THE BRANCHING-TIME TRANSFORMATION TECHNIQUE FOR CHAIN DATALOG PROGRAMS

PANOS RONDOGIANNIS and MANOLIS GERGATSOULIS

3-99

Preprint no. 3-99/1999

Department of Computer Science
University of Ioannina
451 10 Ioannina, Greece

The Branching-Time Transformation Technique for Chain Datalog Programs*

Panos Rondogiannis[1] and **Manolis Gergatsoulis**[2]

[1] Dept. of Computer Science, University of Ioannina,
P.O. BOX 1186, 45110 Ioannina, Greece,
e_mail: prondo@cs.uoi.gr

[2] Inst. of Informatics & Telecommunications,
National Centre for Scientific Research (NCSR) 'Demokritos',
153 10 A. Paraskevi Attikis, Athens, Greece
e_mail: manolis@iit.demokritos.gr

Abstract

The *branching-time transformation* technique has proven to be an efficient approach for implementing functional programming languages. In this paper it is demonstrated that such a technique can also be defined for logic programming languages. More specifically, a transformation algorithm is proposed from a subclass of logic programs (the Chain Datalog ones) to the class of unary branching-time logic programs which have at most one atom in the bodies of clauses. 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: Temporal logic programming, Program transformations, Deductive Databases, Chain Datalog.

1 Introduction

The *branching-time transformation* is a promising technique that has been used for implementing functional programming languages [21, 20, 13, 16, 17]. 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

*This work has been partially supported by the Greek General Secretariat of Research and Technology under the project "Logic Programming Systems and Environments for Developing Logic Programs" of ΠΕΝΕΔ'95, contract no 952.

program, which has a simpler structure and which can be easily evaluated using a demand-driven technique (also called *eduction* [6, 5]). The branching-time technique offers a promising alternative to the usual reduction-based implementations [8] 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, we define a transformation algorithm from the class of *Chain Datalog* programs [19, 1, 2, 4] to the class of unary branching-time logic programs which have (at most) one atom in the bodies of their clauses. In this way we set the basis for a new approach for implementing logic programming languages.

The following example is given in order to motivate the branching-time transformation. The precise presentation of all the concepts involved will be given in subsequent section:

Example 1.1. The following is a Chain Datalog program together with a goal clause:

$$\leftarrow \mathtt{p(a,Y)}.$$
$$\mathtt{p(X,Y)} \leftarrow \mathtt{q(X,Z)}, \mathtt{q(Z,Y)}.$$
$$\mathtt{q(a,b)}.$$
$$\mathtt{q(b,c)}.$$

The output of the transformation is:

$$\leftarrow \mathtt{first}\ \mathtt{p}_1\mathtt{(Y)}.$$
$$\mathtt{first}\ \mathtt{p}_0\mathtt{(a)}.$$
$$\mathtt{p}_1\mathtt{(Y)} \leftarrow \mathtt{next}_2\ \mathtt{q}_1\mathtt{(Y)}.$$
$$\mathtt{next}_2\ \mathtt{q}_0\mathtt{(Z)} \leftarrow \mathtt{next}_1\ \mathtt{q}_1\mathtt{(Z)}.$$
$$\mathtt{next}_1\ \mathtt{q}_0\mathtt{(X)} \leftarrow \mathtt{p}_0\mathtt{(X)}.$$
$$\mathtt{q}_1\mathtt{(b)} \leftarrow \mathtt{q}_0\mathtt{(a)}.$$
$$\mathtt{q}_1\mathtt{(c)} \leftarrow \mathtt{q}_0\mathtt{(b)}.$$

Notice that the resulting program contains only unary predicates and each clause has at most one atom in its body. However, the program also contains certain temporal operators (**first**, $\mathtt{next}_1$, $\mathtt{next}_2$) whose semantics will be introduced later in a later section. □

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

- A novel transformation algorithm from Chain Datalog programs into simple in structure branching-time logic programs is defined. The proposed transformation is the analog of the branching-time transformation that has been defined in the functional programming domain.

- The proposed algorithm can form the basis of new evaluation strategies for Chain Datalog programs. Such issues are discussed in later sections of the paper. It should be noted here

that the class of Chain Datalog programs has been considered as an especially interesting one in the area of deductive databases [19, 1, 2, 4].

- 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 language of Branching Datalog. Section 4 introduces the branching-time transformation algorithm. Section 5 proves the correctness of the proposed transformation. Section 6 discusses evaluation strategies for the programs that result from the transformation. Finally, section 7 concludes the paper with discussion of possible future extensions.

2 Preliminaries

A *Datalog program* is a set of Horn clauses in which terms are either constants or variables. Following the convention in the deductive database literature we separate the set of clauses in a Datalog program into two disjoint parts: the EDB part containing the unit clauses (facts) which are ground, and the IDB part containing the rules (i.e. the clauses with nonempty bodies). Predicates defined in the EDB are called EDB predicates, while those defined in the IDB are called IDB predicates. We also assume the following notation: *constants* are denoted by **a, b, c**, *variables* by **X, Y, Z** and predicates by **p, q, 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 [9].

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

Definition 2.1. [4] 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 **X, 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. Programs are denoted by **P**. A *goal* is of the form $\leftarrow \mathbf{q}(\mathbf{a},\mathbf{X})$, where **a** is a constant, **X** is a variable and **q** is a predicate.

Notice that each chain rule contains no constants and has at least one atom in its body. Moreover, we assume that the first argument of a goal atom is always ground. The necessity of this assumption will become clear in later sections.

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

Definition 2.2. A *simple Chain Datalog program* is one in which all rules have at most two atoms in their body.

