

# DDB trees: a basis for deductive database explanations

Sarah Mallet and Mireille Ducassé

IRISA/INSA  
Campus Universitaire de Beaulieu  
F - 35042 Rennes Cedex, France  
{smallet, ducasse}@irisa.fr

Keywords: debugging, explanations, deductive databases, logic programming

## Abstract

The power of deductive systems in general is that programs express what should be done and not how it should be done. Nevertheless, deductive systems need debugging and explanation facilities. Indeed, their operational semantics is less abstract than the declarative semantics of the programs. If users have to understand all the low level details of the operational semantics much of the benefits of using a deductive system is lost.

Existing explanation systems for deductive databases produce proof trees to be shown to users. Although useful, proof trees give a fragmented view of query evaluations, and users face a, most of the time large, forest of proof trees.

We propose a new data structure, called the DDB tree, which merges the information of a proof tree forest into one concise tree. A DDB tree gives a global picture of a query evaluation in a dramatically reduced structure with no loss of information. DDB trees can be shown to users or can be analyzed further by an explanation system.

## 1 Introduction

Deductive databases are based on a logical data model, in contrast with relational data models of more classical databases. Roughly speaking, a deductive database can be seen as a relational database (referred to as the database in the rest of the introduction) which contains facts, together with a deductive engine which can deduce new facts from the actual ones and a set of rules. A significant advantage of deductive databases is that they allow the full power of recursivity to be enacted [9].

The advantage of deductive systems in general is that programs express **what** should be done and not **how** it should be done. Hence people face more abstract programs and can better concentrate on the essential features of their applications. Nevertheless, deductive systems need debugging and explanation facilities. Indeed, the deductive engine works with an opera-

tional semantics less abstract than the declarative semantics of the programs. If users of deductive systems have to understand all the low level details of the operational semantics much of the benefits of using a deductive system is lost.

The existing explanation systems for deductive databases, proposed by Wieland [15], Specht[12] and Arora and al.[1], produce proof trees to be shown to users. A proof tree recapitulates the steps which enabled to deduce a given fact, showing the rules and the database facts used in the process.

To our point of view, proof trees provide useful information, but they are not quite accurate. First of all, they give a fragmented view of the accesses to the database, pretending that data is retrieved from the database one piece at a time. Furthermore, for each deduced fact a proof tree has to be shown, whereas much of the deduction can be common between several deduced facts. Thus people face a forest of proof trees, which can be very large and not easy to analyze.

Moreover, the already cited systems present proof trees as the end result of an explanation system. To our point of view there is much more analysis to be done by a system before showing information to users.

Our ultimate aim is to build a “trace” analyzer for deductive database systems based on the same principles as Opium, a trace analyzer for Prolog [3]. In Opium, a trace models a Prolog execution and is a representation of an SLD tree.

For deductive databases, as discussed earlier, proof trees are not satisfactory to model query evaluations. Hence we defined a new data structure, called the DDB tree. A DDB tree is similar to an SLD tree, where all branches lead to a success (there is no backtracking in deductive database evaluations) and *substitution sets* represent the accesses to the database.

The main advantage of DDB trees over proof trees is that they avoid the forest of trees, merging the common deduction steps with no loss of information. As will be shown on the examples, the size of the resulting information is dramatically reduced with no loss of information. Furthermore, as the image of the database accesses are more faithful to the reality, the information should be more helpful to users.

In a first part, we briefly present the context of deductive databases. We, then, describe the existing explanation systems. In the last part we introduce the definition of substitution sets, DDB derivations and DDB trees.

## 2 The context of deductive databases

In this section, we briefly describe Datalog, the main language of deductive databases. Finally we introduce the notion of proof tree, a data structure used in explanation systems.

## 2.1 Datalog

A deductive database is composed of three parts: a logic program, a database of facts and a set of constraints. The predicates defined in the logic program make up the **intentional** database, those defined in the database make up the **extensional** database. The implementation of the database part is not detailed in what follows; we will suppose that facts of the database are totally instantiated.

The languages used in deductive databases are very often extensions of *Datalog*. A Datalog program is a set of clauses of the form :

$$L : -L_1, \dots, L_n.$$

L is the head of the clause, the  $L_i$  compose the body. In pure Datalog,  $L_i$  is a positive literal of the form  $P(t_1, \dots, t_n)$  where P is a predicate symbol and the  $t_i$ , called terms, are constants or variables. By convention, terms beginning with an upper case letter are variables. Goals of the form  $:- B_1, \dots, B_n$ . are used to query the program. Note that in pure Datalog there are no functors but they have been introduced as extensions. A precise description of Datalog and its extensions can be found in Ullman's book [13].

