \( P \land F(f, \varphi) \) terminates with output program \( T \). Moreover, \( \text{Def}(f, T) \) is \( f \) iff \( M(P) = \varphi \) and \( \text{Def}(f, T) \) is the empty conjunction iff \( M(P) \not= \varphi \).
Proof. See Appendix 3.

Although very restricted, the classes of MR programs and tree-typed formulas allow us to express some interesting properties, such as the equivalence of finite tree automata (see, for instance, [12]), as shown in the following example.

Example. [Equivalence of Tree Automata] The tree language recognized by a (nondeterministic top-down) finite tree automaton $T = (Q, \Sigma, \delta, q_0, F)$ corresponds to a subset of the perfect model of a tree program $P_T$ (which is also its least Herbrand model) defined as follows:

- for each state $q \in Q$ we define a unary predicate $p_q$,
- for each symbol $a \in \Sigma$ of arity $k$ we define a $k$-ary function symbol $f_a$,
- for each tuple $(q, a, q_1, \ldots, q_k)$ in the transition relation $\delta$, where $a$ is a $k$-ary symbol with $k \geq 1$, $P_T$ has the clause:
  
  \[ p_{a}(f_a(x_1, \ldots, x_k)) \leftarrow p_{q}(x_1), \ldots, p_{q_k}(x_k) \]

- for each state $q \in F \subseteq Q$ and 0-ary symbol $b$ such that $q$ is a final state for $b$, $P_T$ has the unit clause:
  
  \[ p_{q}(b) \]

Then the tree language recognized by the tree automaton $T$ with initial state $q_0$, corresponds to the subset \{ $t \mid M(P_T) \models p_{q_0}(t)$ \}. Given another tree automaton $T_U$ with initial state $r_0$ and represented by program $P_U$, we have that $T$ and $U$ recognize the same language iff

\[ M(P_T \land P_U) \models \forall x (p_{q_0}(x) \rightarrow p_{r_0}(x)) \land \forall x (p_{r_0}(x) \rightarrow p_{q_0}(x)) \]

Thus the equivalence of $T$ and $U$ can be reduced to the verification of a tree-typed formula over the tree program $P_T \land P_U$ (recall that each tree program is also an MR program). \qed

We now introduce a second class of formulas, called tree-typed clausal formulas, and a second class of programs, called deterministic linear programs (or DL programs, for short) for which the unfold/fold proof strategy dUFS terminates. Thus, given a tree-typed clausal formula $\varphi$, and a DL program $P$, we can decide whether or not $M(P) \models \varphi$ holds by using the proof strategy dUFS.

Definition 5. [Tree-typed Clausal Formulas] Let $P$ be any program and $R$ be a tree program. A tree-typed clausal formula over a program $P \land R$ is a closed first-order formula $\varphi$ generated by the following grammar:

\[ \varphi := \forall x (r(x) \rightarrow \varphi) \mid \delta \]

\[ \delta := p(x_1, \ldots, x_n) \mid \neg p(x_1, \ldots, x_n) \mid \delta \lor \delta \]

where: (i) $x_1, \ldots, x_n$, with $n \geq 0$, are distinct variables, (ii) all predicates occurring in $\varphi$ are defined in $P \land R$, (iii) for each predicate $p$, $\text{Def}^+(p, P \land R)$ is a subconjunction of $P$, (iv) for each predicate $r$, $\text{Def}^+(r, P \land R)$ is a subconjunction of $R$, and (v) no predicate is defined in both $P$ and $R$.

Definition 6. [Deterministic Linear Programs] A linear clause is a locally stratified clause of one of these three forms:

\[ p(t_1(x_1, \ldots, x_m), \ldots, t_n(x_{u_1}, \ldots, x_v)) \]

\[ p(t_1(x_1, \ldots, x_m), \ldots, t_n(x_{u_1}, \ldots, x_v)) \leftarrow q(y_1, \ldots, y_k) \]

\[ p(t_1(x_1, \ldots, x_m), \ldots, t_n(x_{u_1}, \ldots, x_v)) \leftarrow \neg q(y_1, \ldots, y_k) \]

with $n \geq 0$, where: (i) $t_1, \ldots, t_n$ are function symbols, (ii) $x_1, \ldots, x_m, \ldots, x_{u_1}, \ldots, x_v$ are distinct variables, and (iii) $y_1, \ldots, y_k$ are distinct variables occurring in $\{ x_1, \ldots, x_m, \ldots, x_{u_1}, \ldots, x_v \}$.

