Journal of Intelligent Information Systems, 17, 71–94, 2001

The Branching-Time Transformation Technique for Chain Datalog Programs

PANOS RONDOGIANNIS prondo@di.uoa.gr
Dept. of Informatics and Telecommunications, University of Athens, 15784 Athens, Greece

MANOLIS GERGATSOULIS manolis@iit.demokritos.gr
Inst. of Informatics and Telecommunications, National Centre for Scientific Research (NCSR) 'Demokritos', 15310 A. Paraskevi Attikis, Athens, Greece

Received July 14, 2000; Revised July 26, 2001

Abstract. The *branching-time transformation* technique has proven to be an efficient approach for implementing functional programming languages. In this paper we demonstrate that such a technique can also be defined for logic programming languages. More specifically, we first introduce Branching Datalog, a language that can be considered as the basis for branching-temporal deductive databases. We then present a transformation algorithm from Chain Datalog programs to the class of unary Branching Datalog programs with at most one IDB atom in the body of each clause. In this way, we obtain a novel implementation approach for Chain Datalog, shedding at the same time new light on the power of branching-time logic programming.

Keywords: deductive databases, chain programs, program transformation, temporal logic programming, branching time

1. Introduction

The *branching-time transformation* is a promising technique that has been used for implementing functional programming languages (Yaghi, 1984; Wadge, 1991; Rondogiannis and Wadge, 1997, 1999). The basic idea behind the technique is that the recursive function calls that take place when a functional program is evaluated, actually form a tree-like structure. This observation has led to the idea of rewriting the source program into a form in which the tree structure of the recursion appears more explicitly. More specifically, the functional program is transformed into a zero-order *branching-time* functional program, which has a simpler structure and which can be easily evaluated using a demand-driven technique (also called *eduction* (Faustini and Wadge, 1987; Du and Wadge, 1990)). During eduction, in order to calculate the value of the source program, one simply *demands* the value at the root of the corresponding tree; the demand for this value generates demands that propagate to the interior nodes of the tree until leaf nodes are reached. The partial results returned from each node are used to compose the final result. The branching-time technique offers a promising alternative to the usual reduction-based implementations (Jones, 1987) of functional languages.

It is therefore natural to ask whether a similar transformation exists for logic programming languages. Our work aims at exactly this point: to examine whether logic programs can be

transformed into simpler in structure branching-time logic programs. More specifically, in this paper we first introduce Branching Datalog, a language that can be considered as the basis for branching-temporal deductive databases. We then define a transformation algorithm from the class of *Chain Datalog* programs (Ullman and Gelder, 1988; Afrati and Papadimitriou, 1987, 1993; Dong and Ginsburg, 1995) to the class of unary Branching Datalog programs with at most one IDB atom in the body of each clause. In this way, we obtain a novel implementation approach for Chain Datalog, shedding at the same time new light on the power of branching-time logic programming. It should be noted that the class of Chain Datalog is an especially important one because it embodies a form of recursion that is very common in many queries (e.g. queries related to graph applications, transitive closures of relations, etc.).

We should note at this point that the proposed technique belongs and contributes to the research area of query optimization for Datalog programs. More specifically, the branching-time transformation shares the same underlying philosophy with techniques which take into account the bound arguments of the query (such as the *magic transformation* (Beeri and Ramakrishnan, 1991), the *counting technique* (Saccà and Zaniolo, 1988), the *pushdown approach* (Greco et al., 1999), etc.). In all these techniques, the bound arguments of the goal clause are incorporated through an appropriate transformation into the source program (query); in this way, the bottom-up evaluation takes these arguments into account pruning the set of atoms that the immediate consequence operator has to produce.

The following example is given in order to motivate the proposed transformation. The precise presentation of all the concepts involved will be given in subsequent sections:

Example 1. The following is a Chain Datalog program together with a goal clause:

$\leftarrow$ p(a,Y).
p(X,Y) $\leftarrow$ e(X,Y).
p(X,Y) $\leftarrow$ q(X,Z), q(Z,Y).
q(X,Y) $\leftarrow$ e(X,Z), p(Z,Y).

where p and q are *IDB* predicates, while e is an *EDB* predicate. The output of the transformation is:

$\leftarrow$ first p_1(Y).
first p_0(a).
p_1(Y) $\leftarrow$ next$_1$ e_1(Y).
next$_1$ e_0(X) $\leftarrow$ p_0(X).
p_1(Y) $\leftarrow$ next$_3$ q_1(Y).
next$_3$ q_0(Z) $\leftarrow$ next$_2$ q_1(Z).
next$_2$ q_0(X) $\leftarrow$ p_0(X).
q_1(Y) $\leftarrow$ next$_5$ p_1(Y).
next$_5$ p_0(Z) $\leftarrow$ next$_4$ e_1(Z).
next$_4$ e_0(X) $\leftarrow$ q_0(X).
e_1(Y) $\leftarrow$ e(X,Y), e_0(X).

Notice that all intensional (IDB) predicates in the resulting program are unary and each clause has at most one IDB atom in its body. Notice also that the bound argument of the

goal clause has been incorporated into the program in the form of a unit clause; in this way this atom will guide from the very beginning of the bottom-up evaluation the production of atoms that are relevant to the query. The program also contains certain temporal operators ($\texttt{first}$, $\texttt{next}_1$, $\texttt{next}_2$, $\texttt{next}_3$, $\texttt{next}_4$, $\texttt{next}_5$) whose semantics will be introduced in a later section.

The main contributions of the paper can be summarized as follows:

- The language Branching Datalog is introduced and its semantics are defined. The new language can be viewed as a formalism that forms the basis for further investigations in the area of branching-temporal deductive databases. Temporal deductive databases constitute a well developed area of research (Baudinet et al., 1993; Orgun, 1996). Branching Datalog forms a particular instance of temporal deductive databases in which time has a branching (i.e. tree-like) structure. Another formalism that can be considered as a branching deductive database language, is Datalog_{nS} (Chomicki, 1995) (in which, however, the notion of time is not as explicit as in our case).
- A novel transformation algorithm from Chain Datalog programs into simpler in structure Branching Datalog programs is defined. The proposed transformation is the analogue of the branching-time transformation that has been defined in the functional programming domain (Rondogiannis and Wadge, 1997, 1999; Yaghi, 1984). This analogy suggests that other interesting optimization techniques from the deductive databases domain (such as (Beeri and Ramakrishnan, 1991; Saccà and Zaniolo, 1988; Greco et al., 1999)) may be applicable in the functional programming area.
- The results are interesting from a foundational point of view, as they shed new light on the power of temporal logic programming languages (branching-time ones in particular) and their relationship to classical logic programming.

The rest of the paper is organized as follows: Section 2 gives preliminary definitions that will be used throughout the paper. Section 3 defines the syntax and semantics of Branching Datalog. Section 4 introduces the branching-time transformation algorithm. Section 5 proves the correctness of the proposed transformation. Section 6 discusses termination issues regarding bottom-up evaluation of the programs obtained by the transformation. Section 7 proposes improvements of the transformation algorithm. Section 8 compares the branching transformation technique to other approaches for query optimization in the area of deductive databases. Finally, Section 9 concludes the paper with a brief discussion of possible future extensions.

2. Preliminaries

A *Datalog program* **P** consists of a finite set of function-free Horn rules. Predicates that appear in the head of some rule in **P** are called *IDB predicates* (IDBs), while predicates appearing only in the bodies of the rules of **P** are called *EDB predicates* (EDBs). A set **D** of ground facts (or unit clauses) defining the EDBs is often called a *database*. We also assume the following notation: *constants* are denoted by **a**, **b**, **c**, *variables* by **X**, **Y**, **Z** and

predicates by $\mathbf{p}$, $\mathbf{q}$, $\mathbf{r}$; also subscripted versions of the above symbols will be used. A *term* is either a variable or a constant. An *atom* is a formula of the form $\mathbf{p}(\mathbf{e}_0, \ldots, \mathbf{e}_{n-1})$ where $\mathbf{e}_0, \ldots, \mathbf{e}_{n-1}$ are terms. In the following, we assume familiarity with the basic notions of logic programming (Lloyd, 1987).

We are particularly interested in the class of *Chain Datalog* programs, whose syntax is defined below:

Definition 1 (Dong and Ginsburg, 1995). A chain rule is a clause of the form

$$\mathbf{q}(\mathbf{X}, \mathbf{Z}) \leftarrow \mathbf{q}_1(\mathbf{X}, \mathbf{Y}_1),\ \mathbf{q}_2(\mathbf{Y}_1, \mathbf{Y}_2), \ldots, \mathbf{q}_{k+1}(\mathbf{Y}_k, \mathbf{Z})$$

where $k \geq 0$, and $\mathbf{X}$, $\mathbf{Z}$ and each $\mathbf{Y}_i$ are distinct variables. Here $\mathbf{q}(\mathbf{X}, \mathbf{Z})$ is the head and $\mathbf{q}_1(\mathbf{X}, \mathbf{Y}_1), \mathbf{q}_2(\mathbf{Y}_1, \mathbf{Y}_2), \ldots, \mathbf{q}_{k+1}\ (\mathbf{Y}_k, \mathbf{Z})$ is the body of the rule. The body becomes $\mathbf{q}_1(\mathbf{X}, \mathbf{Z})$ when $k = 0$. A Chain Datalog program is a Datalog program whose rules are chain rules and whose EDB part consists of facts which are binary. A goal is of the form $\leftarrow \mathbf{q}(\mathbf{a}, \mathbf{X})$, where $\mathbf{a}$ is a constant, $\mathbf{X}$ is a variable and $\mathbf{q}$ is an IDB predicate.

Notice that each chain rule contains no constants and has at least one atom in its body. Notice also that the first argument of a goal is always ground. Assumptions of this form are very common in the area of query optimization in logic databases (Ullman, 1989). For example the magic set transformation is also based on the same assumption for the goal clause. More generally, this assumption is also met in most value propagating Datalog optimizations, and its necessity becomes clearer in later sections.

The first argument of a predicate will often be called its *input* argument, while the second one its *output* argument.

Definition 2. A simple Chain Datalog program is one in which every rule has at most two atoms in its body.

The semantics of (Chain) Datalog programs can be defined in accordance to the semantics of classical logic programming. The notions of *minimum model* $M_{\mathbf{P_D}}$ of $\mathbf{P} \cup \mathbf{D}$, where $\mathbf{P}$ is a Datalog program and $\mathbf{D}$ a database, and *immediate consequence operator* $T_{\mathbf{P_D}}$, transfer directly (Lloyd, 1987).

