An Algorithm for Local Variable Elimination in Normal Logic Programs

Javier Álvez and Paqui Lucio

Faculty of Computer Science, Basque Country University, San Sebastián, Spain.
{jibalgij,jiplucap}@si.ehu.es

Abstract. A variable is *local* if it occurs in a clause body but not in its head. Local variables appear naturally in practical logic programming, but they complicate several aspects such as negation, compilation, memoization, static analysis, program approximation by neural networks etc. As a consequence, the absence of local variables yields better performance of several tools and is a prerequisite for many technical results. In this paper, we introduce an algorithm that eliminates local variables from a wide proper subclass of normal logic programs. The proposed transformation preserves the Clark-Kunen semantics for normal logic programs.

1 Introduction

Local variables are very often used —as auxiliaries— to store intermediate results in logic programs. Their values are passed from one atom to another in a clause body, but they are not lifted to the head. Whilst they are useful in practical logic programming, the occurrence of local variables could cause inefficiency or even prevent the satisfaction of some properties. In the area of *negation in logic programming*, several results are restricted to *local variable free* (*lvf*, for short) programs or are less efficiently applicable when local variables are present. For instance, local variables lead to *floundering* problems in *negation as failure* [8]. In addition, their presence prevents completeness results for more recent techniques like *intensional negation* [5] and other transformational negation techniques [20, 25]. In several computational mechanisms proposed for *constructive negation* [7, 6, 11, 23, 2], local variables force one to deal with universal quantification, which is not easy to compute in an efficient manner. Other logic programming areas where local variables lead to technical problems are related to compilation, memoization, static analysis, program approximation by neural networks etc. Moreover, in equational logic programming, local variables cause problems since, in their presence, narrowing may become incomplete. In [12], some transformations of definite (equational) logic programs into *lvf* ones are presented. Unfortunately, these methods do not preserve failure.

It is well known that every computable function can be computed by a definite logic program [3, 24]. Besides, the program built in [24] is an *lvf* definite program. Therefore, from the theoretical point of view, the function that is computed by any given normal logic program (according to the Clark-Kunen semantics) can

also be computed by an *lvf* definite program (in PROLOG). The question is how to automatically generate an *lvf* program from any given program with local variables. In particular, the *lvf* program given in [24] simulates the computation of a universal Turing machine over the Turing machine that computes the recursive function. This is not the program that we try to generate by transformation. Besides, each *lvf* normal program can be transformed into a definite one (see, for instance, [5]). Hence, the automatic removing of local variables from normal programs is a plausible and encouraging goal.

In this paper, we present an algorithm for eliminating the local variables from normal logic programs while preserving the Clark-Kunen semantics. This method is applicable, in particular, to the class of well-moded logic programs ([9, 10, 19]). We explain the algorithm, prove its correctness and give illustrative examples.

Outline of the paper. In the next section, we fix notation and terminology. In Section 3, we present a preliminary adjustment of the variable occurrences. Section 4 is devoted to partition the arguments of literals depending on the local variables, called mode, and to the notion of tail recursion w.r.t. a mode. In Section 5, we explain how to eliminate the local variables of a literal and the necessary conditions for doing this. The transformation to tail recursion w.r.t. a mode is shown in Section 6. In Section 7, we discuss the algorithm and its termination. Finally, we summarize conclusions and briefly discuss related work.

2 Preliminaries

We assume that the reader is familiar with the basic concepts and notation of logic programming, see e.g. [4]. In this section, we recall some basic terminology and introduce some notational conventions used throughout the paper.

A bar is used to denote tuples, or finite sequences, of objects. For example, $\overline{x}$ denotes an n-tuple of variables $x_1, \ldots, x_n$. Throughout the paper, tuples of variables are assumed to be pairwise distinct. As a consequence, we treat them as sets and use the constant $\emptyset$ and the operators $\setminus$ (set difference), $\cup$ and $\cap$ for tuples of variables with their usual meaning over sets. However, tuples of terms and tuples of literals may have repeated elements. In such cases, concatenation of tuples is denoted by the infix operator $\cdot$ and $\langle\,\rangle$ stands for the empty tuple.

In clauses, we split the variables into *global* and *local*. A variable is *local* in a clause if it occurs in its body but not in its head. Otherwise, it is *global*. For α which is a clause —or any syntactic object inside a clause like atom, term, etc— we denote by $\mathsf{LVar}(\alpha)$ ($\mathsf{GVar}(\alpha)$) the set of *local* (*global*) variables in α. An *lvf* clause/program is a clause/program free of local variables.

A literal is either an atom $p(\overline{t})$ (positive literal) or a negated atom $\neg p(\overline{t})$ (negative literal), where p and $\neg p$ are the *predicate (symbol)* of the literal and $\overline{t}$ is a tuple of terms. We usually denote literals by $L(\overline{t})$, $M(\overline{t})$, $N(\overline{t})$, ... and L, M, N, ... are used to denote the predicate of a literal. In a literal, when we do not need to specify the arguments, we simply omit the tuple of terms, and then

L, M, N, ... denote a literal. Throughout the paper, the context always makes it clear whether a capital letter stands for a literal or a predicate symbol.

Our program transformations preserve the Clark-Kunen semantics of normal programs, which is given by Clark's program completion [8] interpreted in three-valued logic [13]. Note that the logical Clark-Kunen semantics is independent of the order in which the literals appear in the clause bodies.

Given a normal program P and a predicate symbol p, $\mathtt{Def}_P(p)$ is the set of all clauses from P whose head literal begins with p. From Clark's completion, we have that $\neg p(\overline{x}) \leftrightarrow \neg\varphi$ where φ is the disjunction of the body clauses, which are existentially quantified on its local variables. If $\mathtt{Def}_P(p)$ has no local variables, then $\neg p(\overline{x}) \leftrightarrow \neg\varphi$ can be easily reduced to a set of clauses that have a normal body and a negative head.[1] In this case, we say that this is a *normal definition* called $\mathtt{Def}_P(\neg p)$. Otherwise, the universal quantification in clauses with head $\neg p(\overline{x})$ can not be avoided and we say that $\mathtt{Def}_P(\neg p)$ is a *complex definition*. In our transformations, only normal definitions are handled. Complex definitions are delayed until they become normal or, otherwise, they are delayed forever.

We make use of the fold/unfold transformation system described in [17] (see also [15]) where equivalence w.r.t. the Clark-Kunen semantics for definite programs is ensured by forbidding unfolding on direct recursion and requiring that a clause C should be folded with another clause, different from C, taken from the current program. Besides, this result can be extended to normal programs with the proviso that only normal (not complex) definitions of negated predicates are used. However, the fold/unfold technique is not powerful enough to prove some of our results and we complement it with direct induction on the bottom-up computation of the Clark-Kunen semantics.

Given a program P and two predicates L and M, we say that M *directly depends on* L if L occurs in $\mathtt{Def}_P(M)$. By the reflexive transitive closure of this relation, we obtain the set $\mathsf{Dpd}_P(L)$ of all predicates on which L depends. Besides, we define the set of all mutually recursive predicates with L by:

$$\mathsf{MR}_P(L) = \{M \mid M \in \mathsf{Dpd}_P(L) \text{ and } L \in \mathsf{Dpd}_P(M)\}.$$

The induced equivalence relation —given by $(L, M) \in R \iff L \in \mathsf{MR}_P(M)$— partitions the set of all predicates into a finite number of equivalence classes. Classes are $\mathsf{MR}_P(L)$ where L is the class representative predicate. Then, the set consisting of all the equivalence classes (or factor set) can be partially ordered by the following relation:

$$\mathsf{MR}_P(L) \preceq \mathsf{MR}_P(M) \iff \begin{array}{l}\text{There exists } N \in \mathsf{MR}_P(L) \text{ and } Q \in \mathsf{MR}_P(M)\\ \text{such that } N \in \mathsf{Dpd}_P(Q).\end{array}$$

Moreover, given a program P, its set of mutually recursive classes does not contain any infinite decreasing chain with respect to $\preceq$. That is, $\preceq$ is a well-founded order. For technical reasons, we need to distinguish the body literals

[1] Of course, disequality is needed.

that are mutually recursive with the clause head. Hence, when convenient, a clause will be written in the following Dpd-*form*:

$$H \leftarrow \overline{L}^1,\ K_1,\ \overline{L}^2,\ \ldots\ ,\ \overline{L}^n,\ K_n,\ \overline{L}^{n+1}$$

to denote that $K_i \in \mathsf{MR}_P(H)$ for each $1 \leq i \leq n$ and $M \notin \mathsf{MR}_P(H)$ for each $M \in \overline{L}^i$ and each $1 \leq i \leq n+1$. Note that $n = 0$ when no body literal belongs to $\mathsf{MR}_P(H)$. Besides, every $\overline{L}^i$ could be an empty tuple.