The semantics of (Chain) Datalog programs can be defined in accordance to the semantics of classical logic programming. The notions of *minimum model* and *immediate consequence operator* $T_{\mathbf{P}}$, transfer directly [9].

3 Branching Datalog

The technique proposed in this paper transforms a given Chain Datalog program into a simpler in structure Datalog program that contains however appropriate temporal operators. The output language of the transformation will be referred from now on as *Branching Datalog*.

Branching Datalog programs are in fact *Cactus* programs [14, 15] without function symbols. Cactus is a temporal logic programming language in which time has a tree structure. For this reason, Cactus is especially appropriate for describing tree algorithms and computations. It should also be mentioned here that Cactus is an instance of the more general paradigm of intensional logic programming [11].

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 one of the form $\texttt{first next}_{i_1} \cdots \texttt{next}_{i_n}$, where $i_1, \ldots, i_n \in \mathcal{N}$ and $n \geq 0$. An *open temporal reference* is one of the form $\texttt{next}_{i_1} \cdots \texttt{next}_{i_n}$, where $i_1, \ldots, i_n \in \mathcal{N}$ and $n \geq 0$. A *temporal atom* is an atom preceded by either a canonical or an open temporal reference. A *temporal clause* is a formula of the form:

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

where $\mathbf{A}, \mathbf{B}_1, ..., \mathbf{B}_m$ are temporal atoms and $m \geq 0$. If $m = 0$, the clause is said to be a *unit temporal clause*, and when $m \geq 1$, the clause is said to be a *temporal rule*. A *Branching Datalog program* is a finite set of *temporal clauses*. In analogy to classical Datalog, the set of unit temporal clauses of a Branching Datalog program is considered to be the EDB part of the program; moreover, the set of temporal rules constitutes the IDB part of the program.

A *goal clause* in Branching Datalog is a formula of the form $\leftarrow \mathbf{A}_1, ..., \mathbf{A}_n$ where $\mathbf{A}_i$, $i = 1, ..., n$ are temporal atoms. 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 used will consist of a single atom.

The following example, taken from [15], illustrates the use of Branching Datalog:

Example 3.1. Consider the non-deterministic finite automaton shown in Figure 1 which accepts the regular language $L = (01 \cup 010)^*$. We can describe the behaviour of this automaton in Branching Datalog with the following program:

$$\begin{aligned}
&\texttt{first state(q0).}\\
&\texttt{next}_0\ \texttt{state(q1)} \leftarrow \texttt{state(q0).}\\
&\texttt{next}_1\ \texttt{state(q2)} \leftarrow \texttt{state(q1).}\\
&\texttt{next}_1\ \texttt{state(q0)} \leftarrow \texttt{state(q1).}\\
&\texttt{next}_0\ \texttt{state(q0)} \leftarrow \texttt{state(q2).}
\end{aligned}$$

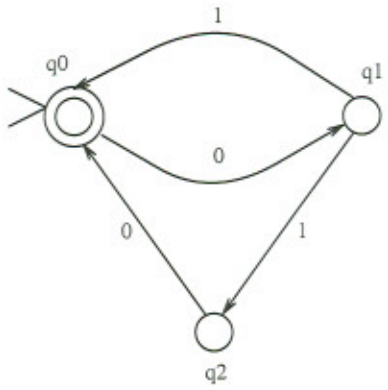

Figure 1: A non-deterministic finite automaton

Notice that in this automaton, q0 is both the initial and the final state. The atom in the goal clause:

$$\leftarrow \texttt{first next}_0\ \texttt{next}_1\ \texttt{next}_0\ \texttt{state(q0).}$$

is a logical consequence of the above program (see the discussion on the semantics of Branching Datalog that follows), which indicates that the string 010 belongs to the language L. □

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 $[i|t]$ (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 **A** is a formula then so are `first` **A** and $\texttt{next}_i$ **A**.

The semantics of temporal formulas of BTL are given using the notion of *branching temporal interpretation* [14, 15]. Branching temporal interpretations extend the (linear) temporal interpretations of the linear time logic of Chronolog [10].

Definition 3.1. 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. $\models_{I,t}$ **A** denotes that a formula **A** is true at a moment t in the temporal interpretation I:

Definition 3.2. The semantics of the elements of the temporal logic BTL are given inductively as follows:

1. For any n-ary predicate symbol **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. $\models_{I,t} \mathbf{A} \vee \mathbf{B}$ *iff* $\models_{I,t} \mathbf{A}$ *or* $\models_{I,t} \mathbf{B}$

5. $\models_{I,t} (\forall \mathbf{x})\mathbf{A}$ *iff* $\models_{I[d/\mathbf{x}],t} \mathbf{A}$ *for all* $d \in D$ *where the interpretation* $I[d/\mathbf{x}]$ *is the same as* I *except that the variable* **x** *is assigned the value* d.

6. $\models_{I,t}$ `first` **A** *iff* $\models_{I,[\,]} \mathbf{A}$

7. $\models_{I,t}$ `next`$_i$ **A** *iff* $\models_{I,[i|t]} \mathbf{A}$

If a formula **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 **A**. If for all interpretations I, $\models_I \mathbf{A}$, we say that **A** is *valid* and write $\models \mathbf{A}$.

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 is formalized below.

Definition 3.3. 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 $\mathbf{C}$.

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

In analogy to the theory of logic programming [9], it can be easily shown that the *model intersection property* holds for temporal Herbrand models. The intersection of all temporal Herbrand models denoted by $M(\mathbf{P})$, 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 $\mathbf{P}$. Again, the proof of the theorem is an easy extension of the corresponding proof for classical logic programming (see also the corresponding proof for the linear time logic programming language Chronolog [10]).

