Available online at www.sciencedirect.com
SCIENCE DIRECT®
ELSEVIER
Electronic Notes in Theoretical Computer Science 137 (2005) 5–24
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

Elimination of Local Variables from Definite Logic Programs ⋆

Javier Álvez [1] and Paqui Lucio [2]

Departamento de Lenguajes y Sistemas Informáticos
Universidad del País Vasco
San Sebastián, Spain

Abstract

In logic programming, a variable is said to be local if it occurs in a clause body but not in its head atom. It is well-known that local variables are the main cause of inefficiency (sometimes even incompleteness) in negative goal computation. The problem is twofold. First, the negation of a clause body that contains a local variables is not expressible without universal quantification, whereas the abscence of local variables guarantees that universal quantification can be avoided to compute negation. Second, computation of universal quantification is an intrinsically difficult task. In this paper, we introduce an effective method that takes a definite logic program and transforms it into a *local variable free* (definite) program. Source and target programs are equivalent w.r.t. three-valued logical consequences of program completion. In further work, we plan to extend our results to normal logic programs.

Keywords: local variables, logic programming, program transformation.

1 Introduction

Local variables are very often used in logic programs to store intermediate results that are passed from one atom to another in a clause body. It is well-known that local variables cause several problems for solving negative goals, since they give raise to unavoidable universal quantification in the negation of a clause body. Depending on the LP or CLP approach, universal quantification

⋆ This work has been partially supported by Spanish Projects TIC 2001-2476-C03 and TIN2004-079250-C03-03.

[1] Email: jibalgij@si.ehu.es
[2] Email: jiplucap@si.ehu.es

doi:10.1016/j.entcs.2005.01.037

affects simple goals or constrained goals. In the so-called *intensional negation* (cf. [2]) for the LP approach, universal quantification prevents from achieving a complete goal computation mechanism. Afterwards, constructive negation was introduced in [4,5] and extended in [8,16] to a complete and sound operational semantics for the whole class of normal logic programs in the CLP framework. Intensional negation was also extended to CLP in [3] where a complete operational semantics is provided. The computational mechanisms proposed in [3,8,16] deal with universally quantified (constrained) goals that, in general, are not easy to compute in an efficient manner. Besides, the *negation technique* is introduced in [14] and local variable absence is claimed as a sufficient condition for the completeness of the technique.

In this paper, we present an effective transformation method for eliminating local variables from definite logic programs. The underlying aim is to improve the performance of a practical implementation of constructive negation (cf. [1]). Efficiency is achieved because: (1) the negative query is computed w.r.t. an equivalent definite logic program that does not contain any local variable, hence universal quantification is avoided; and (2) the target program is built at compilation time. We would like to remark that the transformed program (without local variables) must only be used to compute negative literals, using the original one for positive literals. Source and target programs are equivalent w.r.t. the standard Clark-Kunen semantics for normal (in particular, definite) logic programs. In further work, we plan to extend our results to normal logic programs.

Our method is unfold/fold-based in the sense that its correctness is given by an unfold/fold transformation sequence. Besides, the transformation relies in a preliminary partition of the argument positions inside the atoms. This partition, called *mode specification*, associates a mode (input/output) to each argument position. Mode specifications are automatically inferred according to the local variables that are going to be eliminated. The mode specification is only used during local variable elimination and it has neither to do with restricting user-goals nor with the dataflow that is assumed by the programmer. Mode analysis and specification is used for several purposes such as compiler optimization, parallel goal-evaluation, etc. (for instance, [7,10]), which are far from the aim of this work. The elimination method requires a previous syntactical normalization of the program with respect to its local variable occurences.

Outline of the paper. In the next section, we give some preliminary definitions. Program normalization is presented in Section 3. The fourth section introduces the notion of mode specification. In Section 5, we show how to eliminate the local variables from a definite program in several phases. Fi-

nally, we give some conclusions and reflections about the presented, future and related work.

2 Preliminaries

Every program P is built from symbols of a *signature* $\Sigma \equiv \{FS_\Sigma, PS_\Sigma\}$ of function and predicate symbols respectively, and variables from X. Both function and predicate symbols have associated a number $n \geq 0$, called its *arity*. A Σ-*term* is either a variable or a n-ary function symbol of FS_Σ applied to n Σ-terms. A bar is used to denote tuples, or finite sequences, of objects, like $\overline{x}$ as abbreviation of the n-tuple of variables $x_1, \ldots, x_n$. Concatenation of sequences is denoted by the infix operator $\centerdot$ and $\langle\ \rangle$ stands for the empty sequence. We use the symbols $\setminus$ and $\cap$ as binary infix operators for difference and intersection over sequences respectively, with the obvious meaning. From now on r, s, t, u denote terms and x, y, z variables, possibly with bar and sub/super-scripts.

A *substitution* σ is a mapping from a finite set of variables, called its domain, into a set of terms. It is assumed that σ behaves as the identity for the variables outside its domain. As usual, functional composition of substitutions is denoted by their juxtaposition. The *most general unifier* of a set of terms $\{s_1, \ldots, s_n\}$, denoted by $mgu(\overline{s})$, is an idempotent substitution σ such that $\sigma(s_i) \equiv \sigma(s_j)$ for all $1 \leq i, j \leq n$ and for any other substitution θ with the same property, $\theta \equiv \sigma'\sigma$ holds for some substitution σ'.

A Σ-*atom* $p(\overline{t})$ is a n-ary predicate symbol $p \in PS_\Sigma$ applied to a n-tuple of Σ-terms $\overline{t}$; we say (in abuse of language) that $p(\overline{t})$ is an n-ary atom. We also use the two logical constants True and False as atoms. $Form(\Sigma)$ stands for the set of first-order Σ-formulas that can be built using predicate symbols from $PS_\Sigma \cup \{=\}$, connectives form $\{\neg, \wedge, \vee, \rightarrow, \leftrightarrow\}$ and quantifiers from $\{\forall, \exists\}$. The universal closure of a formula φ is denoted by $\varphi^\forall$. The *three-valued logical consequence* relation between set of Σ-formulas and Σ-formulas is denoted by the infix symbol $\models_3$.

A (definite) *clause* C is an expression of the form $a :- \overline{K}$ where a (*head*) is an atom and $\overline{K} \equiv a_1, \ldots, a_m$ (*body*) is a conjunction of atoms a_i for $1 \leq i \leq m$. When the body is empty (or equivalent to True), the clause a is called a *fact*.

Let α be any syntactic object, we denote by $Var(\alpha)$ the sequence of all the variables that occur in α. In a clause, a variable is *local* if it occurs in its body but not in its head. Local variables are divided into *auxiliary* and *isolated* depending on the number of atoms where they occur in. An *auxiliary* variable occurs in more than one atom, whereas an *isolated* variable occurs in just one atom. Anyway, every (non-auxiliary) local variable can be transformed into

an auxiliary variable (see next section). For α being a clause or any object (atom, term, etc) that occurs in a clause, we denote by $AuxVar(\alpha)$ the set of auxiliary variables in α. Similarly, $\overline{Aux}\,Var(\alpha)$ denotes the set of *non-auxiliary* variables.

A (definite) *program* P is defined by a collection of (definite) clauses. We use the term Σ-*program* whenever the signature is relevant. For a predicate p, we denote by $\mathtt{Def}_P(p)$ the set of all clauses in P with head p. All definitions in this paper are given modulo reordering of the clause bodies and standardization apart is always assumed. Given $p, q \in PS_\Sigma$ and a Σ-program P, we say that p *directly depends* on q if q occurs in some clause in $\mathtt{Def}_P(p)$. By reflexive transitive closure of this relation, we obtain the set $\mathtt{Dpd}_P(p)$ such that, w.r.t. program P, p depends on all predicates in $\mathtt{Dpd}_P(p)$ ($p \in \mathtt{Dpd}_P(p)$ for every p). Besides, $\mathtt{MR}_P(p) \equiv \bigcup \{q \mid q \in \mathtt{Dpd}_P(p) \text{ and } p \in \mathtt{Dpd}_P(q)\}$ is the set of all mutually recursive predicates with p and $\mathtt{MRDef}_P(p) \equiv \bigcup \{\mathtt{Def}_P(q) \mid q \in \mathtt{MR}_P(p)\}$.

The standard declarative meaning of normal (in particular, definite) logic programs is the program completion (proposed by Clark in [6]), interpreted in three-valued logic (as proposed in [11]), that is also known as Clark-Kunen semantics. The *predicate completion formula of a predicate* p such that $\mathtt{Def}_P(p) \equiv \{p(\overline{t}^i) :- \overline{q}^i(\overline{s}^i) | i \in 1..m\}$ is the sentence:

$$\forall \overline{x}(\ p(\overline{x}) \leftrightarrow \bigvee_{i=1}^{m} \exists \overline{y}^i (\overline{x} = \overline{t}^i \wedge \overline{q}^i(\overline{s}^i))\ )$$

where each $\overline{y}^i \equiv Var(\overline{t}^i)\,\mathbf{.}\,Var(\overline{s}^i)$. The *Clark's completion of a program* P, namely $Comp(P)$, consists of the universal closure of the set P^* of the predicate completion formulas for every $p \in PS_\Sigma$ together with the *free equality theory* $FET(\Sigma)$. Program transformation preserves some equivalence relation on programs. Different equivalence relations are induced by the different semantics (see [12] for a systematic comparison). Since we plan to extend our results to normal programs, we are interested in completion semantics. Shepherdson's operators T^P and F^P (for a program P) were introduced in [15] and provide a characterization of the three-valued logical consequences of program completion.

