DEJA VU IN FIXPOINTS OF LOGIC PROGRAMS

by

Michael J. Maher
and
Raghu Ramakrishnan

Computer Sciences Technical Report #893

November 1989

Déjà Vu in Fixpoints of Logic Programs

Michael J. Maher

IBM T.J. Watson Research Center,

P.O. Box 704, Yorktown Heights, NY 10598, U.S.A.

Raghu Ramakrishnan *

Department of Computer Sciences,

University of Wisconsin-Madison, WI 53706, U.S.A.

November 8, 1989

Abstract

We investigate properties of logic programs that permit refinements in their fixpoint evaluation and shed light on the choice of control strategy. A fundamental aspect of a bottom-up computation is that we must constantly check to see if the fixpoint has been reached. If the computation iteratively applies all rules, bottom-up, until the fixpoint is reached, this amounts to checking if any new facts were produced after each iteration. Such a check also enhances efficiency in that duplicate facts need not be re-used in subsequent iterations, if we use the Seminaive fixpoint evaluation strategy. However, the cost of this check is a significant component of the cost of bottom-up fixpoint evaluation, and for many programs the full check is unnecessary. We identify properties of programs that enable us to infer that a much simpler check (namely, whether any fact was produced in the previous iteration) suffices. While it is in general undecidable whether a given program has these properties, we develop techniques to test sufficient conditions, and we illustrate these techniques on some simple programs that have these properties.

*The work of R. Ramakrishnan was supported in part by an IBM Faculty Development Award and NSF grant IRI-8804319. Part of the work was done while visiting IBM Thomas J. Watson Research Center.

1 Introduction

Recent work in the deductive database area suggests that logic programs may be efficiently evaluated by first applying optimizing program transformations and then evaluating the fixpoint of the rewritten program bottom-up [4, 5, 7, 12, 16, 18, 19, 22] It is generally proposed that the fixpoint be computed using the Seminaive algorithm [3, 6, 2], which avoids repeating inferences. However, this method requires that the new facts computed in each iteration be compared with previously generated facts to eliminate duplicates, and this is a costly operation, especially in the presence of non-ground tuples. (Eliminating duplicates now involves subsumption checks rather than equality checks.)

In this paper, we examine Seminaive and related algorithms closely, and identify properties of programs that enable us to replace expensive checks for duplicates by simpler checks. In particular, subsumption-freedom — informally, the property that if two facts are generated then neither subsumes the other — allows us to simply check whether any facts at all were produced in the previous iteration, and to avoid the significant additional cost of checking if any of these facts is subsumed by other (previously) generated facts.

We show that the class of subsumption-free programs is quite large; in particular, every partial recursive function can be computed by such a program. Further, common Prolog programs such as *reverse* and *append* are subsumption-free. Intuitively, it appears that the property holds for a large class of deterministic programs.

We show that it is in general undecidable to determine whether these properties hold for a given program. However, we develop techniques to prove that programs have the subsumption-freedom property. These techniques are sufficiently powerful to deal with the programs considered in this paper, including a class of programs that compute all partial recursive functions. The fibonacci program, presented below, is illustrative of subsumption-freedom and the techniques needed to establish it.

Example 1.1 Consider the fibonacci program:

$r1 : fib(I, N) :- \quad I \geq 2, fib(I1, N1), fib(I2, N2),$
$\qquad\qquad\qquad\qquad plus(I1, 1, I), plus(I2, 2, I), plus(N1, N2, N).$

$r2 : fib(0, 0).$

$r3 : fib(1, 1).$

This program is subsumption-free, based on the following observations: No fact can be produced by more than one rule. Further, consider the first rule. Given a head fact, the instance of the body that generated it is uniquely determined. (To show this, we must establish the functional dependency $fib_1 \rightarrow fib_2$, over the relation for fib computed by the program. This is an inductive proof, and we consider formal techniques to establish such dependencies later in this paper.) Also note that only ground facts can be generated. Thus, every fact is produced by a unique rule, through a unique instantiation of that rule. It follows that there is a unique derivation tree for each fact, and so no fact that is produced in a Seminaive evaluation is subsumed by another; an observation that permits us to avoid the cost of duplicate elimination. (The Not-So-Naive evaluation strategy that is described later is essentially Seminaive evaluation without a subsumption check.) □

The program properties identified in this paper are also useful in identifying classes of programs for which Prolog's evaluation strategy is complete, and in studying the choice of different control strategies that are now viable alternatives in logic program evaluation [15].

The rest of this paper is organized as follows. In Section 2, we extend parts of the theory of logic programs to deal with multisets rather than sets, which is necessary for our study of duplicates. We define a class of bottom-up fixpoint evaluation algorithms that we study in later sections, and present definitions of some important algorithms in this class. In Section 3, we define derivation trees and extend the multiset-based approach of Section 2 to deal with trees. We also characterize the fixpoint algorithms described in Section 2 in terms of the trees that they generate. In Section 4, we introduce some important properties of programs, namely subsumption-freedom, duplicate freedom, finite-subsumption, and finite-forest. We show how these properties enable us to use simpler fixpoint algorithms. In Section 5, we prove that these properties are undecidable. We also show that all partial recursive functions can be computed by programs that have the most restrictive of the properties that we study, namely subsumption-freedom. In Section 6, we present necessary and sufficient conditions for subsumption-freedom and duplicate-freedom. While these cannot be effectively tested, they allow us to develop testable sufficient conditions, and we do this in Section 7. In doing so, we extend the notion of functional dependencies to relations that contain non-ground tuples, and study a class of relations whose tuples correspond to rule instantiations.

2 Bottom-Up Fixpoint Evaluation

The treatment of bottom-up evaluation of logic programs (or Datalog programs) in the literature has considered the evaluation as a computation of relations, and hence has used sets to describe the process [24, 3, 6, 2, 1]. Since our interest is in the occurrence of duplicate atoms (tuples) during the computation, we first extend portions of the existing theory to use multisets of atoms, rather than sets. We also extend most treatments by generating non-ground atoms in a manner similar to [9]. This involves a further extension, since we must now consider the possibility that newly generated atoms might be subsumed by previously generated atoms.

We denote by $ground(X)$ the set of ground instances of elements of X. We assume the existence of at least one constant, so every syntactic construct has a ground instance. We use a partial function $mgu(t_1, t_2)$ which returns a most general unifier of the terms t_1 and t_2, if the terms are unifiable. We say that $a \leq b$ under the subsumption ordering if $a = b\theta$ for some substitution θ. If A and B are multisets then $a \leq B$ iff $\exists b \in B\ a \leq b$, and $A \leq B$ iff $\forall a \in A\ a \leq B$. We denote by $maxims(X)$ the set of maximally general elements of X under the subsumption ordering. In addition to sets and multisets, we will also use in this paper a restricted type of set, called an *irredundant set* or *irrset*. Irrsets are sets in which no element is subsumed by another. Union of irrsets X and Y is $maxims(X \cup Y)$. Subtraction of irrsets $X - Y$ is given by $\{x \mid x \in X, \forall y \in Y\ x \not\leq y\}$.

We now define multisets and operations on them in some detail, since we use them extensively and also use some non-standard operations (such as *col*, and a non-standard notion of multiset difference).

Definition 2.1 (Multisets) A multiset is a collection of elements that are not necessarily distinct. The number of occurrences of an element in a multiset is its *multiplicity* in the multiset. We denote the multiplicity of an element x in a multiset M by $mult(x, M)$. The cardinality $card(M)$ of a multiset M is the sum of the multiplicities of each element of M. A multiset is finite if its cardinality is finite. We define $x \in M$ iff $mult(x, M) > 0$. When a multiset is enumerated we use delimiting brackets. For example, $M = [a, b, b, c]$ is a multiset with $mult(b, M) = 2$.

Given a multiset M, $set(M)$ is the set of elements of M, that is, $set(M) = \{x \mid x \in M\}$. The union of two multisets M_1 and M_2, denoted as $M_1 \oplus M_2$, is a multiset in which the

multiplicity of an element is the sum of its multiplicities in M_1 and M_2. This is extended to obtain the union of an infinite chain of multisets as $S = \lim_{n\to\infty} S_n$, where the multiplicity of any s in S is the least upper bound — in $\mathcal{Z} \cup \{\infty\}$ — of the multiplicities of s in S_n. $M_1 \subseteq M_2$ denotes multiset containment: for every element $x \in M_1$, $mult(x, M_1) \leq mult(x, M_2)$.

The difference of two multisets M_1 and M_2, denoted as $M_1 - M_2$, is a multiset in which the multiplicity of an element x is $mult(x, M_1)$ if $mult(x, M_2) = 0$, and is 0 otherwise. Note that this differs from the usual definition of multiset difference, in which the multiplicity of an element in $X - Y$ is defined to be $max(m - n, 0)$, where m is its multiplicity in X and n is its multiplicity in Y. We have chosen a different definition in order to reflect our use of multisets. in which taking a difference corresponds to the operation of duplicate elimination.

We introduce a "coloring" operation on multisets that is useful for providing constructive definitions of multisets in terms of a defining property.