Theorem 3.1 *Let* $\mathbf{P}$ *be a Branching Datalog program. Then*

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

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 3.4. Let $\mathbf{P}$ be a Branching Datalog program and $TB_{\mathbf{P}}$ the temporal Herbrand base of $\mathbf{P}$. The operator $T_{\mathbf{P}} : 2^{TB_{\mathbf{P}}} \rightarrow 2^{TB_{\mathbf{P}}}$ is defined as follows:

$T_{\mathbf{P}}(I) = \{\mathbf{A} \mid \mathbf{A} \leftarrow \mathbf{B}_1, \ldots, \mathbf{B}_n$ is a canonical ground instance of a program clause in $\mathbf{P}$ and $\{\mathbf{B}_1, \ldots, \mathbf{B}_n\} \subseteq I\ \}$

It can be easily proved (see again the analogous proof for Chronolog [10]) that $2^{TB_{\mathbf{P}}}$ is a complete lattice under the partial order of set inclusion ($\subseteq$). Moreover, $T_{\mathbf{P}}$ is continuous and hence monotonic over the complete lattice $(2^{TB_{\mathbf{P}}}, \subseteq)$, and therefore $T_{\mathbf{P}}$ has a least fixpoint. The least fixpoint of $T_{\mathbf{P}}$ provides a characterization of the minimal Herbrand model of a Branching Datalog program, as it is shown in the following theorem.

Theorem 3.2 *Let* $\mathbf{P}$ *be a Branching Datalog program. Then*

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

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 4.1 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 [21, 16] in which a functional program contains a top-level definition of a special variable `result` whose value is the output of the program.

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

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

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

Proposition 4.1 *Every Chain Datalog program* $\mathbf{P}$ *can be transformed into a simple Chain Datalog program* $\mathbf{P}_s$ *such that for every predicate symbol* $\mathbf{p}$ *of* $\mathbf{P}$*, it holds* $M(\mathbf{P}, \mathbf{p}) = M(\mathbf{P}_s, \mathbf{p})$.

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

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

where $k \geq 2$. The rule (1) can be replaced by the two following rules (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). We can apply the same process on clause (3), and continuing in this way we end-up with a simple Chain Datalog program $\mathbf{P}_s$.

It is easy to see that $M(\mathbf{P}, \mathbf{p}) = M(\mathbf{P}_s, \mathbf{p})$ as the new clauses we introduce can be considered as *Eureka* definitions while the replacement of $\mathbf{q}_2(\mathbf{Y}_1, \mathbf{Y}_2), \ldots, \mathbf{q}_{k+1}(\mathbf{Y}_k, \mathbf{Z})$ in the body of (1) by $\mathbf{r}(\mathbf{Y}_1, \mathbf{Z})$ is a folding step [18, 7]. Now the desired result is an immediate consequence of the correctness of the unfold/fold transformations. □

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 predicate $\mathbf{p}$ in $\mathbf{P}$ two unary predicates $\mathbf{p}_0$ and $\mathbf{p}_1$ are introduced, where $\mathbf{p}_0$ corresponds to the first argument of $\mathbf{p}$ and $\mathbf{p}_1$ to the second. 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. When processing a rule of the source program, the algorithm introduces branching-time operators of the form $\mathtt{next}_i$, $i \in \mathcal{N}$. The operators introduced for a given rule are assumed to have different indices than the operators used for any other rule.

1. Each unit clause (fact) in $\mathbf{P}$ of the form:

   $$\mathbf{p}(\mathbf{a}, \mathbf{b}).$$

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

   $$\mathbf{p}_1(\mathbf{b}) \leftarrow \mathbf{p}_0(\mathbf{a}).$$

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:

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

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

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

is transformed into the set of clauses:

$$\begin{aligned}&\mathtt{p}_1(\mathtt{Y}) \leftarrow \mathtt{next}_i\ \mathtt{r}_1(\mathtt{Y}).\\ &\mathtt{next}_i\ \mathtt{r}_0(\mathtt{Z}) \leftarrow \mathtt{next}_j\ \mathtt{q}_1(\mathtt{Z}).\\ &\mathtt{next}_j\ \mathtt{q}_0(\mathtt{X}) \leftarrow \mathtt{p}_0(\mathtt{X}).\end{aligned}$$

where $i \neq j$.

4. The goal clause:

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

is transformed into:

$$\begin{aligned}&\leftarrow \mathtt{first}\ \mathtt{p}_1(\mathtt{Y}).\\ &\mathtt{first}\ \mathtt{p}_0(\mathtt{a}).\end{aligned}$$

Example 4.1. Let $\mathbf{P} = \{\mathbf{I}_1, \mathbf{I}_2\} \cup \{\mathbf{E}_1, \mathbf{E}_2, \mathbf{E}_3\}$ be a Chain Datalog program and $\mathbf{G}$ be a goal clause, where:

$$\begin{aligned}&(\mathbf{G}) && \leftarrow\ \mathtt{p}(\mathtt{a},\mathtt{Y}).\\ &(\mathbf{I}_1) && \mathtt{p}(\mathtt{X},\mathtt{Z}) \leftarrow\ \mathtt{e}(\mathtt{X},\mathtt{Z}).\\ &(\mathbf{I}_2) && \mathtt{p}(\mathtt{X},\mathtt{Z}) \leftarrow\ \mathtt{p}(\mathtt{X},\mathtt{Y}),\mathtt{e}(\mathtt{Y},\mathtt{Z}).\\ &(\mathbf{E}_1) && \mathtt{e}(\mathtt{a},\mathtt{b}).\\ &(\mathbf{E}_2) && \mathtt{e}(\mathtt{b},\mathtt{c}).\\ &(\mathbf{E}_3) && \mathtt{e}(\mathtt{c},\mathtt{d}).\end{aligned}$$

