A set-oriented meta-interpreter driven by a "relational" trace for deductive databases debugging[1]

Sarah Mallet and Mireille Ducassé

IRISA/INSA
Campus Universitaire de Beaulieu
CS 14315 F - 35042 Rennes Cedex, France
{smallet, ducasse}@irisa.fr

Abstract

The development of deductive databases requires tools, in particular for debugging. Deductive databases manage large quantity of data and, in general, in a set-oriented way. The existing systems of explanation for deductive databases do not take these constraints into account. We propose a tracing technique which consists of integrating a "relational" trace and an instrumented meta-interpreter using substitution sets. The relational trace efficiently gives precise information about data extraction from the relational database. The meta-interpreter manages substitution sets and gives explanation on the deduction. The expensive aspects of meta-interpretation are reduced by the use of the trace which avoids many calculations. The flexibility of meta-interpretation is preserved. It allows traces of different profiles to be easily produced.

1 Introduction

The development of deductive databases requires tools, in particular for debugging. These tools must be useful for the various kinds of users of deductive databases: those who develop the deductive part, those who enrich the data, those who use the database *in fine*. The tools must thus be flexible and be able to provide information and explanations at various levels of abstraction.

Compared to relational databases, the specificity of deductive databases is the deduction component which allows new data to be derived via rules. The tools must be able to explain the deductive "reasoning" which produced the results.

Compared to other paradigms of logic programming deductive databases have three essential specificities: *a large quantity of data*, *set-oriented management of these data*, and *termination*. Indeed, a deductive database program handles basic relationships with a great number of tuples. These tuples are in general retrieved several at the same time and not one at a time, thousands of tuples can be extracted from the base during a single access. Hence, substitutions are managed in a set-oriented way. Lastly, dedicated search strategies ensure that a request on a deductive database always terminates [9].

[1] Extended Abstracts of LOPSTR'98, Eighth International Workshop on Logic-based Program Synthesis and Transformation, 15-19 June 1998, Manchester, UK. Technical Report Series, Department of Computer Science, University of Manchester, ISSN 1361-6161. Report number UMCS-98-6-1. http://www.cs.man.ac.uk/cstechrep/titles98.html

The existing explanation systems for deductive databases [17, 12, 8] give information in the shape of proof trees. Although proof trees are often useful, this representation is not sufficient. Proof trees do not take into account the set-oriented aspect, they present results as if they were obtained a tuple at a time.

Furthermore, in Prolog more varied techniques of debugging exist [5]. In particular, a very flexible technique to produce deduction explanation consists in instrumenting meta-interpreters. This instrumentation can be easily adapted to users' needs. However, this technique is in general too inefficient. This is especially true in deductive database due to the large quantity of data. Another technique, particularly efficient, consists in instrumenting the code generated by the compiler. However, resulting traces are low-level and do not necessarily contain the information necessary to the explanations.

We propose a tracing technique which consists in integrating a relational trace with an instrumented set-oriented meta-interpreter. The relational trace efficiently gives precise and low-level information on the extraction of data from the relational database. The set-oriented meta-interpreter gives explanations on the deduction and allows more abstract views to be built. The expensive aspects of meta-interpretation are reduced by the use of the trace which avoids many calculations. In particular, the accesses to the relational database are not reproduced, the meta-interpreter uses the transitory information generated by the data extraction system, accessible via the relational trace. In addition, the advantages of meta-interpretation are preserved. In particular, the flexibility of the technique enables different traces to be easily produced, as illustrated at the end of the article.

We developped this technique for the Validity system, based on EKS [2], which uses the SLD-AL technique to ensure termination [15].

The main contribution of this article is the integration of a relational trace and a meta-interpreter, which is, to our knowledge, totally original. The practical impact of such a technique is important. As already mentioned, users of deductive databases have many different profiles. The flexibility at reasonable cost offered by our technique enables the debugger to be adapted, in particular, to end users. In general, they do not have an in-depth knowledge of computers but they nevertheless have to rely on the results of the system. They will better accept these results if they can understand how they were produced. One can conjecture that such a tool can help widen the acceptance of logic programming technologies.

