Subsumption-free bottom-up evaluation of logic programs with partially instantiated data structures

Zoltan Somogyi, David Kemp,
James Harland and Kotagiri Ramamohanarao
{zs,kemp,jah,rao}@cs.mu.OZ.AU
Department of Computer Science, University of Melbourne
Parkville, 3052 Victoria, Australia

Abstract

Most deductive databases impose restrictions on the terms that can be used in them: they require that generated tuples be fully ground, i.e. contain no variables. Without this restriction, query evaluation of general programs needs subsumption tests in order to terminate correctly, and subsumption tests are expensive.

We study a class of programs that can handle terms with both variables and function symbols but for which subsumption tests are not necessary. This class is based on the notion of *modes*, which generalise adornments by not requiring that "bound" be synonymous with "ground". We show how programs in this class may be recognized and how they should be transformed into an executable form, and then prove that bottom-up evaluation of the resulting programs does not require subsumption tests.

1 Introduction

Deductive databases are logic programming systems designed for applications that handle large amounts of data. Since logic programs can be viewed as an extension of relational databases and since relational databases handle large amounts of data, deductive database researchers usually try to use techniques developed for relational systems whenever possible. One consequence of this tendency is that unlike implementations of general-purpose logic programming systems, most deductive database systems use bottom-up evaluation methods.

Bottom-up evaluation methods work in two phases. The first phase rewrites the program and the query into a more efficient form using algorithms such as the magic set transformation [1] and the context transformation [3]. The second phase starts with the facts in the rewritten program, and uses the rules of the rewritten program to derive new facts until it can generate no more new facts, and then returns the facts that the query asked for. The task of the first phase is to reduce to the minimum the number of facts the second phase must consider.

Most deductive databases that use bottom-up methods compute only with ground terms (e.g. Aditi [10]). Some (e.g. CORAL [6]) can handle non-ground terms, but only by using subsumption as a frequent operation. Consider the following program:

```
p(X).
p(f(X)) :- p(X).
```

A bottom-up evaluation of this query produces first p(X), then p(f(X)), then p(f(f(X))), and so on. The evaluation will therefore not terminate unless newly derived tuples are tested for subsumption by existing tuples.

Subsumption is especially difficult to implement when the relation being computed is large. When the evaluation algorithm produces a tuple, we must be able to check whether this tuple is subsumed by a previously generated tuple and whether this tuple subsumes any previously generated tuples. This can be done quickly only if we can limit the number of places in which we have to look for potentially interfering tuples. Unfortunately, in general this is not possible.

```
p(a1,b1,c1).
p(X, B, C) :- p(A, B, C).
p(A, Y, C) :- p(A, B, C).
p(A, B, Z) :- p(A, B, C).
```

The first iteration computes p(a1,b1,c1). The second iteration computes the tuples p(X,b1,c1), p(a1,Y,c1) and p(a1,b1,Z). Since p(X,b1,c1) subsumes p(a1,b1,c1), p(X,b1,c1) will replace p(a1,b1,c1) in the temporary relation storing the extension of p; the rest of the iteration will then add p(a1,Y,c1) and p(a1,b1,Z) to this temporary relation. These three tuples will all be replaced in the next iteration with three tuples that each have two variables: p(X,Y,c1), p(X,b1,Z) and p(a1,Y,Z). The final iteration will replace all these with the tuple p(X,Y,Z).

This example illustrates the difficulty of implementing subsumption efficiently. There are eight tuples that subsume p(a1,b1,c1): p(a1,b1,c1) itself, p(X,b1,c1), p(a1,Y,c1), p(a1,b1,Z), p(X,Y,c1), p(X,b1,Z), p(a1,Y,Z) and p(X,Y,Z). When we generate p(a1,b1,c1), we must look in the temporary relation for each of these before we can say that p(a1,b1,c1) is in fact a new tuple. If the temporary relation were bigger, this could mean up to eight disk accesses. This is not all: one can increase the number of probes required exponentially by adding more arguments to p.

Since subsumption can be so expensive, it is natural to ask whether there are any useful classes of programs for which it is not necessary. The problem in this example is that some clauses of p define all three arguments while some other clauses don't. Most systems based on adornments prevent this by requiring that the programmer state for each argument whether it will be free or ground in the query.