Transforming the goal clause $\mathbf{G}$ we get:

$$\begin{aligned}&\leftarrow\ \mathtt{first}\ \mathtt{p}_1(\mathtt{Y}).\\ &\mathtt{first}\ \mathtt{p}_0(\mathtt{a}).\end{aligned}$$

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

$$\begin{aligned}&\mathtt{p}_1(\mathtt{Z}) \leftarrow\ \mathtt{next}_1\ \mathtt{e}_1(\mathtt{Z}).\\ &\mathtt{next}_1\ \mathtt{e}_0(\mathtt{X}) \leftarrow\ \mathtt{p}_0(\mathtt{X}).\end{aligned}$$

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

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

Finally, transforming the clauses $\mathbf{E}_1 - \mathbf{E}_3$ (corresponding to the EDB atoms) we get:

$$\begin{aligned} \mathtt{e}_1(\mathtt{b}) &\leftarrow \mathtt{e}_0(\mathtt{a}). \\ \mathtt{e}_1(\mathtt{c}) &\leftarrow \mathtt{e}_0(\mathtt{b}). \\ \mathtt{e}_1(\mathtt{d}) &\leftarrow \mathtt{e}_0(\mathtt{c}). \end{aligned}$$

□

Notice that in the deductive database area it is customary to leave unchanged the EDB part of a database when performing a transformation. As it is shown in appendix A1, the proposed transformation can easily be modified so as that it leaves the EDB intact.

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

5 Correctness Proof

The correctness proof of the transformation proceeds as follows: at first we show that (see Lemma 5.2 below) if a ground instance $\mathbf{p(a,b)}$ of the atom in the goal clause $\leftarrow \mathbf{p(a,X)}$ is a logical consequence of the simple Chain Datalog program $\mathbf{P}$, then the atom $\mathtt{first}\ \mathbf{p}_1\mathbf{(b)}$ is a logical consequence of the program $\mathbf{P}^*$ obtained by applying the transformation algorithm to $\mathbf{P} \cup \{\leftarrow \mathbf{p(a,X)}\}$. In order to prove this result we establish a more general lemma (see Lemma 5.1 below). The inverse of Lemma 5.2 is given as Lemma 5.4. More specifically, we prove that whenever $\mathtt{first}\ \mathbf{p}_1\mathbf{(b)}$ is a logical consequence of $\mathbf{P}^*$ then $\mathbf{p(a,b)}$ is a logical consequence of $\mathbf{P}$. Again, we establish this result by proving the more general Lemma 5.3. Combining the above results we get the correctness proof of the transformation algorithm.

Lemma 5.1 *For all predicates* $\mathbf{p}$ *defined in* $\mathbf{P}$, *all canonical temporal references* R, *and all* $\mathbf{a}, \mathbf{b} \in U_{\mathbf{P}}$, *if* $R\ \mathbf{p}_0\mathbf{(a)} \in T_{\mathbf{P}^*} \uparrow \omega$ *and* $\mathbf{p(a,b)} \in T_{\mathbf{P}} \uparrow \omega$ *then* $R\ \mathbf{p}_1\mathbf{(b)} \in T_{\mathbf{P}^*} \uparrow \omega$.

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

Induction Basis:

To establish the induction basis, we need to show that if $R\ \mathbf{p}_0\mathbf{(a)} \in T_{\mathbf{P}^*} \uparrow \omega$ and $\mathbf{p(a,b)} \in T_{\mathbf{P}} \uparrow 0$ then $R\ \mathbf{p}_1\mathbf{(b)} \in T_{\mathbf{P}^*} \uparrow \omega$.

But $\mathbf{p(a,b)} \in T_{\mathbf{P}} \uparrow 0$ means that in $\mathbf{P}$ there exists a fact $\mathbf{p(a,b)}$. According to the transformation algorithm, in $\mathbf{P}^*$ there exists the rule $\mathbf{p}_1\mathbf{(b)} \leftarrow \mathbf{p}_0\mathbf{(a)}$. Using this and the fact that $R\ \mathbf{p}_0\mathbf{(a)} \in T_{\mathbf{P}^*} \uparrow \omega$ we conclude that $R\ \mathbf{p}_1\mathbf{(b)} \in T_{\mathbf{P}^*} \uparrow \omega$.

Induction Hypothesis:

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

Induction Step:

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

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

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

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

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

$$\mathbf{p}_1(\mathrm{Y}) \leftarrow \mathtt{next}_i\ \mathbf{r}_1(\mathrm{Y}). \tag{2}$$

$$\mathtt{next}_i\ \mathbf{r}_0(\mathrm{Z}) \leftarrow \mathtt{next}_j\ \mathbf{q}_1(\mathrm{Z}). \tag{3}$$

$$\mathtt{next}_j\ \mathbf{q}_0(\mathrm{X}) \leftarrow \mathbf{p}_0(\mathrm{X}). \tag{4}$$

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

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

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

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

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

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

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

This implies that $\mathbf{q}(\mathbf{a},\mathbf{b}) \in T_{\mathbf{P}} \uparrow k$. Consider now the transformation of the above clause (5) in program $\mathbf{P}^*$. The new clauses obtained are:

$$\mathbf{p}_1(\mathrm{Y}) \leftarrow \mathtt{next}_i\ \mathbf{q}_1(\mathrm{Y}). \tag{6}$$

$$\mathtt{next}_i\ \mathbf{q}_0(\mathrm{X}) \leftarrow \mathbf{p}_0(\mathrm{X}). \tag{7}$$