Definition 2.2 (Colored Sets) Let M be a multiset, and let C be an (infinite) ordered list of colors $c_1, c_2, \ldots$. For every element, say a, with multiplicity $n > 0$, color the copies of a with $c_1, c_2, \ldots c_n$, and denote these distinct colored elements as $a_1, a_2, \ldots, a_n$. The set containing all the elements obtained by thus coloring elements of M is a *colored set*, denoted as $col(M)$.

The inverse operation, $col^{-1}(S)$, is defined to yield a multiset M in which the multiplicity of an element a is equal to the number of colored copies of a in the colored set S.

Finally, we introduce a multiset constructor that allows us to define multisets from a set of tuples (a relation in the database sense) through projection on columns.

Definition 2.3 (Multiset Constructor $[\ldots]$) Let R be a set of n-tuples, and let $\bar{i}$ be a vector of k integers in the range $1 \ldots n$. For every n-tuple $\bar{t} \in R$, consider the k-tuple $(t_{i_1}, \ldots, t_{i_k})$. The multiset M denoted by $[\bar{i}|R]$ is defined to contain every such k-tuple m, with a multiplicity equal to the number of tuples $\bar{t}$ of R such that $m = (t_{i_1}, \ldots, t_{i_k})$.

Formally, each element of our multisets is an equivalence class — a class of all syntactic objects that are equivalent modulo variable renaming. (A variable renaming is a substitution θ that is a bijection on the set of all variables.) However we will (slightly less formally) treat our multisets as containing objects where variable names are not significant. Different elements, or different occurrences of the same element, will be represented by objects which have no common variables.

Throughout this paper we will assume a fixed program P. Readers who are accustomed to the practice, common in the deductive database literature, of viewing the program as distinct from the set of facts should note that in this paper, a program includes the set of facts as well.

We will use the following schema of a bottom-up fixpoint evaluation algorithm:

$$
\begin{aligned}
S_0 = \delta_0 &= \mathcal{F} \\
\delta_{n+1} &= f(S_n, \delta_n) \\
S_{n+1} &= g(S_n, \delta_{n+1}) \\
S &= \lim_{n\to\infty} S_n
\end{aligned}
$$

where $\mathcal{F}$ is the set of facts, or rules with empty bodies, in the program P, and f and g are some functions depending on the multisets S_n, δ_n and δ_{n+1}.

In order to give algebraic expression to some forms of differential bottom-up computation we introduce a special operator, $W(X,Y)$, defined on multisets X and Y where $Y \subseteq X$, that essentially captures the notion of a rule application:

$W(X,Y) = [1\ |R]$, where R is the set of tuples $(h\theta, r, d_1, \ldots, d_k)$ such that:

$h\ :-\ b_1, \ldots, b_k$ is a variant of a rule r of P, $k > 0$,

$d_1, \ldots, d_k \in col(X)$,

$\exists i \in 1 \ldots k$ such that $d_i \in col(Y)$, and

$\theta = mgu((b_1, \ldots, b_k), (col^{-1}(d_1), \ldots, col^{-1}(d_k)))$

Intuitively, W only allows deductions from X that use the "new" facts Y, and the multiplicity of a fact $(h\theta)$ is the number of times that it is deduced.

In our analysis we represent the current known facts as a set (or multiset or irrset) and assume that inference of new facts from this set is performed simultaneously by all rules in the program. Refinements of this analysis are possible for more flexible inference strategies, such as those of [13, 20, 1]. However we do not pursue this point further.

We now define the class of evaluation strategies that we consider, and some related terminology.

Definition 2.4

Let $\delta_n - S_{n-1} \subseteq Y_n \subseteq S_n$ *for* $n \geq 0$, *and* $S_{-1} = S_0 = \delta_0 = \mathcal{F}$.

$$\begin{aligned}
\delta_{n+1} &= dup_elim(W(S_n, Y_n)) \\
S_{n+1} &= dup_elim(S_n \oplus \delta_{n+1}) \\
S &= dup_elim(\lim_{n\to\infty} S_n) \\
GC &= set(S) \\
IGC &= maxims(S)
\end{aligned}$$

where $\mathcal{F}$ *is the set of facts in the program* P, dup_elim *is the identity function for multiset data structures,* set *for sets, and* maxims *for irrsets. The set* GC *is the set of* generated consequences *of the program under this evaluation, and* IGC *is the irrset of* irredundant generated consequences.

The algorithms are defined to terminate (at Step $n+1$) when $S_{n+1} = S_n$. Consequently an algorithm terminates iff S is finite. When the data structures are multisets it suffices to test whether $\delta_{n+1} = \emptyset$. When sets (irrsets) are used, we must test whether $\delta_{n+1} \subseteq S_n$ (respectively, $\delta_{n+1} \leq S_n$). We note that, in regard to early termination, irrsets are always (as good as or) better than sets, which are always (as good as or) better than multisets. In particular the choice of data structure — set, multiset, or irrset — can make the difference between termination and non-termination.

Example 2.1 Consider the following program.

$$\begin{aligned}
&p(s(X)) \quad :- p(X). \\
&p(X). \\
&q \qquad\quad :- q. \\
&q.
\end{aligned}$$

Evaluation of p will terminate only if irrsets are used. Similarly, evaluation of q will not terminate if multisets are used. □

We can verify the correctness of these evaluations with respect to the least Herbrand model M of P and some other semantics of P.

Proposition 2.1 *Consider evaluation strategies that fit into the scheme defined above.*

1. *For multiset and set computations, the generated consequences* GC *of a program are independent of the evaluation strategy. Furthermore,* $ground(GC) = M$.

2. *For multiset, set and irrset computations, the irredundant generated consequences* IGC *of a program are independent of the evaluation strategy. Furthermore,* $ground(IGC) = M$.

Proof: It can be shown by induction that for set and multiset data structures $set(S_n)$ is invariant over all evaluation strategies (we denote this set by GC_n), and for all data structures $maxims(S_n)$ is invariant over all evaluation strategies. □

GC is the minimal S-model of P as defined by [9]. It was recently shown [10, 11] that GC and IGC are fully abstract semantics for logic programs under an operational (respectively, declarative) definition of observable program behaviors.

We present algebraic definitions of some fixpoint evaluation strategies that are the focus of this paper, by instantiating the general schema. There are variants corresponding to algorithms that either check whether generated facts are subsumed by existing facts (using irrsets as the data structure), check whether generated facts are equal to existing facts (using sets), or do no such checking (using multisets). To distinguish different avatars of the same algorithm we will subscript the name of the algorithm with M, S or I when (respectively) multisets, sets or irrsets are used.

Definition 2.5 *Naive evaluation.*

$$\delta_{n+1} = dup\_elim(W(S_n, S_n))$$
$$S_{n+1} = dup\_elim(S_n \oplus \delta_{n+1})$$

In Naive evaluation (N), every possible derivation using the set of known facts is made in each iteration, until no new facts are generated. The set version of Naive evaluation is what is usually referred to by this name in the literature.

Definition 2.6 *Seminaive evaluation.*

$$\delta_{n+1} = dup\_elim(W(S_n, \delta_n - S_{n-1}))$$
$$S_{n+1} = dup\_elim(S_n \oplus \delta_{n+1})$$

The set version of Seminaive evaluation (SN) is usually proposed as the bottom-up fixpoint evaluation algorithm of choice. The set of facts produced in iteration n (δ_n) is compared with the set of already known facts (S_{n-1}) to identify the new facts produced ($\delta_n - S_{n-1}$). Only derivations that use one of these new facts are carried out in iteration $n + 1$. This avoids generating many duplicate facts by avoiding repeated derivations. In addition, duplicates generated in the same iteration are eliminated by the set data structure. However, the costs of identifying the new facts can sometimes outweigh the benefits of avoiding duplicates. With this in mind we introduce a new fixpoint evaluation strategy:

Definition 2.7 *Not-So-Naive evaluation.*

$$\delta_{n+1} = dup\_elim(W(S_n, \delta_n))$$
$$S_{n+1} = dup\_elim(S_n \oplus \delta_{n+1})$$

Not-So-Naive evaluation (NSN) relaxes the condition that every derivation of a fact must use a new fact, by only requiring that every derivation must use a most recently generated fact. In comparison with SN it does not avoid some redundant derivations since a newly generated occurrence of an already known fact will cause a (redundant) derivation. Nonetheless, it does avoid repeatedly deriving a fact from the same occurrences of known facts. Since in this paper we will only use the multiset version of NSN evaluation, we will use NSN for $\mathrm{NSN_M}$.

Example 2.2 The following program illustrates the difference between SN and NSN.