This approach works, but it prevents us from computing with partially instantiated data structures, i.e. terms which may contain function symbols and variables. We would like to be able to execute queries such as ?- path(X, Y, [S1,S2]) with answers such as path(melbourne, new_york, [honolulu,los_angeles]). In this case, the third argument is partially input and partially output to path, so it cannot be adorned either free or bound. One can handle this by loosening the meaning of adornments, allowing queries to bind only some parts of an argument declared bound, as in CORAL [6], but this still requires subsumption.

Our solution is to make use of the type of the third argument, which is a list of placenames, and require that the query define the list skeleton (which our *modes* declare to be input to path) but not the placenames in the list (which they declare to be output from path). This clear division of responsibilities lets us avoid subsumption tests.

The structure of the paper is as follows. In Section 2 we introduce our notions of selectors and types as necessary background for the description of our mode system, which follows in Section 3. In section 4, we show how to transform mode correct programs for bottom-up evaluation, which, as we prove in section 5, does not need subsumption. The proofs of our theorems are available in [8].

2 Selectors and types

2.1 Selectors

Definition 2.1 A *term* is defined as follows: a variable X is a term, and if f is a function symbol of n arguments (n $\geq$ 0), and $t_1, \ldots, t_n$ are terms, then $f(t_1, \ldots, t_n)$ is a term. A function symbol of arity 0 is a *constant*. A term that is not a variable or a constant is also called a *structured* or *complex* term. A term that contains no variables is called a *ground* term. We follow deductive database convention in denoting variables by identifiers starting in upper case and function and predicate symbols by identifiers starting in lower case.

Definition 2.2 We denote the set of function symbols in the program and query by $\mathcal{F}$. We denote the arity of a function symbol f by arity(f); we assume that the arity is uniquely determined by the function symbol itself.

We define two new notions, position selectors and function selectors, to be more precise about the components of terms. Both require viewing terms as trees in the usual way, i.e. with the main function symbol as the root of the tree and an arc from the main function symbol to the roots of the subtrees of the arguments, whose trees are defined similarly.

Definition 2.3 A *position selector* is a finite sequence of pairs describing a position in a term. The root of the term is labelled with the empty sequence, denoted by ϵ. If the label of a node is ps, and the tree contains the function symbol f at that node, then the i'th child of that node is labelled ps:[f,i]. A *function selector* is a position selector followed by the function symbol at that position.

Example 2.1 In the term f(g(a), h(X, b)), the set of all position selectors is: ϵ, [f,1], [f,1]:[g,1], [f,2], [f,2]:[h,1], and [f,2]:[h,2], and the set of all function selectors is: f, [f,1]:g, [f,1]:[g,1]:a, [f,2]:h, and [f,2]:[h,2]:b. The term at the position [f,2] is h(X, b). Note that there is no function selector for the first argument of the h, since the term contains a variable there.

Position selectors and function selectors are intimately related. A position selector can be extended to a function selector by naming the function symbol at the designated position, while a function selector can be extended to a position selector by naming one of the argument positions of the last function symbol, *provided* that the last function symbol is not a constant. On the other hand, one can *always* derive a position selector from a function selector by removing the final function symbol. We will write ps(fs) to denote the position selector derived from the function selector fs in this way; we will write PS(FS) for the set of position selectors derived from the set of function selectors FS.

One can derive a set of function selectors from any ground term t; we will call this set fs(t). This set will have one element for each function symbol in t.

2.2 Types

Definition 2.4 A *type* is a (possibly infinite) set of function selectors FS.

Example 2.2 The infinite set of function symbols in the type "list of booleans" contains

nil	cons	[cons,1]:true	[cons,1]:false
[cons,2]:nil	[cons,2]:cons	[cons,2]:[cons,1]:true	[cons,2]:[cons,1]:false

and many others; all can be generated by the scheme

[cons,2]^n:nil [cons,2]^n:cons [cons,2]^n:[cons,1]:true [cons,2]^n:[cons,1]:false

where n can be any non-negative integer. The term cons(true, cons(false, nil)) belongs to the type. The term cons(nil, false) does not, because neither [cons,1]:nil nor [cons,2]:false is in the scheme above.

