# A Finite Presentation Theorem for Approximating Logic Programs

*(Extended Abstract)*

Nevin Heintze
*School of Computer Science*
*Carnegie Mellon University*
*Pittsburgh, PA 15213*

Joxan Jaffar
*IBM T.J. Watson Research Center*
*PO Box 218*
*Yorktown Heights, NY 10598*

**Summary**
The notion of cartesian closure on a set of unifiers has been used to define approximations of the least models of logic programs. Such approximations, often called types, are not known to be recursive. In this paper, we use cartesian closure to define a similar, but more accurate, approximation. The main result proves that our approximation is not only recursive, but that it can be finitely represented in the form of a cyclic term graph. This explicit representation can be used as a starting point for logic program analyzers.

## 1 Introduction

The problem at hand is: given a logic program, obtain an approximation of its meaning, that is, obtain an approximation of its least model. The definition of the approximation should be declarative (so that results can be proved about soundness and relative completness); the approximation should be accurate (so that it is meaningful in program analysis), and finally, the approximated meaning should be a recursive set.

The meaning of a logic program is defined using the fundamental notions of unification (representing the primitive computation step) and that of collecting unifiers (representing non-determinism). To define an approximation to this meaning, it suffices to define the corresponding notions of approximate unification and that of approximate collection of unifiers. The relevant area in logic programming is that of type inference, in which there are two main approaches. In one, the approximation is based on ignoring pointwise dependencies between two sets of values in two different argument positions[1]. The other approach is based on abstract interpretation; approximation here is based on treating several values as one abstract value.

In [8], [9] and [12], which exemplify the first approach, one can view that regular unification is used (there is no abstraction on the underlying domain), but that the collection of unifiers is approximated by means of a *cartesian closure* operator which enlarges collections of unifiers. Considering unifiers as equations, a collection of unifiers defines a relation (tuples). One can now view that the cartesian closure operator augments a collection of unifiers in such a way that the resulting relation is the (tuple-)cartesian closure of the original relation. For example, the operator augments the two unifiers $[X/a, Y/b], [X/c, Y/d]$ with $[X/a, Y/d], [X/c, Y/b]$.

Intuitively, using cartesian closure corresponds to reasoning about a program using its variables

[1] This approximation is also important, for efficiency reasons, in imperative languages [6].

as ranging over sets as opposed to individual values. It is for this reason, in fact, that the induced approximation is used to formalize types in logic programs. This is despite the fact that there are no known algorithms for inferring such types. That is, the approximate meanings of programs are not known to be recursive.

The approach using abstract interpretation, on the other hand, deals with an abstraction of the underlying domain which then induces an approximation of unifiers and collections of unifiers which is *finitary*[2,3]. (See, for example, [2].) The approximate meaning of a program, therefore, is automatically recursive. Its accuracy is not directly comparable to that of cartesian closure based approximations; it is, however, limited[4] by the essentially finite bound on the number of different states between which the approximate reasoning can discriminate.

In this paper, we define an approximation based only on cartesian closure. We then show that it is more accurate than other approximations which are based on cartesian closure. Our main result proves that our approximation to any program is recursive. Because our approximation is perfectly accurate for programs whose head atoms are linear (no two occurrences of the same variable), and whose body atoms each contain at most one distinct variable which appears elsewhere in the rule, we obtain a new subclass of logic programs which is decidable. This strengthens the result of [1] which used a main result of [10] to prove, amongst other results, that unary logic programs are decidable. The most important aspect of our proof, however, is that it is constructive: it provides an algorithm to represent the approximate meaning in the form of a cyclic term graph. This explicit representation can then be used as a starting point for logic program analyzers.

[2] This need not mean a finite abstract domain.

[3] For efficiency reasons, many abstract interpretation algorithms, e.g. [3], also use cartesian closure.

[4] Abstract interpreters have been useful, however, for inferring *runtime* properties.

## 2 Background & Related Work

The meaning of a logic program may be defined as the least fixpoint of the function $T_P$ which maps from and into interpretations. This function may be defined as follows[5]:

$$T_P(\mathcal{I}) \stackrel{\text{def}}{=} \left\{ \begin{array}{ll} H\beta : & H \longleftarrow \tilde{B} \in P, \text{ and} \\ & \beta \in \{\theta : \tilde{B}\theta \subseteq \mathcal{I}\} \end{array} \right\}$$

where $\theta$ ranges over all substitutions which are ground with respect to $var(P)$, the variables in $P$. Our definition of the approximate meaning of a program will be defined using a similar function.

A *set-substitution* $\alpha$ is like an ordinary substitution except that variables are mapped into sets of terms as opposed to terms. Furthermore, $X\alpha = \{X\}$ except for a finite number of variables. Now define an approximation operator $\mathcal{A}_{\tilde{X}}$ which takes as input a nonempty collection $\Theta$ of substitutions which are ground w.r.t. the (possibly empty set of) variables $\tilde{X}$. It outputs a single set-substitution $\alpha$ such that for every variable $X$, $X\alpha$ is:

$$\left\{ \begin{array}{ll} \{t : X/t \text{ appears in some } \theta \in \Theta\} & \text{if } X \in \tilde{X}; \\ \{X\} & \text{if } X \notin \tilde{X}. \end{array} \right.$$

We say that $\mathcal{A}_{\tilde{X}}$ is undefined when applied to an empty set. Our approximation to the meaning of a logic program $P$ can now be formalized by an adaptation of $T_P$ above. Let $\tilde{X}$ denote $var(P)$; then:

$$\tau_P(\mathcal{I}) \stackrel{\text{def}}{=} \left\{ a : a \in H\alpha \; \left| \begin{array}{l} \text{where} \\ H \longleftarrow \tilde{B} \in P, \text{ and} \\ \alpha = \mathcal{A}_{\tilde{X}}(\{\theta : \tilde{B}\theta \subseteq \mathcal{I}\}) \\ \text{is defined.} \end{array} \right. \right\}.$$

Thus while the least fixpoint of $T_P$, call it $T_P \uparrow \omega$, is the exact meaning of $P$, the least fixpoint of $\tau_P$, call it $\tau_P \uparrow \omega$, is our approximation of the meaning of $P$. Clearly $\tau_P \uparrow \omega \supseteq T_P \uparrow \omega$ for any $P$, and so our approximation is conservative.

Take, for example, the program following pro-

[5] The symbol $\sim$ groups atoms or terms into a possibly empty sequence or set, depending on context.

gram $P$:

```
p(a, b).
p(b, a).
q(X, Y) ⟵ p(X, Y).
```

Here $\tau_P(\emptyset)$, call it $\mathcal{I}$, is simply $\{p(a,b),\ p(b,a)\}$. To compute $\tau_P(\mathcal{I})$, we first note that $\{\theta : p(X,Y)\theta \subseteq \mathcal{I}\}$ contains two substitutions $[X/a, Y/b]$ and $[X/b, Y/a]$. Applying $\mathcal{A}_{X,Y}$ to this collection, we obtain the set subsitution $\alpha = [X/\{a,b\},\ Y/\{a,b\}]$ which, when applied to the head atom $q(X,Y)$, gives rise to $\{q(a,a),\ q(a,b),\ q(b,a),\ q(b,b)\}$. Thus $\tau_P(\mathcal{I}) = \{p(a,b),\ p(b,a),\ q(a,a),\ q(a,b),\ q(b,a),\ q(b,b)\}$, and this, in fact, equals $\tau_P \uparrow \omega$. In contrast, $T_P \uparrow \omega$ contains only four elements.

