Abstract

In this paper the problem of extending the logic database language Datalog with primitives to support array definitions and manipulations is addressed. The syntax and the semantics of this language, called DatalogA, are given by showing that model theoretic properties of ordinary Datalog extend to DatalogA. DatalogA fixpoint semantics and its efficient implementation are also studied and presented. Sufficient conditions assuring program evaluation convergence when manipulating real-valued arrays are finally discussed.

1 Introduction

Many research efforts have been recently spent towards extending the capabilities of data models for databases to capture more semantics of data, in order to improve their amenability for a wider class of applications. The proposed extensions to the conventional data model (i.e., sets of flat tuples) include nested relations, complex objects with or without object identity, bags and lists. However, as clearly pointed out in a recent paper by D. Maier and B. Vance [5], these extensions are "natural" generalizations of the relational model. For this reason, though being effective in new application domains of database systems (e.g., CAD), they still lack in representational capabilities for other important contexts that could otherwise take advantage of database technologies. For instance, many algorithms used in the realm of scientific computation cannot be properly implemented using data models which do not support indexed direct accesses to stored data [5].

The primary objective of this paper is to provide a clean formalization of the semantics and the implementation of a database language supporting this kind of data structures. In particular we have chosen the Datalog language as the framework in which to tackle our study about the problem of adding indexed-based data structures to a set-oriented declarative language. This choice is motivated by the consideration that Datalog is a powerful database language (due to the possibility of programming general recursions), but retains at the same time the structural simplicity and cleanliness of other less powerful languages, like the relational algebra and calculus.

It could be objected that the algorithms that use index structures could be anyway implemented using the available data structures, as many of the database programming environments support (Turing) complete languages. This is certainly true. However, it is important to point out that efficiency, which is often a fundamental constraint for algorithms, is not always to be guaranteed with this implementation. For instance, it is certainly possible to store a vector in a LDL: list, and then scan its element using the cons functor. However, if an algorithm needs to access the central location of a vector of dimension n, it will need n/2 accesses, on the average, to obtain it. Counterwisely, our aim is to design a database language featuring indexed-based structures as first-class citizens amongst the other manipulated data (in particular, relational structures).

As already pointed out, in this paper we will, in particular, consider the logic database language Datalog and propose an extension of this language with primitives allowing a programmer to specify array definitions and manipulations. As we shall show, array manipulation primitives enhance Datalog representational capabilities, making it possible to naturally express many well-known algorithms, that have wide applicability.

The language we are presenting here, called
Datalog\textsuperscript{A}, supports array terms in predicates, and allows the user to manipulate them in a natural way, using the powerful paradigm of logic programming. Arrays are defined using schema definitions, one for each predicate of a Datalog\textsuperscript{A} program. Schema definitions dictate the structure of predicate arguments. Interpreted functions are used to manipulate both indexes and array entries. For methodological reasons, however, we will focus our attention on a “kernel” language first, in which interpreted functions can only be used for index manipulations (in other words, array entries will be treated as uninterpreted constants). We will then relax this constraint, specifically addressing the case of real-valued array entries (see Section 5).

To give an intuitive idea of the kind of computations that Datalog\textsuperscript{A} allows to express, we present next some application examples. In the examples, we shall make use of two predefined atoms:

- the predicate \texttt{sire(A, I, N)}, which is true if \(N\) is the size of the \(I\)-th dimension of the array \(A\);
- the atom \texttt{each(I, H, K)}, which is true whenever \(I\) is an integer value in between \(H\) and \(K\).

We point out that the complexity of our programs is achieved after they are rewritten into an optimized equivalent form (see Section 4 below).

**Example 1:** Binary search. The predicate \texttt{search(X, V)} is true if the element \(X\) is in the array \(V\). The predicate \texttt{bin_src} searches for \(X\) in the array \(V\).

\[
\text{search}(X, V) \leftarrow \text{sire}(V, 1, N), \text{bin_src}(X, V, 1, N).
\]
\[
\text{bin_src}(X, V, I, J) \leftarrow K = (I+J) \div 2, X = V[K].
\]
\[
\text{bin_src}(X, V, I, J) \leftarrow K = (I+J) \div 2, X < V[K], J_1 = K-1, \text{bin_src}(X, V, I, J_1).
\]
\[
\text{bin_src}(X, V, I, J) \leftarrow K = (I+J) \div 2, X > V[K], I_1 = K+1, \text{bin_src}(X, V, I_1, J).
\]

The achieved time complexity is \(O(\log_2n)\), where \(n\) is the size of the input array.