Schemes like the one just above are necessary if we want to represent types in finite storage space. For this example, the scheme says that there are two kinds of positions in lists of booleans. Variables appearing in positions of the form [cons,2]^n are of the type "list of booleans" and must therefore be bound to either nil or cons; variables appearing in positions of the form [cons,2]^n:[cons,1] are of the type "boolean" and must therefore be bound to either true or false. Using the syntax of one of the NU-Prolog [9] type checkers, this can be expressed by declarations such as

```
?- type listbool ---> nil ; cons(bool, listbool).
?- type bool ---> true ; false.
```

In some parts of the paper, we denote the function symbol cons by the infix operator ".".

Types such as integers can in principle be declared just like booleans, although in practice they must be built-in. Our examples in later sections will use a type called listint, which contains a list of integers.

The information in such declarations can be expressed in a graph with two kinds of nodes: one for sets of position selectors and one for function symbols, with the links between them given by the natural relation between position selectors and function selectors. Such graphs not only represent a type, they also state that some position selectors are in some sense *equivalent*. One sense is that two position selectors represent the same type; another sense is introduced in the next section. For example, the declarations and the scheme above state that variables appearing in positions of the form $[\text{cons},2]^n$ are of the same type. We can formalize this idea through the notion of a type graph.

Definition 2.5 A *type graph* is a pair (FS, $\equiv$) where FS is a (possibly infinite) set of function selectors and $\equiv$ is a relation on PS(FS) obeying the property that if ps1 $\equiv$ ps2, then

choice equivalence	$\forall$ f $\in \mathcal{F}$, ps1:f $\in$ FS $\Leftrightarrow$ ps2:f $\in$ FS
subnode equivalence	$\forall$ f $\in \mathcal{F}$, $\forall$ i $\in$ {1..arity(f)}, ps1:f $\in$ FS $\Rightarrow$ ps1:[f,i] $\equiv$ ps2:[f,i]
nonconvergence	ps1 = p1:[f,i] $\wedge$ ps2 = p2:[g,j] $\Rightarrow$ p1 $\equiv$ p2.

The first two properties together imply that ps1:ext $\equiv$ ps2:ext if ps1 $\equiv$ ps2. The third makes sure that ps1:ext $\equiv$ ps2:ext *only* if ps1 $\equiv$ ps2.

Lemma 2.1 *A relation on two sets of position selectors that obeys the properties of choice equivalence and subnode equivalence is reflexive, symmetric and transitive, and hence an equivalence relation.*

Definition 2.6 A *position node* in a type graph $\langle$FS, $\equiv\rangle$ is an equivalence class induced by $\equiv$ on PS(FS). We denote the position node that a position selector ps belongs to by pnode(FS, $\equiv$, ps). We denote the set of position nodes in a type graph by pnode_set(FS, $\equiv$). A node n_1 in a type graph is a descendant of another node n_2 if $n_1 \neq n_2$, ps $\in n_1$ and ps:ext $\in n_2$.

We can label the position nodes by a scheme that yields the position selectors in that equivalence class, or if the set of equivalence classes is known, by a single position selector in the class (preferably the shortest). In this paper, we will use the label *skel* to refer to position nodes of the form $[\text{cons},2]^n$, the label *elt* (short for element) to refer to position nodes of the form $[\text{cons},2]^n$:[cons,1], and the label *root* to refer to position nodes of the form ϵ.

Two type graphs (FS_1,$\equiv_1$) and (FS_2,$\equiv_2$) represent the same type if $FS_1 = FS_2$.

Definition 2.7 A *signature* is a function from a sequence of argument types to a result type.

We require every function symbol to have a unique signature: each argument of the function symbol must have the type given by the corresponding sequence element, and the type of the term created by applying the function symbol to the arguments is the result type of the signature. This allows us to unambiguously deduce the type of every ground term, and the type of every non-ground term as well if we know the types of the variables occurring in each term.

Since we need a strong type system as a foundation for our strong mode system, we assume that the program contains a type declaration for every predicate, giving the type of the arguments of that predicate.

Definition 2.8 A program and query are *type correct* if there is an assignment of types to the variables occurring in the program and the query such that the type of every argument of every atom in the program and the query (including those in clause heads) is identical to the declared type of the corresponding formal argument.