In what follows, we assume a program, always called P, that is being transformed into an *lvf* program. We would like to emphasize that, although our method chooses some ordering on the clause bodies and the target programs depend on that ordering, we always preserve the Clark-Kunen semantics.

3 Local-Regulation

Local-regulation (LR, for short) is a preliminary treatment of the variable occurrences in the clause bodies. Roughly speaking, it is an adjustment for enabling local variable elimination. First, we define some syntactic conditions, called term-apartness, and show that it can be achieved by program transformation.

Definition 1. *Let $L(\overline{t})$ be an n-ary literal in a given clause $H \leftarrow \overline{M},\ L(\overline{t}),\ \overline{N}$ and $\overline{x} \subseteq \mathsf{Var}(L(\overline{t}))$. The* variables $\overline{x}$ are term-apart in $L(\overline{t})$ *iff for every $1 \leq i \leq n$ (at least) one of the following two conditions holds:*

(a) $\mathsf{Var}(t_i) \cap \overline{x} \cap \mathsf{Var}(\overline{M}) = \emptyset$
(b) $\mathsf{Var}(t_i) \cap \overline{x} \cap \mathsf{Var}(\overline{N}) = \emptyset$. □

For example, given the clause $h(x) \leftarrow\ p(x),\ q(x, f(x,y), y),\ \neg\ h(y)$, the variables (x, y) are not term-apart in $q(x, f(x,y), y)$ due to the term $f(x,y)$ in the second argument, but each variable individually is term-apart in that literal. The intuition behind term-apartness is that it allows us to partition the terms $\overline{t}$ of a literal $L(\overline{t})$ in two tuples $\overline{t}_1$ and $\overline{t}_2$ such that the variables that occur in $\overline{t}_1$ ($\overline{t}_2$) only occur in the left-hand side literals $\overline{M}$ (right-hand side literals $\overline{N}$). This partition facilitates the dataflow analysis of the clause.

Lemma 1. *Every normal program P can be transformed into a Clark-Kunen equivalent normal program P' such that all its variables are term-apart in the literals occurring in P'.*

Proof. Let $\overline{x} \subseteq \mathsf{Var}(L(\overline{t}))$ in a clause $C = H \leftarrow\ \overline{M},\ L(\overline{t}),\ \overline{N} \in P$ such that $\overline{w} = \mathsf{Var}(t_i) \cap \overline{x} \cap \mathsf{Var}(\overline{M}) \neq \emptyset$ and $\mathsf{Var}(t_i) \cap \overline{x} \cap \mathsf{Var}(\overline{N}) \neq \emptyset$ for some i. We define a new predicate p by $D = p(\overline{z} \cdot \overline{w} \cdot \overline{w}) \leftarrow\ L(\overline{z})$ and substitute the clause $C' = H \leftarrow\ \overline{M},\ p(\overline{t}' \cdot \overline{w} \cdot \overline{w}'),\ \overline{N}[\overline{w}'/\overline{w}]$ for C, where $\overline{w}'$ is a tuple of fresh variables and $\overline{t}'$ is obtained by replacing t_i with $t_i[\overline{w}'/\overline{w}]$ in $\overline{t}$. In the resulting program, D is an *lvf* clause since only a literal occurs in its body and, besides, we have that $\mathsf{Var}(t_i') \cap \overline{x} \cap \mathsf{Var}(\overline{M}) = \emptyset$ in the clause C'. Furthermore, it is easy to see that the new literal $p(\overline{t}' \cdot \overline{w} \cdot \overline{w}')$ in C' will not be affected by any subsequent transformation. After this process, equivalence is preserved since we obtain the program P by unfolding the new literal $p(\overline{t}' \cdot \overline{w} \cdot \overline{w}')$ in C' with the clause D. □

Now, we formulate the condition of local-regularity on clauses and extend it to programs in an obvious way.

Definition 2. *A clause C:*

$$H \leftarrow \overline{L}^1(\overline{r}^1),\ K_1(\overline{s}^1),\ \overline{L}^2(\overline{r}^2),\ \ldots\ ,\ \overline{L}^n(\overline{r}^n),\ K_n(\overline{s}^n),\ \overline{L}^{n+1}(\overline{r}^{n+1})$$

in Dpd*-form is* local-regular *iff $n = 0$ or it satisfies the following two conditions:*

(a) $\mathsf{LVar}(K_i(\overline{s}^i))$ *are term-apart in* $K_i(\overline{s}^i)$ *for every* $1 \leq i \leq n$
(b) every local variable occurs in $\overline{s}^{i-1} \cdot \overline{r}^i \cdot \overline{s}^i$ *for some* $1 \leq i \leq n+1$ *and does not occur anywhere in the clause (by convention,* $\overline{s}^0 = \overline{s}^{n+1} = \langle\,\rangle$*).*

A program P is local-regular *iff every clause $C \in P$ is local-regular.* □

Next, we show a method to transform any program into a local-regular one.

Lemma 2. *Every normal program P can be transformed into a Clark-Kunen equivalent local-regular normal program P'.*

Proof. Due to Lemma 1, we can always transform P into an equivalent program that satisfies the first condition. Therefore, we only focus on the second condition. Let $C = H \leftarrow \overline{L}^1(\overline{r}^1),\ K_1(\overline{s}^1),\ \ldots\ ,\ K_n(\overline{s}^n),\ \overline{L}^{n+1}(\overline{r}^{n+1}) \in P$ be a clause in Dpd-form such that the leftmost occurrence of a local variable y that violates the second condition is either in $K_{i-1}(\overline{s}^{i-1})$ for $2 \leq i \leq n$ or in $\overline{L}^i(\overline{r}^i)$ for $1 \leq i \leq n$ and let C be rewritten as $H \leftarrow \overline{M},\ \overline{L}^i(\overline{r}^i),\ K_i(\overline{s}^i),\ \overline{N}$. Then, we replace C with:

$$C' = H \leftarrow \overline{M},\ \overline{L}^i(\overline{r}^i),\ p_i(\overline{s}^i \cdot y \cdot y'),\ \overline{N}[y'/y]$$

where y' is a new local variable and p_i is a new predicate defined by the clause $D = p_i(\overline{z} \cdot x \cdot x) \leftarrow K_i(\overline{z})$. Now, y satisfies the second condition since it exactly occurs in the tuple of literals $K_{i-1}(\overline{s}^{i-1}) \cdot \overline{L}^i(\overline{r}^i) \cdot p_i(\overline{s}^i \cdot y \cdot y')$ in C'. The iteration of this process ends because either the number of local variables that violate the condition decreases (that is, y' satisfies it) or y' occurs closer to the end of the clause body in C' than y in C: the leftmost occurrence of y is either in $K_{i-1}(\overline{s}^{i-1})$ or in $\overline{L}^i(\overline{r}^i)$ in the clause C whereas the leftmost occurrence of y' is in $p_i(\overline{s}^i \cdot y \cdot y')$ in the clause C'. Also, the new predicate p_i is mutually recursive to H and, therefore, the clause C' is in Dpd-form. Besides, it is easy to see that the first condition of local regulation is always preserved by the above transformation. We have that the resulting program P' is equivalent to P because we obtain P by unfolding the new literals in P'. □

The following example describes the transformation of a one-clause program into a local-regular program consisting of three clauses and using two fresh predicates.

Example 1. Let us consider the following clause of some program P:

$$E1.1:\ \ p(f(x_1, x_2)) \leftarrow q_1(f(x_2, w_1)),\ p_1(g(w_1, w_2)),\ q_2(f(w_2, w_1)),\ p_2(g(w_3, x_1)),\ q_3(f(w_2, w_3))$$

such that $\mathsf{MR}_P(p) = \{p,\ p_1,\ p_2\}$. The clause is not local-regular because the local variable w_1 violates both conditions of Def. 2 and w_2 violates the second one. Considering w_1, we define a new predicate p'_1 and replace $E1.1$ with the following clauses:

$$\begin{array}{ll} E1.2: & p'_1(z,w,w) \leftarrow p_1(z) \\ E1.3: & p(f(x_1,x_2)) \leftarrow q_1(f(x_2,w_1)),\ p'_1(g(w'_1,w_2),w_1,w'_1), \\ & \qquad\qquad q_2(f(w_2,w'_1)),\ p_2(g(w_3,x_1)),\ q_3(f(w_2,w_3)) \end{array}$$

Now, w_1 and w'_1 violate neither the first condition nor the second one. With regard to w_2, its leftmost occurrence is in the literal $p'_1(g(w'_1,w_2),w_1,w'_1)$. Therefore, we define a new predicate p'_2 and substitute the following clauses for $E1.3$:

$$\begin{array}{ll} E1.4: & p'_2(z,w,w) \leftarrow p_2(z) \\ E1.5: & p(f(x_1,x_2)) \leftarrow q_1(f(x_2,w_1)),\ p'_1(g(w'_1,w_2),w_1,w'_1), \\ & \qquad\qquad q_2(f(w_2,w'_1)),\ p'_2(g(w_3,x_1),w_2,w'_2),\ q_3(f(w'_2,w_3)) \end{array}$$

All the resulting clauses are local-regular. Thus, the source and target programs are shown to be equivalent by unfolding the literals of predicates p'_1 and p'_2. □

4 Input/Output Modes for Literals

Input/output modes were introduced in [16] and further extensively studied from many points of view and for many applications. In the classical view, the mode of a predicate indicates how its arguments will be used in the sense of identifying arguments which belong to the input and to the output. In this paper, in order to eliminate the local variable y, it might be necessary to consider as output the second argument in $p(x_1,y)$ and as input the second argument in $p(x_2,y)$. However, for the sake of clarity, we allow to assign a unique mode to each predicate. Hence, instead of assigning multiple modes to predicates, we consider a different but equivalent program where the predicates are conveniently renamed and different copies of the same predicate are provided, one for each different mode.

