A Tabulation Algorithm for CLP
Revised Report

Fred Mesnard and Sébastien Hoarau

Iremia, Université de la Réunion
15, avenue René Cassin - BP 7151 -
97 715 Saint Denis Messag. Cedex 9 France
E-mail: fred@univ-reunion.fr

Abstract

Since its introduction in logic programming, tabulation has proven to a be powerful tool in many areas. The technique has been lifted to constraint extensions of Datalog and to constraint logic programming (CLP). In this abstract, we describe a new formulation of tabulation for CLP, directly designed towards an implementation on CLP systems. We illustrate the use of our algorithm by comparing the performance of our implementation with bottom-up evaluation for groundness analysis of logic programs.

1 Introduction

Since its introduction in logic programming [10], tabulation has proven to be a powerful tool in many areas [12]. The technique has been implemented (see [9]) and lifted to constraint extensions of Datalog in [11] and to constraint logic programming (CLP) in [3]. In this abstract, we describe a new formulation of tabulation for CLP, directly designed towards an implementation on CLP systems.

Let us give the intuition of the algorithm. It is mainly a meta-interpreter for LD-resolution (i.e., using the left-to-right computation rule), augmented with two tables Sol and Ref. The table Sol records for each unit goal of the form $c \diamond p(\tilde{x})$ the set of solutions obtained so far. The table Ref records for each unit goal G the set of resolvents waiting for new solution of G to proceed. The meta-interpreter works as follows. If the current resolvent is empty, then we have a solution for the initial unit goal G. We record (if necessary) the solution in the table Sol and we unfold all the resolvents in Ref waiting for a solution of G. If the current resolvent is not empty, we select the leftmost atom. If we have never met this new unit goal G', we record the occurrence of G' in the two tables and unfold the resolvent using all the clauses matching G'. If we have already met G', we update the table Ref, telling that the current resolvent is waiting for new solutions of G'. With the already available solutions for G' in Sol, we unfold the current resolvent.

We organize the paper as follows: we first present the basic concepts we need and we describe our tabulation algorithm and possible optimizations. The partial resolution tree is implicitly handled via the recursive structure of the algorithm and we take special care to variable renaming. Then we compare the performance of our implementation with bottom-up evaluation for groundness analysis of logic programs.

2 Preliminaries

A tuple $\langle o_1, \ldots, o_n \rangle$ is written $\tilde{o}$ and $\mid \tilde{o} \mid= n$. The empty tuple (for $n = 0$) is noted $\langle\rangle$. The symbol $*$ denotes concatenation of tuples. The set of natural numbers is noted $\mathbb{N}$ and $\mathbb{N}^*$ denotes the set of finite tuples (or sequences) of natural numbers. The set of free variables of a syntactic object o is denoted $var(o)$. The term $copy(o)$ denotes a "fresh copy" of o.

Constraints We try to stick to the notations and the conventions introduced in [7] except for a few notions that we now define. We say that θ is a *solution* of the *satisfiable* constraint c if θ is a valuation such that (s.t.) $\mathcal{D} \models c\theta$. We now omit the reference to the constraint domain $\mathcal{D}$. Let c_1 and c_2 be two constraints. We write $c_1 \models c_2$ as a shorthand for $\models \forall(c_1 \rightarrow c_2)$. We say that a constraint c is a new solution w.r.t. (with respect to) the set of constraints C (in short $new\_solution(c, C)$) iff $\forall c_i \in C, \models \neg\forall(c \leftrightarrow c_i)$. We recall that the formula $\exists_{\tilde{x}} c$ denotes the projection of the constraint c on its (free) variables, except those in $\tilde{x}$. The set of constraints is noted $\mathcal{CONS}$.

Example 2.1 *Consider the constraint domain $\Re_{Lin}$ (linear arithmetic over the reals or the rational numbers). Let $c_1 \equiv 1 \leq x \wedge x \leq 3 \wedge 2y = x + 1$ and $c_2 \equiv y \leq 3$. The constraint c_2 is satisfiable. The valuation θ which maps each variable to 0 is a solution of c_2. We have $\exists_{-\langle y\rangle} c_1 \equiv 1 \leq y \wedge y \leq 2$. Hence $c_1 \models c_2$. We have $\exists_{-\langle z\rangle} c_1 \equiv \exists_{-\langle\rangle} c_1 \equiv true$.*

We only consider *ideal* CLP. So we assume that complete algorithms are available for *satisfiability* (test for $\models \exists c$), *entailment* (test for $c_1 \models c_2$) and *projection* (given c_0 and $\tilde{x}$, compute c_1 s.t. $\models \forall(c_1 \leftrightarrow \exists_{-\tilde{x}} c_0)$). For instance, the constraint domains $\mathcal{FT}$ (finite trees), $\mathcal{BOOL}$ (booleans) and $\Re_{Lin}$ fulfill the above requirements. In [8], we show how one may define such operators for $\mathcal{BOOL}$ and $\Re_{Lin}$. But the constraint domain $\mathcal{FD}$ (finite domains) does not. Indeed, it seems that there is no *efficient and complete* algorithm for satisfiability, neither for entailment and projection, although each problem is decidable.