*Program (intentional database)*

```
(r1)  trip(begin(X), end(Y), cost(P)) :-
      transit(X,Y,P).

(r2)  transit(X, Y, P) :-
      direct(X,Y,P),
      ok(P).

(r3)  transit(X, Y, P) :-
      direct(X,Z,PA),
      ok(PA),
      transit(Z, Y, PB),
      P is PA + PB.

(r4)  ok(P) :-
      P < 15.
```

*Database (extensional database)*

```
direct(a,b,6).    direct(b,c,3).    direct(e,f,2).
direct(f,b,3).    direct(b,l,7).
                  direct(b,k,18).
```

Figure 1: An example of extended Datalog program (with functors)

Figure 1 shows an example of a deductive program and its associate database. There exists a *trip* with a cost P from a begin point X to an end

point Y, if a *transit* between these two points is possible; that is to say either the trip is *direct* and the cost of the trip is *ok*, that is to say that the cost is lower than 15, either there exists an intermediate point with a cost to go there *ok*. The total cost P is the sum of cost of each stage.

With respect to Datalog terminology, this program contains four clauses which define three predicates. The first clause defines *trip/3*. The next two clauses define *transit/3*. The last clause defines *ok/1*. The extensional database is composed of one predicate *direct/3* defined by six tuples. The evaluation of extensional predicates leads to database accesses.

If the query  $trip(D,F,C)$  is asked, the system produces the set of facts:  $trip(begin(a), end(b), cost(6))$ ,  $trip(begin(b), end(c), cost(3))$ ,  $trip(begin(b), end(l), cost(7))$ ,  $trip(begin(e), end(f), cost(2))$ ,  $trip(begin(f), end(b), cost(3))$ ,  $trip(begin(a), end(c), cost(9))$ ,  $trip(begin(a), end(l), cost(13))$ ,  $trip(begin(e), end(b), cost(5))$ ,  $trip(begin(f), end(l), cost(10))$ ,  $trip(begin(f), end(c), cost(6))$ ,  $trip(begin(e), end(c), cost(8))$ ,  $trip(begin(e), end(l), cost(12))$ .

## 2.2 Proof trees

There exists different deductive database evaluation methods (cf Ullman [14]). Independently from these methods, proofs that a fact can be deduced from the program and data can be represented by proof trees. The nodes of a proof tree are goals used during query evaluation. The sons of a node are connected by an AND relation. To be proven, a goal needs to have all his sons proven. The leaves are, in the case of deductive databases, tuples of the database, facts of the program or built-in predicates. Figure 2 represents the proof tree related to the fact  $trip(begin(a),end(b),cost(6))$ .

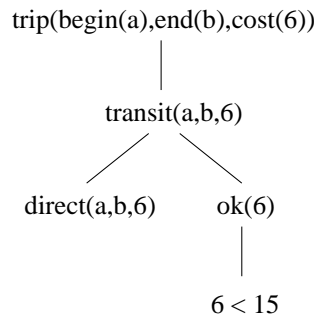


Figure 2: Proof tree related to the fact  $trip(begin(a),end(b),cost(6))$

In this tree,  $transit(a,b,6)$  is deduced from the conjunction of  $direct(a,b,6)$  and  $ok(6)$ , thanks to the second rule of the program of Figure 1.

### 3 Explanations and deductive databases

One of the first debugging systems for deductive databases has been proposed by Shmueli and Tsur [10]. They describe a *diagnosis system* for the deductive database LDL [7]. The principle of this system is to consider the user as an oracle who has an intended interpretation of the deductive program. By querying the user, the system will determine why a fact is missing or why it is wrong. In the case of a wrong fact, the system uses proof trees to direct the diagnosis.

Other *explanation systems* for deductive databases have a different point of view: they construct a trace of the query evaluation that the user can consult.

The first system has been developed for Dedex [6] by Wieland [15]. The proposed explanations are based on proof trees that can be visualized totally or partially. The implementation consists of a dedicated execution mode of Dedex that produces explanations. It decomposes the query evaluation according to the choice of the user.

The *Explain* system of Arora and al. [1] has been developed for CORAL [8]. As the previous system, the chosen structure is a proof tree. *Explain* proposes visualization with zooming possibilities on the different proof trees. *Explain* system consists of adding in CORAL a program which stores information on derivations during query evaluation.

The third system, proposed by Specht [11] for LOLA [4], also uses proof trees as explanations. The implementation principle is a transformation of the user program to insert trace generation as opposed to the other systems which modify deductive engine. The transformed program is queried as usual with the deductive database.

Most of the mixed evaluation and optimization methods, are implemented by program transformations. The queried program is then the optimized one. Sometimes it can be interesting to have explanations on the transformed program, to understand, for example, the applied transformations. But, most of the time, users prefer seeing a trace of their initial program. The system Explain allows only to construct the trace of the transformed program. Wieland's and Specht's system show a trace in terms of the original program, more interesting for users.

The implementations of Explain and Wieland's system modify the deductive database evaluation system. They are not portable. Specht's system transforms the program before optimizations take place. It is therefore independent of the implementation of the deductive database system, and more portable. There are, on the other hand, no information on performance.

A common feature of the three explanation systems is the structure of the traces (or explanations): they all use a forest of proof trees. For each produced fact, there are one or several associated proof trees. Figure 3 shows the forest of the query  $trip(D, F, C)$ .