Definition 3. *A* mode for an n-ary predicate L, *denoted by* $\mathtt{m} : L$, *is an* n*-tuple* $\mathtt{m} \in \{\mathtt{in},\mathtt{out}\}^n$, *where the position* i *(*$1 \leq i \leq n$*) such that* $\mathtt{m}_i = \mathtt{in}$ *(*$\mathtt{m}_i = \mathtt{out}$*) is considered as an* input *(*output*) position.* □

Throughout this paper, we will use the following notation:

Remark 1. Given a mode $\mathtt{m}$ for a predicate L and a literal $L(\overline{t})$, the expression $\overline{t}_{\mathtt{I}} \overset{\mathtt{m}}{\triangleright} \overline{t}_{\mathtt{O}}$ means the unique partition of $\overline{t}$ into the order-preserving subsequences consisting of the input ($\overline{t}_{\mathtt{I}}$) and output ($\overline{t}_{\mathtt{O}}$) arguments of $L(\overline{t})$ according to $\mathtt{m}$. Besides, the expression $L(\overline{t}_{\mathtt{I}} \overset{\mathtt{m}}{\triangleright} \overline{t}_{\mathtt{O}})$ is called the IO-*form* of $L(\overline{t})$, which implicitly represents $\mathtt{m}$ in $L(\overline{t})$. Whenever the upper mode name $\mathtt{m}$ is irrelevant, we simply omit it and write $L(\overline{t}_{\mathtt{I}} \triangleright \overline{t}_{\mathtt{O}})$. If (all or some of) the literals of a clause are in IO-form, then we say that the clause is in IO-form. From now on, the literals of any clause (in particular, in Dpd-form) can be written in IO-form. □

Definition 4. *Let C be a clause:*

$$H \leftarrow \overline{M},\ L(\overline{t}),\ K_1(\overline{u}^1),\ \ldots\ ,\ K_n(\overline{u}^n),\ \overline{N}$$

where:

- $\overline{y} \subseteq \mathsf{Var}(L(\overline{t}))$,
- $\overline{y} \cap \mathsf{Var}(\overline{N}) = \emptyset$, *and*
- $\overline{y} \cap \mathsf{Var}(K_i(\overline{u}^i)) \neq \emptyset$ *for each* $1 \leq i \leq n$

such that the variables $\overline{y}$ are term-apart in $L(\overline{t})$. Then, the collection of modes $\{\mathtt{m} : L,\ \mathtt{m}^1 : K_1,\ \mathtt{m}^2 : K_2,\ \ldots,\ \mathtt{m}^n : K_n\}$, *denoted by* $\mathtt{VarMode}(C, L(\overline{t}), \overline{y})$, *is defined by:*

(a) $\mathtt{m}_i = \mathtt{in}$ *if* $\mathsf{Var}(t_i) \cap \overline{y} \subseteq \mathsf{Var}(\overline{M})$ *and* $\mathtt{m}_i = \mathtt{out}$ *otherwise.*
(b) For each $1 \leq j \leq n$: $\mathtt{m}_i^j = \mathtt{in}$ *if* $\mathsf{Var}(u_i^j) \cap \overline{y} \neq \emptyset$ *and* $\mathtt{m}_i^j = \mathtt{out}$ *otherwise.* □

Note that, by the term-apartness of $\overline{y}$, the variables in $\mathsf{Var}(\overline{t}_{\mathtt{I}}) \cap \overline{y}$ do not occur in $K_1(\overline{u}^1),\ \ldots\ , K_n(\overline{u}^n)$.

Example 2. Let C be the following clause in a program P:

$$p(x_1, x_2) \leftarrow\ q(f(x_1, y_1), y_1),\ r(y_1, f(x_2, y_2)),\ r(x_2, f(y_2, x_2)),\ q(x_1, x_1).$$

First, we need to rename the second literal $r(x_2, f(y_2, x_2))$, which is replaced with $r'(x_2, f(y_2, x_2))$, and provide a copy of $\mathtt{Def}_P(r)$ for the definition of the new predicate r':

$$C' = p(x_1, x_2) \leftarrow\ q(f(x_1, y_1), y_1),\ r(y_1, f(x_2, y_2)),\ r'(x_2, f(y_2, x_2)),\ q(x_1, x_1).$$

Then, the set of modes $\mathtt{VarMode}(C', r(y_1, f(x_2, y_2)), (y_1,\ y_2))$ is:

$$\{(\mathtt{in}, \mathtt{out}) : r,\ (\mathtt{out}, \mathtt{in}) : r'\}$$

which yields the following IO-form of C':

$$p(x_1, x_2) \leftarrow\ q(f(x_1, y_1), y_1),\ r(y_1 \triangleright f(x_2, y_2)),\ r'(f(y_2, x_2) \triangleright x_2),\ q(x_1, x_1).$$

By contrast, $\mathtt{VarMode}(C', r(y_1, f(x_2, y_1)), x_2)$ gives:

$$p(x_1, x_2) \leftarrow\ q(f(x_1, y_1), y_1),\ r(y_1 \triangleright f(x_2, y_2)),\ r'(x_2, f(y_2, x_2) \triangleright \langle\ \rangle),\ q(x_1, x_1).$$

Note that we have not assigned a mode to the literals on q since r is not mutually recursive with the predicate q. □

Our intention is to eliminate the local variables that occur in the terms $\overline{t}_{\mathtt{O}}$ of a given $L(\overline{t}_{\mathtt{I}} \overset{\mathtt{m}}{\triangleright} \overline{t}_{\mathtt{O}})$. For $(\mathtt{in}, \mathtt{out}) : r$ (see Example 2), that variable is y_2. For this purpose, we must associate modes with the clauses in the definition of L according to $\mathtt{m}$. That is, we first fix $\mathtt{m}$ as the mode for the head literal of each clause C in $\mathtt{Def}_P(L)$. Then, taking into account the input/output partition of the head literal and the local variables of C, we associate a mode with every body literal of C that is mutually recursive with L.

Definition 5. *Let* $\mathtt{m}$ *be the mode assigned to a predicate* L *and* C *be the following normal local-regular*[2] *clause in* Dpd*-form:*

$$L(\overline{u}_{\mathtt{I}} \overset{\mathtt{m}}{\triangleright} \overline{u}_{\mathtt{O}}) \leftarrow \overline{L}^1(\overline{r}^1),\ K_1(\overline{s}^1),\ \overline{L}^2(\overline{r}^2),\ \ldots\ ,\ \overline{L}^n(\overline{r}^n),\ K_n(\overline{s}^n),\ \overline{L}^{n+1}(\overline{r}^{n+1})$$

where each $K_i(\overline{s}^i)$ *is an* n_i*-ary literal. Then, the* mode for the clause C*, denoted by* $\mathtt{ClauseMode}(C, \mathtt{m})$*, is given by the set of modes* $\{\mathtt{m} : L,\ \mathtt{m}^1 : K_1,\ \ldots\ ,\ \mathtt{m}^n : K_n\}$*, where each* $\mathtt{m}^i$ *is defined as follows:*

$$\mathtt{m}^i_j = \begin{cases} \mathtt{in} & \textit{if } \mathsf{LVar}(s^i_j) \cap \mathsf{LVar}(\overline{r}^{i-1}) \neq \emptyset \quad \textit{or} \\ & \quad \mathsf{LVar}(s^i_j) = \emptyset \textit{ and } (\mathsf{Var}(s^i_j) \cap \mathsf{Var}(\overline{u}_{\mathtt{I}}) \neq \emptyset \textit{ or } \mathsf{Var}(s^i_j) \cap \mathsf{Var}(\overline{u}_{\mathtt{O}}) = \emptyset) \\ \mathtt{out} & \textit{otherwise} \end{cases}$$

for every $1 \leq j \leq n_i$*. Besides, assuming that* $\mathtt{Def}_P(L) = \{C_1, \ldots, C_k\}$*, the* mode for the definition of L*, denoted by* $\mathtt{DefMode}(P, L, \mathtt{m})$*, is given by the set of clause modes* $\{\mathtt{ClauseMode}(C_1, \mathtt{m}), \ldots, \mathtt{ClauseMode}(C_k, \mathtt{m})\}$. □