Atoms, programs and resolvents An atom is of the form $p(\tilde{x})$. A program $Prog$ is a finite set of clauses of the form $cl : p_0(\tilde{x}_0) \leftarrow c \diamond p_1(\tilde{x}_1), \ldots, p_n(\tilde{x}_n)$ where cl is a unique natural number identifying the clause in $Prog$, $n \geq 0$, the p_i's are user defined predicates and the $\tilde{x}_i$'s are vectors of distinct variables s.t. $i \neq j \rightarrow \tilde{x}_i \cap \tilde{x}_j = \phi$. Hence c is the conjunction of all constraints, including unifications. In the clause cl, a variable x appears only once, except in c where x may reappear many times. A resolvent is coded $c \diamond \tilde{A}$ where c is a satisfiable constraint and $\tilde{A}$ a tuple of atoms. The set of resolvents is noted $\mathcal{RES}$.

Keys A *key* is a term $key(p(\tilde{x}), c)$ where c is a satisfiable constraint s.t. $var(c) \subseteq \tilde{x}$. Let $K_1 = key(p_1(\tilde{x}), c_1)$ and $K_2 = key(p_2(\tilde{y}), c_2)$ be two keys. They are equivalent, i.e. $K_1 \sim K_2$, iff $p_1 = p_2 \wedge \mid \tilde{x} \mid=\mid \tilde{y} \mid \wedge \{\tilde{x}\theta \mid\models c_1\theta\} = \{\tilde{y}\sigma \mid\models c_2\sigma\}$. If $K_1 \sim K_2$ then $e_{K_1=K_2}$ denotes the constraint $\tilde{x} = \tilde{y}$, i.e. $x_1 = y_1 \wedge \ldots \wedge x_n = y_n$. The set of keys is noted $\mathcal{KEY}$.

Tables Let E be a set of terms and 2^E be the powerset of E. A table T is a finite mapping $\mathcal{KEY} \rightarrow 2^E$. The constant *empty_table* denotes the table T s.t. its domain, noted $dom(T)$, is empty. Let K be a key, T be a table and e be an element of E. If $K \notin dom(T)$, then the function *add_key* applied to T and K creates a new table T' identical to T except that $T'(K) = \phi$. If $K \in dom(T)$, the function *update* applied to T, K and e creates a new table T' identical to T except that $T'(K) = T(K) \cup \{e\}$. The function *equiv_key* applied to T and K returns a key $K' \in dom(T)$ s.t. $K \sim K'$.

3 The Tabling Interpreter

The tabling interpreter *tab* is defined figure 1 in a pseudo-Pascal like language. It makes use of two tables.

- If E is the set $\mathcal{CONS} \times \mathbb{N}^*$ then the corresponding table, noted Sol, records for each key K the set of solutions obtained so far while solving K considered as a goal. Associated to a solution s, we record a tuple of natural numbers which denotes a sequence of clauses to apply to the goal K to obtain the solution s using the left-to-right computation rule.

- If E is the set $\mathcal{KEY} \times \mathcal{RES} \times \mathbb{N}^*$ then the corresponding table, noted Ref, records for each key $K = key(p(\tilde{x}), c)$ the set of partial computations waiting for a new solution of K to proceed. Each partial computation is stored as a tuple formed with a key K', its associated pending resolvent $c \diamond \widetilde{A}'$ (the real resolvent R is $c \diamond \langle p(\tilde{x})\rangle * \widetilde{A}'$ but we don't need to store $p(\tilde{x})$ because that atom is already in K) and once again, a tuple of natural numbers which denotes the sequence of clauses to apply to the goal K' to obtain the resolvent R.