The whole presented information is interesting but, this form is hardly

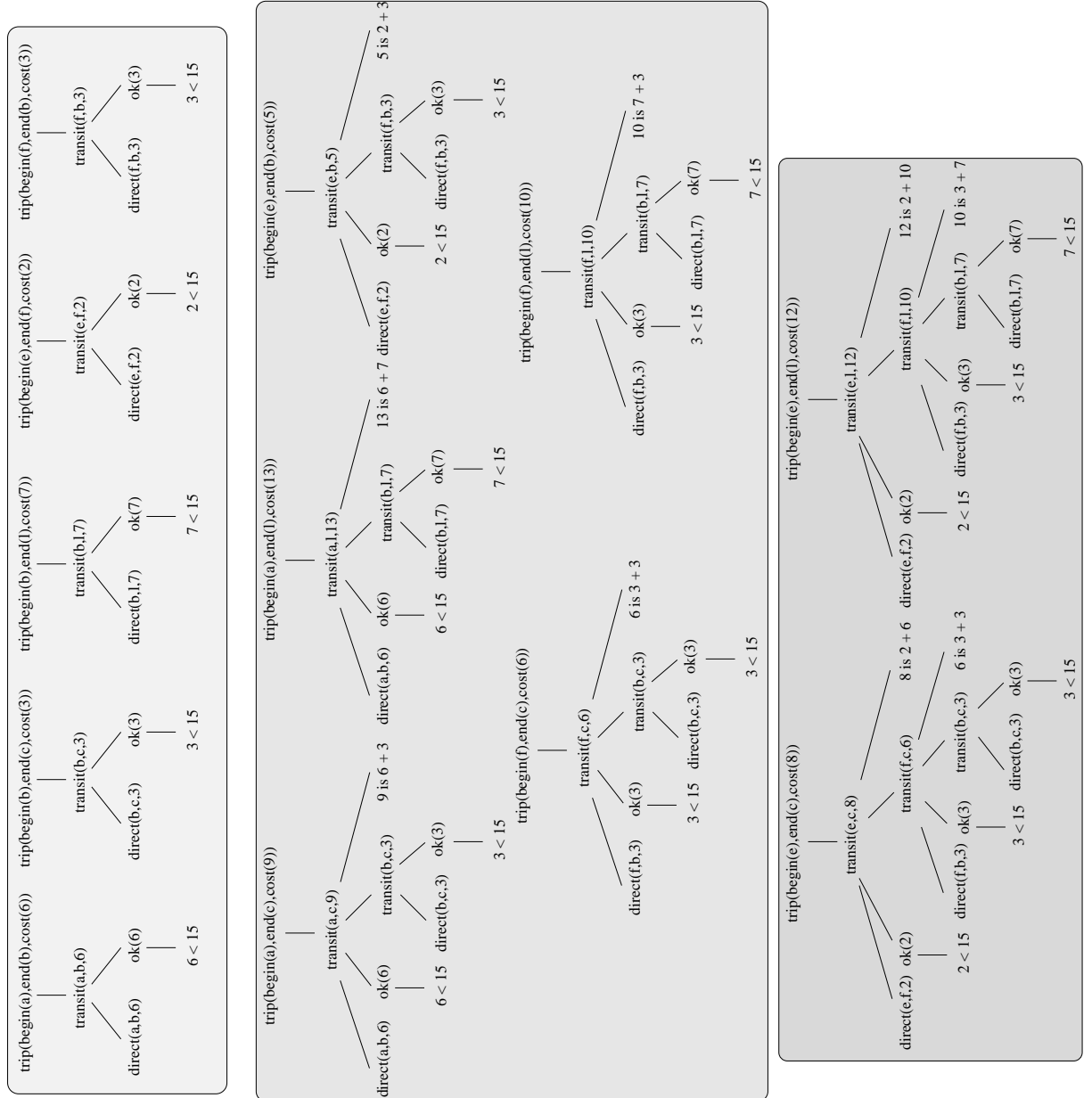


Figure 3: Forest of proof trees for the query  $trip(D, F, C)$  grouped by proof structure

readable in its entirety. On Figure 3, the first five trees represent the same proof structure for different constant values and it is the same thing for the next five trees and the last two.

## 4 The DDB tree

In order to suppress redundancies and to augment readability, we propose a new structure: the DDB tree.

In the following we first recall the definitions of substitution and most general unifier. We then extend the notion of substitution to substitution sets. With this definition, we define a DDB unification. The next part defines a DDB derivation step and a DDB derivation. Finally, we introduce the DDB tree.