Definition 2.1 **([15])** *Let $p \in PS_\Sigma$ be defined by a set of clauses $\mathtt{Def}_P(p)$ as above:*

$$T_0^P(p(\overline{x})) \equiv \mathit{False} \qquad T_{n+1}^P(p(\overline{x})) \equiv \bigvee_{i=1}^{m} \exists \overline{y}^i (\overline{x} = \overline{t}^i \wedge T_n^P(\overline{q}^i(\overline{s}^i)))$$

$$F_0^P(p(\overline{x})) \equiv \mathit{False} \qquad F_{n+1}^P(p(\overline{x})) \equiv \bigwedge_{i=1}^{m} \neg\exists \overline{y}^i (\overline{x} = \overline{t}^i \wedge \neg F_n^P(\overline{q}^i(\overline{s}^i)))$$

where $\overline{y}^i \equiv Var(\overline{t}^i) \centerdot Var(\overline{s}^i)$. *Besides,* $T_n(\mathit{True}) \equiv \mathit{True}$ *and* $F_n(\mathit{True}) \equiv \mathit{False}$ *for all* n. *The extension to connectives and quantifiers is the obvious one.* □

Notice that $T_k^P(p(\overline{x}))$ and $F_k^P(p(\overline{x}))$ are formulas (in first-order logic with equality) that represent the successes and failures of the atom $p(\overline{x})$ which can be derived from P in k steps. Facts produce the one level successes since their clause body is True.

Theorem 2.2 *For any normal (in special, definite) program* P *and any sentence* φ, $Comp(P) \models_3 \varphi \Leftrightarrow$ *there exists* $n \in \mathbb{N}$ *such that* $FET(\Sigma) \models_3 T_n^P(\varphi)$.

Proof. Lemma 4.1 in [15] proves the equivalence $\Phi_P^n \models_3 \varphi \Longleftrightarrow FET(\Sigma) \models_3 T_n^P(\varphi)$ for all $n \in \mathbb{N}$, where Φ_P^n is the n-iteration of Fitting's immediate consequence operator Φ_P (cf. [9]). Besides, Kunen's Theorem 6.3 in [11] proves that $Comp(P) \models_3 \varphi \Longleftrightarrow$ there exists $n \in \mathbb{N}$ such that $\Phi_P^n \models_3 \varphi$. □

As a consequence, the three equivalence relations that are induced on the set of programs by Fitting's and Shepherdson's operators and by logical consequence of program completion have the same strength. *TF*-equivalence is a useful representative of these three notions. Below, we give a more precise definition of the *TF*-equivalence ($\cong_{TF}$) relation on programs. A strictly stronger equivalence relation is given by logical equivalence of program completions. That means to require $FET(\Sigma) \models_3 Comp(P_1) \leftrightarrow Comp(P_2)$ for the equivalence of the programs P_1 and P_2. The interested reader is referred to [12] for an example of *TF*-equivalent programs whose completions are not logically equivalent.

Definition 2.3 *Let* P_1 *and* P_2 *be two* Σ*-programs:*

(i) $P_1 \preceq_{TF} P_2$ *iff for all* $p \in PS_\Sigma^1 \cap PS_\Sigma^2$ *and for all* $k \in \mathbb{N}$, *there exists* $k' \in \mathbb{N}$ *such that* $FET(\Sigma) \models_3 (T_k^{P_1}(p(\overline{x})) \rightarrow T_{k'}^{P_2}(p(\overline{x})) \wedge F_k^{P_1}(p(\overline{x})) \rightarrow F_{k'}^{P_2}(p(\overline{x})))^\forall$

(ii) $P_1 \cong_{TF} P_2$ *iff* $P_1 \preceq_{TF} P_2$ *and* $P_2 \preceq_{TF} P_1$. □

Intuitively, *TF*-equivalent programs have equivalent sets of answers, but not necessarily obtained at the same iteration step. We transform a Σ-program into a *TF*-equivalent Σ'-program without local variables where $\Sigma' \supseteq \Sigma$.

3 Normalization

Program normalization is a preliminary treatment of the local variables occurrences which enables the subsequent elimination method. Here, we explain in detail the syntactic requirements of normalization and we also show that any definite logic program can be transformed into a *TF*-equivalent normalized

one.

The syntactic restriction affects single clauses.

Definition 3.1 *A clause* $C \equiv p(\overline{t}) :- q_1(\overline{s}^1), \ldots, q_n(\overline{s}^n)$ *is* normalized *iff it satisfies the following two conditions:*

(i) *Every local variable* y *exactly occurs in the atoms* $q_{i-1}(\overline{s}^{i-1})$ *and* $q_i(\overline{s}^i)$ *for some* $2 \leq i \leq n$ *and does not occur anymore in* C

(ii) *Let* m_i *be the arity of the predicate* q_i *then, for every* $2 \leq i \leq n-1$ *and every* $1 \leq j \leq m_i$, *either* $AuxVar(s^i_j) \subset AuxVar(\overline{s}^{i-1})$ *or* $AuxVar(s^i_j) \subset AuxVar(\overline{s}^{i+1})$. □

Then, normalization is extended from clauses to programs in the obvious way.

Definition 3.2 *A program* P *is* normalized *iff every clause* $C \in P$ *is normalized.* □

Theorem 3.3 *Every definite logic program* P *can be transformed into a TF-equivalent normalized program* P'.

Proof. Let suppose that the clause $C \equiv p(\overline{t}) :- q_1(\overline{s}^1), \ldots, q_n(\overline{s}^n) \in P$ is not normalized. Then, there are two possibilities, depending on the condition in Definition 3.1 that does not hold.

If condition (i) does not hold, there exists at least a local variable y that violates it. Let $q_i(\overline{s}^i)$ $(1 \leq i \leq n)$ be the leftmost atom of C where the local variable y occurs in. We replace the atom $q_{i+1}(\overline{s}^{i+1})$ with $q'_{i+1}(\overline{r}^{i+1})$ where:

- q'_{i+1} is a new predicate
- $\overline{r}^{i+1} \equiv \overline{s}^{i+1} \centerdot y \centerdot y'$
- y' is a new local variable.

The definition of the new predicate q'_{i+1} is given by the clause $q'_{i+1}(\overline{z}') \equiv q_{i+1}(\overline{z})$ where $\overline{z}' \equiv \overline{z} \centerdot x \centerdot x$ and $x \notin \overline{z}$. In addition, we replace the atom $q_k(\overline{s}^k)$ with $q_k(\overline{s}^k)[y'/y]$ for every $i+2 \leq k \leq n$. Note that for $i = n$, we only need to add a new atom $p'(y)$ where p' is a new predicate that is defined by the single clause $p'(\_)$. The local variable y does already satisfy the first condition, since it occurs in two consecutive atoms and does not occur anymore in the clause. The process ends since either the number of local variables which violate the condition (i) decreases (that is, y' satisfies the condition (i)) or this number does not decrease but the new local variable y' occurs in the atom that is one step closer to the end of the clause body. Besides, source and target programs are proved to be TF-equivalent by unfolding the new atoms.

If condition (ii) is violated, we suppose (without loss of generality) that condition (i) holds for every clause in the program. Then, let $q_i(\overline{s}^i)$ be the left-

most atom of C such that for some $1 \leq j \leq m$ (being m the arity of the predicate q_i) $AuxVar(s_j^i) \cap AuxVar(\overline{s}^{i-1}) \not\equiv \emptyset$ and $AuxVar(s_j^i) \cap AuxVar(\overline{s}^{i+1}) \not\equiv \emptyset$. We replace the atom $q_i(\overline{s}^i)$ with $q_i'(\overline{r}^i)$ where:

- q_i' is a new predicate
- $\overline{x} \equiv AuxVar(s_j^i) \cap AuxVar(\overline{s}^{i-1})$,
- $\overline{r}^i \equiv \overline{t}^i \,.\, \overline{x} \,.\, \overline{x}'$,
- $\overline{x}'$ is a tuple of new variables that corresponds with $\overline{x}$
- $\overline{t}^i$ is obtained by substituting $s_j^i[\overline{x}'/\overline{x}]$ for s_j^i in $\overline{s}^i$.

The definition of q_i' is given by the clause $q_i'(\overline{z}') :- q_i(\overline{z})$ where $\overline{z}' \equiv \overline{z} \,.\, \overline{x} \,.\, \overline{x}$ and $\overline{x} \cap \overline{z} \equiv \emptyset$. In addition, we replace the atom $q_{i+1}(\overline{s}^{i+1})$ with $q_{i+1}'(\overline{r}^{i+1})$ where q_{i+1}' is also a new predicate and $\overline{r}^{i+1} \equiv \overline{s}^{i+1} \,.\, \overline{x}'$. The definition of the predicate q_{i+1}' is given by the clause $q_{i+1}'(\overline{z}') :- q_{i+1}(\overline{z})$ where $\overline{z}' \equiv \overline{z} \,.\, \overline{x}$ and $\overline{x} \cap \overline{z} \equiv \emptyset$. In the resulting clause C', the term r_j^i in the atom $p_i'(\overline{r}^i)$ satisfies the condition (ii) since now $AuxVar(r_j^i) \subset AuxVar(\overline{r}^{i+1})$ and $AuxVar(r_j^i) \cap AuxVar(\overline{r}^{i-1}) \equiv \emptyset$. Furthermore, the new introduced tuples of terms in atoms $q_i'(\overline{r}^i)$ and $q_{i+1}'(\overline{r}^{i+1})$ also satisfy the condition (ii). Besides, condition (i) is preserved in the clause C'. This process also ends since the number of terms that violate the condition (ii) strictly decreases at each step. As above, the programs P and P' are proved to be TF-equivalent by unfolding the new atoms. □

As a consequence, the problem of local variable elimination is reduced to auxiliary variable elimination. From now on, normalization is always assumed in programs. In particular, normalized programs does not contain any isolated variables. Moreover, every auxiliary variable always occurs in two consecutive atoms in a normalized clause.