$d :- c.$

$c :- b.$

$b :- a.$

$a.$

$b.$

At the first step, both evaluation strategies generate c and a second occurrence of b. $\mathrm{SN_S}$ eliminates the duplicate b. In the next step both $\mathrm{SN_M}$ and $\mathrm{SN_S}$ generate only d, since c was the only new fact found at the previous step. In contrast NSN generates both d and a second occurrence of c, since both b and c were most recently generated. The final result of the computation for each strategy is $[a, b, b, c, c, d, d]$ for NSN, $[a, b, b, c, d]$ for $\mathrm{SN_M}$ and $[a, b, c, d]$ for $\mathrm{SN_S}$. NSN evaluation takes one more step to terminate than either version of SN. (To see the difference with respect to N, note that NSN, unlike N, does not generate b in the second step.) □

3 Derivation Trees

A derivation tree is a representation of a proof that $P \models a$, where a is the label of the root of the tree. Every node in a derivation tree for a corresponds to an inference in the proof of a.

Definition 3.1 *Let P be a program. We define derivation trees in P as follows:*

- *Every fact h in P is a derivation tree for itself, consisting of a single node with label h.*

- *Let* $h \ :- \ b_1, b_2, \ldots, b_k$ *be a new variant of a rule* r *in* P*, let* d_i*,* $i = 1, \ldots, k$ *be atoms with derivation trees* t_i*, and let* θ *be the mgu of* $(b_1, \ldots, b_k)$ *and* $(d_1, \ldots, d_k)$*. Then, the following is a derivation tree for* $h\theta$*: The root is a node labeled* $h\theta$*, and each* t_i*,* $i = 1 \ldots n$*, is a child of the root. Each arc from the root to a child has the label* r*.*

Note that the substitution θ is not applied to the children of $h\theta$ in the second part of the above definition. Thus, a derivation tree records which set of (previously generated) facts is used to generate a new fact using a rule, rather than the set of substitution instances of these facts that instantiated the rule. The height of a derivation tree is defined to be the number of nodes in the longest path (which is always from the root to a leaf).

Let $atoms(t)$ be the label on the root of derivation tree t, and extend this to a multiset of trees X by $atoms(X) = [1 \mid R]$ where R is the set of tuples $(atoms(t), t')$ such that $t \in X$, $t' \in col(X)$ and $t = col^{-1}(t)$. The function *atoms* abstracts the generated atoms from the derivation trees that generate them.

Definition 3.2 *Let* P *be a program and let* a *be a fact. The multiset of all derivation trees for* a *in* P *is denoted by* $DT(P, a)$*. The multiset of all derivation trees in* P *is* $DT(P) = \bigcup_a DT(P, a)$*.*

We can view the evaluation strategies described above as acting on derivation trees instead of atoms. With appropriate definitions for the operations used in an evaluation, the multiset of atoms S computed in an evaluation is simply an abstraction of the multiset of derivation trees S' computed by the same evaluation.

We redefine some of the operators used previously to behave differently on multisets of derivation trees. These definitions will be used only in this section. $set(X)$ denotes a maximal subset of X in which the label of every root node is different and $maxims(X)$ denotes a maximal subset of X in which no label of a root node is subsumed by the label of another. When the data structures are sets or multisets, $X - Y$ is the submultiset of X obtained by deleting from X all trees t such that $atoms(t) \in atoms(Y)$. When the data structures are irrsets, $X - Y$ is the subset of X obtained by deleting from X all trees t such that $atoms(t) \leq atoms(Y)$. The operation $\oplus$ remains the same for multisets of trees, but for sets and irrsets we take the union operator to be $X \cup (Y - X)$, where $Y - X$ is defined above for each data structure. The function corresponding to W is W':

$W'(X,Y) = [1\,|R|]$, where R is the set of tuples $(t, r, t_1, \ldots, t_k)$ such that:

$h :- b_1, \ldots, b_k$ is a variant of a rule r of P, $k > 0$,

$t_1, \ldots, t_k \in col(X)$,

$t, col^{-1}(t_1), \ldots, col^{-1}(t_k)$ are derivation trees with roots $h\theta, d_1, \ldots, d_k$,

the root of t has arcs labeled r and children $col^{-1}(t_1), \ldots, col^{-1}(t_k)$,

$\exists i \in 1 \ldots k$ such that $t_i \in col(Y)$, and

$\theta = mgu((b_1, \ldots, b_k), (d_1, \ldots, d_k))$

If we use the above operations and the function W', each evaluation strategy defines multisets S'_n and δ'_n of derivation trees.

Proposition 3.1 *For any evaluation strategy under definition 2.1 and any choice of data structures, $atoms(S'_n) = S_n$ and $atoms(\delta'_n) = \delta_n$.*

If X is a multiset of derivation trees, let $X|_{\leq n}$ ($X|_n$) denote the submultiset of X of trees of height less than or equal to n (exactly n). The following result shows that NSN evaluation produces all the proofs that can be generated from P. Combined with the above result, it characterizes the output of NSN evaluation.

Lemma 3.1 *Under NSN evaluation, $S'_n = DT(P)|_{\leq n}$ and $\delta'_n = DT(P)|_n$ for $n \geq 0$. Thus $S' = DT(P)$ when NSN evaluation is used.*

Proposition 3.2 *NSN evaluation generates $S = atoms(DT(P))$.*

Definition 3.3 *A collection of derivation trees in program P for a set of facts has the Seminaive property if the following holds:*

1. *For every non-root node in every tree: If the label of the node is a, then the height of the subtree rooted at this node is equal to the height of the smallest tree in $DT(P, a)$.*

2. *For every pair of non-root nodes - possibly in different trees - with the same label, the subtrees rooted at these nodes are identical. (Thus, whenever a label a is associated with a non-root node, the subtree rooted at this node is uniquely determined.)*

Note that there may be more than one collection of derivation trees (for a set of facts in a program P) with the Seminaive property. Intuitively, these differ in the subtree associated

with internal nodes that have a given label; however all subtrees associated with an internal node having a given label a must have the same height, which is the height of the smallest tree in $DT(P, a)$.

Proposition 3.3 *The set of derivation trees constructed in an SN_I evaluation of a program P has the Seminaive property.*

4 Some Useful Properties of Programs

In this section we present some properties that enable a more complex and computationally expensive evaluation strategy or data structure to be replaced by a simpler strategy and/or data structure. In particular, we can refine the well-known Seminaive evaluation strategy by replacing subsumption tests by the much simpler check for emptiness.

A program P is *subsumption-free* if $atoms(DT(P))$ is an irrset. A program P is *duplicate-free* if $atoms(DT(P))$ is a set. Essentially, P is subsumption-free if subsumption tests are unnecessary for some evaluation strategies. Similarly, P is duplicate-free if tests for duplicates (i.e. use of the set data structure) is unnecessary for some evaluation strategies.

Example 4.1 Consider the following program.

$p(X, Y) \quad :- X = 5.$

$p(X, Y) \quad :- Y = 5.$

This program is subsumption-free. However, the following program is not:

$q(X) \quad :- p(X, Y), X = 5, Y = 5.$

$p(X, Y) \quad :- X = 5.$

$p(X, Y) \quad :- Y = 5.$

The reader is invited to verify that the definition of subsumption-freedom presented in this paper makes this distinction correctly. (Seminaive evaluation and Not-So-Naive evaluation perform identically on the first program. However, the fact $q(5)$ is produced twice in the second program, and Seminaive evaluation would discard this fact when it is produced a second time, unlike Not-So-Naive evaluation.)

An alternative definition of subsumption-freedom that might seem natural is the following: A program is subsumption-free if for every ground fact p, there is a unique tree such that p unifies with the root. Under this definition, neither of the above programs is subsumption-free.

However, our intent is to capture the behavior of fixpoint evaluation algorithms. As discussed in [16], neither $p(5, Y)$ nor $p(X, 5)$ can be discarded since each represents facts not represented by the other, although $p(5, 5)$ is represented by both. □

It follows from Proposition 2.1 and Proposition 3.2 that P is subsumption-free (duplicate-free) iff $atoms(DT(P)) = IGC$ ($atoms(DT(P)) = GC$). In Section 6 we will present more detailed characterizations that are more amenable to practical approximation.

One measure of the cost of an evaluation is the number of inferences performed. It is clear that, for a fixed evaluation strategy, an algorithm using irrsets performs fewer (or the same) inferences than an algorithm using sets which, in turn, performs fewer (or the same) inferences than an algorithm using multisets. Similarly, for a fixed data structure, if $\delta_n - S_{n-1} \subseteq Y_n \subseteq Z_n$ for every $n \geq 0$ then an evaluation strategy using Y_n performs fewer (or the same) inferences than an evaluation strategy using Z_n. Thus, for example, $\mathrm{SN_I}$ performs fewer (or the same) inferences than NSN.

Theorem 4.1 *In terms of the number of inferences, SN_I = NSN for subsumption-free programs P.*

> **Proof:** The number of inferences made at step $n + 1$ is $card(W(S_n, Y_n))$. For NSN, $W(S_n, Y_n) = atoms(DT(P)|_{n+1})$ and so the total number of inferences made by NSN is $card(atoms(DT(P)))$, which is equal to $card(IGC)$. As observed earlier, $\mathrm{SN_I}$ performs fewer or the same number of inferences as NSN. However $\mathrm{SN_I}$ must perform at least as many inferences as $card(S) = card(IGC)$. Thus $\mathrm{SN_I}$ and NSN perform the same number of inferences. □

We can obtain several similar results.