Assume we want to prove the goal $G :\leftarrow c \diamond p(\tilde{x})$ where $var(c) \subseteq \tilde{x}$. Let $K = key(p(\tilde{x}), c)$. The evaluation of $tab(K, c \diamond p(\tilde{x}), empty\_table, empty\_table, \langle\rangle)$, if it terminates, returns a pair $\langle Ref, Sol\rangle$ where $Sol(K)$ is the set of 'computed answers' for G.
We now comment the code line by line. First we check (line 2) whether the current resolvent is a solution (i.e. a leaf in the LD-tree). In that case and if it is a new solution (line 5), we update the table Sol and we prove all the partial computations waiting for a new solution of K. Then we return the possibly updated tables Ref and Sol.
If the current resolvent is not a solution (line 13), we select the left-most atom and construct the associated key K'. If we have never met such a key (line 17), we initialize the two tables for that key and prove all the corresponding once-unfolded goals. Then we go back to our initial task. If we have already met K' (line 25), we update Ref telling to go on if there is a new solution for K' and with all the solutions already computed, we proceed with the computation. Then we return the possibly updated tables Ref and Sol.

4 Properties and Optimizations of the Algorithm

We informally present some properties of the algorithm. First we point out that $dom(Sol) = dom(Ref)$ and the variables of $var(dom(Sol))$ are local to the tables. It allows us to implement the test $\{\tilde{x}\theta \mid \models c_1\theta\} = \{\tilde{y}\sigma \mid \models c_2\sigma\}$ as $\tilde{x} = \tilde{y} \models c_1 \leftrightarrow c_2$ (see [8]). Note also that each time we want to use a recorded solution or a pending resolvent, we first take a fresh copy, in order to avoid in the implementation wrong implicit unifications.

4.1 Termination

We show that if the domain is finite (e.g. $\mathcal{BOOL}$), then termination is insured. It is easy to see that the number of non-equivalent keys is finite, and for each key, the number of solutions for a given key is finite. Hence recursive calls from line 9, 22 and 24 occur a finite number of times. Note that the size of the longest resolvent that the algorithm has to handle is equal to the size of the longest body of the clauses in $Prog$. So if an evaluation of $tab(K, c \diamond p(\tilde{x}), empty\_table, empty\_table, \langle\rangle)$ diverges, it implies that after a certain amount of computation, we only have recursive calls coming from line 31. But each time we suppress an atom from the resolvent. Hence the algorithm terminates.

function $tab(K, c \diamond \widetilde{A}, Ref, Sol, Cls)$

if $|\widetilde{A}| = 0$ **then**

    $\bar{K} \leftarrow equiv\_key(Sol, K)$

    $s \leftarrow \exists_{-var(\bar{K})}(c \wedge e_{K=\bar{K}})$

    if $new\_solution(s, Sol(\bar{K}))$ **then**

        $Sol \leftarrow update(Sol, \bar{K}, \langle s, Cls \rangle)$

        $\langle s, R \rangle \leftarrow copy(\langle s, Ref(\bar{K}) \rangle)$

        for each $\langle K', c' \diamond \widetilde{B}, Cls' \rangle \in R$ s.t. $\models \exists (c' \wedge s)$

            $\langle Ref, Sol \rangle \leftarrow tab(K', c' \wedge s \diamond \widetilde{B}, Ref, Sol, Cls' * Cls)$

        end for

    end if

    return $\langle Ref, Sol \rangle$

else

    let $\widetilde{A} = \langle p(\tilde{x}) \rangle * \widetilde{A}'$

    $c' \leftarrow \exists_{-\tilde{x}} c$

    $K' \leftarrow key(p(\tilde{x}), c')$

    if there is no key $\bar{K}' \in dom(Sol)$ s.t. $\bar{K}' \sim K'$ **then**

        $\bar{K}' \leftarrow copy(K')$

        $Sol \leftarrow add\_key(Sol, \bar{K}')$

        $Ref \leftarrow add\_key(Ref, \bar{K}')$

        for each (fresh copy) $cl_i : p(\tilde{y}) \leftarrow c'' \diamond \widetilde{B} \in Prog$ s.t. $\models \exists (c'' \wedge c' \wedge \tilde{x} = \tilde{y})$

            $\langle Ref, Sol \rangle \leftarrow tab(K', c'' \wedge c' \wedge \tilde{x} = \tilde{y} \diamond \widetilde{B}, Ref, Sol, \langle cl_i \rangle)$

        end for

        return $tab(K, c \diamond \widetilde{A}, Ref, Sol, Cls)$

    else

        $\bar{K}' \leftarrow equiv\_key(Sol, K')$

        $Ref \leftarrow update(Ref, \bar{K}', \langle K, c \wedge e_{K'=\bar{K}'} \diamond \widetilde{A}', Cls \rangle)$

