Using Branching-Time Logic to Optimize an Extended Class of Datalog Queries*

Petros Potikas[1,3], Manolis Gergatsoulis[2], and Panos Rondogiannis[3]

[1] Department of Electrical and Computer Engineering,
National Technical University of Athens, 157 73 Zografou, Athens, Greece
ppotik@cs.ntua.gr

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

[3] Department of Informatics & Telecommunications,
University of Athens, Panepistimiopolis, 157 84 Athens, Greece
prondo@di.uoa.gr

Abstract. We propose an extension of the *branching-time transformation* [10] which can handle a significantly broader class of Datalog programs. The initial transformation could only be applied to Chain Datalog, a useful but restricted class of programs. In this paper we demonstrate that the transformation of [10] can be extended to handle all Datalog programs that do not allow multiple consumptions of variables in clauses. We demonstrate the correctness of the new transformation and provide certain optimizations that further improve the programs obtained by the transformation.

Keywords: Deductive Databases, Temporal Logic Programming.

1 Intoduction

The work presented in this paper contributes to the area of *value-passing* 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 have a long-standing tradition in the area of deductive databases (as examples we should cite the *counting transformation* [14], the *magic sets* [1,15], the *pushdown approach* [4], and so on). Recently, a technique that contributes to this stream of research has been proposed: the *branching-time transformation* [10] uses ideas from temporal logic programming in order to optimize Chain Datalog programs. The branching-time transformation has its roots in the area of functional programming where a similar idea has been developed and used as an implementation technique for functional languages [19,18,12,13].

The technique of [10] applies to Chain Datalog programs, a subset of Datalog which has found many uses in deductive databases. The syntax of Chain Datalog

* This work has been partially supported by the University of Athens under the project "Extensions of the Logic Programming Paradigm" (grant no. 70/4/5827).

programs is somewhat restrictive in the sense that the value of a variable that is produced in an atom must be consumed immediately in the next atom in the clause; moreover, each atom has exactly two variables (one input and one output). In this paper we raise these restrictions. More specifically, we allow clauses in which variables that are produced in an atom must be consumed in any subsequent atom in the clause; moreover, each atom may have many variables (and not just two). This new class is obviously a superset of Chain Datalog and it allows more freedom in the creation of more demanding queries.

The contributions of the paper can be summarized as follows:

- We demonstrate how the branching-time transformation can be extended to apply to the class of productive-consumptive Datalog programs. This is a much broader class than that of Chain Datalog programs and can certainly allow more interesting queries to be expressed.
- The new transformation is equally simple as the one of [10]. It uses the same target language and the programs obtained have the same desirable properties as the ones produced by [10]. It should be noted that if one allows multiple consumptive occurrences of variables, then the target language should be enriched with additional constructs (as we demonstrate in [8]).

2 The Source Language of the Transformation

In the following, we assume a familiarity with the basic concepts behind deductive databases [9] and logic programming [5].

A *Datalog program* P is a finite set of function-free Horn rules. Predicates appearing in the head of some rule in P are called *IDB predicates*, while those appearing only in the rule bodies are called *EDB predicates*. A set D of ground unit clauses defining the EDBs is called a *database*. If P is a Datalog program and D a database then by P_D we denote the program P along with the database D. The least Herbrand model of P_D is denoted by $M(P_D)$ while $M(p, P_D)$ denotes the subset of $M(P_D)$ containing all atoms whose predicate is p. We also use the following notation: *constants* are denoted by a, b, c; *variables* by uppercase letters such as X, Y, Z and vectors of variables by $\boldsymbol{v}$; predicates by lower case letters such as p, q, r; also subscripted versions of the above symbols will be used. The source language of the transformation is defined bellow:

Definition 1. *A clause*

$$p_0(\boldsymbol{v}_0, Z_n) \leftarrow p_1(\boldsymbol{v}_1, Z_1), p_2(\boldsymbol{v}_2, Z_2), \ldots, p_n(\boldsymbol{v}_n, Z_n).$$

with $n > 0$, is called productive-consumptive clause *(or* pc-clause *for short) if:*