Proposition 4.1 *In terms of the number of inferences,*

1. *SN_S = NSN for duplicate-free programs P.*
2. *$SN_I = SN_S$ for programs P where no element of GC strictly subsumes another.*

A multiset has *finite character* if every element has finite multiplicity. A multiset has the *finite subsumption* property if it has finite character and every element subsumes only

finitely many elements. A program P has the *finite forest* property if $atoms(DT(P))$ has finite character. A program P has the *finite subsumption* property if $atoms(DT(P))$ has the finite subsumption property.

Note that the following trivial rule in a program would destroy the finite subsumption and forest properties whenever the relation p in M is non-empty: $p(X) :- p(X)$.

Theorem 4.2 *If SN_I terminates, then NSN terminates, for programs P with the finite subsumption property.*

> **Proof:** If $\mathrm{SN_I}$ terminates then IGC is finite, and by Proposition 2.1 $maxims(S)$ is finite, where S is the set computed by the (possibly non-terminating) NSN evaluation. By Proposition 3.2, $S = atoms(DT(P))$. Hence, by the finite subsumption property, for every $a \in maxims(S)$ there are only finitely many $b \in S$ such that $b \leq a$, and each b has finite multiplicity. Thus S is finite, and so NSN terminates. □

With a similar proof we can obtain:

Proposition 4.2 *If SN_S terminates, then NSN terminates, for programs P with the finite forest property.*

5 Decision Problems

For Datalog programs, we can test whether the properties hold, although the test in general involves more work than evaluating the program.

Theorem 5.1 *For Datalog programs P, it is decidable whether P*

1. *is subsumption-free.*
2. *is duplicate-free.*
3. *has the finite forest property.*
4. *has the finite subsumption property.*

Proof: SN_S evaluation of Datalog programs is guaranteed to terminate. Suppose it terminates at step n. Consider the multisets S_n and S'_m in a NSN evaluation, where $m = 1 + card(set(S_n))$. We have the following characterizations of the properties, justified below:

- P is subsumption-free (duplicate-free) iff S_n is an irrset (a set).
- P has the finite forest property iff there is no tree in S'_m in which the label of the root also labels an interior node, modulo variable renaming.
- For Datalog programs, the finite subsumption and finite forest properties are equivalent.

It follows from Proposition 3.1 and Lemma 3.1 that $IGC \subseteq GC \subseteq S_{n-1}$. Using this, P is duplicate-free (subsumption-free) iff $S = GC$ ($S = IGC$) iff NSN evaluation also terminates at Step n and S_{n-1} is a set (irrset) iff S_n is a set (irrset).

If P does not have the finite forest property then some atom has infinite multiplicity in $atoms(DT(P))$. It follows that δ'_m has an element t (of height m). Since some branch of t has m nodes, there must be an atom a that occurs at least twice in this branch. The subtree rooted at the highest such occurrence must appear in S'_m. Conversely, if S'_m contains a tree t in which the label a of the root also labels an interior node (modulo variable renaming), then we can construct infinitely many trees for a as follows: We can replace the subtree of t rooted at the interior node by a copy of t (under an appropriate variable renaming) to obtain a larger tree t' for a. Similarly we can replace the subtree by t' to get t'' and so on. □

This proof extends to any class of programs for which SN_S always terminates (i.e. GC is finite) except for Part 3 (the finite subsumption property).

Sebelik and Stepanek [21] showed that every partial recursive function can be expressed as a logic program, by encoding standard rules (e.g. [26]) for defining partial recursive functions into definite clauses. We use their encoding scheme to prove that all partial recursive functions can be expressed as subsumption-free logic programs.

Theorem 5.2 *[21] Every partial recursive function can be computed by a logic program constructed using only predicate definitions of the following form:*

- $f_z(X, 0)$.
- $f_s(X, s(X))$.
- $f_{m,i}(X_1, \ldots, X_m, X_i)$.
- $f_{compose}(\bar{X}, Z) \ :- \ g_1(\bar{X}, Y_1), \ldots, g_k(\bar{X}, Y_k), h(Y_1, \ldots, Y_k, Z)$.
- $f_{pr}(0, X_2, \ldots, X_n, Z) \ :- \ g(X_2, \ldots, X_n, Z)$.
  $f_{pr}(s(X_1), X_2, \ldots, X_n, Z) \ :- \ f_{pr}(X_1, \ldots, X_n, Y), h(X_1, \ldots, X_n, Y, Z)$.
- $f_{min}(\bar{X}, Z) \ :- \ g(\bar{X}, 0, Y), r(\bar{X}, 0, Y, Z)$.
  $r(\bar{X}, U, 0, U)$.
  $r(\bar{X}, U, s(V), Z) \ :- \ g(\bar{X}, s(U), Y), r(\bar{X}, s(U), Y, Z)$.

where $g, g_1, \ldots, g_k, h$ are previously defined predicates.

For each partial recursive n-ary function f there is program P in the above form with least model M defining a $n+1$-ary predicate F where $f(a_1, \ldots, a_n) = b$ iff $F(a_1, \ldots, a_n, b) \in M$. A query q is said to be satisfiable by a program P if the existential closure of q holds in the least model of P. Clearly satisfiability of queries for the class of programs defined above is undecidable, by reduction of the "halting problem" for partial recursive functions.

Lemma 5.1 *Consider programs defined according to the above scheme. If $a, b \in GC$, $a \neq b$, then a and b do not unify. Furthermore, no duplicates are generated in NSN evaluation.*

Proof: Predicates defined according to the above scheme are partially ordered such that p precedes q if p is used to define q. Our proof proceeds by induction on this partial order. Let the result hold for all predicates preceding p (Induction Hypothesis 1). We prove that it also holds for p.

We note that if a and b are generated by different rules, they cannot unify, since no two rule heads unify. Suppose that they are generated from the same rule. We proceed by cases according to the scheme. All cases are straightforward, except for composition ($f_{compose}$), primitive recursion (f_{pr}) and the rules for r used in defining minimization (f_{min}).

First we note that for every ground fact in M there is a unique ground instance of a rule with this fact as the head and with every body fact in M. The existence

of at least one such instance is clear, and since no two rule heads unify, all such instances must come from the same rule. It is straightforward to verify that for every predicate defined by M the last argument is functionally dependent on the other arguments. Application of this fact to the predicates in the appropriate rule gives uniqueness.

Suppose there are two unifying facts, f_1 and f_2, generated by the rule for $f_{compose}$. Let $(g_{11}, \ldots, g_{1k}, h_1) \in GC^{k+1}$ unify with the body of the rule, with mgu θ to produce f_1. Similarly, let $(g_{21}, \ldots, g_{2k}, h_2) \in GC^{k+1}$ unify with the body of the rule, with mgu σ to produce f_2. Since f_1 and f_2 unify, there is a ground unifier α. Consider $t_1 = (g_{11}, \ldots, g_{1k}, h_1)\theta\alpha$ and $t_2 = (g_{21}, \ldots, g_{2k}, h_2)\sigma\alpha$.

As we noted earlier, corresponding to the ground head fact $f_1\alpha$ there is a unique ground instance such that the body facts are in M. If t_1 and t_2 do not unify, then let μ and ν be grounding substitutions for t_1 and t_2. $t_1\mu$ and $t_2\nu$ must be different. The facts of $t_1\mu$ and $t_2\nu$ are in M, since they are instances of facts in GC, and this gives a contradiction. If t_1 and t_2 unify then g_{1i} and g_{2i} unify for every i, and h_1 and h_2 unify. By the induction hypothesis g_{1i} and g_{2i} are equal for every i, and h_1 and h_2 are equal. Thus f_1 and f_2 are equal. By the second part of the induction hypothesis, h_1 and g_{1i} are generated only once. Thus f_1 is generated only once.

Consider $a = f_{pr}(t_1, \ldots, t_n, t) \in GC$. We use induction on the number of rule applications m used to produce a to prove that it does not unify with any other fact in GC. We observe that the first argument must be ground, and is $s^k(0)$, where k is the number of applications of the recursive rule used to generate a. Thus, the only facts that a might unify with are produced by the same number of rule applications. The claim holds for $m = 0$ by the outer induction hypothesis. Let the claim hold for fewer than m applications. It must then hold for m applications, by the same argument used for $f_{compose}$.

Consider the rules defining r. Again we use induction on the number of rule applications. The facts produced are of the form $r(\bar{X}, U, s(V), s^n(U))$ after n applications of the recursive rule. Consequently, two facts produced using different numbers of rule applications cannot unify. The rest of the argument proceeds as for f_{pr}. □

The next result follows almost immediately.

Corollary 5.1 *Every program defined according to the above scheme is subsumption-free.*

> **Proof:** We must show that $atoms(DT(P))$, is an irrset. That it is a set follows from the second part of the preceding lemma and Proposition 3.2, and so $atoms(DT(P)) = GC$. That it is an irrset now follows from the first part of the preceding lemma, since a and b not unifiable implies that a does not subsume b. □

By combining the Sebelik-Stepanek theorem and the previous corollary we have that the class of subsumption-free logic programs is as expressive as the class of all logic programs:

Theorem 5.3 *Every partial recursive function can be computed by a subsumption-free logic program.*

Thus, although the properties that we consider are decidable for function-free programs, the addition of a single unary function symbol results in undecidability.