Another merit of this paper is to show a practical application of theoretical results. Indeed, the technique is theoretically grounded. Firstly, in order to take into account the set-oriented aspects and the SLD-AL strategy, we adapt the multi-SLD operational semantics of Smith [10] and specify the *multi-SLD-AL* operational semantics. Secondly, we extend the meta-interpreters proposed by Lefebvre [6] to implement at the same time the SLD-AL method and the substitution sets.

In the following we first present the multi-SLD-AL operational semantics. We then describe the set-oriented meta-interpreter. The relational trace is briefly described, followed by its integration with the meta-interpreter. We briefly mention two different instrumentations of the meta-interpreter which generate either a set-oriented search tree, or a box-oriented trace. A comparison with related work finishes the article.

2 The multi-SLD-AL operational semantics

The Validity system is based on SLD-AL resolution (*SLD with test of Admissibility and resolution on Lemmas*) [15]. This resolution is an optimization of SLD resolution which cuts infinite branches from the search tree.

At present there is no formalization of the SLD-AL operational semantics. On the other hand, the *multi-SLD* resolution presented by Smith [10] takes into account data parallelism, similar to the database one, but formalizes an unrestricted search.

We therefore extend the multi-SLD operational semantics to take the test of admissibility and resolution on lemmas into account. Moreover, evaluation of built-in predicates is added. The new semantics is called *multi-SLD-AL*.

The following definition of the multi-SLD-AL resolution uses the syntax of the abstracted operational models of constraint programming languages. A derivation state is defined as $G \diamond C$ where G is a resolvent and C a substitution set, noted in a disjunctive way, $\diamond$ has the signification of the logical "and". $\square$ denotes the empty resolvent.

In the following, "=" indicates the unification of two atoms; R is a function which, applied to C, reduces substitutions and removes those which are inconsistent. If all substitutions are inconsistent, $R(C)$ is false.

A multi-SLD-AL derivation is defined for a query G_0 and a program P as a sequence of states $S_0, S_1, \ldots$, where $S_0 \equiv (G_0 \diamond true)$ and for $i > 0$, $S_{i-1} \Rightarrow S_i$ by the multi-SLD-AL resolution, defined below. S_{i-1} is called the *direct predecessor* of S_i. *Predecessor* is defined as the transitive closure of direct predecessor. A derivation finishes either when the substitution set is reduced to false, or when the resolvent is empty.

The preliminary concept of *non admissibility* of a state with respect to a set context is introduced[2]. A *call mode* of a predicate represents the arguments which are instantiated at call time.

Definition 1 (Non admissible state) *A state $S \equiv (G \diamond C)$ is non admissible wrt a subset $C_{Sub} \subseteq C$ if, S has a predecessor $S' \equiv (G' \diamond C')$ such that the selected atom A in S and the selected atom A' in the state S' have the same call mode and for each substitution $\theta_i \in C_{Sub}$, A and A' are variants.*

Definition 2 (Lemma) *A lemma is an atom built by instantiating a goal g with a substitution of the substitution set obtained at the end of the resolution of g.*

Lemmas are global to the whole search space of the multi-SLD-AL resolution.

Finally, the multi-SLD-AL resolution is defined as follows. It covers four cases. For a database predicate, the result of the unification of the selected atom and the tuples of the database is applied to each substitution. For a non admissible predicate, the predicate is solved on a set of lemmas. For a built-in predicate, the result of the evaluation of the predicate in the current context is recovered. Lastly, for a predicate defined using rules, called derived predicate, the result of the unification of the selected atom and the head of rule is applied to each substitution and the body of the rule is included into the current query.