        $\langle \bar{K}', S \rangle \leftarrow copy(\langle \bar{K}', Sol(\bar{K}') \rangle)$

        $e \leftarrow c \wedge e_{K'=\bar{K}'}$

        for each $\langle c'', Cls' \rangle \in S$ s.t. $\models \exists (e \wedge c'')$

            $\langle Ref, Sol \rangle \leftarrow tab(K, e \wedge c'' \diamond \widetilde{A}', Ref, Sol, Cls * Cls')$

        end for

        return $\langle Ref, Sol \rangle$

    end if

end if

Figure 1: The function *tab*.

4.2 Correctness

We note the following facts.
For the table Sol, for any solution s recorded for the key K, the corresponding tuple Cls denotes a sequence of clauses to apply to the goal K to obtain the solution s using the left-to-right computation rule.
For the table Ref, any partial computation is stored as a tuple formed with a key K', its associated pending resolvent $c \diamond \widetilde{A}'$ and a tuple of natural numbers which denotes the sequence of clauses to apply to the goal K' to obtain the resolvent $c \diamond \langle p(\tilde{x})\rangle * \widetilde{A}'$.
For a call $tab(K, c \diamond \widetilde{A}, Ref, Sol, Cls)$, the tuple Cls is a sequence of clauses for unfolding the goal K to the resolvent $c \diamond \widetilde{A}$.
Hence correctness follows from the first point. Indeed, the *raison d'être* of the last argument of *tab* is only justified for the help it provides in the correctness proof.

4.3 Completeness

If $\leftarrow c' \diamond$ is a computed answer for the initial goal $c \diamond \widetilde{A}$ obtained in n LD-resolution steps and if a call to $tab(K, c \diamond \widetilde{A}, Ref, Sol, Cls)$ returns $\langle Ref', Sol'\rangle$ then there exits Cls' s.t. $\langle c', Cls'\rangle \in Sol'(K)$. The proof by induction on n, detailed in section 7, is mainly based on two remarks. First, there is a strong analogy in one step of LD-resolution and two consecutive calls of *tab* (line 1 and lines 22-31). Second, when a call to $tab(K, Res, Ref, Sol, Cls)$ returns $\langle Ref', Sol'\rangle$, $Sol \subseteq Sol'$.

4.4 Optimizations

We have coded the tabling interpreter for CLP($\mathcal{BOOL}$) in SICStus Prolog. The implementation and the algorithm were developed together. We use the projection operator defined in [8] and we experiment three optimizations.
As proposed in [3], we replace the definition *equiv_key* in lines 3, 17 and 26 by *geq_key*. The function *geq_key* applied to a table T and a key $K = key(p(\tilde{x}), c)$ returns the "most general" key $K' = key(p(\tilde{y}), c')$ s.t. $\tilde{x} = \tilde{y} \models c \rightarrow c'$. The idea is to re-use already done computations if one finds a more general key in the table. Hence the size of the tables are smaller.
We also modify the definition of *new_solution* (line 5) We say that a constraint c is a new solution in the set of constraints C iff $\forall c_i \in C, \models \exists(c \wedge \neg c_i)$. Hence we spare space by reducing the table Sol.
Last but not least, we obtain the biggest improvement in speed by changing the definition of the function *update* (line 6) as follows. If $K \in dom(T)$, the function *update* applied to T, K and e creates a new table T' identical to T except that $T'(K) = \{e \vee [\bigvee_{c \in T(K)} c]\}$. So if $K \in dom(Sol)$, either $Sol(K) = \phi$ or $Sol(K) = \{c\}$ where c is the disjunction of all the solutions found so far. The third optimization assumes that disjunctions of constraints are constraints, which is true for $\mathcal{BOOL}$ but false in general for other constraint domains.

5 Tabulation versus Bottom-Up Evaluation For Groundness Analysis of Logic Programs

5.1 A data structure for tables

The *tab* algorithm handles pairs of the form $\langle key, set\rangle$ where *key* represents a constraint atom we want to prove and *set* is the value associated with *key*. The tabling interpreter

makes use of two kinds of set: a set of constraints (the computed solutions) or a set of resolvents (which are waiting for a new solution).
So we need a data structure which manipulates these pairs $\langle key, set\rangle$ and s.t. the following operations are as efficient as possible:

- to access to a pair,
- to create a new pair,
- to access to the set associated with a key,

Since there exists a total order over Sicstus terms, an AVL tree like structure seems a good choice.