A deterministic linear program (or DL program, for short) is a conjunction of linear clauses such that no two clause heads are unifiable.
Notice that there are MR programs which are not DL programs and vice versa. We have the following result whose proof is given in Appendix 4.

**Theorem 5.** [Termination of the dUFS Proof Strategy for DL Programs] Let $P$ be a DL program, $R$ be a tree program, and $\varphi$ be a tree-typed clausal formula over $P \land R$. Then the dUFS strategy with input program $P \land F(f, \varphi)$ terminates with output program $T$. Moreover, $\text{Def}(f, T)$ is $f$ if $M(P) \models \varphi$ and $\text{Def}(f, T)$ is the empty conjunction if $M(P) \not\models \varphi$.

**Example.** [Clausal wMSnS] The clausal fragment of the weak monadic second order theory of two successor functions (cwMS2S) [24] is the set of closed formulas $\varphi$ such that $\mathcal{W} \models \varphi$, where:

(i) $\varphi$ is a closed formula generated by the following grammar:

$\varphi ::= \forall w \ (\text{word}(w) \rightarrow \varphi) \mid \forall x \ (\text{set}(x) \rightarrow \varphi) \mid \delta$

$\delta ::= \text{member}(w, x) \mid \lnot \text{member}(w, x) \mid \delta \lor \delta$

(ii) $\mathcal{W}$ is the structure $\mathcal{P}_{\text{fin}}(\{0,1\}^*)$ of finite sets of words over $\{0,1\}$, where: (ii.1) the successor functions $s_0$ and $s_1$ are interpreted as $s_0(w) = w0$ and $s_1(w) = w1$, respectively, (ii.2) the predicates $\text{word}$ and $\text{set}$ hold of all words and finite sets of words, respectively, and (ii.3) $\text{member}(w, x)$ is interpreted as membership of word $w$ to set $x$.

In order to write a logic program which computes the $\text{word}$, $\text{set}$, and $\text{member}$ predicates, we represent a finite set of words as a finite tree, where (1) $t$ (of arity 3) and $\text{leaf}$ (of arity 1) are the tree constructors, (2) both the internal nodes and the leaf nodes are labeled by either the constant $\text{accept}$ or the constant $\text{refuse}$, and they are called $\text{accept}$ nodes or $\text{refuse}$ nodes, respectively, and (3) left arcs are labeled by 0 and right arcs are labeled by 1. The empty word is represented by the constant $\text{emptyword}$. The $\text{accept paths}$ of a tree $x$ are the sequences of labels from an $\text{accept}$ node to the root. A word $w$ is member of a set $x$ if $w$ is an accept path of $x$. For instance, the set $\{0, 01\}$ is represented by the tree $t(\text{leaf}(\text{accept}), \text{refuse}, t(\text{leaf}(\text{accept}), \text{refuse}, \text{leaf}(\text{refuse})))$.

We may define the $\text{word}$, $\text{set}$, and $\text{member}$ predicates by means of the following $\text{Member}$ program, which is $P \land R$, where:

$$
\begin{align*}
&\text{P:} \\
&\text{member(\text{emptyword}, leaf(n))} \leftarrow \text{acceptlabel(n)} \\
&\text{member(\text{emptyword}, t(x, n, y))} \leftarrow \text{acceptlabel(n)} \\
&\text{member(s_0(w), t(x, n, y))} \leftarrow \text{member(w, x)} \\
&\text{member(s_1(w), t(x, n, y))} \leftarrow \text{member(w, y)} \\
&\text{acceptlabel(\text{accept})} \\

&\text{R:} \\
&\text{word(\text{emptyword})} \\
&\text{word(s_0(w))} \leftarrow \text{word(w)} \\
&\text{word(s_1(w))} \leftarrow \text{word(w)} \\
&\text{set(leaf(n))} \leftarrow \text{label(n)} \\
&\text{set(t(x, n, y))} \leftarrow \text{set(x), label(n), set(y)} \\
&\text{label(\text{accept})} \\
&\text{label(\text{refuse})}
\end{align*}
$$