**Example 2:** Transpose of a matrix. The predicate \texttt{tmatrix(A, B)} constructs the transpose of the matrix \(A\) in the matrix \(B\).

\[
l_1 : \text{tmatrix}(A, B) \leftarrow \text{sire}(A, 1, M), \text{each}(I, 1, M), \text{trow}(A, B, 1).
\]
\[
l_2 : \text{trow}(A, B, I) \leftarrow \text{sire}(A, 2, N), \text{each}(J, 1, N), B[I, J] = A[J, I].
\]

If the size of the input matrix is \(m \times n\), then the time complexity of evaluating the Datalog\textsuperscript{A} rules above is in \(O(mn)\).

The proofs of the results which follow, a comparison with related works and a substantial number of examples can be found in the full paper [4].

## 2 The Datalog\textsuperscript{A} language

### 2.1 Syntax

In this section we illustrate the syntax of the Datalog\textsuperscript{A} language. We assume countable sets of constants, variables and predicate symbols. For any program \(P\), included in the set of constants, we assume the existence of a subset of the cardinal numbers \(\{1 \ldots n_P\}\) denoted by \(\text{card}(P)\). Intuitively, \(n_P\) will be the maximum allowed index for any array manipulated in the given Datalog\textsuperscript{A} program \(P\). This subset of constants is especially treated in Datalog\textsuperscript{A}. Indeed, a set of interpreted arithmetic binary functions \(F\) (such as sum, difference, product etc.) is associated to this set. In other words, the set \(F\) contains the usual arithmetic operators, but restricted on \(\text{card}(P)\). For the time being, interpreted functions do not apply to constants not belonging to \(\text{card}(P)\). This constraint will be relaxed in Section 5.

Next, we give the definition of Datalog\textsuperscript{A} term:

**Definition 1:** A Datalog\textsuperscript{A} term is inductively defined as follows:

- constants and variables are (simple) terms;
- if \(t_1, \ldots, t_n\) are terms then \(<t_1, \ldots, t_n>\) is a (array) term;
- if \(A\) is a variable or an array term and \(i\) is variable or a cardinal number then \(A[i]\) is a term;
- if \(f \in F\), and \(i_1\) and \(i_2\) are cardinal numbers, then \(f(i_1, i_2)\) is a (function) term.

**Notation:** We shall often use the infix notation \(i_1 f i_2\) for function terms. Furthermore, we shall use the notation \(A[i_1, \ldots, i_k]\) as a shorthand for \(A[i_1][i_2] \ldots [i_k]\). Let \(A\) be a \(n\)-dimensional array term, then \(A[i_1, \ldots, i_k] (k \leq n)\) denotes the \(n-k\) dimensional array term obtained from \(A\) by fixing the first \(k\) indexes. Thus, \(A[i_1, \ldots, i_n]\) denotes a single element of \(A\).

A Datalog\textsuperscript{A} atom has the form \(p(t_1, \ldots, t_n)\) where \(p\) is a predicate symbol and \(t_1, \ldots, t_n\) are terms. In particular, we assume the existence of the following built-in atoms:

1. \texttt{each(I, H, K)} that is true if the relation \(H \leq I \leq K\) holds, where \(H, I, K\) are cardinal numbers.
2. \texttt{size(A, I, N)} which is true whenever \(N\) is the size of the \(I\)-th dimension of the array \(A\).

A Datalog\textsuperscript{A} rule has the form

\[
A \leftarrow B_1, \ldots, B_n
\]
where $A$, the head of the rule, is an atom and $B_1, \ldots, B_n$, the body of the rule, is a conjunction of atoms, comparison predicates and built-in atoms. A rule containing an each atom is called each-rule, otherwise it is called standard rule.

We impose the constraint that an each-rule body contains one only each atom and has the following form:

$$A \leftarrow B_1, \ldots, B_n, \text{each}(I, H, K)$$

where $I$ is a variable not appearing in $A$ and $H$ and $K$ are constants or variables appearing in $B_1, \ldots, B_n$.

The rationale for this constraint we are imposing on the form of the each-rules is motivated by the consideration that the each-atoms intuitively implement for-loop iterations. Then, to allow more than one such atom in the body of a rule would have implied the necessity to impose an order on the evaluation of body atoms, thus losing much in terms of declarativity. On the other hand, nothing is lost in terms of expressive power, as nested for-loops, often used when handling arrays, correspond to sets of dependent each-rules, each of which implements one of the nested loops. It is then quite clear that this constraint does not limit the expressivity of the language.

The definition of a predicate symbol $p$ is the set of rules with head predicate symbol $p$. A Datalog$^A$ program is the union of the definition of a finite set of predicate symbols. A Datalog$^A$ query is a pair $(P, Q)$ where $P$ is a Datalog$^A$ program and $Q$ is a Datalog$^A$ atom, called (query-)goal, whose predicate symbol appears in the head of a rule of $P$.

A term, an atom or a rule is ground if it does not contain any variables or function terms (this means that each function (sub)term, if any, has been substituted by its image). Given a Datalog$^A$ program $P$ the Herbrand universe $U_P$ of $P$ consists of all ground terms that can be constructed by using constants in $P$ and the array constructor $<>$. The Herbrand base $B_P$ of $P$ is built by combining predicate symbols of $P$ and ground terms of $U_P$.