In the literature, the following cartesian closure operator $\star$ has been used over sets of terms and atoms rather than sets of substitutions: for any set $S$ of ground terms or atoms,

$$S^\star \stackrel{\text{def}}{=} \begin{array}{l} \{c : c \text{ is a constant in } S\} \ \cup \\ \{f(t_1,\ \ldots,\ t_n) : t_i \in (f^{-1}_{(i)}(S))^\star\} \end{array}$$

where $f$ ranges over the function and predicate symbols in $S$, and $f^{-1}_{(i)}(S) \stackrel{\text{def}}{=} \{t_i : f(t_1, \cdots, t_n) \in S\}$ denotes a projection of $S$. The work on types [12], for example, deals with the following function:

$$Y_P(\mathcal{I}) \stackrel{\text{def}}{=} \left\{ H\beta : \begin{array}{l} H \longleftarrow \tilde{B} \in P, \text{ and} \\ \beta \in \{\theta : \tilde{B}\theta \subseteq \mathcal{I}\} \end{array} \right\}^\star$$

and so the approximate meaning of the program $P$ is the least fixpoint of $Y_P$, call it $Y_P \uparrow \omega$. It is easy to see that $Y_P$ is less accurate than $\tau_P$. For example, program $P$ consisting of just two facts `p(a, b)` and `p(b, a)` is such that $Y_P \uparrow \omega = \{p(a,a), p(a,b), p(b,a), p(b,b)\} \supset T_P \uparrow \omega$ whereas $\tau_P \uparrow \omega = T_P \uparrow \omega$. The decidability of $Y_P \uparrow \omega$ was left open.

The earlier work in [8] associates to a program, not a function, but a formula in which variables range over sets. The approximate meaning of a program is then given by a certain model of these formulas. It turns out that this model is essentially a fixpoint of $Y_P$. The accuracy of the program approximation that this model defines is therefore bounded by $Y_P \uparrow \omega$. In [8] and in a subsequent paper [9], algorithms were presented for solving subclasses of these set based formulas. The solvability problem for the general class was left open.

We conclude this section by illustrating the differences between $Y_P$, $\tau_P$ and $T_P$. Consider the following program $P$:

```
p(f(a, b)).
p(f(b, a)).
r(X) ⟵ p(f(X, X)).
s(f(Y, Z)) ⟵ p(f(Y, Z)).
```

Here $Y_P \uparrow \omega$ is

$$\left\{ \begin{array}{l} p(f(a,a)), p(f(a,b)), p(f(b,a)), p(f(b,b)) \\ r(a), r(b) \\ s(f(a,a)), s(f(a,b)), s(f(b,a)), s(f(b,b)) \end{array} \right\}$$

while $\tau_P \uparrow \omega$ is

$$\left\{ \begin{array}{l} p(f(a,b)), p(f(b,a)) \\ s(f(a,a)), s(f(a,b)), s(f(b,a)), s(f(b,b)) \end{array} \right\}$$

and $T_P \uparrow \omega$ is

$$\left\{ \begin{array}{l} p(f(a,b)), p(f(b,a)) \\ s(f(a,b)), s(f(b,a)) \end{array} \right\}.$$

In general, we have $Y_P \uparrow \omega \supseteq \tau_P \uparrow \omega \supseteq T_P \uparrow \omega$ for any program $P$.

# 3 A Closed Form for Fixpoints

Our first step toward a finite representation of $\tau_P \uparrow \omega$ will be a quantified formula whose least model equals $\tau_P \uparrow \omega$. Before we define this formula, we require some preliminary definitions.

Consider the denumerable alphabets $\Sigma$ of function symbols, $\Pi$ of predicate symbols, and $\mathcal{V}$ of variables. Let $HB$ and $HU$ respectively denote the Herbrand universe and base associated with these alphabets. A *$HB$-set expression* is defined to be either:

- a special variable, not in $\mathcal{V}$, of the form $\Upsilon_p$ where $p \in \Pi$ (call these the *type variables*), or
- of the form $p(t_1, \ldots, t_n)$ where $p \in \Pi$ and $t_1, \ldots, t_n$ are $HU$-set expressions, or
- of the form $t \cup s$ or $t \cap s$ where $t$ and $s$ are $HB$-set expressions.

A *$HU$-set expression* is defined to be either:

- a variable $X$ in $\mathcal{V}$, or a special constant of the form $\bot$ or $\top$, or
- of the form $f(t_1, \ldots, t_n)$ where $f \in \Sigma$ and $t_1, \ldots, t_n$ are $HU$-set expressions, or
- of the form $t \cup s$ or $t \cap s$ where $t$ and $s$ are $HU$-set expressions.

Finally, a *set expression* is either a $HB$-set expression or a $HU$-set expression.

A *set constraint* is of the form $s = t$ where either both $s$ and $t$ are $HB$-set expressions or both $s$ and $t$ are $HU$-set expressions[6]. An *interpretation* $\mathcal{I}$ maps each type variable into $HB$ and each variable in $\mathcal{V}$ into $HU$. $\mathcal{I}$ can be extended, in an obvious way, to map set expressions as follows. If $t$ is:

- the special constant $\bot$, then $\mathcal{I}(t)$ is $\emptyset$;
- the special constant $\top$, then $\mathcal{I}(t)$ is $HU$;
- a constant symbol $c$, then $\mathcal{I}(t)$ is $\{c\}$;
- $f(t_1, \ldots, t_n)$ where $n \geq 1$ and $f \in \Pi \cup \Sigma$, then $\mathcal{I}(t)$ is $\{f(s_1, \cdots, s_n) : s_1 \in \mathcal{I}(t_1), \cdots, s_n \in \mathcal{I}(t_n)\}$;
- $u \cup s$ or $u \cap s$, then $\mathcal{I}(t)$ is $\mathcal{I}(u) \cup \mathcal{I}(s)$ or $\mathcal{I}(u) \cap \mathcal{I}(s)$ respectively.

This mapping, in turn, induces a mapping from collections $\mathcal{C}$ of set constraints into $\{true, false\}$ in the obvious way. If $\mathcal{I}(\mathcal{C})$ is *true*, we say that $\mathcal{I}$ is a *model* of $\mathcal{C}$.

[6] The solvability problem for set constraints is open. A decision procedure for a restricted class appears in [5].