In the rest of the paper, we assume that the program and the query are type correct, and that the type checker that establishes type correctness gives us the type of every variable in the program and the query in the form of a finite type graph.

Our type system is very similar to the Mycroft-O'Keefe type system [5], which has its roots in the type systems of functional languages such as ML [4], and which has been adopted for new logic programming languages such as Gödel [2]. The main differences are that for simplicity of presentation, in this paper we assume that each function symbol has one signature, and that all types appearing in the program are monomorphic, i.e. we do not deal with polymorphic types. Since both differences make type checking easier, we omit a description of the type checking algorithm and refer the reader to papers describing type checking algorithms for related type systems (see e.g. [5]).

3 A strong mode system

Definition 3.1 A *substitution* is a set of mappings from variables to terms, denoted by $\{v_1/t_1, \ldots, v_n/t_n\}$.

Definition 3.2 An *instantiation state* of a variable of type (FS,$\equiv$) is a mapping from the set PS(FS) to the set {bound,notbound}. A variable is in a given instantiation state with respect to a substitution if the substitution maps the variable to a term such that for all the position selectors in that term

- if that position is mapped to bound, then there is a function symbol at that position
- and if that position is mapped to notbound, then there is a variable at that position.

Example 3.1 Consider the instantiation state for lists of booleans that maps position selectors of the form $[\text{cons},2]^n$ to bound and those of the form $[\text{cons},2]^n$:[cons,1] to notbound. The term cons(A, cons(B, nil)) is in this instantiation state. The term cons(A, L) is not, because [cons,2] is mapped to bound yet the subterm in that position (L) is a variable, not a function symbol.

During the evaluation of a query, a variable may go through several instantiation states. For describing mappings between instantiation states, we use the concept of *modes* that we introduced in [7].

Definition 3.3 A *mode* is a mapping from the nodes of a set of type graphs, one type graph for each argument of a predicate, onto the set {inbound,infree,outbound,outfree} (elements of which are also called modes). We refer to the modes inbound and infree as *input modes*, and the modes outbound and outfree as *output modes*.

The intention of each of these four modes are as follows:

- Positions mapped to inbound represent information that must be provided by the caller of the predicate. Initial and final instantiation states are both bound.
- Positions mapped to outbound represent information that is provided by the predicate to its caller. Initial instantiation state is notbound while final instantiation state is bound.
- Positions mapped to infree mark parts of data structures that are ignored by the predicate. Initial and final instantiation states are both notbound.
- Positions mapped to outfree represent distinct free variables created by the predicate. Initial and final instantiation states are both notbound.

In a conventional deductive database such as Aditi [10], an argument all of whose pieces are mapped to inbound would be given the "b" (bound) adornment, while an argument all of whose pieces are mapped to outbound would be given the adornment "f" (free). Our modes thus subsume the notion of adornments, and extend that notion with the ability to talk with precision about partially instantiated terms.

The semantics of modes impose the following constraints on mode declarations:

- If ps is mapped to inbound, then any ps:[f,i] must not be mapped to outfree.
- If ps is mapped to infree, then any ps:[f,i] must be mapped to infree.
- If ps is mapped to outbound, then any ps:[f,i] must not be mapped to infree.
- If ps is mapped to outfree, then any ps:[f,i] must be mapped to outfree.

A mode declaration obeying these constraints is *valid*.

A mode declaration associates one or more modes with a predicate: it lists the inbound nodes, infree nodes, outbound nodes and outfree nodes (in that order) of the type graphs of the arguments. We assume that the program contains a valid mode declaration for every predicate.

Example 3.2 Consider the predicate len:

```
?- pred len(L: listint, N: int).
?- mode len([L:skel], [L:elt], [N:root], []).
?- mode len([N:root], [], [L:skel], [L:elt]).
len(L, N) :- L = nil, N = 0.
len(L, N) :- L = cons(H, T), len(T, N1), plus(N1, 1, N).
```

It has two principal modes:

mode 1 (forward):	mode 2 (backward):
L:$[\text{cons},2]^n \rightarrow$ inbound	L:$[\text{cons},2]^n \rightarrow$ outbound
L:$[\text{cons},2]^n$:[cons,1] $\rightarrow$ infree	L:$[\text{cons},2]^n$:[cons,1] $\rightarrow$ outfree
N:$\epsilon \rightarrow$ outbound	N:$\epsilon \rightarrow$ inbound

The queries len(cons(A, cons(B, nil)), N) and len(cons(1, cons(2, nil)), N) are legal invocations of len in its first mode, and len(A, 3) is a legal invocation of len in its second mode.