Theorem 5.4 *It is undecidable whether an arbitrary logic program*

1. *is subsumption-free.*
2. *is duplicate-free.*
3. *has the finite subsumption property.*
4. *has the finite forest property.*

> **Proof:** Satisfiability of a query (say q) is undecidable for the class of programs defined in Theorem 5.2. From Corollary 5.1, these programs are subsumption-free. Let p be a new 0-ary predicate, and let P' be a new program constructed from a program P in this class by adding the fact p and the rule $p :- p, q$. P' is subsumption-free (duplicate-free, has the finite subsumption property, has the finite forest property) iff q is unsatisfiable in P. □

6 Characterizations

Although deciding whether these properties hold is undecidable, we have the following necessary and sufficient conditions for a program to be subsumption-free. Suppose P is the multiset of rules $[r_1, \ldots, r_n]$. Let $F(X) = set(W(X, X))$ and let F_i denote the function F for a program consisting of the single rule r_i.

Theorem 6.1 *A program P is subsumption-free if and only if it satisfies the following conditions:*

1. $\forall x \in F_i(IGC)\ \forall y \in F_j(IGC)\ x \not\leq y$ *when* $i \neq j$.
2. *For every rule* r_i: $h\ :-\ b_1, \ldots, b_k$ *in* P, *if* $g \in F_i(IGC)$ *then* g *uniquely determines* $(d_1, \ldots, d_k) \in IGC^k$ *such that* μ *is the mgu of* $(b_1, \ldots, b_k)$ *and* $(d_1, \ldots, d_k)$, *and* $h\mu \leq g$.

Proof: It is straightforward to show that if (1) or (2) does not hold then P cannot be subsumption-free (using the fact that every element of IGC has a corresponding derivation tree).

If P is not subsumption-free then IGC is a submultiset of $atoms(DT(P))$. Consider atoms a such that $a \in atoms(DT(P)) - IGC$ or, if that multiset is empty, atoms a that have duplicates (i.e. have multiplicity greater than 1 in $atoms(DT(P))$). Choose a from these atoms so that it has a derivation tree t_a of minimal height. Let $d_1, \ldots, d_k$ label the children of the root and r_m label the arcs from the root. Since the height is minimal $\{d_1, \ldots, d_k\} \subseteq IGC$.

There is an atom $b \in IGC$ such that $a \leq b$, since $a \in GC$. Now b has some derivation tree $t_b \neq t_a$ with arcs from root labeled r_n and children labeled $f_1, \ldots, f_l$. Although it may be that $f_i \notin IGC$, there are $g_i \in IGC$ such that $f_i \leq g_i$ for $i = 1, \ldots, l$. These g_i have derivation trees t_i. So b has a derivation tree t'_b where the subtrees of the root of t_b are replaced by $t_1, \ldots, t_l$. (Clearly the root should be labeled by a more general atom than b, but since $b \in IGC$ it must be equal.)

Thus $a \in F_m(IGC)$ and $b \in F_n(IGC)$, for some m, n. If $m \neq n$ then (1) does not hold. If $m = n$, (2) is violated unless $g_i = d_i$ for $i = 1, \ldots, k$, which implies $a = b$, by definition of a derivation tree. Thus we are in the case where P only generates duplicates, and $t'_b = t_b$. But, since $t_a \neq t_b$, some g_i has two different

derivation trees. That is, g_i has duplicates in $atoms(DT(P))$ and has a smaller derivation tree than t_a, which contradicts the minimality of the height of t_a. □

If we replace IGC by GC in the conditions of the above theorem then we obtain a second characterization of subsumption-free programs. The proof is essentially the same, but slightly simpler.

Similarly we have:

Theorem 6.2 *A program P is duplicate-free if and only if it satisfies the following conditions:*

1. $F_i(GC) \cap F_j(GC) = \emptyset$ *when* $i \neq j$,
2. *For every rule* r_i: $h \ :- \ b_1, \ldots, b_k$ *in* P, *if* $g \in F_i(GC)$ *then* g *uniquely determines* $(d_1, \ldots, d_k) \in GC^k$ *such that* μ *is the mgu of* $(b_1, \ldots, b_k)$ *and* $(d_1, \ldots, d_k)$, *and* $g = h\mu$.

Currently, we have no comparable characterization of the finite forest and finite subsumption properties. We present the following example (due to J.F. Naughton) to show that a simple generalization of the characterization for Datalog programs (in the proof of Theorem 5.1) is not sufficient.

Example 6.1

$r1 : p(X) \ :- \quad p(s(X)).$

$r2 : p(X) \ :- \quad q(X).$

$r3 : q(s(X)) \ :- \quad q(X).$

$r4 : q(0).$

No derivation tree contains a "loop". Nevertheless $p(s^5(0))$ for example has infinitely many derivation trees, since for each k we can generate $q(s^{5+k}(0))$, then $p(s^{5+k}(0))$, and then $p(s^5(0))$. Thus $p(s^5(0))$ has derivation trees of height $5 + 2k$ for every $k \geq 1$. □

7 Functional Dependencies Over GC and M

Clearly, the conditions presented in the previous section cannot be tested. However, they help us develop testable sufficient conditions, as we demonstrate in Section 8. Before we do so, we need to develop the notion of functional dependencies in the context of relations that contain

possibly non-ground tuples, since such dependencies play a central role in the conditions that we will explore. Informally, we will associate relations with rules such that each tuple denotes the use of the rule to generate a new program fact from a set of previously derived facts. Functional dependencies over these relations allow us to make assertions of the form that each fact is generated using a unique set of facts. With the additional condition that no two rules can generate the same fact, we essentially obtain the sufficient condition that we seek.

We begin by extending the notions of relations and functional dependencies to allow for facts that contain (possibly non-ground) terms, rather than just constants. A (generalized) relation is a set of (possibly non-ground) facts with the same predicate name (which is also the name of the relation). We sometimes refer to facts in a relation as *tuples*, using relational terminology. Let $\tilde{t} = (t_1, \ldots, t_m)$ be a tuple of terms. Then $\tilde{t}$ is a *template* for a relation R if every fact in R is an instance, not necessarily ground, of $\tilde{t}$. In this case we call R a $\tilde{t}$-relation. By choosing $\tilde{t}$ to be a tuple of distinct variables, any relation can be viewed as a $\tilde{t}$-relation.

For fixed $\tilde{t}$, let U and V be (respectively) the set of argument positions in $\tilde{t}$ and the set of variables that appear in $\tilde{t}$. Let $S = U \cup V$. Let f be a fact in a $\tilde{t}$-relation, so that $f = \tilde{t}\theta_f$ for some substitution θ_f. For any $I \subseteq S$, say $I = \{u_1, \ldots, u_k, v_1, \ldots, v_l\}$ where $u_i \in U$ and $v_j \in V$. let $\tilde{s} = (t_{u_1}, \ldots, t_{u_k}, v_1, \ldots, v_l)$. Then $f[I]$ denotes the tuple $\tilde{s}\theta_f$. Similarly, for a $\tilde{t}$-relation R, $R[I]$ denotes $\{\tilde{s}\theta_f | f \in R\}$.

A *functional dependency (FD)* $I \rightarrow J$ where $I, J \subseteq S$ is said to hold over a $\tilde{t}$-relation R iff for every pair of facts f and g in R, whenever there is a variable renaming σ such that $f[I]\sigma = g[I]$, there is a variable renaming θ such that $f[I, J]\theta = g[I, J]$. This definition is, in fact, symmetrical in f and g since a variable renaming is invertible. We say a set T of FDs holds over a relation R if every element of T holds over R.

We present the following axioms for inferring FDs over $\tilde{t}$-relations. The first three axioms are Armstrong's axioms for FDs (originally introduced for ground relations); the other two axioms are specific to $\tilde{t}$-relations, and reflect the constraints introduced by the arguments of the template $\tilde{t}$.

Axioms for FDs over $\tilde{t}$-relations

1. $I, J \subseteq S$, $I \subseteq J$ implies $J \rightarrow I$.
2. $I, J, K \subseteq S$, $I \rightarrow K, K \rightarrow J$ implies $I \rightarrow J$.

3. $I, J, K \subseteq S$, $I \rightarrow J$ implies $I \cup K \rightarrow J \cup K$.

4. $I \in U, X \in V$, X appears in position I in $\tilde{t}$ implies $\{I\} \rightarrow \{X\}$.

5. $I \in U, X \subseteq V$, X is the set of variables that appear in position I in $\tilde{t}$ implies $X \rightarrow \{I\}$.

When an FD f can be derived from a set F of FDs using these axioms, we write $F \vdash f$. The following theorem expresses the soundness and completeness of our axioms as a method for inferring FDs on $\tilde{t}$-relations.