3. Branching Datalog

In this section we introduce the language *Branching Datalog* and define its denotational semantics. The new language can be viewed as a formalism for defining branching-temporal deductive databases. However, in this paper Branching Datalog will be used as the target language of the branching transformation (and not as a potentially useful new deductive database language).

Branching Datalog is actually a *temporal logic programming* language (Orgun and Ma, 1994; Gergatsoulis, 2001). Such languages are usually designed in such a way so as that they are capable of representing time-dependent information in a succint way. Many temporal

languages have been proposed differing among other things in the notion of time that they adopt (e.g. linear or branching, discrete or continuous, etc.). Branching temporal logic programming languages are those temporal languages in which a moment in time may have more than one immediately next moments. One of the first branching temporal languages is *Cactus* which we introduced in Rondogiannis et al. (1998). The branching (i.e. tree-like) structure of time underlying Cactus, makes it especially appropriate for describing tree algorithms and computations. Branching Datalog is the subset of Cactus in which programs do not contain any function symbols. The importance of Branching Datalog is that (as we demonstrate in the following sections) it can be used as the target language of the proposed optimization technique for deductive database queries. From a foundational point of view, the results of this paper show that Chain Datalog programs are equivalent to simpler in structure Branching Datalog programs.

The syntax of Branching Datalog is an extension of the syntax of Datalog. More specifically, the temporal operators `first` and $\texttt{next}_i$, $i \in \mathcal{N}$, are added to the syntax of Datalog. The declarative reading of these temporal operators will be discussed shortly.

A *temporal reference* is a sequence (possibly empty) of temporal operators. A *canonical temporal reference* is of the form $\texttt{first next}_{i_1} \ldots \texttt{next}_{i_n}$, where $i_1, \ldots, i_n \in \mathcal{N}$ and $n \geq 0$. An *open temporal reference* is of the form $\texttt{next}_{i_1} \ldots \texttt{next}_{i_n}$, where $i_1, \ldots, i_n \in \mathcal{N}$ and $n \geq 0$. A *temporal atom* is a classical atom preceded by either a canonical or an open temporal reference. A *temporal rule* is a formula of the form:

$$\mathbf{A} \leftarrow \mathbf{B}_1, \ldots, \mathbf{B}_m,$$

where $\mathbf{A}, \mathbf{B}_1, \ldots, \mathbf{B}_m$ are temporal atoms and $m > 0$. A *Branching Datalog program* $\mathbf{P}$ is a finite set of temporal rules. As usual, we consider all predicates defined in $\mathbf{P}$ as IDB predicates while those appearing in the bodies but not in the head of any rule in $\mathbf{P}$ as EDB predicates. A set of ground temporal atoms $\mathbf{D}$ defining the EDBs is called a *database*. The atoms in the database are considered to be independent of time.

A *goal* in Branching Datalog is a formula of the form $\leftarrow\mathbf{A}$ where $\mathbf{A}$ is a temporal atom. As it will become clear in subsequent sections, the target language of the transformation algorithm will be a subset of Branching Datalog and the goal clauses that will be used will consist of a single atom.

Branching Datalog is based on a relatively simple *branching-time logic* (BTL). In BTL, time has an initial moment and flows towards the future in a tree-like way. The set of moments in time can be modelled by the set $List(\mathcal{N})$ of lists of natural numbers $\mathcal{N}$. The empty list [ ] corresponds to the beginning of time and the list $[i \mid t]$ (that is, the list with head i, where $i \in \mathcal{N}$, and tail t) corresponds to the i-th child of the moment identified by the list t. BTL uses the temporal operators `first` and $\texttt{next}_i$, $i \in \mathcal{N}$. The operator `first` is used to express the first moment in time, while $\texttt{next}_i$ refers to the i-th child of the current moment in time. The syntax of BTL extends the syntax of first-order logic with two formation rules: if $\mathbf{A}$ is a formula then so are `first` $\mathbf{A}$ and $\texttt{next}_i$ $\mathbf{A}$.

The semantics of temporal formulas of *BTL* are given using the notion of *branching temporal interpretation* (Rondogiannis et al., 1998). Branching temporal interpretations extend the temporal interpretations of the linear time logic of Chronolog (Orgun, 1991).

Definition 3. A branching temporal interpretation or simply a temporal interpretation I of the temporal logic *BTL* comprises a non-empty set D, called the domain of the interpretation, together with an element of D for each variable; for each constant, an element of D; and for each n-ary predicate symbol, an element of $[List(\mathcal{N}) \rightarrow 2^{D^n}]$.

In the following definition, the satisfaction relation $\models$ is defined in terms of temporal interpretations. $\models_{I,t} \mathbf{A}$ denotes that a formula $\mathbf{A}$ is true at a moment t in the temporal interpretation I:

Definition 4. The semantics of the elements of the temporal logic BTL are given inductively as follows:

1. For any n-ary predicate symbol $\mathbf{p}$ and terms $\mathbf{e}_0, \ldots, \mathbf{e}_{n-1}$,
   $\models_{I,t} \mathbf{p}(\mathbf{e}_0, \ldots, \mathbf{e}_{n-1})$ iff $\langle I(\mathbf{e}_0), \ldots, I(\mathbf{e}_{n-1})\rangle \in I(\mathbf{p})(t)$
2. $\models_{I,t} \neg\mathbf{A}$ iff it is not the case that $\models_{I,t} \mathbf{A}$
3. $\models_{I,t} \mathbf{A} \wedge \mathbf{B}$ iff $\models_{I,t} \mathbf{A}$ and $\models_{I,t} \mathbf{B}$
4. $\models_{I,t} (\forall \mathbf{x})\, \mathbf{A}$ iff $\models_{I[d/\mathbf{x}],t} \mathbf{A}$ for all $a \in D$ where the interpretation $I[d/\mathbf{x}]$ is the same as I except that the variable $\mathbf{x}$ is assigned the element d.
5. $\models_{I,t}$ `first` $\mathbf{A}$ iff $\models_{I,[\,]} \mathbf{A}$
6. $\models_{I,t}$ $\mathtt{next}_i$ $\mathbf{A}$ iff $\models_{I,[i\,|\,t]} \mathbf{A}$

The Boolean connectives $\vee$, $\rightarrow$ and $\leftrightarrow$, and the existential quantifier $\exists$ are defined in the usual way.

If a formula $\mathbf{A}$ is true in a temporal interpretation I at all moments in time, it is said to be true in I (we write $\models_I \mathbf{A}$) and I is called a *model* of $\mathbf{A}$. If for all interpretations I, $\models_I \mathbf{A}$, we say that $\mathbf{A}$ is *valid* and write $\models \mathbf{A}$.

It should be noted here that the syntax of BTL allows atoms with temporal references that are more complicated from the ones we adopt for Branching Datalog. Moreover it allows temporal references to be applied to whole formulas (and not just atoms). However, it is easy to define axioms and rules of inference for BTL and use them to demonstrate that every formula of BTL can be transformed into an equivalent formula in which all temporal references are either canonical or open and are only applied to atoms. We do not pursue these issues any further here (however, the interested reader can consult (Rondogiannis et al., 1998)).

3.1. Semantics of Branching Datalog

The semantics of Branching Datalog are defined in terms of *temporal Herbrand interpretations*. A notion that is crucial in the discussion that follows, is that of *canonical instance of a clause*, which corresponds to a temporally ground instance of the clause. This notion is formalized below.

Definition 5. A canonical temporal atom is a temporal atom whose temporal reference is canonical. An open temporal atom is a temporal atom whose temporal reference is open.

A canonical temporal clause is a temporal clause whose temporal atoms are canonical. A canonical temporal instance of a temporal clause $\mathbf{C}$ is a canonical temporal clause $\mathbf{C}'$ which can be obtained by applying the same canonical temporal reference to all open atoms of $\mathbf{C}$.

Let $\mathbf{P}$ be a Branching Datalog program and $\mathbf{D}$ be a database. As in Datalog, the finite set $U_{\mathbf{P_D}}$ containing all constant symbols that appear in $\mathbf{P} \cup \mathbf{D}$, called *Herbrand universe*, is used to define *temporal Herbrand interpretations*. Temporal Herbrand interpretations can be regarded as subsets of the *temporal Herbrand Base* $TB_{\mathbf{P_D}}$ of $\mathbf{P} \cup \mathbf{D}$, consisting of all *ground canonical temporal atoms* whose predicate symbols appear in $\mathbf{P} \cup \mathbf{D}$ and whose arguments are terms in the Herbrand universe $U_{\mathbf{P_D}}$ of $\mathbf{P} \cup \mathbf{D}$. A *temporal Herbrand model* is a temporal Herbrand interpretation which is a model of $\mathbf{P} \cup \mathbf{D}$.

The theorems of this section and their proofs are analogous to those of classical logic programming (Lloyd, 1987), or linear-time logic programming (Orgun, 1991). For example, it can be easily shown that the *model intersection property* holds for temporal Herbrand models. Moreover, the intersection of all temporal Herbrand models, denoted by $M(\mathbf{P_D})$, is a temporal Herbrand model, called the *least temporal Herbrand model*.

The following theorem says that the least temporal Herbrand model consists of all ground canonical temporal atoms which are logical consequences of $\mathbf{P} \cup \mathbf{D}$. Again, the proof of the theorem is an easy extension of the corresponding proof for classical logic programming.

Theorem 1. *Let* $\mathbf{P}$ *be a Branching Datalog program and* $\mathbf{D}$ *be a database. Then*

$$M(\mathbf{P_D}) = \{\mathbf{A} \in TB_{\mathbf{P_D}} \mid (\mathbf{P} \cup \mathbf{D}) \models \mathbf{A}\}.$$

A fixpoint characterization of the semantics of Branching Datalog programs is provided using a closure operator that maps temporal Herbrand interpretations to temporal Herbrand interpretations:

Definition 6. Let $\mathbf{P}$ be a Branching Datalog program and $\mathbf{D}$ be a database. The operator $T_{\mathbf{P_D}}$: $2^{TB_{\mathbf{P_D}}} \rightarrow 2^{TB_{\mathbf{P_D}}}$ is defined as follows: if I is a temporal Herbrand interpretation in $2^{TB_{\mathbf{P_D}}}$ then $T_{\mathbf{P_D}}(I) = \{\mathbf{A} \mid \mathbf{A} \leftarrow \mathbf{B}_1, \ldots, \mathbf{B}_n$ is a canonical ground instance of a program clause in $\mathbf{P} \cup \mathbf{D}$ and $\{\mathbf{B}_1, \ldots, \mathbf{B}_n\} \subseteq I\}$.