Let $P$ be an arbitrary program and denote its rules by $R_1, \cdots, R_N$, $1 \leq N$. We assume that rules are written so that they do not share variables. Let $\Pi_P$ denote the predicate symbols in $P$. Let $P|_p$ denote the subset of rules in $P$ defining the relation $p$. Let $R_i$ be of the form $H_i \longleftarrow \tilde{B}_i$, $1 \leq i \leq N$. We are now in a position to define, for each program $P$, a formula which represents the approximate meaning of $P$. This formula is comprised of two main parts. The first part is a collection of set equations, $\mathcal{C}$, and is defined as follows:

$$\bigwedge_{p \in \Pi_P} \Upsilon_p = \bigcup_{i:R_i \in P|_p} H_i \ .$$

That is, each equation in $\mathcal{C}$ equates a type variable $\Upsilon_p$ and the union of the head atoms in $P|_p$.

The second part of our formula is a conjunction of several subformulas, one for each rule in $P$. Consider the following collection $\mathcal{D}_i$ of pairs corresponding to a rule $R_i$ in $P$:

$$\{(p(\tilde{t}), \Upsilon_p) : p(\tilde{t}) \text{ is a body atom in } R_i\}.$$

The subformula corresponding to $R_i$, call it $\mathcal{F}(\mathcal{D}_i)$, will be defined in terms of the pairs $\mathcal{D}_i$. Before we formally define $\mathcal{F}(\mathcal{D}_i)$, consider some example programs. Let $P$ be

```
p(a).
p(b).
q(X) ⟵ p(X).
```

so that $\mathcal{C}$ is $\Upsilon_p = p(a) \cup p(b) \wedge \Upsilon_q = q(X)$, $\mathcal{D}_1 = \mathcal{D}_2 = \emptyset$, and $\mathcal{D}_3$ is the singleton $\{(p(X), \Upsilon_p)\}$. In this example, if $\mathcal{F}(\mathcal{D}_1)$ and $\mathcal{F}(\mathcal{D}_2)$ are simply *true* and $\mathcal{F}(\mathcal{D}_3)$ is the set equation $p(X) = \Upsilon_p$, then the formula

$$\mathcal{C} \wedge \mathcal{F}(\mathcal{D}_1) \wedge \mathcal{F}(\mathcal{D}_2) \wedge \mathcal{F}(\mathcal{D}_3)$$

captures $\tau_P \uparrow \omega$ in the sense that its least model (in which $X = \{a, b\}$) is equal to $\tau_P \uparrow \omega$. In general, however, obtaining a subformula $\mathcal{F}(\mathcal{D}_i)$ by simply making the pairs $\mathcal{D}_i$ into set equations will not be appropriate. Suppose $P$ were

```
p(a).
p(b).
r(a).
r(c).
q(X) ←— p(X), r(X).
```

Then $\mathcal{C}$ is $\Upsilon_p = p(a) \cup p(b) \wedge \Upsilon_r = r(a) \cup r(c) \wedge \Upsilon_q = q(X)$, $\mathcal{D}_1 = \mathcal{D}_2 = \mathcal{D}_3 = \mathcal{D}_4 = \emptyset$, and $\mathcal{D}_5 = \{(p(X), \Upsilon_p), (q(X), \Upsilon_q)\}$. For this example, an appropriate formulation of the subformulas $\mathcal{F}(\mathcal{D}_1)$, $\mathcal{F}(\mathcal{D}_2)$, $\mathcal{F}(\mathcal{D}_3)$ and $\mathcal{F}(\mathcal{D}_4)$ is simply that they are each *true*, and an appropriate formulation of $\mathcal{F}(\mathcal{D}_5)$ is

$$X = \{x : (p(x) \in \Upsilon_p \wedge r(x) \in \Upsilon_r)\}\,.$$

The least model of the formula $\mathcal{C} \wedge \mathcal{F}(\mathcal{D}_1) \wedge \mathcal{F}(\mathcal{D}_2) \wedge \mathcal{F}(\mathcal{D}_3) \wedge \mathcal{F}(\mathcal{D}_4) \wedge \mathcal{F}(\mathcal{D}_5)$ (in which $X = \{a\}$) thus captures $\tau_P \uparrow \omega$.

The above two examples only had only one variable in the body atoms. Now suppose $P$ were

```
p(a, b).
p(c, d).
q(X,Y) ←— p(X,Y).
```

then $\mathcal{C}$ is $\{\Upsilon_p = p(a,b) \cup p(c,d) \wedge \Upsilon_q = q(X,Y)\}$, $\mathcal{D}_1 = \mathcal{D}_2 = \emptyset$, and $\mathcal{D}_3 = \{(p(X,Y), \Upsilon_p)\}$. As above, the subformulas $\mathcal{F}(\mathcal{D}_1)$ and $\mathcal{F}(\mathcal{D}_2)$ can each be assigned *true*. An appropriate formulation of $\mathcal{F}(\mathcal{D}_3)$ is a conjunction of two equations:

$$\begin{array}{rcl} X & = & \{x : \exists y\,(p(x,y) \in \Upsilon_p)\}\,, \\ Y & = & \{y : \exists x\,(p(x,y) \in \Upsilon_p)\} \end{array}$$

so that the least model of our formula $\mathcal{C} \wedge \mathcal{F}(\mathcal{D}_1) \wedge \mathcal{F}(\mathcal{D}_2) \wedge \mathcal{F}(\mathcal{D}_3)$ (in which $X = \{a, c\}$ and $Y = \{b, d\}$) captures $\tau_P \uparrow \omega$.

In general, $\mathcal{D}_i$ will be a collection of pairs of the form $\{(a_1, s_1), \cdots, (a_m, s_m)\}$ such that for all $1 \leq j \leq m$,

- $a_j$ is an atom or term, and
- $s_j$ is a set expression.

Letting $\mathcal{V} = \{X_1, \cdots, X_k\}$ denote the variables in $a_1, \cdots, a_m$, we define $\mathcal{F}(\mathcal{D}_i)$ to be[7] the conjunction, over $1 \leq j \leq k$, of:

$$X_j = \left\{ x_j : \exists x_1 ... x_{j-1} x_{j+1} ... x_k \left( \begin{array}{l} a_1\mu \in s_1 \\ \vdots \\ a_m\mu \in s_m \end{array} \right) \right\}$$

where $x_1, \cdots, x_k$ are new variables, and $\mu$ denotes the renaming substitution $[X_1/x_1, \cdots\ X_k/x_k]$.

An *interpretation* and *model* of such a formula are defined by extending, in the obvious way, the definitions given in section 2 for an interpretation and a model of set equations.

We are now in position to define formally, given a program $P$, a formula whose least model characterizes the desired set $\tau_P \uparrow \omega$. In order to keep this definition as simple as possible, we assume that programs have no ground atoms in the head of any non-fact rule; call such restricted programs *non-degenerate*. To see that no generality is in fact lost, consider a program $P$ which contains such offending rules. Rewrite $P$ into $P'$ by adding a new variable, as an extra argument in the first position, to every atom in $P$. Clearly $\tau_P \uparrow \omega$ can be easily extracted from $\tau_{P'} \uparrow \omega$ by simply ignoring the first argument of all atoms in the latter set. We shall deal with degenerate programs at the end of this paper.