### 4.1 SLD derivation

DDB trees are related to SLD trees, a well-known structure in logic programming. An SLD tree is an OR tree. Each branch represents one answer to the query and each node is labeled with a pair composed of a conjunction of subgoals and a substitution. The notion of substitution is defined as follows by Lloyd [5] :

**Definition 1 (Substitution)** *A substitution  $\theta$  is a finite set of the form  $v_1/t_1, \dots, v_n/t_n$ , where each  $v_i$  is a variable, each  $t_i$  is a term distinct from  $v_i$  and the variables  $v_1, \dots, v_n$  are distinct. Each  $v_i/t_i$  is called a binding for  $v_i$ .*

Substitutions are used to construct transitions in SLD trees. A substitution is a unifier of two terms A and B if  $A\theta = B\theta$ . Such a unifier  $\theta$  is called the most general unifier (mgu for short) for two terms if for all other unifiers  $\lambda$ , for some  $\sigma$ , we have  $\lambda = \theta\sigma$ . SLD trees come from SLD resolution. Let  $\langle \leftarrow A_1, \dots, A_m, \dots, A_k, \sigma \rangle$  ( $k \geq 1$ ) be a node n of the SLD tree and  $c_i$  a clause of the form  $A \leftarrow B_1, \dots, B_q$ ,  $q \geq 0$ , if  $A_m$  is the selected atom and if A and  $A_m$  are unifiable with the mgu  $\theta$  then there exists a transition between n and the node n' labeled with

$$\langle \leftarrow (A_1, \dots, A_{m-1}, B_1, \dots, B_q, A_{m+1}, \dots, A_k)\theta, \sigma\theta \rangle$$

For precisions on SLD resolution and SLD tree, see, for example, Deransart and Maluszyński's book [2].

### 4.2 Substitution sets

Substitution is a notion associated with intentional predicates. We define a new structure associated with extensional predicates.

**Definition 2 (Substitution set)**

A substitution set  $\Sigma_{v_1, \dots, v_n}$  relative to a variable set  $\{v_1, \dots, v_n\}$  is a finite set of the form  $\{\{v_1/t_{1_1}, \dots, v_n/t_{n_1}\}, \dots, \{v_1/t_{1_m}, \dots, v_n/t_{n_m}\}\}$ ,  $m \geq 1$ , where the  $t_{i_j}$  are totally instantiated terms.

Another way to define a substitution set is to say that it is a relation, subset of  $T_1 \times \dots \times T_n$  where  $T_i$  are term sets, with  $\{v_1, \dots, v_n\}$  as attribute set and  $(t_{1_i}, \dots, t_{n_i})$  as tuples.

The following notation will be, preferably, used in the following

$$\{v_1, \dots, v_n\} \text{ in } \{(t_{1_1}, \dots, t_{n_1}), \dots, (t_{1_m}, \dots, t_{n_m})\}$$

where *in* has the significance “takes one value in”.

The definition derives from the substitution definition. The other form comes from the notion of relation in relational algebra.

To denote possible values of one variable  $v_i$  of a substitution set  $\Sigma_{v_1, \dots, v_n}$ , we use  $\Sigma_{v_1, \dots, v_n}.v_i$ .

An operator is defined on substitution sets: the join ( $\bowtie$ ) of two substitution sets.

**Definition 3 (Join)**

Let  $\Sigma_{u_1, \dots, u_k, a_1, \dots, a_m}$  and  $\Sigma_{u_1, \dots, u_k, b_1, \dots, b_n}$  be two substitution sets, their join,  $\Sigma_{u_1, \dots, u_k, a_1, \dots, a_m} \bowtie \Sigma_{u_1, \dots, u_k, b_1, \dots, b_n}$ , is a substitution set  $\Sigma_{u_1, \dots, u_k, a_1, \dots, a_m, b_1, \dots, b_n}$ , such that its tuples merge one tuple of  $\Sigma_{u_1, \dots, u_k, a_1, \dots, a_m}$  and one tuple of  $\Sigma_{u_1, \dots, u_k, b_1, \dots, b_n}$  such that  $\Sigma_{u_1, \dots, u_k, a_1, \dots, a_m}.u_i = \Sigma_{u_1, \dots, u_k, b_1, \dots, b_n}.u_i$ , for all  $i$ .

For example, let  $\Sigma_{U,V} = (U, V)$  in  $\{(1, 2), (3, 4), (5, 2), (6, 1), (1, 4)\}$  and  $\Sigma'_{V,W} = (V, W)$  in  $\{(2, 7), (1, 3), (6, 8)\}$  be two substitution sets, their join gives

$$\Sigma''_{U,V,W} = \Sigma_{U,V} \bowtie \Sigma'_{V,W} = (U, V, W) \text{ in } \{(1, 2, 7), (5, 2, 7), (6, 1, 3)\}$$

The join of  $\Sigma_{U,V}$  and  $\Sigma'_{V,W}$  restricts the possible values of  $(U, V)$  and  $(V, W)$  thanks to the equality between  $\Sigma_{U,V}.V$  and  $\Sigma'_{V,W}.V$ .

**4.3 DDB unification**

After a database access, the variables that have been instantiated and whose values are in the substitution set are replaced by what we call a *choice constant*.

**Definition 4 (Choice constant)**

A choice constant  $X^*$  is a term whose value domain is defined by a substitution set  $\Sigma_X$ .

The unification of one term containing choice constants with an other term can restrict their value domain. We define the DDB unification that formalizes the unification of two terms when the first one contains choice constants and the second one does not. This is characteristic of unification with clause head, which does not contain choice constant.

