Fundamenta Informaticae 71 (2006) 1–43
IOS Press

A Value-propagating Transformation Technique for Datalog Programs Based on Non-Deterministic Constructs

Petros Potikas
Computer Science Division, Department of Electrical and Computer Engineering
National Technical University of Athens, 157 73 Zografou, Athens, Greece
ppotik@cs.ntua.gr

Panos Rondogiannis
Department of Informatics & Telecommunications,
University of Athens, Panepistimiopolis, 157 84 Athens, Greece
prondo@di.uoa.gr

Manolis Gergatsoulis[*]
Department of Archive and Library Sciences,
Ionian University, Palea Anaktora, Plateia Eleftherias, 49100 Corfu, Greece
manolis@ionio.gr

Abstract. The branching-time transformation is a recent technique for optimizing Chain Datalog programs. In this paper we propose a significant extension of the branching-time transformation which we believe opens up a promising new direction of research in the area of value-propagating Datalog optimizations. More specifically, the proposed transformation can handle more general programs that allow multiple consumptive occurrences of variables in the bodies of clauses. This extension is achieved by using as target language the temporal logic programming formalism Datalog$_{nS}$ enriched with choice predicates (a non-deterministic construct that was originally introduced in the area of intensional logic programming). We demonstrate the correctness of the transformation and propose several optimizations that can be applied to the target code. Moreover, we define a bottom-up proof procedure that applies to the target programs and demonstrate that it always terminates (despite the fact that the Herbrand base of these programs is generally infinite).

[*]Address for correspondence: Department of Archive and Library Sciences, Ionian University, Palea Anaktora, Plateia Eleftherias, 49100 Corfu, Greece

Keywords: Logic Programming, Deductive Databases, Program and Query Transformations, Non-deterministic Constructs.

1. Introduction

The *branching-time transformation* [18, 19] is a recent transformation technique that applies to Chain Datalog programs. More specifically, the branching-time approach belongs to the class of *value-propagating* Datalog optimizations in which the input values of the top level goal of the source program are propagated in order to restrict the generation of atoms in the bottom-up computation. Such techniques include the *counting transformation* [24], the *magic sets* [3, 25], the *pushdown approach* [9], and so on. The branching-time transformation was inspired by a similar technique that has been proposed for functional programming languages [30, 29, 21, 22].

In this paper, we extend the branching-time approach so as that it can handle a significantly broader class of well-moded Datalog programs. One of the novel characteristics of our new approach is that the target language is Datalog$_{nS}$ [5, 4] extended with choice predicates [14], a non-deterministic construct that was originally introduced in intensional logic programming [13]. The use of choice predicates allows the transformation of clauses containing multiple consumptive occurrences of variables. We believe that the use of non-deterministic constructs opens up a promising direction of research in the area of Datalog optimizations. For the programs that result from the transformation we define a bottom-up proof procedure and we demonstrate that it always terminates (despite the fact that the Herbrand base of the programs of the target language can be infinite). Finally, we define several optimizations on the target code, which enhance the performance of the bottom-up computation. The main contributions of the paper can therefore be summarized as follows:

- We propose a new value-propagating transformation technique for a large class of moded Datalog programs and demonstrate its correctness. Actually, the class of programs that we consider is broader than those considered by other related transformations.
- We demonstrate that temporal languages such as Datalog$_{nS}$ extended with non-deterministic constructs can prove especially useful for defining new powerful transformations for Datalog programs (and possibly for more general logic programs). In particular, we demonstrate that multiple consumptions of variables in Datalog programs can be treated effectively using choice predicates.
- We define a proof procedure that applies to the target programs of the transformation, and demonstrate that it always terminates. Moreover, we propose several optimizations of the target code that enhance its efficiency.

Some of the results outlined above appeared in preliminary form as a conference paper [16].

The rest of this paper is organized as follows: Section 2 gives an outline of the proposed transformation technique. Sections 3 and 4 introduce the source and the target languages of the transformation, while Section 5 introduces the transformation algorithm itself. Section 6 provides the correctness proof of the transformation. Section 7 introduces a terminating proof procedure for the programs that result from the transformation and Section 8 presents optimizations of the target code. Section 9 evaluates the proposed technique with respect to other related transformations, and Section 10 discusses possible future extensions.

2. An Outline of the Technique

In the area of deductive databases, Datalog and bottom-up fixpoint computation are favored. The effectiveness of bottom-up execution for Datalog programs is based on optimization techniques, often referred to as *query optimization techniques* [17]. A query optimization is a transformation of a query (program and goal) to a new query, that is semantically equivalent to the initial one, in a form suitable for more efficient bottom-up evaluation. A known family of query optimizations is the *value-propagating techniques*, that treat queries in which goals have some bound arguments.

The branching-time transformation was recently introduced by two of the authors in the area of value-propagating Datalog optimizations, and applies (in its initial form) to the class of Chain Datalog programs [18, 19]. The name "branching-time" is due to the fact that the recursive predicate calls in a program form a tree-like structure which can be captured using a branching-time language. The branching-time transformation is applied on all clauses of the initial program, and for each one of them produces a set of new clauses (each one of which contains at most one IDB predicate in its body). Intuitively, the resulting clauses reflect in a more direct way the flow of the argument values that takes place when the initial program is executed.

In the branching-time transformation, for every predicate in the initial program two new predicates are introduced; each one of them has two arguments, a control one (in the form of a list of natural numbers) and a data one which encodes the argument being passed. The intuition behind the control argument is that it "links" the two new predicates and coordinates them so as that the correct answers will be produced.

To illustrate the branching-time transformation in its initial form, consider the following Chain Datalog program (in which p is an IDB predicate while e an EDB one):

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

In the class of Chain Datalog programs, the first argument of each predicate is considered as an input one while the second as an output (this is due to the fact that we consider goal atoms with their first argument bound). The new predicates $\texttt{p}_1^+$ and $\texttt{p}^-$ introduced by the transformation for a program predicate p, correspond to the calls (inputs) and the answers (outputs) respectively, for the predicate p in the top-down computation.

We demonstrate the transformation by considering in turn each clause of the initial program. The transformation of the goal clause results in:

$$\begin{aligned}&\leftarrow \texttt{p}^-\texttt{([],Y)}.\\ &\texttt{p}_1^+\texttt{([],a)}.\end{aligned}$$

Notice that the bound argument of the initial goal clause has become an argument of a unit clause in the transformed program. As a result, the bottom-up evaluation of the resulting program would use this unit clause in its first step in order to restrict the set of atoms produced in subsequent steps.

The transformation of the first clause of the source program results in:

$$\begin{aligned}&\texttt{p}^-\texttt{(L,Z)} \leftarrow \texttt{e}^-\texttt{([1|L],Z)}.\\ &\texttt{e}_1^+\texttt{([1|L],X)} \leftarrow \texttt{p}_1^+\texttt{(L,X)}.\end{aligned}$$

Notice the label 1 that appears in both $\texttt{e}^{-}\texttt{([1|L],Z)}$ and $\texttt{e}_1^{+}\texttt{([1|L],X)}$. The basic idea is that this label relates the two atoms that have resulted from the same call in the initial program (namely the call e(X,Z)). It is important to note that *any* label can be used instead of 1, as long as this label is different from the ones that are assigned to other calls of the initial program.

The second clause of the initial program is transformed as follows:

$$\begin{aligned}&\texttt{p}^{-}\texttt{(L,Z)} \leftarrow \texttt{p}^{-}\texttt{([3|L],Z)}.\\&\texttt{p}_1^{+}\texttt{([3|L],Y)} \leftarrow \texttt{p}^{-}\texttt{([2|L],Y)}.\\&\texttt{p}_1^{+}\texttt{([2|L],X)} \leftarrow \texttt{p}_1^{+}\texttt{(L,X)}.\end{aligned}$$

Finally, the branching-time transformation also introduces the clause:

$$\texttt{e}^{-}\texttt{(L,Y)} \leftarrow \texttt{e(X,Y)},\texttt{e}_1^{+}\texttt{(L,X)}.$$

which plays the role of an interface to the database atoms whose predicate is e.

Notice that the program obtained by the transformation is not a Datalog one. In fact, it is a Datalog$_{nS}$ program [5, 4]. We should also note that in the original papers defining the branching-time transformation [18, 19], sequences of temporal operators are used instead of lists of natural numbers, and the resulting program is a *Branching Datalog* one (Branching Datalog is the function-free subset of the branching-time logic programming language *Cactus* [20]). It is however easy to show that the two approaches are equivalent. Notice also that the Herbrand base of the programs resulting from the transformation is not finite due to the lists that have been introduced. However, as we have demonstrated in [19] (based on the results in [4]) there exists a terminating bottom-up computation that produces all the answers to the goal clause. For a more detailed description of the branching-time transformation, the interested reader should consult [19, 18].

However, the branching-time technique in the form described above, does not apply to Datalog programs in which there exist multiple consumptions of variables. The following example demonstrates the issues that arise in such a case. Consider a Datalog program that contains the clause:

$$\texttt{p(X,Z)} \leftarrow \texttt{q(X,W)},\texttt{r(X,W,Z)}.$$

The problem with the above clause arises from the fact that X appears twice in the body of the clause. When attempting to apply the branching-time technique to the above program, the relationship between the two different occurrences of X is lost, and the resulting program is no longer semantically equivalent to the initial one. More specifically, a naive translation of the above clause would produce (among others) the two following clauses:

$$\begin{aligned}&\texttt{q}_1^{+}\texttt{([l1|L],X)} \leftarrow \texttt{p}_1^{+}\texttt{(L,X)}.\\&\texttt{r}_1^{+}\texttt{([l2|L],X)} \leftarrow \texttt{p}_1^{+}\texttt{(L,X)}.\end{aligned}$$

where l1 and l2 are natural numbers. Notice that the two occurrences of X in the body of the initial clause have been separated from each other as they have been placed in different clauses in the target program, and it is therefore possible for them to instantiate to different values (something which was not the case in the original clause). In this way the resulting program may produce answers which are not included in the set of answers of the initial program.

In this paper, we propose a solution to the above problem based on *choice predicates* [14], a non-deterministic construct that has been proposed in the area of *intensional logic programming* [13] (similar non-deterministic constructs have also been considered in other forms in [7, 8]). Choice predicates are declarative in nature and have a well-defined and elegant semantics [14]. The basic idea behind choice predicates is that under a given context (represented by the list L in the above example), a predicate can only be true of a unique value. Therefore, the above two clauses can be instead written as:

$$\texttt{q}_1^+\texttt{([l1|L],X)} \leftarrow \texttt{\#p}_1^+\texttt{(L,X)}.$$
$$\texttt{r}_1^+\texttt{([l2|L],X)} \leftarrow \texttt{\#p}_1^+\texttt{(L,X)}.$$

where $\texttt{\#p}_1^+$ is the choice version of the predicate $\texttt{p}_1^+$, which at any given L can be true of only one value X. This restores the connection between the two occurrences of X, resulting in a target program equivalent to the source one.

3. The Source Language of the Transformation

In the following, we assume familiarity with the basic notions of logic programming [11]. A finite set D of ground facts (or unit clauses) without function symbols is often referred as an *extensional database* or simply a *database*. The predicates of the atoms in a database are called *EDB predicates*. A *Datalog program* P consists of a finite set of clauses without function symbols. Predicates that appear in the heads of clauses of P are called *intensional* or *IDB predicates* (IDBs). We assume that EDB predicates do not appear in the head of program clauses; moreover, we assume that predicates appearing only in the bodies of the clauses of a Datalog program are EDB predicates. A Datalog program P together with a database D is denoted by P_D.

In the rest of this paper we adopt the following notation: *constants* are denoted by lower case letters such as a, b, c and vectors of constants by $\vec{e}$; *variables* by uppercase letters such as X, Y, Z and vectors of variables by $\vec{v}$; predicates by p, q, r, s; also subscripted versions of the above symbols will be used.

The class of programs on which the proposed transformation applies is a subclass of Datalog:

Definition 3.1. A clause
$$p_0(\vec{v}_0, Z_n) \leftarrow p_1(\vec{v}_1, Z_1), p_2(\vec{v}_2, Z_2), \ldots, p_n(\vec{v}_n, Z_n).$$
with $n > 0$, is called *consecutive consumption clause* (or *cc-clause* for short) if:

1. Each $\vec{v}_i$, for $i = 0, \ldots, n$ is a nonempty vector of distinct variables, and $Z_1, \ldots, Z_n$ are distinct variables.
2. $vars(\vec{v}_0) = vars(\vec{v}_1)$ and $vars(\vec{v}_i) = \{Z_{i-1}\} \cup u_{i-1}$, for $i = 2, \ldots, n$, where $u_{i-1} \subseteq vars(\vec{v}_{i-1})$.
3. $Z_i \notin \bigcup_{j \le i} vars(\vec{v}_j)$, for $i = 1, \ldots, n$.

A program P is said to be a *consecutive consumption Datalog program* (or *cc-Datalog program*) if all its clauses are cc-clauses. A *goal* G is of the form $\leftarrow q(\vec{e}, Z)$, where $\vec{e}$ is a nonempty vector of constants, Z is a variable and q is an IDB predicate.

It should be mentioned here that cc-clauses are moded; the terms $\vec{v}_i$ of the above definition correspond to input arguments while each Z_i corresponds to the single output argument of each atom. An occurrence of a variable in an input argument of the head or in the output argument of an atom in the body will be called *productive*; otherwise it will be called *consumptive*.

Example 3.1. The following program is a cc-Datalog program:

$$\begin{aligned}&\mathtt{q}(\overset{+}{\mathtt{X}},\overset{-}{\mathtt{Z}}) \leftarrow \mathtt{f}(\overset{+}{\mathtt{X}},\overset{-}{\mathtt{Z}}).\\ &\mathtt{q}(\overset{+}{\mathtt{X}},\overset{-}{\mathtt{Z}}) \leftarrow \mathtt{e}(\overset{+}{\mathtt{X}},\overset{-}{\mathtt{Y}}),\mathtt{q}(\overset{+}{\mathtt{Y}},\overset{-}{\mathtt{W}}),\mathtt{g}(\overset{+}{\mathtt{Y}},\overset{+}{\mathtt{W}},\overset{-}{\mathtt{Z}}).\end{aligned}$$

where the $+$ and $-$ signs above the variables denote the input and output arguments respectively.

The intuition behind the class of cc-Datalog programs is that each value produced by an atom can only be consumed in a sequence of (one or more) consecutive atoms immediately following the atom that produced it. Many natural Datalog programs belong to this class; for example, the class of Chain Datalog programs is a proper subset of this class.

Example 3.2. Consider the following generalized `path` predicate which searches for paths of a given color in a graph whose edges are colored:

```
path(X,Color,Z) ← edge(X,Color,Z).
path(X,Color,Z) ← edge(X,Color,W),path(W,Color,Z).
```

The above is clearly a cc-Datalog program. The EDB predicate `edge(X,Color,Z)` signifies that the edge (`X`,`Z`) has color `Color`. The goal ← `path(a,red,Z)` asks for those vertices of the graph that are reachable from vertex `a` through `red` edges.

Example 3.3. The usual logic programming definitions of the modulo and the greatest common divisor operations involve cc-Datalog clauses:

```
mod(X,Y,Z) ← mod_base(X,Y,Z).
mod(X,Y,Z) ← minus(X,Y,W),mod(W,Y,Z).
gcd(X,Y,Z) ← gcd_base(X,Y,Z).
gcd(X,Y,G) ← mod(X,Y,Z),gcd(Y,Z,G).
```

where `mod_base`, `gcd_base`, and `minus` are EDB predicates defined as follows: `mod_base(X,Y,Z)` if `X < Y` and `X = Z`; `gcd_base(X,Y,Z)` if `X = Z` and `Y = 0`; and `minus(X,Y,Z)` if `Z = X - Y`.