Example 3. Let P be the following program:

$E3.1:\ \ h(x) \leftarrow\ p(x, y),\ q(y),\ r(x)$
$E3.2:\ \ p(a, b)$
$E3.3:\ \ p(f(v), z) \leftarrow\ h(v),\ r(z)$

such that $\mathsf{MR}_P(p) = \{h,\ p\}$. Then, the set of modes $\mathtt{VarMode}(E3.1, p(x, y), y)$ yields the following IO-form for the clause $E3.1$:

$$h(x) \leftarrow\ p(x \triangleright y),\ q(y \triangleright \langle\,\rangle),\ r(x)$$

Besides, since $(\mathtt{in}, \mathtt{out})$ is the mode for the predicate p, the set of clause modes $\mathtt{DefMode}(P, p, (\mathtt{in}, \mathtt{out}))$ yields the following clauses:

$$p(a \triangleright b)$$
$$p(f(v) \triangleright z) \leftarrow\ h(v \triangleright \langle\,\rangle),\ r(z)$$

where the mode $(\mathtt{in})$ is assigned to the predicate h. Finally, the set of clause modes $\mathtt{DefMode}(P, h, (\mathtt{in}))$ gives the clause:

$$h(x \triangleright \langle\,\rangle) \leftarrow\ p(x \triangleright y),\ q(y \triangleright \langle\,\rangle),\ r(x)$$

Note that, in this example, we have assigned a unique mode to the predicates h and p without renaming. □

Now, starting with a mode $\mathtt{m}$ assigned to a predicate L, we collect the modes that are assigned to all the predicates that are mutually recursive with L.

Definition 6. *Let* $\mathtt{m}$ *be the mode assigned to a predicate* L *in a program* P*. The* relation $\prec$ *is defined by:*

$$\mathtt{m} : L\ \prec\ \mathtt{m}' : M\ \iff\ \mathtt{m}' : M \in \mathtt{DefMode}(P, L, \mathtt{m}).$$

[2] By Lemma 2, we can assume local-regularity.

Besides, the mode for the predicates that are mutually recursive with L, *denoted by* $\texttt{ModeMR}(P, L, \mathtt{m})$, *is given by the least set of modes that contains the singleton* $\{\mathtt{m} : L\}$ *and is upwards closed with respect to the relation* $\prec$. □

Example 4. In the program of Example 3, the set of modes $\texttt{ModeMR}(P, p, (\mathtt{in}, \mathtt{out}))$ is given by:

$$\{(\mathtt{in}, \mathtt{out}) : p,\ (\mathtt{in}) : h\ \}$$

Note that, since we assign a unique mode to each predicate, we have that $\texttt{ModeMR}(P, p, (\mathtt{in}, \mathtt{out})) = \texttt{ModeMR}(P, h, (\mathtt{in}))$. □

Next, we present the notion of tail recursion, which is relative to a mode.

Definition 7. *The definition of a predicate* L *in a program* P *is* tail recursive w.r.t. a mode $\mathtt{m}$ *iff the following two conditions hold:*

(a) $\mathsf{MR}_P(L) = \{L\}$
(b) the set of clause modes $\texttt{DefMode}(P, L, \mathtt{m})$ *yields clauses of the following two forms:*

$$(1)\ \ L(\overline{r}_{\mathtt{I}} \overset{\mathtt{m}}{\triangleright} \overline{r}_{\mathtt{O}}) \leftarrow \overline{E}$$
$$(2)\ \ L(\overline{s}_{\mathtt{I}} \overset{\mathtt{m}}{\triangleright} \overline{z}) \leftarrow \overline{F},\ L(\overline{s}'_{\mathtt{I}} \overset{\mathtt{m}}{\triangleright} \overline{z})$$

where $N \notin \mathsf{MR}_P(L)$ *for each* $N \in \overline{E} \cdot \overline{F}$ *and* $\overline{z}$ *is a tuple of fresh variables.* □

This notion of tail recursion is more restrictive than the classical one. Intuitively stated, it means that, besides the fact that only the rightmost literal is head-dependent, all the recursive calls return exactly the same value.

5 Elimination of the Local Variables of a Literal

When $\texttt{Def}_P(L)$ is tail recursive w.r.t. a mode, we are able to eliminate the local variables that occur in the output arguments of the selected literal $L(\overline{t})$. This is done by substituting a set of clauses, called $\mathsf{LVF}_P(C, L(\overline{t}))$, for the clause C.

Definition 8. *Let* $\texttt{VarMode}(C, L(\overline{t}), \overline{y})$ *yield the following* IO*-form for* C*:*

$$C = H \leftarrow \overline{M},\ L(\overline{t}_{\mathtt{I}} \overset{\mathtt{m}}{\triangleright} \overline{t}_{\mathtt{O}}),\ K_1(\overline{u}^1_{\mathtt{I}} \overset{\mathtt{m}^1}{\triangleright} \overline{u}^1_{\mathtt{O}}),\ \ldots\ ,\ K_n(\overline{u}^n_{\mathtt{I}} \overset{\mathtt{m}^n}{\triangleright} \overline{u}^n_{\mathtt{O}}),\ \overline{N}$$

where $\overline{y} = \mathsf{LVar}(L(\overline{t}))$ *are term-apart in* $L(\overline{t})$ *and* $\overline{y} \cap \mathsf{Var}(\overline{N}) = \emptyset$. *If* $\texttt{Def}_P(L)$ *is tail recursive w.r.t.* $\mathtt{m}$, *then* $\mathsf{LVF}_P(C, L(\overline{t}))$ *consists of the following clauses:*

(a) One single clause of the form:

$$H \leftarrow \overline{M},\ p(\overline{t}_{\mathtt{I}}, \overline{w}_{\mathtt{I}} \triangleright \overline{u}_{\mathtt{O}}, \overline{w}_{\mathtt{O}}),\ \overline{N}$$

where $\overline{u}_{\mathtt{O}} = \overline{u}^1_{\mathtt{O}} \cup \ldots \cup \overline{u}^n_{\mathtt{O}}$.
(b) For each non-recursive clause $L(\overline{r}_{\mathtt{I}} \overset{\mathtt{m}}{\triangleright} \overline{r}_{\mathtt{O}}) \leftarrow \overline{E} \in \texttt{Def}_P(L)$, *a clause:*

$$p(\overline{r}_{\mathtt{I}}\sigma, \overline{w}_{\mathtt{I}}\sigma \triangleright \overline{v}, \overline{w}_{\mathtt{O}}\sigma) \leftarrow \overline{E}\sigma,\ K_1(\overline{u}^1_{\mathtt{I}}\sigma \overset{\mathtt{m}^1}{\triangleright} \overline{v}^1),\ \ldots\ ,\ K_n(\overline{u}^n_{\mathtt{I}}\sigma \overset{\mathtt{m}^n}{\triangleright} \overline{v}^n)$$

where $\sigma = mgu(\overline{r}_0, \overline{t}_0)$ *and* $\overline{v} = \overline{v}^1 \cup \ldots \cup \overline{v}^n$.

(c) For each recursive clause $L(\overline{s}_I \overset{\mathtt{m}}{\triangleright} \overline{z}) \leftarrow \overline{F},\ L(\overline{s}'_I \overset{\mathtt{m}}{\triangleright} \overline{z}) \in \mathtt{Def}_P(L)$*, a clause:*

$$p(\overline{s}_I, \overline{w}_I \triangleright \overline{v}, \overline{w}_0) \leftarrow \overline{F},\ p(\overline{s}'_I, \overline{w}_I \triangleright \overline{v}, \overline{w}_0)$$

where p *is a fresh predicate symbol,* $\overline{v}$ *is a fresh tuple of variables of the size of* $\overline{u}_0$, $\overline{w}_I = \mathsf{GVar}(\overline{t}_I) \setminus \mathsf{GVar}(\overline{t}_I \cdot \overline{u}_0)$ *and* $\overline{w}_0 = \mathsf{GVar}(\overline{u}_I) \setminus \mathsf{GVar}(\overline{t}_I \cdot \overline{u}_0)$. □

In the above definition, $\overline{w}_I$ and $\overline{w}_0$ are used to keep the links between literals through global variables. Note that, in the original clause C, the exact variables $\overline{y}$ occur in $\overline{t}_0$ and $\overline{u}^j_I$ for $1 \leq j \leq n$, but do not occur in $\mathsf{LVF}_P(C, L(\overline{t}))$. Hence, $\overline{y}$ has been eliminated from C. Some occurrences of local variables could remain in the clauses of the form *(b)*, but they are in literals that do not depend on the head. We will return to this matter for a discussion about termination.