Definition 2: Let $p$ be a predicate symbol. The schema of $p$, denoted $sch(p)$, consists of the predicate symbol $p$ and a list of attributes. An attribute consists of the specification of the structure and the dimensions of the corresponding argument. Thus, an attribute has either the form flat (denoting a simple argument) or $[i_1, \ldots, i_k]$ where $i_1, \ldots, i_k$ are cardinal numbers. Given a program $P$, the associated program schema, denoted $sch(P)$, consists of the schemas of all predicate symbols appearing in $P$.

Let $P$ be a Datalog$^A$ program. We assume that $\text{card}(P)$ is equal to the set $\{1, n_P\}$ where $n_P$ is the maximum dimension appearing in $sch(P)$.

### 2.2 Declarative Semantics

Here we present the model theoretic semantics of Datalog$^A$ programs. Before that, we introduce a transformation of programs with each-rules into programs containing only standard rules. This will allow us to define the semantics of Datalog$^A$ by referring to each-rule-free programs only.

Definition 3: Let $r$ be an each-rule of the form

$$p(X) \leftarrow B(Y), C(Z, I), \text{each}(l, N, K)$$

where $B(Y)$ is the conjunction of body atoms not containing the variable $I$, $C(Z, I)$ denotes the conjunction of atoms containing $I$, $X, Y$ are lists of variables and $H, K$ are constants or variables appearing elsewhere in the body. Then the set of standard rules $st(r)$ derived from $r$ is:

- $p(X) \leftarrow B(Y), \text{each}_r(Z, K, K)$
- $\text{each}_r(Z, H, K) \leftarrow C(Z, H)$
- $\text{each}_r(Z, I, K) \leftarrow \text{each}_r(Z, I-1, K), I \leq K, C(Z, I)$

If $r$ is a standard rule then $st(r) = \{r\}$.

Now, given a Datalog$^A$ program $P$, the standard form of $P$, denoted $st(P)$, is defined as $\cup_{r \in P} st(r)$. Observe that given a rule $r$ defining a predicate $p$, the each$_r$ atoms have also associated a schema which can be automatically deduced from the schema of $p$.

The semantics of a Datalog$^A$ program $P$ is defined equal to the semantics of $st(P)$.

Example 3: Consider the program of Example 2 that computes the transpose of a matrix. Its corresponding standard form is as follows:

$$\text{tmat}(A, B) \leftarrow \text{size}(A, 1, M), \text{each}_1(A, B, M, M),$$
$$\text{each}_1(A, B, 1, M) \leftarrow \text{trow}(A, B, 1),$$
$$\text{each}_1(A, B, I, M) \leftarrow \text{each}_1(A, B, I-1, M), I \leq M,$$
$$\text{trow}(A, B, I, 1, N),$$
$$\text{each}_2(A, B, I, J, N) \leftarrow \text{each}_2(A, B, I, J-1, N), J \leq N, B[I, J] = A[J, I].$$

From now on, whenever talking about programs, we shall intend programs in standard form. Let $A = p(e_1, \ldots, e_n)$ be a ground atom and $p(T_1, \ldots, T_n)$ be

\[\text{The rationale here is that all indexes have limited and fixed ranges (see also below).}\]
the schema of \( p \). Then we say that \( A \) is consistent with respect to the schema of \( p \) if each \( t_i \) is either a simple term and \( T_i = \text{flat} \) or \( t_i \) is an array whose structure and size are as specified by \( T_i \).

**Definition 4:** Let \( P \) be a DatalogA program with schema \( \text{sch}(P) \). The restricted Herbrand base \( \hat{B}_P \) of \( P \) is defined as follows:

\[
\hat{B}_P = \{ p(t) \mid p(t) \in B_P \text{ and } p(t) \text{ is consistent w.r.t. } \text{sch}(p) \}.
\]

Thus, the restricted Herbrand base contains only the ground atoms of \( B_P \) which are consistent w.r.t. the corresponding schema. A ground instance rule \( r \) of a program \( P \) is consistent w.r.t. \( \text{sch}(P) \) if each atom \( p(t) \) appearing in \( r \) is consistent w.r.t. \( \text{sch}(p) \). The consistent instance of \( P \), denoted \( \text{cground}(P) \), is the set of all consistent ground instances of rules in \( P \). Analogously, let \( I \) be a set of atoms, \( \text{cground}(I) \) denotes the set of all consistent ground instances of atoms in \( P \).

**Definition 5:** An interpretation \( I \) for a DatalogA program \( P \) is any subset of the restricted Herbrand base \( \hat{B}_P \) of \( P \). A ground rule \( A \leftarrow B_1, \ldots, B_n \) is satisfied w.r.t. \( I \) if \( A \in I \) or if there exists a \( D_i (1 \leq i \leq n) \) such that \( B_i \notin I \). An interpretation \( I \) is a model for \( P \) if all rules of \( \text{cground}(P) \) are satisfied.