Let $R_1, \cdots, R_N$ denote the rules in the rules in $P$, and we now define that the formula corresponding to $P$ is

$$\mathcal{C} \wedge \mathcal{F}(\mathcal{D}_1) \wedge \cdots \wedge \mathcal{F}(\mathcal{D}_N)$$

where $\mathcal{C}$ is

$$\bigwedge_{p \,\in\, \Pi_P} \Upsilon_p = \bigcup_{i : R_i \in P|_p} H_i$$

and each $\mathcal{D}_i$, $1 \leq i \leq N$, is the collection of pairs

$$\{(p(\tilde{t}), \Upsilon_p) : p(\tilde{t}) \text{ is a body atom in } P\} \bigcup \left\{(X,\ \top) : X \text{ appears in } H_i \text{ but not in } \tilde{B}_i\right\}.$$

[7] If $\mathcal{D}_i$ is empty, then $\mathcal{F}(\mathcal{D}_i)$ is simply *true*.

For example, consider the program:

```
p(a, b).
p(b, a).
q(X, W) ⟵ p(X, X).
r(Y, Z) ⟵ p(Y, Z).
```

$\mathcal{C}$ is $\Upsilon_p = p(a,b) \cup p(b,a) \wedge \Upsilon_q = q(X,W) \wedge \Upsilon_r = r(Y,Z)$, $\mathcal{D}_1$ is $\emptyset$, $\mathcal{D}_2$ is $\emptyset$, $\mathcal{D}_3$ is $\{(p(X,X), \Upsilon_p), (W, \top)\}$ and $\mathcal{D}_4$ is $\{(p(Y,Z), \Upsilon_p)\}$. The formula $\mathcal{C} \wedge \mathcal{F}(\mathcal{D}_1) \wedge \mathcal{F}(\mathcal{D}_2) \wedge \mathcal{F}(\mathcal{D}_3) \wedge \mathcal{F}(\mathcal{D}_4)$ is therefore given by:

$$
\begin{array}{lrcl}
(4a) & \Upsilon_p & = & p(a,b) \cup p(b,a), \\
(4b) & \Upsilon_q & = & q(X,W), \\
(4c) & \Upsilon_r & = & r(Y,Z), \\
(4d) & X & = & \left\{ x : \exists w \left( \begin{array}{rcl} p(x,x) & \in & \Upsilon_p \\ w & \in & \top \end{array} \right) \right\}, \\
(4e) & W & = & \left\{ w : \exists x \left( \begin{array}{rcl} p(x,x) & \in & \Upsilon_p \\ w & \in & \top \end{array} \right) \right\}, \\
(4f) & Y & = & \{ y : \exists z \, (p(y,z) \in \Upsilon_p) \}, \\
(4g) & Z & = & \{ z : \exists y \, (p(y,z) \in \Upsilon_p) \}.
\end{array}
$$

where equations (4a), (4b) and (4c) arise from the definition of $\mathcal{C}$, equations (4d) and (4e) arise from $\mathcal{D}_3$, and equations (4f) and (4g) arise from $\mathcal{D}_4$. The one and only model for this formula is given by

$$
\begin{array}{rcl}
\Upsilon_p & = & \{p(a,b), p(b,a)\}, \\
\Upsilon_q & = & \emptyset, \\
\Upsilon_r & = & \{r(a,a), r(b,b), r(a,b), r(b,a)\}, \\
X & = & \emptyset, \\
W & = & \emptyset, \\
Y & = & \{a, b\}, \\
Z & = & \{a, b\}.
\end{array}
$$

In what follows, we shall abbreviate the sequence $\mathcal{D}_1, \cdots, \mathcal{D}_N$, where $N$ denotes the number of rules in the program, by $\tilde{\mathcal{D}}$, and the formula $\mathcal{F}(\mathcal{D}_1) \wedge \cdots \wedge \mathcal{F}(\mathcal{D}_N)$ by $\mathcal{F}(\tilde{\mathcal{D}})$. Where $\mathcal{I}$ is a model of a collection of set equations, we write $\mathcal{I}_{\Upsilon_P}$ to denote the subset of $HB$ defined by the union of $\mathcal{I}(\Upsilon_p)$ over all the type variables $\Upsilon_p$ associated with $P$. We now formally state that the models of the formula $\mathcal{C} \wedge \mathcal{F}(\tilde{\mathcal{D}})$ correspond to the fixpoints of $\mathcal{T}_P$.

**Main Lemma 1**
*Let $P$ be a non-degenerate logic program. Then*
*(a) $\mathcal{I}$ is a model of $\mathcal{C} \wedge \mathcal{F}(\tilde{\mathcal{D}})$ implies that $\mathcal{I}_{\Upsilon_P}$ is a fixpoint of $\mathcal{T}_P$;*
*(b) $\mathcal{I}'$ is a fixpoint of $\mathcal{T}_P$ implies that there is a model $\mathcal{I}$ of $\mathcal{C} \wedge \mathcal{F}(\tilde{\mathcal{D}})$ such that $\mathcal{I}_{\Upsilon_P} = \mathcal{I}'$.* □

# 4 A Simplification Algorithm for Set Constraints

We shall deal with the formula $\mathcal{F}_P(\tilde{\mathcal{D}})$ in the next section. In this section, we describe an algorithm, called $SIMPLIFY$, which inputs a collection of set constraints in the form $\{X_1 = s_1, X_2 = s_2, \cdots X_n = s_n\}$ where the $X_i$ are distinct variables and the $s_i$ are set expressions, into an equivalent system written in a form such that the least model is apparent.

For simplicity, we assume that the set expressions $s_i$ are in disjunctive normal form w.r.t. $\cup$ and $\cap$. We write $DNF(s)$, where $s$ is a set expression, to denote the equivalent expression obtained by applying the usual distributive operation on $\cap$ over $\cup$ so that the result is in $DNF$. Clearly such a function is easily implementable. We tacitly assume that the symbols $\cup$ and $\cap$ are associative, commutative and idempotent. Thus, for example, we may regard $s \cap (t \cap u)$ and $(s \cap t) \cap u$ as identical expressions. We shall also assume that the set expression $s_i$ are not deeply nested[8]. We lose no generality because we can rewrite any conjunction $\mathcal{C}$ of set equations by exhaustively applying the following transformation: if $\mathcal{C}$ contains a subexpression $f(t_1, \cdots, t_n)$ where $t_i$, $1 \leq i \leq n$, is not a variable, then replace that occurrence of $t_i$ by a new variable $X$ and add to $\mathcal{C}$ the equation $X = t_i$. Clearly the resulting collection is equivalent to the original collection and is not deeply nested.

We now define formally the input and output format of the $SIMPLIFY$ algorithm.

[8] The term nesting means encapsulation within the scope of a function or predicate symbol.