Theorem 1. *Let* $\mathsf{LVF}_P(C, L(\overline{t}))$, $L(\overline{t})$ *and the clause* C *be as in Definition 8. The programs* P *and* $P' = P \setminus \{C\} \cup \mathsf{LVF}_P(C, L(\overline{t}))$ *are Clark-Kunen equivalent.*

Proof. Let us assume that $\mathtt{VarMode}(C, L(\overline{t}), \overline{y})$ yields the following IO-form:

$$C: \quad H \leftarrow \overline{M},\ L(\overline{t}_I \overset{\mathtt{m}}{\triangleright} \overline{t}_0),\ K_1(\overline{u}^1_I \overset{\mathtt{m}^1}{\triangleright} \overline{u}^1_0),\ \ldots,\ K_n(\overline{u}^n_I \overset{\mathtt{m}^n}{\triangleright} \overline{u}^n_0),\ \overline{N}$$

where $\overline{y} = \mathsf{LVar}(L(\overline{t}))$. A program P_0 is obtained by introducing the new predicate p in P. In P_0, p is defined by the single clause:

$$D: \quad p(\overline{x}, \overline{w}_I \triangleright \overline{z}, \overline{w}_0) \leftarrow L(\overline{x} \overset{\mathtt{m}}{\triangleright} \overline{t}_0),\ K_1(\overline{u}^1_I\sigma \overset{\mathtt{m}^1}{\triangleright} \overline{z}^1),\ \ldots,\ K_n(\overline{u}^n_I\sigma \overset{\mathtt{m}^n}{\triangleright} \overline{z}^n)$$

where $\overline{x}$, $\overline{z}^1$, $\ldots$, $\overline{z}^n$ are tuples of fresh variables, the sets $\overline{w}_I$ and $\overline{w}_0$ are obtained as in Definition 8 and $\overline{z} = \overline{z}^1 \cup \ldots \cup \overline{z}^n$. The programs P and P_0 are trivially equivalent. Next, we obtain P_1 from P_0 by folding C using D:

$$C': \quad H \leftarrow \overline{M},\ p(\overline{t}_I, \overline{w}_I \triangleright \overline{u}_0, \overline{w}_0),\ \overline{N}$$

where $\overline{u}_0 = \overline{u}^1_0 \cup \ldots \cup \overline{u}^n_0$. Then, the program P_2 is obtained by unfolding the literal $L(\overline{x} \triangleright \overline{t}_0)$ in the clause D. Since $\mathtt{Def}_P(L)$ (and therefore, $\mathtt{Def}_{P_1}(L)$) is tail recursive w.r.t. $\mathtt{m}$, it consists of clauses of the following two forms:

$$T1.1: \ L(\overline{r}_I \overset{\mathtt{m}}{\triangleright} \overline{r}_0) \leftarrow \overline{E}$$
$$T1.2: \ L(\overline{s}_I \overset{\mathtt{m}}{\triangleright} \overline{z}) \leftarrow \overline{F},\ L(\overline{s}'_I \overset{\mathtt{m}}{\triangleright} \overline{z})$$

and, hence, after the unfolding step, we get clauses of the form:

$$T1.3: \ p(\overline{r}_I\sigma, \overline{w}_I\sigma \triangleright \overline{z}, \overline{w}_0\sigma) \leftarrow \overline{E}\sigma,\ K_1(\overline{u}^1_I\sigma \overset{\mathtt{m}^1}{\triangleright} \overline{z}^1),\ \ldots,\ K_n(\overline{u}^n_I\sigma \overset{\mathtt{m}^n}{\triangleright} \overline{z}^n)$$
$$T1.4: \ p(\overline{s}_I, \overline{w}_I \triangleright \overline{z}, \overline{w}_0) \leftarrow \overline{F},\ L(\overline{s}'_I \overset{\mathtt{m}}{\triangleright} \overline{t}_0),\ K_1(\overline{u}^1_I\sigma \overset{\mathtt{m}^1}{\triangleright} \overline{z}^1), \ldots, K_n(\overline{u}^n_I\sigma \overset{\mathtt{m}^n}{\triangleright} \overline{z}^n)$$

where $\sigma = mgu(\overline{r}_0, \overline{t}_0)$. The programs P_2 and P' are syntactically equal except for the clauses $T1.4$, where $L(\overline{s}'_I \overset{\mathtt{m}}{\triangleright} \overline{t}_0),\ K_1(\overline{u}^1_I\sigma \overset{\mathtt{m}^1}{\triangleright} \overline{z}^1),\ \ldots,\ K_n(\overline{u}^n_I\sigma \overset{\mathtt{m}^n}{\triangleright} \overline{z}^n)$ correspond with the literal $p(\overline{s}'_I, \overline{w}_I \triangleright \overline{z}, \overline{w}_0)$ in the program P'. It is easy (but tedious) to prove that P_2 and P' are equivalent using induction on bottom-up computation of the Clark-Kunen semantics. The interested reader may find the details in [1]. Therefore, the programs P and P' are also equivalent. □

Example 5. Let P be the following program:

$E5.1:\ \ q(x_1, x_2) \leftarrow\ member(y, x_1),\ \neg\, member(y, x_2)$
$E5.2:\ \ member(x, [x|\_])$
$E5.3:\ \ member(x_1, [\_|x_2]) \leftarrow\ member(x_1, x_2)$

In order to eliminate the local variable y, we first obtain the set of modes $\texttt{VarMode}(E5.1, member(y, x_1), y)$, that yields the following IO-form of $E5.1$:

$$q(x_1, x_2) \leftarrow\ member(x_1 \triangleright y),\ \neg\, member(y \triangleright x_2)$$

where the mode $(\texttt{out}, \texttt{in})$ is assigned to *member*. Besides, the set of clause modes $\texttt{DefMode}(P, member, (\texttt{out}, \texttt{in}))$ gives the following clauses:

$$member([x|\_] \triangleright x)$$
$$member([\_|x_2] \triangleright x_1) \leftarrow\ member(x_2 \triangleright x_1)$$

Since $\texttt{Def}_P(member)$ is tail recursive w.r.t. the mode $(\texttt{out}, \texttt{in})$, then the set of clauses $\mathsf{LVF}_P(E5.1, member(y, x_1))$ is a compound of:

$E5.4:\ \ q(x_1, x_2) \leftarrow\ p(x_1 \triangleright x_2)$
$E5.5:\ \ p([x|\_] \triangleright z) \leftarrow\ \neg\, member(x \triangleright z)$
$E5.6:\ \ p([\_|x] \triangleright z) \leftarrow\ p(x \triangleright z)$ □

6 Tail Recursive Transformation

We use the well known technique of the *call stack* for transforming recursion into tail recursion. The method in [21] also uses the call stack to convert a definite program into a continuation passing style program which, in particular, is a binary program (clauses have at most one literal in the body). We perform a similar, but simpler, treatment since our target program is not required to be binary. For the sake of readability, consider that the clauses to be transformed have (at most) two head-dependent literals in the body (see (1) below). Generalization to n literals is not difficult but unnecessarily complicates the description of the transformation we explain below.

A given $\texttt{Def}_P(L)$, that is not tail recursive with respect to a mode $\texttt{m}$, is transformed into a new definition formed by the single clause:

$$L(\overline{x} \overset{\texttt{m}}{\triangleright} \overline{z}) \leftarrow\ q(\overline{x}, [c_L] \triangleright \overline{z}).$$

where q is a fresh predicate and c_L is a constant that stands for L. This new definition is obviously tail recursive w.r.t. any mode (in particular w.r.t. $\texttt{m}$). Then, we use the original definition of L for providing a tail recursive definition of q w.r.t. the mode in $q(\overline{x}, [c_L] \triangleright \overline{z})$, which is the extension of $\texttt{m}$ by adding an input position. The definition of q is given by:

– the single clause $q(\overline{z}, [\,] \triangleright \overline{z})$,

– plus, for each element $\mathtt{m}' : K \in \mathtt{ModeMR}(P, L, \mathtt{m})$ and each clause from $\mathtt{Def}_P(K)$ (in Dpd-form):

$$K(\overline{t}_{\mathtt{I}} \overset{\mathtt{m}'}{\triangleright} \overline{t}_{\mathtt{O}}) \leftarrow \overline{L}^1,\ K_1(\overline{s}^1_{\mathtt{I}} \overset{\mathtt{m}^1}{\triangleright} \overline{s}^1_{\mathtt{O}}),\ \overline{L}^2,\ K_2(\overline{s}^2_{\mathtt{I}} \overset{\mathtt{m}^2}{\triangleright} \overline{s}^2_{\mathtt{O}}),\ \overline{L}^3 \qquad (1)$$

the following four clauses:

(i) $q(\overline{t}_{\mathtt{I}}, [c_K|S] \triangleright \overline{z}) \leftarrow q(\overline{t}_{\mathtt{I}}, [\overline{w}^1, c^C_1, \overline{w}^2, c^C_2, \overline{w}^C, c^C|S] \triangleright \overline{z})$
(ii) $q(\overline{t}_{\mathtt{I}}, [\overline{w}^1, c^C_1|S] \triangleright \overline{z}) \leftarrow \overline{L}^1,\ q(\overline{s}^1_{\mathtt{I}}, [c_{K_1}|S] \triangleright \overline{z})$
(iii) $q(\overline{s}^1_{\mathtt{O}}, [\overline{w}^2, c^C_2|S] \triangleright \overline{z}) \leftarrow \overline{L}^2,\ q(\overline{s}^2_{\mathtt{I}}, [c_{K_2}|S] \triangleright \overline{z})$
(iv) $q(\overline{s}^2_{\mathtt{O}}, [\overline{w}^C, c^C|S] \triangleright \overline{z}) \leftarrow \overline{L}^3,\ q(\overline{t}_{\mathtt{O}}, S \triangleright \overline{z})$

where $\overline{z}$ is a tuple of fresh variables, $\overline{w}^1 = \mathsf{GVar}(\overline{L}^1 \cdot \overline{s}^1_{\mathtt{I}})$, $\overline{w}^2 = \mathsf{GVar}(\overline{s}^1_{\mathtt{O}} \cdot \overline{L}^2 \cdot \overline{s}^2_{\mathtt{I}})$ and $\overline{w}^C = \mathsf{GVar}(\overline{s}^2_{\mathtt{O}} \cdot \overline{L}^3 \cdot \overline{t}_{\mathtt{O}}) \cap (\overline{w}^1 \cup \overline{w}^2)$.

The sets of variables in the stack are used to keep the links between literals through global variables. As defined, these sets are not minimal. A more sophisticated analysis would produce smaller sets of variables. Note that the fresh predicate q is used with tuples of terms of different sizes in the input arguments and, therefore, we are really defining several predicates, one for each arity. This is not a minor difference if we consider the dependences of predicates, especially in Definition 8. By contrast, the size of the tuple of fresh variables $\overline{z}$ is equal in all the clauses and it coincides with the number of output positions in $\mathtt{m}$.

Example 6. Let P be the following program:

$E6.1:\ k(a \overset{m}{\triangleright} b)$
$E6.2:\ k(f(x_1) \overset{m}{\triangleright} f(x_2)) \leftarrow q(x_1 \overset{m}{\triangleright} x_2)$
$E6.3:\ q(x_1 \overset{m}{\triangleright} x_2) \leftarrow \neg h(x_1, x_2)$
$E6.4:\ q(f(x_1) \overset{m}{\triangleright} f(x_2))) \leftarrow k(x_1 \overset{m}{\triangleright} y),\ q(g(y, x_1) \overset{m}{\triangleright} x_2)$

where $\mathtt{ModeMR}(P, k, \mathtt{m}) = \{\mathtt{m} : k,\ \mathtt{m} : q\}$ for $\mathtt{m} = (\mathtt{in}, \mathtt{out})$. Since $\mathtt{Def}_P(k)$ is not tail recursive w.r.t. $\mathtt{m}$, a tail recursive definition of k w.r.t. $\mathtt{m}$ consists of the following clauses. First, the initial clauses:

$E6.5:\ k(x \overset{m}{\triangleright} z) \leftarrow p(x, [c_k] \triangleright z)$
$E6.6:\ p(z, [\,] \triangleright z)$

where the constant c_k stands for the predicate k. Since $E6.1$ is non-recursive, the above clauses **(ii)** and **(iii)** are not necessary (c^1 corresponds with $E6.1$):

$E6.7:\ p(a, [c_k|S] \triangleright z) \leftarrow p(a, [c^1|S] \triangleright z)$
$E6.8:\ p(a, [c^1|S] \triangleright z) \leftarrow p(b, S \triangleright z)$

In the clause $E6.2$, there is one recursive literal. Hence, **(iii)** is not necessary:

$E6.9:\ p(f(x_1), [c_k|S] \triangleright z) \leftarrow p(f(x_1), [c^2_1, c^2|S] \triangleright z)$
$E6.10:\ p(f(x_1), [c^2_1|S] \triangleright z) \leftarrow p(x_1, [c_q|S] \triangleright z)$
$E6.11:\ p(x_2, [c^2|S] \triangleright z) \leftarrow p(f(x_2), S \triangleright z)$

where c^2 stands for the clause $E6.2$, c_1^2 for the literal $q(x_1 \triangleright x_2)$ and c_q for the predicate q. The clause $E6.3$ is non-recursive (c^3 corresponds to $E6.3$), then:

$E6.12 : p(x_1, [c_q|S] \triangleright z) \leftarrow p(x_1, [c^3|S] \triangleright z)$
$E6.13 : p(x_1, [c^3|S] \triangleright z) \leftarrow \neg\, h(x_1, y),\ p(y, S \triangleright z)$

Finally, the clause $E6.4$ has two recursive literals, where c^4, c_1^4 and c_2^4 respectively stand for the clause itself and the literals $k(x_1 \triangleright x_2)$ and $q(g(y, x_1) \triangleright x_2)$:

$E6.14 : p(f(x_1), [c_q|S] \triangleright z) \leftarrow p(f(x_1), [c_1^4, x_1, c_2^4, c^4|S] \triangleright z)$
$E6.15 : p(f(x_1), [c_1^4|S] \triangleright z) \leftarrow p(x_1, [c_k|S] \triangleright z)$
$E6.16 : p(f(x), [x_1, c_2^4|S] \triangleright z) \leftarrow p(g(x, x_1), [c_q|S] \triangleright z)$
$E6.17 : p(x_2, [c^4|S] \triangleright z) \leftarrow p(f(x_2), S \triangleright z)$ □

It is easy to see that, in the above example, the transformed program uses the stack to simulate the computation of the original program for any goal. In general, the transformation works similarly for clauses with any arbitrary number of head-dependent body literals and it preserves the Clark-Kunen equivalence. The proof of this result is omitted for brevity and it can be found in [1].

Theorem 2. *Let* $\mathtt{m}$ *be a mode for a predicate* L *in a program* P*. Then, the definition of* L *can be transformed into a Clark-Kunen equivalent definition that is tail recursive w.r.t.* $\mathtt{m}$*.* □

7 An Algorithm for Elimination of Local Variables

Now, we present our algorithm —see Figure 1— for eliminating local variables. In previous sections, we have shown the basic transformations that our algorithm performs. These are: local-regulation (LR), elimination of a tuple of local variables (LVF) and tail-recursive transformation (TR). These three basic operations have different effects on the set of local variable occurrences in the transformed program. Sometimes, new local variables could arise in the new clauses and, at the same time, other local variables are erased. In this section, we show that it is possible to give decidable conditions that guarantee termination. Such conditions are concerned with the notion of a literal candidate, which we will make precise after a brief introduction of the algorithm itself.

The algorithm outlined in Figure 1 first collects all the literals that contain some local variables in the set $\mathsf{Lit}(P)$. Then, until $\mathsf{Lit}(P)$ becomes empty, we select any literal $L(\overline{t})$ from that set. Let assume that $L(\overline{t})$ occurs in a clause C such that $\mathtt{m}$ is the mode assigned to the predicate L by $\mathtt{VarMode}(C, L(\overline{t}), \mathsf{LVar}(L(\overline{t})))$ and H is the literal in the clause head. On the one hand, if the selected literal is not a candidate (line 5), then we simply delete it from $\mathsf{Lit}(P)$ and proceed to the next iteration. On the other hand, if $L(\overline{t})$ is a candidate, then there are two main cases. If $\mathtt{Def}_P(L)$ is tail recursive w.r.t. $\mathtt{m}$ (line 7), we substitute $\mathsf{LVF}_P(C, L(\overline{t}))$ for C. Besides, we delete from $\mathsf{Lit}(P)$ all the literals that occur in C and add to $\mathsf{Lit}(P)$ all the literals from $\mathsf{LVF}_P(C, L(\overline{t}))$ that contain some local variables. Otherwise, if the definition of L is not tail recursive w.r.t. $\mathtt{m}$ (line 12),

we transform $\mathtt{Def}_P(L)$ into tail recursive w.r.t. that mode. Besides, we delete from $\mathsf{Lit}(P)$ the literals that occur in $\mathtt{Def}_P(L)$ and add to $\mathsf{Lit}(P)$ the literals that contain some local variables in the new clauses. In the former case (line 7), we directly eliminate the subset of local variables from the clause C. In the latter case (line 12), the definition of L is adapted for fulfilling the condition of the first case. Note that, in the last case (line 12), the clause C is removed from P only if L depends on H, because $\mathtt{Def}_P(L) = \mathtt{Def}_P(H)$. Therefore, we have to select a different candidate in the next step. Otherwise, $L(\overline{t})$ can be selected as a candidate in the next step and the subset of local variables will be eliminated.

```
1  collect in Lit(P) all the literals that contain some local variables
2  while Lit(P) ≠ ∅ loop
3     select a literal L(t̄) in a clause C = H ← M̄, L(t̄), N̄
4     let m : L ∈ VarMode(C, L(t̄), LVar(L(t̄)))
5     if L is not a candicate then
6        delete L from Lit(P)
7     elsif Def_P(L) is tail recursive w.r.t. m then
8        substitute LVF_P(C, L) for the clause C in P
9        delete from Lit(P) the literals in C
10       add to Lit(P) the literals in LVF_P(C, L) that contain
11                        some local variables
12    else
13       transform Def_P(L) into tail recursive w.r.t. m
14       delete from Lit(P) the literals in Def_P(L)
15       add to Def_P(L) the literals in the new clauses that
16                        contain some local variables
17    end if
18 end loop
```