Using the assumption that $R\ \mathbf{p}_0(\mathbf{a}) \in T_{\mathbf{P}^\bullet} \uparrow \omega$ together with clause (7) above, we get that $R\ \mathtt{next}_i\ \mathbf{q}_0(\mathbf{a}) \in T_{\mathbf{P}^\bullet} \uparrow \omega$. Given this, we can now apply the induction hypothesis on $\mathbf{q}$ which gives:

Since $R\ \mathtt{next}_i\ \mathbf{q}_0(\mathbf{a}) \in T_{\mathbf{P}^\bullet} \uparrow \omega$ and $\mathbf{q}(\mathbf{a},\mathbf{b}) \in T_{\mathbf{P}} \uparrow k$ then $R\ \mathtt{next}_i\ \mathbf{q}_1(\mathbf{b}) \in T_{\mathbf{P}^\bullet} \uparrow \omega$.

But this together with clause (6) above gives $R\ \mathbf{p}_1(\mathbf{b}) \in T_{\mathbf{P}^*} \uparrow \omega$, which is the desired result. □

Lemma 5.2 *Let* $\mathbf{P}$ *be a simple Chain Datalog program and* $\leftarrow \mathbf{p}(\mathbf{a}, \mathbf{X})$ *be a goal clause. Let* $\mathbf{P}^*$ *be the Branching Datalog program obtained by applying the tranformation algorithm to* $\mathbf{P} \cup \{\leftarrow \mathbf{p}(\mathbf{a}, \mathbf{X})\}$. *If* $\mathbf{p}(\mathbf{a}, \mathbf{b}) \in T_{\mathbf{P}} \uparrow \omega$ *then* $\texttt{first}\ \mathbf{p}_1(\mathbf{b}) \in T_{\mathbf{P}^*} \uparrow \omega$.

Proof: Since by transforming the goal clause, the fact $\texttt{first}\ \mathbf{p}_0(\mathbf{a})$ is added to $\mathbf{P}^*$, this lemma is a special case of Lemma 5.1. □

We now show the following lemma which is the "inverse" of Lemma 5.1:

Lemma 5.3 *For all predicates* $\mathbf{p}$ *defined in* $\mathbf{P}$, *for all canonical temporal references* R, *and for all* $\mathbf{b} \in U_{\mathbf{P}}$, *if* $R\ \mathbf{p}_1(\mathbf{b}) \in T_{\mathbf{P}^*} \uparrow \omega$ *then there exists a constant* $\mathbf{a} \in U_{\mathbf{P}}$ *such that* $\mathbf{p}(\mathbf{a}, \mathbf{b}) \in T_{\mathbf{P}} \uparrow \omega$ *and* $R\ \mathbf{p}_0(\mathbf{a}) \in T_{\mathbf{P}^*} \uparrow \omega$.

Proof:

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

Induction Basis:

To establish the induction basis, we need to show that if $R\ \mathbf{p}_1(\mathbf{b}) \in T_{\mathbf{P}^*} \uparrow 0$ then there exists a constant $\mathbf{a}$ such that $\mathbf{p}(\mathbf{a}, \mathbf{b}) \in T_{\mathbf{P}} \uparrow \omega$ and $R\ \mathbf{p}_0(\mathbf{a}) \in T_{\mathbf{P}^*} \uparrow 0$.

But $R\ \mathbf{p}_1(\mathbf{b}) \in T_{\mathbf{P}^*} \uparrow 0$ is false because (as it can be easily seen from the definition of the transformation algorithm) in $T_{\mathbf{P}^*} \uparrow 0$ there only belongs one 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 $R\ \mathbf{p}_1(\mathbf{b}) \in T_{\mathbf{P}^*} \uparrow k$ then there exists a constant $\mathbf{a}$ such that $\mathbf{p}(\mathbf{a}, \mathbf{b}) \in T_{\mathbf{P}} \uparrow \omega$ and $R\ \mathbf{p}_0(\mathbf{a}) \in T_{\mathbf{P}^*} \uparrow k$.

Induction Step:

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

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

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

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

$$\mathbf{p}_1(\mathbf{Y}) \leftarrow \texttt{next}_i\ \mathbf{r}_1(\mathbf{Y}). \tag{2}$$

$$\texttt{next}_i\ \mathbf{r}_0(\mathbf{Z}) \leftarrow \texttt{next}_j\ \mathbf{q}_1(\mathbf{Z}). \tag{3}$$

$$\texttt{next}_j\ \mathbf{q}_0(\mathbf{X}) \leftarrow \mathbf{p}_0(\mathbf{X}). \tag{4}$$

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

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

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

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

Consider now the transformation of the above clause (5) in program $\mathbf{P}^*$. The new clauses obtained are:

$$\mathbf{p}_1(\mathbf{Y}) \leftarrow \mathtt{next}_i\ \mathbf{q}_1(\mathbf{Y}). \tag{6}$$

$$\mathtt{next}_i\ \mathbf{q}_0(\mathbf{X}) \leftarrow \mathbf{p}_0(\mathbf{X}). \tag{7}$$

Assume also that $R\ \mathbf{p}_1(\mathbf{b})$ has been introduced in $T_{\mathbf{P}^*} \uparrow (k+1)$ by clause (6) above. Then, this means that $R\ \mathtt{next}_i\ \mathbf{q}_1(\mathbf{b}) \in T_{\mathbf{P}^*} \uparrow k$. By the induction hypothesis, we get that there exists a constant $\mathbf{a}$ such that $R\ \mathtt{next}_i\ \mathbf{q}_0(\mathbf{a}) \in T_{\mathbf{P}^*} \uparrow k$ and $\mathbf{q}(\mathbf{a}, \mathbf{b}) \in T_{\mathbf{P}} \uparrow \omega$. Using clause (5), we get that $\mathbf{p}(\mathbf{a}, \mathbf{b}) \in T_{\mathbf{P}} \uparrow \omega$.