1. *Each $\boldsymbol{v}_i$, for $i = 0, \ldots, n$ is a nonempty vector containing distinct variables; moreover $Z_1, \ldots, Z_n$ are distinct variables.*
2. *$vars(\boldsymbol{v}_i) \subseteq vars(\boldsymbol{v}_0) \cup \{Z_1, \ldots, Z_{i-1}\}$, for $1 \leq i \leq n$.*
3. *for every $V \in vars(\boldsymbol{v}_0)$ there exists exactly one vector $\boldsymbol{v}_i, 1 \leq i \leq n$ such that $V \in vars(\boldsymbol{v}_i)$.*
4. *for every Z_i with $1 \leq i \leq n-1$, there exists exactly one vector $\boldsymbol{v}_j$ with $i < j \leq n$ such that $Z_i \in vars(\boldsymbol{v}_j)$.*

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

It should be mentioned here that pc-clauses are moded. More specifically, we assume that each predicate has only one mode, i.e. each argument position is used either as input or as output, but not both. In particular the terms $\boldsymbol{v}_i$ of the above definition correspond to input arguments, while each Z_i corresponds to the single output argument of each atom.

Example 1. The following clause is a pc-clause:

$$\mathtt{p}(\overset{+}{\mathtt{X}}, \overset{+}{\mathtt{Y}}, \overset{-}{\mathtt{Z}}) \leftarrow \mathtt{q}(\overset{+}{\mathtt{Y}}, \overset{-}{\mathtt{W}}), \mathtt{r}(\overset{+}{\mathtt{X}}, \overset{-}{\mathtt{R}}), \mathtt{s}(\overset{+}{\mathtt{W}}, \overset{+}{\mathtt{R}}, \overset{-}{\mathtt{Z}}).$$

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

Definition 2. *An occurrence of a variable in an input argument of the head or in the output argument of an atom in the body of a clause will be called* productive*; otherwise it will be called* consumptive*.*

The intuition behind the class of pc-Datalog programs is that each value produced by an atom must be consumed in exactly one atom following (not necessarily immediately) the atom that produced it (except for the production of the last atom which is returned to the head atom). Thus, each variable appears exactly twice in a pc-clause. Many natural Datalog programs belong to this class; for example, the class of Chain Datalog programs is a proper subset of this class.

Definition 3. *A* simple pc-Datalog program *is a pc-Datalog program in which every clause has at most two atoms in its body.*

The following proposition (which can be proved easily using unfold/fold transformations [2, 7]) establishes the equivalence between pc-Datalog programs and simple pc-Datalog ones.

Proposition 1. *Every pc-Datalog program P can be transformed into a simple pc-Datalog program P' such that for every predicate symbol p appearing in P and for every database D, $M(p, P_D) = M(p, P'_D)$.*

Example 2. Consider the following pc-Datalog program P:

```
(1)  p(X,Y,Z) ← e(X,W),p(W,Y,R),f(R,Z).
(2)  p(X,Y,Z) ← g(X,Y,Z).
```

The corresponding simple pc-Datalog program P' is:

```
(1')  p(X,Y,Z) ← e(X,W),q(W,Y,Z).
(E)   q(W,Y,Z) ← p(W,Y,R),f(R,Z).
(2)   p(X,Y,Z) ← g(X,Y,Z).
```

P' has been obtained from P by introducing a new definition (clause E) and then folding clause 1 using E to obtain $1'$.

Since by Proposition 1, for every pc-Datalog program we can obtain an equivalent simple pc-Datalog program, for practical reasons we define the transformation algorithm on simple pc-Datalog programs.

3 The Target Language of the Transformation

The target language of the transformation is *Branching Datalog* which is a temporal logic programming language that supports a branching notion of time. This formalism has its roots in the Chronolog [17, 6] and Cactus [11] temporal logic programming languages. In particular, Branching Datalog programs are Cactus programs without function symbols. Every atom in a Branching Datalog program is preceded by a *temporal reference*, which is a (possibly empty) sequence of the temporal operators $first$ and $next_i$, $i \geq 0$. A temporal reference of the form $first\ next_{i_1} \cdots next_{i_k}$, where $k \geq 0$, is called *canonical*. A temporal reference of the form $next_{i_1} \cdots next_{i_k}$ is said to be *open*. A *temporal atom* is an atom preceded by either a canonical or an open temporal reference. A *canonical* (resp. *open*) temporal atom is a temporal atom whose temporal reference is canonical (resp. open). A *goal* in Branching Datalog is of the form $\leftarrow A$, where A is either a canonical temporal atom or an open one. A *temporal clause* in Branching Datalog is a formula of the form:

$$H \leftarrow A_1, \ldots, A_n.$$

where $H, A_1, \ldots, A_n$ are temporal atoms and $n \geq 0$. If $n = 0$, the clause is said to be a *unit temporal clause*. A Branching Datalog program is a finite set of temporal clauses. A *canonical temporal clause* is a temporal clause in which all atoms that occur in it are canonical. A *canonical temporal instance* of a temporal clause C is a canonical temporal clause which is obtained by applying the same canonical temporal reference to all open atoms of C.

Branching Datalog is based on a relatively simple *branching-time logic* (BTL). In BTL time has an initial moment and flows towards the future in a tree-like way. The set of moments in time can be modeled by the set $List(\omega)$ of lists of natural numbers. The empty list $[\,]$ corresponds to the beginning of time and the list $[i|t]$ (that is, the list with head i, where $i \in \omega$, and tail t) corresponds to the i-th alternative successor of the moment identified by the list t. BTL uses the temporal operators $first$ and $next_i$, $i \in \omega$. The operator $first$ is used to identify the first moment in time, while $next_i$ refers to the i-th alternative successor of the current moment in time. The syntax of BTL extends the syntax of first-order logic with two formation rules: if A is a formula then so are $first\ A$ and $next_i\ A$. The semantics of temporal formulas of BTL are given using the notion of *branching temporal interpretation* [11]:

Definition 4. *A* branching temporal interpretation *or simply a* temporal interpretation *I of BTL comprises a non-empty set D, called the domain of the interpretation, together with an element of D for each variable or constant symbol and an element of $[List(\omega) \to 2^{D^n}]$ for each n-ary predicate symbol.*

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

Definition 5. *The semantics of the elements of the temporal logic BTL are given recursively as follows:*

1. *For any n-ary predicate symbol p and terms $e_0, \ldots, e_{n-1}$,* $\models_{I,t} p(e_0, \ldots, e_{n-1})$ *iff* $\langle I(e_0), \ldots, I(e_{n-1})\rangle \in I(p)(t)$
2. $\models_{I,t} \neg A$ *iff it is not the case that* $\models_{I,t} A$
3. $\models_{I,t} A \wedge B$ *iff* $\models_{I,t} A$ *and* $\models_{I,t} B$
4. $\models_{I,t} (\forall x)A$ *iff* $\models_{I[d/x],t} A$ *for all $d \in D$, where the interpretation $I[d/x]$ is the same as I except that the variable x is assigned the element d.*
5. $\models_{I,t}$ *first* A *iff* $\models_{I,[\,]} A$
6. $\models_{I,t}$ *next*$_i$ A *iff* $\models_{I,[i|t]} A$

If a formula A is true in a temporal interpretation I at all moments in time, it is said to be true in I (we write $\models_I A$) and I is called a *model* of A.

3.1 Semantics of Branching Datalog

When we focus on Branching Datalog programs, the interpretations we consider are Herbrand ones. As usual, the *Herbrand universe* U_{P_D} of a program P together with a database D is the set of all constant symbols that appear in P_D. *Temporal Herbrand interpretations* can be regarded as subsets of the *temporal Herbrand base* B_{P_D} of P_D, consisting of all *canonical ground temporal atoms* whose predicate symbols appear in P_D and whose arguments are terms in the Herbrand universe U_{P_D} of P_D. In particular, given a subset H of B_{P_D}, we can define a temporal Herbrand interpretation I by the following:

$$\langle c_0, \ldots, c_{n-1}\rangle \in I(p)([i_1, \ldots, i_k]) \ \textit{iff} \quad \textit{first next}_{i_k} \cdots \textit{next}_{i_1}\, p(c_0, \ldots, c_{n-1}) \in H$$

A *temporal Herbrand model* is a temporal Herbrand interpretation which is a model of the program. In the rest of the paper, when we refer to a "model of a program" we always mean a temporal Herbrand model.