The semantics of cc-Datalog programs can be defined in accordance to the semantics of classical logic programming. The notions of *minimum model* M_{P_D} of P_D, where P_D is a cc-Datalog program P together with a database D, and *immediate consequence operator* T_{P_D}, transfer directly [11].

We now define a subclass of cc-Datalog programs which has the same power as the full class of cc-Datalog programs.

Definition 3.2. A *simple cc-Datalog program* is a cc-Datalog program in which every clause has at most two atoms in its body.

The following proposition (which can be proved using unfold/fold transformations [26, 6, 15]) establishes the equivalence between cc-Datalog programs and simple cc-Datalog ones. Notice that by $M(p, P_D)$ we denote the set of atoms in the minimum model of P_D whose predicate symbol is p.

Proposition 3.1. Every cc-Datalog program P can be transformed into a simple cc-Datalog program P' such that for every predicate symbol p appearing in P and for every database D, $M(p, P_D) = M(p, P'_D)$.

Proof:
(Outline) For a cc-clause C of the form:

$$A_0 \leftarrow A_1, A_2, \dots, A_n$$

with $n > 2$, we introduce a new clause C_N of the form:

$$new(\vec{v}, Z) \leftarrow A_2, \dots, A_n$$

where Z is the output variable of A_n and $\vec{v}$ is a vector of distinct variables such that $vars(\vec{v}) = (vars(A_2, \dots, A_n) \cap vars(A_0, A_1)) - \{Z\}$. Then we fold C using C_N and we get the clause C':

$$A_0 \leftarrow A_1, new(\vec{v}, Z)$$

C is now replaced by $\{C_N, C'\}$. Then we apply the same process to C_N which has $n - 1$ body atoms until all clauses have at most two body atoms. It is easy to verify that the program obtained by this process is a cc-Datalog program. □

In the presentation of the proposed transformation we use simple cc-Datalog programs as the source language. Because of the above proposition this is not a restriction of the power of the algorithm. Moreover, the transformation could be easily formulated so as to apply directly to cc-clauses with more than two body atoms (but this would imply a more complicated presentation and correctness proof).

Example 3.4. The cc-Datalog program in Example 3.1 is not a simple cc-Datalog one since its second clause has three atoms in its body. To transform it into a simple cc-Datalog program we introduce the new clause:

```
p(Y,Z) ← q(Y,W),g(Y,W,Z).
```

and we use it to fold the second clause of the program. In this way we get the following simple cc-Datalog program:

```
q(X,Z) ← f(X,Z).
q(X,Z) ← e(X,Y),p(Y,Z).
p(Y,Z) ← q(Y,W),g(Y,W,Z).
```

4. The Target Language of the Transformation

The target language of the transformation is the language Choice Datalog$_{nS}$ which is a version of Datalog$_{nS}$ [5, 4] extended with choice predicates [14]. Datalog$_{nS}$ is a powerful temporal deductive database language and choice predicates are non-deterministic constructs useful for temporal languages.

4.1. Choice Predicates

Choice predicates [14] were initially introduced in the area of temporal logic programming as a means for ensuring that a given predicate is single-valued at a particular moment in time (or more generally at a particular context). Actually, with every predicate symbol p of a given program, a predicate $\#p$ (called the *choice version* of p), is associated. Choice predicates can only appear in the bodies of program clauses (their axiomatization is implicit, see [14] for details).

To motivate choice predicates consider writing a program whose purpose is to assign a classroom to persons (teachers) over different moments in time. The problem is to find all different such assignments in such a way that at every different moment only a single person occupies a classroom. The predicate `requests_class(Time,Person)` expresses the fact that `Person` requests the particular classroom at time `Time`. The predicate `uses_class(Time,Person)` expresses the fact that `Person` *actually* uses the classroom at time `Time`.

```
requests_class(0,tom).
requests_class(0,nick).
requests_class(1,mary).
uses_class(Time,Person) ← #requests_class(Time,Person).
```

In the above program, `#requests_class` is the choice predicate that corresponds to the (classical) predicate `requests_class`. The crucial property of a choice predicate is that it is single-valued for any given time-point. This means that either the atom `#requests_class(0,tom)` or `#requests_class(0,nick)` but not both, will be considered as true at time-point `0`. Therefore, the above program does not have a unique minimum model (as is the case in classical logic programming); instead, it has a set of minimal models, one for every different possible (functional) assignment of persons over the moments in time. More specifically, the two minimal models of the program are the following:

$M_1 = \{$`#uses_class(0,tom),#uses_class(1,mary), uses_class(0,tom),`
`uses_class(1,mary), #requests_class(0,tom),#requests_class(1,mary)`
`requests_class(0,tom),requests_class(0,nick),requests_class(1,mary)`$\}$

and

$M_2 = \{$`#uses_class(0,nick),#uses_class(1,mary), uses_class(0,nick),`
`uses_class(1,mary), #requests_class(0,nick),#requests_class(1,mary)`
`requests_class(0,tom),requests_class(0,nick),requests_class(1,mary)`$\}$

Choice predicates are not necessarily restricted to apply to simple temporal logic programming languages such as the one used in the above example (in which time is linear); they are also applicable to more general intensional programming languages [13] that include the language Datalog$_{nS}$ which we have adopted for defining the proposed transformation. More specifically, in [14], Orgun and Wadge develop a general semantic framework for choice predicates that can apply to a wide class of intensional logic programming languages.

4.2. Syntax of Choice Datalog$_{nS}$

The target language of the transformation is a temporal deductive database language that supports choice predicates. We will refer to this target language as *Choice Datalog*$_{nS}$ since it is a variant of the language Datalog$_{nS}$ [4], augmented with choice predicates [14].

Datalog$_{nS}$ is a temporal extension of Datalog in which atoms may have a single distinguished *temporal* argument in addition to the usual *data arguments*. Time in Datalog$_{nS}$ may have a very rich structure (i.e., it is not necessarily linear). For our purposes we adopt a variant of Datalog$_{nS}$ in which time has a branching structure. It can easily be seen that branching-time can be modeled if we allow the temporal argument to range over lists of natural numbers[1]. We extend this variant of Datalog$_{nS}$ to support choice predicates. Before we proceed to a formal introduction of the syntax, we give an example of how a Choice Datalog$_{nS}$ clause looks like:

```
q(L,Z) ← g(X,Y,Z),#p([3|L],X),r([4,3|L],Y).
```

In the above example, `L`, `[3|L]` and `[4,3|L]` are terms that correspond to the distinguished temporal arguments of the atoms; `X`, `Y` and `Z` are usual data variables. The atoms `q(L,Z)`, `#p([3|L],X)` and `r([4,3|L],Y)` are IDB atoms while `g(X,Y,Z)` is an EDB atom. In other words, we assume that only IDB atoms have a temporal argument (while EDB atoms do not). The atom `#p([3|L],X)` is an example of a choice atom.

We now define formally the syntax of the target language. We assume the existence of a distinguished variable L which will be the only variable that can appear in temporal terms of a program; all other variables that appear in a program must be different from L. A temporal term is defined as follows:

Definition 4.1. A *temporal term* is recursively defined as follows:

$$tt ::= [\,] \mid L \mid [l|tt]$$

where L is a distinguished temporal variable and $l \in \omega$.

An *IDB atom* is an atom of the form $p(tt, t)$ or of the form $\#p(tt, t)$, where tt is a temporal term and t is an ordinary Datalog term (i.e., either a variable or a constant); the former atoms are called *non-choice* while the latter are called *choice atoms*. Notice that in the variant of Datalog$_{nS}$ that we consider, IDB atoms have only one data argument. An *EDB atom* is of the form $p(t_0, \dots, t_{n-1})$ where $t_0, \dots, t_{n-1}$ are ordinary Datalog terms. As usual, an atom is said to be ground, if it is variable-free. A *clause* in Choice Datalog$_{nS}$ is of the form:

$$H \leftarrow B_1, B_2, \dots, B_n.$$

where H is a non-choice IDB atom, and $B_1, B_2, \dots, B_n$ are atoms (either IDB or EDB ones) and $n \geq 0$. If $n = 0$, then the clause is said to be a *fact* or *unit clause*. A Choice Datalog$_{nS}$ program is a finite set of clauses of the aforementioned form. As usual, P_D is a Choice Datalog$_{nS}$ program P along with a database D. A *goal* in Choice Datalog$_{nS}$ is of the form $\leftarrow p([\,], X)$.

4.3. Semantics of Choice Datalog$_{nS}$

The Herbrand universe U_{P_D} of a Choice Datalog$_{nS}$ program P along with a database D includes all constants that appear in D or as data arguments in P. The Herbrand base B_{P_D} includes:

- ground EDB atoms constructed using constants from U_{P_D}

[1]In the original formulation of the branching-time transformation [19], sequences of temporal operators were used instead of lists. It can be easily shown that both formulations are equivalent.

- ground IDB atoms (either choice or non-choice) whose first argument is a list of natural numbers and their second argument is an element of U_{P_D}.

A Herbrand interpretation I of P_D is (as usual) a subset of its Herbrand base B_{P_D}. The semantics of Choice Datalog$_{nS}$ programs can be defined using the general principles developed in [14]. For reasons of completeness, we adapt the basic notions from [14] to fit our purposes. For a deeper exposition, the interested reader should consult the results of [14].

Before defining the notion of model we need to define a set of axioms establishing the relationship between predicates and choice predicates.

Definition 4.2. (Choice Formulas) Let P be a Choice Datalog$_{nS}$ program, and p be a predicate defined in P. Then the *choice formulas* associated with p are the following:

$$\forall L \forall X (\#p(L, X) \rightarrow p(L, X)).$$
$$\forall L \forall X (\#p(L, X) \rightarrow \forall Y (\#p(L, Y) \rightarrow X = Y)).$$

Definition 4.3. Let P be a Choice Datalog$_{nS}$ program. Then a *model* of P is an interpretation that satisfies the clauses of P together with the associated choice formulas for all predicates defined in P.

We can define a T_{P_D} operator in the usual way. However, we need something stronger than the usual T_{P_D} in order to define the semantics of Choice Datalog$_{nS}$ programs. We therefore define the operator NT_{P_D} that works exactly like T_{P_D} but does not throw away any choice atoms [14]. More formally:

Definition 4.4. Let P be a Choice Datalog$_{nS}$ program, D a database and I a Herbrand interpretation of P_D. Then $NT_{P_D}(I)$ is defined as follows:

$$NT_{P_D}(I) = T_{P_D}(I) \cup choices(I)$$

where $choices(I)$ is the set of the choice atoms belonging to I.

Furthermore, the semantics of Choice Datalog$_{nS}$ programs require another operator, namely the C_{P_D} operator, which returns all possible immediate extensions of a given Herbrand interpretation of P_D, determined by arbitrary choices (as will be further explained below).

Let I be a Herbrand interpretation of a Choice Datalog$_{nS}$ program P_D together with a database D and let p be a predicate symbol that appears in P_D. Define:

$$E_{p,L}(I) = \{a \mid p(L, a) \in I\}$$

and also:

$$E_{\#p,L}(I) = \{a \mid \#p(L, a) \in I\}$$

Also let:

$$S_I = \{\langle p, L, a\rangle \mid E_{p,L}(I) \neq \emptyset,\ E_{\#p,L}(I) = \emptyset \ and \ a \in E_{p,L}(I)\}$$

In other words, S_I is a set such that $\langle p, L, a\rangle \in S_I$, if no choice has been made for p at the context L and a is a possible candidate for this choice. The formal definition of C_{P_D} is given below:

Definition 4.5. Let P be a Choice Datalog$_{nS}$ program, D a database and I a Herbrand interpretation of P_D. Then $C_{P_D}(I)$ is defined as follows:

$$C_{P_D}(I) = \begin{cases} I & \textit{if } S_I = \emptyset \\ \{I \cup \{\#p(L,a)\} \mid \langle p, L, a\rangle \in S_I\} & \textit{if } S_I \neq \emptyset \end{cases}$$

Some observations can be made about the C_{P_D} operator. The first one is that C_{P_D} preserves the previously computed atoms (it does not reject anything). The next thing is that C_{P_D} when applied to an interpretation I, returns a set of interpretations each one corresponding to a different choice atom being added to I.

Our C_{P_D} operator is slightly different than the one in [14] since it only introduces one choice atom at each new interpretation it creates.

We now define the notion of P_D-chain which intuitively describes a bottom-up computation of a Choice Datalog$_{nS}$ program P together with a database D. During the bottom-up computation the operators $NT_{P_D} \uparrow \omega$ and C_{P_D} alternate as shown in the following definition:

Definition 4.6. Let P be a Choice Datalog$_{nS}$ program and D a database. A P_D-*chain* is a sequence $\langle M_0, M_1, M_2, \cdots\rangle$ satisfying the following conditions:

$$M_0 = \emptyset,$$

$$M_{2i+1} = NT_{P_D} \uparrow \omega(M_{2i}), \;\; i \geq 0,$$

and

$$M_{2i+2} \in C_{P_D}(M_{2i+1}), \;\; i \geq 0.$$

Notice that in the above definition by $NT_{P_D} \uparrow \omega(M_{2i})$ we denote as usual the set $\bigcup_{n\in\omega} NT^n_{P_D}(M_{2i})$.

The lemma stated below will be used in the proofs that follow.

Lemma 4.1. Let P be a Choice Datalog$_{nS}$ program, D a database and let $\langle M_0, M_1, \cdots\rangle$ be a P_D-chain. Then for all $i \in \omega$, $M_i \subseteq M_{i+1}$.

Proof:
An easy consequence of the definition of a P_D-chain. □

The least upper bound of a P_D-chain $\langle M_0, M_1, M_2, \cdots\rangle$ is $M = \bigcup_{i<\omega} M_i$. Then $M_0, M_1, \cdots$ will be called *approximations* of M and M is called a *limit interpretation* of P_D. Notice that a limit interpretation is not necessarily a fixpoint. However a limit interpretation is necessarily a model of the program as the following theorem indicates. Given two elements M_i and M_j of a P_D-chain, we will say that M_i is an *ancestor* of M_j if $i < j$. Based on the above definitions the following theorem is straightforward to establish:

Theorem 4.1. Let P be a Choice Datalog$_{nS}$ program and D a database. Then every limit interpretation of P_D is a model of P_D.

Proof:
The proof is similar to that in [14]. □

Example 4.1. Consider the following Choice Datalog$_{nS}$ program:

```
p([1|L],X) ← #q(L,X).
p([2|L],X) ← #q(L,X).
q([],a).
q([],b).
```

Applying the $NT_{P_D} \uparrow \omega$ and C_{P_D} operators as shown in Definition 4.6 we get two limit interpretations:

$$M = \{\#\mathtt{p}([1],\mathtt{a}), \#\mathtt{p}([2],\mathtt{a}), \mathtt{p}([1],\mathtt{a}), \mathtt{p}([2],\mathtt{a}), \#\mathtt{q}([\,],\mathtt{a}), \mathtt{q}([\,],\mathtt{a}), \mathtt{q}([\,],\mathtt{b})\}$$
$$M' = \{\#\mathtt{p}([1],\mathtt{b}), \#\mathtt{p}([2],\mathtt{b}), \mathtt{p}([1],\mathtt{b}), \mathtt{p}([2],\mathtt{b}), \#\mathtt{q}([\,],\mathtt{b}), \mathtt{q}([\,],\mathtt{a}), \mathtt{q}([\,],\mathtt{b})\}$$

It can be easily verified that these limit interpretations are models of the program. In the following we demonstrate the steps required for the construction of the model M:

$$\begin{array}{lcl}
M_0 & = & \emptyset \\
M_1 & = & \{\mathtt{q}([\,],\mathtt{a}), \mathtt{q}([\,],\mathtt{b})\} \\
M_2 & = & M_1 \cup \{\#\mathtt{q}([\,],\mathtt{a})\} \\
M_3 & = & M_2 \cup \{\mathtt{p}([1],\mathtt{a}), \mathtt{p}([2],\mathtt{a})\} \\
M_4 & = & M_3 \cup \{\#\mathtt{p}([1],\mathtt{a})\} \\
M_5 & = & M_4 \\
M & = & M_5 \cup \{\#\mathtt{p}([2],\mathtt{a})\}
\end{array}$$

The model M' is constructed in an entirely symmetric way.