Definition 3.4 A *mode assignment* is an assignment to every atom in the body of a clause (including the head) of one of the modes of the corresponding predicate. An *assigned clause* is a clause together with a mode assignment on that clause. An *assigned program* consists of *assigned clauses*.

If a predicate p has n mode declarations in the original program, then the assigned program has n predicates derived from p, one for each mode, that are treated as completely separate predicates. Systems that use adornments do the same thing (see e.g. [1]).

Definition 3.5 The head of an assigned clause is the *producer* of a position if the mode of the head maps that position to inbound; a body atom is the producer of a position if the mode of that atom maps that position to outbound. The head of an assigned clause is a *consumer* of a position if the mode of the head maps that position to outbound; a body atom is a consumer of a position if the mode of that atom maps that position to inbound.

Example 3.3 In the first mode of len, the head is the producer of L:skel in both clauses; the unification L = nil is its consumer in the first clause and the unification L = cons(H, T) is its consumer in the second clause.

Definition 3.6 An assigned clause is *mode correct* if every position that has at least one consumer also has at least one producer in the clause, and if every position that is mapped to infree or outfree either in the head or in a unification in the body has no producer in the clause.

Definition 3.7 A mode-correct assigned clause is *non-circular* if the atoms in the body can be ordered so that each atom consumes only positions whose producer is either an atom on its left or the head.

The algorithm for finding a correct mode assignment for a clause keeps track of the instantiation state of each node of each variable of the clause. Initially, each node is unknown except the ones mapped to inbound in the head. The algorithm then considers the atoms in the body: if an atom takes as inbound only nodes whose instantiation state is bound, then we mark the atom and set the instantiation state of its outbound nodes to bound. We continue until all atoms are marked (in which case the clause has a non-circular assignment) or until we cannot mark any more atoms (in which case the clause either has a circular assignment or has no correct assignment at all). For the details of the algorithm, including how to find circular mode assignments, see [7].

4 Transformations

Constraints on modes may be imposed by the implementation as well as by their semantics. For example, a mode that makes an inbound node a descendant of an outbound node requires a top-down implementation that supports coroutining; other implementations have to prohibit such "upside-down" modes.

Definition 4.1 An assigned program P is a *sequential program* if in all arguments of all predicates, if a node is mapped to an output mode, then all descendants of that node are also mapped to an output mode.

Implementations that are unable to take a partially instantiated term and further instantiate it (by binding some of its variables) will outlaw predicate arguments that contain both input and output positions. We call a program in which all arguments of all predicates contain either only input modes or only output modes a *separated program* (referring to the separation of inputs from outputs in different arguments).

Definition 4.2 An assigned program P is a *separated program* if for every argument of every atom in the program, either the mode assignment maps all nodes in that argument to input modes or it maps all nodes in that argument to output modes.

Bottom-up implementations are even more strict. Because naive bottom-up computation cannot use any bindings given in the query, all the producers have to be in the body.

Definition 4.3 An assigned program P is an *output program* if in the head of every clause, every node of every argument is mapped to an output mode.

We will prove in section 5 that bottom-up evaluation of non-circular output programs does not require subsumption tests. To make this result useful, we need to be able to transform a wide class of programs to this form.

We do this transformation in two stages. First, we apply a new transformation called the *argument-splitting transformation* that takes possibly circular sequential programs and produces non-circular separated programs. Second, we apply the magic set algorithm to transform these programs further into non-circular output programs.

To make the presentation of the transformations and the theorems simpler, we will assume that the programs we handle are in *superhomogeneous* form. In this form as in homogeneous form, clause heads must contain distinct variables as arguments. However, in superhomogeneous form atoms in clause bodies must also contain distinct variables as arguments, and all unifications must be of the form X = Y or of the form X = f(Y1, ..., Yn). It is clear that all clauses can be rewritten into this form.