We have that the structure $\mathcal{W}$ corresponds to the perfect model (equal to the least Herbrand model) of $\text{Member}$, in the sense that: $\mathcal{W} \models \varphi$ if $M(\text{Member}) \models \varphi$. Since (1) $P$ is a DL program, (2) $R$ is a tree program, and (3) every formula $\varphi$ which is generated by the grammar of Point (i) above is a tree-typed clausal formula over $P \land R$, by Theorem 5 we have that by using our dUFS proof strategy we can test whether or not $M(\text{Member}) \models \varphi$ holds. Thus, dUFS is a decision procedure for cwMS2S. The extension from cwMS2S to cwMSnS (with $n$ successors, instead of 2) can be obtained by a straightforward modification of the $\text{Member}$ program.

There are various possibilities of enhancing the classes of programs for which our dUFS proof strategy terminates. In particular, we may add constant arguments to predicates, we may add nullary predicates, and we may also combine the MR and LD programs in several ways. We do not discuss these enhancements here.
7. Conclusions and Related Work

The idea of using unfold/fold transformations for proving program properties goes back to [17], where it was advocated as a method for proving the equivalence of functional terms. Some limitations and completeness results for this proof method are presented in [2, 9]. Our completeness result (Theorem 4) shows that the unfold, fold, clause deletion, and tautology rules are sufficient to prove the equivalence between tree automata, while Courcelle in [9] proves a similar result by making use of the (restricted) unfold, (restricted) fold, cycle elimination, and redefinition rule. We leave for future research further investigation on the power of the various sets of program transformations.

The present paper extends the techniques proposed in [21] for showing equivalences of definite programs w.r.t. least Herbrand models. The main extensions presented here are the following: (i) we consider logic programs with locally stratified negation and perfect model semantics, (ii) we prove first-order formulas, and (iii) we present an automated strategy for performing proofs. A different extension of [21] has been recently presented in [26], where the authors prove equivalences of definite programs w.r.t. least Herbrand models by using a more powerful folding rule.

Our transformational method for proving properties of the perfect model of a locally stratified logic program is related to other methods for theorem proving. Among these methods we would like to mention the following ones: (i) the method based on the Clark completion [7, 19], (ii) the methods based on resolution augmented by the negation as (finite or infinite) failure rule [1, 23], (iii) the method based on partial evaluation [4], (iv) the methods based on induction, and (v) the proof by consistency method [8, 15, 16, 20].

(i) The method based on the Clark completion amounts to prove that \( M(P) \models \varphi \) by showing that \( \text{comp}(P) \vdash \varphi \), where \( \text{comp}(P) \) denotes the Clark completion of program \( P \). (Equivalently, since the Lloyd-Topor transformation from \( f \leftarrow \varphi \) to \( F(f, \varphi) \) preserves the completion semantics, one may show that \( \text{comp}(P \land F(f, \varphi)) \vdash f \).) This method is sound because \( M(P) \models \text{comp}(P) \) and it reduces the proof of a satisfaction relation to the proof a theorem in first-order predicate calculus. Notice that for some program \( P \) and formula \( \varphi \), such as the ones introduced in our Semaphore and Even-Odd Paths Examples, the relation \( M(P) \models \varphi \) can be proved by our unfold/fold proof method and yet \( \text{comp}(P) \not\models \varphi \). This is due to the fact that the transformation rules presented in this paper, with the only exception of the tautology rule, preserve the perfect model of a program \( P \), which is a model of \( \text{comp}(P) \), but they do not necessarily preserve all models of \( \text{comp}(P) \).

(ii) Several enhancements of the resolution method have been proposed in the literature for verifying properties of logic programs with negation w.r.t. the perfect model semantics (or generalizations thereof). SLDNF-resolution [19] is based on the negation as finite failure rule, but it is unable to deal with infinitely failed derivations and moreover, it is not complete w.r.t. the Clark completion. We have shown that the ability of detecting and eliminating infinitely failed derivations (via the clause deletion rule) is a crucial step in our method for proving theorems.

SLS-resolution [1, 23] enhances the resolution method by using the negation as (finite or infinite) failure rule. In the absence of floundering (see [19] for this notion), SLS-resolution is sound and complete w.r.t. the perfect model semantics, however it is not an effective inference rule, because the set of consequences of the perfect model of a logic program is not recursively enumerable.