Theorem 7.1 *Let f be an FD and F a set of FDs. Consider the axioms defined above for a fixed template $\tilde{t}$. $F \vdash f$ iff for every $\tilde{t}$-relation R, if F holds over R then f holds over R.*

Proof: The soundness of the axioms is straightforward. Before proving completeness we introduce some notation. Let $J = J_1 \cup J_2$ where $J_1 \subseteq U$ and $J_2 \subseteq V$. Then $var(J)$ denotes the set of variables that either occur in $\tilde{t}$ in an argument position of J_1 or appear in J_2. It follows from axioms 4 and 5 that $\vdash J \rightarrow var(J)$ and $\vdash var(J) \rightarrow J$.

Let f be $I \rightarrow J$ and let $X \subseteq V$ be the largest set of variables such that $F \vdash I \rightarrow X$. (It is not hard to see that X is well-defined.) Let a and b be two terms that are not equal up to renaming. Let θ_1 be the substitution that maps every variable in V to a, and let θ_2 map every variable in X to a and the remaining variables to b. Let R be the $\tilde{t}$-relation $\{\tilde{t}\theta_1, \tilde{t}\theta_2\}$. Clearly an FD $K \rightarrow L$ holds over R iff $var(K) \not\subseteq X$ or $var(L) \subseteq X$.

Suppose that $F \not\vdash f$. Then $var(J) \not\subseteq X$ since otherwise $F \vdash X \rightarrow J$ and so $F \vdash f$. Consequently f does not hold over R. On the other hand, suppose that an FD $K \rightarrow L$ in F does not hold over R, that is, $var(K) \subseteq X$ and, for some variable x, $x \in var(L) - X$. Then we can conclude $F \vdash X \rightarrow \{x\}$ (since $F \vdash X \rightarrow K$, $F \vdash K \rightarrow L$, $F \vdash L \rightarrow var(L)$, $F \vdash var(L) \rightarrow \{x\}$), which contradicts the maximality of X. Thus F holds over R. □

In order to capture the application of a rule to generate facts, we associate a $\tilde{t}$-relation $\mathcal{R}$ with rule r in program P as follows:

- Each argument position of the (head or body) literals of rule r corresponds to an argument position of $\mathcal{R}$. $\tilde{t}$ is the tuple of terms in these argument positions in r.
- Let the body of r be $b_1, \ldots, b_k$, let $\{d_1, \ldots, d_k\} \subseteq GC$, and let $\theta = \text{mgu}((b_1, \ldots, b_k), (d_1, \ldots, d_k))$. Then, the relation $\mathcal{R}$ contains a tuple such that the value in each argument position is the term in the corresponding position of $r\theta$.

We denote by $ground(\mathcal{R})$ the relation of all ground instances of $\mathcal{R}$. Thus $\mathcal{R}$ and $ground(\mathcal{R})$ are $\tilde{t}$-relations, where $\tilde{t}$ is the tuple of all argument positions in rule r. The relationship between a relation R and its ground version $ground(R)$ is expressed by the following proposition.

Proposition 7.1 *Let R be a $\tilde{t}$-relation and $ground(R)$ its corresponding ground version. Suppose that the language allows the expression of at least two ground terms. For any FD $I \rightarrow J$, if $I \rightarrow J$ holds over $ground(R)$ then $I \rightarrow J$ holds over R.*

> **Proof:** We prove the contrapositive. Suppose f does not hold over R, that is, there are facts f and g in R and a variable renaming σ such that $f[I]\sigma = g[I]$, but $f[I, J]$ and $g[I, J]$ are not equal modulo variable renaming. By corollary 12 of [14] there is a grounding substitution α for one of the facts, say g, such that $g[I, J]\alpha$ is not an instance of $f[I, J]$. Let β be a ground extension of α to all variables of $f\sigma$ and g. So $f\sigma\beta$ and $g\beta$ are in $ground(R)$. Then $f[I]\sigma\beta = g[I]\beta$ but $f[I, J]\sigma\beta \neq g[I, J]\beta$. Thus $f\sigma\beta$ and $g\beta$ are facts of $ground(R)$ showing that $I \rightarrow J$ does not hold. □

The restriction on the language is seen to be necessary by considering the relation $R = \{(X, Y), (X, a)\}$ when only one ground term, the constant a, is expressible. Thus $ground(R) = \{(a, a)\}$. If I denotes the first position and J denotes the second position then $I \rightarrow J$ holds in $ground(R)$, but not in R. Under the restriction on the language, the converse of this proposition does not hold. For example, if R is simply $\{(X, Y)\}$ then $I \rightarrow J$ holds in R, but not in $ground(R)$, since $ground(R)$ contains $\{(a, a), (a, b)\}$ where $a \neq b$.

As the proposition shows, FDs over $ground(\mathcal{R})$ and M are stronger than FDs over $\mathcal{R}$ and GC, and so occur less often. This can make a difference to our reasoning about programs as a later example will show.

In order to reason about $\mathcal{R}$, it is convenient to introduce a family of relations $\mathcal{R}_n$, that are defined analogously to $\mathcal{R}$, using GC_n instead of GC. Intuitively, these relations represent

approximations to $\mathcal{R}$, computed after n iterations of a bottom-up fixpoint computation of program P.

We use $Head$ to denote the set of argument positions corresponding to argument positions in the head of r, and $Body$ to denote the remaining argument positions. Note that $\mathcal{R}_n[Head]$ consists of the facts in GC_{n+1} generated by r.

We observe that GC can be partitioned into a set of relations with predicate names that appear in the program P. FDs over such relations are of the form $p[I] \rightarrow p[J]$, where I and J are sets of argument positions in p. We will refer to FDs over this collection of relations as FDs over GC. Since a predicate name uniquely determines the relevant subset of GC, this should cause no confusion.

Let T be a set of FDs over GC. In order to reason about T, we consider corresponding FDs over the relations $\mathcal{R}$ for program rules. For each FD, say $p[K] \rightarrow p[L]$, in T and every p-atom in the body of a rule r, there is a corresponding FD $\mathcal{R}[K'] \rightarrow \mathcal{R}[L']$, where K' and L' are the argument positions in $\mathcal{R}$ corresponding to argument positions K and L in the p-atom of r. Let T_b be the set of all such FDs for a given r. Similarly, for a given rule r let T_h be the set of FDs over $\mathcal{R}$ corresponding to the FDs in T acting on the head of r.

The following proposition states a simple and useful connection between T and T_b.

Proposition 7.2 *Let T be a set of FDs that holds over GC for a given program. Then, for every rule r in the program, T_b holds over $\mathcal{R}$.*

We have the following sufficient condition for establishing FDs over GC.

Theorem 7.2 *An FD $p[K] \rightarrow p[L]$ holds over GC if there exists some set T of FDs over GC that includes the FD $p[K] \rightarrow p[L]$ such that the following holds:*

- *For every pair of facts $f \in F_i(GC)$ and $g \in F_j(GC)$, where r_i and r_j are different rules defining q, and every FD $q[I] \rightarrow q[J]$ on q in T, either:*
  - *there is no variable renaming σ such that $f[I]\sigma = g[I]$, or*
  - *$f[I, J]\sigma = g[I, J]$ for some variable renaming σ.*
- *For every rule r in P (and its corresponding relation $\mathcal{R}$), whenever T_b holds in $\mathcal{R}$, T_h holds in $\mathcal{R}$.*

Proof: We prove, by induction on n, that every FD in T holds over GC_n, for every n. It then follows that every FD in T holds over GC, and, in particular, $p[K] \rightarrow p[L]$ holds over GC. The base case is trivial. Let r be a rule in the program P.

If the elements of T are FDs over GC_n then the corresponding elements of T_b are FDs over $\mathcal{R}_n$. By the second condition it follows that every element of T_h is an FD over $\mathcal{R}_n$. Now $\mathcal{R}_n[Head]$ is the set of facts in GC_{n+1} generated by r, so the subset of T corresponding to T_h holds over this set. By the first condition no other rule generates a fact that destroys a FD in T. Since this argument applies to every rule of P, the elements of T are FDs over GC_{n+1}. □

It is also useful to consider a similar result for FDs over the least Herbrand model M (which is equal to $ground(GC)$). The development is similar to the above, with the following changes: Instead of GC, we consider relations corresponding to program predicates over M, instead of $\mathcal{R}$, we consider $ground(\mathcal{R})$, and instead of F_i we use $ground \circ F_i$, which is equivalent to the van Emden-Kowalski function T for the rule r_i [24].

We have the following sufficient condition for establishing FDs over M. The proof is similar to that of the previous theorem, and is omitted.

Theorem 7.3 *An FD $p[K] \rightarrow p[L]$ holds in M if there exists some set T of FDs over M that includes the FD $p[K] \rightarrow p[L]$ such that the following holds:*

- *For every pair of facts $f \in ground(F_i(M))$ and $g \in ground(F_j(M))$, where r_i and r_j are different rules defining q, and every FD $q[I] \rightarrow q[J]$ on q in T, either:*
  - *$f[I] \neq g[I]$, or*
  - *$f[I, J] = g[I, J]$.*
- *For every rule r in P, whenever T_b holds in $ground(\mathcal{R})$, T_h holds in $ground(\mathcal{R})$, where T_b and T_h are defined over $ground(\mathcal{R})$.*

The following proposition shows that detecting that an FD holds over GC is in general undecidable.