**Definition 1 (Normal Form)** *A collection of set equations $\mathcal{C}$ is in* normal form *if it is (a) not deeply nested, (b) of the form $X_1 = s_1$, $X_2 = s_2$, $\cdots$, $X_n = s_n$ where the $X_i$, $1 \leq i \leq n$, are distinct, and (c) the $s_i$, $1 \leq i \leq n$, are set expressions.* □

Given a collection $\mathcal{C} = \{X_1 = s_1, X_2 = s_2, \cdots, X_n = s_n\}$ of set equations in normal form, we refer to the variables $X_1, \cdots, X_n$ as the *solved variables* of $\mathcal{C}$. We write $lhs(\mathcal{C})$ to denote the set of variables on the left-hand-sides of equations in $\mathcal{C}$, and we write $rhs(\mathcal{C})$ to denote the set of expressions appearing in the right-hand-sides. We write $rhs(\mathcal{C})|_X$ to denote

$$\begin{cases} \text{s} & \text{if } X = s \text{ appears in } \mathcal{C}; \\ \text{X} & \text{otherwise.} \end{cases}$$

**Definition 2 (Solved Form)** *A normal form collection of set equations $\mathcal{C}$ is in* solved form *if it is of the form:*

$$\begin{aligned} X_1 &= s_{1,1} \cup s_{1,2} \cup \cdots \cup s_{1,m_1} \\ &\vdots \\ X_n &= s_{n,1} \cup s_{n,2} \cup \cdots \cup s_{n,m_n} \end{aligned}$$

*where for each $1 \leq i \leq n$ and $1 \leq j \leq m_i$, the set expression $s_{i,j}$ is either (a) a constant, or (b) of the form $f(Y_1, \cdots, Y_n)$ where $f$ is an $n$-ary function or predicate symbol and each $Y_i, 1 \leq i \leq n$, is a variable, or (c) a non-solved variable, or (d) of the form $X \cap s$ where $X$ is a non-solved variable.* □

Where $X = s_1 \cup \cdots \cup s_n$ is an equation in a solved form collection $\mathcal{C}$, we write $crhs(\mathcal{C})$ to denote the subset of the expressions in $rhs(\mathcal{C})$ which fall into cases (a) and (b) above. Similarly, we define $crhs(\mathcal{C})|_X$ to be subset of the expressions in $rhs(\mathcal{C})|_X$ which fall into cases (a) and (b) above.

Despite the above seemingly complicated definition of solved form, the main property that is required from constraints $\mathcal{C}$ in solved form is simply that its least model (and it is easy to show this exists) can be immediately extracted. More specifically, the least model of $\mathcal{C}$ can be expicitly presented by replacing all occurrences of non-solved variables by $\perp$ (since in the least model they are the empty set) and simplifying appropriately. What will remain is a collection in the following form:

**Definition 3 (Explicit Form)** *A collection of set equations $\mathcal{C}$ is in* explicit form *if it is of the form:*

$$\begin{aligned} X_1 &= s_{1,1} \cup s_{1,2} \cup \cdots \cup s_{1,m_1} \\ &\vdots \\ X_n &= s_{n,1} \cup s_{n,2} \cup \cdots \cup s_{n,m_n} \end{aligned}$$

*where for $1 \leq i \leq n$, the $X_i$ are distinct variables, and for each $1 \leq j \leq m_i$, the set expression $s_{i,j}$ is either (a) a constant, or (b) of the form $f(Y_1, \cdots, Y_n)$ where $f$ is an $n$-ary function or predicate symbol and each $Y_i, 1 \leq i \leq n$, is a solved variable.* □

For example, let the solved form collection $\mathcal{C}$ be

$$\begin{aligned} &X = f(Y) \cup a, \\ &Y = g(X) \cup (h(X) \cap Z). \end{aligned}$$

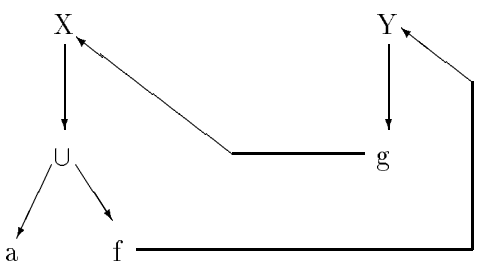
We may describe the least model of $\mathcal{C}$, which is:

$$\begin{aligned} X &= \{a, fg(a), \cdots (fg)^n(a), \cdots\}, \\ Y &= \{g(a), gfg(a), \cdots (gf)^n g(a), \cdots\} \end{aligned}$$

by the explicit form:

$$\begin{aligned} &X = f(Y) \cup a, \\ &Y = g(X). \end{aligned}$$

We may also construct the following term graph as an alternative representation:




We now present the $SIMPLIFY$ algorithm which is essentially a controlled sequence of applications

of a number of transformations. Consider first the following two transformation steps on a collection $\mathcal{C}$ in normal form:

**Transformation 1 ($\mathcal{C}$-Substitute)** *If $\mathcal{C}$ contains two distinct equations $X = s$, $Y = t$ such that $s$ contains an un-nested occurrence of the variable $Y$, then replace that occurrence of $Y$ by $t$ and apply $DNF$ appropriately.*

**Transformation 2 (Cycle-Removal)** *If $\mathcal{C}$ contains the equation $X = s$, such that $s$ contains an un-nested occurrence of the variable $X$, then replace that occurrence of $X$ by a new variable.*

Essentially, the result of repeatedly applying these transformations is that $\mathcal{C}$ will contain no un-nested solved variables on the right-hand-sides. While these two transformations serve to the implement obvious notion of substitution, there are two noteworthy points: first, they cannot be applied in an arbitrary order, and second, the latter transformation is not intuitive.

We next subject $\mathcal{C}$ to the following three transformations.

**Transformation 3 (Intersection Clash)** *If $\mathcal{C}$ contains an occurrence of the expression $t \cap s$ where $t$ and $s$ have distinct outermost symbols, replace that occurrence by $\bot$.*

**Transformation 4 (Intersect$^{\mathbf{n}}$ Simplification)** *If $\mathcal{C}$ contains an occurrence of the expression $t \cap \bot$ or $t \cap \top$, then replace that occurrence by $\bot$ or $t$ respectively.*

**Transformation 5 (Intersection Distribution)** *If $\mathcal{C}$ contains an occurrence of the expression $f(X_1, \cdots, X_n) \cap f(Y_1, \cdots, Y_n)$, then replace that occurrence by $f(V_1, \cdots, V_n)$, and add the constraint $V_i = DNF(rhs(\mathcal{C})|_{X_i} \cap rhs(\mathcal{C})|_{Y_i})$, $1 \leq i \leq n$, where the $V_i$ are new variables.*

**input** a normal form collection $\mathcal{C}$;
1. **rename** each variable $X$ in $\mathcal{C}$ into $V_{\{X\}}$;
2. **for** each solved variable $X$ in $\mathcal{C}$ **do**
3. **exhaustively-apply**
transformation 2 to $\mathcal{C}$;
4. **exhaustively-apply**
transformation 1 using $X$ to $\mathcal{C}$;
**endfor**
5. **repeat**
6. **exhaustively-apply**
transformations 3 and 4 to $\mathcal{C}$;
7. **if** transformation 5 is applicable to $\mathcal{C}$
that is, $\mathcal{C}$ contains a subexpression
of the form
$f(V_{S_1}, V_{S_2}, \cdots, V_{S_n}) \cap f(V_{T_1}, V_{T_2}, \cdots, V_{T_n})$
8. **then**
replace the above subexpression by
$f(V_{S_1 \cup T_1}, \cdots, V_{S_n \cup T_n})$;
9. **for** each $i : 1 \leq i \leq n$ **do**
10. **if** $V_{S_i \cup T_i}$ is non-solved **then**
11. add to $\mathcal{C}$ the equation
$V_{S_i \cup T_i} =$
$DNF(rhs(\mathcal{C})|_{V_{S_i}} \cap rhs(\mathcal{C})|_{V_{T_i}})$;
**endif**
**endfor**
**endif**
12. **until** none of transformations 3, 4 and 5
is applicable to $\mathcal{C}$;
**rename** each variable $V_{\{X\}}$, $X \in \mathcal{V}$, into $X$;
**output** $\mathcal{C}$;

Figure 1: **SIMPLIFY**

The entire $SIMPLIFY$ algorithm is presented in figure 1.