It can be easily proved that $2^{TB_{\mathbf{P_D}}}$ is a complete lattice under the partial order of set inclusion ($\subseteq$). Moreover, $T_{\mathbf{P_D}}$ is continuous and hence monotonic over the complete lattice $(2^{TB_{\mathbf{P_D}}}, \subseteq)$ and therefore $T_{\mathbf{P_D}}$ has a least fixpoint. The least fixpoint of $T_{\mathbf{P_D}}$ provides a characterization of the minimal Herbrand model of a Branching Datalog program, as it is stated by the following theorem.

Theorem 2. *Let* $\mathbf{P}$ *be a Branching Datalog program and* $\mathbf{D}$ *be a database. Then*

$$M(\mathbf{P_D}) = lfp(T_{\mathbf{P_D}}) = T_{\mathbf{P_D}} \uparrow \omega.$$

Notice that although in classical Datalog the least fixpoint of a program is reached in a finite number of iterations, this is not the case for Branching Datalog due to the existence of temporal operators. This point will be further discussed in Section 6.

4. The transformation algorithm

The branching-time transformation algorithm takes as input a simple Chain Datalog program together with a goal clause, and produces as output a Branching Datalog program and a new goal clause. Certain remarks are in order:

- The fact that the proposed algorithm is defined for simple Chain Datalog programs is not a real restriction because, as it is illustrated by Proposition 1 that follows, every Chain Datalog program can be transformed into an equivalent simple one.
- The input to the algorithm is a program *together* with a goal clause. This is similar to the spirit of the corresponding transformation in functional programming (Yaghi, 1984; Rondogiannis and Wadge, 1997) in which a functional program contains a top-level definition of a special variable `result` whose value is the output of the program. Moreover, this is also similar to the spirit of many well known optimization techniques for Datalog programs (counting, magic sets, pushdown method, etc.).

It should also be noted that the output of the transformation is a Branching Datalog program in which:

1. All IDB predicates are unary.
2. There is at most one IDB atom in the body of each clause in the program.[1]

The following proposition establishes the equivalence between Chain Datalog and simple Chain Datalog programs. Notice that $M(\mathbf{P_D}, \mathbf{p})$ denotes the set of atoms in $M(\mathbf{P_D})$ whose predicate symbol is $\mathbf{p}$.

Proposition 1. *Every Chain Datalog program* $\mathbf{P}$ *can be transformed into a simple Chain Datalog program* $\mathbf{P}^s$ *such that for every database* $\mathbf{D}$ *and for every predicate symbol* $\mathbf{p}$ *of* $\mathbf{P}$, *it holds* $M(\mathbf{P_D}, \mathbf{p}) = M(\mathbf{P}^s_\mathbf{D}, \mathbf{p})$.

Proof: Let k be the maximum number of atoms in a clause body in $\mathbf{P}$. We will prove by induction on k that $\mathbf{P}$ can be transformed into a simple Chain Datalog program $\mathbf{P}^s$ such that $M(\mathbf{P_D}, \mathbf{p}) = M(\mathbf{P}^s_\mathbf{D}, \mathbf{p})$.

For $k \leq 2$ the result holds trivially. Assume that the result holds for some $k \geq 2$. We will prove that it also holds for $k + 1$.

Consider a *Chain rule* in $\mathbf{P}$ of the form:

$$\mathbf{p}(\mathbf{X}, \mathbf{Z}) \leftarrow \mathbf{q}_1(\mathbf{X}, \mathbf{Y}_1), \mathbf{q}_2(\mathbf{Y}_1, \mathbf{Y}_2), \ldots, \mathbf{q}_{k+1}(\mathbf{Y}_k, \mathbf{Z}). \quad (1)$$

This rule can be replaced by the two following ones (in which $\mathbf{r}$ is a new predicate name that we introduce):

$$\mathbf{p}(\mathbf{X}, \mathbf{Z}) \leftarrow \mathbf{q}_1(\mathbf{X}, \mathbf{Y}_1), \mathbf{r}(\mathbf{Y}_1, \mathbf{Z}). \quad (2)$$

$$\mathbf{r}(\mathbf{Y}_1, \mathbf{Z}) \leftarrow \mathbf{q}_2(\mathbf{Y}_1, \mathbf{Y}_2), \ldots, \mathbf{q}_{k+1}(\mathbf{Y}_k, \mathbf{Z}). \quad (3)$$

Now, clause (2) has two atoms in its body, while clause (3) has k (one less than clause (1) initially had). This procedure is applied to all clauses in $\mathbf{P}$ whose bodies contain $k+1$ atoms. Let $\mathbf{P}'$ be the resulting program. It is easy to see that $M(\mathbf{P_D}, \mathbf{p}) = M(\mathbf{P'_D}, \mathbf{p})$ for every predicate symbol $\mathbf{p}$ in $\mathbf{P}$. This is true because the new clauses (of the form 3) that we introduce can be considered as *Eureka* definitions (Proietti and Pettorossi, 1990), while the clauses of the form 2 are obtained by folding (Tamaki and Sato, 1984; Gergatsoulis and Katzouraki, 1994) clauses of the form 1 using clauses of the form 3.

Now in $\mathbf{P}'$ all clauses have at most k atoms. Therefore, we can apply the induction hypothesis getting the desired result. □

Notice that the proof of the above proposition is a constructive one, and therefore it suggests a method for obtaining a simple Chain Datalog program from a Chain Datalog one.

We can now formally define the transformation algorithm which takes a simple Chain Datalog program together with a goal clause as input and returns as output a Branching Datalog program (of the form discussed above) together with a corresponding goal clause.

The algorithm: Let $\mathbf{P}$ be a given simple Chain Datalog program and $\mathbf{G}$ a given goal clause. For each IDB or EDB predicate $\mathbf{p}$ in $\mathbf{P}$, two unary IDB predicates $\mathbf{p}_0$ and $\mathbf{p}_1$ are introduced. The transformation processes each clause in $\mathbf{P}$ and the goal clause $\mathbf{G}$ and gives as output a Branching Datalog program $\mathbf{P}^*$ together with a new goal clause $\mathbf{G}^*$. When processing a rule of the source program, the algorithm introduces branching-time operators of the form $\texttt{next}_i$, $i \in \mathcal{N}$. It is important to notice that:

The operators introduced for a given rule are assumed to have different indices than the operators used for any other rule.

1. For every EDB predicate $\mathbf{p}$ of $\mathbf{P}$, add a new clause to $\mathbf{P}^*$ of the form:

$$\mathbf{p}_1(\mathbf{Y}) \leftarrow \mathbf{p}(\mathbf{X}, \mathbf{Y}), \mathbf{p}_0(\mathbf{X}).$$

2. Each clause in $\mathbf{P}$ of the form:

$$\mathbf{p}(\mathbf{X}, \mathbf{Y}) \leftarrow \mathbf{q}(\mathbf{X}, \mathbf{Y}).$$

is transformed into two clauses in $\mathbf{P}^*$ of the form:

$$\mathbf{p}_1(\mathbf{Y}) \leftarrow \texttt{next}_i\ \mathbf{q}_1(\mathbf{Y}).$$
$$\texttt{next}_i\ \mathbf{q}_0(\mathbf{X}) \leftarrow \mathbf{p}_0(\mathbf{X}).$$

3. Each clause in $\mathbf{P}$ of the form:

$$\mathbf{p}(\mathbf{X}, \mathbf{Y}) \leftarrow \mathbf{q}(\mathbf{X}, \mathbf{Z}), \mathbf{r}(\mathbf{Z}, \mathbf{Y}).$$

is tansformed into the set of clauses in $\mathbf{P}^*$:

$$\mathbf{p}_1(\mathbf{Y}) \leftarrow \texttt{next}_i\ \mathbf{r}_1(\mathbf{Y}).$$
$$\texttt{next}_i\ \mathbf{r}_0(\mathbf{Z}) \leftarrow \texttt{next}_j\ \mathbf{q}_1(\mathbf{Z}).$$
$$\texttt{next}_j\ \mathbf{q}_0(\mathbf{X}) \leftarrow \mathbf{p}_0(\mathbf{X}).$$

where $i \neq j$.

4. The goal clause:

$$\leftarrow \mathbf{p}(\mathbf{a},\mathbf{Y}).$$

is transformed into a new goal clause $\mathbf{G}^*$:

$$\leftarrow \texttt{first}\ \mathbf{p}_1(\mathbf{Y}).$$

and the following unit clause which is added to $\mathbf{P}^*$:

$$\texttt{first}\ \mathbf{p}_0(\mathbf{a}).$$

Notice that the algorithm introduces for each predicate **p** in **P** two unary IDB predicates $\mathbf{p}_0$ and $\mathbf{p}_1$. Intuitively, $\mathbf{p}_0$ corresponds to the first argument of **p** while $\mathbf{p}_1$ to the second. Moreover, the clause introduced in Step 1, relates the arguments of the EDB predicate **p** to the arguments of the corresponding predicates $\mathbf{p}_0$ and $\mathbf{p}_1$ in $\mathbf{P}^*$. In fact, this clause plays the role of an interface between the program $\mathbf{P}^*$ and the database **D**. The clauses produced in Steps 2 and 3 actually reflect the flow of the argument values during the execution of the source Chain Datalog program. Finally, in Step 4 the input argument of the goal clause is actually transferred into the database through the introduction of a new fact (so as that it will be taken into account from the beginning of a bottom-up evaluation of the target program).

Example 2. Let $\mathbf{P} = \{\mathbf{I}_1, \mathbf{I}_2\}$ be a Chain Datalog program and **G** be a goal clause, where:

$$(\mathbf{G}) \quad \leftarrow \texttt{p(a,Y)}.$$
$$(\mathbf{I}_1) \quad \texttt{p(X,Z)} \leftarrow \texttt{e(X,Z)}.$$
$$(\mathbf{I}_2) \quad \texttt{p(X,Z)} \leftarrow \texttt{p(X,Y),e(Y,Z)}.$$

and where **e** is an EDB predicate. Transforming the goal clause **G** we get:

$$\leftarrow \texttt{first}\ \texttt{p}_1\texttt{(Y)}.$$
$$\texttt{first}\ \texttt{p}_0\texttt{(a)}.$$

Transforming $\mathbf{I}_1$ we get:

$$\texttt{p}_1\texttt{(Z)} \leftarrow \texttt{next}_1\ \texttt{e}_1\texttt{(Z)}.$$
$$\texttt{next}_1\ \texttt{e}_0\texttt{(X)} \leftarrow \texttt{p}_0\texttt{(X)}.$$

Transforming $\mathbf{I}_2$ we get:

$$\mathtt{p}_1(\mathtt{Z}) \leftarrow \mathtt{next}_3\ \mathtt{e}_1(\mathtt{Z}).$$
$$\mathtt{next}_3\ \mathtt{e}_0(\mathtt{Y}) \leftarrow \mathtt{next}_2\ \mathtt{p}_1(\mathtt{Y}).$$
$$\mathtt{next}_2\ \mathtt{p}_0(\mathtt{X}) \leftarrow \mathtt{p}_0(\mathtt{X}).$$

Finally, for the EDB predicate e we introduce the following clause:

$$\mathtt{e}_1(\mathtt{Y}) \leftarrow \mathtt{e}(\mathtt{X},\mathtt{Y}),\mathtt{e}_0(\mathtt{X}).$$

Certain remarks concerning the intuition behind the use of temporal operators in the transformation algorithm, are in order. For each predicate in the initial program the transformation algorithm separates its input from its output argument by producing two distinct unary predicates. The coordination of these two predicates so as to produce the correct answers is ensured through the use of canonical sequences of temporal operators (as it will become obvious from the lemmas that constitute the correctness proof of the transformation).