Proposition 7.3 *Let p be a predicate and let I and J be subsets of the set of argument positions of p. It is not decidable for arbitrary program P whether $p[I] \rightarrow p[J]$ holds over GC.*

Proof: Consider a program P consisting of fact $p(0,1)$ and rule $p(0,2) \ :- \ q$ defining p, and a program P' not using or defining p (where q is some body not containing p). Then $p_1 \rightarrow p_2$ holds over GC iff q is unsatisfiable in P', and this is undecidable. □

The proof also shows that the problem of determining whether an FD holds over M for an arbitrary program is undecidable. Similarly, the corresponding problem for FDs over $\mathcal{R}$ and $ground(\mathcal{R})$ is not decidable, which can be seen by further adding the rule $r \ :- \ p(X,Y)$ to P.

Faced with uncomputability, we suggest the following methodology for establishing FDs:

To establish an FD d over $\mathcal{R}$, find a set T of FDs over relations of GC such that:

- $T_b \vdash d$

and for each FD $p[I] \rightarrow p[J]$ in T

- Let f be a fact in $F_i(GC)$ and g be a fact in $F_j(GC)$, where r_i and r_j are rules defining p. Then, $f[I]$ and $g[I]$ do not unify.
- For every rule r defining p, $T_b \vdash T_h$.

Theorem 7.4 *The above methodology is sound.*

Proof: The second condition implies that T holds over GC, by Theorem 7.2. Consequently T_b, holds over $\mathcal{R}$. By Theorem 7.1 and the first condition, d is an FD over $\mathcal{R}$. □

We do not consider how to test the above conditions in this paper. However, fairly simple tests are sufficient for all the examples considered in this paper, including programs constructed using the Sebelik-Stepanek scheme. For example, to test the first part of the second condition, it is sufficient to establish that $h_i[I]$ and $h_j[I]$ do not unify, for every pair of heads h_i and h_j of different rules of P defining p.

The methodology is easily modified to establish sets of FDs over GC; to establish a set T' of FDs over GC, we must find $T \supseteq T'$ which satisfies the second condition. It is also easily adapted to establish FDs over $ground(\mathcal{R})$: we must simply consider FDs over M instead of GC.

and define T_b and T_h to be FDs over $ground(\mathcal{R})$. Clearly it can also be modified to establish sets of FDs over M.

Our work generalizes results of Reddy [17] and Debray and Warren [8], who also present techniques for inferring FDs. Reddy infers FDs over M; we generalize his results by inferring FDs over GC and $\mathcal{R}$, in addition to M. Debray and Warren infer FDs in which a set of arguments that must be bound to ground terms determine the rest of the arguments. (Mode inference is used to identify these sets of arguments.) They are able to infer FDs in some situations where the determined arguments may be non-ground; thus, they infer certain FDs over GC. Further, they use more sophisticated criteria to determine when the sets of facts generated by two rules are disjoint; these criteria can be used to refine our approach as well as Reddy's. However, they do not reason directly in terms of GC or M, which limits their use of the transitivity axiom. As a consequence, Reddy's algorithm infers certain FDs over M that they cannot infer.

Reddy and Debray-Warren both rely upon mode inference to determine which arguments are ground in a call (of a top-down evaluation method such as Prolog). We expect that such a mode analysis can be used in conjunction with our results as well. Indeed, similar (but not identical) analyses are an essential part of program transformations used in the bottom-up approach to logic program evaluation [4, 16]. The work of [17] and [8] on FD inference was carried out in the context of the problem of identifying functional calls in logic programs. A call is functional if it instantiates a set of "input" arguments to ground terms, and the remaining ("output") arguments can be instantiated in only one way. This property can be phrased as an FD. Our techniques also allow us to deal with FDs over GC and $\mathcal{R}$, in addition to their ground counterparts, and this may allow the detection of functionality — extended suitably in terms of GC, by not requiring the "input" arguments to be ground — in some additional cases.

8 A Sufficient Condition

We present a sufficient condition for subsumption-freedom that is powerful enough to establish the subsumption-freedom of the class of programs defined by Sebelik and Stepanek; in essence, we strengthen a non-subsumption stipulation in the characterization presented in Section 6 to a non-unifiability stipulation. Further, we consider $ground(\mathcal{R})$.

Theorem 8.1 *Suppose that the following conditions hold for a program P:*

- $f_1 \in F_i(GC), f_2 \in F_j(GC), i \neq j$ *implies* f_1 *and* f_2 *do not unify.*
- *For every rule* r, $ground(\mathcal{R})[Head] \rightarrow ground(\mathcal{R})[Body]$.

Then P is subsumption-free.

Proof: We show that for all $f_1, f_2 \in GC$, f_1 and f_2 do not unify. If f_1 and f_2 do not unify, then certainly one cannot subsume the other. Using this, we show that $atoms(DT(P)) = GC = IGC$, and so P is subsumption-free.

For the first part we use induction on n, with the following induction hypothesis: For all $f_1, f_2 \in GC_n$, f_1 and f_2 do not unify or $f_1 = f_2$. The base case is trivial. Suppose $f_1, f_2 \in GC_{n+1}$, and f_1 and f_2 unify with ground unifier α. Then, by the first condition, $f_1, f_2 \in F_j(GC_n)$ for some rule r_j with body $b_1, \ldots, b_k$.

Let $(d_{i1}, \ldots, d_{ik}) \in GC_n^k$ unify with $(b_1, \ldots, b_k)$ giving mgu θ_i to produce f_i, for $i = 1, 2$. Let $t_i = (d_{i1}, \ldots, d_{ik})\theta_i\alpha = (b_1, \ldots, b_k)\theta_i\alpha$ for $i = 1, 2$. Suppose t_1 and t_2 do not unify. Let t_1' and t_2' be ground instances of t_1 and t_2 respectively, so $t_i' \in M^k$ for $i = 1, 2$, and $t_1' \neq t_2'$. Hence $ground(\mathcal{R})$ contains tuples corresponding to $f_1\alpha, t_1'$ and $f_2\alpha, t_2'$. But $f_1\alpha = f_2\alpha$, so this contradicts the second condition. Thus t_1 and t_2 unify. It follows that $(d_{11}, \ldots, d_{1k})$ and $(d_{21}, \ldots, d_{2k})$ must unify, and so d_{1j} and d_{2j} unify for $j = 1, \ldots, k$. By the induction hypothesis $d_{1j} = d_{2j}$ for $j = 1, \ldots, k$, and hence $f_1 = f_2$.

The proof that $atoms(DT(P)) = GC$ follows the proof above, using as induction hypothesis $atoms(DT(P)|_n) = GC_n$. We suppose $[f, f] \subseteq atoms(DT(P)|_{n+1})$ and similarly to the above proof, both occurrences of f must be generated by the same rule r_j from $atoms(DT(P)|_n)$. t_1 and t_2 are defined as before, and by the same argument as above must unify. It then follows that both occurrences of f are generated by the same rule from the same tuple of atoms in $atoms(DT(P)|_n)$. Thus (using the induction hypothesis) they have identical derivation trees, that is, no duplicate f is generated. Hence $atoms(DT(P)|_{n+1}) = GC_{n+1}$. □

From the proof of the previous theorem we can derive a stronger result:

Corollary 8.1 *Under the conditions of the previous theorem, the elements of GC are pairwise non-unifiable.*

We now illustrate the use of this theorem with some simple examples. We remark that the theorem is applicable to the class of programs defined by Sebelik and Stepanek; indeed, it can be seen as an abstraction of the proof that this class of programs is subsumption-free. We use p_i to denote the i^{th} argument position in a relation for predicate symbol p, and in writing FDs we omit the set notation.

Example 8.1 Consider the program for naive reverse:

$r1 : reverse(X.Y, W) \ :- \quad reverse(Y, Z), append(Z, X.nil, W).$

$r2 : reverse(nil, nil).$

$r3 : append(U.V, X, U.Y) \ :- \quad append(V, X, Y).$

$r4 : append(nil, X, X).$

First, we must show that at the ground level, the head fact uniquely determines the body fact. That is, for each rule r the head determines the body in $ground(\mathcal{R})$. This can be established by first considering the set of FDs $T = \{reverse_1 \rightarrow reverse_2; append_1, append_2 \rightarrow append_3\}$ over M. We can easily show that for every rule, any head dependency in the table follows by applying the axioms to the dependencies for the body literals. Furthermore, $r1$ cannot generate a fact whose first argument unifies with nil, and $r3$ cannot generate a fact whose first two arguments are of the form nil, x. Thus T holds over M, using the methodology of Section 7. It is now straightforward to show that, for the non-trivial rules $r1$ and $r3$, the head determines the body in $ground(\mathcal{R})$.