The correctness of the individual transformations, and the fact that we obtain a solved form upon termination of $SIMPLIFY$, can be easily verified. Termination, however, cannot be guaranteed if these transformations were applied in an arbitrary order. (For example we can apply the intersection distribution transformation to the constraint $X = f(X) \cap f(X)$ forever.) To obtain termination, we showed, in [4], that the number of new variables which need to be introduced by the intersection transformation can be bounded by $2^{|\mathcal{V}|}$ where $\mathcal{V}$ denote the variables in $\mathcal{C}$. The details are lengthy and hence omitted.

In summary,

**Lemma 1** *Let $\mathcal{C}$ denote a normal form collection of set constraints. Then the SIMPLIFY algorithm, when input with $\mathcal{C}$, outputs a collection $\mathcal{C}_s$ of set constraints such that*

- *$\mathcal{C}_s$ is equivalent to $\mathcal{C}$, and*
- *$\mathcal{C}_s$ is in solved form.* □

# 5 The Main Result

In section 3, we defined, corresponding to a given program $P$, a formula consisting of a normal form collection $\mathcal{C}$ of set equations, and a formula $\mathcal{F}(\tilde{\mathcal{D}})$ where $\tilde{\mathcal{D}}$ is a family of collections of pairs. We now present our main algorithm which inputs such a formula and performs a finite number of reduction steps

$$\mathcal{C},\ \tilde{\mathcal{D}} \Longrightarrow \mathcal{C}^{(0)},\ \tilde{\mathcal{D}}^{(0)} \Longrightarrow \ldots \Longrightarrow \mathcal{C}^{(n)},\ \tilde{\mathcal{D}}^{(n)}$$

and finally obtains a collection of solved form set equations $\mathcal{C}^{(n)}$ such that the least model of $\mathcal{C}^{(n)}$ is the same as the least model of the formula $\mathcal{C} \wedge \mathcal{F}(\tilde{\mathcal{D}})$. The essence of the algorithm is that it progressively transfers information about the least model we seek from $\tilde{\mathcal{D}}^{(i)}$ to $\mathcal{C}^{(i)}$, $0 \leq i \leq n$. In the final step, therefore, $\mathcal{C}^{(n)} \wedge \mathcal{F}(\tilde{\mathcal{D}}^{(n)})$ will be such that all the desired information will be in $\mathcal{C}^{(n)}$.

The basis of this algorithm, as for the $SIMPLIFY$ algorithm described previously, is a group of transformations. Each transformation below takes two items as input: (a) a collection of set equations $\mathcal{C}$ and (b) a family $\tilde{\mathcal{D}} = \{\mathcal{D}_1, \cdots, \mathcal{D}_N\}$ of collections of pairs. Each transformation outputs two items: (a) a family $\tilde{\mathcal{D}}'$ of collections of pairs, and (b) a collection $\delta$ of set equations in normal form. If a transformation is applied on $\mathcal{C}$ and $\tilde{\mathcal{D}}$ to obtain $\tilde{\mathcal{D}}'$ and $\delta$, we write

$$\mathcal{C}, \tilde{\mathcal{D}} \Longrightarrow \tilde{\mathcal{D}}', \delta$$

and we say that a transformation was, in the *context* of $\mathcal{C}$, *applied* on $\tilde{\mathcal{D}}$ to obtain the the new family $\tilde{\mathcal{D}}'$ and an *extracted collection* $\delta$ of set equations.

**Transformation 6 ($\tilde{\mathcal{D}}$-Substitution)** *If $\tilde{\mathcal{D}}$ contains an un-nested occurrence of a solved variable $Y$ in the right-hand-side of a pair, then replace the occurrence by $rhs(\mathcal{C})|_Y$ and apply DNF appropriately. Output the resulting $\tilde{\mathcal{D}}$, and $\delta = \emptyset$.* □

**Transformation 7 (Clash)** *If $\mathcal{D}_j \in \tilde{\mathcal{D}}$ contains a pair $(f(t_1, \cdots, t_n), g(s_1, \cdots, s_m))$ where $f \neq g$, or a pair $(s, \perp)$, then output $\tilde{\mathcal{D}} - \{\mathcal{D}_j\}$ and $\delta = \{Y_1 = \perp, \cdots, Y_k = \perp\}$ where $Y_1, \cdots, Y_k$ are the variables in the left-hand-sides of $\mathcal{D}_j$.* □

**Transformation 8 (Cancellation)** *If $\tilde{\mathcal{D}}$ contains a pair $(f(s_1, \cdots, s_n), f(t_1, \cdots, t_n))$, then replace the pair by the pairs $(s_1, t_1), \cdots, (s_n, t_n)$. Output the resulting $\tilde{\mathcal{D}}$ and $\delta = \emptyset$.* □

Define the function $NONEMPTY(\mathcal{C}, s)$, where $\mathcal{C}$ is a collection of set equations in solved form and $s$ is a set expression, to be *true* iff $\mathcal{C} \models s \neq \emptyset$. Clearly applying $NONEMPTY$ to $(\mathcal{C}, s)$ is equivalent to applying it to

$$(SIMPLIFY(\mathcal{C} \cup \{X = s\}), X)$$

where $X$ is a new variable. Thus the problem of implementing the function $NONEMPTY$ can be reduced to the problem of determining if a given variable is always non-empty in a given solved form. Such a test, detailed in [4], can be easily implemented.