Example 5.1 *Let us consider the following set of pairs:* $\{\langle key(go,1),\{1\}\rangle, \langle key(p(X,Y),1),\{X =< Y\}\rangle, \langle key(q(A),1),\{A\}\rangle, \langle key(p(X,Y),X),\{X * Y\}\rangle\}$. *It can be coded as:*

$$\begin{array}{ccc} & key(p(X,Y),1) - \{X =< Y\} & \\ \swarrow & & \searrow \\ key(p(X',Y'),X') - \{\} & & key(q(Z),1) - \{1\} \\ \swarrow & & \\ key(go,1) - \{1\} & & \end{array}$$

But with this choice there is some redundant information. In the previous example $p(X,Y)$ appears in two different nodes. So we propose to use p/n as a key of the tree and a list of element of the form $key(p(\tilde{X}),C) - set$ as the associated value. The first advantage is that we reduce the size of the tree. The second one is that the set of p/n in a program is known at the beginning of the computation. So the construction of the AVL tree is made only once. Then we just need to update the value associated with a key of the tree.

Example 5.2 *The tree for the same set of the example 5.1:*

$$\begin{array}{ccc} & p/2 - [key(p(X,Y),1) - \{X =< Y\}, key(p(X',Y'),X') - \{\}] & \\ \swarrow & & \searrow \\ go/0 - [key(go,1) - \{1\}] & & q/1 - [key(q(Z),1) - \{1\}] \end{array}$$

Let P be a program with m predicates and p/n one of them. This data structure gives the following results:

- to access to the set of pairs referenced by p/n: $O(\log m)$ in the worst case,
- to create a new pair $\langle key(p(\tilde{X}),C),\phi\rangle$: $O(\log m)$ in the worst case,
- to access to the value associated with a key $key(p(\tilde{X}),C)$: $O(2^n + \log m)$ in the worst case.

The last action seems very inefficient but we noticed that in all the examples we have encountered, for a predicate p/n there were at most three or four different keys $key(p(\tilde{X}),C)$. In other words, the number of distinct "calling modes" of a logic procedure is low.

5.2 Results

We selected some well-known programs available at `www.cs.huji.ac.il/~naomil`. We applied four algorithms for the computation of their least boolean models. Results are summarized in the following table.

Language: Sicstus prolog 3.5 - compiled code
Machine: ULTRA 1 SUN Station
Frequency: 167MHz
RAM: 128Mo
Time unit: ms

Name	#ClP	#VarP	Trans	Scc	#ClB	#VarB	Bu	TabScc	ImBu	TabGo
fib	5	11	10	0	5	25	10	30	40	30
qsortapp	15	43	30	0	15	72	60	330	250	150
grammar	18	44	15	0	18	78	25	115	150	70
zebra	19	44	30	0	19	81	200	18500	2300	20000
money	16	59	30	0	16	96	200	1700	4000	1740
progeom	20	68	30	10	20	110	110	400	680	250
bid	50	111	50	0	50	196	130	640	1050	350
credit	58	105	50	0	58	218	110	510	2500	450
warplan	102	242	110	10	109	415	230	1670	5400	3000
read	88	344	115	10	90	533	830	2900	1500	1050
tictactoe	72	515	150	0	74	714	240	1050	1000	680
qplan	150	471	210	15	152	841	1060	6800	17000	8100

Name:	the name of the program,
#ClP:	number of clauses of the initial program,
#VarP:	number of variables of the initial program,
Trans:	time spent for translating the initial program into its boolean version,
Scc:	time spent for constructing, reducing and sorting the call graph,
#ClB:	number of clauses of the boolean program,
#VarB:	number of variables of the boolean program,
Bu:	time for computing the boolean model using bottom-up evaluation and scc,
TabScc:	idem using tabulation and scc,
ImBu:	idem by induced magic-sets bottom-up evaluation on the root of the call graph,
TabGo:	idem by tabulation on the root of the call graph.

Let us give some supplementary explanations.
The label *Bu* means that we compute the least boolean model using the Prolog bottom-up evaluator described in [1]. To obtain efficient computations, it is well known that one has to compute the fixpoint step by step, according to the strongly connected components (scc) of the call graph. We compute the same model using our tabling interpreter and the scc (*TabScc*). The timings show that bottom-up evaluation is faster. One drawback of our algorithm is the important number of satisfiability tests for boolean constraints. In the program called "zebra", these boolean constraints are surprisingly complex, hence the bad figure.
However, the fight is unfair as our tabling interpreter also computes calling modes for each predicate. In [1], the author presents a Prolog interpreter for goal directed bottom-up evaluation which simulates magic-sets transformation. We have coded the algorithm (*ImBu*). We compare it to our tabling interpreter, launched on the same main call of the program (*TanGo*). For the problem of computing the least boolean model and the calling modes, tabulation is faster.