**Definition 5 (DDB unification)**

Let  $A$  and  $B$  be two terms where  $A$  contains the choice constants  $C_1^*, \dots, C_m^*$  with  $\Sigma_{C_1^*, \dots, C_m^*}$  as value domain, and where  $B$  does not contain choice constants ;  $A$  and  $B$  DDB unify if there exists one mgu  $\theta$  for  $A$  and  $B$  and a substitution set  $\Sigma'_{C_1^*, \dots, C_m^*}$  which is a restriction of  $\Sigma_{C_1^*, \dots, C_m^*}$  to the values of  $C_1^*, \dots, C_m^*$  which unify with  $B$ .

The unification of  $p(U^*, V^*)$  with  $p(1, Z)$ , if the value domain of  $(U^*, V^*)$  is  $\Sigma_{U, V} = (U, V)$  in  $\{(1, 2), (3, 4), (5, 2), (6, 1), (1, 4)\}$  gives the mgu  $\theta = \{Z/V^*\}$  and the substitution set  $\Sigma_{U, V} = (U, V)$  in  $\{(1, 2), (1, 4)\}$ .

**4.4 Built-in predicates**

Built-in predicates are often implemented in low-level languages. Furthermore, they are supposed to be bug free and users are seldomly interested in the details of their executions. In DDB trees we therefore represent the result of their evaluation (and not unification steps), in terms of the resulting substitution and substitution set.

Due to choice constants the resulting substitutions are not straightforward. For example, the goal to evaluate can be  $P$  is  $PA^* + PB^*$ , where  $P$  is unbound, and  $PA^*$  and  $PB^*$  are choice constants with substitution set  $(PA, PB)$  in  $\{(6, 3), (6, 7), (2, 3), (3, 3), (3, 7)\}$ . Whereas in pure Prolog, there is at most one result and it is sufficient to add a binding to the substitution, here  $P$  can take several values, namely one of  $\{9, 13, 5, 6, 10\}$ .

Therefore  $P$  becomes a choice constant.

As for unification, the substitution set of the choice constants can be restricted after the evaluation of a built-in predicate. For example, the goal to evaluate can be  $PB^* < 15$  where  $PB^*$  is a choice constant with substitution set  $PB$  in  $\{3, 7, 18\}$ . The resulting substitution set is  $PB$  in  $\{3, 7\}$ .

In summary, the resulting substitution is “as usual” except for the variables which have become choice constants. The substitution set is augmented with the values of the new choice constants and restricted to the values which make the goal succeed.

**4.5 DDB derivation**

As already mentioned, in SLD trees there exists only one type of transition labeled with a clause. In the case of deductive databases some transitions do not use clauses: when the subgoal is an extensional predicate or a built-in predicate. These three types of transition appear in the DDB derivation step defined as follows.

**Definition 6 (DDB derivation step)** Let  $P$  be a program, a DDB derivation step is of the form:

$$s_i, a, s_j$$

such that :

- each  $s_k$ , called a state of DDB derivation, is labeled with  $\langle g, \sigma, \Sigma \rangle$  where  $g$  is a goal,  $\sigma$  a substitution and  $\Sigma$  a substitution set.
- $a$  is an edge labeled with  $\langle c \rangle$  where  $c$  is either a clause or the “builtin” constant or the “database” constant.
- let  $g$  be the goal  $\leftarrow A_1, \dots, A_m, \dots, A_k$  and  $A_m$  be the selected atom,  $\langle g, \sigma, \Sigma \rangle \xrightarrow{\langle c \rangle} \langle g', \sigma', \Sigma' \rangle$  is a transition where  $g', \sigma', \Sigma'$  are obtained as follows:
  - If  $A_m$  is an extensional predicate, and if  $\Sigma_{A_m}$  is the substitution set representing the answer to the query  $A_m$  on the database:
    - \*  $c = \text{“database”}$
    - \*  $\sigma' = \sigma$
    - \*  $\Sigma' = \Sigma \bowtie \Sigma_{A_m}$
    - \*  $g' = \leftarrow A_1, \dots, A_{m-1}, A_{m+1}, \dots, A_k$
  - If  $A_m$  is an intentional predicate and  $c$  a clause  $A \leftarrow B_1, \dots, B_q$ ,  $q \geq 0$  such that  $A_m$  and  $A$  DDB unify with  $\theta$  as mgu and  $\Sigma''$  as substitution set:
    - \*  $c = \text{clause identifier}$
    - \*  $\sigma' = \sigma\theta$
    - \*  $\Sigma' = \Sigma \bowtie \Sigma''$
    - \*  $g' = \leftarrow (A_1, \dots, A_{m-1}, B_1, \dots, B_q, A_{m+1}, \dots, A_k)\theta$
  - If  $A_m$  is a built-in predicate, then  $\theta$  is the resulting substitution of the evaluation of the built-in predicate after the DDB unification and  $\Sigma''$  is the resulting substitution set:
    - \*  $c = \text{“builtin”}$
    - \*  $\sigma' = \sigma\theta$
    - \*  $\Sigma' = \Sigma \bowtie \Sigma''$
    - \*  $g' = \leftarrow (A_1, \dots, A_{m-1}, A_{m+1}, \dots, A_k)\theta$

We, now, define the DDB derivation which is a sequence of DDB derivation steps that ends successfully, i.e. with the empty goal.