Example 3.4 *Consider the following program P:*

$$E3.4.1 : \ preorder(nil, [\,])$$
$$E3.4.2 : \ preorder(tree(x_1, x_2, x_3), x_4) :- \ preorder(x_2, y_1), \ preorder(x_3, y_2), \ append([x_1|y_1], y_2, x_4)$$

The clause $E3.4.2$ is not normalized because the local variable y_1 does not satisfy the condition (i) of Definition 3.1. We obtain a TF-equivalent normalized program by substituting the next two clauses for the clause $E3.4.2$:

$$E3.4.3 : \ preorder(tree(x_1, x_2, x_3), x_4) :- \ preorder(x_2, y_1), \ preorder'(x_3, y_2, y_1, y_1'), \ append([x_1|y_1'], y_2, x_4)$$
$$E3.4.4 : \ preorder'(x_1, x_2, w, w) :- \ preorder(x_1, x_2)$$ □

4 Mode Specification

In order to define the mode specification, it is worthwhile to distinguish between the argument position j and the corresponding argument u_j for some $1 \leq j \leq m$ in a given atom $p(u_1, \ldots, u_m)$. Then, each argument position is associated to a *mode*, that can be either *input* (`in`) or *output* (`out`). The resulting partition of the n argument positions in $p(\overline{u})$ is a *mode specification*.

Definition 4.1 *The* mode specification in an m-ary atom a, *denoted by* $\mathtt{ms}(a)$, *is either* $\perp$ *or a* m*-tuple* $(\mathtt{ms}_1(a), \ldots, \mathtt{ms}_m(a)) \subseteq \{\mathtt{in}, \mathtt{out}\}^m$. $\mathtt{ms}(a)$ *is* undefined *if it is* $\perp$. *Otherwise, it is* defined. □

For example, if the mode specification $(\mathtt{out}, \mathtt{in}, \mathtt{out}, \mathtt{in})$ is in the atom $p(f(a, y), g(a), y, x)$, then $\langle g(a), x\rangle$ is the order-preserving sequence of arguments occurring in input positions, whereas $\langle f(a, y), y\rangle$ corresponds with the output positions.

For the rest of the paper, we adopt the following notation that allows us to implicitly represent the mode specification in an atom.

Notational Convention 4.2 *Suppose some fixed mode specification* $\mathtt{ms}(p(\overline{u}))$. *The atom* $p(\overline{u})$ *is written as* $p(\!|\overline{u}_{\mathtt{I}} \triangleright \overline{u}_{\mathtt{O}}|\!)$ *to denote that* $\overline{u}_{\mathtt{I}}$ *and* $\overline{u}_{\mathtt{O}}$ *are the order-preserving subsequences of* $\overline{u}$ *that respectively correspond with the input and the output positions.* □

For instance, in the previous example, the atom $p(f(a, y), g(a), y, x)$ whose mode specification is $(\mathtt{out}, \mathtt{in}, \mathtt{out}, \mathtt{in})$ is written as $p(\!|\langle g(a), x\rangle \triangleright \langle f(a, y), y\rangle|\!)$.

Definition 4.3 *Let* P *be a program and* $C \equiv p(\overline{t}) :- q_1(\overline{s}^1), \ldots, q_n(\overline{s}^n) \in P$ *be a clause, the* mode specification in the clause C, *denoted by* $\mathtt{ms}(C)$, *is a* $n+1$*-tuple* $(\ \mathtt{ms}(p(\overline{t})), \mathtt{ms}(q_1(\overline{s}^1)), \ldots, \mathtt{ms}(q_n(\overline{s}^n))\ )$. *If* $\mathtt{ms}(p(\overline{t}))$ *and* $\mathtt{ms}(q_i(\overline{s}^i))$ *are defined for each* $1 \leq i \leq n$ *then* $\mathtt{ms}(C)$ *is* total. *Otherwise,* $\mathtt{ms}(C)$ *is* partial. □

We would like to remark that mode specifications are about positions, but not about terms being their actual holders. Notice that the same term (in particular, variable) could occur in distinct atoms (or, even, in the same atom) in positions with different modes. Moreover, the mode specification does not restrict the goals and has nothing to do with the dataflow that is assumed by the programmer.

In this section, we will explain how to *automatically infer the mode specification* in the atoms of a program. This inference will be directed by the auxiliary variables that are going to be eliminated. To start with, the following two definitions set the criteria for inferring the mode specification in a clause regarding the occurrence of the auxiliary variables. Intuitively, auxil-

iary variables take a value in their leftmost occurrence atom (mode `out`) that is used in the remaining atoms (mode `in`).

Definition 4.4 *Let $C \equiv p(\overline{t}) :- q_1(\overline{s}^1), \ldots, q_n(\overline{s}^n)$ be a clause and m_i the arity of the atom q_i, the* mode specification in the j-th argument position of the atom $q_i(\overline{s}^i)$ *for $1 \leq j \leq m_i$, denoted by $\mathtt{ms}_j(q_i(\overline{s}^i))$, such that $AuxVar(s^i_j) \not\equiv \emptyset$ is:*

- $\mathtt{ms}_j(q_i(\overline{s}^i)) := \mathtt{in}$ *if* $AuxVar(s^i_j) \subset AuxVar(\overline{s}^{i-1})$
- $\mathtt{ms}_j(q_i(\overline{s}^i)) := \mathtt{out}$ *if* $AuxVar(s^i_j) \subset AuxVar(\overline{s}^{i+1})$ □

However, the mode specification in the argument positions where no auxiliary variable occurs in is defined according to a given mode specification in the clause head atom.

Definition 4.5 *Let $C \equiv p(\overline{t}) :- q_1(\overline{s}^1), \ldots, q_n(\overline{s}^n)$ be a clause such that $\mathtt{ms}(p(\overline{t}))$ is defined and m_i be the arity of the predicate q_i, the* mode specification in the j-th argument position of the atom $q_i(\overline{s}^i)$ *for $1 \leq j \leq m_i$, denoted by $\mathtt{ms}_j(q_i(\overline{s}^i))$, such that $AuxVar(s^j_i) \equiv \emptyset$ is:*

- $\mathtt{ms}_j(q_i(\overline{s}^i)) := \mathtt{in}$ *if* $AuxVar(s^i_j) \cap AuxVar(\overline{t}_{\mathtt{I}}) \not\equiv \emptyset$
- $\mathtt{ms}_j(q_i(\overline{s}^i)) := \mathtt{out}$ *if* $AuxVar(s^i_j) \cap AuxVar(\overline{t}_{\mathtt{I}}) \equiv \emptyset$ □

In the sequel, given a mode specification in an atom $p(\overline{u})$ of a program P, we extend it to the clauses that define all the predicates that are mutually recursive with p, that is, to $\mathtt{MRDef}_P(p)$. This extension is made in different phases. First, $\mathtt{ms}(p(\overline{u}))$ is extended to the set of clauses $\mathtt{Def}_P(p)$. Second, we collect the mode specifications in the atoms of predicates $h \in \mathtt{MR}_P(p)$. Finally, each collected mode specification is extended to $\mathtt{Def}_P(h)$. In the next section, we will see how the starting mode specification $\mathtt{ms}(p(\overline{u}))$ is obtained.

Definition 4.6 *Let P be a program and $\mathtt{ms}(p(\overline{u}))$ be a mode specification, the* mode specification of $\mathtt{Def}_P(p)$ w.r.t. $\mathtt{ms}(p(\overline{u}))$, *denoted by $\mathtt{MS}[\mathtt{Def}_P(p)\backslash\mathtt{ms}(p(\overline{u}))]$, consists of the mode specification $\mathtt{ms}(C)$ in each clause $C \equiv p(\overline{t}) :- \overline{K} \in \mathtt{Def}_P(p)$ where $\mathtt{ms}(p(\overline{t})) \equiv \mathtt{ms}(p(\overline{u}))$, according to Definitions 4.4 and 4.5.* □

Definition 4.7 *Let P be a program and $\mathtt{ms}(p(\overline{u}))$ a mode specification in an atom of P, the* set of mode specifications in $\mathtt{Def}_P(p)$ w.r.t. $\mathtt{ms}(p(\overline{u}))$, *denoted by $\mathtt{LMS}[\mathtt{Def}_P(p)\backslash\mathtt{ms}(p(\overline{u}))]$, is defined by:*

$$\{\ (h, \mathtt{ms}(h(\overline{s}))) \mid h \in \mathtt{MR}_P(p) \text{ and } \mathtt{ms}(h(\overline{s})) \in \mathtt{MS}[\mathtt{Def}_P(p)\backslash\mathtt{ms}(p(\overline{u}))]\ \}$$

The set of mode specifications in $\mathtt{MRDef}_P(p)$ w.r.t. $\mathtt{ms}(p(\overline{u}))$, *denoted by $\mathtt{LMS}[\mathtt{MRDef}_P(p)\backslash\mathtt{ms}(p(\overline{u}))]$, is obtained by transitive closure.* □

Definition 4.8 *Let P be a program and $\mathtt{ms}(p(\overline{u}))$ be a mode specification in an atom of P, the* mode specification in $\mathtt{MRDef}_P(p)$ w.r.t. $\mathtt{ms}(p(\overline{u}))$, *denoted*

by $\mathtt{MS}[\mathtt{MRDef}_P(p)\backslash\mathtt{ms}(p(\overline{u}))]$*, is defined by:*

$$\{\ \mathtt{MS}[\mathtt{Def}_P(h)\backslash\mathtt{ms}(h(\overline{s}))] \mid \mathtt{ms}(h(\overline{s})) \in \mathtt{LMS}[\mathtt{MRDef}_P(p)\backslash\mathtt{ms}(p(\overline{u}))]\ \}$$

□

It is important to stress that there exists a unique $\mathtt{MS}[\mathtt{MRDef}_P(p)\backslash\mathtt{ms}(p(\overline{u}))]$ for each $\mathtt{ms}(p(\overline{u}))$.