In the following section we demonstrate the correctness of the proposed transformation algorithm.

5. Correctness proof

Let $\mathbf{P}$ be a simple Chain Datalog program, $\mathbf{D}$ a database and $\leftarrow \mathbf{p}(\mathbf{a}, \mathbf{X})$ a goal clause. The correctness proof of the transformation proceeds as follows: at first we show (see Lemma 2 below) that if a ground instance $\mathbf{p}(\mathbf{a}, \mathbf{b})$ of the goal clause is a logical consequence of $\mathbf{P} \cup \mathbf{D}$ then the atom $\mathtt{first}\ \mathbf{p}_1(\mathbf{b})$ is a logical consequence of $\mathbf{P}^* \cup \mathbf{D}$, where $\mathbf{P}^* \cup \{\leftarrow \mathtt{first}\ \mathbf{p}_1(\mathbf{X})\}$ is obtained by applying the transformation algorithm to $\mathbf{P} \cup \{\leftarrow \mathbf{p}(\mathbf{a}, \mathbf{X})\}$. In order to prove this result we establish a more general lemma[2] (Lemma 1 below). The inverse of Lemma 2 is given as Lemma 4. More specifically, we prove that whenever $\mathtt{first}\ \mathbf{p}_1(\mathbf{b})$ is a logical consequence of $\mathbf{P}^* \cup \mathbf{D}$ then $\mathbf{p}(\mathbf{a}, \mathbf{b})$ is a logical consequence of $\mathbf{P} \cup \mathbf{D}$. Again, we establish this result by proving the more general Lemma 3. Combining the above results we get the correctness proof of the transformation algorithm.

Lemma 1. *Let* $\mathbf{P}$ *be a simple Chain Datalog program,* $\mathbf{D}$ *a database and* $\mathbf{G}$ *a goal clause. Let* $\mathbf{P}^*$ *be the Branching Datalog program obtained by applying the transformation algorithm to* $\mathbf{P} \cup \mathbf{G}$*. For all predicates* $\mathbf{p}$ *defined in* $\mathbf{P} \cup \mathbf{D}$*, all canonical temporal references* R*, and all* $\mathbf{a}, \mathbf{b} \in U_{\mathbf{P_D}}$*, if* $R\ \mathbf{p}_0(\mathbf{a}) \in T_{\mathbf{P}^*_\mathbf{D}} \uparrow \omega$ *and* $\mathbf{p}(\mathbf{a}, \mathbf{b}) \in T_{\mathbf{P_D}} \uparrow \omega$ *then* $R\ \mathbf{p}_1(\mathbf{b}) \in T_{\mathbf{P}^*_\mathbf{D}} \uparrow \omega$*.*

Proof: We show the above by induction on the approximations of $T_{\mathbf{P_D}} \uparrow \omega$.

Induction Basis: To establish the induction basis, we need to show that if $R\ \mathbf{p}_0(\mathbf{a}) \in T_{\mathbf{P}^*_\mathbf{D}} \uparrow \omega$ and $\mathbf{p}(\mathbf{a}, \mathbf{b}) \in T_{\mathbf{P_D}} \uparrow 0$ then $R\ \mathbf{p}_1(\mathbf{b}) \in T_{\mathbf{P}^*_\mathbf{D}} \uparrow \omega$. The induction basis trivially holds because $T_{\mathbf{P_D}} \uparrow 0 = \emptyset$ and thus $\mathbf{p}(\mathbf{a}, \mathbf{b}) \in T_{\mathbf{P_D}} \uparrow 0$ is false.

Induction Hypothesis: We assume that if $R\ \mathbf{p}_0(\mathbf{a}) \in T_{\mathbf{P}^*_\mathbf{D}} \uparrow \omega$ and $\mathbf{p}(\mathbf{a}, \mathbf{b}) \in T_{\mathbf{P_D}} \uparrow k$ then $R\ \mathbf{p}_1(\mathbf{b}) \in T_{\mathbf{P}^*_\mathbf{D}} \uparrow \omega$. Notice that the induction hypothesis holds for any $\mathbf{p}$ in $\mathbf{P}$ and any temporal reference R.

Induction Step: We show that if $R\ \mathbf{p}_0(\mathbf{a}) \in T_{\mathbf{P}^*_\mathbf{D}} \uparrow \omega$ and $\mathbf{p}(\mathbf{a}, \mathbf{b}) \in T_{\mathbf{P}_\mathbf{D}} \uparrow (k+1)$ then $R\ \mathbf{p}_1(\mathbf{b}) \in T_{\mathbf{P}^*_\mathbf{D}} \uparrow \omega$.

Case 1. Assume that $\mathbf{p}(\mathbf{a}, \mathbf{b})$ has been added to $T_{\mathbf{P}_\mathbf{D}} \uparrow (k+1)$ because it is a fact in $\mathbf{D}$. According to the transformation algorithm, in $\mathbf{P}^*$ there exists the rule $\mathbf{p}_1(\mathbf{Y}) \leftarrow \mathbf{p}(\mathbf{X}, \mathbf{Y}),\ \mathbf{p}_0(\mathbf{X})$. Using this and the fact that $R\ \mathbf{p}_0(\mathbf{a}) \in T_{\mathbf{P}^*_\mathbf{D}} \uparrow \omega$ we conclude that $R\ \mathbf{p}_1(\mathbf{b}) \in T_{\mathbf{P}^*_\mathbf{D}} \uparrow \omega$.

Case 2. Assume that $\mathbf{p}(\mathbf{a}, \mathbf{b})$ has been added to $T_{\mathbf{P}_\mathbf{D}} \uparrow (k+1)$ using a rule of the form:

$$\mathbf{p}(\mathbf{X}, \mathbf{Y}) \leftarrow \mathbf{q}(\mathbf{X}, \mathbf{Z}), \mathbf{r}(\mathbf{Z}, \mathbf{Y}). \tag{1}$$

Then, there exists a constant $\mathbf{c}$ such that $\mathbf{q}(\mathbf{a}, \mathbf{c}) \in T_{\mathbf{P}_\mathbf{D}} \uparrow k$ and $\mathbf{r}(\mathbf{c}, \mathbf{b}) \in T_{\mathbf{P}_\mathbf{D}} \uparrow k$.

Consider now the transformation of the above clause (1) in program $\mathbf{P}^*$. The new clauses obtained are:

$$\mathbf{p}_1(\mathbf{Y}) \leftarrow \mathtt{next}_i\ \mathbf{r}_1(\mathbf{Y}). \tag{2}$$
$$\mathtt{next}_i\ \mathbf{r}_0(\mathbf{Z}) \leftarrow \mathtt{next}_j\ \mathbf{q}_1(\mathbf{Z}). \tag{3}$$
$$\mathtt{next}_j\ \mathbf{q}_0(\mathbf{X}) \leftarrow \mathbf{p}_0(\mathbf{X}). \tag{4}$$

Using the assumption that $R\ \mathbf{p}_0(\mathbf{a}) \in T_{\mathbf{P}^*_\mathbf{D}} \uparrow \omega$ together with clause (4) above, we get that $R\ \mathtt{next}_j\ \mathbf{q}_0(\mathbf{a}) \in T_{\mathbf{P}^*_\mathbf{D}} \uparrow \omega$. Given this, we can now apply the induction hypothesis on $\mathbf{q}$ and on temporal reference $R\ \mathtt{next}_j$, which gives:

Since $R\ \mathtt{next}_j\ \mathbf{q}_0(\mathbf{a}) \in T_{\mathbf{P}^*_\mathbf{D}} \uparrow \omega$ and $\mathbf{q}(\mathbf{a}, \mathbf{c}) \in T_{\mathbf{P}_\mathbf{D}} \uparrow k$ then $R\ \mathtt{next}_j\ \mathbf{q}_1(\mathbf{c}) \in T_{\mathbf{P}^*_\mathbf{D}} \uparrow \omega$.

Using now the fact that $R\ \mathtt{next}_j\ \mathbf{q}_1(\mathbf{c}) \in T_{\mathbf{P}^*_\mathbf{D}} \uparrow \omega$ together with clause (3) we get $R\ \mathtt{next}_i\ \mathbf{r}_0(\mathbf{c}) \in T_{\mathbf{P}^*_\mathbf{D}} \uparrow \omega$. Given this, we can now apply the induction hypothesis on $\mathbf{r}$ which gives:

Since $R\ \mathtt{next}_i\ \mathbf{r}_0(\mathbf{c}) \in T_{\mathbf{P}^*_\mathbf{D}} \uparrow \omega$ and $\mathbf{r}(\mathbf{c}, \mathbf{b}) \in T_{\mathbf{P}_\mathbf{D}} \uparrow k$ then $R\ \mathtt{next}_i\ \mathbf{r}_1(\mathbf{b}) \in T_{\mathbf{P}^*_\mathbf{D}} \uparrow \omega$.

Finally, using the fact that $R\ \mathtt{next}_i\ \mathbf{r}_1(\mathbf{b}) \in T_{\mathbf{P}^*_\mathbf{D}} \uparrow \omega$ together with clause (2), we get the desired result which is that $R\ \mathbf{p}_1(\mathbf{b}) \in T_{\mathbf{P}^*_\mathbf{D}} \uparrow \omega$.