**Definition 7 (DDB derivation)** Let  $P$  be a program and  $g_0$  a goal, a DDB derivation for  $P$  and  $g_0$  is a finite sequence of the form

$$s_0, a_0, s_1, \dots, a_{n-1}, s_n$$

such that :

- $s_0$  is  $\langle g_0, \{\}, \{\} \rangle$
- $s_n$  is  $\langle \square, \sigma_n, \Sigma_n \rangle$  where  $\square$  denotes the empty goal and  $\sigma_n$  and  $\Sigma_n$  are the resulting substitution and substitution set.

	<i>Substitution</i>	<i>Substitution set</i>
<b>trip(D,F,C)</b>	$\{\}$	$\{\}$
r1		
<b>transit(X,Y,P)</b>	$\{D/begin(X),F/end(Y),C/cost(P)\}$	$\{\}$
r3		
<b>direct(X<sub>1</sub>,Z,PA),ok(PA), transit(Z,Y<sub>1</sub>,PB),P<sub>1</sub> is PA + PB</b>	$\{D/begin(X),F/end(Y),C/cost(P),X/X_1, Y/Y_1, P/P_1\}$	$\{\}$
database		
<b>ok(PA*),transit(Z*,Y<sub>1</sub>,PB), P<sub>1</sub> is PA* + PB</b>	$\{D/begin(X),F/end(Y),C/cost(P),X/X_1, Y/Y_1, P/P_1\}$	$(X_1,Z,PA) \text{ in } \{(a,b,6),(e,f,2),(f,b,3)(b,c,3),(b,l,7),(b,k,18)\}$
r4		
<b>PA* &lt; 15,transit(Z*,Y<sub>1</sub>,PB), P<sub>1</sub> is PA* + PB</b>	$\{D/begin(X),F/end(Y),C/cost(P),X/X_1, Y/Y_1, P/P_1, P_2/PA\}$	$(X_1,Z,PA) \text{ in } \{(a,b,6),(e,f,2),(f,b,3)(b,c,3),(b,l,7),(b,k,18)\}$
builtin		
⑥ <b>transit(Z*,Y<sub>1</sub>,PB)P<sub>1</sub> is PA* + PB</b>	$\{D/begin(X),F/end(Y),C/cost(P),X/X_1, Y/Y_1, P/P_1, P_2/PA\}$	$(X_1,Z,PA) \text{ in } \{(a,b,6),(e,f,2),(f,b,3)(b,c,3),(b,l,7)\}$
r2		
⑦ <b>direct(Z*,Y<sub>3</sub>,P<sub>3</sub>), ok(P<sub>3</sub>), P<sub>1</sub> is PA* + P<sub>3</sub></b>	$\{D/begin(X),F/end(Y),C/cost(P),X/X_1, Y/Y_1, P/P_1, P_2/PA\}$ $Z/X_3, Y_1/Y_3, P_3/PB\}$	$(X_1,Z,PA) \text{ in } \{(a,b,6),(e,f,2),(f,b,3)(b,c,3),(b,l,7)\}$
database		
⑧ <b>ok(P<sub>3</sub>*),P<sub>1</sub> is PA* + P<sub>3</sub>*</b>	$\{D/begin(X),F/end(Y),C/cost(P),X/X_1, Y/Y_1, P/P_1, P_2/PA\}$ $Z/X_3, Y_1/Y_3, P_3/PB\}$	$(X_1,Z,Y_3,PA,P_3) \text{ in } \{(a,b,c,6,3),(a,b,l,6,7),(e,f,b,2,3),(f,b,c,3,3),(f,b,l,3,7),(a,b,k,6,18),(f,b,k,3,18)\}$
r4		
⑨ <b>P<sub>3</sub>* &lt; 15, P<sub>1</sub> is PA* + P<sub>3</sub>*</b>	$\{D/begin(X),F/end(Y),C/cost(P),X/X_1, Y/Y_1, P/P_1, P_2/PA\}$ $Z/X_3, Y_1/Y_3, P_3/PB, P_4/P_3\}$	$(X_1,Z,Y_3,PA,P_3) \text{ in } \{(a,b,c,6,3),(a,b,l,6,7),(e,f,b,2,3),(f,b,c,3,3),(f,b,l,3,7),(a,b,k,6,18),(f,b,k,3,18)\}$
builtin		
⑩ <b>P<sub>1</sub> is PA* + P<sub>3</sub>*</b>	$\{D/begin(X),F/end(Y),C/cost(P),X/X_1, Y/Y_1, P/P_1, P_2/PA\}$ $Z/X_3, Y_1/Y_3, P_3/PB, P_4/P_3\}$	$(X_1,Z,Y_3,PA,P_3) \text{ in } \{(a,b,c,6,3),(a,b,l,6,7),(e,f,b,2,3),(f,b,c,3,3),(f,b,l,3,7)\}$
builtin		
⑪ $\square$	$\{D/begin(X),F/end(Y),C/cost(P),X/X_1, Y/Y_1, P/P_1, P_2/PA\}$ $Z/X_3, Y_1/Y_3, P_3/PB, P_4/P_3\}$	$(X_1,Z,Y_3,PA,P_3,P_1) \text{ in } \{(a,b,c,6,3,9),(a,b,l,6,7,13),(e,f,b,2,3,5),(f,b,c,3,3,6),(f,b,l,3,7,10)\}$