Example 4.9 *Given the normalized program P in Example 3.4, the mode specification in $\mathtt{MRDef}_P(preorder)$ w.r.t. $(\mathtt{in},\mathtt{out})$ is (in the Notational Convention 4.2 for implicitly representing it):*

$$\begin{array}{ll} E4.9.1: & preorder(\!|nil \triangleright [\,]|\!) \\ E4.9.2: & preorder(\!|tree(x_1,x_2,x_3) \triangleright x_4|\!) :- \ preorder(\!|x_2 \triangleright y_1|\!), \\ & \qquad preorder'(\!|x_3, y_1 \triangleright y_2, y_1'|\!),\ append(\!|[x_1|y_1'], y_2 \triangleright x_4|\!) \\ E4.9.3: & preorder'(\!|x_1, w \triangleright x_2, w|\!) :- \ preorder(\!|x_1 \triangleright x_2|\!) \end{array}$$

where $\mathtt{LMS}[\mathtt{MRDef}_P(preorder)\backslash(\mathtt{in},\mathtt{out})]$ *is:*

$$\{\ (preorder,(\mathtt{in},\mathtt{out})),(preorder',(\mathtt{in},\mathtt{out},\mathtt{in},\mathtt{out}))\ \}$$

Notice that, in this case, each predicate is associated to a unique mode specification, although in general there can be several mode specifications associated to each one. □

5 The Elimination Method

Next, we present a method for transforming definite logic programs in order to eliminate the auxiliary variables while preserving TF-equivalence. We devoted the first two subsections to explain in detail the two main subtasks of this transformation method: to transform the definition of a predicate p into a tail recursive definition (w.r.t. a mode specification, see below) and to eliminate the auxiliary variables that are located in the leftmost atom where some of them occur in. The algorithm that, using these two subtasks, accomplishes the auxiliary variable elimination is introduced in the third subsection.

5.1 Tail Recursive Transformation

In Prolog, a predicate is said to be tail recursive whenever the recursive call is the rightmost in every recursive clause that defines the predicate. However, we define a slightly stronger notion by relating tail recursion to the mode specification and by imposing some extra syntactic restrictions.

Definition 5.1 *The definition of a predicate p in a program P is* tail recursive w.r.t. a mode specification $\mathtt{ms}(p(\overline{u}))$ *iff for each pair* $(h,\mathtt{ms}(h(\overline{u}'))) \in$

$\mathtt{LMS}[\mathtt{MRDef}_P(p)\backslash\mathtt{ms}(p(\overline{u}))]$, $h \equiv p$ *and* $\mathtt{Def}_P(h)$ *consists of clauses of the following two forms (w.r.t.* $\mathtt{MS}[\mathtt{Def}_P(h)\backslash\mathtt{ms}(h(\overline{u}'))]$*):*

(1) $h(\!|\overline{t}_{\mathtt{I}} \triangleright \overline{t}_{\mathtt{O}}|\!) :- \overline{K}$

(2) $h(\!|\overline{s}_{\mathtt{I}} \triangleright \overline{z}|\!) :- \overline{L},\ h(\!|\overline{r}_{\mathtt{I}} \triangleright \overline{z}|\!)$

where $h \notin \mathtt{Dpd}_P(q)$ *(that is,* $q \notin \mathtt{MR}_P(p)$*) for every atom* $q(\overline{u}')$ *in* $\overline{K}$ *and* $\overline{L}$*, and* $\overline{z}$ *is a fresh tuple of pairwise distinct variables.* □

That is, as well as the standard condition of recursion in the rightmost atom, we also demand that, in the recursive clauses (2), the same tuple of pairwise distinct variables (namely $\overline{z}$) occurs in output arguments of both the head and the rightmost body atom. Notice that only direct recursion is considered in our definition. These restrictions will be useful during the auxiliary variable elimination process.

Next, we show how to transform the definition of a predicate into a *TF*-equivalent tail recursive one w.r.t. a mode specification. It is based on the well-known technique that uses a stack for storing the recursive calls. Here, we use new constants with predicate names and clauses as super/sub-scripts to be stored in the stack for representing them.

Definition 5.2 *Let* P *be a program and* $\mathtt{ms}(p(\overline{u}))$ *be the mode specification in an atom that occurs in* P*, the* tail recursive definition of p w.r.t. $\mathtt{ms}(p(\overline{u}))$*, denoted by* $\mathtt{TailRDef}_P[p\backslash\mathtt{ms}(p(\overline{u}))]$*, consists of (where, in each clause,* $\overline{z}$ *is a* m*-tuple of fresh variables and* m *is the number of output positions in* $\mathtt{ms}(p(\overline{u}))$*):*

- *A clause (really, a fact):*

  (1) $p\_tr(\!|\overline{z}, [\,] \triangleright \overline{z}|\!)$

  where p_tr *is a new predicate.*

- *For each* $(h, \mathtt{ms}(h(\overline{s}))) \in \mathtt{LMS}[\mathtt{MRDef}_P(p)\backslash\mathtt{ms}(p(\overline{u}))]$*, a clause:*

  (2) $h(\!|\overline{x} \triangleright \overline{z}|\!) :- p\_tr(\!|\overline{x}, [c_h] \triangleright \overline{z}|\!)$

  where the constant c_h *is associated to* $\mathtt{ms}(h(\overline{s}))$*,* $\overline{x}$ *is a* k*-tuple of new variables and* k *is the number of output positions in* $\mathtt{ms}(h(\overline{s}))$*.*

- *For each* $h(\!|\overline{s}_{\mathtt{I}} \triangleright \overline{s}_{\mathtt{O}}|\!) :- q_1(\!|\overline{r}^1_{\mathtt{I}} \triangleright \overline{r}^1_{\mathtt{O}}|\!), \ldots, q_n(\!|\overline{r}^n_{\mathtt{I}} \triangleright \overline{r}^n_{\mathtt{O}}|\!) \in \mathtt{MS}[\mathtt{MRDef}_P(p)\backslash\mathtt{ms}(p(\overline{u}))]$ *such that* $\{q_1, \ldots q_n\} \cap \mathtt{MR}_P(p) \equiv \emptyset$*, a clause:*

  (3) $p\_tr(\!|\overline{s}_{\mathtt{I}}, [c_h|S] \triangleright \overline{z}|\!) :- q_1(\!|\overline{r}^1_{\mathtt{I}} \triangleright \overline{r}^1_{\mathtt{O}}|\!),\ \ldots,\ q_n(\!|\overline{r}^n_{\mathtt{I}} \triangleright \overline{r}^n_{\mathtt{O}}|\!),\ p\_tr(\!|\overline{s}_{\mathtt{O}}, S \triangleright \overline{z}|\!)$

- *For each* $h(\!|\overline{s}_{\mathtt{I}} \triangleright \overline{s}_{\mathtt{O}}|\!) :- q_1(\!|\overline{r}^1_{\mathtt{I}} \triangleright \overline{r}^1_{\mathtt{O}}|\!),\ \ldots, q_n(\!|\overline{r}^n_{\mathtt{I}} \triangleright \overline{r}^n_{\mathtt{O}}|\!) \in \mathtt{MS}[\mathtt{MRDef}_P(p)\backslash\mathtt{ms}(p(\overline{u}))]$*, namely* $\mathtt{ms}(C)$*, such that* $\{q_1, \ldots q_n\} \cap \mathtt{MR}_P(p) \not\equiv \emptyset$*, two clauses*[3] *:*

  (4) $p\_tr(\!|\overline{s}_{\mathtt{I}}, [c_h|S] \triangleright \overline{z}|\!) :- p\_tr(\!|\overline{s}_{\mathtt{I}}, [\overline{w}^1, c^C_{q_1}, \ldots, \overline{w}^n, c^C_{q_n}, \overline{w}, c^C_h|S] \triangleright \overline{z}|\!)$

[3] By convention, $\overline{r}^0_{\mathtt{O}} \equiv \overline{s}_{\mathtt{I}}$.

(5) $p\_tr(\langle \overline{r}_{\mathtt{O}}^{n}, [\overline{w}, c_h^C | S] \triangleright \overline{z} \rangle) :- \; p\_tr(\langle \overline{s}_{\mathtt{O}}, S \triangleright \overline{z} \rangle)$

and, for each $1 \leq j \leq n$, *a clause of the form* (6) *if* $q_j \notin \mathtt{MR}_P(p)$ *and a clause of the form* (7) *if* $q_j \in \mathtt{MR}_P(p)$*:*

(6) $p\_tr(\langle \overline{r}_{\mathtt{O}}^{j-1}, [\overline{w}^j, c_{q_j}^C | S] \triangleright \overline{z} \rangle) :- \; q_j(\langle \overline{r}_{\mathtt{I}}^{j} \triangleright \overline{r}_{\mathtt{O}}^{j} \rangle), p\_tr(\langle \overline{r}_{o}^{j}, S \triangleright \overline{z} \rangle)$

(7) $p\_tr(\langle \overline{r}_{\mathtt{O}}^{j-1}, [\overline{w}^j, c_{q_j}^C | S] \triangleright \overline{z} \rangle) :- \; p\_tr(\langle \overline{r}_{\mathtt{I}}^{j}, [c_{q_j} | S] \triangleright \overline{z} \rangle)$

where:

- *each constant* $c_{q_j}^C$ *is associated to* $q_j(\langle \overline{s}_{\mathtt{I}}^{j} \triangleright \overline{s}_{\mathtt{O}}^{j} \rangle)$ *in* $\mathtt{ms}(C)$
- *the constant* c_h^C *is associated to* $\mathtt{ms}(C)$ *itself*
- $\overline{w} \equiv (\bigcup_{k=1}^{n} \overline{v}^k \setminus Var(\overline{s}_{\mathtt{I}})) \cup (Var(\overline{s}_{\mathtt{O}}) \setminus \overline{Aux}\, Var(\overline{r}_{\mathtt{O}}^{n}))$
- $\overline{w}^j \equiv \overline{v}^j \cup (\overline{Aux}\, Var(\overline{r}_{\mathtt{I}}^{j} \centerdot \overline{r}_{\mathtt{O}}^{j}) \setminus \overline{Aux}\, Var(\overline{r}_{\mathtt{O}}^{j-1}))$ *if* $q_j \notin \mathtt{MR}_P(p)$ *(type (6))*
- $\overline{w}^j \equiv \overline{v}^j \cup (\overline{Aux}\, Var(\overline{r}_{\mathtt{I}}^{j}) \setminus \overline{Aux}\, Var(\overline{r}_{\mathtt{O}}^{j-1}))$ *if* $q_j \in \mathtt{MR}_P(p)$ *(type (7))*
- $\overline{v}^j \equiv \overline{Aux}\, Var(\overline{r}_{\mathtt{I}}^{j} \centerdot \overline{r}_{\mathtt{O}}^{j}) \cap \bigcup_{k=1,k\neq j}^{n} \overline{Aux}\, Var(\overline{r}_{\mathtt{I}}^{k} \centerdot \overline{r}_{\mathtt{O}}^{k})$ □

Roughly speaking, the clauses of the form (2) store the initial call for each $(h, \mathtt{ms}(h(\overline{s}))) \in \mathtt{LMS}[\mathtt{MRDef}_P(p) \backslash \mathtt{ms}(p(\overline{u}))]$, whereas the clause (1) gives the output value when no recursive call remains. The clauses of the form (3) encode the non-recursive clauses and the clauses of the form (4) and (5) the recursive ones (each clause (5) gives the output value of the original clause). Notice that the clauses are encoded in different ways according to the each pair $(h, \mathtt{ms}(h(\overline{s}))) \in \mathtt{LMS}[\mathtt{MRDef}_P(p) \backslash \mathtt{ms}(p(\overline{u}))]$. The clauses of the form (6) and (7) are used for rebuilding the set of terms of each atom depending on the mode specification in it. In addition, the variables stored in the stack ensure that the links between atoms through (non-auxiliary) variables are kept.

Theorem 5.3 *Let* P *be a program and* $\mathtt{ms}(p(\overline{u}))$ *the mode specification in an atom that occurs in* P*, the programs* P *and* $P \backslash \mathtt{Def}_P(p) \cup \mathtt{TailRDef}_P[p \backslash \mathtt{ms}(p(\overline{u}))]$ *are TF-equivalent.*

Proof. Let P_0 be obtained by adding to $P \setminus \mathtt{Def}_P(p)$ one set of clauses of the following form for each pair $(h, \mathtt{ms}(h(\overline{u}'))) \in \mathtt{LMS}[\mathtt{MRDef}_P(p) \backslash \mathtt{ms}(p(\overline{u}))]$:

$C5.3.1: \quad h(\langle \overline{x} \triangleright \overline{z} \rangle) :- \; h'(\langle \overline{x}, [c_h] \triangleright \overline{z} \rangle)$

$C5.3.2^*: \quad h'(\langle \overline{s}_{\mathtt{I}}, [c_h] \triangleright \overline{s}_{\mathtt{O}} \rangle) :- \; q_1(\langle \overline{r}_{\mathtt{I}}^{1} \triangleright \overline{r}_{\mathtt{O}}^{1} \rangle), \; \ldots, \; q_n(\langle \overline{r}_{\mathtt{I}}^{n} \triangleright \overline{r}_{\mathtt{O}}^{n} \rangle)$

$C5.3.3^*: \quad h'(\langle \overline{s}_{\mathtt{I}}, [c_h] \triangleright \overline{z} \rangle) :- \; h'(\langle \overline{r}_{\mathtt{I}}^{1}, [\overline{w}^1, c_{q_1}^C] \triangleright \overline{r}_{\mathtt{O}}^{1} \rangle), \; \ldots, \; h'(\langle \overline{r}_{\mathtt{I}}^{n}, [\overline{w}^n, c_{q_n}^C] \triangleright \overline{r}_{\mathtt{O}}^{n} \rangle), \; h'(\langle \overline{r}_{\mathtt{O}}^{n}, [\overline{w}, c_h^C] \triangleright \overline{z} \rangle)$

$C5.3.4^*: \quad h'(\langle \overline{r}_{\mathtt{O}}^{n}, [\overline{w}, c_h^C] \triangleright \overline{s}_{\mathtt{O}} \rangle)$

$C5.3.5^*: \quad h'(\langle \overline{r}_{\mathtt{I}}^{j}, [\overline{w}^j, c_{q_j}^C] \triangleright \overline{r}_{\mathtt{O}}^{j} \rangle) :- \; q_j(\langle \overline{r}_{\mathtt{I}}^{j} \triangleright \overline{r}_{\mathtt{O}}^{j} \rangle)$

where the constants and the sets of variables are obtained as in Definition 5.2, and the clauses that are marked with an asterisk * denote schemes on one or more syntactic objects. In concrete, each one of the previous sets consists of a single clause $C5.3.1$, a clause $C5.3.2^*$ for each non-recursive clause in

$\texttt{MRDef}_P(p)$ and a clause $C5.3.3^*$, a clause $C5.3.4^*$ and n clauses $C5.3.5^*$ for each recursive clause in $\texttt{MRDef}_P(p)$ (being n the number of atoms in the corresponding clause). For technical convenience, we consider that the predicates $h \in \texttt{MR}_P(p)$ (except for p) are renamed in the previous sets of clauses.

The programs P and P_0 are TF-equivalent since by unfolding the $n+1$ body atoms in the clauses of the form $C5.3.3^*$ and then unfolding the atom $h'(\!|\overline{x}, [c_h] \triangleright \overline{z}|\!)$ in each clause of the form $C5.3.1^*$ we obtain P (eliminating the repeated clauses).

Next, we define the program P_1 from P_0 by introducing a new predicate p_tr that is defined by:

$C5.3.6: \quad p\_tr(\!|\overline{x}, [\,] \triangleright \overline{x}|\!)$
$C5.3.7: \quad p\_tr(\!|x_1, [x_2|S] \triangleright \overline{z}|\!) :- \; h'(\!|x_1, [x_2] \triangleright x_3|\!),\; p\_tr(\!|x_3, S \triangleright \overline{z}|\!)$

Trivially $P_0 \cong_{TF} P_1$. Now, we unfold in the clauses of the form $C5.3.5^*$ the atoms $q_j(\!|\overline{r}^j_{\mathtt{I}} \triangleright \overline{r}^j_{\mathtt{0}}|\!)$ such that $q_j \in \texttt{MR}_P(p)$ and then we unfold the atom $h'(\!|x_1, [x_2] \triangleright x_3|\!)$ in the clause $C5.3.7$, obtaining the program P_2. $\texttt{Def}_P(p\_tr)$ and $\texttt{Def}_{P_2}(p\_tr)$ are equal except for the clauses of the form (4) (see Definition 5.2). In the program P_2, the corresponding clauses are of the form:

$$C5.3.8^*: \; p\_tr(\!|\overline{s}_{\mathtt{I}}, [c_h|S] \triangleright \overline{z}|\!) :- \; h'(\!|\overline{r}^1_{\mathtt{I}}, [\overline{w}^1, c^C_{q_1}] \triangleright \overline{r}^1_{\mathtt{0}}|\!),\; \ldots,\; h'(\!|\overline{r}^n_{\mathtt{I}}, [\overline{w}^n, c^C_{q_n}] \triangleright \overline{r}^n_{\mathtt{0}}|\!),\; h'(\!|\overline{r}^n_{\mathtt{0}}, [\overline{w}, c^C_h] \triangleright \overline{y}|\!),\; p\_tr(\!|\overline{y}, S \triangleright \overline{z}|\!)$$

It suffices to fold $n+1$ times the rightmost couple of atoms in these clauses using the clause $C5.3.7$, obtaining the program P_3. At each folding step, we get a new atom of predicate p_tr that is used in the next step. At the end of this process, $\texttt{Def}_P(p\_tr)$ and $\texttt{Def}_{P_3}(p\_tr)$ are equal and, therefore, the obtained program is also identical to $P \setminus \texttt{Def}_P(p) \cup \texttt{TailRDef}_P[p \backslash \texttt{ms}(p(\overline{u}))]$ (eliminating the superfluous predicates).

Finally, it is important to remark that, since P is normalized, the links between the auxiliary variables are preserved in the transformed program. □

Example 5.4 *Given the normalized program P in Example 3.4, the tail recursive definition of preorder w.r.t.* $(\texttt{in}, \texttt{out})$ *consists of*[4] *:*

$E5.4.1: \quad prdr\_tr(\!|z, [\,] \triangleright z|\!)$
$E5.4.2: \quad preorder(\!|x \triangleright z|\!) :- \; prdr\_tr(\!|x, [c_1] \triangleright z|\!)$
$E5.4.3: \quad preorder'(\!|x_1, x_2 \triangleright z|\!) :- \; prdr\_tr(\!|x_1, x_2, [c_2] \triangleright z|\!)$
$E5.4.4: \quad prdr\_tr(\!|nil, [c_1|S] \triangleright z|\!) :- \; prdr\_tr(\!|[\,], S \triangleright z|\!)$
$E5.4.5: \quad prdr\_tr(\!|tree(x_1, x_2, x_3), [c_1|S] \triangleright z|\!) :-$
$\qquad prdr\_tr(\!|tree(x_1, x_2, x_3), [[\,], c_3, [x_3], c_4, [x_1], c_5, [\,], c_6|S] \triangleright z|\!)$
$E5.4.6: \quad prdr\_tr(\!|tree(x_1, x_2, x_3), [[\,], c_3|S] \triangleright z|\!) :- \; prdr\_tr(\!|x_2, [c_1|S] \triangleright z|\!)$

[4] For brevity, we use numbers, instead of predicates names, as constant sub-scripts.