5. The Transformation Algorithm

In this section we provide a formal definition of the transformation algorithm. The algorithm is subsequently illustrated by a representative example.

The algorithm: Let P be a given simple cc-Datalog program and G a given goal clause. For each $(n+1)$-ary predicate p in P, we introduce $n+1$ binary IDB predicates $p_1^+, \ldots, p_n^+$, p^-, where p_i^+ corresponds to the i-th input argument of p and p^- to the $(n+1)$-th argument of p (which is the output one). The choice versions of certain of these predicates may also be used in the target program. The transformation processes the goal clause G and each clause in P and gives as output a new goal clause G^* together with a Choice Datalog$_{nS}$ program P^*. We assume the existence of a labeling function which assigns different labels to the atoms that appear in the bodies of the clauses of P. Labels are natural numbers and are denoted by $l_1, l_2, \cdots$. The algorithm is defined by a case analysis, depending on the form of the clause being processed every time:

Case 1: The transformation of the goal clause:

$$\leftarrow p(a_1, \ldots, a_n, Y).$$

results to a set of n new unit clauses, which are added to P^*:

$$p_i^+([\,], a_i).$$

for $i = 1, \ldots, n$. The new goal clause G^* is[2]:

$$\leftarrow\ p^-([\,], Y).$$

Case 2: Let C be a clause of the form:

$$p(\vec{v}_0, Z)\ \leftarrow\ q(\vec{v}_1, Y), r(\vec{v}_2, Z).$$

and let l_1, l_2 be the labels of $q(\vec{v}_1, Y)$ and $r(\vec{v}_2, Z)$ respectively. Then C is transformed in the following way:

1. The following clause is added to P^*:

$$p^-(L, Z) \leftarrow\ r^-([l_2|L], Z).$$

2. Let X be a variable that appears in the k-th position of $\vec{v}_0$ and also in the m-th position of $\vec{v}_1$. Then the following clause is added to P^*:

$$q_m^+([l_1|L], X) \leftarrow\ [\#]p_k^+(L, X).$$

where $[\#]p_k^+ = \#p_k^+$ if X appears twice in the body of C and $[\#]p_k^+ = p_k^+$ otherwise. Similarly, let X be a variable that appears in the k-th position of $\vec{v}_0$ and also in the m-th position of $\vec{v}_2$. Then the following clause is added to P^*:

$$r_m^+([l_2|L], X) \leftarrow\ \#p_k^+(L, X).$$

3. If the output variable Y of q appears in the m-th position of $\vec{v}_2$, then the following clause is added to P^*:

$$r_m^+([l_2|L], Y) \leftarrow\ q^-([l_1|L], Y).$$

Case 3: Let C be a clause of the form:

$$p(\vec{v}_0, Z)\ \leftarrow\ q(\vec{v}_1, Z).$$

and let l_3 be the label of $q(\vec{v}_1, Z)$. Then C is transformed as follows:

1. The following clause is added to P^*:

$$p^-(L, Z) \leftarrow\ q^-([l_3|L], Z).$$

[2]Notice that in a bottom-up execution of the target program the goal clause G^* is not actually needed. However, we include it in order to emphasize that the desired answers correspond to this particular query.

2. Let X be a variable that appears in the k-th position of $\vec{v}_0$ and also in the m-th position of $\vec{v}_1$. Then the following clause is added to P^*:

$$q_m^+([l_3|L], X) \leftarrow\ p_k^+(L, X).$$

Case 4: For every EDB predicate p of P with n input variables and one output variable, a new clause of the following form is added to P^*:

$$p^-(L, Y) \leftarrow p(X_1, \ldots, X_n, Y), p_1^+(L, X_1), \ldots, p_n^+(L, X_n).$$

In the algorithm presented above we use choice predicates only when they are absolutely necessary. In the initial form of this algorithm [16] certain superfluous choice predicates were used in order to simplify the correctness proof of the algorithm. However, as we have realized since then, these redundant choice predicates cause certain performance problems in the bottom-up execution of the target code.

Example 5.1. Let P be the following simple cc-Datalog program, obtained in Example 3.4, together with a goal clause, where `p`, `q` are IDB predicates and `e`, `f`, `g` are EDB predicates:

```
← q(a1,Z).
q(X,Z) ← f(X,Z).
q(X,Z) ← e(X,Y),p(Y,Z).
p(Y,Z) ← q(Y,W),g(Y,W,Z).
```

The target program (together with a new goal clause) obtained by applying the transformation algorithm to $P \cup \{\leftarrow$ `q(a1,Z)`$\}$ is:

$$\begin{array}{l}
\leftarrow \mathtt{q^-([],Z).}\\
\mathtt{q_1^+([],a1).}\\
\mathtt{q^-(L,Z)} \leftarrow \mathtt{f^-([1|L],Z).}\\
\mathtt{f_1^+([1|L],X)} \leftarrow \mathtt{q_1^+(L,X).}\\
\mathtt{q^-(L,Z)} \leftarrow \mathtt{p^-([3|L],Z).}\\
\mathtt{p_1^+([3|L],Y)} \leftarrow \mathtt{e^-([2|L],Y).}\\
\mathtt{e_1^+([2|L],X)} \leftarrow \mathtt{q_1^+(L,X).}\\
\mathtt{p^-(L,Z)} \leftarrow \mathtt{g^-([5|L],Z).}\\
\mathtt{g_2^+([5|L],W)} \leftarrow \mathtt{q^-([4|L],W).}\\
\mathtt{g_1^+([5|L],Y)} \leftarrow \mathtt{\#p_1^+(L,Y).}\\
\mathtt{q_1^+([4|L],Y)} \leftarrow \mathtt{\#p_1^+(L,Y).}\\
\mathtt{e^-(L,Y)} \leftarrow \mathtt{e(X,Y),\ e_1^+(L,X).}\\
\mathtt{f^-(L,Y)} \leftarrow \mathtt{f(X,Y),\ f_1^+(L,X).}\\
\mathtt{g^-(L,Z)} \leftarrow \mathtt{g(X,Y,Z),\ g_1^+(L,X),\ g_2^+(L,Y).}
\end{array}$$

Consider now the following database D:

```
e(a1,a2).        f(a2,a).        g(a2,a,b3).
e(a1,b1).        f(b1,b).        g(b1,a,a3).
```

Both the initial and the final program have the single answer `Z = b3`. However, if we replace the choice predicates with the corresponding classical ones, then the resulting program will have an extra (incorrect) answer, namely `Z = a3`. The efficient bottom-up execution of programs such as the above will be discussed in Sections 7 and 8.

One might wonder if it would be possible not to separate the input arguments of a source program predicate. This might be possible but then the target language would be more complicated. Moreover, by separating the input arguments the flow of the information during execution is more explicit.

6. The Correctness of the Transformation

The correctness of the transformation algorithm is demonstrated in this section.

Let P be a simple cc-Datalog program, D a database and $\leftarrow p(a_1, \dots, a_n, Y)$ a goal clause. The correctness proof of the transformation proceeds as follows: at first we show (see Lemma 6.4 below) that if a ground instance $p(a_1, \dots, a_n, b)$ of the goal clause is a logical consequence of P_D then $p^-([\,], b)$ belongs to a limit interpretation M of P_D^*, where $P^* \cup \{\leftarrow p^-([\,], Y)\}$ is obtained by applying the transformation algorithm to $P \cup \{\leftarrow p(a_1, \dots, a_n, Y)\}$. In order to prove this result we establish a more general lemma (Lemma 6.3 below).

The inverse of Lemma 6.4 is given as Lemma 6.6. More specifically, we prove that whenever $p^-([\,], b)$ belongs to a limit interpretation of P_D^* then $p(a_1, \dots, a_n, b)$ is a logical consequence of P_D. Again, we establish this result by proving the more general Lemma 6.5. Combining the above results we get the correctness proof of the transformation algorithm (Theorem 6.1).

In the following we give some definitions that will be used in the proofs of this section.

Definition 6.1. Let P be a Choice Datalog$_{nS}$ program and D a database. Let I be an approximation to a limit interpretation of P_D and let A be an atom in I. Then a *derivation set* of A in I is recursively defined as follows:

- If A is a unit clause in P_D, then $\{A\}$ is a derivation set of A in I.
- If $S_1, \dots, S_n$ are corresponding derivation sets of $B_1, \dots, B_n$ in I and $A \leftarrow B_1, \dots, B_n$ is a ground instance of a clause in P_D, then $\{A\} \cup S_1 \cup \cdots \cup S_n$ is a derivation set of A in I.
- If A is a choice-atom in I that was introduced at an ancestor J of I and S is a derivation set of the non-choice version of A in J, then $\{A\} \cup S$ is a derivation set of A in I.

The notion of derivation set captures one of the (possibly many) ways that can lead to the generation of an atom in an approximation. Therefore, one atom may have many different derivation sets under a given approximation.

The following definitions will also be necessary:

Definition 6.2. An atom A *depends on* an atom B in I if there exists a derivation set S of A in I such that $B \in S$.

Definition 6.3. Let P be a Choice Datalog$_{nS}$ program and D a database. Let I be an approximation of a limit interpretation of P_D and assume that there exist A and B such that A depends on B in I. Let S

be a derivation set of A in I which contains B. We define the *distance between A and B in S* to be the minimum number of ground instances of program clauses used in order to establish the dependence of A from B in S. We define the *distance* between A and B in I to be the minimum of the distances between A and B in every derivation set S of A in I which contains B. When the distance between A and B in I is greater than 0 we say that *A depends essentially on B in I*.

Definition 6.4. Let P be a Choice Datalog$_{nS}$ program and D a database. Let I be an approximation to a limit interpretation of P_D and let $S \subseteq I$. Then I will be called *minimal* with respect to S if the set of choice atoms of I coincides with the set of choice atoms on which the members of S depend on in I.

In other words, the choice atoms that I contains are all "relevant" to the production of the atoms in S.

Before we proceed to the correctness proof of the algorithm, we need to establish Lemmas 6.1 and 6.2, whose proofs are given in Appendix A and B, respectively.

The first lemma that follows states that under a given list L it is not possible for $p_i^+(L, b)$ to depend on $p^-(L, a)$ (i.e. given an approximation I, there is no derivation set S for the atom $p_i^+(L, b)$ in I such that $p^-(L, a) \in S$). Intuitively, this means that the output argument of p is introduced into an approximation of a limit interpretation later than all the input arguments of p (under a given context).

Lemma 6.1. Let P be a simple cc-Datalog program, D a database and P^* the Choice Datalog$_{nS}$ program that results from the transformation. For all predicates p defined in P_D, all $L \in List(\mathcal{N})$, all $a, b \in U_{P_D}$, all input positions i of p, and all approximations I, $p_i^+(L, b)$ does not depend on $p^-(L, a)$ in I.

The lemma that follows states that under a given list L it is not possible for $p_i^+(L, b)$ to depend on $p_j^+(L, a)$ (where $p_i^+(L, b) \neq p_j^+(L, a)$). Intuitively, this means that the productions of the input arguments of p (under a given context) are independent.

Lemma 6.2. Let P be a simple cc-Datalog program and D a database. Let P^* be the Choice Datalog$_{nS}$ program that results from the transformation. For all predicates p defined in P_D, all $L \in List(\mathcal{N})$, all $a, b \in U_{P_D}$, and all input positions i, j of p, there does not exist any approximation I to any limit interpretation of P_D^* such that $p_i^+(L, b)$ depends on $p_j^+(L, a)$ where $p_i^+(L, b) \neq p_j^+(L, a)$.

The above lemma is important in proving the transformation algorithm correct. If it was possible for $p_i^+(L, b)$ to depend on $p_i^+(L, a)$ then we can imagine a scenario in which in all approximations $\#p_i^+(L, a)$ was needed to be produced before the production of $p_i^+(L, b)$. But then it would never be possible to produce $\#p_i^+(L, b)$ and as a result we might lose certain of the solutions of the initial program. In other words, the fact that the above lemma holds is important in establishing the total correctness of the transformation algorithm.

The following definition will be used in order to simplify the notation that appears in the proofs.

Definition 6.5. Let C be a simple cc-clause and p, q two predicate symbols that appear in C. Then $S_{p^+q^+}$ is the set of positions of input variables of p that are also input variables of q. Moreover, $f_{p^+q^+} : S_{p^+q^+} \to S_{q^+p^+}$ is the function that takes the position where an input variable appears in predicate p and returns its position in predicate q.

Example 6.1. Consider the following cc-clause: $\mathtt{p}(\overset{+}{\mathtt{X}}, \overset{+}{\mathtt{Y}}, \overset{-}{\mathtt{W}}) \leftarrow \mathtt{q}(\overset{+}{\mathtt{Y}}, \overset{+}{\mathtt{X}}, \overset{-}{\mathtt{Z}}), \mathtt{r}(\overset{+}{\mathtt{X}}, \overset{+}{\mathtt{Z}}, \overset{-}{\mathtt{W}})$.
In this clause, $S_{\mathtt{p}^+\mathtt{q}^+} = \{1, 2\}$, $S_{\mathtt{q}^+\mathtt{r}^+} = \{2\}$, $S_{\mathtt{r}^+\mathtt{q}^+} = \{1\}$, and $f_{\mathtt{q}^+\mathtt{r}^+}(2) = 1$.

We can now proceed to the proof of the main lemmas regarding the correctness of the algorithm.

Lemma 6.3. Let P be a simple cc-Datalog program and D a database. Let P^* be the Choice Datalog$_{nS}$ program that results from the transformation. For all predicates p defined in P_D, all $L \in List(\mathcal{N})$, and all constants $a_1, \ldots, a_n, b \in U_{P_D}$, if $p(a_1, \ldots, a_n, b) \in T_{P_D} \uparrow \omega$ and there exists an approximation I to a limit interpretation of P_D^* that is minimal with respect to the set $S = \{p_1^+(L, a_1), \ldots, p_n^+(L, a_n)\}$, then there exists an approximation J, where $I \subseteq J$, that is minimal with respect to the set $\{p^-(L, b)\}$.

Proof:
We show the above by induction on the approximations of $T_{P_D} \uparrow \omega$.
Induction Basis: The induction basis trivially holds because $T_{P_D} \uparrow 0 = \emptyset$ and thus $p(a_1, \ldots, a_n, b) \in T_{P_D} \uparrow 0$ is false.
Induction Hypothesis: We assume that: for all predicates p defined in P_D, all $L \in List(\mathcal{N})$, and all constants $a_1, \ldots, a_n, b \in U_{P_D}$, if $p(a_1, \ldots, a_n, b) \in T_{P_D} \uparrow k$ and there exists an approximation I to a limit interpretation of P_D^* that is minimal with respect to the set $S = \{p_1^+(L, a_1), \ldots, p_n^+(L, a_n)\}$, then there exists an approximation J, where $I \subseteq J$, that is minimal with respect to the set $\{p^-(L, b)\}$.
Induction Step: We demonstrate the desired result for the $k + 1$ iteration of the T_{P_D} operator. We will therefore prove that: if $p(a_1, \ldots, a_n, b) \in T_{P_D} \uparrow (k + 1)$ and there exists an approximation I to a limit interpretation of P_D^* that is minimal with respect to the set $S = \{p_1^+(L, a_1), \ldots, p_n^+(L, a_n)\}$, then there exists an approximation J, where $I \subseteq J$ that is minimal with respect to the set $\{p^-(L, b)\}$.