We must also show that no two facts produced by different rules unify; this follows because different rule heads do not unify. From Theorem 8.1, it follows that this program is subsumption-free.

Note that although our sufficient condition deals with $ground(\mathcal{R})$, it allows us to show subsumption-freedom of programs that generate non-ground tuples, as in this example. (The given program generates all possible tuples in *append* and *reverse*; to consider the behaviour when it is used to reverse a given list, we could consider the program obtained by the Magic Templates transformation [16].) □

Example 8.2 The following example illustrates that the technique is equally applicable to programs containing mutual recursion:

$r1 : even(X) \ :- \quad plus(Y, 1, X), odd(Y).$

$r2 : even(0).$

$r3 : odd(X) \ :- \quad plus(Y, 1, X), even(Y).$

From the dependency $plus_2, plus_3 \rightarrow plus_1$, and the axioms for FDs, we can show that the head determines the body in $ground(\mathcal{R})$ for each rule. Further, the first rule cannot generate $even(0)$, and so no fact can be generated by two different rules. Thus, the conditions of the previous theorem are satisfied, and it follows that the program is subsumption-free. □

9 Discussion of the Sufficient Condition

Theorem 8.1 describes a sufficient condition for subsumption-freedom, and is one of our main results. While this theorem is quite powerful, as we have seen, it is worth attempting to develop other sufficient conditions that either generalize or complement it. The following example demonstrates that there are some quite simple programs that cannot be determined to be subsumption-free by using Theorem 8.1.

Example 9.1 The sufficient condition for subsumption-freedom that we have presented does not utilize FDs over non-ground relations. Consequently, it might be that dependencies hold over GC and $\mathcal{R}$ that could allow us to establish this property (using more powerful sufficient conditions), but such a set of FDs does not hold over the ground counterparts. Consider the following program:

$q(U) \ :- \quad s(U, V).$

$s(X, Y).$

The dependency $s_1 \rightarrow s_2$ holds only on GC, not M. Thus, we can see that the program is subsumption-free (since the above dependency allows us to show that the head determines the body in the $\mathcal{R}$ relation corresponding to the first relation), using Theorem 6.1. However, our sufficient condition for subsumption-freedom (Theorem 8.1) cannot establish this. □

From an examination of Theorems 6.2 and 8.1, it is natural to consider whether simply replacing the second condition of Theorem 8.1 by the corresponding condition on $\mathcal{R}$ would yield a more powerful sufficient condition. Unfortunately the condition is no longer sufficient, as the next example shows.

Example 9.2 Consider the program P:

$q(X,Y) \ :- \quad p(X,Y,Z).$

$p(U,V,5).$

$p(U,f(W),6).$

P is not subsumption-free since the facts $q(U,V)$ and $q(U,f(W))$ are generated, and the former subsumes the latter. However the head determines the body in the relation $\mathcal{R}$ for the first rule, and so the proposed condition would (incorrectly) infer that P is subsumption-free. □

The examples suggest that although we need to develop stronger sufficient conditions, reasoning with GC and $\mathcal{R}$ in addition to their ground counterparts, their development is not straightforward. However the techniques we have developed for establishing FDs over non-ground relations provide a starting point.

10 Conclusions and Future Work

We have studied bottom-up fixpoint evaluation techniques for logic programs and identified some important properties that permit novel optimizations. We have developed a framework for reasoning about properties like subsumption-freedom by associating relations with rules of a logic program, extended the notion of functional dependencies to reason over such relations, and presented methods for establishing such functional dependencies. In particular, we have obtained a testable sufficient condition for subsumption-freedom that is powerful enough to deal with a class of logic programs that can compute all partial recursive functions.

This work raises a number of issues that need further study. The foremost problem is to develop stronger suffient conditions for detecting subsumption-freedom and duplicate-freedom. The finite forest and finite subsumption properties present a more difficult task; we must formulate characterizations that are amenable to efficient approximation, and then develop sufficient conditions from them.

On a different note, given two programs with property P, what properties can be guaranteed for the union of the two programs? In general, neither duplicate-freedom nor the finite forest property is preserved. Under what conditions on the two programs can we guarantee that some property is preserved? If not, how do we utilize the properties of the individual programs in evaluating the fixpoint of the union? If a program has a property P, what can we say about

the program obtained by applying various program transformations such as Magic Sets [4, 16]?

The optimization of fixpoint evaluation is critical to the efficiency of the bottom-up approach to logic program evaluation. We believe that the results in this paper have practical and theoretical significance, and point to an important new direction for further research.

11 Acknowledgements

We thank J-L. Lassez for placing the right emphasis on the title, and I.S. Mumick and J.F. Naughton for useful comments.

References

[1] K. Apt, Efficient Computing of Least Fixpoints. In *Technical Report TR-88-33, University of Texas at Austin, 1988.*

[2] I. Balbin and K. Ramamohanarao, A Generalization of the Differential Approach to Recursive Query Evaluation. In *Journal of Logic Programming 4, 259-262, 1987.*

[3] F. Bancilhon, A Note on the Performance of Rule Based Systems. In *MCC Technical Report DB-022-85, 1985.*

[4] F. Bancilhon, D. Maier, Y. Sagiv and J. Ullman, Magic Sets and Other Strange Ways to Implement Logic Programs. In *Proc. 5th ACM SIGMOD-SIGACT Symposium on Principles of Database Systems, 1986.*

[5] F. Bancilhon and R. Ramakrishnan, An Amateur's Introduction to Recursive Query Processing Strategies. In *Proc. SIGMOD, 1986.*

[6] R. Bayer, Query Evaluation and Recursion in Deductive Database Systems. In *Technical Report TUM-I8503, Technische Universitat Munchen, 1985.*

[7] C. Beeri and R. Ramakrishnan, On the Power of Magic. In *Proc. 6th ACM SIGMOD-SIGACT Symposium on Principles of Database Systems, 1987.*

[8] S.K. Debray and D.S. Warren, Functional Compositions in Logic Programs. To appear in ACM TOPLAS.

[9] M. Falaschi, G. Levi, M. Martelli and C. Palamidessi, A New Declarative Semantics for Logic Languages. In *Proc. 5th International Conference and Symposium on Logic Programming, 1988.*

[10] H. Gaifman and E. Shapiro, Fully Abstract Compositional Semantics for Logic Programs. In *Proc. Principles of Programming Languages, 1989.*

[11] H. Gaifman and E. Shapiro, Proof Theory and Semantics of Logic Programs. In *Proc. Logic in Computer Science, 1989.*

[12] M. Kifer and E. Lozinskii, A Framework for an Efficient Implementation of Deductive Databases. In *Proc. Advanced Database Symposium, Tokyo, 1986.*

[13] J-L. Lassez and M. Maher, Closures and Fairness in the Semantics of Programming Logic. In *Theoretical Computer Science, 29, 167-184, 1984.*

[14] J-L. Lassez, M.J. Maher and K.G. Marriott, Unification Revisited. In *Foundations of Deductive Databases and Logic Programming, J. Minker (Ed), Morgan Kaufmann, 587-625, 1988.*

[15] J. Naughton and R. Ramakrishnan, A Unified Approach to Logic Program Evaluation. In preparation.

[16] R. Ramakrishnan, Magic Templates: A Spellbinding Approach to Logic Programs. In *Proceedings of the International Conference on Logic Programming, pages 140-159, Seattle, Washington, August 1988.*

[17] U. Reddy, Transformation of Logic Programs into Functional Programs. In *Proc. International Symposium on Logic Programming, 1984.*

[18] J. Rohmer , R. Lescoeur and J.M. Kerisit, The Alexander Method, a Technique for the Processing of Recursive Axioms in Deductive Databases. In *New Generation Computing, 4, 3, 273-285, 1986.*

[19] D. Sacca and C. Zaniolo, The Generalized Counting Methods for Recursive Logic Queries. In *Proceedings of the First International Conference on Database Theory, 1986*

[20] H. Schmidt, W. Kiessling, U. Guntzer and R. Bayer, Compiling Exploratory and Goal-Directed Deduction into Sloppy Delta-Iteration. In *Proceedings of the 1987 Symposium on Logic Programming, 1987.*

[21] J. Sebelik and P. Stepanek, Horn Clause Programs for Recursive Functions. In *Logic Programming, Eds. K. Clark and S.-A. Tarnlund, Academic Press, 1982.*

[22] H. Seki, On the Power of Alexander Templates. In *Proc. 8th ACM SIGMOD-SIGACT Symposium on Principles of Database Systems, 1989.*

[23] J.D. Ullman, Bottom-Up Beats Top-Down for Datalog. In *Proc. 8th ACM SIGMOD-SIGACT Symposium on Principles of Database Systems, 1989.*

[24] M. van Emden and R. Kowalski, Semantics of Predicate Logic as a Programming Language. In *JACM 23, 4, 733-742, 1976.*

[25] L. Vieille, Recursive axioms in Deductive Databases: The Query/Subquery Approach. In *Proc. First International Conference on Expert Database Systems, Charleston, 1986.*

[26] K. Weihrauch, Computability. Springer-Verlag, 1987.