Let \( I \) be an interpretation. A ground atom \( A \) is true with respect to \( I \) if \( A \in I \). The truth valuation of comparison predicates is the obvious one.

**Example 4:** Consider the program \( P \) whose associated schema is \( \{ p(\text{flat}, \{3\}), q(\text{flat}, \{3\}) \} \) and containing the following rules:

\[
q(40, < 10, 20, 30 >).
p(x, < x_3, x_2, x_1 >) \leftarrow q(x, < x_1, x_2, x_3 >).
\]

The cardinal numbers used as indexes for array terms is \( \text{card}(P) = \{1, 2, 3\} \). The Herbrand Universe consists of the constants in \( \text{card}(P) \), the constants 10, 20, 30, 40 plus the array terms built by using the constants and the array constructor.

The Herbrand base \( B_P \) consists of all ground predicates built using the binary predicate symbols \( p \) and \( q \) and the ground terms of the Herbrand universe. The restricted Herbrand base \( \hat{B}_P \) of \( P \) contains only the ground atoms of \( B_P \) whose first argument is a simple constant and whose second argument is a monodimensional array of size 3.

**Proposition 1:** Let \( P \) be a DatalogA program. Let \( \mathcal{M}_P \) denote the family of all the models for \( P \). Then \( \mathcal{L}_M = \cap_{M \in \mathcal{M}_P} \) is the unique minimal model of \( P \). \( \Box \)

Thus, the intersection model property of Datalog programs is preserved in DatalogA. Therefore, as for standard Datalog, for any DatalogA program \( P \), \( \mathcal{L}_M \) will denote its intended semantics.

Before closing this section it is worth pointing out that the declarative semantics we have given above implies that DatalogA supports only a form of non-destructive assignment to array entries: if more than one value is to be assigned to the same array entry, then one distinct array term is "computed" for each of these (distinct) values. This is opposed to have a language supporting destructive assignments, like the Coral language [8].

**2.3 Fixpoint Semantics**

With the introduction of array terms the classical fixpoint procedure [10] is to be modified. Indeed, the fixpoint computations associated to DatalogA programs will in general start with atoms whose array arguments (if any) are only partially specified, i.e., some of their entries are undefined. Values for these entries are then computed during the various steps constituting the fixpoint computation.

Therefore, differently from the case of ordinary Datalog programs, our fixpoint procedure computes non-ground terms in its intermediate steps. As a consequence, most general unifiers (m.g.u.) are computed in the place of matchers.

Let \( P \) be a DatalogA program. \( (2^{\hat{B}_P}, \subseteq) \), i.e., the set of all restricted Herbrand interpretations for \( P \) under the usual set inclusion, is a complete lattice. The immediate consequence operator \( T_P : 2^{\hat{B}_P} \rightarrow 2^{\hat{B}_P} \) defines a mapping between interpretations and is defined as follows:

\[
T_P(I) = \{ A \mid A \leftarrow B_1, \ldots, B_n \in \text{cground}(P) \text{ and } B_1, \ldots, B_n \in I \}
\]

**Proposition 2:** Let \( P \) be a DatalogA program. The mapping \( T_P \) is continuous and, therefore, also monotonic over the complete lattice \( (2^{\hat{B}_P}, \subseteq) \). Moreover, for each \( I \in 2^{\hat{B}_P} \), if \( T_P(I) \subseteq I \) then \( I \) is a model for \( P \). \( \Box \)

**Theorem 1:** Let

\[
\begin{align*}
T_0^P & = \emptyset \\
T_{n+1}^P & = T_P(T_n^P) \\
T_{\infty}^P & = \bigcup_{n=0}^{\infty} T_n^P
\end{align*}
\]
then $T_p^\infty = LM_p$. Moreover $T_p^k = T_p^\infty$ for some finite $k$. \hfill \Box

3 Evaluating Datalog$^A$ programs

3.1 Computation

We partition the set of predicate symbols into two sets. Predicates whose definition consists of only ground facts are called base predicates while all others predicates are called derived. The set of facts whose predicate symbol is a base predicate defines the database while the set of clauses whose head predicate symbols is a derived predicate symbol defines the query program. In the following given a Datalog$^A$ program $P$ we denote with $P_Q$ and $P_D$ the query program and the database derived from $P$ ($P = P_Q \cup P_D$).