We use a case analysis on the way that $p(a_1, \ldots, a_n, b)$ has been introduced into $T_{P_D} \uparrow (k + 1)$.
Case 1: Assume that $p(a_1, \ldots, a_n, b)$ has been added to $T_{P_D} \uparrow (k+1)$ because it is a fact in D. According to the transformation algorithm, in P^* there exists a clause of the form:

$$p^-(L, Y) \leftarrow p(X_1, \ldots, X_n, Y), p_1^+(L, X_1), \ldots, p_n^+(L, X_n).$$

Notice that there is no other clause in P^* defining the predicate p^-. Therefore, since I is a superset of $S = \{p_1^+(L, a_1), \ldots, p_n^+(L, a_n)\}$ then I also contains $p^-(L, b)$. Moreover, I is obviously minimal with respect to $\{p^-(L, b)\}$.
Case 2: Assume that $p(a_1, \ldots, a_n, b)$ has been added to $T_{P_D} \uparrow (k + 1)$ using a clause of the form:

$$p(\cdots) \leftarrow q(\cdots), r(\cdots). \tag{1}$$

and suppose that the arities of p, q and r are $n + 1$, $n + 1$ (because of condition 2 of Definition 3.1) and $m + 1$ respectively. Then there exists a constant c instantiating the output variable of $q(\cdots)$ such that the corresponding instances of $q(\cdots)$ and $r(\cdots)$ in clause (1) are in $T_{P_D} \uparrow k$.

Consider now an input argument of q (say the t-th one) which shares its variable with an input argument of p (say the s-th one). Then a clause of the form:

$$q_t^+([l_1|L], X) \leftarrow [\#]p_s^+(L, X). \tag{2}$$

where $s = f_{q^+p^+}(t)$, has been added to P^* by the transformation algorithm. Moreover, for each input argument of r which shares its variable with an input argument of p, we have in P^* a clause of the form:

$$r_t^+([l_2|L], X) \leftarrow \#p_s^+(L, X). \tag{3}$$

where $s = f_{r^+p^+}(t)$. We start from I and alternate $C_{P_D^*}$ and $NT_{P_D^*} \uparrow \omega$ until we reach an approximation I' which contains the choice versions of those atoms in S that are needed for the production of q_t^+ under the list $[l_1|L]$ from clauses of the form (2). Notice that this can be done because no other atoms of the form $p_s^+(L, \cdots)$ can have choice versions in I (due to the minimality of I w.r.t. S, if some $\#p_i^+(L, b)$ belonged to I, then some $p_j^+(L, a_j)$ would depend on this in I, which leads to contradiction because of Lemma 6.2). Using clauses of the form (2), we see that $q_t^+([l_1|L], a_s) \in I'$, for all $t = 1, \ldots, n$ and $s = f_{q^+p^+}(t)$. Moreover, from clauses of the form (3), we derive that $r_t^+([l_2|L], a_s) \in I'$, for all $t \in S_{r^+p^+}$ and $s = f_{r^+p^+}(t)$.

Now, we need to demonstrate that I' is minimal with respect to the set $\{q_1^+([l_1|L], a_{f_{q^+p^+}(1)}), \ldots, q_n^+([l_1|L], a_{f_{q^+p^+}(n)})\}$. This is indeed the case because all input arguments of p that belong to I' are used for the production of the input arguments of q and also all choice atoms regarding input arguments of p that belong to I' are used for the production of atoms regarding the input arguments of q. Therefore we can apply the induction hypothesis on q under the list $[l_1|L]$, and on approximation I', which gives:

> Since the approximation I' is minimal with respect to the set $S' = \{q_1^+([l_1|L], a_{f_{q^+p^+}(1)}), \ldots, q_n^+([l_1|L], a_{f_{q^+p^+}(n)})\}$ and since we have that $q(a_{f_{q^+p^+}(1)}, \ldots, a_{f_{q^+p^+}(n)}, c) \in T_{P_D} \uparrow k$ then there exists an approximation J', where $I' \subseteq J'$ which is minimal with respect to $\{q^-([l_1|L], c)\}$.

Since for the input argument of r which shares its variable with the output argument of q there is a clause in P^* of the form:

$$r_w^+([l_2|L], X) \leftarrow q^-([l_1|L], X). \tag{4}$$

it follows that $r_w^+([l_2|L], c) \in J'$.

Notice also that J' is minimal with respect to the set of atoms that correspond to the input arguments of r under the context $[l_2|L]$ (because J' is minimal with respect to $\{q^-([l_1|L], c)\}$). Therefore, we can apply the induction hypothesis on r under the list $[l_2|L]$ and on approximation J', getting:

> Since $r_w^+([l_2|L], c) \in J'$ and $r_t^+([l_2|L], a_s) \in J'$, for all $t \in S_{r^+p^+}$, $s = f_{r^+p^+}(t)$, and $r(a_{r_1}, \ldots, a_{r_m}, b) \in T_{P_D} \uparrow k$ where $a_{r_l} = c$, if $l = w$, otherwise $a_{r_l} = a_s$, where $s = f_{r^+p^+}(l)$, for all $l \in S_{r^+p^+}$, then there exists an approximation J such that $J' \subseteq J$, and J is minimal with respect to the set $\{r^-([l_2|L], b)\}$.

Finally, using the fact that $r^-([l_2|L], b) \in J$ together with clause:

$$p^-(L, Y) \leftarrow r^-([l_2|L], Y). \tag{5}$$

we conclude that $p^-(L, b) \in J$. It is easy to see that J is minimal with respect to the set $\{p^-(L, b)\}$, because J is minimal with respect to the set $\{r^-([l_2|L], b)\}$.

Case 3: Assume that $p(a_1, \ldots, a_n, b)$ has been added to $T_{P_D} \uparrow (k + 1)$ using a clause of the form:

$$p(\cdots) \leftarrow q(\cdots). \tag{6}$$

The proof for this case is a simplified version of Case 2. □

Lemma 6.4. Let P be a simple cc-Datalog program, D a database and $\leftarrow$p$(a_1, \ldots, a_n, Y)$ a goal clause. Let P^* be the Choice Datalog$_{nS}$ program obtained by applying the transformation algorithm to $P \cup \{\leftarrow p(a_1, \ldots, a_n, Y)\}$. For all $b \in U_{P_D}$ the following holds: if $p(a_1, \ldots, a_n, b) \in T_{P_D} \uparrow \omega$ then there exists a limit interpretation M of P^*_D such that $p^-([\,], b) \in M$.

Proof:
Since by transforming the goal clause, the facts $p_i^+([\,], a_i)$, for $i = 1, \ldots, n$, are added to P^*, this lemma is a special case of Lemma 6.3. □

Lemma 6.5. Let P be a simple cc-Datalog program and D a database. Let P^* be the Choice Datalog$_{nS}$ program that results from the transformation. Let M be a limit interpretation of P^*_D. Then for all predicates p defined in P_D, for all $L \in List(\mathcal{N})$ and for all $b \in U_{P_D}$, if $p^-(L, b) \in M$, then there exist constants $a_1, \ldots, a_n \in U_{P_D}$ such that $p(a_1, \ldots, a_n, b) \in T_{P_D} \uparrow \omega$ and $p_i^+(L, a_i) \in M$, for $i = 1, \ldots, n$.

Proof:
We show the above by induction on the approximations of M.
Induction Basis: The induction basis is: if $p^-(L, b) \in M_0$, then there exist constants $a_1, \ldots, a_n \in U_{P_D}$ such that $p(a_1, \ldots, a_n, b) \in T_{P_D} \uparrow \omega$ and $p_i^+(L, a_i) \in M_0$, for $i = 1, \ldots, n$. This statement vacuously holds because $M_0 = \emptyset$ and thus $p^-(L, b) \in M_0$ is false.
Induction Hypothesis: If $p^-(L, b) \in M_k$, then there exist $a_1, \ldots, a_n \in U_{P_D}$ such that $p(a_1, \ldots, a_n, b) \in T_{P_D} \uparrow \omega$ and $p_i^+(L, a_i) \in M_k$, for $i = 1, \ldots, n$.
Induction Step: We prove that if $p^-(L, b) \in M_{k+1}$, then there exist constants $a_1, \ldots, a_n \in U_{P_D}$ such that $p(a_1, \ldots, a_n, b) \in T_{P_D} \uparrow \omega$ and $p_i^+(L, a_i) \in M_{k+1}$, for $i = 1, \ldots, n$.

As a first remark, notice that if M_{k+1} corresponds to a $C_{P^*_D}$ step in the chain leading to M, then the induction step holds directly (due to the induction hypothesis). Therefore we need only examine the case where $M_{k+1} = NT_{P^*_D} \uparrow \omega(M_k)$. We distinguish the following cases:
Case 1: Assume that $p^-(L, b)$ has been introduced in M_{k+1} by a clause of the form:

$$p^-(L, Y) \leftarrow p(X_1, \ldots, X_n, Y), p_1^+(L, X_1), \ldots, p_n^+(L, X_n). \tag{1}$$

Then p is an EDB predicate in D and there exist constants $a_1, \ldots, a_n$ in U_{P_D} such that $p(a_1, \ldots, a_n, b) \in T_{P_D} \uparrow 1$ and $p_i^+(L, a_i) \in M_{k+1}$, for $i = 1, \ldots, n$.
Case 2: Assume now that there exists in P a clause of the form:

$$p(\cdots) \leftarrow q(\cdots), r(\cdots). \tag{2}$$

such that the arities of p, q and r are $n + 1$, $n + 1$ (because of condition 2 of Definition 3.1) and $m + 1$ respectively, and the labels of $q(\cdots)$ and $r(\cdots)$ are l_1 and l_2 respectively. The translation of the above clause results in clauses of the form:

$$p^-(L, Y) \leftarrow r^-([l_2|L], Y). \tag{3}$$
$$r_t^+([l_2|L], Y) \leftarrow q^-([l_1|L], Y). \tag{4}$$
$$r_t^+([l_2|L], Y) \leftarrow \# p_s^+(L, Y). \tag{5}$$
$$q_t^+([l_1|L], Y) \leftarrow [\#] p_s^+(L, Y). \tag{6}$$

in P^* (where the range of the indices t and s can be easily derived from the description of the transformation algorithm). Assume now that $p^-(L,b) \in M_{k+1}$ and that $p^-(L,b)$ has been introduced in M_{k+1} by using clause (3) above.

Recall now that $M_{k+1} = NT_{P_D^*} \uparrow \omega(M_k)$. We perform an inner induction, i.e., we demonstrate that: for all $\nu \geq 0$, if $p^-(L,b) \in NT^{\nu}_{P_D^*}(M_k)$, then there exist constants $a_1,\ldots,a_n \in U_{P_D}$ such that $p(a_1,\ldots,a_n,b) \in T_{P_D} \uparrow \omega$ and $p_i^+(L,a_i) \in NT^{\nu}_{P_D^*}(M_k)$, for $i = 1,\ldots,n$. The basis case for $\nu = 0$ is trivial since it coincides with the outer induction hypothesis (since $NT^0_{P_D^*}(M_k) = M_k$); for $\nu = 1$ it also holds because of the form of the clauses derived from the transformation algorithm. Assume the statement holds for ν; we demonstrate that it is valid for $\nu+1$. Since $p^-(L,b) \in NT^{\nu+1}_{P_D^*}(M_k)$ we get that $r^-([l_2|L],b) \in NT^{\nu}_{P_D^*}(M_k)$. We can therefore apply the inner induction hypothesis on r to get:

> Since $r^-([l_2|L],b) \in NT^{\nu}_{P_D^*}(M_k)$, then there exist constants $c_1,\ldots,c_m \in U_{P_D}$ such that $r(c_1,\ldots,c_m,b) \in T_{P_D} \uparrow \omega$ and $r_j^+([l_2|L],c_j) \in NT^{\nu}_{P_D^*}(M_k)$, for $j = 1,\ldots,m$.

Notice now that the only way that $r_t^+([l_2|L],c_t)$, for all input argument positions t of r, can have been introduced in $NT^{\nu}_{P_D^*}(M_k)$, is by using either a clause of the form (4) or a clause of the form (5) above (all other clauses defining r_t^+, have a different head in the list and can not be used). We have two cases:

<u>*Subcase 2.1:*</u> If $r_t^+([l_2|L],c_t)$, with $t \in S_{r^+p^+}$ has been introduced using clause (5) above then $p_s^+(L,c_t) \in NT^{\nu}_{P_D^*}(M_k)$, for all $s \in S_{p^+r^+}$.

<u>*Subcase 2.2:*</u> If $r_t^+([l_2|L],c_t)$, where t is the input position of r which shares its variable with the output position of q in clause (2), has been introduced using clause (4), we conclude that $q^-([l_1|L],c_t) \in NT^{\nu}_{P_D^*}(M_k)$. Using the induction hypothesis on q, we get that:

> Since $q^-([l_1|L],c_t) \in NT^{\nu}_{P_D^*}(M_k)$, then there exist constants $d_1,\ldots,d_n \in U_{P_D}$ such that $q(d_1,\ldots,d_n,c_t) \in T_{P_D} \uparrow \omega$ and $q_j^+([l_1|L],d_j) \in NT^{\nu}_{P_D^*}(M_k)$, for $j = 1,\ldots,n$.

Using clauses of the form (6) above as before we get that $p_s^+(L,d_{f_{p^+q^+}(s)}) \in NT^{\nu}_{P_D^*}(M_k)$, for all $s \in S_{p^+q^+}$.

Therefore, we have that $q(d_1,\ldots,d_n,c_t) \in T_{P_D} \uparrow \omega$ and $r(c_1,\ldots,c_m,b) \in T_{P_D} \uparrow \omega$. In order for these two ground atoms to be combined using clause (2) to obtain an atom for p, we have to make sure that if a d_i and a c_j correspond to input argument positions of q and r which in clause (2) share the same variable, then $d_i = c_j$. But it is easy to see that this is ensured because both values are obtained from the same $\#p_s^+$ (which holds because the choice predicates have a unique value under a given context).

Therefore, $q(d_1,\ldots,d_n,c_t)$ and $r(c_1,\ldots,c_m,b)$ can be combined using clause (2) in order to give $p(a_1,\ldots,a_n,b) \in T_{P_D} \uparrow \omega$, where $a_1,\ldots,a_n$ is a permutation of $d_1,\ldots,d_n$.

<u>*Case 3:*</u> Assume that in P there exists a clause of the form:

$$p(\cdots) \leftarrow q(\cdots). \tag{7}$$

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

Lemma 6.6 is a special case of Lemma 6.5.

Lemma 6.6. Let P be a simple cc-Datalog program, D a database and $\leftarrow p(a_1,\ldots,a_n,Y)$ a goal clause. Let P^* be the Choice Datalog$_{nS}$ program obtained by applying the transformation algorithm to $P \cup \{\leftarrow p(a_1,\ldots,a_n,Y)\}$. For all $b \in U_{P_D}$ the following holds: if there exists a limit interpretation M of P^*_D such that $p^-([\,],b) \in M$ then $p(a_1,\ldots,a_n,b) \in T_{P_D} \uparrow \omega$.

Proof:
As $p^-([\,],b) \in M$, from Lemma 6.5 we have that there are constants $c_1,\ldots,c_n \in U_{P_D}$ such that $p(c_1,\ldots,c_n,b) \in T_{P_D} \uparrow \omega$ and $p_i^+([\,],c_i) \in M$, for $i = 1,\ldots,n$. But as the only instances of $p_i^+([\,],X)$ in M, are $p_i^+([\,],a_i)$, we conclude that $c_i = a_i$, for $i = 1,\ldots,n$. □

The following theorem demonstrates the correctness of the transformation algorithm.

Theorem 6.1. Let P be a simple cc-Datalog program, D a database and $\leftarrow p(a_1,\ldots,a_n,Y)$ a goal clause. Let P^* be the Choice Datalog$_{nS}$ program obtained by applying the transformation algorithm to $P \cup \{\leftarrow p(a_1,\ldots,a_n,Y)\}$ and $\leftarrow p^-([\,],Y)$ the new goal clause. Then for all $b \in U_{P_D}$ the following holds: there exists a limit interpretation M of P^*_D such that $p^-([\,],b) \in M$ iff $p(a_1,\ldots,a_n,b) \in T_{P_D} \uparrow \omega$.