Many interesting results of classical logic programming can be easily extended to hold for Branching Datalog. The following theorem states that the least Herbrand model of P_D consists of all canonical ground temporal atoms which are logical consequences of P_D.

Theorem 1. *Let P be a Branching Datalog program and D a database. Then* $M(P_D) = \{A \in B_{P_D} \mid P_D \models A\}$

Next, we define a fixpoint operator for Branching Datalog.

Definition 6. *Let P be a Branching Datalog program and D a database. The operator $T_{P_D} : 2^{B_{P_D}} \to 2^{B_{P_D}}$ is defined as follows: if I is a temporal Herbrand interpretation in $2^{T_{P_D}}$ then $T_{P_D}(I) = \{A \mid A \leftarrow B_1, ..., B_n$ is a canonical ground instance of a program clause in P_D and $\{B_1, ..., B_n\} \subseteq I\ \}$.*

It is easy to prove that T_{P_D} is continuous and monotonic, therefore it provides a characterization of the least Herbrand model of Branching Datalog programs.

Theorem 2. *Let P be a Branching Datalog program and D a database. Then $M(P_D) = lfp(T_{P_D}) = T_{P_D} \uparrow \omega$.*

4 The Transformation Algorithm

In this section we define formally the transformation algorithm.

The algorithm: Let P be a simple pc-Datalog program and G a goal clause. For each $(n+1)$-ary predicate p in P, we introduce $n+1$ unary 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 transformation processes the goal clause G and each clause in P and gives as output a new goal clause G^* and a Branching Datalog program P^*. When processing a clause in P, the algorithm introduces branching-time operators of the form next_i, $i \in \omega$. This can be done, by assigning to each body atom in P a different natural number. Then, if i is the index assigned to an atom, next_i is the operator corresponding to that atom. *The operators introduced in this way for a given clause are guaranteed to be different than the operators introduced for any other clause in P.*

The algorithm processes the program and goal clauses in the following way:

Case 1: Let C be a clause of the form:

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

and let $next_i, next_j$ be the branching-time operators of $q(\boldsymbol{v}_1, Y)$ and $r(\boldsymbol{v}_2, Z)$, respectively. Then C is transformed in the following way:

a) The following clause is added to P^*:

$$p^-(Z) \leftarrow next_j \ r^-(Z).$$

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

$$next_i \ q_m^+(X) \leftarrow p_k^+(X).$$

Variables appearing in both $\boldsymbol{v}_0$ and $\boldsymbol{v}_2$, are treated analogously.

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

$$next_j \ r_m^+(Y) \leftarrow next_i \ q^-(Y).$$

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

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

and let $next_i$ be the branching-time operator of $q(\boldsymbol{v}_1, Z)$. Then C is transformed as follows:

a) The following clause is added to P^*:

$$p^-(Z) \leftarrow next_i \ q^-(Z).$$

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

$$next_i \ q_m^+(X) \leftarrow p_k^+(X).$$

Case 3: For every (n+1)-ary EDB predicate p of P a new clause of the following form is added to P^*:

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

Case 4: 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^*:

$$first\ \ p_i^+(a_i).$$

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

$$\leftarrow first\ p^-(Y).$$

Example 3. Consider the program P' obtained in Example 2 together with the goal clause (G), where the predicates e, f and g are EDBs.

$$(G): \qquad \leftarrow \mathtt{p(a,1,Y)}.$$

Transforming $P' \cup \{G\}$ we get the new goal clause G^*:

$$\leftarrow \mathtt{first}\ \ \mathtt{p^-(Y)}.$$

and the program P^*:

$$\begin{array}{l}
\mathtt{first}\ \mathtt{p_1^+(a)}.\\
\mathtt{first}\ \mathtt{p_2^+(1)}.\\
\mathtt{p^-(Z)} \leftarrow \mathtt{next_2}\ \mathtt{q^-(Z)}.\\
\mathtt{next_2}\ \mathtt{q_2^+(Y)} \leftarrow \mathtt{p_2^+(Y)}.\\
\mathtt{next_2}\ \mathtt{q_1^+(W)} \leftarrow \mathtt{next_1}\ \mathtt{e^-(W)}.\\
\mathtt{next_1}\ \mathtt{e_1^+(X)} \leftarrow \mathtt{p_1^+(X)}.\\
\mathtt{q^-(Z)} \leftarrow \mathtt{next_4}\ \mathtt{f^-(Z)}.\\
\mathtt{next_4}\ \mathtt{f_1^+(R)} \leftarrow \mathtt{next_3}\ \mathtt{p^-(R)}.\\
\mathtt{next_3}\ \mathtt{p_2^+(Y)} \leftarrow \mathtt{q_2^+(Y)}.\\
\mathtt{next_3}\ \mathtt{p_1^+(W)} \leftarrow \mathtt{q_1^+(W)}.\\
\mathtt{p^-(Z)} \leftarrow \mathtt{next_5}\ \mathtt{g^-(Z)}.\\
\mathtt{next_5}\ \mathtt{g_2^+(Y)} \leftarrow \mathtt{p_2^+(Y)}.\\
\mathtt{next_5}\ \mathtt{g_1^+(X)} \leftarrow \mathtt{p_1^+(X)}.\\
\mathtt{e^-(Y)} \leftarrow \mathtt{e(X,Y)}, \mathtt{e_1^+(X)}.\\
\mathtt{f^-(Y)} \leftarrow \mathtt{f(X,Y)}, \mathtt{f_1^+(X)}.\\
\mathtt{g^-(Z)} \leftarrow \mathtt{g(X,Y,Z)}, \mathtt{g_1^+(X)}, \mathtt{g_2^+(Y)}.
\end{array}$$

The correctness of the transformation algorithm is established by the following theorem.

Theorem 3. *Let P be a simple pc-Datalog program, D a database and $\leftarrow p(a_1, \ldots, a_n, Y)$ be a goal clause. Let P^* be the Branching Datalog program obtained by applying the transformation algorithm to $P \cup \{\leftarrow p(a_1,\ldots,a_n,Y)\}$. Then $first\ p^-(b) \in T_{P_D^*} \uparrow \omega$ iff $p(a_1,\ldots,a_n,b) \in T_{P_D} \uparrow \omega$.*

5 Refinements of the Transformation

We now demonstrate that the target program obtained by the above transformation technique can be further optimized by taking into account specific characteristics of the source and target programs. The optimizations presented in this section are similar in spirit to the ones obtained in [10].

(a) All unary predicates in the resulting program that correspond to EDB predicates of the original program (and the clauses defining them in P^*) can be eliminated using unfolding [3]. For more details see [10].

Example 4. Applying this improvement to P^* of Example 3 we get:

$$\begin{array}{l}
\leftarrow \texttt{first}\ \texttt{p}^-(\texttt{Y}).\\
\texttt{first}\ \texttt{p}_1^+(\texttt{a}).\\
\texttt{first}\ \texttt{p}_2^+(\texttt{1}).\\
\texttt{p}^-(\texttt{Z}) \leftarrow \texttt{next}_2\ \texttt{q}^-(\texttt{Z}).\\
\texttt{next}_2\ \texttt{q}_2^+(\texttt{Y}) \leftarrow \texttt{p}_2^+(\texttt{Y}).\\
\texttt{next}_2\ \texttt{q}_1^+(\texttt{W}) \leftarrow \texttt{e}(\texttt{X},\texttt{W}), \texttt{p}_1^+(\texttt{X}).\\
\texttt{q}^-(\texttt{Z}) \leftarrow \texttt{f}(\texttt{R},\texttt{Z}), \texttt{next}_3\ \texttt{p}^-(\texttt{R}).\\
\texttt{next}_3\ \texttt{p}_2^+(\texttt{Y}) \leftarrow \texttt{q}_2^+(\texttt{Y}).\\
\texttt{next}_3\ \texttt{p}_1^+(\texttt{W}) \leftarrow \texttt{q}_1^+(\texttt{W}).\\
\texttt{p}^-(\texttt{Z}) \leftarrow \texttt{g}(\texttt{X},\texttt{Y},\texttt{Z}), \texttt{p}_1^+(\texttt{X}), \texttt{p}_2^+(\texttt{Y}).
\end{array}$$