Case 3. Assume that $\mathbf{p}(\mathbf{a}, \mathbf{b})$ has been added to $T_{\mathbf{P}_\mathbf{D}} \uparrow (k+1)$ using a rule of the form:

$$\mathbf{p}(\mathbf{X}, \mathbf{Y}) \leftarrow \mathbf{q}(\mathbf{X}, \mathbf{Y}). \tag{5}$$

The proof of this Case is similar (and actually simpler) to that for Case 2. □

Lemma 2. *Let* $\mathbf{P}$ *be a simple Chain Datalog program,* $\mathbf{D}$ *be a database and* $\leftarrow\mathbf{p}(\mathbf{a}, \mathbf{X})$ *be a goal clause. Let* $\mathbf{P}^* \cup \{\leftarrow \mathtt{first}\ \mathbf{p}_1(\mathbf{X})\}$ *be the output obtained by applying the transformation algorithm to* $\mathbf{P} \cup \{\leftarrow\mathbf{p}(\mathbf{a}, \mathbf{X})\}$. *If* $\mathbf{p}(\mathbf{a}, \mathbf{b}) \in T_{\mathbf{P}_\mathbf{D}} \uparrow \omega$ *then* $\mathtt{first}\ \mathbf{p}_1(\mathbf{b}) \in T_{\mathbf{P}^*_\mathbf{D}} \uparrow \omega$.

Proof: Since by transforming the goal clause, the fact $\mathtt{first}\ \mathrm{p}_0(\mathbf{a})$ is added to $\mathbf{P}^*_\mathbf{D}$, this lemma is a special case of Lemma 1. □

We now show the following lemma which is the "inverse" of Lemma 1:

Lemma 3. *Let* $\mathbf{P}$ *be a simple Chain Datalog program,* $\mathbf{D}$ *a database and* $\mathbf{G}$ *a goal clause. Let* $\mathbf{P}^*$ *be the Branching Datalog program obtained by applying the transformation algorithm to* $\mathbf{P} \cup \mathbf{G}$. *For all predicates* $\mathbf{p}$ *in* $\mathbf{P} \cup \mathbf{D}$, *for all canonical temporal references* R, *and for all* $\mathbf{b} \in U_{\mathbf{P_D}}$, *if* $R\ \mathbf{p}_1(\mathbf{b}) \in T_{\mathbf{P}^*_\mathbf{D}} \uparrow \omega$ *then there exists a constant* $\mathbf{a} \in U_{\mathbf{P_D}}$ *such that* $\mathbf{p}(\mathbf{a}, \mathbf{b}) \in T_{\mathbf{P_D}} \uparrow \omega$ *and* $R\ \mathbf{p}_0(\mathbf{a}) \in T_{\mathbf{P}^*_\mathbf{D}} \uparrow \omega$.

Proof: We show the above by induction on the approximations of $T_{\mathbf{P}^*_\mathbf{D}} \uparrow \omega$.

Induction Basis: It is convenient to show the induction basis for both $T_{\mathbf{P}^*_\mathbf{D}} \uparrow 0$ and $T_{\mathbf{P}^*_\mathbf{D}} \uparrow 1$. We therefore will show that: (a) if $R\ \mathbf{p}_1(\mathbf{b}) \in T_{\mathbf{P}^*_\mathbf{D}} \uparrow 0$ then there exists a constant $\mathbf{a}$ such that $\mathbf{p}(\mathbf{a}, \mathbf{b}) \in T_{\mathbf{P_D}} \uparrow \omega$ and $R\ \mathbf{p}_0(\mathbf{a}) \in T_{\mathbf{P}^*_\mathbf{D}} \uparrow 0$, and (b) if $R\ \mathbf{p}_1(\mathbf{b}) \in T_{\mathbf{P}^*_\mathbf{D}} \uparrow 1$ then there exists a constant $\mathbf{a}$ such that $\mathbf{p}(\mathbf{a}, \mathbf{b}) \in T_{\mathbf{P_D}} \uparrow \omega$ and $R\ \mathbf{p}_0(\mathbf{a}) \in T_{\mathbf{P}^*_\mathbf{D}} \uparrow 1$.

The statement (a) vacuously holds because $T_{\mathbf{P}^*_\mathbf{D}} \uparrow 0 = \emptyset$ and thus $R\ \mathbf{p}_1(\mathbf{b}) \in T_{\mathbf{P}^*_\mathbf{D}} \uparrow 0$ is false. For (b), $R\ \mathbf{p}_1(\mathbf{b}) \in T_{\mathbf{P}^*_\mathbf{D}} \uparrow 1$ is also false because (as it can be easily seen from the definition of the transformation algorithm) in $T_{\mathbf{P}^*_\mathbf{D}} \uparrow 1$, besides the atoms in $\mathbf{D}$, there only belongs one temporal atom whose predicate is an input one. This atom has been obtained by transforming the goal clause. Therefore, the basis case holds vacuously.

Induction Hypothesis: If $R\ \mathbf{p}_1(\mathbf{b}) \in T_{\mathbf{P}^*_\mathbf{D}} \uparrow k$ then there exists a constant $\mathbf{a}$ such that $\mathbf{p}(\mathbf{a}, \mathbf{b}) \in T_{\mathbf{P_D}} \uparrow \omega$ and $R\ \mathbf{p}_0(\mathbf{a}) \in T_{\mathbf{P}^*_\mathbf{D}} \uparrow k$.

Induction Step: We show that if $R\ \mathbf{p}_1(\mathbf{b}) \in T_{\mathbf{P}^*_\mathbf{D}} \uparrow (k+1)$ then there exists a constant $\mathbf{a}$ such that $\mathbf{p}(\mathbf{a}, \mathbf{b}) \in T_{\mathbf{P_D}} \uparrow \omega$ and $R\ \mathbf{p}_0(\mathbf{a}) \in T_{\mathbf{P}^*_\mathbf{D}} \uparrow (k+1)$.

Case 1. Assume now that there exists in $\mathbf{P}$ a rule of the form:

$$\mathbf{p}(\mathbf{X}, \mathbf{Y}) \leftarrow \mathbf{q}(\mathbf{X}, \mathbf{Z}), \mathbf{r}(\mathbf{Z}, \mathbf{Y}). \tag{1}$$

which has been transformed into the clauses:

$$\mathbf{p}_1(\mathbf{Y}) \leftarrow \mathtt{next}_i\ \mathbf{r}_1(\mathbf{Y}). \tag{2}$$
$$\mathtt{next}_i\ \mathbf{r}_0(\mathbf{Z}) \leftarrow \mathtt{next}_j\ \mathbf{q}_1(\mathbf{Z}). \tag{3}$$
$$\mathtt{next}_j\ \mathbf{q}_0(\mathbf{X}) \leftarrow \mathbf{p}_0(\mathbf{X}). \tag{4}$$

in $\mathbf{P}^*$. Assume that $R\ \mathbf{p}_1(\mathbf{b})$ has been introduced in $T_{\mathbf{P}^*_\mathbf{D}} \uparrow (k+1)$ by clause (2) above. Thus $R\ \mathtt{next}_i\ \mathbf{r}_1(\mathbf{b}) \in T_{\mathbf{P}^*_\mathbf{D}} \uparrow k$. By the induction hypothesis, we get that there exists a constant $\mathbf{c}$ such that $\mathbf{r}(\mathbf{c}, \mathbf{b}) \in T_{\mathbf{P_D}} \uparrow \omega$ and $R\ \mathtt{next}_i\ \mathbf{r}_0(\mathbf{c}) \in T_{\mathbf{P}^*_\mathbf{D}} \uparrow k$.

Notice now that the only way that $R\ \mathtt{next}_i\ \mathbf{r}_0(\mathbf{c}) \in T_{\mathbf{P}^*_\mathbf{D}} \uparrow k$ can have been obtained is by using clause (3) above (all other clauses defining $\mathbf{r}_0$, have a different index in the `next` operator). Therefore, using clause (3) above, we get that[3] $R\ \mathtt{next}_j\ \mathbf{q}_1(c) \in T_{\mathbf{P}^*_\mathbf{D}} \uparrow (k-1)$ which also means that $R\ \mathtt{next}_j\ \mathbf{q}_1(\mathbf{c}) \in T_{\mathbf{P}^*_\mathbf{D}} \uparrow k$. Using the induction hypothesis, we get that there exists a constant $\mathbf{a}$ such that $\mathbf{q}(\mathbf{a}, \mathbf{c}) \in T_{\mathbf{P_D}} \uparrow \omega$ and $R\ \mathtt{next}_j\ \mathbf{q}_0(\mathbf{a}) \in T_{\mathbf{P}^*_\mathbf{D}} \uparrow k$. But then, using clause (4) above as before we get $R\ \mathbf{p}_0(\mathbf{a}) \in T_{\mathbf{P}^*_\mathbf{D}} \uparrow (k-1)$, which implies that $R\ \mathbf{p}_0(\mathbf{a}) \in T_{\mathbf{P}^*_\mathbf{D}} \uparrow k$. Moreover, since $\mathbf{q}(\mathbf{a}, \mathbf{c}) \in T_{\mathbf{P_D}} \uparrow \omega$ and $\mathbf{r}(\mathbf{c}, \mathbf{b}) \in T_{\mathbf{P_D}} \uparrow \omega$ from (1) we also get $\mathbf{p}(\mathbf{a}, \mathbf{b}) \in T_{\mathbf{P_D}} \uparrow \omega$. Using these, we derive the desired result.

Case 2. Assume that in $\mathbf{P}$ there exists a rule of the form:

$$\mathbf{p}(\mathbf{X}, \mathbf{Y}) \leftarrow \mathbf{q}(\mathbf{X}, \mathbf{Y}). \tag{5}$$

The proof of this Case is similar (and actually simpler) to that for Case 1.

Case 3. Assume that in $\mathbf{P}$ there exists an EDB predicate $\mathbf{p}$ and therefore a clause of the form:

$$\mathbf{p}_1(\mathbf{Y}) \leftarrow \mathbf{p}(\mathbf{X}, \mathbf{Y}),\ \mathbf{p}_0(\mathbf{X}). \tag{6}$$

has been intoduced in $\mathbf{P}^*$. Assume now that $R\ \mathbf{p}_1(\mathbf{b})$ has been introduced in $T_{\mathbf{P}^*_\mathbf{D}} \uparrow (k+1)$ by clause (6) above. Then there exists a constant $\mathbf{a}$ such that $\mathbf{p}(\mathbf{a}, \mathbf{b}) \in T_{\mathbf{P}_\mathbf{D}} \uparrow 1$, and $R\ \mathbf{p}_0(\mathbf{a}) \in T_{\mathbf{P}^*_\mathbf{D}} \uparrow k$.