Proof:
It is an immediate consequence of Lemmas 6.4 and 6.6. □

7. An Optimized and Terminating Bottom-up Evaluation

In this section we demonstrate a terminating bottom-up evaluation procedure for the Choice Datalog$_{nS}$ programs that result from the transformation. The basic idea behind this procedure that we propose is that during the bottom-up computation one need only take into consideration atoms whose lists (temporal terms) have length bounded by a constant which depends on the initial program P and the database D. In other words, according to this modified proof procedure, the NT_{P_D} operator reaches a fixpoint as soon as no new atoms whose list length is less than or equal to the bound are introduced.

The following definition is necessary:

Definition 7.1. A set S of atoms is *list-bounded* by $k < \omega$ if the list of each member of S has length that is less than or equal to k.

We are now in a position to state the main theorem of this section. Intuitively, the theorem states that if an atom $p(a_1,\ldots,a_n,b)$ is an answer to the goal clause G of P_D and this answer can be obtained in less than or equal to k iterations of the T_{P_D} operator, then a corresponding solution can be detected by a bottom-up computation of P^*_D that considers only atoms whose lists have length less than or equal to k.

Theorem 7.1. Let P be a simple cc-Datalog program, D a database and $\leftarrow p(a_1,\ldots,a_n,Y)$ a goal clause. Let $P^* \cup \{\leftarrow p^-([\,],Y)\}$ be the Choice Datalog$_{nS}$ program and the goal clause that result from the transformation. Assume that $p(a_1,\ldots,a_n,b) \in T_{P_D} \uparrow k$, where $k < \omega$. Then there exists a limit interpretation M of P^*_D such that $p^-([\,],b) \in M$ and $p^-([\,],b)$ has a derivation set in M that is list-bounded by k.

The proof of the theorem is a direct consequence of the following lemma:

Lemma 7.1. Let p be a predicate symbol in P_D and assume that $p(a_1, \ldots, a_n, b) \in T_{P_D} \uparrow k$, $k < \omega$. Moreover, assume that there exists an approximation I to a limit interpretation of P^*_D that is minimal with respect to the set $S = \{p^+_1(L, a_1), \ldots, p^+_n(L, a_n)\}$ and that each member of S has a derivation set in I which is list-bounded by $|L| + k$. Then there exists an approximation J, where $I \subseteq J$, that is minimal with respect to the set $\{p^-(L, b)\}$ and $p^-(L, b)$ has a derivation set in J that is list-bounded by $|L| + k$.

Proof:
The proof of the lemma is by induction on k. Actually, the proof is almost identical to that of Lemma 6.3, the only difference being that at each case of the induction step it has to be confirmed that the derivation set for $p^-(L, b)$ in J is list-bounded by $|L| + k$. The details of the proof are straightforward and are omitted. □

The above theorem suggests that it is possible to find a natural number (that depends on the characteristics of the source program P and the database D) such that the proof procedure defined for the target program P^* only considers atoms that are list-bounded by this constant. More specifically, it suffices to derive a constant k such that $T_{P_D} \uparrow k$ is the least fixpoint of T_{P_D}.

Proposition 7.1. Let P be a simple cc-Datalog program and D a database. Moreover, let c be the number of constants that appear in D, π the number of IDB predicates in P and α the maximum arity of IDB predicates in P. Let $k = \pi \cdot c^\alpha$. Then $T_{P_D} \uparrow k$ is the least fixpoint of T_{P_D}.

Proof:
In the worst case, all IDB predicates may have the same arity α and in the least fixpoint of T_{P_D} all possible combinations of constants may appear in all IDB predicates. Moreover, in the worst case, one new IDB atom is introduced at each step of the bottom-up computation. This gives a $\pi \cdot c^\alpha$ worst case bound for the number of iterations required to reach the least fixpoint of T_{P_D}. □

The above theorem easily leads to a terminating proof procedure for the target programs that result from the transformation. More specifically, during the bottom-up computation we can reject all atoms produced whose lists have length greater than the bound specified by Proposition 7.1. Then obviously, the proof procedure will terminate since the set of atoms that belong to the Herbrand base of a Choice Datalog$_{nS}$ program and that are list-bounded by a constant, is finite.

There is one further remark that can be used in order to optimize the bottom-up execution of the target programs. Let P be a program that has resulted from the transformation and let S_P be the set of choice predicates that appear in clauses of P. It is easy to see that during the bottom-up computation of the program one need only consider choice atoms whose predicate symbols belong to S_P. In other words, the choice predicates that are outside S_P are irrelevant with respect to the production of the correct answers to the goal clause of P. This optimization reduces significantly the different branches that one would have to consider during the bottom-up execution.

The above discussion leads to an optimized and terminating proof procedure described as follows: let $\widehat{NT}_{P_D}$ be exactly like NT_{P_D} the only difference being that $\widehat{NT}_{P_D}$ does not introduce any atoms whose lists have length greater than the constant k related to P_D and specified by Proposition 7.1. Moreover, let

$\widehat{C}_{P_D}$ be exactly like C_{P_D} the only difference being that $\widehat{C}_{P_D}$ returns interpretations whose choice atoms have predicates that belong to S_P. Then the notion of a P_D-chain can be modified as follows:

Definition 7.2. Let P be a Choice Datalog$_{nS}$ program and D a database. A *list-bounded P_D-chain* is a sequence $\langle M_0, M_1, M_2, \ldots\rangle$ satisfying the following conditions:

$$M_0 = \emptyset,$$

$$M_{2i+1} = \widehat{NT}_{P_D} \uparrow \omega(M_{2i}),\ \ i \geq 0,$$

and

$$M_{2i+2} \in \widehat{C}_{P_D}(M_{2i+1}),\ \ i \geq 0.$$

The following proposition is then straightforward to demonstrate:

Proposition 7.2. Let P be a Choice Datalog$_{nS}$ program that results from the transformation and D a database. Let $\langle M_0, M_1, M_2, \ldots\rangle$ be a list-bounded P_D-chain. Then for every $i \in \omega$ there exists $m \in \omega$ such that $M_{2i+1} = \widehat{NT}^{m}_{P_D}(M_{2i})$. Moreover, there exists $n \in \omega$ such that $M_n = M_{n+1}$.

The above proposition indicates that we can compute the limit of a list-bounded P_D-chain in a finite amount of time. This gives a terminating proof procedure for the target programs of the transformation.

A question that naturally arises from the above discussion is how practical the above proof procedure is. Obviously, the constant specified in Proposition 7.1 can be rather large; moreover, the number of atoms that are produced in the bottom-up computation and that are list-bounded by this constant, can be extremely large. Therefore the bound is of practical value only in those cases where the parameters that are involved in its definition are very small. The following arguments can be given in favor of the proposed approach:

- Many well-known value-propagating Datalog optimizations lead to target programs whose bottom-up execution suffers from the problem of non-termination ([28]). Therefore even the existence of Proposition 7.1 adds a desirable characteristic to the proposed technique.

- As we have realized in practice, there exist many cases in which the bottom-up execution of the target program terminates even without resorting to the above list-bounded proof procedure.

- As it will become obvious in the next section, there exist transformations that can reduce the number of list labels that appear in the target program of the transformation. This means that the number of list-bounded atoms is also reduced and therefore the above proof-procedure becomes more efficient. Actually, in many interesting cases, the list labels can be completely eliminated, and therefore the target program is in fact an optimized Datalog one that can be executed very efficiently ([19]).

In the next section we discuss certain transformations that enhance the performance of the target code of the proposed transformation.

8. Optimizing the Target Program

In this section we demonstrate that the programs obtained by the transformation of Section 5 can be significantly simplified by using unfolding and elimination of redundant clauses. For this we define and prove the correctness of the unfolding and elimination rules for Choice Datalog$_{nS}$ programs in Subsection 8.1, and then in Subsections 8.2 and 8.3, we demonstrate how these rules can be used to simplify the program obtained by the algorithm presented in Section 5.

8.1. Unfolding and elimination of redundant clauses in Choice Datalog$_{nS}$

The unfolding rule and elimination of redundant clauses that we will describe below are similar to the ones defined for classical logic programming [26, 6, 15]. The main difference is the existence of choice atoms in clauses.

Definition 8.1. Let P be a Choice Datalog$_{nS}$ program and C be a clause in P of the form:

$$A \leftarrow A_1, \dots, A_{i-1}, A_i, A_{i+1}, \dots, A_n$$

where each one of $A_1, \dots, A_{i-1}$ and $A_{i+1}, \dots, A_n$ may be either choice or non-choice atom while A_i is a non-choice atom. Let $C_1, C_2, \dots, C_m$ be all clauses in P, whose heads are unifiable with A_i by most general unifiers $\theta_1, \theta_2, \dots, \theta_m$ respectively. The result of *unfolding C at* A_i is the set of clauses $\{C'_1, \dots, C'_m\}$ such that, for each j, with $1 \le j \le m$, if C_j is the clause:

$$B_j \leftarrow B_{j_1}, \dots, B_{j_h}.$$

with $h \ge 0$ and $B_j\theta_j = A_i\theta_j$, then C'_j is the clause:

$$(A \leftarrow A_1, \dots, A_{i-1}, B_{j_1}, \dots, B_{j_h}, A_{i+1}, \dots, A_n)\theta_j$$

Then the program obtained by unfolding is the program $P' = (P - \{C\}) \cup \{C'_1, \dots, C'_m\}$. The clause C is called the *unfolded clause*, $C_1, \dots, C_m$ are called the *unfolding clauses* and A_i is called the *unfolded atom*.

Notice that the unfolding rule defined above can be applied only to non-choice body atoms. The following lemma demonstrates the correctness of the unfolding rule.

Lemma 8.1. Let P be a Choice Datalog$_{nS}$ program, and P' be the program obtained by unfolding a clause C in P. Then for every database D both P_D and P'_D have the same sets of limit interpretations.

Proof:
Let S_k and S'_k be the sets of the approximations to the limit interpretations of P_D and P'_D respectively that are obtained during the k-th iterations of the bottom-up computation (i.e. the set of the k-th elements of all P_D-chains obtained as described in Definition 4.6). We will use induction on k to prove that $S_k = S'_k$ for all $k \in \omega$.

Base Case: The lemma holds for $k = 0$ since $S_0 = S'_0 = \{\emptyset\}$.
Induction Hypothesis: We assume that $S_k = S'_k$ for all $k \le n$, with $n \in \omega$.

Induction Step: We will prove that $S_{n+1} = S'_{n+1}$. We distinguish the following cases:
Case 1: Let $n = 2l + 1$, $l \geq 0$. It is easy to see that $S_{n+1} = S'_{n+1} = \{M | M \in C_{P_D}(M'),\ M' \in S_n\}$.
Case 2: Let $n = 2l + 2$ for some $l \geq 0$ and let $J \in S_n$. Then because of the induction hypothesis, $J \in S'_n$. It is easy to see that the approximations obtained from J in S_{n+1} and in S'_{n+1} are $I = NT_{P_D} \uparrow \omega(J)$ and $I' = NT_{P'_D} \uparrow \omega(J)$, respectively. We will prove that $I = I'$ by proving that *a)* $I \subseteq I'$, and *b)* $I' \subseteq I$.

Proof of (a): We use induction on i to prove that $NT^i_{P_D}(J) \subseteq I'$.
Base Case (of a): For $i = 0$ we have: $NT^0_{P_D}(J) = NT^0_{P'_D}(J) = J$. Because of Lemma 4.1, $J \subseteq I'$, and thus we derive that $NT^0_{P_D}(J) \subseteq I'$.
Induction Hypothesis (of a): If $A \in NT^m_{P_D}(J)$, then $A \in I'$.
Induction Step (of a): We will prove that if $A \in NT^{m+1}_{P_D}(J)$, then $A \in I'$. Suppose that A has been introduced in $NT^{m+1}_{P_D}(J)$ by applying a clause C of P_D.
Case 1 (of a): C also belongs to P'_D. Then as the body atoms in the instance of C used to introduce A in $NT^{m+1}_{P_D}(J)$ belong to $NT^m_{P_D}(J)$, they also belong to I', because of the induction hypothesis. Therefore, A is also an atom in I' as it can be introduced using C.
Case 2 (of a): C does not belong to P' because it has been unfolded in P. Let C be of the form:

$$A_0 \leftarrow A_1, \ldots, A_{i-1}, A_i, A_{i+1}, \ldots, A_n.$$

where A_i is the unfolded atom. Let $C_1, C_2, \ldots, C_s$ be all clauses in P, whose heads are unifiable with A_i by most general unifiers $\theta_1, \theta_2, \ldots, \theta_s$ respectively. It is easy to see that since (a ground instance of) C has been used to introduce A in $NT^{m+1}_{P_D}(J)$, the corresponding instance of A_i is in $NT^m_{P_D}(J)$ and has been introduced using one of the clauses $C_1, C_2, \ldots, C_s$ as these are all the clauses in P whose heads are unifiable with A_i. Suppose that this clause is C_j:

$$B_j \leftarrow B_{j_1}, \ldots, B_{j_h}.$$

with $h \geq 0$. As C_j is an unfolding clause, a clause C'_j of the form:

$$(A_0 \ \leftarrow\ A_1, \ldots, A_{i-1}, B_{j_1}, \ldots, B_{j_h}, A_{i+1}, \ldots, A_n)\theta_j.$$

has been introduced in P', where $\theta_j = mgu(B_j, A_i)$ and thus $B_j\theta_j = A_i\theta_j$. It is easy to see that A also belongs to I' as it can be introduced in I' by the appropriate ground instance of the clause C'_j. The atoms in the body of this clause are the instances of the atoms in the bodies of the instances of the clauses C and C_j that have been used in the introduction of A in $NT^{m+1}_{P_D}(J)$. These atoms also belong to I' because of the induction hypothesis (as they belong to $NT^m_{P'_D}(J)$).

Proof of (b): As in the proof of (a) we will use induction on i to prove that $NT^i_{P'_D}(J) \subseteq I$.
Base Case (of b): As in Base Case of (a) we prove that $NT^0_{P'_D}(J) \subseteq I$.
Induction Hypothesis (of b): If $A \in NT^m_{P'_D}(J)$, then $A \in I$.
Induction Step (of b): We will prove that if $A \in NT^{m+1}_{P'_D}(J)$, then $A \in I$. Suppose that A has been introduced in $NT^{m+1}_{P_D}(J)$ by applying a clause E of P'.
Case 1 (of b): E also belongs to P. Then A has also been introduced in I using E (the proof is similar to Case 1 of (a)).

Case 2 (of b): E is between the clauses of P' obtained by unfolding a clause C in P. It is easy to prove (in a similar way as in the proof of Case 2 of (a)), that the atom A has also been introduced in I by using the clause C and the unfolding clause used to obtain E. □

As we will see in Subsection 8.2, when applying the unfolding rule some unfolding clauses become redundant (since we are interested in retaining equivalence only with respect to specific predicates). We can thus eliminate these clauses. The elimination rule is given formally in the following two definitions.

Definition 8.2. Let P be a Choice Datalog$_{nS}$ program. The *predicate dependency graph* G_P of P is a directed graph (V, E) where V is the set of predicate symbols in P and E is a set of edges such that $(p, q) \in E$ iff there is a clause C in P whose head predicate is p, and there is an atom or a choice version of an atom in the body of C whose predicate is q.

Definition 8.3. Let P be a Choice Datalog$_{nS}$ program, and S be a set of predicate symbols in P. A clause C in P is said to be *redundant with respect to* S *in* P iff there is no path in the predicate dependency graph G_P of P leading from a predicate in S to the head predicate of C.