6 Conclusion

We have precisely described an algorithm for tabulation in CLP, based on three operators: projection, satisfiability and entailment.
As shown in [4], table-based logic programming can also be useful in program analysis. We have implemented our algorithm for groundness analysis of logic programs. Our experiment shows that computation of the least boolean model is faster with bottom-up evaluation. But if one needs both the model and the calling modes of each predicates, then tabulation seems to be a good choice.
Finally, we give two possible extensions of this work. How to take frozen atoms into account appears to be an interesting non-trivial extension of tabulation to study. Combining tabulation and Constraint Handling Rules [5, 6] seems to be a promising generalization of the approach given in this paper. These are works in progress.

References

[1] M. CODISH. Efficient goal directed bottom-up evaluation of logic programs. In *1st International Workshop on Constraint Reasoning For Constraint Programming*, 1997. available at `www.cs.bgu.ac.il/~codish`.

[2] M. CODISH and B. DEMOEN. Analyzing logic programs using `prop`-ositional logic programs and a magic wand. *Journal of Logic Programming*, pages 249–274, 1995.

[3] P. CODOGNET. A tabulation method for constraint logic programming. *?*, 1996.

[4] S. DAWSON, C.R. RAMAKRISHNAN, and D.S. WARREN. Practical program analysis using general purpose logic programming systems - a case study. *Proc. of POPL'96*, pages 1–10, 1996.

[5] T. FRÜWIRTH. Introducing simplification rules. *Workshop on Rewriting and Constraints,Dagstuhl, Germany*, 1991.

[6] T. FRÜWIRTH. Theory and practice of constraint handling rules. *J. Logic programming*, 37:95–138, 1998.

[7] J. JAFFAR and M.J. MAHER. Constraint logic programming: a survey. *Journal of Logic Programming*, pages 503–581, 1994.

[8] F. MESNARD. Entailment and projection for clp(b) and clp(q) in sicstus prolog. *1st International Workshop on Constraint Reasoning For Constraint Programming*, 1997.

[9] K. SAGONAS, T. SWIFT, and D.S. WARREN. Xsb as an efficient deductive database system. *ACM SIGMOD Symposium on Management of Data*, 1994.

[10] H. TAMAKI and T. SATO. Old resolution with tabulation. *Proc. of the 3rd ICLP*, pages 84–98, 1986.

[11] D. TOMAN. Top-down beats bottom-up for constraint extensions of datalog. In *Proc. of International Logic Programming Symposium ILPS'95*. MIT Press, 1995.

[12] D.S. WARREN. Memoing for logic programs. *Communications of the ACM*, 30(3):93–111, 1992.

7 Appendix: A Detailed Proof of Completeness

We now give a detailed proof of the completeness of our tabulation algorithm w.r.t. LD-resolution.

Remark 7.1 *For all call* $tab(K, c\diamond\tilde{A}, Ref, Sol, Cls)$ *that returns the pair* $\langle Ref', Sol'\rangle$, *we have:* $dom(Sol) \subseteq dom(Sol')$ *and for all* $K \in dom(Sol), Sol(K) \subseteq Sol'(K)$.

Notation - Let $c_0 \diamond p_0(\tilde{x_0})$ be a resolvent, in the following, we denote by *initial call* the call:

$$tab(key(p_0(\tilde{x_0}), \exists_{-\tilde{x_0}} c_0), empty\_table, empty\_table, \langle\rangle)$$

We write $\langle Ref', Sol'\rangle \leftarrow tab(K, c \diamond \tilde{A}, Ref, Sol, Cls)$ as shorthand for a call $tab(K, c \diamond \tilde{A}, Ref, Sol, Cls)$ appearing in the computation of the initial call, which terminates and returns the pair $\langle Ref', Sol'\rangle$.