This concludes the proof of the particular case and of the lemma. □

Lemma 4. *Let* $\mathbf{P}$ *be a simple Chain Datalog program*, $\mathbf{D}$ *be a database and* $\leftarrow\mathbf{p}(\mathbf{a}, \mathbf{X})$ *be a goal clause. Let* $\mathbf{P}^* \cup \{\leftarrow \mathtt{first}\ \mathbf{p}_1(\mathbf{X})\}$ *be the output obtained by applying the transformation algorithm to* $\mathbf{P} \cup \{\leftarrow\mathbf{p}(\mathbf{a}, \mathbf{X})\}$. *If* $\mathtt{first}\ \mathbf{p}_1(\mathbf{b}) \in T_{\mathbf{P}^*_\mathbf{D}} \uparrow \omega$ *then* $\mathbf{p}(\mathbf{a}, \mathbf{b}) \in T_{\mathbf{P}_\mathbf{D}} \uparrow \omega$.

Proof: From Lemma 3 we have that there is a constant $\mathbf{c}$ such that $\mathbf{p}(\mathbf{c}, \mathbf{b}) \in \mathrm{T}_{\mathbf{P}_\mathbf{D}} \uparrow \omega$ and $\mathtt{first}\ \mathbf{p}_0(\mathbf{c}) \in \mathrm{T}_{\mathbf{P}^*_\mathbf{D}} \uparrow \omega$. But as the only instance of $\mathtt{first}\ \mathbf{p}_0(\mathbf{X})$ in $\mathrm{T}_{\mathbf{P}^*_\mathbf{D}} \uparrow \omega$ is $\mathtt{first}\ \mathbf{p}_0(\mathbf{a})$ then $\mathbf{c} = \mathbf{a}$. □

Theorem 3. *Let* $\mathbf{P}$ *be a simple Chain Datalog program*, $\mathbf{D}$ *be a database and* $\leftarrow\mathbf{p}(\mathbf{a}, \mathbf{X})$ *be a goal clause. Let* $\mathbf{P}^* \cup \{\leftarrow \mathtt{first}\ \mathbf{p}_1(\mathbf{X})\}$ *be the output obtained by applying the transformation algorithm to* $\mathbf{P} \cup \{\leftarrow\mathbf{p}(\mathbf{a}, \mathbf{X})\}$. *Then* $\mathtt{first}\ \mathbf{p}_1(\mathbf{b}) \in T_{\mathbf{P}^*_\mathbf{D}} \uparrow \omega$ *iff* $\mathbf{p}(\mathbf{a}, \mathbf{b}) \in T_{\mathbf{P}_\mathbf{D}} \uparrow \omega$.

Proof: It is an immediate consequence of Lemmas 2 and 4. □

6. Termination of bottom-up evaluation

It is customary in the deductive database area to investigate bottom-up evaluation strategies for Datalog programs (Naughton and Ramakrishnan, 1991). In particular, queries for such programs can be evaluated in a bottom-up way using essentially the definition of the operator $T_{\mathbf{P}_\mathbf{D}}$. As the Herbrand universe of a Datalog program is finite, the calculation of its least fixpoint is completed in a finite number of steps.

Branching Datalog programs can also be evaluated bottom-up through the use of the $T_{\mathbf{P}_\mathbf{D}}$ operator of Definition 6. However, as the temporal Herbrand base of a Branching Datalog program is (in general) infinite, the calculation of the least fixpoint may not terminate in a finite number of iterations.

Fortunately, in the case of the Branching Datalog programs obtained by the transformation, the calculation of the answers to the goal clause requires only a finite number of iterations. In fact, the number of steps for this calculation is bounded by a number which depends on certain characteristics of the program (e.g. the number of unit clauses

in the database of **P**, the number of different data constants in the database, etc.). In order to prove this claim we use the results of Chomicki (1995) which refer to the language Datalog$_{nS}$.

For this, we transform the Branching Datalog program (together with the corresponding goal clause), that has been obtained by the branching-time transformation algorithm, into a Datalog$_{nS}$ program. This transformation is defined as follows:

– Replace every IDB atom of the form $\mathbf{p}(\mathbf{e})$ with $\mathbf{p}(\mathbf{T}, \mathbf{e})$.
– Replace every IDB atom of the form $\texttt{next}_i\ \mathbf{p}(\mathbf{e})$ with $\mathbf{p}([i \mid \mathbf{T}], \mathbf{e})$.
– Replace every IDB atom of the form $\texttt{first}\ \mathbf{p}(\mathbf{e})$ with $\mathbf{p}([\ ], \mathbf{e})$.

Example 3. Consider the following Chain Datalog program:

$$\texttt{p(X,Y)} \leftarrow \texttt{e(X,Y)}.$$

where e is an EDB predicate, and the goal clause

$$\leftarrow \texttt{p(a,Y)}.$$

The output of the branching-time transformation is:

$$\leftarrow \texttt{first}\ \texttt{p}_1\texttt{(Y)}.$$
$$\texttt{first}\ \texttt{p}_0\texttt{(a)}.$$
$$\texttt{p}_1\texttt{(Y)} \leftarrow \texttt{next}_1\ \texttt{e}_1\texttt{(Y)}.$$
$$\texttt{next}_1\ \texttt{e}_0\texttt{(X)} \leftarrow \texttt{p}_0\texttt{(X)}.$$
$$\texttt{e}_1\texttt{(Y)} \leftarrow \texttt{e(X, Y)}, \texttt{e}_0\texttt{(X)}.$$

The above can be transformed into the following Datalog$_{nS}$ program together with a goal clause:

$$\leftarrow \texttt{p}_1\texttt{([ ], Y)}.$$
$$\texttt{p}_0\texttt{([ ],a)}.$$
$$\texttt{p}_1\texttt{(T, Y)} \leftarrow \texttt{e}_1\texttt{([1 | T] , Y)}.$$
$$\texttt{e}_0\texttt{([1 | T] ,X)} \leftarrow \texttt{p}_0\texttt{(T, X)}.$$
$$\texttt{e}_1\texttt{(T,Y)} \leftarrow \texttt{e(X,Y)} , \texttt{e}_0\texttt{(T,X)}.$$

The following lemma demonstrates the equivalence between the Branching Datalog program and the corresponding Datalog$_{nS}$ program.

Lemma 5. *Let* $\mathbf{P}^*$ *be a Branching Datalog program that results from the branching-time transformation and* $\mathbf{D}$ *be a database. Let* $\mathbf{P}^{nS}$ *be the Datalog*$_{nS}$ *program that results from the above transformation. Then, for all* $k \in \mathcal{N}$,

$$\texttt{first}\ \texttt{next}_{i_1} \ldots \texttt{next}_{i_n}\ \mathbf{p}(\mathbf{a}) \in T_{\mathbf{P}^*_{\mathbf{D}}} \uparrow k \textit{ iff } \mathbf{p}([i_n, \ldots, i_1], \mathbf{a}) \in T_{\mathbf{P}^{nS}_{\mathbf{D}}} \uparrow k.$$

Proof: The proof is obtained by a straightforward induction on k. □

In Chomicki (1995), J. Chomicki demonstrates that for Datalog$_{nS}$ programs there exists a bound on the number of iterations of the immediate consequence operator to produce all answers to a specific class of queries (which include the queries used in our case). In particular, given a Datalog$_{nS}$ program $\mathbf{P}$ and a database $\mathbf{D}$, J. Chomicki derives a closed formula that can be used to easily calculate this bound based on certain characteristics of $\mathbf{P} \cup \mathbf{D}$.

This result is used to derive the following theorem:

Theorem 4. *Let* $\mathbf{P}$ *be a simple Chain Datalog program,* $\leftarrow \mathbf{p}(\mathbf{a}, \mathbf{X})$ *be a goal clause, and* $\mathbf{P}^*$ *be the Branching Datalog program obtained by applying the branching-time transformation algorithm to* $\mathbf{P} \cup \{\leftarrow \mathbf{p}(\mathbf{a}, \mathbf{X})\}$. *Then for every database* $\mathbf{D}$ *there is a natural number* k (*easily determinable from the characteristics of* $\mathbf{P} \cup \mathbf{D}$) *such that all the answers to the goal clause* $\leftarrow \texttt{first}\ \mathbf{p}_1(\mathbf{X})$ *can be computed* (*bottom-up*) *in at most* k *iterations.*

Proof: As discussed above, $\mathbf{P}^*$ can be transformed into a Datalog$_{nS}$ program $\mathbf{P}^{nS}$. As it is shown in Chomicki (1995), for every Datalog$_{nS}$ program $\mathbf{P}^{nS}$ there is a natural number $m(\mathbf{P}^{nS})$ such that all the answers to a goal clause can be computed in $m(\mathbf{P}^{nS})$ iterations. Because of Lemma 5 the corresponding answers to the goal clauses in both programs are obtained in the same number of steps (i.e. $k = m(\mathbf{P}^{nS})$). This completes the proof of the theorem. □

The above theorem is a somewhat surprising one. In general the immediate consequence operator for Branching Datalog programs does not terminate. However the above theorem suggests that all answers to a given query will be obtained in a finite number of iterations.

It should be noted that Branching Datalog programs can also be executed top-down using a resolution-type proof procedure (Rondogiannis et al., 1998). However, a further discussion of proof procedures for Branching Datalog is outside the scope of the present paper.

7. Improving the transformation

In this section we show that the branching-time transformation technique can be further improved. In particular: (a) unary predicates corresponding to EDB atoms in the initial program can be eliminated (b) some of the operators $\texttt{next}_i$ introduced by the transformation can be eliminated, and (c) temporal operators concerning left recursive calls of the head predicate of a clause can be eliminated.

(a) *Elimination of unary predicates corresponding to EDB atoms*: All unary predicates in the resulting program that correspond to EDB predicates of the source program can be eliminated using unfolding (Gergatsoulis and Spyropoulos, 1998). In particular, the predicates in the heads of the clauses added to $\mathbf{P}^*$ in step 1 of the algorithm (which are output predicates) appear only in the bodies of (some of) the clauses in $\mathbf{P}^*$ added in steps 2–4. All the clauses containing these body atoms can be unfolded using the clauses introduced in Step 1. The clauses obtained by the above unfolding steps contain occurrences of

input predicates corresponding to EDB predicates. These can be further unfolded resulting in the complete elimination of all unary predicates corresponding to EDB predicates of the initial program. Notice that all temporal operators corresponding to EDB atoms in the initial program are also eliminated.

Example 4. Let **P*** be the program obtained in Example 2:

$\leftarrow$first p_1(Y).
(**C**$_1$) first p_0(a).
(**C**$_2$) p_1(Z) $\leftarrow$ next$_1$ e_1(Z).
(**C**$_3$) next$_1$ e_0(X) $\leftarrow$ p_0(X).
(**C**$_4$) p_1(Z) $\leftarrow$ next$_3$ e_1(Z).
(**C**$_5$) next$_3$ e_0(Y) $\leftarrow$ next$_2$ p_1(Y).
(**C**$_6$) next$_2$ p_0(X) $\leftarrow$ p_0(X).
(**C**$_7$) e_1(Y) $\leftarrow$ e(X,Y), e_0(X).