SLG-resolution [6] combines resolution and memoing (also called tabling). SLG-resolution is an effective method. It is more powerful than the SLDNF-resolution methods for dealing
with infinitely failed derivations. Indeed, in [25] it has been shown that logic programming systems based on tabling are good for inductive theorem proving and they are able to verify CTL properties of finite-state transition system [11]. However, the unfold/fold proof method is more powerful than logic programming systems based on tabling, because during program transformation one has the possibility of using the folding rule. This rule is more powerful than tabling, because it allows us to tabulate a conjunction of literals, instead of one literal only. Moreover, during program transformation we may apply the goal replacement rule, which is equivalent to applying lemmas in the proofs by induction. Of course, the automation of the unfold/fold method needs more effort than the automation of tabling, but we believe that this difficulty can be overcome by considering specific classes of programs and properties. And indeed, in this paper we have presented two such classes.

(iii) Brass and Dix [4] have proposed a query evaluation algorithm for disjunctive logic programs based on transformation rules. The rules considered in [4] include the positive unfolding and tautology rules, and they preserve several semantics including the perfect model semantics for the class of locally stratified logic programs. However, Brass and Dix do not take into consideration the folding and goal replacement rules, which play a crucial role in our technique.

(iv) The satisfaction relation \( M(P) \models \varphi \) may also be proved by adding to \( \text{comp}(P) \) a set of formulas, or induction schemata, that formalize an induction principle over terms of the Herbrand universe. Thus, one may use standard techniques for inductive theorem proving [3]. The main difference between this method and our unfold/fold proof method is that the latter does not require any induction schema. We leave for future research a formal comparison between the class of formulas that can be proved by induction and the class of formulas that can be proved by the unfold/fold proof method.

(v) The unfold/fold proof method is related to methods for proof by consistency (also called inductionless induction method) of equational formulas by using term rewriting systems (see [15, 16, 20] and [8] for a brief survey). This relationship is based on the ability of the unfold/fold proof method of proving inductive properties without using an explicit induction schema. However, the proofs by consistency are refutational proofs, they work by finding minimal counterexamples, and they require suitable well-founded orderings on terms, while the unfold/fold proof method does not require such term orderings.

Finally, we would like to mention that in [26] the use of unfold/fold proofs (of definite programs) is advocated for verifying parametrized concurrent systems. More generally, since finite and infinite transition systems can be represented in a straightforward way by logic programs, we can use our unfold/fold transformation techniques for proving properties of those systems and thus, performing finite or infinite model checking. The fact that transitions may be subject to negative conditions [14] is not a difficulty in our approach because, as we already mentioned, we allow logic programs with locally stratified negation. The use of logic programming techniques for infinite state model checking seems to be very promising, as shown in [10, 13, 18]. We believe that the combination of these techniques with transformational proof methods may be very fruitful.

Acknowledgements

We would like to thank L. Fribourg and S. Renault for stimulating discussions on various aspects of the work presented in this paper. S. Renault implemented parts of our unfold/fold proof strategy in the MAP transformation system, and performed various experiments of automated theorem proving using that system. We would also like to thank anonymous referees for
constructive comments.

Appendix 1.

Derivation of program $Q$ from clauses 1-6. By unfolding clause 6 w.r.t. $up(x,y)$ we get three clauses: (i) one of them can be deleted because of the atom $0 > 0$ in its body, (ii) a second one: $g \leftarrow down(x)$, $\neg up(sssx,0)$ can also be deleted, because by unfolding $\neg up(sssx,0)$ using clause 3 we get: $g \leftarrow down(x)$, $\neg down(ssx)$, and eventually, we get: $g \leftarrow down(x)$, $\neg down(x)$, and (iii) the third clause produces, after a goal replacement step based on the equivalence $(sx > sy, x > y) \leftrightarrow x > y$, the following clause:

7. $g \leftarrow x > y, up(sx,y), \neg up(sssx,0)$

By the definition and folding rules, from clause 7 we get:

8. $h \leftarrow x > y, up(sx,y), \neg up(sssx,0)$

7. $g \leftarrow h$

By unfolding clause 8 w.r.t. $up(sx,y)$ and by performing the same steps which led us from clause 6 to clause 7, we get:

9. $h \leftarrow x > sy, up(sx,y), \neg up(sssx,0)$

From this clause, by applying the goal replacement rule based on the equivalence $plus(y,s0, sy) \leftrightarrow true$, and then by applying the definition and folding rules, we get:

10. $k(n) \leftarrow x > z, plus(y,n,z), up(sx,y), \neg up(sssx,0)$

9. $f \leftarrow k(s0)$

Notice that clause 10 has been derived by: (i) comparing clause 8 and clause 9, and (ii) generalizing their mismatching terms $y$ and $sy$ to the term $s\ldots s0$ with $n \geq 0$ occurences of $s$, that is, generalizing $y$ and $sy$ to the variable $z$ such that $plus(y,n,z)$. This generalization avoids the introduction of an unbounded number of new definitions, and it makes the UFS strategy to halt. Indeed, by unfolding clause 10 w.r.t. $up(sx,y)$ and $\neg up(sssx,0)$ and by performing simple goal replacement steps involving properties of $j$, and the equivalence $plus(y,n,z) \leftrightarrow plus(y,sn,z)$, we get:

11. $k(n) \leftarrow x > z, plus(y,sn,z), up(sx,y), \neg up(sssx,0)$

which can be folded using clause 10 and we get:

11. $f \leftarrow k(sn)$.

Appendix 2.

Proof of Theorem 3. Point (i) is proved as in Theorem 1. For Point (ii), during the derivation of the stratified program $F(f, \varphi)$ from $f \leftarrow \varphi$, for every derived statement $S$ we preserve the following invariant: (ii.1) $S$ is of the form $newp(x_1, \ldots, x_n) \leftarrow r_1(x_1) \land \ldots \land r_n(x_n) \land \gamma$, where $r_1(x_1), \ldots, r_n(x_n)$ are all tree atoms occurring as conjuncts at top level in the body of the statement, (ii.2) $freevars(S) \subseteq \{x_1, \ldots, x_n\}$, and (ii.3) $\gamma$ is a function-free formula.

This invariant holds for the initial statement $f \leftarrow \varphi$, because $\varphi$ is closed and it is function-free. This invariant is obviously maintained for the application of the rules B.1 and B.2.

Let us now consider the case of rule B.3$^*$. Assume that we apply rule B.3$^*$ to a statement of the form $newp(x_1, \ldots, x_m) \leftarrow r_1(x_1) \land \ldots \land r_n(x_n) \land \gamma$, where $r_1(x_1), \ldots, r_n(x_n)$ are all tree atoms occurring as conjuncts at top level in the body of the statement, (ii.2) $freevars(S) \subseteq \{x_1, \ldots, x_n\}$, and (ii.3) $\gamma$ is a function-free formula. Then we have that $\gamma$ is of the form: $D[\neg \varphi]$ and we derive the two statements: (S1) $A \leftarrow r_1(x_1) \land \ldots \land r_n(x_n) \land D[\neg newp1(y_1, \ldots, y_k)]$, and
(S2) \( \text{newp} \left( y_1, \ldots, y_k \right) \leftarrow t_1(y_1) \land \ldots \land t_k(y_k) \land \varphi \land \psi \), where \( \{y_1, \ldots, y_k\} = \text{freevars}(\varphi \land \psi) \subseteq \{x_1, \ldots, x_n\} \) and \( \{t_1(y_1), \ldots, t_k(y_k)\} \subseteq \{r_1(x_1), \ldots, r_n(x_n)\} \). Thus, for the two statements (S1) and (S2) properties (ii.1), (ii.2), and (ii.3) hold.

The proof for the case of rule B.4\textsuperscript{+} is similar to the one of rule B.3\textsuperscript{+}. The proof for the case of rule B.5 is straightforward because, by construction, \( D[\exists x \varphi] \) is of the form \( D[\exists x (r(x) \land \delta)] \) for some tree atom \( r(x) \) and some formula \( \delta \).

Finally, property (ii.4) of Theorem 3 derives from the fact that the LT transformation generates precisely one clause for every new predicate introduced.