**Transformation 9 (Compaction)** *If $\mathcal{D}_j \in \tilde{\mathcal{D}}$ is of the form $(\tilde{X}, \tilde{\Phi})$ and if $NONEMPTY(C, s_{Y_i})$ for all $1 \leq i \leq k$ where*

- *$\tilde{X}$ is a family of variables, and $var(\tilde{X}) = \{Y_1, \cdots, Y_k\}$;*
- *the notation $s_Y$, where $Y$ ranges over $Y_1, \cdots, Y_k$, denotes the set expression $\phi_1 \cap \cdots \cap \phi_m$ such that $(Y, \phi_1), \cdots, (Y, \phi_m)$ are the pairs in $\mathcal{D}_j$ of the form $(Y, \phi)$,*

*then output* $\tilde{\mathcal{D}} - \{\mathcal{D}_j\}$ *and* $\delta = \{Y_1 = s_{Y_1}, \cdots, Y_k = s_{Y_k}\}$. □

**Transformation 10 (Union)** *If* $\mathcal{D}_j \in \tilde{\mathcal{D}}$ *is of the form* $\{(s, t_1 \cup t_2), (\tilde{\Psi}, \tilde{\Phi})\}$ *then*

- *let* $Y_1, \cdots, Y_k$ *be the variables in* $s$ *and* $\tilde{\Psi}$*;*
- *let* $Y_1', \cdots, Y_k', Y_1'', \cdots, Y_k''$ *be distinct new variables and let* $\eta_1$ *and* $\eta_2$ *denote the renaming substitutions* $[Y_1/Y_1', \cdots, Y_k/Y_k']$ *and* $[Y_1/Y_1'', \cdots, Y_k/Y_k'']$*;*
- *let* $\tilde{\mathcal{D}}'$ *be* $\tilde{\mathcal{D}} - \{\mathcal{D}_j\} \cup \{\mathcal{D}_j', \mathcal{D}_j''\}$ *where* $\mathcal{D}_j' = \{(s\eta_1,\ t_1), (\tilde{\Psi}\eta_1,\ \tilde{\Phi})\}$ *and* $\mathcal{D}_j'' = \{(s\eta_2,\ t_2), (\tilde{\Psi}\eta_2,\ \tilde{\Phi})\}$*, and*
- *let* $\delta$ *be* $\{Y_1 = Y_1\eta_1 \cup Y_1\eta_2, \cdots, Y_k = Y_k\eta_1 \cup Y_k\eta_2\}$*;*

*and output* $\tilde{\mathcal{D}}'$ *and* $\delta$. □

**Transformation 11 (Intersection)** *If* $\mathcal{D}_j \in \tilde{\mathcal{D}}$ *contains a pair* $(s, t_1 \cap t_2)$*, then replace that pair by the two pairs* $(s, t_1)$*,* $(s, t_2)$*. Output the resulting* $\tilde{\mathcal{D}}$ *and* $\delta = \emptyset$. □

Whenever one of the above transformations is applied on $\tilde{\mathcal{D}}$ in the context of $\mathcal{C}$ to obtain $\delta$ and $\tilde{\mathcal{D}}'$, we write $\mathcal{C}, \tilde{\mathcal{D}} \overset{\text{tf}}{\Longrightarrow} \tilde{\mathcal{D}}, \delta$ to indicate by $tf$ the particular transformation used. We say that $\tilde{\mathcal{D}}$ is *closed* under a transformation if that transformation is not applicable to $\tilde{\mathcal{D}}$.

Intuitively, these transformations fall into three classes. The cancel, union and intersection transformations simplify the collections $\mathcal{D}_j$ independently of $\mathcal{C}$. The substitution transformation, on the other hand, modifies $\mathcal{D}_j$ using $\mathcal{C}$. The clash and compact transformations perform the essential work of transferring information to $\mathcal{C}$ by augmenting it with new constraints. Crucial to the termination, correctness and completeness of this system of transformations is the synchronization between the compaction transformation, which contains a semantic condition, and the others (but mainly the substitution transformation). Specifically, while the semantic condition ensures correctness, we need to prove that it is indeed satisfied when necessary.

```
function REDUCE(C, D̃) {
    Δ := ∅;
 1. repeat {
 2.     while (the substitution transformation
                is applicable to C, D̃) do {
                C, D̃ ⟹(D̃ = subs) D̃, δ
                Δ := Δ ∪ δ;
        }
 3.     while (the union or int^n transformation
                is applicable to C, D̃) do {
                C, D̃ ⟹union D̃, δ or
                C, D̃ ⟹intersect D̃, δ accordingly;
                Δ := Δ ∪ δ;
        }
 4.     while (the cancel or clash transformation
                is applicable to C, D̃) do {
                C, D̃ ⟹cancel D̃, δ or
                C, D̃ ⟹clash D̃, δ accordingly;
                Δ := Δ ∪ δ;
        }
 5.     while (the compaction transformation
                is applicable to C, D̃) do {
                C, D̃ ⟹compact D̃, δ
                Δ := Δ ∪ δ;
        }
    } until (no transf^n is applicable to C, D̃);
 6. return (D̃, Δ);
}
```

Figure 2: **REDUCE**

We are now in a position to define, in figure 2, the reduction step of our algorithm. This step is described in the form of a function whose essential operation is an exhaustive application of the above transformations in the context of a fixed collection $\mathcal{C}$ of set contraints. For the sake of simpler proofs, however, we have restricted the order in which transformations may be applied.

Our main algorithm can now be described in figure 3. In [4], we proved, in two main parts, that

**input** an non-degenerate logic program $P$;
1. **construct** $\mathcal{C}, \tilde{\mathcal{D}}$ as defined in section 3;
2. **rewrite** $\mathcal{C}$ into normal form;
3. $\mathcal{C}^{(0)} := SIMPLIFY(\mathcal{C})$;
4. $\mathcal{V}_0 := var(\mathcal{C}^{(0)})$;
   $i := 0$;
5. **repeat** {
6. $\quad (\tilde{\mathcal{D}}^{(i+1)},\ \Delta) := REDUCE(\mathcal{C}^{(i)},\ \tilde{\mathcal{D}}^{(i)})$;
7. $\quad \mathcal{C}^{(i+1)} := SIMPLIFY(\mathcal{C}^{(i)} \cup \Delta)$;
   $\quad i := i + 1$;
8. } **until**
   $\quad (crhs(\mathcal{C}^{(i+1)})|_X = crhs(\mathcal{C}^{(i)})|_X$
   $\quad$ for all $X \in \mathcal{V}_0$);
   **output** $\mathcal{C}^{(i)}$;

Figure 3: **Main Algorithm**

$REDUCE(\mathcal{C}, \tilde{\mathcal{D}})$ terminates. The proof that the main algorithm terminates then followed.

**Lemma 2 (Termination of REDUCE)**
*(a) Each of the four while-loops terminates.*
*(b) The main repeat-loop terminates.* □

**Main Lemma 2 (Termination)**
*The main algorithm terminates.* □

The next property that is required of these transformations is that they are correct. This notion of correctness has two aspects. First, we require that the transformations preserve the least model. That is

**Lemma 3 (Partial-Correctness of Transformations)** *If* $\mathcal{C}, \tilde{\mathcal{D}} \Longrightarrow \tilde{\mathcal{D}}', \delta$, *then* $\mathcal{C} \wedge \mathcal{F}(\tilde{\mathcal{D}})$ *is equivalent to* $(\mathcal{C} \cup \delta) \wedge \mathcal{F}(\tilde{\mathcal{D}}')$.