$E5.4.7:\ prdr\_tr(\!|y_1, [[x_3], c_4|S] \triangleright z|\!) :- prdr\_tr(\!|y_1, x_3, [c_2|S] \triangleright z|\!)$
$E5.4.8:\ prdr\_tr(\!|y_3, y_2, [[x_1], c_5|S] \triangleright z|\!) :- append(\!|[x_1|y_3], y_2 \triangleright y|\!),$
$\quad prdr\_tr(\!|y, S \triangleright z|\!)$
$E5.4.9:\ prdr\_tr(\!|x_4, [[\,], c_6|S] \triangleright z|\!) :- prdr\_tr(\!|x_4, S \triangleright z|\!)$
$E5.4.10:\ prdr\_tr(\!|y, x_1, [c_2|S] \triangleright z|\!) :- prdr\_tr(\!|y, x_1, [[\,], c_7, [[y], c_8|S] \triangleright z|\!)$
$E5.4.11:\ prdr\_tr(\!|y, x_1, [[\,], c_7|S] \triangleright z|\!) :- prdr\_tr(\!|x_1, [c_1|S] \triangleright z|\!)$
$E5.4.12:\ prdr\_tr(\!|x_2, [[y], c_8|S] \triangleright z|\!) :- prdr\_tr(\!|y, x_2, S \triangleright z|\!)$ □

Notice that for every program P and after transforming the definition of a predicate p in this way, no other predicate is mutually recursive to p. That is, $\mathtt{MR}_{P'}(p) \equiv \{p\}$ being P' the obtained program.

5.2 A Single Step of Auxiliary Variable Elimination

The second subtask is to eliminate the auxiliary variables that are located in the leftmost atom where some auxiliary variables occur in. With this aim, we have to infer the mode specification in the above mentioned atom and the next one. The Definition 4.4 already fixes the mode specification in the argument positions where auxiliary variables occur in. In the remaining ones, the mode specification that is inferred by Definition 4.5 is subject to the mode specification in the clause head atom, that is undefined for the time being. Therefore, the mode specification in these positions is inferred as follows.

Definition 5.5 *Let $C \equiv p(\overline{t}) :- q_1(\overline{s}^1), \ldots, q_n(\overline{s}^n)$ be a normalized clause such that $\mathtt{ms}(p(\overline{t}))$ is undefined, $q_i(\overline{s}^i)$ be the leftmost atom that contains some auxiliary variables for $1 \leq i \leq n$ and let m_i and m_{i+1} be the arities of the predicates q_i and q_{i+1} respectively, the* mode specification in the j-th argument position of $q_i(\overline{s}^i)$ *for $1 \leq j \leq m_i$, denoted by $\mathtt{ms}_j(q_i(\overline{s}^i))$, such that $AuxVar(s^i_j) \equiv \emptyset$ is* `in`. *In the same way, the* mode specification in the j-th argument position of $q_{i+1}(\overline{s}^{i+1})$ *for $1 \leq j \leq m_{i+1}$, denoted by $\mathtt{ms}_j(q_{i+1}(\overline{s}^{i+1}))$, such that $AuxVar(s^{i+1}_j) \equiv \emptyset$ is* `out`. □

Thus, due to Definitions 4.4 and 5.5, the mode specification in the leftmost atom that contains some auxiliary variables and in the next one is defined.

Example 5.6 *Consider the normalized clause E3.4.3 that is in Example 3.4. $preorder(x_2, y_1)$ is the leftmost atom that contains some auxiliary variables. To be precise, the auxiliary variable y_1. The mode specification in this atom and in the next one, that is the atom $preorder'(x_3, y_2, y_1, y'_1))$, is* (`in`, `out`) *and* (`out`, `out`, `in`, `out`) *respectively.* □

Next, we replace the clause where the auxiliary variables occur in with the set of clauses that is given in the following definition. We proceed in this way on the condition that the definition of the leftmost atom predicate is tail

recursive w.r.t. the inferred mode specification according to Definitions 4.4 and 5.5. Notice that Theorem 5.3 guarantees that this condition holds.

Definition 5.7 *Let* $C \equiv h(\overline{u}) :- \ \overline{M}, p_i(\!|\overline{t}^i_{\mathtt{I}} \triangleright \overline{t}^i_{\mathtt{O}}|\!), p_{i+1}(\!|\overline{t}^{i+1}_{\mathtt{I}} \triangleright \overline{t}^{i+1}_{\mathtt{O}}|\!), \overline{N} \in P$ *be a clause such that* $p_i(\!|\overline{t}^i_{\mathtt{I}} \triangleright \overline{t}^i_{\mathtt{O}}|\!)$ *is the leftmost atom that contains some auxiliary variables and* $\mathtt{Def}_P(q_i)$ *is tail recursive w.r.t.* $p_i(\!|\overline{t}^i_{\mathtt{I}} \triangleright \overline{t}^i_{\mathtt{O}}|\!)$*, the* set of auxiliary variable free clauses of C in P*, denoted by* $AVF_P(C)$*, consists of (where* $\overline{z}$ *is a* m*-tuple of fresh variables and* m *is the number of terms in* $\overline{t}^{i+1}_{\mathtt{O}}$*):*

- *A clause:*

  (1) $h(\overline{u}) :- \ \overline{M}, \ p'(\!|\overline{t}^i_{\mathtt{I}}, \overline{w}_{\mathtt{I}} \triangleright \overline{t}^{i+1}_{\mathtt{O}}, \overline{w}_{\mathtt{O}}|\!), \ \overline{N}$

  where p' *is a new predicate,* $\overline{w}_{\mathtt{I}} \equiv \overline{Aux}\, Var(\overline{t}^i_{\mathtt{O}}) \setminus \overline{Aux}\, Var(\overline{t}^i_{\mathtt{I}} \centerdot \overline{t}^{i+1}_{\mathtt{O}})$ *and* $\overline{w}_{\mathtt{O}} \equiv \overline{Aux}\, Var(\overline{t}^{i+1}_{\mathtt{I}}) \setminus \overline{Aux}\, Var(\overline{t}^i_{\mathtt{I}} \centerdot \overline{t}^{i+1}_{\mathtt{O}})$.
- *For each* $p_i(\!|\overline{r}_{\mathtt{I}} \triangleright \overline{r}_{\mathtt{O}}|\!) :- \ \overline{K} \in \mathtt{Def}_P(p_j)$ *(non-recursive clauses), a clause:*

  (2) $p'(\!|\overline{r}_{\mathtt{I}}\sigma, \overline{w}_{\mathtt{I}}\sigma \triangleright \overline{z}, \overline{w}_{\mathtt{O}}\sigma|\!) :- \ \overline{K}, \ p_{i+1}(\!|\overline{t}^{i+1}_{\mathtt{I}}\sigma \triangleright \overline{z}|\!)$

  where $\sigma \equiv mgu(\overline{r}_{\mathtt{O}}, \overline{t}^i_{\mathtt{O}})$.
- *For each* $p_i(\!|\overline{s}_{\mathtt{I}} \triangleright \overline{z}|\!) :- \ \overline{L}, \ p_i(\!|\overline{s}'_{\mathtt{I}} \triangleright \overline{z}|\!) \in \mathtt{Def}_P(p_i)$ *(recursive clauses), a clause:*

  (3) $p'(\!|\overline{s}_{\mathtt{I}}, \overline{w}_{\mathtt{I}} \triangleright \overline{z}, \overline{w}_{\mathtt{O}}|\!) :- \ \overline{L}, \ p'(\!|\overline{s}'_{\mathtt{I}}, \overline{w}_{\mathtt{I}} \triangleright \overline{z}, \overline{w}_{\mathtt{O}}|\!)$ □

Note that, since the tuples $\overline{t}^i_{\mathtt{O}}$ and $\overline{t}^{i+1}_{\mathtt{I}}$ have disappeared, the involved auxiliary variables have been eliminated. Again, the set of variables $\overline{w}_{\mathtt{I}}$ and $\overline{w}_{\mathtt{O}}$ ensure that the links between atoms through (non-auxiliary) variables are kept.