**Appendix 3.**

*Proof of Theorem 4.* We will prove that each iteration of the **FOR** loop terminates. To this aim we use the fact that the following invariant holds after each iteration of the **FOR** loop:

(i1) the derived program \( T \) is a conjunction of: (i) MR clauses, and (ii) definite clauses of the form:

\[
\delta_1. \text{newp}_0(t_1(x_1, \ldots), \ldots, t_m(x_1, \ldots, x_n)) \leftarrow p_1(x_1), \ldots, p_n(x_n) \quad \text{with} \ m, n \geq 0
\]

where: (i) \( t_1, \ldots, t_m \) are function symbols, (ii) \( x_1, \ldots, x_n \) are distinct variables, and (iii) \( p_1, \ldots, p_n \) are predicates whose definitions in \( T \) are conjunctions of MR clauses.

Before the execution of **FOR** loop \( T \) is \( P \), and thus (I1) holds.

Now we suppose that (I1) holds after the \( i \)-th iteration of the **FOR** loop and we show that it is preserved by the \( (i+1) \)-th iteration of the body of the **FOR** loop.

We first prove that after each iteration of the **WHILE** loop (†) the following invariant holds:

(ii2) the derived \( Pos \) is a conjunction of clauses of the form:

\[
\delta_2. \text{newp} \leftarrow r(x), \text{p}_1(x), \ldots, \text{p}_k(x), \neg \text{p}_{k+1}(x), \ldots, \neg \text{p}_i(x)
\]

where: (i) \( r(x) \) is a tree atom, (ii) \( \text{p}_1, \ldots, \text{p}_i \) are unary predicates whose definitions in \( T \) are conjunctions of MR clauses, and (iii) \( \text{newp} \) is either a propositional atom or a unary atom of the form \( \text{newp}_x(x) \).

Indeed, by Theorem 3 and Definition 3, when the **WHILE** loop (†) is initialized, \( Pos \) is a conjunction of clauses of the form:

\[
\delta_3. \text{newp}_0(x_1, \ldots, x_m) \leftarrow r_1(x_1), \ldots, r_n(x_n), \text{p}_1(y_1), \ldots, \text{p}_k(y_k),
\neg \text{p}_{k+1}(y_{k+1}), \ldots, \neg \text{p}(y_i), \neg \text{newp}_1(y_{i+1}, \ldots, y), \ldots, \neg \text{newp}_j(y_a, \ldots, y_c)
\]

where: (i) \( x_1, \ldots, x_m \) are distinct variables occurring in \( \{x_1, \ldots, x_n\} \), (ii) \( y_1, \ldots, y_c \) are not necessarily distinct) variables occurring in \( \{x_1, \ldots, x_n\} \), (iii) the predicates \( r_1, \ldots, r_n \) are tree predicates defined in \( T \), (iv) the predicates \( \text{p}_1, \ldots, \text{p}_k, \text{p}_{k+1}, \ldots, \text{p}_i \) are unary predicates whose definitions in \( T \) are conjunctions of MR clauses, and (v) \( \text{newp}_1, \ldots, \text{newp}_j \) are predicates whose definitions in \( T \) are conjunctions of clauses of the form (\( \delta_1 \)). By applying the BREADTH-FIRST UNFOLD(\( T, \text{Pos}, U \)) substrategy, followed by the TAUTOLOGY(\( T, U, R \)) substrategy, and by the BLOCK-DEFINE & FOLD(\( T, R, \text{Defs}, \text{OutClauses}, \text{NewDefs} \)) substrategy, we derive:

(a) by folding, the clauses in \( \text{OutClauses} \), which are of the form:

\[
\delta_4. \text{newp}_0(t_1(x_1, \ldots), \ldots, t_m(x_1, \ldots, x_n)) \leftarrow \text{newp}_1(x_1), \ldots, \text{newp}_n(x_n), \text{newp}_q
\]

with \( m, n, k \geq 0 \) and

(b) by the definition rule, the clauses in \( \text{NewDefs} \), which are of the form (\( \delta_2 \)). Thus, because of the assignment \( \text{Pos} := \text{NewDefs} \), and since (\( \delta_2 \)) is an instance of (\( \delta_3 \)) (for \( n = 1 \) and \( j = 0 \)), property (I2) holds after each iteration of the **WHILE** loop (†).