We assume that the query program, say $P_Q$, which we want to evaluate on the database $P_D$, has been partitioned according to a topological order $<P_1, \ldots, P_n>$ such that each two predicate symbols $p$ and $q$ defined in the same component $P_i$ are mutually recursive. This means that each atom appearing in $P_i$ depends only on atoms belonging to $P_j$ such that $j \leq i$. We assume also that the computation follows the topological order so that when we compute the component $P_i$ the components $P_1, \ldots, P_{i-1}$ have been already computed. When we compute the component $P_i$ all the atoms obtained from the computation of the components $P_1, \ldots, P_{i-1}$ are basically treated in the same way as database facts. A rule in a component $P_i$ is called exit rule if each predicate in its body belongs to a component $P_j$ such that $j < i$. All the other rules are recursive rules.

As already pointed out, in the bottom-up evaluation of Datalog$^A$ rules, array terms are stepwise evaluated. Therefore, in a generic step of the evaluation, we will, in general, derive also non ground atoms, whose array arguments are only partially instantiated. As a consequence, the usual matching mechanism employed in the fixpoint computation of Datalog programs cannot be utilized in Datalog$^A$. In fact, general unification must be used to provide the needed (not necessarily ground) instantiation of Datalog$^A$ rules during the fixpoint computation. Thus, we define the operator $V_p$ which derives sets of (not necessarily ground) atoms as follows:

$$V_p(I) = \{ \theta | \sigma[A \leftarrow B_1, \ldots, B_n] \in I, \theta \text{ is a m.g.u. and } \exists \sigma \text{ a matcher s.t. } A\theta \sigma \leftarrow B_1\theta \sigma, \ldots, B_n\theta \sigma \in cground(P) \}$$

Thus, the operator $V_p$ computes a set of atoms whose instantiations are consistent with respect to the schema. A rule $A \leftarrow B_1, \ldots, B_n$ is said to be fireable w.r.t. a set of atoms $I$ if there is a substitution $\theta$ such that $B_1\theta, \ldots, B_n\theta \in I$.

Observe that, if the program $P$ is ground then the operators $V_p$ and $T_p$ coincide, that is $V_p(I) = T_p(I)$ for all interpretations $I$. This implies also that if $P$ is not ground $V_{cground(P)}(I) = T_p(I)$ since the ground instantiation of $P$ is implicit in the definition of the operator $T_p$. The fixpoint of a Datalog$^A$ program is computed by the following algorithm:

**Algorithm Fixpoint**

**Input:** Datalog$^A$ program $P = (P_Q, P_D)$ where $P_Q = <P_1, \ldots, P_n>;$

**Output:** Set of atoms $M$;

begin

$M := P_D$;

for $i := 1$ to $n$ do

$M := M \cup \text{solve-component}(P_i, M)$;

end.

where the function solve_component computing the fixpoint of a single component is as follows:

**Function solve_component(P : set of rules; D : set of atoms) : set of atoms;**

begin

$I := 0$;

repeat

$J := I$;

$I := I \cup V_p(J \cup D)$;

until ($J = I$);

return $J$

end.

Let $P = (P_Q, P_D)$ be a Datalog$^A$ program and let $G$ be an atom. A query is a pair $(G, P_Q)$ where $G$ is called query-goal. Let $FM_p$ be the set of atoms computed by the Fixpoint algorithm.

**Definition 6:** An answer $\theta$ for the query $Q = (G, P_Q)$ on a database $P_D$ is a substitution for the variables of $G$. A correct answer $\theta$ for the query $(G, P_Q)$ is an answer such that $\forall G \theta. \sigma$ is a logical consequence of $P$. A computed answer $\theta$ for the query $(G, P_Q)$ is an answer such that $G\theta \in FM_p$.

**Proposition 3:** Let $P$ be Datalog$^A$ program and let $G$ be an atom. A substitution $\theta$ such that $G\theta$ is ground is a correct answer for $(G, P_Q)$ iff $G\theta \in LM_p$. \hfill \Box

The soundness and completeness of a fixpoint algorithm guarantee that the tuples in the set computed by the algorithm are correct answers and for each correct answer $\theta$ there is an element $\sigma$ computed by the
algorithm and a unifier $\delta$ such that $\theta = \sigma \delta$. That is, each correct answer is subsumed by a (not necessarily ground) fact of the set computed by the fixpoint algorithm. We point out that $FM_P = V_P^\infty$ since the Fixpoint Algorithm, as for $V_P^\infty$, applies the rules of $P$ until saturation.

**Theorem 2:** The Fixpoint Algorithm is sound and complete for Datalog$^A$ programs.

### 3.2 Safe queries

In the previous subsection we have described how the minimal model of a program is constructed by repeatedly applying the rules until no further new atoms are produced. Some of the atoms computed could not be ground and this means that there is a domain of some atom which is not limited. On the other hand, for some queries, although the fixpoint computation generates unground atoms in intermediate steps, the answer is well-defined, that is, it contains only ground substitutions.