Theorem 7.1 (Completeness) *Let* $c \diamond \tilde{A}$ *be a resolvent and* $d\diamond$ *be an answer of* $c \diamond \tilde{A}$ *computed in* n *steps of LD-resolution. Let* $\langle Ref', Sol'\rangle \leftarrow tab(K, c\diamond\tilde{A}, Ref, Sol, Cls)$. *We have the following result:*

$$\exists Cls' \in I\!N^*, \exists \bar{K} \in dom(Sol) \text{ s.t. } \bar{K} \sim K \text{ and } \langle \exists_{-var(\bar{K})}(d \wedge e_{K=\bar{K}}), Cls'\rangle \in Sol'(\bar{K})$$

Proof (by induction on n)
•$n = 0$. In this case, $|\tilde{A}| = 0$ and $d \equiv c$. Let $\langle Ref', Sol'\rangle \leftarrow tab(K, c \diamond \tilde{A}, Ref, Sol, Cls)$. This call goes to line 2 and after lines 3 and 4, we have:

$$\bar{K} \sim K$$

and

$$s \equiv \exists_{-var(\bar{K})}(c \wedge e_{K=\bar{K}}) \equiv \exists_{-var(\bar{K})}(d \wedge e_{K=\bar{K}})$$

Then, if s is not already in $Sol(\bar{K})$, we update Sol (line 6). Hence the returned table Sol' verifies:

$$\langle s, Cls\rangle \in Sol'(\bar{K})$$

Induction Hypothesis: we suppose there exists $n \in I\!N$ s.t. for all resolvent $c \diamond \tilde{A}$, for all solution $d\diamond$ of $c \diamond \tilde{A}$ computed by LD-resolution with m steps ($m \leq n$), for all $\langle Ref', Sol'\rangle \leftarrow tab(K, c \diamond \tilde{A}, Ref, Sol, Cls)$, we have the following property:

$$\exists Cls' \in I\!N^*, \exists \bar{K} \in dom(Sol) \text{ s.t. } \bar{K} \sim K \text{ and } \langle \exists_{-var(\bar{K})}(d \wedge e_{K=\bar{K}}), Cls'\rangle \in Sol'(\bar{K})$$

• Now, let $c \diamond \tilde{A}$ be a resolvent and $d\diamond$ be a solution via $n+1$ steps of LD-resolution. Let $\langle Ref', Sol'\rangle \leftarrow tab(K, c \diamond \tilde{A}, Ref, Sol, Cls)$. We must prove that the property holds, namely:

$$\exists Cls' \in I\!N^*, \exists \bar{K} \in dom(Sol) \text{ s.t. } \bar{K} \sim K \text{ and } \langle \exists_{-var(\bar{K})}(d \wedge e_{K=\bar{K}}), Cls'\rangle \in Sol'(\bar{K})$$

We can rewrite $\tilde{A}$ as $\langle p(\tilde{x})\rangle * \tilde{A}'$. Let us decompose the $n+1$ steps of the derivation as follows:

$$
\begin{array}{rl}
1 \text{ step} & \left\{ \begin{array}{l} \leftarrow c \diamond p(\tilde{x}), \tilde{A}' \\ \Big\downarrow \qquad (fresh)cl : p(\tilde{y}) \leftarrow c'' \diamond \tilde{B} \in Prog, \\ \qquad\quad \models \exists (c \wedge c'' \wedge \tilde{x} = \tilde{y}) \end{array} \right. \\
\leq n \text{ steps} & \left\{ \begin{array}{l} \leftarrow c \wedge c'' \wedge \tilde{x} = \tilde{y} \diamond \tilde{B}, \tilde{A}' \\ \Big\downarrow * \end{array} \right. \\
\leq n \text{ steps} & \left\{ \begin{array}{l} \leftarrow c \wedge d \diamond \tilde{A}' \\ \Big\downarrow * \\ \leftarrow d' \diamond \end{array} \right.
\end{array}
$$

Remark 7.2 *We notice that $d\diamond$ is a solution of $c'' \wedge \tilde{x} = \tilde{y} \diamond \tilde{B}$ computed in less than n steps of LD-derivation. Let $c' \equiv \exists_{-\tilde{x}} c$, since $\models \exists(c \wedge c'' \wedge \tilde{x} = \tilde{y})$, we can say that $c' \wedge d\diamond$ is a solution of $c' \wedge c'' \wedge \tilde{x} = \tilde{y} \diamond \tilde{B}$ computed in less than n steps of LD-derivation.*