By Point (ii) of the BLOCK-DEFINE & FOLD substrategy, if a clause \( C \) occurs in \( \text{Pos} \) at a given iteration, then no clause which differs from \( C \) for the head predicate symbol and the
variable names only, may occur in $Pos$ at a subsequent iteration. Thus, the **while** loop (i) terminates, because there is only a finite number of clauses that are of the form (δ2), modulo the head predicate symbol and the variable names. Moreover, because of the assignment $Out := Out \land Out\text{Clauses}$, all clauses in $Out$ are of the form (δ4), and thus, after the end of the **while** loop (i) and the assignment $T := T \land Out' \land Neg$, program $T$ is a conjunction of: (i) MR clauses, (ii) clauses of the form (δ4) which may include propositional definite clauses, and (iii) clauses in $Neg$, which are propositional clauses with negative literals only in their bodies (recall that, if a variable occurs in a negative literal in the body of a clause in $P^i$, then also a tree atom should occur in its body). Thus, all definitions of nullary predicates in $T$ are conjunctions of propositional clauses.

All propositional clauses in $T$ are replaced by a (possibly empty) conjunction of unit clauses by deleting useless predicates and then performing the sequence of **unfold** and **tautology** steps as indicated in the **while** loop (ii). This **while** loop terminates because, after the deletion of useless predicates, no nullary predicate depends on itself in $T$.

All propositional atoms occurring in the bodies of non-propositional clauses (such as $newq_1, \ldots, newq_k$ in (δ4) with $m > 0$) are eliminated by the unfolding (iii). Thus the iteration of the **for** loop terminates and the invariant (11) holds. □

**Appendix 4.**

**Proof of Theorem 5.** For a tree-typed clausal formula $\varphi$ the LT transformation generates a conjunction of clauses $F(f, \varphi)$ of the form:

\[
\begin{align*}
\delta1. f & \leftarrow -g \\
\delta2. g & \leftarrow r_1(x_1), \ldots, r_n(x_n), p_1(y_1, \ldots, y_j), \ldots, p_k(y_k, \ldots, y_m), \\
& \quad -p_{n+1}(y_{m+1}, \ldots, y_u), \ldots, -p_i(y_i, \ldots, y_z)
\end{align*}
\]

where: (i) $y_1, \ldots, y_z$ are variables occurring in $\{x_1, \ldots, x_n\}$, (ii) the predicates $r_1, \ldots, r_n$ are tree predicates defined in $R$, and (iii) for $s = 1, \ldots, i$, the $Def^s(p_s, P)$ is a DL program. Thus, the predicates defined in $F(f, \varphi)$ have two levels only, and the **for** loop of the dUFS strategy is iterated only twice. The first iteration of the **for** loop starts off with $W = W \land R$, $Pos = \text{clause } \delta2$, and $Neg$ being the empty conjunction.

After each iteration of the **while** loop (i) we have that the derived $Pos$ is a conjunction of clauses of the form:

\[
\begin{align*}
\delta3. newp & \leftarrow r_1(x_1), \ldots, r_m(x_m), \\
& \quad p_1(y_1, \ldots, y_j'), \ldots, p_k(y_k', \ldots, y_m'), -p_{n'+1}(y_{m'+1}, \ldots, y_{m''}), \ldots, -p_{i'}(y_{i'}, \ldots, y_{i''})
\end{align*}
\]

with $i' \leq i$, and the derived $Out$ is a conjunction of propositional definite clauses.

Thus, the **while** loop (i) terminates, because there is only a finite number of clauses that are of the form (δ3) with $i' \leq i$, modulo the head predicate symbol and the variable names.

After the termination of the **while** loop (i) and the assignment $T := T \land Out' \land Neg$ program $T$ is a conjunction of $P \land R$ and propositional definite clauses.

All propositional clauses in $T$ are replaced by a (possibly empty) conjunction of unit clauses by deleting useless predicates and then performing the sequence of **unfold** and **tautology** steps as indicated in the **while** loop (ii). In particular, the definition of $g$ is replaced either by the unit clause $g$ or by the empty conjunction.

The unfoldings (iii) are not performed.

At the second iteration of the **for** loop, the **while** loop (i) is vacuously executed because $Pos$ is the empty conjunction. Clause (δ1) is replaced by: (i) the unit clause $f$, if the definition of $g$ is the empty conjunction, or (ii) by the empty conjunction, otherwise. □
References