(b) In the presentation of the transformation we assumed that each body atom has been assigned a different *next* operator. However, it can be easily seen that some of these operators may be redundant. In particular, it is not necessary to assign a *next* operator to atoms whose predicate symbol appears only once in the body atoms of $P \cup \{G\}$. We therefore can reformulate Cases 1 and 2 of the transformation algorithm as follows:
Case 1′: From a clause of the form:

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

instead of the clauses in a), b) and c) of the Case 1 of the transformation algorithm, we now get the clauses of the following form, respectively:

$$\begin{array}{l}
p^-(Z) \leftarrow Op_j\ r^-(Z).\\
Op_i\ q_m^+(X) \leftarrow p_k^+(X).\\
Op_j\ r_m^+(Y) \leftarrow Op_i\ q^-(Y).
\end{array}$$

where Op_j is $next_j$ if there is another body atom in $P \cup \{G\}$, with the same predicate symbol r, otherwise Op_j is empty. Similar for Op_i. Case 2 of the algorithm can be redefined in a similar way.

Example 5. If we apply the above to Example 4, we get the program that follows in which all temporal operators, except the operator $next_3$, have been eliminated. Notice that $next_3$ cannot be eliminated since the corresponding predicate p appears twice in $P \cup \{G\}$:

$$\begin{array}{l}
\leftarrow \texttt{first}\ \texttt{p}^-(\texttt{Y}).\\
\texttt{first}\ \texttt{p}_1^+(\texttt{a}).\\
\texttt{first}\ \texttt{p}_2^+(\texttt{1}).\\
\texttt{p}^-(\texttt{Z}) \leftarrow \texttt{q}^-(\texttt{Z}).
\end{array}$$

$$\begin{array}{l}
\mathtt{q}_2^+(\mathtt{Y}) \leftarrow \mathtt{p}_2^+(\mathtt{Y}).\\
\mathtt{q}_1^+(\mathtt{W}) \leftarrow \mathtt{e}(\mathtt{X},\mathtt{W}), \mathtt{p}_1^+(\mathtt{X}).\\
\mathtt{q}^-(\mathtt{Z}) \leftarrow \mathtt{f}(\mathtt{R},\mathtt{Z}), \mathtt{next}_3\ \mathtt{p}^-(\mathtt{R}).\\
\mathtt{next}_3\ \mathtt{p}_2^+(\mathtt{Y}) \leftarrow \mathtt{q}_2^+(\mathtt{Y}).\\
\mathtt{next}_3\ \mathtt{p}_1^+(\mathtt{W}) \leftarrow \mathtt{q}_1^+(\mathtt{W}).\\
\mathtt{p}^-(\mathtt{Z}) \leftarrow \mathtt{g}(\mathtt{X},\mathtt{Y},\mathtt{Z}), \mathtt{p}_1^+(\mathtt{X}), \mathtt{p}_2^+(\mathtt{Y}).
\end{array}$$

(c) In the branching-time transformation for Chain Datalog programs presented in [10] it was demonstrated that temporal operators that correspond to left recursive calls can be eliminated. This is not generally the case for the present transformation. We can only eliminate such operators when they result from the transformation of a clause of the following form:

$$p(X_1, X_2, ..., X_n, Z) \leftarrow p(X_1, X_2, ..., X_n, Y), q(Y, Z).$$

i.e. when the head and the first body atom of the clause have the same variables in the same input argument positions. Unfortunately, in the more general case, where the occurrences of the input variables of `p` get "shuffled", this optimization can not be applied. For example, if we have the clause:

```
p(X,Y,Z) ← p(Y,X,W),f(W,Z).
```

the temporal operator corresponding to `p(Y,X,W)` cannot be eliminated.

The correctness of all the above refinements can be easily established by appropriately adapting the correctness proof of the transformation.

It is easy to verify that there are certain subclasses of pc-Datalog programs for which our method produces classical unary Datalog programs (i.e. programs that do not contain any temporal operators). One such subclass contains each non-recursive pc-Datalog program P in which each IDB predicate symbol appears only once in the bodies of the program clauses in $P \cup \{G\}$ where G is the goal clause. Notice that the occurrences of the EDB predicates in P do not affect this property as all operators introduced for these atoms, along with the corresponding IDB predicates are eliminated by applying the refinement (a).