Figure 4: A DDB derivation of the query  $trip(D,F,C)$

- each  $s_i, a_i, s_{i+1}$  is a DDB derivation step

Figure 4 shows a DDB derivation of the query  $trip(D, F, C)$ .

The first transition type is a transition with a clause. The transition between state 6 and state 7 uses the clause  $r2$ , the substitutions of states 6 and 7 are different but they have the same substitution sets since the database has not been accessed and no restriction has been entailed by unification.

The transition between state 7 and state 8 illustrates a transition by database access. The substitution set of 8,  $(X\_1, Z, Y\_3, PA, P\_3)$  in  $\{(a, b, c, 6, 3), (a, b, l, 6, 7), (e, f, b, 2, 3), (f, b, c, 3, 3), (f, b, l, 3, 7), (a, b, k, 6, 18), (f, b, k, 3, 18)\}$  has been constructed by the join between  $(Z, Y\_3, P\_3)$  in  $\{(a, b, 6), (b, c, 3), (b, l, 7), (e, f, 2), (f, b, 3), (b, k, 18)\}$ , result of the database query  $direct(Z^*, Y\_3, P\_3)$ , and  $\Sigma_7 = (X\_1, Z, PA)$  in  $\{(a, b, 6), (b, c, 3), (b, l, 7), (e, f, 2), (f, b, 3)\}$ . In  $direct(Z^*, Y\_3, P\_3)$ ,  $Z^*$  is a choice constant, it has been instantiated.

The transition between state 9 and state 10 illustrates the value domain restriction of choice constants appearing in a predicate. The predicate  $P\_3^* < 15$  is evaluated with value domain  $P\_3^*$  ( $P\_3$  in  $\{3, 7, 18\}$ ). The evaluation restricts the value domain of  $P\_3$  to ( $P\_3$  in  $\{3, 7\}$ ). Finally, the substitution set of 10 is the result of the join between the substitution set of 9 and the resulting substitution set.

In state 10, we find the built-in predicate  $is$ . Its evaluation add  $P\_1$  in the substitution set. The different values of  $P\_1$  have been computed by the sum of the values of  $PA$  and  $P\_3$ .

This derivation corresponds to five proof trees of Figure 3 with same structure.

### Computation of the solution

The solution of the query  $g_0$  along a DDB derivation is computed, by instantiating the variables of  $g_0$ , firstly according to the final substitution  $\sigma$  as far as possible ; secondly, according to the final substitution set  $\Sigma$ . The result is, in general, a set of solutions.

For each binding  $X/t$  in  $\sigma$ , each occurrence of  $X$  in the right part of a binding is replaced by  $t$ . When no more binding is applicable, only bindings with variables of  $g_0$  in their left part are kept.

On the example of Figure 4, the final substitution is  $\sigma = \{D/begin(X), F/end(Y), C/cost(P), X/X_1, Y/Y_1, P/P_1, P_2/PA, Z/X_3, Y_1/Y_3, P_3/PB, P_4/P_3\}$  and the variables of  $trip(D, F, C)$  are  $D, F, C$ . The obtained substitution after restriction to  $D, F, C$  is  $\{D/begin(X_1), F/end(Y_3), C/cost(P_1)\}$ .

The substitution set  $\Sigma_{X_1, \dots, X_n}$  is then applied to the restricted substitution. For each tuple of  $\Sigma_{X_1, \dots, X_n}$ , a substitution is created by replacing in  $\sigma$ ,  $X_1, \dots, X_n$  by their value in the tuple. Each substitution is a solution.

The substitution set obtained on the example is  $\Sigma = (X_1, Z, Y_3, PA, P_3, P_1)$  in  $\{(a, b, c, 6, 3, 9), (a, b, l, 6, 7, 13), (e, f, b, 2, 3, 5), (f, b, c, 3, 3, 6), (f, b, l, 3, 7, 10)\}$ .

The five solutions are  $\{D/begin(a), F/end(c), C/cost(9)\}$ ,  
 $\{D/begin(a), F/end(l), C/cost(13)\}$ ,  $\{D/begin(e), F/end(b), C/cost(5)\}$ ,  
 $\{D/begin(f), F/end(c), C/cost(6)\}$ ,  $\{D/begin(f), F/end(l), C/cost(10)\}$  .

## 4.6 DDB tree

Finally, here is the DDB tree definition. A DDB derivation is a branch of a DDB tree.