4.1 The argument splitting transformation

In the next example program, the `pred` declaration specifies the types of the arguments, and the `mode` declaration gives a valid mode. This mode says that iota1 should be called with a query such as ?- iota1(A.B.C.nil, 1), which will compute an answer such as iota1(1.2.3.nil, 1).

```
?- pred iota1(L: listint, N: int).
?- mode iota1([L:skel,N:root], [], [L:elt], []).
iota1(L, N) :- L = nil.
iota1(L, N) :- L = cons(N, T), plus(N, 1, M), iota1(T, M).
```

This program is circular because in the second clause, the unification L = cons(N, T) instantiates the skeleton of T before the recursive call to iota1 but transmits the values of the elements of T to L after that call.

The idea behind the transformation is to split the argument(s) that are used for both input and output (in this case, L), into two separate arguments, one for input and one for output. The result of applying the transformation to iota1 is iota2, whose first argument is all input while the second is all output:

```
?- pred iota2(L0: listint, L2: listint, N: int).
?- mode iota2([L0:skel,N:root], [L0:elt], [L2:skel,L2:elt], []).
iota2(L0, L2, N) :- L0 = nil, L2 = nil.
iota2(L0, L2, N) :- L0 = cons(_, T0), plus(N, 1, M),
                    iota2(T0, T1, M), L2 = cons(N, T1).
```

The query is transformed into ?- iota2(A.B.C.nil, L, 1), and its answer is iota2(A.B.C.nil, 1.2.3.nil, 1): the original list skeleton is untouched while iota2 creates another list of the same shape and fills it in. The circularity is gone too: the first unification instantiates T0, the recursive call creates T1 from T0 (and M), and the second unification creates L2 without touching L0. Therefore iota2 is non-circular and separated.

The mode of the argument replacing L in its input role (L0) is derived from the instantiation state of L in the query: whatever was bound (the skeleton) becomes inbound and whatever was notbound (the elements) becomes infree. The mode of the argument replacing L in its output role (L2) is derived from the instantiation state of L in the answer: whatever was bound (the skeleton and the elements) becomes outbound and whatever was notbound (in this case nothing) becomes outfree.

The transformation of a clause starts from the mode assignment of the clause. For each variable that has more than one instantiation state (apart from all notbound, which is the state variables are in before their first appearance), we create several replacement variables, one for each such instantiation state. Occurrences of the original variable in an atom, either in the head or in the body, fall into two classes. In an atom whose mode says that it *does not* cause the variable to become more instantiated, we substitute the replacement variable corresponding to the instantiation state the mode requires. In an atom whose mode says that it *does* cause the variable to become more instantiated, we substitute two replacement variables, corresponding to the initial and final instantiation states. (We count transitions from "all notbound" as falling into the first class.)

Unifications in which data flows one way present no problem. Unifications in which data flows two ways have to be split into two or more unifications in which data flows one way, one for each instantiation state to be achieved. The first unification in the second clause of iota2 binds T0 while the second binds L2. The details of the algorithm can be found in [8].

4.2 The magic set transformation

Naive bottom-up evaluation of logic programs does not make use of data provided in the query and is therefore very inefficient. The magic set transformation is the standard technique for restricting the set of tuples generated to the ones actually relevant to the query. It does this by building and using a "magic set", which is essentially a representation of the queries and subqueries on a predicate. It replaces each clause H :- $L_1, \ldots, L_n$ with another clause H :- magic_H, $L_1, \ldots, L_n$, and for each call to a derived predicate in (say) L_m it adds the clause magic_L_m :- $L_1, \ldots, L_m$. It also adds a clause for the query itself. The magic relations magic_L_m have one attribute for each *input* attribute of the predicate being called.