Theorem 5.8 *Let* $C \equiv h(\overline{u}) :- \ \overline{M}, p_i(\!|\overline{t}^i_{\mathtt{I}} \triangleright \overline{t}^i_{\mathtt{O}}|\!), p_{i+1}(\!|\overline{t}^{i+1}_{\mathtt{I}} \triangleright \overline{t}^{i+1}_{\mathtt{O}}|\!), \overline{N} \in P$ *be a clause such that* $p_i(\!|\overline{t}^i_{\mathtt{I}} \triangleright \overline{t}^i_{\mathtt{O}}|\!)$ *is the leftmost atom that contains some auxiliary variables and* $\mathtt{Def}_P(q_i)$ *is tail recursive w.r.t.* $p_i(\!|\overline{t}^i_{\mathtt{I}} \triangleright \overline{t}^i_{\mathtt{O}}|\!)$*. The programs* P *and* $P \setminus \{C\} \cup AVF_P(C)$ *are TF-equivalent.*

Proof. Starting from the program P, we obtain P_0 by introducing a new predicate p' that is defined by the single clause:

$$C5.8.1: \quad p'(\!|\overline{x}, \overline{w}_{\mathtt{I}} \triangleright \overline{z}, \overline{w}_{\mathtt{O}}|\!) :- \ p_i(\!|\overline{x} \triangleright \overline{t}^i_{\mathtt{O}}|\!), \ p_{i+1}(\!|\overline{t}^{i+1}_{\mathtt{I}} \triangleright \overline{z}|\!)$$

where:

- $\overline{x}$ is a m-tuple of fresh variables and m is the number of terms in $\overline{t}^i_{\mathtt{I}}$
- $\overline{z}$ is a n-tuple of fresh variables and n is the number of terms in $\overline{t}^{i+1}_{\mathtt{O}}$
- the sets of variables $\overline{w}_{\mathtt{I}}$ and $\overline{w}_{\mathtt{O}}$ are obtained as in Definition 5.7
- the mode specifications in $p_i(\!|\overline{x} \triangleright \overline{t}^i_{\mathtt{O}}|\!)$ and $p_{i+1}(\!|\overline{t}^{i+1}_{\mathtt{I}} \triangleright \overline{z}|\!)$ coincide with the ones in the atoms $p_i(\!|\overline{t}^i_{\mathtt{I}} \triangleright \overline{t}^i_{\mathtt{O}}|\!)$ and $p_{i+1}(\!|\overline{t}^{i+1}_{\mathtt{I}} \triangleright \overline{t}^{i+1}_{\mathtt{O}}|\!)$ of the clause C.

The programs P and P_0 are trivially TF-equivalent. Since $\mathtt{Def}_P(p_i)$ (and therefore, $\mathtt{Def}_{P_0}(p_i)$) is tail recursive w.r.t. $p_i(\overline{x} \triangleright \overline{t}_{\mathtt{O}}^i)$, by unfolding this atom in the clause $C5.8.1$ we get clauses of the following two forms[5]:

$$C5.8.2^* : \ p'(\overline{r}_{\mathtt{I}}\sigma, \overline{w}_{\mathtt{I}}\sigma \triangleright \overline{z}, \overline{w}_{\mathtt{O}}\sigma) \ :- \ \overline{K}, \ p_{i+1}(\overline{t}_{\mathtt{I}}^{i+1}\sigma \triangleright \overline{z})$$
$$C5.8.3^* : \ p'(\overline{s}_{\mathtt{I}}, \overline{w}_{\mathtt{I}} \triangleright \overline{z}, \overline{w}_{\mathtt{O}}) \ :- \ \overline{L}, \ p_i(\overline{s}'_{\mathtt{I}} \triangleright \overline{t}_{\mathtt{O}}^i), \ p_{i+1}(\overline{t}_{\mathtt{I}}^{i+1}, \overline{w}_{\mathtt{I}} \triangleright \overline{z}, \overline{w}_{\mathtt{O}})$$

where $\sigma \equiv mgu(\overline{r}_{\mathtt{O}}, \overline{t}_{\mathtt{O}}^i)$. We now fold each clause of the form $C5.8.3^*$ using $C5.8.1$, obtaining clauses of the following form:

$$C5.8.4^* : \ p'(\overline{s}_{\mathtt{I}}, \overline{w}_{\mathtt{I}} \triangleright \overline{z}, \overline{w}_{\mathtt{O}}) \ :- \ \overline{L}, \ p'(\overline{s}'_{\mathtt{I}}, \overline{w}_{\mathtt{I}} \triangleright \overline{z}, \overline{w}_{\mathtt{O}})$$

By folding the clause C using the clause $C5.8.1$, we get:

$$C5.8.5 : \ h(\overline{u}) \ :- \ \overline{M}, \ p'(\overline{t}_{\mathtt{I}}^i, \overline{w}_{\mathtt{I}} \triangleright \overline{t}_{\mathtt{O}}^{i+1}, \overline{w}_{\mathtt{O}}), \ \overline{N}$$

The resulting program P_1 is defined by the set of clauses:

$$P_1 \equiv (P_0 \setminus \{C, C5.8.1\}) \cup \{C5.8.5\} \cup C5.8.2^* \cup C5.8.4^*$$

considering $C5.8.2^*$ and $C5.8.4^*$ to be sets of clauses. Since the programs P_1 and $P \setminus \{C\} \cup AVF_P(C)$ are syntactically equal, P and $P \setminus \{C\} \cup AVF_P(C)$ are TF-equivalent. □

Example 5.9 *Let P be the program in Example 5.4, it only remains to eliminate the auxiliary variable y from the clause $E5.4.8$:*

$$prdr\_tr(y_3, y_2, [[x_1], c_5|S], z) \ :- \ append([x_1|y_3], y_2, y), \ prdr\_tr(y, S, z)$$

According to Definitions 4.4 and 5.5, the mode specification in the atoms that share the variable y is $append([x_1|y_3], y_2 \triangleright y)$ and $prdr\_tr(y \triangleright S, z)$ respectively. Let $\mathtt{Def}_P(append)$ consist of (representing the mode specification $append([x_1|y_3], y_2 \triangleright y)$):

$$E5.9.1 : \ append([\,], x \triangleright x)$$
$$E5.9.2 : \ append([x_1|x_2], x_3 \triangleright [x_1|x_4]) \ :- \ append(x_2, x_3 \triangleright x_4)$$

Since $\mathtt{Def}_P(append)$ is not tail recursive w.r.t. $append([x_1|y_3], y_2 \triangleright y)$, we substitute $\mathtt{TailRDef}_P[append \backslash append([x_1|y_3], y_2 \triangleright y)]$ for $\mathtt{Def}_P(append)$:

$$E5.9.3 : \ app\_tr(z, [\,] \triangleright z)$$
$$E5.9.4 : \ append(x_1, x_2 \triangleright z) \ :- \ app\_tr(x_1, x_2, [c_9] \triangleright z)$$
$$E5.9.5 : \ app\_tr([\,], x, [c_9|S] \triangleright z) \ :- \ app\_tr(x, S \triangleright z)$$
$$E5.9.6 : \ app\_tr([x_1|x_2], x_3, [c_9|S] \triangleright z) \ :- \ app\_tr([x_1|x_2], x_3, [[\,], c_{10}, [x_1], c_{11}|S] \triangleright z)$$
$$E5.9.7 : \ app\_tr([x_1|x_2], x_3, [[\,], c_{10}|S] \triangleright z) \ :- \ app\_tr(x_2, x_3, [c_9|S] \triangleright z)$$
$$E5.9.8 : \ app\_tr(x_4, [[x_1], c_{11}|S] \triangleright z) \ :- \ app\_tr([x_1|x_4], S \triangleright z)$$

[5] As before, the asterisk is used to denote clause schemes.

As it is explained in the next subsection, we substitute $app\_tr([x_1|y_3], y_2, [c_9], y)$ *for* $append([x_1|y_3], y_2, y)$ *in the clause E5.4.8 and we get:*

$$E5.5.9: \quad prdr\_tr(y_3, y_2, [[x_1], c_5|S], z) :- \ app\_tr(\langle\!|[x_1|y_3], y_2, [c_9] \triangleright y|\!\rangle), \\ prdr\_tr(\langle\!|y \triangleright S, z|\!\rangle)$$

Finally, $AVF_P(E5.5.9)$ *consists of:*

$$E5.5.10: \ prdr\_tr(y_3, y_2, [[x_1], c_5|S], z) :- \ app\_tr'(\langle\!|[x_1|y_3], y_2, [c_9] \triangleright S, z|\!\rangle)$$
$$E5.5.11: \ app\_tr'(\langle\!|x, [\,] \triangleright S, z|\!\rangle) :- \ p\_tr(\langle\!|x \triangleright S, z|\!\rangle)$$
$$E5.5.12: \ app\_tr'(\langle\!|[\,], x, [c_9|S] \triangleright z_1, z_2|\!\rangle) :- \ app\_tr'(\langle\!|x, S \triangleright z_1, z_2|\!\rangle)$$
$$E5.5.13: \ app\_tr'(\langle\!|[x_1|x_2], x_3, [c_9|S] \triangleright z_1, z_2|\!\rangle) :- \\ app\_tr'(\langle\!|[x_1|x_2], x_3, [[\,], c_{10}, [x_1], c_{11}|S] \triangleright z_1, z_2|\!\rangle)$$
$$E5.5.14: \ app\_tr'(\langle\!|[x_1|x_2], x_3, [[\,], c_{10}|S] \triangleright z_1, z_2|\!\rangle) :- \\ app\_tr'(\langle\!|x_2, x_3, [c_9|S] \triangleright z_1, z_2|\!\rangle)$$
$$E5.5.15: \ app\_tr'(\langle\!|x_4, [[x_1], c_{11}|S] \triangleright z_1, z_2|\!\rangle) :- \ app\_tr'(\langle\!|[x_1|x_4], S \triangleright z_1, z_2|\!\rangle) \ \square$$

5.3 An Auxiliary Variable Elimination Algorithm

Making use of the previous two transformations, we give an algorithm for eliminating the auxiliary variables from a definite logic program. Remember that normalization is always assumed.