Fig. 1. An Algorithm for Elimination of Local Variables

The algorithm in Figure 1 terminates when there is no candidate, although the resulting program may not necessarily be *lvf*. With regard to termination, the transformation LR introduces new local variables but, since local-regularity is preserved by the other two transformations, LR is applied only finitely many times. The transformation LVF does not introduce new local variables, but the local variables $\overline{y}$ to be removed could be only partially eliminated in some cases. In particular, a clause of the form *(b)* of Definition 8 could contain some variables from $\overline{y}$. However, they necessarily occur in literals that do not depend on the clause head. This fact ensures termination because the set of all mutually recursive components of a program is well-foundedly ordered w.r.t. predicate dependencies. The transformation TR, explained in Section 6, could introduce new local variables, which could lead to non-termination problems through clauses **(i)** and **(iv)** (see Section 6). The problem arises when new local variables exactly occur in literals which depend on the clause head because, in that case,

the resulting definition is never tail recursive w.r.t that is given by `VarMode` with respect to the new local variables. Therefore, we would transform the definition of a predicate into tail recursive w.r.t. the corresponding mode infinitely many times. On the one hand, since clauses **(i)** are binary, they must be *lvf* in order to avoid that problem. On the other hand, let us illustrate the problem of clauses **(iv)** by the following example.

Example 7. Consider the following program P:

$E7.1:\ perfectsq(v) \leftarrow mult(y, y, v)$
$E7.2:\ mult(0, x, 0)$
$E7.3:\ mult(s(x_1), x_2, x_3) \leftarrow mult(x_1, x_2, y),\ sum(x_2, y, x_3)$

To eliminate the local variable y in the first clause, $\texttt{VarMode}(E7.1, mult(y, y, v), y)$ assigns the mode $(\texttt{out}, \texttt{out}, \texttt{in})$ to the predicate *mult*. Since $\texttt{Def}_P(mult)$ is not tail recursive w.r.t. $(\texttt{out}, \texttt{out}, \texttt{in})$, we transform the definition of the predicate *mult* as explained in Section 6. Then, the clause **(iv)** obtained from $E7.2$ is:

$$q(0, [c^{E7.2}|S]_{\triangleright} z_1, z_2) \leftarrow q(0, y', S_{\triangleright} z_1, z_2)$$

where y' is a new local variable that occurs in a literal that depends on the clause head. However, if the clause $E7.2$ is replaced with the following two clauses:

$E7.4:\ mult(0, 0, 0)$
$E7.5:\ mult(0, s(x), 0) \leftarrow mult(0, x, 0)$

then, the respective clauses **(iv)** are *lvf* (therefore, not problematic):

$$q(0, [c^{E7.4}|S]_{\triangleright} z_1, z_2) \leftarrow q(0, 0, S_{\triangleright} z_1, z_2)$$
$$q(0, x, [c^{E7.5}|S]_{\triangleright} z_1, z_2) \leftarrow q(0, s(x), S_{\triangleright} z_1, z_2)$$

Note that the programs P and $P \setminus \{E7.2\} \cup \{E7.4, E7.5\}$ are not Clark-Kunen equivalent. □

In order to avoid the termination problem, we can establish syntactic conditions to ensure that the clauses **(i)** are *lvf* and also that the new local variables of **(iv)** occur in literals which do not depend on the head. This is a sufficient condition for termination assuming that the literal selection rule (line 3 in Figure 1) is fair, although the resulting program might be not *lvf*. This condition is formally stated in the following definition.

Definition 9. *Let P be a program and $N(\overline{u})$ be a literal in a clause C such that $\texttt{VarMode}(C, N(\overline{u}), \mathsf{LVar}(N(\overline{u})))$ assigns the mode $\texttt{m}$ to the predicate N and $\overline{u}_{\texttt{I}\,\triangleright}\,\overline{u}_{\texttt{O}}$ is the unique partition of the tuple of terms $\overline{u}$ w.r.t. $\texttt{m}$. Then, the literal $N(\overline{u})$ is called* candidate *iff:*

(a) $\mathsf{LVar}(N(\overline{u}))$ *are term-apart in* $N(\overline{u})$
(b) $\texttt{Def}_P(K)$ *is local-regular for every* $K \in \mathsf{MR}_P(N)$
(c) $\mathsf{LVar}(\overline{u}_{\texttt{O}}) \neq \emptyset$

(d) for each element $\mathtt{m}' : K \in \mathtt{ModeMR}(P, N, \mathtt{m})$ *and each clause in* $\mathtt{Def}_P(K)$ *of the form* $K(\overline{t}_{\mathtt{I}} \overset{\mathtt{m}'}{\triangleright} \overline{t}_{\mathtt{O}}) \leftarrow \overline{L}^1,\ K_1(\overline{s}^1_{\mathtt{I}} \triangleright \overline{s}^1_{\mathtt{O}}),\ \ldots,\ \overline{L}^n,\ K_n(\overline{s}^n_{\mathtt{I}} \triangleright \overline{s}^n_{\mathtt{O}}),\ \overline{L}^{n+1}$[3]*:*

(d.1) $\mathsf{GVar}(\overline{L}^1 \cdot \overline{s}^1_{\mathtt{I}} \cdot \overline{s}^1_{\mathtt{O}} \cdot \ldots \cdot \overline{L}^n \cdot \overline{s}^n_{\mathtt{I}}) \subseteq \mathsf{GVar}(\overline{t}_{\mathtt{I}})$

(d.2) $\mathsf{GVar}(\overline{t}_{\mathtt{O}}) \subseteq \mathsf{GVar}(\overline{t}_{\mathtt{I}} \cdot \overline{L}^{n+1} \cdot \overline{s}^n_{\mathtt{O}})$

(e) $\mathtt{Def}_P(K)$ *is a normal definition for every* $K \in \mathsf{MR}_P(N)$ □

The conditions $(d.1)$ and $(d.2)$ in the above definition respectively ensure that the clauses **(i)** and **(iv)** are not problematic. Note that, in Example 7, the literal $mult(y, y, v)$ is not a candidate because $(d.2)$ does not hold for $E7.2$ and mode $(\mathtt{out}, \mathtt{out}, \mathtt{in})$, since $\mathsf{GVar}(\overline{t}_{\mathtt{O}}) = \{x\}$ and $\mathsf{GVar}(\overline{t}_{\mathtt{I}} \cdot \overline{L}^{n+1} \cdot \overline{s}^n_{\mathtt{O}}) = \emptyset$. On the contrary, the clauses $E7.4$ and $E7.5$ avoid this problem, because the former does not have any global variable, whereas for the latter $\mathsf{GVar}(\overline{t}_{\mathtt{O}}) = \{x\}$ and $\mathsf{GVar}(\overline{t}_{\mathtt{I}} \cdot \overline{L}^{n+1} \cdot \overline{s}^n_{\mathtt{O}}) = \{x\}$. Besides, the condition (e) ensures that, dealing with normal logic programs, the transformations LVF and LR are applicable to the involved definition. Note that the elimination of some local variables could turn a non-candidate literal into a candidate one when its complex definition becomes normal.

We have verified that the algorithm in Figure 1 successfully works for most of the programs in Sterling and Shapiro [22] (about 35 definite programs and 6 normal programs). The interested reader may find them and their *lvf* version in the URL: http://www.sc.ehu.es/jiwlucap/LVF.html. Outstanding exceptions are the programs that use difference-lists. Let us give an example.