For the query ?- iota2(A.B.C.nil, L, 1), the magic set transformation gives:

```
?- pred magic_iota3(L0: listint, N: int).
?- mode magic_iota3([], [], [L0:skel,N:root], [L0:elt]).
magic_iota3(L0, N) :- L0 = cons(A, cons(B, cons(C, nil))), N = 1.
magic_iota3(T0, M) :- magic_iota3(L0, N), L0 = cons(_, T0),
                      plus(N, 1, M).

?- pred iota3(L0: listint, L2: listint, N: int).
?- mode iota3([], [], [L0:skel,L2:skel,L2:elt,N:root], [L0:elt]).
iota3(L0, L2, N) :- magic_iota3(L0, N), L0 = nil, L2 = nil.
iota3(L0, L2, N) :- magic_iota3(L0, N), L0 = cons(_, T0),
                    plus(N, 1, M), iota3(T0, T1, M),
                    L2 = cons(N, T1).
```

This program still handles partially instantiated data structures, but it is an output program and as such can be evaluated bottom-up. Neither iota1 nor iota2 could be evaluated bottom-up in finite time.

Proposition 4.1 *The result of applying the magic set transformation to a non-circular separated program and a query is a non-circular output program.*

The proof of this result appears in [8]. Previous results imply that with a well-founded SIPS (sideways information passing strategy) the well-founded models of the original and the transformed programs agree on the query.

The magic set transformation is not the only transformation that can turn non-output programs into output programs. Similar results ought to be possible for other such transformations, e.g. the context transformation [3]. This is important because such more specialized transformations can produce more efficient code.

5 Subsumption freedom for output programs

Each predicate in an assigned program has a single mode. This means that if an answer to query has a variable at a particular position selector, then *all* answers to that query must have a variable at that position selector, and that if an answer has a function symbol at a particular position selector, then *all* answers to that query must have a function symbol at that position selector. Therefore every answer will have the same basic "shape".

Definition 5.1 A set of terms $\mathcal{T}$ is *independent* if $\forall \mathrm{t}_1, \mathrm{t}_2 \in \mathcal{T}$, either t_1 is identical to t_2 up to renaming or t_1 does not unify with t_2.

Example 5.1 {a, f(X)} is independent, and so is {f(X), f(Y)}. However {f(a), f(X)} is not independent, and neither is {g(a, Y), g(Y, b)}.

Definition 5.2 We denote the set of all ground atoms as $\mathcal{H}$ and the set of all atoms as $\mathcal{H}'$. An *interpretation* is a subset of $\mathcal{H}'$.

Definition 5.3 Let I be a set of atoms. We refer to the set of all instances of all elements of I as $inst(I)$. Note that $I \subseteq inst(I)$. We define the relation $\sqsubseteq$ by $I_1 \sqsubseteq I_2$ iff $\mathrm{inst}(I_1) \subseteq \mathrm{inst}(I_2)$.

Note that interpretations form a complete lattice under the partial order $\sqsubseteq$. Note also that if $I_1 \sqsubseteq I_2$ and $I_2 \sqsubseteq I_1$, then I_1 and I_2 only differ by the names of variables.

In order to show that subsumption is not necessary for output programs, we use the fixpoint semantics for logic programs; in particular, we will show that each application of the T_P operator produces an *independent* set of atoms. Since the standard T_P operator can only generate ground atoms, we must introduce a non-ground version of the T_P operator that is careful about preserving the independence of the interpretations it manipulates. However, we first have to make sure that the first iteration of T_P generates some interpretations.

Definition 5.4 Given a superhomogeneous program P, we can derive a corresponding program $|P|$ that contains no explicit calls to =. Where P contains $X = term$, in $|P|$ we apply the substitution $\{\mathrm{X}/term\}$ to the clause containing that unification.

Definition 5.5 Let P be a program and I be an interpretation. Then we define

$$T_P(I) = \{A\theta \mid A \in |P| \text{ and } \theta = \{\},$$
$$\text{or A :- G} \in |P| \text{ such that } G\theta \subseteq I$$
$$\text{and } \theta \text{ binds only variables that occur in } G\}$$

$$T_P^\omega(I) = \bigcup_{i=1}^{\infty} T_P^i(I)$$

We have the following results, the proofs of which appear in [8]:

Proposition 5.1 *The least fixed point of T_P is $T_P^\omega(\emptyset)$.*

Theorem 5.1 *Let P be a non-circular output program. Then for any $k \leq \omega$, $T_P^k(\emptyset)$ is independent.*

The only way that two answers can have any instances in common is for them to have all instances in common, and thus to be identical. Therefore when we generate an answer during bottom-up evaluation, which essentially implements the T_P operator, we know exactly the place in the answer set we should look at to see if we have generated a variant of this answer before. And since no variable occurs twice in the answer set, we can replace all variables by the same new constant and use equality tests instead of subsumption.