Roughly speaking, the algorithm in Figure 1 works as follows. For each clause $C \equiv h(\overline{u}) :- \overline{M}, p_i(\langle\!|\overline{t}^i_{\mathtt{I}} \triangleright \overline{t}^i_{\mathtt{O}}|\!\rangle), p_{i+1}(\langle\!|\overline{t}^{i+1}_{\mathtt{I}} \triangleright \overline{t}^{i+1}_{\mathtt{O}}|\!\rangle), \overline{N}$, we select the leftmost atom that contains some auxiliary variables and the next one. The mode specification in these atoms, $p_i(\langle\!|\overline{t}^i_{\mathtt{I}} \triangleright \overline{t}^i_{\mathtt{O}}|\!\rangle)$ and $p_{i+1}(\langle\!|\overline{t}^{i+1}_{\mathtt{I}} \triangleright \overline{t}^{i+1}_{\mathtt{O}}|\!\rangle)$, is inferred according to Definitions 4.4 and 5.5. Besides, the mode specification in $p_i(\langle\!|\overline{t}^i_{\mathtt{I}} \triangleright \overline{t}^i_{\mathtt{O}}|\!\rangle)$ is extended to $\mathtt{MRDef}_P(p_i)$ according to Definitions 4.4, 4.6, 4.7 and 4.8 when necessary. Then, there are two main cases:

- If p_i does not depend on h (line 5), we substitute the set of clauses $AVF_P(C)$ (see Definition 5.7) for the clause C. If necessary (line 6), we transform $\mathtt{Def}_P(p_i)$ into $\mathtt{TailRDef}_P[p_i \backslash p_i(\langle\!|\overline{t}^i_{\mathtt{I}} \triangleright \overline{t}^i_{\mathtt{O}}|\!\rangle)]$ (see Definition 5.2) and substitute $p_i\_tr(\langle\!|\overline{t}^i_{\mathtt{I}}, [c_{p_i}] \triangleright \overline{t}^i_{\mathtt{O}}|\!\rangle)$ for $p_i(\langle\!|\overline{t}^i_{\mathtt{I}} \triangleright \overline{t}^i_{\mathtt{O}}|\!\rangle)$ in the clause C, where p_i_tr is the new predicate that is introduced by the transformation.
- If p_i depends on h (line 12), $\mathtt{Def}_P(p_i)$ is not tail recursive w.r.t. $p_i(\langle\!|\overline{t}^i_{\mathtt{I}} \triangleright \overline{t}^i_{\mathtt{O}}|\!\rangle)$ because recursion is not restricted to the rightmost atom. Since $\mathtt{ms}(h(\overline{u}))$ is defined after extending $p_i(\langle\!|\overline{t}^i_{\mathtt{I}} \triangleright \overline{t}^i_{\mathtt{O}}|\!\rangle)$ to $\mathtt{MRDef}_P(p_i)$, we transform $\mathtt{Def}_P(h)$ into $\mathtt{TailRDef}_P[h \backslash \mathtt{ms}(h(\overline{u}))]$. This transformation eliminates the clause C from the program P. Therefore, a different clause is selected in the next step.

In the former case ($p_i \notin \mathtt{MR}_P(h)$), the involved auxiliary variables are eliminated by substituting $AVF_P(C)$ for the clause C. In the latter case

PRE: { P_0 is a normalized program and $P \equiv P_0$ }

1 **repeat**

2 select any clause $C \equiv h(\overline{u}) :- \overline{M}, p_i(\!|\overline{t}^i_{\mathtt{I}} \triangleright \overline{t}^i_{\mathtt{O}}|\!), p_{i+1}(\!|\overline{t}^{i+1}_{\mathtt{I}} \triangleright \overline{t}^{i+1}_{\mathtt{O}}|\!), \overline{N}$

3 such that $p_i(\!|\overline{t}^i_{\mathtt{I}} \triangleright \overline{t}^i_{\mathtt{O}}|\!)$ is the leftmost atom where some

4 auxiliary variable occurs in

5 **if** $p_i \notin \mathtt{MR}_P(h)$ **then**

6 **if** $\mathtt{Def}_P(p_i)$ is not tail recursive w.r.t. $p_i(\!|\overline{t}^i_{\mathtt{I}} \triangleright \overline{t}^i_{\mathtt{O}}|\!)$ **then**

7 transform $\mathtt{Def}_P(p_i)$ into $\mathtt{TailRDef}_P[p_i \backslash \mathtt{ms}(p_i(\overline{t}))]$

8 substitute $p_i\_tr(\!|\overline{t}^i_{\mathtt{I}}, [c_{p_i}] \triangleright \overline{t}^i_{\mathtt{O}}|\!)$ for $p_i(\!|t^i_{\mathtt{I}} \triangleright t^i_{\mathtt{O}}|\!)$ in C where

9 p_i_tr is the new introduced predicate

10 **end if**

11 substitute $AVF_P(C)$ for the clause C

12 **else**

13 transform $\mathtt{MRDef}_P(h)$ into $\mathtt{TailRDef}_P[h \backslash \mathtt{ms}(h(\overline{u}))]$

14 **end if**

15 **until** no auxiliary variable remains in P

POST: { P is an auxiliary variable free program and $P \cong_{TF} P_0$ }

Fig. 1. An Auxiliary Variable Elimination Algorithm

($p_i \in \mathtt{MR}_P(h)$), the transformation of $\mathtt{Def}_P(h)$ into $\mathtt{TailRDef}_P[h \backslash \mathtt{ms}(h(\overline{u}))]$ ensures that recursion is restricted to the rightmost most in every clause $C' \in \mathtt{MRDef}_P(h)$. Therefore, this transformation is performed at most once for each predicate in the program and, after that, auxiliary variables are directly eliminated according to the first case.

It is important to stress that the transformation to tail recursive itself often eliminates many of the auxiliary variables (see Example 5.4 where two auxiliary variables are eliminated). In addition, after the tail recursive transformation of h, when a clause $D \notin \mathtt{Def}_P(h)$ is selected, the clauses in $\mathtt{Def}_P(h)$ can never be affected by the auxiliary variable elimination process.

Although no transformational step introduces new auxiliary variables, unfortunately we do not have a formal termination proof. Hence, further inves-

tigations must be done to achieve a total correctness proof for the presented algorithm.

6 Conclusions

The presented method can be automatically applied to every definite logic program and eliminates its auxiliary (local) variables. Each definite clause (with auxiliary variables) is replaced with a set of definite clauses with new predicates. As a result, the negative version of the target program does not require universal quantification. Hence, much efficiency is gained in negative goal computation. However, positive goals should be computed with respect to the source program since, in general, the new predicates that are introduced by the transformation reduce the efficiency of the positive goal computation. The aim of the present work is to prove the existence of a general algorithm for auxiliary variable elimination in definite logic programs. Much work and further improvements should be made on implementation and experimentation in order to obtain more efficient target programs.

A method for eliminating local (there called *unnecessary*) variables from definite logic programs was introduced in [13]. Their main aim was to eliminate the redundant computations that are made by means of local variables. Hence, the target program yields more efficient SLD-computations. This motivation is essentially different from ours. They present different strategies for guiding the application of unfold/fold transformations in order to achieve local variable elimination. The strategies are syntactically based and only guarantee the complete elimination of local variables for a very restricted subclass of definite logic programs. To the best of our knowledge, there is no other published result on the elimination of this kind of variables in logic programs.

We plan to extend our results to normal logic programs. By now, we think that full generality could not be achieved in this case. However, we believe that the method can be easily adapted for a wide subclass of normal logic programs. Future work also includes the extension to constraint logic programming.

Acknowledgement

We are very grateful for the helpful comments we receive from our referees.

References

[1] Álvez, J., P. Lucio, F. Orejas, E. Pasarella and E. Pino, *Constructive negation by bottom-up computation of literal answers*, in: *SAC '04: Proceedings of the 2004 ACM Symposium on Applied Computing*, 2004, pp. 1468–1475.

[2] Barbuti, M., P. Mancarella, D. Pedreschi and F. Turini, *A transformational approach to negation in logic programming*, Journal of Logic Programming **8** (1990), pp. 201–228.

[3] Bruscoli, P., F. Levi, G. Levi and M. C. Meo, *Compilative constructive negation in constraint logic programs*, in: S. Tison, editor, *Proc. of the Trees in Algebra and Programming 19th Int. Coll.(CAAP '94)*, LNCS **787** (1994), pp. 52–67.

[4] Chan, D., *Constructive negation based on the completed database*, in: R. A. Kowalski and K. A. Bowen, editors, *Proc. of the 5th Int. Conf. and Symp. on Logic Progr.* (1988), pp. 111–125.

[5] Chan, D., *An extension of constructive negation and its application in coroutining*, in: E. Lusk and R. Overbeek, editors, *Proc. of the NACLP'89* (1989), pp. 477–493.

[6] Clark, K. L., *Negation as failure*, in: H. Gallaire and J. Minker, editors, *Logic and Databases* (1978), pp. 293–322.

[7] Debray, S. K. and D. S. Warren, *Automatic mode inference for logic programs*, Journal of Logic Programming **5** (1988), pp. 207–229.

[8] Drabent, W., *What is failure? an approach to constructive negation*, Acta Informatica **32** (1995), pp. 27–59.

[9] Fitting, M., *A Kripke-Kleene semantics for logic programs*, Journal of Logic Programming **2** (1985), pp. 295–312.

[10] King, A., K. Shen and F. Benoy, *Lower-bound time-complexity analysis of logic programs*, in: J. Maluszynski, editor, *International Symposium on Logic Programming* (1997), pp. 261 – 276.

[11] Kunen, K., *Negation in logic programming*, Journal of Logic Programming **4** (1987), pp. 289–308.

[12] Maher, M. J., *Equivalences of logic programs*, in: J. Minker, editor, *Foundations of deductive databases and logic programming* (1988), pp. 627–658.

[13] Proietti, M. and A. Pettorossi, *Unfolding - definition - folding, in this order, for avoiding unnecessary variables in logic programs*, Theoretical Computer Science **142** (1995), pp. 89–124.

[14] Sato, T. and H. Tamaki, *Transformational logic program synthesis*, in: *Proceedings of International Conference on Fifth Generation Computer Systems*, 1984, pp. 195–201.

[15] Shepherdson, J., *Language and equality theory in logic programming*, Technical Report No. PM-91-02, University of Bristol (1991).

[16] Stuckey, P. J., *Negation and constraint logic programming*, Information and Computation **118** (1995), pp. 12–33.