Using clause (7) above together with the fact that $R\ \mathtt{next}_i\ \mathbf{q}_0(\mathbf{a}) \in T_{\mathbf{P}^*} \uparrow k$, we get[2] $R\ \mathbf{p}_0(\mathbf{a}) \in T_{\mathbf{P}^*} \uparrow (k-1)$, which implies that $R\ \mathbf{p}_0(\mathbf{a}) \in T_{\mathbf{P}^*} \uparrow k$.

Case 3: Assume that in $\mathbf{P}$ there exists a fact of the form:

$$\mathbf{p}(\mathbf{a}, \mathbf{b}). \tag{8}$$

Consider now the transformation of the above clause (8) in program $\mathbf{P}^*$. The new clause obtained is:

$$\mathbf{p}_1(\mathbf{b}) \leftarrow \mathbf{p}_0(\mathbf{a}). \tag{9}$$

Assume now that $R\ \mathbf{p}_1(\mathbf{b})$ has been introduced in $T_{\mathbf{P}^*} \uparrow (k+1)$ by clause (9) above. This means that $R\ \mathbf{p}_0(\mathbf{a}) \in T_{\mathbf{P}^*} \uparrow k$ and therefore $R\ \mathbf{p}_0(\mathbf{a}) \in T_{\mathbf{P}^*} \uparrow (k+1)$. Moreover, $\mathbf{p}(\mathbf{a}, \mathbf{b}) \in T_{\mathbf{P}} \uparrow \omega$, because $\mathbf{p}(\mathbf{a},\mathbf{b})$ is a fact in $\mathbf{P}$.

[1] It can be easily seen that Case 1 of the induction step is only applicable for values of k which are greater than 2.

[2] It can be easily seen that Case 2 of the induction step is only applicable for values of k which are greater than 1.

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

Lemma 5.4 *Let* $\mathbf{P}$ *be a simple Chain Datalog program and* $\leftarrow \mathbf{p(a,X)}$ *be a goal clause. Let* $\mathbf{P}^*$ *be the Branching Datalog program obtained by applying the tranformation algorithm to* $\mathbf{P} \cup \{\leftarrow \mathbf{p(a,X)}\}$. *If* $\texttt{first}\ \mathtt{p}_1(\mathbf{b}) \in T_{\mathbf{P}^*} \uparrow \omega$ *then* $\mathbf{p(a,b)} \in T_{\mathbf{P}} \uparrow \omega$.

Proof: From Lemma 5.3 we have that there is a constant $\mathbf{c}$ such that $\mathbf{p(c,b)} \in \mathrm{T}_{\mathbf{P}} \uparrow \omega$ and $\texttt{first}\ \mathtt{p}_0(\mathbf{c}) \in \mathrm{T}_{\mathbf{P}^*} \uparrow \omega$. But as the only instance of $\texttt{first}\ \mathtt{p}_0(\mathbf{X})$ in $\mathrm{T}_{\mathbf{P}^*} \uparrow \omega$ is $\texttt{first}\ \mathtt{p}_0(\mathbf{a})$ then $\mathbf{c} = \mathbf{a}$. □

Theorem 5.1 *Let* $\mathbf{P}$ *be a simple Chain Datalog program and* $\leftarrow \mathbf{p(a,X)}$ *be a goal clause. Let* $\mathbf{P}^*$ *be the Branching Datalog program obtained by applying the tranformation algorithm to* $\mathbf{P} \cup \{\leftarrow \mathbf{p(a,X)}\}$. *Then* $\texttt{first}\ \mathtt{p}_1(\mathbf{b}) \in T_{\mathbf{P}^*} \uparrow \omega$ *iff* $\mathbf{p(a,b)} \in T_{\mathbf{P}} \uparrow \omega$.

Proof: It is an immediate consequence of lemmas 5.2 and 5.4. □

6 Evaluation Strategies

In this section we examine two different evaluation strategies that are applicable to the Branching Datalog programs that result from the transformation.

6.1 Bottom-up Evaluation

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

Branching Datalog programs can also be evaluated bottom-up through the use of the $T_{\mathbf{P}}$ operator of Definition 3.4. 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 the calculation of the answers to the goal clause is bounded by a number which depends on certain characteristics of the program (e.g. the number of unit clauses in the database of $\mathbf{P}$, the number of different data constants in the database, etc.). In order to prove this claim we use the results of [3] which refer to the language $\mathrm{Datalog}_{nS}$.