The following lemma can then be easily proved.

Lemma 8.2. Let P be a Choice Datalog$_{nS}$ program, and S be a set of predicate symbols in P. Let P' be the program obtained by deleting all clauses in P which are redundant with respect to S. Let A be an atom or a choice version of an atom whose predicate symbol is in S. Then for every database D, if there exists a limit interpretation M of P_D such that $A \in M$, then there exists a limit interpretation M' of P'_D such that $A \in M'$.

The transformation rules of this section can be used to optimize the target program as shown below.

8.2. Elimination of IDB predicates obtained from EDB body atoms of the source program

All IDB predicates that appear in the target program P^* which correspond to EDB atoms that appear in the bodies of the clauses in the source program P, can be eliminated using unfolding. In particular, the predicates in the heads of the clauses added to P^* in Case 4 of the algorithm (which are output predicates) appear only in the bodies of (some of) the clauses in P^*, added in Cases 2-3 of the algorithm. All clauses containing these atoms in their bodies can be unfolded using the clauses introduced in Case 4. The clauses obtained by these unfolding steps contain occurrences of input predicates corresponding to EDB predicates. These occurrences can be further unfolded resulting to the complete elimination of all (input and output) predicates corresponding to EDB predicates of the source program. It is easy to see (because of the structure of the program P^*) that none of the unfolded atoms occurs in its choice version and therefore all these unfolding steps are valid according to Definition 8.1. Notice that all labels corresponding to EDB body atoms in the source program are also eliminated through this process.

Example 8.1. Consider the program P^* obtained in Example 5.1.

$$
\begin{array}{ll}
(G) & \leftarrow \mathtt{q^-([],Z).} \\
(C_1) & \mathtt{q_1^+([],a1).} \\
(C_2) & \mathtt{q^-(L,Z)} \leftarrow \mathtt{f^-([1|L],Z).} \\
(C_3) & \mathtt{f_1^+([1|L],X)} \leftarrow \mathtt{q_1^+(L,X).} \\
(C_4) & \mathtt{q^-(L,Z)} \leftarrow \mathtt{p^-([3|L],Z).} \\
(C_5) & \mathtt{p_1^+([3|L],Y)} \leftarrow \mathtt{e^-([2|L],Y).} \\
(C_6) & \mathtt{e_1^+([2|L],X)} \leftarrow \mathtt{q_1^+(L,X).} \\
(C_7) & \mathtt{p^-(L,Z)} \leftarrow \mathtt{g^-([5|L],Z).} \\
(C_8) & \mathtt{g_2^+([5|L],W)} \leftarrow \mathtt{q^-([4|L],W).} \\
(C_9) & \mathtt{g_1^+([5|L],Y)} \leftarrow \mathtt{\#p_1^+(L,Y).} \\
(C_{10}) & \mathtt{q_1^+([4|L],Y)} \leftarrow \mathtt{\#p_1^+(L,Y).} \\
(C_{11}) & \mathtt{e^-(L,Y)} \leftarrow \mathtt{e(X,Y),\ e_1^+(L,X).} \\
(C_{12}) & \mathtt{f^-(L,Y)} \leftarrow \mathtt{f(X,Y),\ f_1^+(L,X).} \\
(C_{13}) & \mathtt{g^-(L,Z)} \leftarrow \mathtt{g(X,Y,Z),\ g_1^+(L,X),\ g_2^+(L,Y).}
\end{array}
$$

We will eliminate all IDB atoms in P^* (and the clauses defining them) corresponding to the EDB predicates `e`, `f` and `g`, by using the transformation rules defined in Subsection 8.1.

Unfolding (C_5) using (C_{11}) we get:

$$(C_{14})\quad \mathtt{p_1^+([3|L],Y)} \leftarrow \mathtt{e(X,Y),\ e_1^+([2|L],X).}$$

Now unfolding (C_{14}) using (C_6) we get:

$$(C_{15})\quad \mathtt{p_1^+([3|L],Y)} \leftarrow \mathtt{e(X,Y),\ q_1^+(L,X).}$$

Now we can replace (C_{15}) for (C_5) in P^*. Clauses (C_6) and (C_{11}) can be eliminated as they are now redundant with respect to the set $S = \{\mathtt{q^-}\}$ containing the goal predicate.

Applying the same process we can eliminate the predicates $\mathtt{f^-}$ and $\mathtt{f_1^+}$ and the clauses defining them. In particular, we unfold (C_2) using (C_{12}) and then we unfold the clause obtained using (C_3). In this way we get the clause:

$$(C_{16})\quad \mathtt{q^-(L,Z)} \leftarrow \mathtt{f(X,Z),\ q_1^+(L,X).}$$

which replaces clause (C_2) in P^*. Again we eliminate the redundant clauses (C_3) and (C_{12}).

Applying the same process we can eliminate the predicates $\mathtt{g_1^+}$, $\mathtt{g_2^+}$ and $\mathtt{g^-}$ that correspond to the EDB predicate `g`. For this we unfold (C_7) using (C_{13}), and then we unfold the resulting clause using the clauses (C_8) and (C_9). In this way we get the clause:

$$(C_{17})\quad \mathtt{p^-(L,Z)} \leftarrow \mathtt{g(X,Y,Z),\ \#p_1^+(L,X),\ q^-([4|L],Y).}$$

Clause (C_{17}) can replace (C_7) while clauses (C_8), (C_9) and (C_{13}) can be eliminated as they become redundant with respect to the set $S = \{\mathtt{q^-}\}$. The program obtained by applying the above transformation is:

(G) $\leftarrow$ `q`$^{-}$`([],Z).`

(C_1) `q`$_1^{+}$`([],a1).`

(C_4) `q`$^{-}$`(L,Z)` $\leftarrow$ `p`$^{-}$`([3|L],Z).`

(C_{10}) `q`$_1^{+}$`([4|L],Y)` $\leftarrow$ `#p`$_1^{+}$`(L,Y).`

(C_{15}) `p`$_1^{+}$`([3|L],Y)` $\leftarrow$ `e(X,Y), q`$_1^{+}$`(L,X).`

(C_{16}) `q`$^{-}$`(L,Z)` $\leftarrow$ `f(X,Z), q`$_1^{+}$`(L,X).`

(C_{17}) `p`$^{-}$`(L,Z)` $\leftarrow$ `g(X,Y,Z), #p`$_1^{+}$`(L,X), q`$^{-}$`([4|L],Y).`

It is easy to see that we have managed to eliminate IDB atoms in P^* (and the clauses defining them) that correspond to the EDB predicates `e`, `f` and `g`. Moreover, the labels 1, 2, and 5, corresponding to the EDB body atoms of the source program have also been eliminated.

8.3. More optimizations using unfolding

In the previous subsection we have seen how we can eliminate the IDB predicates that correspond to EDB body atoms from the program obtained by applying the transformation algorithm of Section 5. However, as we will see in Example 8.2, we can apply some more unfolding steps to further simplify the program.

Example 8.2. Consider the program obtained in Example 8.1. Observe that we can unfold the clause (C_4) at `p`$^{-}$`([3|L],Z)` using the clause (C_{17}). In this way we get the clause:

(C_{18}) `q`$^{-}$`(L,Z)` $\leftarrow$ `g(Y,W,Z), #p`$_1^{+}$`([3|L],Y), q`$^{-}$`([4,3|L],W).`

The clause (C_{18}) can now replace (C_4) in the program and the clause (C_{17}) is redundant with respect to $S = \{$`q`$^{-}\}$ and can be eliminated. In this way we get the program:

(G) $\leftarrow$ `q`$^{-}$`([],Z).`

(C_1) `q`$_1^{+}$`([],a1).`

(C_{10}) `q`$_1^{+}$`([4|L],Y)` $\leftarrow$ `#p`$_1^{+}$`(L,Y).`

(C_{15}) `p`$_1^{+}$`([3|L],Y)` $\leftarrow$ `e(X,Y), q`$_1^{+}$`(L,X).`

(C_{16}) `q`$^{-}$`(L,Z)` $\leftarrow$ `f(X,Z), q`$_1^{+}$`(L,X).`

(C_{18}) `q`$^{-}$`(L,Z)` $\leftarrow$ `g(Y,W,Z), #p`$_1^{+}$`([3|L],Y), q`$^{-}$`([4,3|L],W).`

We can also unfold clauses (C_{15}) and (C_{16}) at the atom `q`$_1^{+}$`(L,X)`. In this way we get the clauses:

(C_{19}) `p`$_1^{+}$`([3],Y)` $\leftarrow$ `e(a1,Y).`

(C_{20}) `p`$_1^{+}$`([3,4|L],Y)` $\leftarrow$ `e(X,Y), #p`$_1^{+}$`(L,X).`

(C_{21}) `q`$^{-}$`([],Z)` $\leftarrow$ `f(a1,Z).`

(C_{22}) `q`$^{-}$`([4|L],Z)` $\leftarrow$ `f(X,Z), #p`$_1^{+}$`(L,X).`

Clauses (C_{19}), (C_{20}), (C_{21}), and (C_{22}) can replace clauses (C_{15}) and (C_{16}) in the final program and clauses (C_1) and (C_{10}) can be eliminated as they become redundant with respect to the set $S = \{\texttt{q}^-\}$. In this way we obtain the program:

(G) $\leftarrow \texttt{q}^-\texttt{([],Z).}$
(C_{19}) $\texttt{p}_1^+\texttt{([3],Y)} \leftarrow \texttt{e(a1,Y).}$
(C_{20}) $\texttt{p}_1^+\texttt{([3,4|L],Y)} \leftarrow \texttt{e(X,Y), \#p}_1^+\texttt{(L,X).}$
(C_{21}) $\texttt{q}^-\texttt{([],Z)} \leftarrow \texttt{f(a1,Z).}$
(C_{22}) $\texttt{q}^-\texttt{([4|L],Z)} \leftarrow \texttt{f(X,Z), \#p}_1^+\texttt{(L,X).}$
(C_{18}) $\texttt{q}^-\texttt{(L,Z)} \leftarrow \texttt{g(Y,W,Z), \#p}_1^+\texttt{([3|L],Y), q}^-\texttt{([4,3|L],W).}$

It is important to note here that in the case of the elimination of the IDB atoms corresponding to the EDB body atoms of the source program, we have an algorithm (presented in Subsection 8.2), which can be applied to perform this optimization. On the other hand, for the unfolding steps applied in Example 8.2 we do not have at the moment a concrete algorithm. However, we believe that one can be guided by the structure of the source program so as to perform most of the appropriate unfoldings in an algorithmic way. However, we do not pursue these issues any further here.

8.4. Eliminating Choice Predicates

The bottom-up execution of the target programs could be significantly enhanced if one could eliminate certain choice atoms or replace them by their non-choice versions. This is possible if we know that a given predicate is deterministic:

Definition 8.4. Let P be a Choice Datalog$_{nS}$ program and D a database. An IDB predicate p of P will be called *deterministic* with respect to the database D, if for every L, whenever two atoms $p(L, a)$ and $p(L, b)$ arise during the bottom-up evaluation of P_D, then $a = b$. An EDB predicate p is *deterministic* if its output argument is uniquely determined by its input arguments.

Detecting determinism in the program obtained by our technique (after applying also the optimizations described above) is very important since one can replace the choice atoms of a deterministic predicate that appear in the program by their non-choice versions. Such a transformation results in fewer branches that have to be followed during the bottom-up evaluation of the program. Moreover it makes the program amenable to other optimizing transformations.

There are many cases of determinism that can be detected in a static way. Consider a database D and a program P that has resulted after applying the transformation and the optimizations that have been described so far. The following definition and the subsequent lemma provides a means for detecting a non-trivial class of deterministic predicates.

Definition 8.5. An IDB predicate p will be called *potentially deterministic* if its definition consists of clauses of the following forms:

$i)\quad p([\,],a).$
$ii)\quad p([l],Y) \leftarrow e(a_1,\ldots,a_n,Y).$
$iii)\quad p(L,Y) \leftarrow [\#]q(L_1,Y).$
$iv)\quad p(L,Y) \leftarrow e(X_1,\ldots,X_n,Y),[\#]q_1(L_1,X_1),\ldots,[\#]q_n(L_n,X_n).$

where each L_i, for $i = 1,\ldots,n$, is a proper suffix of L.

Lemma 8.3. (**Testing determinism**) Let S be a set of predicates of P and assume that the following conditions hold for every member p of S:

1. If p is an EDB predicate then it is deterministic and if it is an IDB predicate then it is potentially deterministic.
2. The temporal arguments of the head atoms of all clauses defining p specify disjoint sets of time points.
3. The predicate of every atom that appears in the body of a clause that defines p belongs to S.

Then every predicate in S is a deterministic predicate with respect to D.

Proof:
(Sketch) The lemma can be easily proved by induction on the structure of the temporal argument (list).
□

Example 8.3. Consider the program obtained in Example 8.2. It is easy to see that the test defined above applies to the set $\{\mathtt{p}_1^+,\ \mathtt{e}\}$ in all cases that the EDB predicate `e` is deterministic. In these cases we can replace the choice atoms by their non-choice versions obtaining the program:

$(G)\quad \leftarrow \mathtt{q}^-\mathtt{([],Z)}.$
$(C_{19})\quad \mathtt{p}_1^+\mathtt{([3],Y)} \leftarrow \mathtt{e(a1,Y)}.$
$(C'_{20})\quad \mathtt{p}_1^+\mathtt{([3,4|L],Y)} \leftarrow \mathtt{e(X,Y)},\ \mathtt{p}_1^+\mathtt{(L,X)}.$
$(C_{21})\quad \mathtt{q}^-\mathtt{([],Z)} \leftarrow \mathtt{f(a1,Z)}.$
$(C'_{22})\quad \mathtt{q}^-\mathtt{([4|L],Z)} \leftarrow \mathtt{f(X,Z)},\ \mathtt{p}_1^+\mathtt{(L,X)}.$
$(C'_{18})\quad \mathtt{q}^-\mathtt{(L,Z)} \leftarrow \mathtt{g(X,Y,Z)},\ \mathtt{p}_1^+\mathtt{([3|L],X)},\ \mathtt{q}^-\mathtt{([4,3|L],Y)}.$

Notice however that if `e` was not deterministic, this optimization could not have been applied.

8.5. Reducing the overhead in list manipulation

One aspect of the branching transformation that seems (at first sight) to impose performance limitations, is the fact that the transformation is heavily based on list computations. For example, assume that during the bottom-up evaluation of a program, an atom is created; in order to verify that this atom has not already been introduced in a previous step, we have to compare it with the atoms that have already been produced. However, since an atom can contain an arbitrarily long list, this seems to suggest that the operation of comparing two atoms can be very costly. Fortunately, there is a simple and elegant solution out of this problem, one that has been extensively used in the corresponding version of the branching transformation for functional programming languages [21]. The idea is that lists can be encoded by small natural numbers using a technique known as *hash-consing*. The main advantage of hash-consing is that two lists can be tested for equality with a single operation, rather than with a loop which scans the lists and compares corresponding elements. The only (minor) disadvantage is that with each cons operation we must consult a hash table to check that the list we are constructing does not already have a representation.

We store the list representatives in a table each row of which is a pair *(head, index of tail)*. The list is then represented (or encoded) by the index of the appropriate pair in the table. The following primitive functions are used in order to implement hash-consing:

- $hashcons(head, tail\_code)$: Uses a hash function to check if the pair $(head, tail\_code)$ already exists in the hash table. If it does not, then it inserts it. Finally, it returns the position of the pair in the table.
- $hashhead(list\_code)$: It returns the first element of the pair found in the $list\_code$ position of the hash table.
- $hashtail(list\_code)$: It returns the second element of the pair found in the $list\_code$ position of the hash table.

The above operations can be performed efficiently and the space occupied by the hash table is reasonable. Using hash-consing and the above primitives, an efficient implementation of the technique can be built, that avoids expensive list management. For more details regarding this technique, see for example [21].