Then unfolding **C**$_2$ using **C**$_7$ we get:

(**C**$_8$) p_1(Z) $\leftarrow$ e(X,Z), next$_1$ e_0(X).

Unfolding further C_8 using C_3 we get:

(**C**$_9$) p_1(Z) $\leftarrow$ e(X,Z), p_0(X).

In a similar way, after performing all the unfoldings described above, we get the following program:

$\leftarrow$first p_1(Y).
first p_0(a).
p_1(Z) $\leftarrow$ e(X,Z), p_0(X).
p_1(Z) $\leftarrow$ e(X,Z), next$_2$ p_1(X).
next$_2$ p_0(X) $\leftarrow$ p_0(X).

(b) *Elimination of redundant next operators*: The number of different next$_i$ operators introduced by the transformation algorithm when applied to a program **P**, is equal to the total number of atoms in the bodies of all clauses in **P**. However, some of these operators are redundant in the final Branching Datalog program.

The generation of redundant next operators can be avoided by redefining steps 2 and 3 of the transformation algorithm as follows:

2′. Each clause in **P** of the form:

$$\mathbf{p}(\mathbf{X}, \mathbf{Y}) \leftarrow \mathbf{q}(\mathbf{X}, \mathbf{Y}).$$

is transformed into two clauses in $\mathbf{P}^*$ of the form:

$$\mathbf{p}_1(\mathbf{Y}) \leftarrow Op\ \mathbf{q}_1(\mathbf{Y}).$$
$$Op\ \mathbf{q}_0(\mathbf{X}) \leftarrow \mathbf{p}_0(\mathbf{X}).$$

where Op is $\texttt{next}_\text{i}$ if there is another body atom in $\mathbf{P} \cup \{\mathbf{G}\}$, with the same predicate symbol $\mathbf{q}$. Otherwise, Op is empty.

3′. Each non unit clause in $\mathbf{P}$ of the form:

$$\mathbf{p}(\mathbf{X}, \mathbf{Y}) \leftarrow \mathbf{q}(\mathbf{X}, \mathbf{Z}), \mathbf{r}(\mathbf{Z}, \mathbf{Y}).$$

is transformed into the set of clauses:

$$\mathbf{p}_1(\mathbf{Y}) \leftarrow Op_1\ \mathbf{r}_1(\mathbf{Y}).$$
$$Op_1\ \mathbf{r}_0(\mathbf{Z}) \leftarrow Op_2\ \mathbf{q}_1(\mathbf{Z}).$$
$$Op_2\ \mathbf{q}_0(\mathbf{X}) \leftarrow \mathbf{p}_0(\mathbf{X}).$$

where Op_1 is $\texttt{next}_i$ if there is another body atom in $\mathbf{P} \cup \{\mathbf{G}\}$, with the same predicate symbol r; otherwise Op_1 is empty. Op_2 is $\texttt{next}_j$ if there is an another body atom in a clause in $\mathbf{P}$, with the same predicate symbol $\mathbf{q}$; otherwise Op_2 is empty.

Notice that in steps (2′) and (3′) the call in the goal $\mathbf{G}$ counts as an appearance of the corresponding predicate in a body atom. It is therefore clear that the improvement proposed here refers to some of the predicates (either non-recursive or implicitly recursive) in particular those appearing only once in the bodies of the clauses in $\mathbf{P}$ and the goal $\mathbf{G}$.

(c) *Elimination of temporal operators concerning left recursive calls*: Temporal operators that concern left recursive calls can be eliminated. More specifically, each non unit clause in $\mathbf{P}$ of the form:

$$\mathbf{p}(\mathbf{X}, \mathbf{Y}) \leftarrow \mathbf{p}(\mathbf{X}, \mathbf{Z}), \mathbf{r}(\mathbf{Z}, \mathbf{Y}).$$

is transformed into the set of clauses:

$$\mathbf{p}_1(\mathbf{Y}) \leftarrow \text{next}_i\ \mathbf{r}_1(\mathbf{Y}).$$
$$\text{next}_i\ \mathbf{r}_0(\mathbf{Z}) \leftarrow \mathbf{p}_1(\mathbf{Z}).$$

Example 5. The program of Example 4 after applying the elimination of temporal operators concerning left recursion and the elimination of predicates corresponding to EDB atoms, results in the much simpler program:

$\leftarrow$`first` $\texttt{p}_1(\texttt{Y})$.
`first` $\texttt{p}_0(\texttt{a})$.
$\texttt{p}_1(\texttt{Z}) \leftarrow \texttt{e(X,Z)}, \texttt{p}_0(\texttt{X})$.
$\texttt{p}_1(\texttt{Z}) \leftarrow \texttt{e(X,Z)}, \texttt{p}_1(\texttt{X})$.

The correctness of all the above transformations can be easily demonstrated by performing a more detailed case analysis in the proof of Section 5. We believe that there exist other interesting optimizations that are applicable to the branching transformation. For example, we believe that there exists a corresponding optimization concerning *right recursive* clauses. However, we have not been able to derive such an optimization (the proofs do not appear to extend easily to cover such a case). We are currently further investigating such a possibility.

It is important to note that all the above optimizations could actually be embedded inside the transformation algorithm. However, we have preferred to list them separately in order to make clearer both the presentation and the correctness proof of the transformation.

As a final remark to this section it is worth mentioning that the optimizations defined above are quite effective: a large class of Chain Datalog programs can be transformed into Branching Datalog ones that contain very few `next` operators (or in many cases no temporal operators at all). Examples of this phenomenon will be given in the next section.

8. Related work

The transformation presented in this paper belongs and contributes to the research area of query optimization for deductive database systems. The particular area is far from new: many important results have been obtained during the last 15 years and the associated bibliography is very extensive. In the rest of this section we describe the connections between the branching transformation and methods that have been developed in the deductive query optimization domain. Moreover, we provide certain examples that demonstrate that the branching transformation is indeed a promising one. However, the comparison attempted in this section is indeed at a very initial stage. More experiments are clearly needed and in order to perform a fair comparison, one needs to implement all the techniques presented below, optimize them, and test them in a large number of examples. Such a detailed experimentation is clearly outside the scope of the present paper. The examples presented below have been chosen in such a way so as to emphasize certain "strong points" of the branching transformation.

The branching transformation belongs to a family of optimization techniques for deductive databases that are based on the idea of propagating the input values of the top level goal in order to restrict the generation of atoms in the bottom-up computation. Such techniques are the *counting technique* (Saccà and Zaniolo, 1988), the *pushdown approach* (Greco et al., 1999), and the *magic sets transformation* (Beeri and Ramakrishnan, 1991; Sippu and Soisalon-Soininen, 1996).

The magic sets technique is one of the most well-known transformations for optimizing Datalog queries. In this technique, for each IDB predicate a magic predicate is introduced. The arguments of the magic predicate related to an IDB predicate **p**, correspond to the bound (input) arguments of **p**. The role of the magic predicates is to propagate the value of the bound arguments, in order to restrict the set of atoms produced in a bottom-up computation. Consider the following program:

```
←p(a, Y).
p(X, Y) ← e(X, Y).
p(X, Y) ← p(X, Y), e(Z, Y).
```

The magic transformation of the above program produces as output the following program:

$\leftarrow$p(a, Y).
m_p(a).
m_p(X) $\leftarrow$ sup$_{2.0}$(X).
sup$_{1.0}$(X) $\leftarrow$ m_p(X)
sup$_{2.0}$(X) $\leftarrow$ m_p(X).
sup$_{2.1}$(X, Z) $\leftarrow$ sup$_{2.0}$(X), p(X, Z).
p(X, Y) $\leftarrow$ sup$_{1.0}$(X), e(X, Y).
p(X, Y) $\leftarrow$ sup$_{2.1}$(X, Z), e(Z, Y).

The target program that results from the branching transformation (as demonstrated in Example 5) is the following:

$\leftarrow$first p$_1$(Y).
first p$_0$(a).
p$_1$(Z) $\leftarrow$ e(X, Z),p$_0$(X).
p$_1$(Z) $\leftarrow$ e(X, Z),p$_1$(X).

Notice that the above program is actually a Datalog one (the first operator can be removed without affecting the answer set). The above program is a much more efficient one than the program produced by the magic approach (in which the introduction of the magic predicates has almost no effect). A magic program that could compete with the above one could only be produced if one applied advanced optimization techniques concerning left linear recursion (as those described in Chapter 15 of Ullman (1989)).

In the counting approach integer indices are used in order to encode the structure of the computation used to generate an atom. Similarly, in the branching transformation sequences of operators are employed in order to control the generation of results. However, to our knowledge, the counting approach has never before been extended to apply to the whole class of chain queries. Counting was initially defined for certain restricted classes of queries (Ullman, 1989; Haddad and Naughton, 1988). The method was later extended to *generalized counting* which covers a significantly broader class of queries (Saccà and Zaniolo, 1988). More specifically, the generalized counting approach covers all those queries that have the *binding-passing property* (*BPP*) (Saccà and Zaniolo, 1988). Intuitively, BPP guarantees that "bindings can be passed to any level of recursion." Although broad, the class of BPP queries does not include all the Chain Datalog ones. For example, consider the program:

$\leftarrow$p(a, Y).
p(X, Y) $\leftarrow$ e(X, Y).
p(X, Y) $\leftarrow$ p(X, Z), p(Z, Y).

which is a double recursive version of the well known transitive closure problem. It can be shown that the above program does not satisfy the BPP. More specifically, the *binding graph*

(Saccà and Zaniolo, 1988) of the program contains a node whose set of bound arguments is empty. There exist however chain programs that are transformable by the generalized counting technique. Consider for example the program:

$\leftarrow$p(a, Y).
p(X, Z) $\leftarrow$ p(X, Y), e(Y, Z).
p(X, Z) $\leftarrow$ p(X, Y), f(Y, Z).
p(X, Z) $\leftarrow$ g(X, Z).

in which e, f and g are EDB predicates. The transformation of the above query under the generalized counting scheme is the following:

p(a, Y) $\leftarrow$ p^1(0, 0, 0, Y).
cnt.p^1(0, 0, 0, a).
cnt.p^1(J + 1, 2 * K, H, X) $\leftarrow$ cnt.p^1(J, K, H, X).
cnt.p^1(J + 1, 2 * K + 1, H, X) $\leftarrow$ cnt.p^1(J, K, H, X).
p^1(J - 1, K/2, H, Z) $\leftarrow$ cnt.p^1(J, K, H, Y), e(Y, Z).
p^1(J - 1, (K - 1)/2, H, Z) $\leftarrow$ p^1(J, K, H, Y), f(Y, Z).
p^1(J, K, H, Z) $\leftarrow$ cnt.p^1(J, K, H, X), g(X, Z).