While this is relatively easy to show for the substitution, clash, cancel and intersection transformations the proofs for the compaction and union transformations are more involved. In the case of the compaction transformation, this is because there is a non-trivial transfer of information from

**input** an arbitrary logic program $P$;
1. rewrite $P$ by systematically replacing each
   $\quad$ atom $p(t_1, \cdots, t_n)$, $n \geq 0$, by the atom
   $\quad$ $p(Z, t_1, \cdots, t_n)$, where $Z$ is a new variable
   $\quad$ each time, and call the result $P'$;
2. **input** $P'$ to the main algorithm,
   $\quad$ and let the output be $\mathcal{C}'$;
3. **rewrite** $\mathcal{C}'$ by systematically replacing each
   $\quad$ atom $p(t_1, t_2, \cdots, t_{n+1})$, $n \geq 1$, by the atom
   $\quad$ $p(t_2, \cdots, t_{n+1})$, and call the result $\mathcal{C}$;
4. **simplify** $\mathcal{C}$ into explicit form;
   **output** $\mathcal{C}$;

Figure 4: **Final Algorithm**

$\mathcal{D}_j$ to $\mathcal{C}$. In the case of the union transformation, this is because the transformation itself is complicated.

The second aspect of the correctness argument is more involved. Essentially, it consists of establishing an invariant that is preserved by each transformation application. The purpose of this invariant is to show that upon termination, we have actually computed the least fixpoint of $\tau_P$. In other words, this invariant, in conjunction with the termination condition

$$crhs(\mathcal{C}^{(n+1)})|_X = crhs(\mathcal{C}^{(n)})|_X \text{ for all } X \in \mathcal{V}_0,$$

must imply that the least model of $\mathcal{C}^{(n)}$ is equivalent to the least model of the original formula $\mathcal{C} \wedge \mathcal{F}(\tilde{\mathcal{D}})$. A major complication is that the least model of $\mathcal{C}^{(n)}$ is in general not a model of the final formula $\mathcal{C}^{(n)} \wedge \mathcal{F}(\tilde{\mathcal{D}}^{(n)})$. The details are left to [4].

Before formally stating the main result, we present, in figure 4, a simple extension to the main algorithm to deal with arbitrary programs. This algorithm formalizes the comments made in section 3 concerning degenerate programs.

We now state the main result of this paper which is that the final algorithm computes an explicit representation of $\tau_P \uparrow \omega$ for an arbitrary program $P$.

**Theorem:** Let $P$ be an arbitrary logic program input to the final algorithm. Then

- the final algorithm terminates, returning a formula $\mathcal{C}$;
- $\mathcal{C}$ is a collection of set equations in explicit form, and
- where $\mathcal{I}$ is the least model of $\mathcal{C}$, $\mathcal{I}|_{\Upsilon_P} = \tau_P \uparrow \omega$.

□

# 6 Conclusion

Using the notion of cartesian closure on a collection of unifiers we defined an approximation to the meaning of a logic program. We argued that our approximation is more accurate than similar approximations in the literature, and this literature primarily being that on type inference in logic programming. The main result, that our approximation of a program is a recursive and finitely presentable set, was then presented in the form of an algorithm which has two parts.

The first part of our algorithm, called $SIMPLIFY$, was a decision procedure for a class of set equations, that is, set equations in normal form. Earlier related work is that of Reynolds [11] and Jones and Muchnick [7], the latter being an independent treatise of work similar to the former. In these works, the set expressions were allowed to contain projection functions, but they were not allowed to contain the intersection function (these works were not concerned with problem of shared variables inherent in logic programs). Thus their transformations mainly served to eliminate occurrences of projection functions while ours mainly served to eliminate occurrences of the intersection function[9].

The second and main part of our algorithm is also based upon a group of transformations. When used directly, these transformations do not always lead to a terminating computation. The key aspect of the main algorithm, therefore, is in its controlled use of the transformations. This is realized by the function REDUCE each of whose invocations performed transformations local to a fixed collection of set equations. The main loop in the main algorithm then ensured that REDUCE is invoked sufficiently often.

We finally note that two important practical issues have yet to be addressed: the first is efficiency; second, our algorithm does not, in general, return a *minimal* representation of $\tau_P \uparrow \omega$. Early investigations into the former issue reveal that our algorithm will be practical when applied to a large class of programs. The latter issue can be dealt with by algorithms similiar to those used for minimizing finite-state automata.

[9] In [5], we deals with *both* projection and intersection.

# Acknowledgements

The authors are grateful to Saumya Debray, Thom Fruehwirth, Michael Maher, Peter Stuckey and Eyal Yardeni who reviewed the complete version of this paper.

# References

[1] D. Angluin and D.N. Hoover, "Regular Prefix Relations" *Mathematical Systems Theory* 17, pp 167 - 191, 1984.

[2] M. Bruynooghe, "A Practical Framework for the Abstract Interpretation of Logic Programs", revised version of report CW-62, Dept. Comp. Sci., K.U. Leuven, 33 pages, October 1987.

[3] M. Bruynooghe and G. Janssens, "An Instance of Abstract Interpretation integrating Type and Mode Inference", *Proceedings* $5^{th}$ *Int. Conf. on Logic Programming*, K.A. Bowen and R.A. Kowalski (eds), MIT Press, pp 669 - 683, August 1988.

[4] N.C. Heintze and J. Jaffar, "A Finite Presentation Theorem for Approximating Logic Programs", forthcoming CMU and IBM Research Report, 56 pages, October 1989.

[5] N.C. Heintze and J. Jaffar, "A Decision Procedure for a Class of Herbrand Set Constraints",

forthcoming CMU and IBM Research Report, 40 pages, October 1989.

[6] N.D. Jones and S.S. Muchnick, "Complexity of Flow Analysis, Inductive Assertion Synthesis, and a Language due to Dijkstra", in *Proceedings* $20^{th}$ *Conference on Foundations of Computer Science*, pp 185 - 190, also in, *Program Flow Analysis: Theory and Applications*, N.D. Jones and S.S. Muchnick (eds), Prentice-Hall, 1981.

[7] N.D. Jones and S.S. Muchnick, "Flow Analysis and Optimization of LISP-like Structures", *Proceedings* $6^{th}$ *ACM Symposium on Principles of Programming Languages*, pp 244 - 256, 1979.

[8] P. Mishra, "Toward a Theory of Types in PROLOG", *Proceedings* $1^{st}$ *IEEE Symposium on Logic Programming*, Atlantic City, pp 289 - 298, 1984.

[9] P. Mishra and U.S. Reddy, "Declaration-free Type Checking", *Proceedings* $12^{th}$ *ACM Symposium on Principles of Programming Languages*, New Orleans, pp 7 - 21, 1985.

[10] M.O. Rabin, "Decidability of Second-order Theories and Automata on Infinite Trees", *Transactions of the American Math. Society* 141, pp 1 - 35, 1969.

[11] J.C. Reynolds, "Automatic Computation of Data Set Definitions", *Information Processing 68*, North-Holland, pp 456 - 461, 1969.

[12] E. Yardeni and E.Y. Shapiro, "A Type System for Logic Programs", in *Concurrent PROLOG: Collected Papers*, Vol. 2, MIT Press, pp 211 - 244, 1987.