8.6. Discussion on the optimizations

As the reader may have realized by now, the target program of our transformation is amenable to a variety of optimizing transformations. Many of these transformations may be expressed in an algorithmic way and may apply to wide classes of target programs.

It is important to note that the elimination of unnecessary choice atoms is of paramount importance because the presence of non-determinism is a source of execution overhead. As we have seen in Subsection 8.4 the necessity of using the choice version of an atom usually depends on the structure of the database on which a program applies (and not only on the structure of the program itself). The criterion that we propose in Subsection 8.4 is a first step towards reducing the overhead introduced by non-determinism. The invention of more widely applicable tests will be the subject of future work.

Another important source of optimizations concerns the elimination of list labels that are not actually necessary. This improvement is essential since it reduces the execution time of the program. Other promising optimizations that help reduce the number of list labels can be obtained by refining the *Elimination of redundant next operators,* and the *Elimination of temporal operators concerning left recursive calls* proposed in [19]. However, it is not immediately obvious how these optimizations interact with the existence of choice atoms. We are currently investigating these issues.

Concluding, we believe that the issue of optimizing the target program of our transformation can still produce interesting results. However, even in its present form, the proposed approach competes with the existing approaches (which have been developed and undergone improvements for many years).

9. Related Work

The work presented in this paper contributes to the area of query optimization in deductive databases. More specifically, the proposed transformation belongs to a well-known class of techniques in which the input values of the query-goal are propagated in order to restrict the generation of irrelevant atoms during the bottom-up computation of the desired final answers. In this section we present a comparison of the proposed transformation with the most well-known value-propagating techniques. The comparison is not exhaustive since this would require the availability of stable implementations for all the techniques involved and the experimentation with a large number of programs and underlying databases. However, in the following we attempt to identify the main advantages and disadvantages of our approach with respect to its main competitors.

Among the existing Datalog optimizations, the ones that are most closely related to the present approach are the *magic sets transformation* [3, 25], the *counting method* [24] and the *pushdown approach* [9]. In the following, we discuss the relative merits and drawbacks of each one of them with respect to our technique. In the last subsection we compare all these techniques with respect to their performance on specific examples.

9.1. A Comparison with Magic Sets

The most widely known approach in the area of value-propagating Datalog optimizations, is the magic sets transformation. In this approach, for each IDB predicate of the source program a new predicate, called *magic predicate*, is introduced. The arguments of a magic predicate are the bound arguments of the corresponding IDB predicate of the initial program and the aim is to push selections into clauses. The magic sets can be applied to general Datalog programs [3, 25] and therefore it is the most general among all similar techniques. This generality, however, can prove a disadvantage in many cases: as we argue in subsection 9.3, the proposed transformation has the same advantages over magic sets as the counting technique does. More specifically, it is well-known [28] that on a variety of databases, counting outperforms magic sets; in particular, there exist programs which (under certain databases) terminate in $O(n)$ time using counting but require $\Omega(n^2)$ time for magic sets. As we will demonstrate at the end of this section, this favorable situation appears as well for the proposed transformation.

9.2. A Comparison with Counting and Pushdown

The counting technique uses integer indices in order to encode two fixpoint computations: one for the propagation of the bindings in the top-down phase and the other for the generation of the desired result in the bottom-up computation. In other words, the integer indices of counting play the role of the lists in the present transformation.

In its initial form, counting was applicable only to a restricted class of queries [28, 10]. Later, it was extended to *generalized counting*, which applies to a broader class of queries having the so called *binding-passing property (BPP)*[24]. Intuitively, BPP guarantees that “bindings can be passed down to any level of recursion”. However, the class of BPP programs does not include all the Chain Datalog ones and hence not all the cc-Datalog ones. For an example, consider the traversal of a graph using double recursion (the program in Section 2). The generalized counting method can not treat this case, since the corresponding program does not have the BPP; but this program is actually a Chain Datalog one, and therefore it can be easily transformed by the branching-time transformation. In other words, counting is less general than the branching-time transformation.

Another issue that arises when comparing these techniques concerns the termination of bottom-up evaluation of the resulted programs. Consider first the case of branching transformation. Then as we have demonstrated, for every program and database there exists a bound (that depends on the program and database characteristics) such that all the answers to a query are obtained in a number of iterations that does not exceed this bound. On the other hand, in case of the counting technique alternative methods have been proposed in order to ensure termination. For example, in [12] a criterion is proposed that applies to databases that can be represented as graphs; this criterion, however, is not general enough to cover all the database cases (i.e. the non-graph representable ones). Another method that has been proposed for the same purpose is the *magic counting* [23] approach. This method provides a way to deal with cyclic databases provided one knows in advance the structure of the database in order to decide which of the variants of the method is most appropriate.

Recapitulating, the counting and the branching transformations share some common philosophy (the former uses integer indices while the latter lists of integers to control bottom-up evaluation). However, the branching transformation appears to have a more general termination criterion and applies to a wider class of programs.

The pushdown method is based on the relationship between chain queries and context-free languages. Moreover, the context-free language corresponding to a query is associated to a pushdown automaton; in this way the initial query is rewritten in a more suitable form for efficient bottom-up execution. The *push-down method* applies to all chain queries [9], and hence it covers a subclass of the proposed technique, as it cannot treat the case of multiple consumptions of variables.

9.3. A Comparison through Examples

In the following, we compare the above techniques using four example programs. The first one is the well-known *same generation* program (which is actually a Chain Datalog program); the second one is the `path` program of Section 3, which has multiple consumptions of variables. The third one is a doubly recursive program which computes the odd-length paths of a given color in a graph; finally, the fourth example is the running example that we have been using so far.

9.3.1. The same-generation program

The same-generation program has been used extensively in comparing the performance of various Datalog optimizations:

```
← sg(a,Z).
sg(X,Y) ← equal(X,Y).
sg(X,Y) ← par(X,Xp),sg(Xp,Yp),rap(Yp,Y).
```

Consider a database consisting of the unit clauses `par(a,b`$_i$`)`, `par(b`$_i$`,c)`, `rap(b`$_i$`,a)` and `rap(c, b`$_i$`)`, where $1 \leq i \leq n$, and `equal` is the usual equality predicate (this is a slightly modified version of the program given in [28], page 831). The proposed transformation when applied to the (simple cc-Datalog version of the) above program produces as output (after applying the appropriate unfoldings described in

Subsection 8.2):

$$\begin{array}{l}\leftarrow \texttt{sg}^{-}\texttt{([],Z).}\\ \texttt{sg}_1^{+}\texttt{([],a).}\\ \texttt{sg}^{-}\texttt{(L,Y)} \leftarrow \texttt{equal(X,Y),sg}_1^{+}\texttt{(L,X).}\\ \texttt{sg}^{-}\texttt{(L,Y)} \leftarrow \texttt{rap(Yp,Y),sg}^{-}\texttt{([5,2|L],Yp).}\\ \texttt{sg}_1^{+}\texttt{([5,2|L],Xp)} \leftarrow \texttt{par(X,Xp),sg}_1^{+}\texttt{(L,X).}\end{array}$$

It can be easily seen that the above program terminates producing the desired solution (Z = a) in time $O(n)$. On the other hand, the corresponding magic sets program ([28], page 856) requires time $\Omega(n^2)$ (see also the discussion in [28], page 947). In other words, our transformation appears to give promising results with respect to magic sets.

9.3.2. The colored-path program

Consider the cc-Datalog program of Example 3.2:

$$\begin{array}{l}\leftarrow \texttt{path(a,red,Z).}\\ \texttt{path(X,Color,Z)} \leftarrow \texttt{edge(X,Color,Z).}\\ \texttt{path(X,Color,Z)} \leftarrow \texttt{edge(X,Color,W),path(W,Color,Z).}\end{array}$$

together with the database: edge(a,red,b$_i$), edge(b$_i$,red,c), edge(c,red,d$_i$), $i = 1, ..., n$.

The target program obtained after applying the proposed transformation and the appropriate unfoldings is as follows:

$$\begin{array}{l}\leftarrow \texttt{path}^{-}\texttt{([],Z).}\\ \texttt{path}_1^{+}\texttt{([],a).}\\ \texttt{path}_2^{+}\texttt{([],red).}\\ \texttt{path}^{-}\texttt{(L,Z)} \leftarrow \texttt{edge(X,Color,Z),path}_1^{+}\texttt{(L,X),path}_2^{+}\texttt{(L,Color).}\\ \texttt{path}^{-}\texttt{(L,Z)} \leftarrow \texttt{path}^{-}\texttt{([3|L],Z).}\\ \texttt{path}_1^{+}\texttt{([3|L],W)} \leftarrow \texttt{edge(X,Color,W),path}_1^{+}\texttt{(L,X),\#path}_2^{+}\texttt{(L,Color).}\\ \texttt{path}_2^{+}\texttt{([3|L],Color)} \leftarrow \texttt{\#path}_2^{+}\texttt{(L,Color).}\end{array}$$

It is easy to see that Lemma 8.3 can be applied to $\{\texttt{path}_2^{+}\}$, thus $\texttt{path}_2^{+}$ is deterministic with respect to the given/every database (in fact, in every context its data argument gets the value red). Therefore, the choice atoms can be replaced by their non-choice versions obtaining a program free of choice atoms. Moreover, it is easy to see that the atoms whose predicate is $\texttt{path}_2^{+}$ can be completely removed, obtaining in this way the following program:

$$\begin{array}{l}\leftarrow \texttt{path}^{-}\texttt{([],Z).}\\ \texttt{path}_1^{+}\texttt{([],a).}\\ \texttt{path}^{-}\texttt{(L,Z)} \leftarrow \texttt{edge(X,red,Z),path}_1^{+}\texttt{(L,X).}\\ \texttt{path}^{-}\texttt{(L,Z)} \leftarrow \texttt{path}^{-}\texttt{([3|L],Z).}\\ \texttt{path}_1^{+}\texttt{([3|L],W)} \leftarrow \texttt{edge(X,red,W),path}_1^{+}\texttt{(L,X).}\end{array}$$

The answers to the goal (`Z = b`$_i$`, c, d`$_i$, for $i = 1, ..., n$) are produced in $O(n)$ time.
On the other hand, the magic sets program:

```
m_path(a,red).
m_path(W,Color) ← m_path(X,Color),edge(X,Color,W).
path(X,Color,Z) ← m_path(X,Color),edge(X,Color,Z).
path(X,Color,Z) ← m_path(X,Color),edge(X,Color,W),path(W,Color,Z).
```

produces all facts `path(b`$_i$`,red,d`$_j$`)`, for $i, j = 1, .., n$ and hence needs $\Omega(n^2)$ time. Even the more sophisticated extension of the magic sets, i.e. the *supplementary magic sets* [3], does not offer any significant improvement since the $\Omega(n^2)$ facts still have to be computed.

The counting method results to the following program:

```
c_path(0,a,red).
c_path(I+1,W,Color) ← c_path(I,X,Color),edge(X,Color,W).
path(I,Z) ← c_path(I,X,Color),edge(X,Color,Z).
path(I,Z) ← path(I+1,Z).
```

A bottom-up evaluation of this program computes the answers to the goal ← `path(0,Z)` in $O(n)$ time.

9.3.3. The odd-length colored-path program

Consider the cc-Datalog program that finds all the nodes that are accessible from node `a` through odd-length red paths (double recursion):

```
← path(a,red,Y).
path(X,C,Y) ← edge(X,C,Y).
path(X,C,Y) ← edge(X,C,Z), path(Z,C,R), path(R,C,Y).
```

together with the database: `edge(a,red,b`$_i$`)`, `edge(b`$_i$`,red,c)`, `edge(c,red,d`$_i$`)` and `edge(d`$_i$`,red, e`$_i$`)`, for $i = 1, ..., n$.

The target program after applying the proposed transformation and the appropriate unfoldings is:

```
← path⁻([],Z).
path₁⁺([],a).
path₂⁺([],red).
path⁻(L,Y) ← edge(X,C,Y), path₁⁺(L,X), path₂⁺(L,C).
path⁻(L,Y) ← path⁻([4|L],Y).
path₂⁺([4|L],C) ← #path₂⁺(L,C).
path₂⁺([3|L],C) ← #path₂⁺(L,C).
path₁⁺([4|L],R) ← path⁻([3|L],R).
path₁⁺([3|L],Z) ← edge(X,C,Z), path₁⁺(L,X), #path₂⁺(L,C).
```

or after the replacement of the choice predicate with its corresponding classical predicate and the elimination of $\texttt{path}_2^+$ predicate (in a similar way as in Subsection 9.3.2) we get:

```
← path⁻([],Z).
path₁⁺([],a).
path⁻(L,Y) ← edge(X,red,Y), path₁⁺(L,X).
path⁻(L,Y) ← path⁻([4|L],Y).
path₁⁺([4|L],R) ← path⁻([3|L],R).
path₁⁺([3|L],Z) ← edge(X,red,Z), path₁⁺(L,X).
```

The answers to the goal ($\texttt{Z = b}_i\texttt{, d}_i$, for $i = 1, ..., n$) are produced in $O(n)$ time (since only $O(n)$ facts are produced during the computation).

On the other hand, the corresponding magic sets program:

```
m_path(a,red).
m_path(Z,C) ← m_path(X,C), edge(X,C,Z).
m_path(R,C) ← m_path(X,C), edge(X,C,Z), path(Z,C,R).
path(X,C,Y) ← m_path(X,C), edge(X,C,Y).
path(X,C,Y) ← m_path(X,C), edge(X,C,Z), path(Z,C,R), path(R,C,Y).
```

needs $\Omega(n^2)$ time to produce the answers, due to the fact that it produces all facts of the form $\texttt{path}(\texttt{b}_\texttt{i}, \texttt{red}, \texttt{e}_\texttt{j})$, for $i, j = 1, ..., n$.

Similarly, the counting method produces the following program:

```
cnt_path(0,0,a,red).
cnt_path(I+1,2*H,Z,C) ← cnt_path(I,H,X,C), edge(X,C,Z).
cnt_path(I+1,2*H+1,R,C) ← cnt_path(I,H,X,C), edge(X,C,Z),
path(I+1,2*H,Z,C,R).
path(I,H/2,X,C,Y) ← cnt_path(I,H/2,X,C), edge(X,C,Y).
path(I,H/2,X,C,Y) ← cnt_path(I,H/2,X,C), edge(X,C,Z),
path(I+1,H,Z,C,R), path(I+1,H+1,R,C,Y).
```

So, the computation of the resulting program produces all facts of the form $\texttt{path}(1, 0, \texttt{b}_\texttt{i}, \texttt{red}, \texttt{e}_\texttt{j})$ for $i, j = 1, ..., n$. Hence, $\Omega(n^2)$ time is needed. Notice that the counting program requires two indices (instead of one that is needed in the case of the branching transformation).

9.3.4. The running example

Consider again the running example together with the database: $\texttt{e(a1,b}_i\texttt{)}$, $\texttt{f(b}_i\texttt{,c}_i\texttt{)}$ and $\texttt{g(b}_i\texttt{,c}_i\texttt{,d}_i\texttt{)}$, for $i = 1, ..., n$. The answers ($\texttt{Z} = \texttt{d}_\texttt{i}$, for $i = 1, ..., n$) for the target program given in Example 8.3 are produced in $O(n)$ time, since $O(n)$ atoms are produced.

The corresponding magic sets program is:

```
m_q(a1).
m_q(Y) ← m_q(X),e(X,Y).
q(X,Z) ← m_q(X),f(X,Z).
q(X,Z) ← m_q(X),e(X,Y),q(Y,W),g(Y,W,Z).
```

and needs $O(n)$ time.

Finally, the program produced by the counting method is:

```
cnt_q(0,a1).
cnt_q(I+1,Y) ← cnt_q(I,X),e(X,Y).
q(I,X,Z) ← cnt_q(I,X),f(X,Z).
q(I,X,Z) ← cnt_q(I,X),e(X,Y),q(I+1,Y,W),g(Y,W,Z).
```

and needs also $O(n)$ time to produce the answers.