On the other hand, the corresponding program obtained by the branching transformation (and optimized according to Section 7), is the following:

$\leftarrow$first p_1(Y).
first p_0(a).
p_1(Z) $\leftarrow$ g(X, Z), g_0(X).
g_0(X) $\leftarrow$ p_0(X).
p_1(Z) $\leftarrow$ e(Y, Z), e_0(Y).
e_0(Y) $\leftarrow$ p_1(Y).
p_1(Z) $\leftarrow$ f(Y, Z), f_0(Y).
f_0(Y) $\leftarrow$ p_1(Y).

It is important to note that in the above program all next operators have been eliminated (and in this case the operator first is actually useless and can also be eliminated). In other words, the program produced is in fact a classical Datalog program. On the other hand, the counting approach introduces extra arguments playing the role of counters (of which two are actually used in this example) and therefore, the resulting program is much less efficient.

In Greco et al. (1999), an approach which optimizes chain programs, called *pushdown method*, is proposed. The pushdown method is based on the relationship between chain queries and context-free languages. More specifically, the method is based on the fact that a chain query can be associated to a context-free language. The relationship between context-free languages and pushdown automata is then used to rewrite the queries in a form suitable for bottom up evaluation.

Using the pushdown method, the same example is transformed as follows:

```
← q(Y, [ ]).
q(a, [p]).
q(Y, T]) ← q(X, [p|T]), g(X, Y).
q(Y, [p,e|T]) ← q(Y, [p|T]).
q(Y, [p,f|T]) ← q(Y, [p|T]).
q(Y, T) ← q(X, [e|T]), e(X, Y).
q(Y, T) ← q(X, [f|T]), f(X, Y).
```

The program obtained by the pushdown method is clearly less simple than the one produced by the branching transformation (as it still contains lists of constants).

Generally, the pushdown method covers the same class of Datalog programs as the branching transformation technique, namely the Chain Datalog programs. However, it differs from our transformation in the following: (a) The pushdown method adds an extra argument to IDB predicates. This extra argument is a list of constants. Our method uses temporal operators instead. (b) The program obtained by the pushdown method has only one IDB predicate, while the programs obtained by the branching transformation use several IDB predicates (in fact two IDB predicates for every different predicate of the initial program). The use of different predicates often results in more efficient evaluation of the programs. (c) More than one constant symbols are often put in the list argument in the program obtained by the pushdown method. On the other hand, in the branching transformation technique, at most one temporal operator is applied to each atom.

Our transformation algorithm also relates to techniques that aim at transforming nonlinear into linear Datalog programs (Afrati et al., 1996). However, if the target language is (classical) Datalog, then the whole class of chain queries cannot be linearized (Afrati and Cosmadakis, 1989).

9. Conclusions

In this paper, we introduce a transformation algorithm from Chain Datalog programs to Branching Datalog ones. The programs obtained have the following interesting properties: (a) All IDB predicates are unary, and (b) Every rule has at most one IDB atom in its body.

There are certain points however which we believe require further investigation:

Implementation Issues: Apart from its theoretical interest, the transformation algorithm can be viewed as the basis of new evaluation strategies for Chain Datalog programs. An interesting point for future investigation would be to consider the performance of the proposed transformation algorithm when compared with the standard procedures for implementing Chain Datalog (or simply Datalog) programs.

Extension of the Transformation: Clearly, the transformation does not apply directly to general Datalog programs (that are not in chain form). We believe however that the algorithm can be extended to larger subsets of Datalog. We are currently investigating such a possibility.

Acknowledgments

The authors would like to thank F. Afrati, C. Nomikos and P. Potikas for their helpful comments and suggestions. We would also like to thank the anonymous reviewers for all their insightful comments (in particular, the comments of one of the reviewers motivated us in introducing the optimization of left recursion).

Notes

1. Such programs are usually called *linear* in the deductive database terminology. However we will avoid using this term, as it may be confused with the term *linear* which is often used to characterize time in temporal logic programming.
2. Lemma 1 is a result that concerns all the predicates of the source program. It is not obvious how one can prove Lemma 2 without resorting to such a more general result.
3. It can be easily seen that Case 1 of the induction step is only applicable for values of k which are greater than 2.

References

Afrati, F. and Cosmadakis, S. (1989). Expressiveness of Restricted Recursive Queries. In *Proc. 21st ACM Symp. on Theory of Computing* (pp. 113–126).

Afrati, F., Gergatsoulis, M., and Katzouraki, M. (1996). On Transformations into Linear Database Logic Programs. In D. Bjørner, M. Broy and I. Pottosin (Eds.), *Perspectives of Systems Informatics (PSI'96), Proceedings* (pp. 433–444).

Afrati, F. and Papadimitriou, C.H. (1987). The Parallel Complexity of Simple Chain Queries. In *Proc. 6th ACM Symposium on Principles of Database Systems* (pp. 210–213).

Afrati, F. and Papadimitriou, C.H. (1993). Parallel Complexity of Simple Logic Programs, *Journal of the ACM*, 40(3), 891–916.

Baudinet, M., Chomicki, J., and Wolper, P. (1993). Temporal Deductive Databases. In L.F. del Cerro and M. Penttonen (Eds.), *Temporal Databases: Theory, Design, and Implementation* (pp. 294–320). The Benjamin/Cummings Publishing Company, Inc.

Beeri, C. and Ramakrishnan, R. (1991). On the power of magic, *The Journal of Logic Programming*, 10(1–4), 255–299.

Chomicki, J. (1995). Depth-Bounded Bottom-Up evaluation of Logic Programs, *The Journal of Logic Programming*, 25(1), 1–31.

Dong, G. and Ginsburg, S. (1995). On Decompositions of Chain Datalog Programs into $\mathcal{P}$ (Left-)Linear 1-Rule Components, *The Journal of Logic Programming*, 23(3), 203–236.

Du, W. and Wadge, W.W. (1990). The Eductive Implementation of a Three-dimensional Spreadsheet, *Software-Practice and Experience*, 20(11), 1097–1114.

Faustini, A. and Wadge, W. (1987). An Eductive Interpreter for the Language pLucid. In *Proceedings of the SIGPLAN 87 Conference on Interpreters and Interpretive Techniques (SIGPLAN Notices* 22(7)) (pp. 86–91).

Gergatsoulis, M. (2001). Temporal and Modal Logic Programming Languages. In A. Kent and J.G. Williams (Eds.), *Encyclopedia of Microcomputers, Vol. 27*, Suppl. 6. Marcel Decker, Inc., pp. 393–408.

Gergatsoulis, M. and Katzouraki, M. (1994). Unfold/Fold Transformations for Definite Clause Programs. In M. Hermenegildo and J. Penjam (Eds.), *Programming Language Implementation and Logic Programming (PLILP'*94*), Proceedings* (pp. 340–354).

Gergatsoulis, M. and Spyropoulos, C. (1998). Transformation Techniques for Branching-Time Logic Programs. In W.W. Wadge (Ed.), *Proc. of the* 11*th International Symposium on Languages for Intensional Programming (ISLIP'*98*), May* 7−9*, Palo Alto, California, USA* (pp. 48–63).

Greco, S., Saccà, D., and Zaniolo, C. (1999). Grammars and Automata to Optimize Chain Logic Queries, *International Journal on Foundations of Computer Science*, 10(3), 349–372.

Haddad, R.W. and Naughton, J.F. (1988). Counting Methods for Cyclic Relations. In *Proc. 7th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (pp. 333–340).

Jones, S.L.P. (1987). *The Implementation of Functional Programming Languages*. Prentice-Hall.

Lloyd, J.W. (1987). *Foundations of Logic Programming*. Springer-Verlag, Germany.

Naughton, J.F. and Ramakrishnan, R. (1991). Bottom-Up Evaluation of Logic Programs. In J.L. Lasser and G. Plotkin (Eds.), *Computational Logic. Essays in the Honor of Alan Robinson* (pp. 640–699). MIT Press.

Orgun, M.A. (1991). Intensional Logic Programming. Ph.D. Thesis, Dept. of Computer Science, University of Victoria, Canada.

Orgun, M.A. (1996). On Temporal Deductive Databases, *Computational Intelligence*, 12(2), 235–259.

Orgun, M.A. and Ma, W. (1994). An Overview of Temporal and Modal Logic Programming. In D.M. Gabbay and H.J. Ohlbach (Eds.), *Proc. of the First International Conference on Temporal Logics (ICTL'94)* (pp. 445–479).

Proietti, M. and Pettorossi, A. (1990). Synthesis of Eureka Predicates for Developing Logic Programs. In *Proc. of the 3rd European Symposium on Programming* (pp. 306–325).

Rondogiannis, P., Gergatsoulis, M., and Panayiotopoulos, T. (1998). Branching-Time Logic Programming: The Language Cactus and Its Applications, *Computer Languages*, 24(3), 155–178.

Rondogiannis, P. and Wadge, W.W. (1997). First-Order Functional Languages and Intensional Logic, *Journal of Functional Programming*, 7(1), 73–101.

Rondogiannis, P. and Wadge, W.W. (1999). Higher-Order Functional Languages and Intensional Logic, *Journal of Functional Programming*, 9(5), 527–564.

Saccà, D. and Zaniolo, C. (1988). The Generalized Counting Method for Recursive Logic Queries, *Theoretical Computer Science*, 4(4), 187–220.

Sippu, S. and Soisalon-Soininen, E. (1996). An Analysis of Magic Sets and Related Optimization Strategies for Logic Queries, *Journal of the ACM*, 43(6), 1046–1088.

Tamaki, H. and Sato, T. (1984). Unfold/Fold Transformations of Logic Programs. In S.-Å. Tarnlund (Ed.), *Proc. of the Second International Conference on Logic Programming* (pp. 127–138).

Ullman, J.D. (1989). *Principles of Database and Knowledge-Base Systems*, Vols. I & II. Computer Science Press, USA.

Ullman, J.D. and Gelder, A.V. (1988). Parallel Complexity of Logical Query Programs, *Algorithmica*, 3, 5–42.

Wadge, W.W. (1991). Higher-Order Lucid. In *Proceedings of the Fourth International Symposium on Lucid and Intensional Programming*.

Yaghi, A. (1984). The Intensional Implementation Technique for Functional Languages. Ph.D. Thesis, Dept. of Computer Science, University of Warwick, Coventry, UK.