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

Example 6.1. Consider the following Chain Datalog program:

$$\begin{aligned} &\texttt{p(X,Y)} \leftarrow \texttt{q(X,Y)}. \\ &\texttt{q(a,b)}. \end{aligned}$$

and the goal clause

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

The output of the branching time transformation is:

$$\begin{aligned} &\leftarrow \texttt{first}\ \texttt{p}_1\texttt{(Y)}. \\ &\texttt{first}\ \texttt{p}_0\texttt{(a)}. \\ &\texttt{p}_1\texttt{(Y)} \leftarrow \texttt{next}_1\ \texttt{q}_1\texttt{(Y)}. \\ &\texttt{next}_1\ \texttt{q}_0\texttt{(X)} \leftarrow \texttt{p}_0\texttt{(X)}. \\ &\texttt{q}_1\texttt{(b)} \leftarrow \texttt{q}_0\texttt{(a)}. \end{aligned}$$

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

$$\begin{aligned} &\leftarrow \texttt{p}_1\texttt{([],Y)}. \\ &\texttt{p}_0\texttt{([],a)}. \\ &\texttt{p}_1\texttt{(T,Y)} \leftarrow \texttt{q}_1\texttt{([1|T],Y)}. \\ &\texttt{q}_0\texttt{([1|T],X)} \leftarrow \texttt{p}_0\texttt{(T,X)}. \\ &\texttt{q}_1\texttt{(T,b)} \leftarrow \texttt{q}_0\texttt{(T,a)}. \end{aligned}$$

□

The following lemma demonstrates the equivalence between the source Branching Datalog program and the corresponding Datalog_{nS} program that results from the above transformation.

Lemma 6.1 *Let* $\mathbf{P}^*$ *be a Branching Datalog program that results from the branching-time transformation. Let* $\mathbf{P}^*_{nS}$ *be the* Datalog_{nS} *program that results from the above transformation. Then, for all* $k \in \mathcal{N}$,

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

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

Using the results in [3], the following theorem can then be established:

Theorem 6.1 *Let* $\mathbf{P}$ *be a simple Chain Datalog program,* $\leftarrow \mathbf{p(a, X)}$ *be a goal clause, and* $\mathbf{P}^*$ *be the Branching Datalog program obtained by applying the branching-time transformation algorithm to* $\mathbf{P} \cup \{\leftarrow \mathbf{p(a, X)}\}$. *Then there is a natural number* k *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 [3], 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 6.1 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 suggests that one way to implement the programs that result from the transformation is through bottom-up evaluation (which can stop as soon as the bound of Theorem 6.1 is attained). In the present paper we do not consider complexity issues regarding bottom-up evaluation.

6.2 Top-down Evaluation

Branching Datalog programs can also be executed using a resolution-type proof procedure called *BSLD-resolution* (**B**ranching-time **SLD**-resolution), which was initially proposed for the more general category of Cactus programs [15]. BSLD-resolution is a refutation procedure which extends SLD-resolution [9], and is similar to TiSLD-resolution [12], the proof procedure for Chronolog programs.

It should be noted here that the programs that result from the branching-time transformation are much simpler in structure than general Cactus programs, because clauses contain at most one atom in their bodies. Therefore, BSLD-resolution becomes much simpler for the case of programs that result from the transformation. This simpler form of BSLD-resolution is defined below:

Definition 6.1. Let $\mathbf{P}^*$ be a Branching Datalog program that results from the branching-time transformation algorithm and $\mathbf{G}$ be the corresponding goal clause. A *BSLD-derivation* of $\mathbf{P}^* \cup \{\mathbf{G}\}$ consists of a (possibly infinite) sequence of canonical temporal goals $\mathbf{G}_0 = \mathbf{G}, \mathbf{G}_1, \ldots, \mathbf{G}_n, \ldots$, a sequence $\mathbf{C}_1, \ldots, \mathbf{C}_n, \ldots$ of canonical instances of clauses of $\mathbf{P}^*$ (called the *input clauses*), and a sequence $\theta_1, \ldots, \theta_n \ldots$ of most general unifiers such that for all i, the goal $\mathbf{G}_{i+1}$ is obtained from the goal:

$\mathbf{G}_i = \leftarrow \mathbf{A}$

as follows:

1. $\mathbf{A}$ is the canonical temporal atom in $\mathbf{G}_i$

2. $\mathbf{H} \leftarrow \mathbf{B}$ is the input clause $\mathbf{C}_{i+1}$ (standardized apart from $\mathbf{G}_i$),

3. $\theta_{i+1} = mgu(\mathbf{A}, \mathbf{H})$

4. $\mathbf{G}_{i+1}$ is the goal: $\mathbf{G}_{i+1} = \ \leftarrow \mathbf{B}\theta_{i+1}$

Definition 6.2. A *BSLD-refutation* of $\mathbf{P}^* \cup \{\mathbf{G}\}$ is a finite BSLD-derivation of $\mathbf{P}^* \cup \{\mathbf{G}\}$ which has the empty goal clause $\square$ as the last clause of the derivation.

Definition 6.3. A *computed answer* for $\mathbf{P}^* \cup \{\mathbf{G}\}$ is the substitution obtained by restricting the composition $\theta_1\theta_2 \ldots \theta_n$ to the variables of $\mathbf{G}$, where $\theta_1, \theta_2, \ldots, \theta_n$, is the sequence of the most general unifiers used in a BSLD-refutation of $\mathbf{P}^* \cup \{\mathbf{G}\}$.

BSLD-resolution is a sound and complete proof procedure for Branching Datalog (see the corresponding theorems for the language Cactus [15]).

Example 6.2. Consider the program that was also used in Example 6.1:

(1) `first` $\mathtt{p}_0(\mathtt{a})$.

(2) $\mathtt{p}_1(\mathtt{Y}) \leftarrow \mathtt{next}_1\ \mathtt{q}_1(\mathtt{Y})$.

(3) $\mathtt{next}_1\ \mathtt{q}_0(\mathtt{X}) \leftarrow \mathtt{p}_0(\mathtt{X})$.

(4) $\mathtt{q}_1(\mathtt{b}) \leftarrow \mathtt{q}_0(\mathtt{a})$.