Thus, it is interesting to define sufficient conditions singling out this kind of queries, i.e., which guarantee that computed answers contain only ground substitutions. We call such a class of queries safe. We recall that in the general case (programs with function symbols) testing a program for safety is undecidable [9] while for Datalog program is trivial [10]. For general Datalog$^A$ programs testing for safety is also undecidable since the values of the indexes could depend on the database instance.

We first introduce the concept of range restricted variable which is a variable whose associated domain is finite [10].

A variable $X$ in a rule $r$ is range restricted if one of the following conditions holds:

1. it appears in a body atom;
2. there is a predicate $\text{size}(A, N, X)$ or a predicate $\text{each}(X, J, K)$ in $r$;
3. there is a predicate $X = Y$ (or $Y = X$) such that $Y$ is a constant or a range restricted variable;
4. there is a predicate $X = Y$, where $Y$ is an expression such that all its variables are range restricted;
5. $X$ is an array variable whose first dimension is $n$ and $X[1], ..., X[n]$ are range restricted.

We recall that a Datalog rule is said to be safe if each variable is range restricted [10]. Thus, to extend the class of safe programs to Datalog$^A$ programs we need some preliminary transformation.

Given a Datalog$^A$ program $P$ then the expanded version of $P$, denoted by $P^e$, is a Datalog program derived as follows:

1. the built-predicates $\text{size}$ are deleted;
2. each rule of the form
   \[ p(\vec{X}) \leftarrow B(\vec{Y}), C(\vec{Z}, I), \text{each}(I, H, K) \]
   is substituted by
   \[ p(\vec{X}) \leftarrow B(\vec{Y}), C(\vec{Z}, H), ..., C(\vec{Z}, K). \]
3. Each non-flat variable $A$ whose first dimension is $n$ is substituted by the $n$ variables $A_1, ..., A_n$; this process is iterated until all variables are flat.
4. Each variable $A[i]$ is substituted by $A_i$; this process is iterated until all indexed variables are substituted.

For instance, given following program $P$ where $\text{sch}(p) = p(<10>)$

\[ p(A) \leftarrow \text{size}(A, 1, N), \text{each}(I, 1, N), A[1] = 0. \]

The expanded version $P^e$ is as follows:

\[ p(A_1, ..., A_{10}) \leftarrow A_1 = 0, ..., A_{10} = 0. \]

An adorned program is a program where each atom has associated a string $\alpha$, defined on the alphabet $\{b, f\}$, of length equal to the arity of the predicate symbol. A character $b$ (resp. $f$) in the $i$-th position of the adornment associated with an atom $A$ means that the $i$-th argument of $A$ is bound (resp. free).

Let $p(t_1, ..., t_k)$ be the head atom of a rule. Then the associated adornment $\alpha$ is derived as follows:

1. $\alpha_i = b$ if $t_i$ is a constant;
2. $\alpha_i = b$ if $t_i$ is a variable and $t_i$ occur in a body predicate whose associated adornment is $b$;
3. $\alpha_i = f$ otherwise (that is, if $t_i$ is a variable and all occurrences of $t_i$ in the body of the rule are adorned with $f$).

Observe that for each rule $r$ there should be more than one adorned rules derived from it.

**Definition 7:** An adorned rule is safe if the head adornment does not contains $f$ symbols. An rule $r$ is safe if all adorned rules derived from $r$ are safe. A Datalog query $(p(t), P)$ is safe if all rules in $P^e$ defining the predicate symbol $p$ are safe. A Datalog$^A$ query $(p(t), P)$ is safe if the associated Datalog query $(\rho(t), P^e)$ is safe.
Observe that most of the programs we have presented are not safe. Thus, the range restricted conditions which we have illustrated seem to be too restrictive. A first way to go beyond safe programs, in order to give semantics also to a class of unsafe programs, is based on the top-down computation typical of Prolog. We do not explore such an alternative since in the next section, we shall show that the same result can be obtained by rewriting our programs using well known rewriting techniques which propagate the bindings of the query goal inside the program in order to optimize the fixpoint computation; these techniques include the pushing constant, the magic-set, the supplementary magic-set and others (see [10]). This issue will be addressed in the next section.

4 Optimization

Most of the optimization techniques which have been defined for Datalog can be extended also to DatalogA programs. Such techniques are based on the rewriting of the original program into a new program which can be executed more efficiently by the fixpoint algorithm. The optimization is essentially based on the reduction of the number of tuples generated by the fixpoint computation and on the reduction of their arity. We point out that the reduction of the arity becomes a very important task in the presence of complex term.

In this section we show how the pushing of constants and the magic-set method can be profitably extended to DatalogA programs. Here, we shall adopt a rather informal presentation style.

4.1 Pushing constants

Consider the following program computing the reverse of a vector with the query goal of the form \texttt{reverse($A,B$)} where the symbol "$" indicates that the argument is bound:

\begin{verbatim}
reverse(A,B) ← size(A,1,N), each(A,B,N,N).


bound($A$).
\end{verbatim}

The query can be answered by computing the entire model and by selecting the tuples of \texttt{reverse} with the first argument equal to $A$.