The call we consider is: $tab(K, c \diamond \langle p(\tilde{x})\rangle * \tilde{A}', Ref, Sol, Cls)$. The test on line 2 fails and we go to lines 13, 14 , 15 and 16 and so we have: $K' = key(p(\tilde{x}), c')$. Here we must deal with two cases.
Case 1: K' is a new key (line 17). We create the key $\bar{K}'$ which is a renaming of K' and we update Ref and Sol (lines 18, 19 and 20). Then the loop lines 20-23 is executed in which there is the call:

$$tab(K', c'' \wedge c' \wedge \tilde{x} = \tilde{y} \diamond \tilde{B}, Ref, Sol, \langle cl \rangle)$$

This call is supposed to terminate and returns the pair $\langle Ref_0, Sol_0 \rangle$. By induction hypothesis and the remark 7.2 we can say that:

$$\exists Cls_1 \in \mathbb{N}^* \text{ s.t. } \langle \exists_{-var(\bar{K}')}(c' \wedge d \wedge e_{K'=\bar{K}'}), Cls_1 \rangle \in Sol_0(\bar{K}')$$

Before we go on, let us simplify the constraint $\exists_{-var(\bar{K}')}(c' \wedge d \wedge e_{K'=\bar{K}'})$. Since $c' \equiv \exists_{-\tilde{x}} c$ and $\tilde{x} = var(K')$ we have:

$$\exists_{-var(\bar{K}')}(c' \wedge d \wedge e_{K'=\bar{K}'}) \equiv \exists_{-var(\bar{K}')}(c \wedge d \wedge e_{K'=\bar{K}'})$$

Then the computation goes to line 24 and we have a new call $tab(K, c \diamond \tilde{A}, Ref, Sol, Cls)$. This call goes to line 25 (the tests at lines 2 and 17 fail). At line 28, we store in S the solutions of $\bar{K}'$ we have already found, in particular, the solution we have just described above: $\langle \exists_{-var(\bar{K}')}(c \wedge d \wedge e_{K'=\bar{K}'}), Cls_1 \rangle$. And at line 31, the call $tab(K, c \wedge e_{K'=\bar{K}'} \wedge \exists_{-var(\bar{K}')}(c \wedge d \wedge e_{K'=\bar{K}'}) \diamond \tilde{A}', Ref, Sol, Cls * Cls_1)$ is executed and returns the pair $\langle Ref'', Sol'' \rangle$. At this point, let us do a last remark:

Remark 7.3 *The derivation branch described above shows that $d'\diamond$ is a solution of $c \wedge d \diamond \tilde{A}'$ computed in less than n steps of LD-derivation. Since $c \wedge d \wedge e_{K'=\bar{K}'} \models c \wedge e_{K'=\bar{K}'} \wedge \exists_{-var(\bar{K}')}(c \wedge d \wedge e_{K'=\bar{K}'})$, we can say that $d' \wedge e_{K'=\bar{K}'}\diamond$ is a solution of $c \wedge e_{K'=\bar{K}'} \wedge \exists_{-var(\bar{K}')}(c \wedge d \wedge e_{K'=\bar{K}'}) \diamond \tilde{A}'$ computed with less than n steps of LD-derivation.*

Once again the induction hypothesis can be applied:

- $d' \wedge e_{K'=\bar{K}'}$ is a solution of $c \wedge e_{K'=\bar{K}'} \wedge \exists_{-var(\bar{K}')}(c \wedge d \wedge e_{K'=\bar{K}'}) \diamond \tilde{A}'$ computed in less than n steps of LD-derivation (remark 7.3).

- $tab(K, c \wedge e_{K'=\bar{K}'} \wedge \exists_{-var(\bar{K}')}(c \wedge d \wedge e_{K'=\bar{K}'}) \diamond \tilde{A}', Ref, Sol, Cls * Cls_1)$ exists and returns the pair $\langle Ref'', Sol'' \rangle$.

- Conclusion: $\exists Cls' \in \mathbb{N}^*$ and $\bar{K} \in dom(Sol)$ s.t. $\langle \exists_{-var(\bar{K})}(d' \wedge e_{K'=\bar{K}'} \wedge e_{K=\bar{K}}), Cls' \rangle \in Sol''(\bar{K})$.

But $\exists_{-var(\bar{K})}(d' \wedge e_{K'=\bar{K}'} \wedge e_{K=\bar{K}})$ can be simplified in $\exists_{-var(\bar{K})}(d' \wedge e_{K=\bar{K}})$ because $var(\bar{K}) \cap var(\bar{K}') = \phi$. Finally, by remark 7.1, the property holds:

$$\langle \exists_{-var(\bar{K})}(d' \wedge e_{K=\bar{K}}), Cls' \rangle \in Sol'(\bar{K})$$

Case 2: the proof is similar to case 1. $\square$