**Definition 8 (DDB tree)** *A DDB tree for a program  $P$  and a goal  $g_0$  is a tree with labeled nodes and arcs defined as follows:*

- *The label of each **node** is a state of DDB derivation of  $P$  and  $g_0$ .*
- *The label of each **arc** is a clause of  $P$  or the “builtin” constant or the “database” constant.*
- *The root node is labeled by the initial state  $\langle g_0, \{\}, \{\} \rangle$ .*
- *A node  $n$  labeled by a state  $s$  is connected by an arc labeled  $c$  with a node  $n'$  labeled  $s'$  if,  $s, c, s'$  is a DDB derivation step.*
- *Each branch is a DDB derivation*

Figure 5 presents a DDB tree of the query  $trip(D, F, C)$ , where substitutions ( $\sigma$ ) are omitted.

The tree is equivalent to the forest of proof trees given Figure 3. The first branch to the left represents the first 5 proof trees. The second branch represents the next 5 proof trees and the last branch represents the last 2 trees.

Note that the forest of proof trees gathers 105 nodes whereas the DDB tree consists of only 25.

Note also that the substitutions and substitution sets may be ignored in a first stage, if the user is mainly interested in the structure of the evaluation. If the user is only interested in a single fact, the corresponding proof tree is of course better suited and can be reconstructed from the DDB tree.

Note however that our aim is to build a general explanation system which is able to answer general queries about the whole evaluation. The DDB tree presents information which will be abstracted by analyses in order to propose to the user more interesting explanations.

## 5 Conclusion

In existing explanation systems for deductive databases, explanations were represented by forests of proof trees. With this form, trace information is fragmented. Furthermore, for each deduced fact a proof tree has to be shown,



whereas much of the deduction can be common between several deduced facts.

We have defined the DDB tree, a new structure based on SLD trees. This tree merges the common deduction steps thanks to the notion of substitution set. The DDB tree is much smaller and better structured than a plain forest of proof trees.

DDB trees can be shown to users or can be analyzed further by an explanation system.

## References

- [1] T. Arora, R. Ramakrishnan, W.G. Roth, P. Seshadri, and D. Srivastava. Explaining program execution in deductive systems. In S. Ceri, K. Tanaka, and S. Tsur, editors, *Proceedings of the Deductive and Object-Oriented Databases Conference*, number 760 in Lecture Notes in Computer Science. Springer-Verlag, December 1993.
- [2] Pierre Deransart and Jan Maluszyński. *A grammatical view of logic programming*. The MIT Press, 1993.
- [3] M. Ducassé. *An extendable trace analyser to support automated debugging*. PhD thesis, University of Rennes I, France, June 1992. European Doctorate.
- [4] G. Specht et B. Freitag. Amos : a natural language parser implemented as a deductive database in LOLA. In R. Ramakrishnan, editor, *Applications of logic databases*, pages 197–214. Kluwer Academic Publishers, 1995.
- [5] J.W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, Heidelberg, 1987. Second edition.
- [6] R. Marti, C. Wieland, and B. Wüthrich. Adding inferencing to a relational database management system. In T. Härder, editor, *Datenbanksysteme in Büro, Technik und Wissenschaft*, pages 266–270. Springer Verlag, 1989.
- [7] S. Naqvi and S. Tsur. *A Logical Language for Data and Knowledge Bases*. W. H. Freeman, 1989.
- [8] R. Ramakrishnan, D. Srivastava, and P. Sheshadri. Coral : Control, relations and logic. In Li-Yan Yuan, editor, *Proceedings of the 18th International Conference on Very Large Databases*, 1992.
- [9] R. Ramakrishnan and J. D. Ullman. A survey of deductive database systems. *The journal of logic programming*, pages 125–149, 1995.

- [10] O. Shmueli and S. Tsur. Logical diagnosis of LDL programs. In D.H.D. Warren and P. Szeredi, editors, *Proceedings of the International Conference on Logic Programming*, pages 112–129, Jerusalem, Israël, June 1990. MIT Press. ICLP'90.
- [11] G. Specht. *Source-to-source Transformationen zur Erklärung des Programmverhaltens bei deduktiven Datenbanken*. PhD thesis, Technischen Universität München, Juni 1992. In German.
- [12] G. Specht. Generating explanation trees even for negations in deductive database systems. In M. Ducassé, B. Le Charlier, Y.-J. Lin, and U. Yalcinalp, editors, *Proceedings of ILPS'93 Workshop on Logic Programming Environments*, Vancouver, October 1993. Publication IRISA, Campus de Beaulieu, 35042 Rennes, France. LPE'93.
- [13] J. D. Ullman. *Principles of Database and knowledge-base systems*, volume 1. Computer Science Press, 1988.
- [14] J. D. Ullman. *Principles of Database and knowledge-base systems*, volume 2. Computer Science Press, 1988.
- [15] C. Wieland. Two explanation facilities for the deductive database management system DeDex. In H. Kangassalo, editor, *Proceedings of the 9th Conference on Entity-Relationship Approach*, pages 189–203, 1990. ETH Zurich.