There are two problems with this approach: first, it is very inefficient, and second, the program is not safe since there are variables which are not range restricted. For instance, the variable $A$ used in the predicate $B[1] = A[N+1-I]$ of the exit rule defining the predicate \texttt{each} do not appear elsewhere in the body.

In this case the selection of the tuples of \texttt{reverse} having the first argument equal to $A$ can be pushed inside the program, eliminating a large number of irrelevant tuples during the computation. Such an optimization can be carried out by adding a predicate of the form \texttt{bound($A$)} in the body of the rules, where \texttt{bound} is a new predicate symbol whose definition consists of the fact \texttt{bound($A$)}. The resulting query equivalent program is then

\begin{verbatim}
reverse(B) ← bound(A), size(A,1,N), each(A,B,N,N).

eachr(B,1,N) ← bound(A), size(A,1,N), B[1] = A[N+1-I].


bound($A$).
\end{verbatim}

The predicates of the rewritten program have less arguments with respect to the predicates of the standard program. Therefore, other than reducing the number of tuples used we have also reduced their size. In a similar manner, we are able to optimize any other DatalogA program, as long as we know the structure of the query (in terms of which arguments come bound and which come unbound). As a final remark, we would point out the naturalness of the application of the pushing constant method to DatalogA programs.

4.2 Magic-set rewriting

The magic sets method provides an efficient fixpoint based computation of Horn clauses in the presence of bound arguments in recursive predicates [1]. This technique can still be used for optimising DatalogA recursive rules, by reconsidering the binding-passing property.

The pushing constant method presented above leaves room for further optimizations. For instance, the program presented in the previous subsection is still not safe since the variable $N$ in the body of the rules defining \texttt{each} is not range restricted. The magic-set rewriting method further optimizes the program...
by propagating the bindings inside the program and makes others variables range restricted. For the program at hand, the predicate size(A, I, N) provides a binding for the variable N. This binding is propagated inside the definition of eachr yielding the following safe program:

```
magic_eachr(N, N) ← bound(A), size(A, 1, N).
magic_eachr(I-1, N) ← magic_eachr(I, N), I-1 > 0.
reverse(B) ← bound(A), size(A, 1, N), eachr(B, N, N).
eachr(C, 1, N) ← magic_eachr(I, N), bound(A),
eachr(C, I, N) ← magic_eachr(I, N), bound(A),
                 eachr(C, I-1, N), I ≤ N,
                 R[I] = A[N+1-I].
```

The magic-set rewriting is carried out after a first rewriting of the program in order to reduce the arity of the predicates (applying the pushing constant technique, or any other equivalent method). Observe that it is also possible to use other rewriting techniques such as the supplementary magic-set method [2] or others less general methods which are applicable to restricted classes of programs. These include factorization techniques such as those based on the reduction of the program [6] or on the combination of the propagation of bindings with the successive reduction [7], the counting method [1] and the pushdown method [3].

Special techniques are important since, although they are applicable to some special classes of programs only, they often yield and order of magnitude of improvement in efficiency.

5 Handling interpreted array entries

Many applications use real numbers. Since the case of real-valued array entries is very relevant in practice, we discuss this specific case in relaxing the constraint on having only uninterpreted constants as array entries. So, assume now that Datalog\(^A\) programs manipulate real-valued arrays, providing several interpreted functions to operate on their entries. As we shall see, the main problem to be faced in this extended context is that of the convergence of program evaluations. Indeed, we shall define restrictions that assure the termination of the evaluation.

A set of interpreted arithmetic binary functions \(G\) (such as sum, difference, product etc.) is associated to the domain of the real numbers \(R\)^2. Interpreted functions from \(G\) do not apply to constants not belonging to \(R\). Therefore, the definition of term is extended as follows.

**Definition 8:** Besides those defined in Definition 1, the following are also terms:

1. constants from \(R\) are Datalog\(^A\) (simple) terms,
2. if \(g \in G\) and \(i_1\) and \(i_2\) belong to \(R\), then \(g(i_1, i_2)\) is a Datalog\(^A\) (function) term.

The introduction of the real domain implies the existence of programs whose least model computation requires an infinite number of application of the immediate consequence operator (note that Datalog\(^A\) computations are counterwisely guaranteed to terminate in a finite number of steps, see Theorem 1 above). An example follows:

**Example 5:** Consider the following simple program generating the succession \(\{1/2^n\}, n \geq 0\).

\[ p(1),
   p(x) ← p(y), x = y/2, x \geq 0. \]

In this case the computation requires an infinite number of applications of the \(T_p\) operator (for this program, the mapping \(T_P\) and \(V_P\) coincide) to obtain the atom \(p(0)\), that indeed belongs to \(LM_P\). □

Therefore, it is important to single out a non-trivial class of Datalog\(^A\) programs manipulating real-valued arrays, whose least model can be finitely computed. This issue is addressed next.