A BSLD-refutation of the canonical goal $\leftarrow$ `first` $\mathtt{p}_1(\mathtt{Y})$ is given below:

$\leftarrow$ `first` $\mathtt{p}_1(\mathtt{Y})$
using clause (2)
$\leftarrow$ `first` $\mathtt{next}_1\ \mathtt{q}_1(\mathtt{Y})$
using clause (4) (and `Y = b`)
$\leftarrow$ `first` $\mathtt{next}_1\ \mathtt{q}_0(\mathtt{a})$
using clause (3) (and `X = a`)
$\leftarrow$ `first` $\mathtt{p}_0(\mathtt{a})$
using clause (1)
$\square$

$\square$

Further evaluation related topics are outside the scope of the present paper. In the next section we indicate certain directions that seem promising for further research.

7 Discussion

In this paper, we have developed a transformation algorithm from Chain Datalog programs to Branching Datalog ones. The programs obtained by this transformation have the following interesting properties:

- All predicates are unary
- Every rule has at most one 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. In the previous section we have presented two different approaches for the execution of the programs that result from the transformation algorithm. The first approach is a bottom-up evaluation strategy which is guaranteed to produce all the answers to a given goal clause in a finite number of steps. The second approach is more closely connected to the usual resolution-based proof procedures for logic programming languages. It is outside the scope of the present paper to consider the performance comparison among the two approaches. We believe however that such a question deserves further research. Another 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 to full Datalog: The authors have attempted to extend the proposed transformation to Datalog programs that are not in chain form. Clearly, the transformation does not extend directly to general Datalog. We believe however that, although not in a straightforward manner, the algorithm can be generalized to apply to full Datalog. We are currently investigating such a possibility.

Appendix A.1. Retaining the EDB predicates of P

As we have mentioned in section 4, it is customary in the database community to leave unchanged the EDB part of a database when performing a transformation. The transformation algorithm that we present in section 4 can be easily modified so as that it leaves the EDB intact. The only change concerns the first rule of the algorithm which now becomes:

1′ (a) Add all unit clauses of $\mathbf{P}$ to $\mathbf{P}^*$.

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

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

It is easy to see now that the properties of $\mathbf{P}^*$ mentioned in section 4, i.e. the property that all atoms in $\mathbf{P}^*$ are unary, and the property that all program clauses of $\mathbf{P}^*$ have at most one atom in their bodies, hold for all program clauses of $\mathbf{P}^*$ except the clauses introduced by applying the rule (1'). In fact, the clauses introduced by this rule in (B) play the role of an interface to the program database, which is now retained by the transformation in its initial form.

References

[1] F. Afrati and C. H. Papadimitriou. The parallel complexity of simple chain queries. In *Proc. 6th ACM Symposium on Principles of Database Systems*, pages 210–213, 1987.

[2] F. Afrati and C. H. Papadimitriou. Parallel complexity of simple logic programs. *Journal of the ACM*, 40(3):891–916, 1993.

[3] Jan Chomicki. Depth-bounded bottom-up evaluation of logic programs. *J. of Logic Programming*, 25(1):1–31, 1995.

[4] G. Dong and S. Ginsburg. On decompositions of chain datalog programs into $\mathcal{P}$ (left-)linear 1-rule components. *J. of Logic Programming*, 23(3):203–236, 1995.

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

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

[7] M. Gergatsoulis and M. Katzouraki. Unfold/fold transformations for definite clause programs. In M. Hermenegildo and J. Penjam, editors, *Programming Language Implementation and Logic Programming (PLILP'94), Proceedings*, Lecture Notes in Computer Science (LNCS) 844, pages 340–354. Springer-Verlag, 1994.

[8] S. L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice-Hall, 1987.

[9] J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 1987.

[10] M. A. Orgun. *Intensional logic programming.* PhD thesis, Dept. of Computer Science, University of Victoria, Canada, December 1991.

[11] M. A. Orgun and W. W. Wadge. Towards a unified theory of intensional logic programming. *The Journal of Logic Programming*, 13(4):113–145, August 1992.

[12] M. A. Orgun and W. W. Wadge. Chronolog admits a complete proof procedure. In *Proc. of the Sixth International Symposium on Lucid and Intensional Programming (ISLIP'93)*, pages 120–135, 1993.

[13] P. Rondogiannis. *Higher-order functional languages and intensional logic.* PhD thesis, Dept. of Computer Science, University of Victoria, Canada, December 1994.

[14] P. Rondogiannis, M. Gergatsoulis, and T. Panayiotopoulos. *Cactus*: A branching-time logic programming language. In *Proc. of the First International Joint Conference on Qualitative and Quantitative Practical Reasoning, ECSQARU-FAPR'97, Bad Honnef, Germany*, Lecture Notes in Artificial Intelligence (LNAI) 1244, pages 511–524. Springer, June 1997.

[15] P. Rondogiannis, M. Gergatsoulis, and T. Panayiotopoulos. Branching-time logic programming: The language Cactus and its applications. *Computer Languages*, 24(3):155–178, October 1998.

[16] P. Rondogiannis and W. W. Wadge. First-order functional languages and intensional logic. *Journal of Functional Programming*, 7(1):73–101, 1997.

[17] P. Rondogiannis and W. W. Wadge. Higher-Order Functional Languages and Intensional Logic. *Journal of Functional Programming*, 1999. (to appear).

[18] H. Tamaki and T. Sato. Unfold/fold transformations of logic programs. In *Second International Conference on Logic Programming*, pages 127–138, 1984.

[19] Jeffrey D. Ullman and Allen Van Gelder. Parallel complexity of logical query programs. *Algorithmica*, 3:5–42, 1988.

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

[21] A. Yaghi. *The intensional implementation technique for functional languages.* PhD thesis, Dept. of Computer Science, University of Warwick, Coventry, UK, 1984.