Example 8. Let P be the following program:

$$
\begin{array}{ll}
E8.1: & flatten(x_1, x_2) \leftarrow flatten\_dl(x_1,\ x_2 \setminus [\,]) \\
E8.2: & flatten\_dl([\,],\ x \setminus x) \\
E8.3: & flatten\_dl(x_1,\ [x_1|x_2] \setminus x_2) \leftarrow constant(x_1),\ x_1 \neq [\,] \\
E8.4: & flatten\_dl([x_1|x_2],\ x_3 \setminus x_4) \leftarrow flatten\_dl(x_1,\ x_3 \setminus y), \\
 & \qquad\qquad\qquad\qquad\quad flatten\_dl(x_2,\ y \setminus x_4)
\end{array}
$$

that flattens a list of lists using difference-lists. The algorithm in Figure 1 cannot eliminate the local variable y in the clause $E8.4$. In order to eliminate y, the mode $(\mathtt{in}, \mathtt{out}) : flatten\_dl$ is given by $\mathtt{VarMode}(E8.4, flatten\_dl(x_1,\ x_3 \setminus y), y)$. However, the clause $E8.2$ does not hold the condition $(d.2)$ in Definition 9 with respect to this mode. □

In spite of this drawback, our algorithm successfully eliminates all the local variables from a wide subclass of normal logic programs. That is, many pograms satisfy the conditions in Definition 9. In order to provide some more intuitition on the Definition 9, we would like to point out that any well-moded program ([9, 10, 19]) satisfy it. Roughly speaking, in a well-moded program, each predicate has assigned a unique mode such that the literals (in clause bodies) satisfy a left-to-right *producer-consumer* relation. For mostly well-moded clauses, that relation directly ensures the conditions $(d.1)$ and $(d.2)$ of Definition 9. Some

[3] For technical reasons, we assume that $\overline{s}^0_{\mathtt{O}}$ is the tuple $\overline{t}_{\mathtt{I}}$

exceptions are due to the fact that we only assign modes to some predicates (the ones that are mutually recursive with the predicate in the clause head). This shortcoming can be avoided by requiring local-regulation (see Definition 2) in the global variables as it is required for the local ones. Notice that, in Example 7, the clauses $E7.4$ and $E7.5$ are well-moded. The first one is trivially well-moded because its body is empty and all the terms in its head are ground. In the clause $E7.5$, there is a well-moded *producer-consumer* relation since the literal in the clause body *produces* the output variable x of the head. Notice also that, on the contrary, the clause $E7.2$ cannot be well-moded because the output variable x is neither *produced* by the clause body (it is empty), nor is it also an input variable.

8 Conclusions and Related Work

We have introduced an algorithm that removes local variables from normal programs while preserving Clark-Kunen semantics and show that our algorithm eliminates all the local variables from a wide range of normal logic programs. However, it is still an open problem to decide whether any normal (or, even, definite) program can be transformed into an *lvf* one, despite the result of [24].

Since our transformation includes fresh symbols and new literals, a real implementation should clean up the target program to prevent a hypothetical performance deterioration. Superfluous literals can be removed by unfolding. Note that unfolding in an *lvf* program preserves the *lvf* feature. Furthermore, we can reduce the blow-up of new symbols and literals by keeping the original definition of predicates in the program, although their tail recursive versions are used for performing the local variable elimination.

The work most closely related to our own is [21], where a continuation passing style (CPS, for short) transformation is introduced for definite programs. The CPS conversion is also related to both the call stack technique and the notion of mode. Moreover, the authors introduce the possibility of local variable elimination through CPS conversion and give a sufficient condition for successful elimination, called *ground I/O condition*, in a definite program. The ground I/O condition of a clause depends on some (arbitrary) mode for every predicate occurring in it. Anyway, there are marked differences in our approach and results. First, we deterministically assign one mode to each literal taking into account the local variables to be eliminated, whereas the transformation of [21] depends on an arbitrary mode. Second, we only manage the clauses that belong to the predicate definition (and the mutually recursive ones) of the selected literal. By contrast, in the approach of [21], the definition of all the literals in the affected clause are handled and all of them must satisfy the ground I/O condition.

The aim of [18] is to yield more efficient SLD-computations of definite programs by the elimination of redundant computations caused by the so-called *unnecessary* variables. Local variables are a special kind of such unnecessary variables, because that method also considers as unnecessary those variables that occur more than once in the clause body (called *multiple* variables), even when they occur in the head. Different strategies for guiding the application of un-

fold/fold transformations are presented in order to eliminate such unnecessary variables. These strategies guarantee the complete elimination of unnecessary variables for a syntactically characterized subclass of definite programs.

Other significant related work is [14], which introduces an algorithm, called RAF, for eliminating *redundant arguments* from definite programs. Actually, RAF is intended as a post-processing phase for program transformers, since automatic transformation produces many redundant arguments. Usually, local variables appear in redundant arguments. Hence, local variable elimination can be performed through a combination of some program transformation method (for instance, conjunctive partial deduction) and the RAF algorithm. This kind of system is closer in spirit to [18] than our own method.

Acknowledgment: We would like to thank the anonymous referees for their valuable comments, which aided in improving the quality of this paper and in clarifying the presentation. This work was partially supported by the Spanish Project TIN2004-079250-C03-03.

References

1. J. Álvez and P. Lucio. An algorithm for local variable elimination in normal logic programs. Technical Report LSI/TR 10-2005, Basque Country University, 2005.
2. J. Álvez, 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*, pages 1468–1475, New York, NY, USA, 2004. ACM Press.
3. H. Andréka and I. Németi. The generalized completeness of Horn predicate-logic as a programming language. *Acta Cybern.*, 4:3–10, 1980.
4. K. R. Apt. Logic programming. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, pages 493–574. Elsevier, 1990.
5. R. Barbuti, P. Mancarella, D. Pedreschi, and F. Turini. A transformational approach to negation in logic programming. *J. Log. Program.*, 8(3):201–228, 1990.
6. P. Bruscoli, F. Levi, G. Levi, and M. C. Meo. Compilative constructive negation in constraint logic programs. In *CAAP '94: Proceedings of the 19th International Colloquium on Trees in Algebra and Programming*, pages 52–67, London, UK, 1994. Springer-Verlag.
7. D. Chan. An extension of constructive negation and its application in coroutining. In *NACLP*, pages 477–493, 1989.
8. K. L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 293–322, New York, 1978. Plenum Press.
9. P. Dembinski and J. Maluszynski. And-parallelism with intelligent backtracking for annotated logic programs. In *SLP*, pages 29–38, 1985.
10. W. Drabent. Do logic programs resemble programs in conventional languages? In *SLP*, pages 289–396, 1987.
11. W. Drabent. What is failure? an approach to constructive negation. *Acta Inf.*, 32(1):27–29, 1995.
12. M. Hanus. On extra variables in (equational) logic programming. In *ICLP*, pages 665–679, 1995.
13. K. Kunen. Negation in logic programming. *J. Log. Program.*, 4(4):289–308, 1987.

14. M. Leuschel and M. H. Sørensen. Redundant argument filtering of logic programs. In J. Gallagher, editor, *Logic Program Synthesis and Transformation, Proceedings of LoPSTr '96, Stockholm, Sweden*, Lecture Notes in Computer Science 1207, pages 83–103. Springer-Verlag, 1996.
15. M. J. Maher. Correctness of a logic program transformation system. Technical Report RC 13496, IBM T.J. Watson Research Center, 1988.
16. C. S. Mellish. The automatic generation of mode declarations for prolog programs. Technical Report 163, Dept. of Artificial Intelligence, University of Edinburgh, Scotland, 1981.
17. A. Pettorossi and M. Proietti. Transformation of logic programs. In D. M. Gabbayand, C. J. Hogger, and J. A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming, Vol. 5*, pages 697–787. Oxford University Press, 1998.
18. M. Proietti and A. Pettorossi. Unfolding - definition - folding, in this order, for avoiding unnecessary variables in logic programs. *Theor. Comput. Sci.*, 142(1):89–124, 1995.
19. D. A. Rosenblueth. Chart parsers as proof procedures for fixed-mode logic programs. In *FGCS*, pages 1125–1132, 1992.
20. T. Sato and H. Tamaki. Transformational logic program synthesis. In *FGCS*, pages 195–201, 1984.
21. T. Sato and H. Tamaki. Existential continuation. *New Generation Comput.*, 6(4):421–438, 1989.
22. L. Sterling and E. Shapiro. *The art of Prolog: advanced programming techniques.* MIT Press, Cambridge, MA, USA, 1986.
23. P. J. Stuckey. Negation and constraint logic programming. *Inf. Comput.*, 118(1):12–33, 1995.
24. S.-Å. Tärnlund. Horn clause computability. *BIT*, 17(2):215–226, 1977.
25. H. C. Wasserman, K. Yukawa, and Z. Shen. An alternative transformation rule for logic programs. In *SAC '95: Proceedings of the 1995 ACM symposium on Applied computing*, pages 364–368, New York, NY, USA, 1995. ACM Press.