**Definition 9:** Given a program \(P\) and a \(n\)-ary predicate \(p\) appearing in \(P\), we say that an attribute \(a_i\) of \(sch(p)\), \(1 \leq i \leq n\), is an index attribute if the \(i\)-th argument of \(p\) takes values only from \(card(P)\).

It is easy for a compiler to understand when an argument takes its values only from \(card(P)\). For instance, a predicate of the form \(size(A, I, N)\) assigns to \(N\) a cardinal value; an expression which uses cardinal numbers and the integer operators (+, −, *, div and mod) is a cardinal number if its result is in \(card(P)\). Consider the program of Example 1. The third and fourth arguments of \(bin\_src\) are index attributes since the values assigned to the variables appearing in these arguments are either cardinal constants, or cardinal values generated by a predicate \(size\) (in the rule defining the predicate \(search\)) or by expressions whose result is a cardinal number (in the second and third rules defining the predicate \(bin\_src\)). □

of convenience. Functions with different arities could be freely
used, without any consequence on the validity of the considerations
that follow.
We now define the notion of monotonic program using dependency graphs. Let $P$ be a Datalog$^A$ program and let $A = p(t_1, ..., t_m)$ be an atom appearing in $P$. Then $\hat{A}$ denotes the atom $p(t_1, ..., t_m)$ where $t_1, ..., t_m$ are all and the only arguments of $A$ corresponding to index attributes.

We say that $\hat{A}$ depends on $\hat{B}$ through the rule $r$ ($\hat{A} \leftarrow \hat{B}$) if there is a rule $r$ in $P$ of the form $A \leftarrow B_1, ..., B_n$. Moreover, $\hat{A}$ depends on $\hat{B}$ through the rules $r_1, ..., r_m$ ($\hat{A} \leftarrow (r_1, ..., r_m) \hat{B}$) if there is an atom $\hat{C}$ such that $\hat{A} \leftarrow (r_1, ..., r_m) \hat{C}$ and $\hat{C} \leftarrow (r_m, ..., r_1) \hat{B}$ (for some $1 \leq k \leq m$). Note that the definition of dependency here introduced uses atoms derived from the program whereas the classical definitions use predicate symbols (e.g. for the definition of stratified programs) or ground atoms (e.g. for the definition of locally stratified programs). We shall write $\hat{A} \leftarrow \hat{B}$ instead of $\hat{A} \leftarrow (r_1, ..., r_m) \hat{B}$ whenever the sequence of rules used in the dependency is clear from the context.

**Definition 10:** A Datalog$^A$ program $P$ is said to be monotonic if for each pair of atoms $A = p(X_1, ..., X_n)$ and $B = p(Y_1, ..., Y_n)$ if $\hat{A} \leftarrow \hat{B}$ then there exists an index argument at some position $i$ ($1 \leq i \leq n$) such that for all possible sequences of rules $r_1, ..., r_m$ with $\hat{A} \leftarrow (r_1, ..., r_m) \hat{B}$ and for all consistent instantiations of $r_1, ..., r_m$ either the condition $X_i > Y_i$ or the condition $X_i < Y_i$ is satisfied.

An intelligent compiler can obviously perform this analysis on the index arguments by assuming sufficient conditions which guarantee that the values of the index arguments belong to $\text{card}(P)$.

**Theorem 3:** When applied to monotonic programs, the fixpoint procedure terminates in a finite number of steps.

All the programs presented in this paper are monotonic. Consider, for instance, the definition of the predicate $\text{bin\_src}$ of Example 1. In the second rule we have that the value of the fourth argument in the head (variable $J$) is greater than the corresponding value in the body (variable $J_1$). In the third rule we have that the third argument in the head (variable $I$) is less than the third argument in the body predicate (variable $I_1$).

Observe that an each-rule $r$, when rewritten in standard form, defines also a monotonic component since the last but one argument of the predicate each$_s$ in the head is equal to the corresponding argument in the body predicate plus 1.

The class of monotonic Datalog$^A$ programs is rich enough to express most of the classical algorithms that use arrays. We present next two examples.

**Example 6:** Numbers of Fibonacci. The following program computes the number of Fibonacci using the dynamic programming based method.

\[
\]

Observe that the fixpoint procedure computes the answer in linear time.

**Example 7:** Scalar product of two vectors.

\[
\begin{align*}
\text{scalar}(A, B, K) & \leftarrow \text{size}(A, 1, N), \text{size}(B, 1, N), \\
C[I] & = A[I] \times B[I], \\
\text{scalar1}(A, B, C) & \leftarrow \text{size}(A, 1, N), \text{each}(I, 2, N), \\
\end{align*}
\]

The program above is evaluated linearly in the dimension of the input array.

**References**