Furthermore, since the modes specify exactly where in the structure variables will be found, this theorem lets us avoid storing them altogether, since one knows how to put them back in later. This is needed only if they occur in the final answer.

This approach requires us to omit certain argument positions, both from type declarations and from the code, as in this version of iota:

```
?- type listint1 ---> nil1 ; cons1(listint1).

?- pred magic_iota4(L0: listint1, N: int).
?- mode magic_iota4([], [], [L0:skel,N:root], []).
magic_iota4(L0, N) :- L0 = cons1(cons1(cons1(nil1))), N = 1.
magic_iota4(T0, M) :- magic_iota4(L0, N), L0 = cons1(T0),
                      plus(N, 1, M).

?- pred iota4(L0: listint1, L2: listint, N: int).
?- mode iota4([], [], [L0:skel,L2:skel,L2:elt,N:root], []).
iota4(L0, L2, N) :- magic_iota4(L0, N), L0 = nil1, L2 = nil.
iota4(L0, L2, N) :- magic_iota4(L0, N), L0 = cons1(T0),
                    plus(N, 1, M), iota4(T0, T1, M),
                    L2 = cons(N, T1).
```

This version manipulates only ground data structures, and as such can be executed on any current deductive database.

6 Conclusion

Mode declarations give us precise and reliable information about which parts of its arguments a predicate expects to be bound in queries to it and which parts it will itself instantiate. We can use this information to separate the input and output parts via the argument splitting transformation, and then make *all* arguments output via the magic set transformation. The resulting program can then be evaluated bottom-up without subsumption tests.

We would like to thank Will Winsborough for conversations that clarified some of the concepts in section 2, the referees for their comments, and the Australian Research Council and Centre for Intelligent Decision Systems for their support.

References

[1] F. Bancilhon, D. Maier, Y. Sagiv, and R. Ramakrishnan. Magic sets and other strange ways to implement logic programs. *Proceedings of the Symposium on Principles of Database Systems*, pages 1–15, 1986.

[2] P. M. Hill and J. W. Lloyd. The Gödel report. Technical Report 91-02, Department of Computer Science, University of Bristol, England, 1991.

[3] D. Kemp, K. Ramamohanarao, and Z. Somogyi. Right-, left- and multi-linear rule transformations that maintain context information. In *Proceedings of the Sixteenth International Conference on Very Large Data Bases*, pages 380–391, Brisbane, Australia, 1990.

[4] R. Milner. A proposal for standard ML. *Conference Record of the ACM Symposium on LISP and Functional Programming*, pages 184–197, 1984.

[5] A. Mycroft and R. A. O'Keefe. A polymorphic type system for Prolog. *Artificial Intelligence*, 23:295–307, 1984.

[6] R. Ramakrishnan, P. Bothner, D. Srivastava, and S. Sudarshan. CORAL - a database programming language. In *Proceedings of the NACLP '90 Workshop on Deductive Databases*, pages 2.1–2.9, Austin, Texas, 1990.

[7] Z. Somogyi. A system of precise modes for logic programs. In *Proceedings of the Fourth International Conference on Logic Programming*, pages 769–787, Melbourne, Australia, 1987.

[8] Z. Somogyi, D. Kemp, J. Harland, and K. Ramamohanarao. Subsumption-free bottom-up evaluation of logic programs with partially instantiated data structures. Technical Report 93/26, Department of Computer Science, University of Melbourne, Melbourne, Australia, 1993.

[9] J. Thom and J. Zobel. NU-Prolog reference manual, version 1.0. Technical Report 86/10, Department of Computer Science, University of Melbourne, Melbourne, Australia, 1986.

[10] J. Vaghani, K. Ramamohanarao, D. Kemp, Z. Somogyi, and P. Stuckey. Design overview of the Aditi deductive database system. In *Proceedings of the Seventh International Conference on Data Engineering*, pages 240–247, Kobe, Japan, 1991.