9.3.5. Overall comparison

Recapitulating, the comparison of the three techniques is summarized in the following table:

$Method$	$Magic\,Sets$	$Counting\,method$	$Branching$
$Same\,generation$	$\Omega(n^2)$	$O(n)$	$O(n)$
$Colored-path$	$\Omega(n^2)$	$O(n)$	$O(n)$
$Odd-length\,colored-path$	$\Omega(n^2)$	$\Omega(n^2)$	$O(n)$
$Running\,example$	$O(n)$	$O(n)$	$O(n)$

As a closing comment, our transformation appears to have a similar behavior to that of counting for the programs we have presented (as well as for other programs we have tried). An advantage of the proposed technique is that it has a clear termination condition which is realistic for many examples. Finally, both the counting and the branching transformation appear to outperform the magic sets in many examples (but of course the latter is clearly more generally applicable).

10. Discussion

In this paper we have presented a significant extension of the branching-time transformation [18, 19]. More specifically, we have demonstrated that if we introduce choice predicates in the target language, then the branching-time approach can be extended to handle programs that allow multiple consumptive occurrences of variables in the bodies of clauses. The programs that result from the transformation have a number of interesting properties:

- Every clause in the resulting program corresponding to an IDB predicate in the source program, uses only a single data variable.

- Every IDB predicate in the resulting program is binary, its first argument actually being a control argument in the form of a list.

Moreover, in many interesting cases (e.g. for all source programs where all EDB predicates have arity 2), the target programs have at most one IDB body atom. Such programs are usually called *linear* [1, 2] and have many interesting properties.

We believe that the work presented in this paper can be extended in various ways. We have been investigating possible extensions of the source language, perhaps allowing more than one output arguments, as well as more general patterns of consumptions. In particular, we believe that the restriction to consecutive consumption of the arguments is not essential but lifting this restriction seems to require a more involved correctness proof. Another point for further research is the use of non-deterministic constructs that have been used for deductive databases, like those proposed in [8].

We currently have a working implementation of the initial branching-time transformation [27]. More specifically, the implementation of [27] takes as input Chain Datalog programs, translates them into Datalog$_{nS}$, optimizes them, and finally executes them in a bottom-up way. The implementation is written in Prolog and has given quite promising results for the case of Chain Datalog programs. We have recently undertaken the task of extending the implementation to handle cc-Datalog programs. This is obviously a more demanding goal since the existence of choice predicates in the target code imposes a different bottom-up execution strategy.

Another possible direction for future work would be to investigate whether the results of this paper can be applied to more general logic programs (i.e., programs that use function symbols). Consider for example the following program that performs multiplication of natural numbers:

```
times(X,Y,Z) ← zero_product(X,Y,Z).
times(s(X),Y,Z) ← times(X,Y,W),plus(W,Y,Z).
plus(X,Y,Z) ← zero_sum(X,Y,Z).
plus(X,s(Y),s(Z)) ← plus(X,Y,Z).
zero_product(0,X,0).
zero_sum(0,X,X).
```

Moreover, assume that we have a fixed goal, say ← `times(s(0),s(s(0)),Z)`. Then the above program can be transformed in the usual way. The key idea that allows us to do the translation is that although there exist function symbols, each compound term contains only one variable. The application of the transformation to such more general programs poses a whole new set of interesting questions.

In conclusion, we believe that the technique described in this paper has a significant potential for further extensions. Moreover, we believe that answers to the questions posed in this section will enable us to get further insight on the relationships between classical logic programming and intensional logic programming languages.

Acknowledgments

The authors would like to thank the anonymous reviewers for all their insightful comments and suggestions which have helped us to significantly improve our paper.

References

[1] Afrati, F., Gergatsoulis, M., Katzouraki, M.: On Transformations into Linear Database Logic Programs, *Perspectives of Systems Informatics (PSI'96), Proceedings* (D. Bjørner, M. Broy, I. Pottosin, Eds.), Lecture Notes in Computer Science (LNCS) 1181, Springer-Verlag, 1996.

[2] Afrati, F., Gergatsoulis, M., Toni, F.: Linearizability on Datalog Programs, *Theoretical Computer Science*, **308**(1 – 3), November 2003, 199–226.

[3] Beeri, C., Ramakrishnan, R.: On the power of magic, *The Journal of Logic Programming*, **10**(1,2,3 & 4), 1991, 255–299.

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

[5] Chomicki, J., Imielinski, T.: Finite representation of Infinite Query Answers, *ACM Transaction of Database Systems*, **18**(2), June 1993, 181–223.

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

[7] Giannotti, F., Greco, S., Saccà, D., Zaniolo, C.: Programming with Non-determinism in Deductive Databases, *Annals of Mathematics and Artificial Intelligence*, **19**(1–2), 1997, 97–125.

[8] Giannotti, F., Pedreschi, D., Zaniolo, C.: Semantics and Expressive Power of Non-deterministic Constructs in Deductive Databases, *Journal of Computer and Systems Sciences*, **62**(1), 2001, 15–42.

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

[10] Haddad, R. W., Naughton, J. F.: Counting Methods for Cyclic Relations, *Proc. 7th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, ACM Press, 1988.

[11] Lloyd, J. W.: *Foundations of Logic Programming*, 2nd edition, Springer-Verlag, 1987.

[12] Marchetti-Spaccamella, A., Pelaggi, A., Sacca, D.: Worst-case complexity analysis of methods for logic query implementation, *PODS '87: Proceedings of the sixth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, ACM Press, New York, USA, 1987, ISBN 0-89791-223-3.

[13] Orgun, M. A., Wadge, W. W.: Towards a unified theory of intensional logic programming, *The Journal of Logic Programming*, **13**(4), 1992, 413–440.

[14] Orgun, M. A., Wadge, W. W.: Extending Temporal Logic Programming with Choice Predicates Non-determinism, *Journal of Logic and Computation*, **4**(6), 1994, 877–903.

[15] Pettorossi, A., Proietti, M.: Transformation of Logic Programs, in: *Handbook of Logic in Artificial Intelligence and Logic Programming* (D. M. Gabbay, C. J. Hogger, J. A. Robinson, Eds.), vol. 5, Oxford University Press, 1997, 697–787.

[16] Potikas, P., Rondogiannis, P., Gergatsoulis, M.: A Transformation Technique for Datalog Programs based on Non-Deterministic Constructs, *Logic Based Program Synthesis and Transformation, 11th Int. Workshop, LOPSTR 2001, Paphos, Cyprus, November 28-30* (A. Pettorossi, Ed.), Lecture Notes in Computer Science (LNCS), Vol. 2372, Springer-Verlag, 2002.

[17] Ramakrishnan, R., Ullman, J. D.: A Survey of Deductive Database Systems, *The Journal of Logic Programming*, **23**(2), 1995, 125–149.

[18] Rondogiannis, P., Gergatsoulis, M.: The Intensional Implementation Technique for Chain Datalog Programs, *Proc. of the 11th International Symposium on Languages for Intensional Programming (ISLIP'98), May 7-9, Palo Alto, California, USA*, 1998.

[19] Rondogiannis, P., Gergatsoulis, M.: The Branching-Time Transformation Technique for Chain Datalog Programs, *Journal of Intelligent Information Systems*, **17**(1), 2001, 71–94.

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

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

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

[23] Sacca, D., Zaniolo, C.: Magic counting methods, *SIGMOD '87: Proceedings of the 1987 ACM SIGMOD international conference on Management of data*, ACM Press, New York, USA, 1987, ISBN 0-89791-236-5.

[24] Saccà, D., Zaniolo, C.: The generalized counting method for recursive logic queries, *Theoretical Computer Science*, **4**(4), 1988, 187–220.

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

[26] Tamaki, H., Sato, T.: Unfold/Fold Transformations of Logic Programs, *Proc. of the Second International Conference on Logic Programming* (S.-Å. Tarnlund, Ed.), 1984.

[27] Tsopanakis, K.: *Implementation and Evaluation of the Branching-Time Transformation for Chain Datalog Programs*, Diploma thesis, Dept. of Informatics and Telecommunications, University of Athens, Greece, 2003.

[28] Ullman, J. D.: *Principles of Database and Knowledge-Base Systems*, vol. I & II, Computer Science Press, 1989.

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

[30] Yaghi, A.: *The intensional implementation technique for functional languages*, Ph.D. Thesis, Dept. of Computer Science, University of Warwick, Coventry, UK, 1984.

Appendix A

In this Appendix we give the proof of Lemma 6.1.

Lemma 6.1. Let P be a simple cc-Datalog program, D a database, and P^* the Choice Datalog$_{nS}$ program that results from the transformation. For all predicates p defined in P_D, all $L \in List(\mathcal{N})$, all $a, b \in U_{P_D}$, all input positions i of p, and all approximations I, $p_i^+(L, b)$ does not depend on $p^-(L, a)$ in I.

Proof:
When the list L is empty it is straightforward to check that the lemma holds (as there is no clause in P^*, other than the unit clauses obtained from the transformation of the goal clause, whose head has as instance the atom $p_i^+([\,], b)$). We therefore need to examine the case where the length of L is greater than

or equal to 1. Assume that $L = [l_1|M]$. The proof is by induction on the distance between atoms in I. The cases for distances 0 and 1, hold trivially (the former because of Definitions 6.2 and 6.1 and the latter because of the form of the clauses produced by the transformation). Assume the lemma holds for distances up to k. We demonstrate the claim for distances $k+1$.

Assume for the sake of contradiction that there exists an approximation I in which $p_i^+([l_1|M], b)$ depends on $p^-([l_1|M], a)$ in I with distance $k+1$. This means that there is a derivation set S for $p_i^+([l_1|M], b)$ in I for which there exists a sequence of $k+1$ ground instances of clauses of P_D^* that establishes the dependency between these two atoms; the first clause in this sequence must contain $p^-([l_1|M], a)$ as a body atom, while the last clause must have $p_i^+([l_1|M], b)$ as its head. We distinguish the following cases:

Case 1: The first clause in the sequence is of the form:

$$q^-(M, a) \leftarrow\ p^-([l_1|M], a).$$

This is a ground instance of a clause that has resulted after transforming a clause of the original program that is either of the form:

$$q(\cdots) \leftarrow r(\cdots), p(\cdots)$$

or of the form:

$$q(\cdots) \leftarrow p(\cdots)$$

Now, in order to get from $q^-(M, a)$ to $p_i^+([l_1|M], b)$, the label l_1 must be restored by an intermediate clause of the sequence. This can be performed by either a ground instance of a clause of the form:

$$p_i^+([l_1|M], b) \leftarrow [\#]q_j^+(M, b).$$

or by:

$$p_i^+([l_1|M], b) \leftarrow\ r^-([l_2|M], b).$$

But in the first case this implies that $q_j^+(M, b)$ depends on $q^-(M, a)$ in I and the distance between $q_j^+(M, b)$ and $q^-(M, a)$ is $\leq k$ (contradiction from the induction hypothesis). In the second case, the only way that label l_2 can have been introduced is by a ground instance of a clause of the form:

$$r_n^+([l_2|M], d) \leftarrow [\#]q_m^+(M, d).$$

But then $q_m^+(M, d)$ depends on $q^-(M, a)$ in I, and the distance between $q_m^+(M, d)$ and $q^-(M, a)$ is $\leq k$ (contradiction from the induction hypothesis).

Case 2: The first clause in the sequence is of the form:

$$q_j^+([l_2|M], a) \leftarrow\ p^-([l_1|M], a).$$

This is a ground instance of a clause that has resulted after transforming a clause of the original program that is of the form:

$$r(\cdots) \leftarrow p(\cdots), q(\cdots)$$

Now, in order to get from $q_j^+([l_2|M],a)$ to $p_i^+([l_1|M],b)$, the label l_1 must be restored by an intermediate clause of the sequence. This can only be performed by a ground instance of a clause of the form:

$$p_i^+([l_1|M],d) \leftarrow [\#]r_n^+(M,d).$$

But in order to get from $q_j^+([l_2|M],a)$ to $r_n^+(M,d)$ the label l_2 must be removed, and this can only be performed by a ground instance of a clause of the form:

$$r^-(M,c) \leftarrow \ q^-([l_2|M],c).$$

Then $r_n^+(M,d)$ depends on $r^-(M,c)$ in I and the distance between them is $\leq k$ (contradiction by the induction hypothesis). □

Appendix B

In this Appendix we give the proof of Lemma 6.2.

Lemma 6.2. Let P be a simple cc-Datalog program and D a database. Let P^* be the Choice Datalog$_{nS}$ program that results from the transformation. For all predicates p defined in P_D, all $L \in List(\mathcal{N})$, all $a, b \in U_{P_D}$, and all input positions i, j of p, there does not exist any approximation I to any limit interpretation of P_D^* such that $p_i^+(L,b)$ depends on $p_j^+(L,a)$, where $p_i^+(L,b) \neq p_j^+(L,a)$.

Proof:
When the list L is empty it is straightforward to check that the lemma holds. We therefore need to examine the case where the length of L is greater than or equal to 1. Assume that $L = [l_1|M]$. The proof is by induction on the distance between atoms. The cases for distances 0 and 1, hold trivially (the former because of Definitions 6.2 and 6.1 and the latter because of the form of the clauses produced by the transformation). Assume the lemma holds for distances up to k. We demonstrate the claim for distances $k+1$.

Assume for the sake of contradiction that there exists an approximation I such that $p_i^+([l_1|M],b)$ depends on $p_j^+([l_1|M],a)$ in I with distance $k+1$. This means that there exists a sequence of $k+1$ ground instances of clauses of P_D^* that establishes the dependency between these two atoms; the first clause in this sequence must contain $p_j^+([l_1|M],a)$ as a body atom, while the last clause must have $p_i^+([l_1|M],b)$ as its head. We distinguish the following cases:
Case 1: The last clause in the sequence is of the form:

$$p_i^+([l_1|M],b) \leftarrow [\#]q_k^+(M,b).$$

This is a ground instance of a clause that has resulted after transforming a clause of the original program that is either of the form:

$$q(\cdots) \leftarrow p(\cdots).$$

or:

$$q(\cdots) \leftarrow r(\cdots), p(\cdots).$$

or

$$q(\cdots) \leftarrow p(\cdots), r(\cdots).$$

Now, in order to get from $p_j^+([l_1|M], a)$ to $q_k^+(M, b)$, the label l_1 must be removed by an intermediate clause of the sequence. One way that this can have been performed is by using a ground instance of a clause of the form:

$$q^-(M, c) \leftarrow\ p^-([l_1|M], c).$$

But then $q_k^+(M, b)$ would depend on $q^-(M, c)$ in I (contradiction by Lemma 6.1).

The removal of l_1 can have alternatively been performed by using a ground instance of a clause of the form:

$$r_m^+([l_2|M], c) \leftarrow\ p^-([l_1|M], c).$$

But then $p_i^+([l_1|M], b)$ would depend on $p^-([l_1|M], c)$ in I (contradiction by Lemma 6.1).

Case 2: The last clause in the sequence is of the form:

$$p_i^+([l_1|M], b) \leftarrow\ q^-([l_2|M], b).$$

This is a ground instance of a clause that has resulted after transforming a clause of the original program that is of the form:

$$r(\cdots) \leftarrow q(\cdots), p(\cdots).$$

But then there must have existed some intermediate step in the whole derivation that removed label l_1 from the initial list $[l_1|M]$ of the atom $p_j^+([l_1|M], a)$. This can have only been performed by using a ground instance of a clause of the form:

$$r^-(M, c) \leftarrow\ p^-([l_1|M], c).$$

In this case $p_i^+([l_1|M], b)$ would depend on $p^-([l_1|M], c)$ in I (contradiction from Lemma 6.1). □