6 Conclusions

This paper presents an extension of the branching-time transformation to the class of productive-consumptive Datalog programs. The new transformation preserves all the desirable characteristics of its predecessor while extending significantly the class of allowable Datalog queries. Recently, the transformation of [10] was implemented and evaluated [16]. We plan to extend the implementation to the broader class presented in this paper. Moreover, we would like to investigate further optimizations that would enhance the performance of the bottom-up evaluation of the target program.

References

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

2. M. Gergatsoulis and M. Katzouraki. Unfold/fold transformations for definite clause programs. In M. Hermenegildo and J. Penjam, editors, *Programming Language Implementation and Logic Programming (PLILP'94), Proceedings*, Lecture Notes in Computer Science (LNCS) 844, pages 340–354. Springer-Verlag, 1994.
3. M. Gergatsoulis and C. Spyropoulos. Transformation techniques for branching-time logic programs. In W. W. Wadge, editor, *Proc. of the 11th International Symposium on Languages for Intensional Programming (ISLIP'98), May 7-9, Palo Alto, California, USA*, pages 48–63, 1998.
4. S. Greco, D. Saccà, and C. Zaniolo. Grammars and automata to optimize chain logic queries. *International Journal on Foundations of Computer Science*, 10(3):349–372, 1999.
5. J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1987.
6. M. A. Orgun and W. W. Wadge. Towards a unified theory of intensional logic programming. *The Journal of Logic Programming*, 13(4):413–440, 1992.
7. A. Pettorossi and M. Proietti. Transformation of logic programs. In D. M. Gabbay, C. J. Hogger, and J. A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 5, pages 697–787. Oxford University Press, 1997.
8. P. Potikas, P. Rondogiannis, and M. Gergatsoulis. A Transformation Technique for Datalog Programs based on Non-Deterministic Constructs. In A. Pettorossi, editor, *Logic Based Program Synthesis and Transformation, 11th Int. Workshop, LOPSTR 2001, Paphos, Cyprus, November 28-30*, Lecture Notes in Computer Science (LNCS), Vol. 2372, pages 25–45. Springer-Verlag, 2002.
9. R. Ramakrishnan and J. D. Ullman. A survey of deductive database systems. *The Journal of Logic Programming*, 23(2):125–149, 1995.
10. P. Rondogiannis and M. Gergatsoulis. The branching-time transformation technique for chain datalog programs. *Journal of Intelligent Information Systems*, 17(1):71–94, 2001.
11. P. Rondogiannis, M. Gergatsoulis, and T. Panayiotopoulos. Branching-time logic programming: The language Cactus and its applications. *Computer Languages*, 24(3):155–178, October 1998.
12. P. Rondogiannis and W. W. Wadge. First-order functional languages and intensional logic. *Journal of Functional Programming*, 7(1):73–101, 1997.
13. P. Rondogiannis and W. W. Wadge. Higher-Order Functional Languages and Intensional Logic. *Journal of Functional Programming*, 9(5):527–564, 1999.
14. D. Saccà and C. Zaniolo. The generalized counting method for recursive logic queries. *Theoretical Computer Science*, 4(4):187–220, 1988.
15. S. Sippu and E. Soisalon-Soininen. An analysis of magic sets and related optimization strategies for logic queries. *Journal of the ACM*, 43(6):1046–1088, 1996.
16. K. Tsopanakis. Implementation and evaluation of the branching-time transformation for chain datalog programs. Diploma thesis, Dept. of Informatics and Telecommunications, University of Athens, Greece, 2003.
17. W. W. Wadge. Tense logic programming: A respectable alternative. In *Proc. of the 1988 International Symposium on Lucid and Intensional Programming*, pages 26–32, 1988.
18. W. W. Wadge. Higher-Order Lucid. In *Proceedings of the Fourth International Symposium on Lucid and Intensional Programming*, 1991.
19. A. Yaghi. *The intensional implementation technique for functional languages*. PhD thesis, Dept. of Computer Science, University of Warwick, Coventry, UK, 1984.