---

[2]By extension in the following, we will speak of non admissible goals

Definition 3 (multi-SLD-AL resolution)
Current state $A_1, \ldots, A_n \diamond (E_1 \vee \ldots \vee E_m)$, *Selected atom* A_i,

- A_i ***defined from a database predicate***
  $T_1, \ldots, T_l$ *tuples of the database*
  If
  $$C \equiv \bigvee_{1 \leq k \leq l, 1 \leq j \leq m} (E_j \wedge (A_i = T_k)) \text{ with } R(C) \not\equiv false.$$
  Then
  $$A_1, \ldots, A_n \diamond (E_1 \vee \ldots \vee E_m) \Rightarrow A_1, \ldots, A_{i-1}, A_{i+1}, \ldots, A_n \diamond R(C)$$

- A_i ***non admissible***
  $Lem_1, \ldots, Lem_l$ *set of lemmas*
  If
  $$C \equiv \bigvee_{1 \leq k \leq l, 1 \leq j \leq m} (E_j \wedge (A_i = Lem_k)) \text{ with } R(C) \not\equiv false.$$
  Then
  $$A_1, \ldots, A_n \diamond (E_1 \vee \ldots \vee E_m) \Rightarrow A_1, \ldots, A_{i-1}, A_{i+1}, \ldots, A_n \diamond R(C)$$

- A_i ***defined from a built-in predicate***
  If
  $$C^{Sol} \text{ answer substitutions set of } (A_i \diamond E_1 \vee \ldots \vee E_m) \text{ with } C^{Sol} \not\equiv false.$$
  Then
  $$A_1, \ldots, A_n \diamond E_1 \vee \ldots \vee E_m \Rightarrow A_1, \ldots, A_{i-1}, A_{i+1}, \ldots, A_n \diamond C^{Sol}$$

- A_i ***derived predicate***
  Clause $H \leftarrow B_1, \ldots, B_k$
  If
  $$C \equiv \vee_{1 \leq j \leq m} (E_j \wedge (A_i = H)) \text{ with } R(C) \not\equiv false.$$
  Then
  $$A_1, \ldots, A_n \diamond (E_1 \vee \ldots \vee E_m) \Rightarrow A_1, \ldots, A_{i-1}, B_1, \ldots, B_k, A_{i+1}, \ldots, A_n \diamond R(C)$$

3 A Multi-SLD-AL meta-interpreter

The Prolog meta-interpreter, which we propose to fully implement the multi-SLD-AL resolution, is an extension of SLD-AL meta-interpreters introduced in [6] which did not take substitution sets into account.

The meta-interpreter can be divided in two parts: the search tree traversal and the resolution which builds the substitutions sets. These two parts are briefly described in the remaining of this section. The full paper will describe the meta-interpreter in detail.

A particular traversal of the search tree of the multi-SLD-AL resolution is defined. A first pass evaluates the query like a multi-SLD-AL query, holding account of the non admissible goals but without trying to solve them. A second pass solves the non admissible goals using lemmas.

The resolution of a goal is composed of two stages: initially, the atom to be evaluated is chosen then it is solved according to his type following the four cases of the multi-SLD-AL resolution previously defined. The evaluation of non admissible atoms is delayed.

In this case, the goal and its environment are stored then the procedure fails in order to continue the traversal of the search tree.

The function R, simplifying the substitution sets is integrated into the functions which calculate the new substitution sets of the states: when accessing the database, evaluating built-in predicates, unifying the head of a clause with the selected atom and solving the non admissible goals using lemmas.

The multi-SLD-AL evaluation requires to store some information: the produced lemmas, the sub-goals to detect non admissible sub-goals and the non admissible states.

4 The Validity "relational" trace

The Validity system is an object-oriented deductive database marketed by Next Century Media[3]. It is based on the *EKS* system [16]. It uses a relational database to handle facts.

At compilation time, each rule is transformed into a *dataflow sequence* which defines the flow of the data inside a rule via relations and operations on these relations (projection, joint). These *dataflow sequences* are then assembled in a *query evaluation graph*, composed of relational operations.

The execution instanciates this graph with the tuples actually present in the database. To handle non admissibility, the execution manages and traverses an internal dynamic structure: *the dataflow graph.*

During this execution, a "relational" trace is generated. It describes a succession of events which reflect the execution steps. These events are of two types: management of the graph and operations on the data.

The management of the graph gives information on the non admissible sub-goals. The events on the operations give a pointer to a table stored in the database containing a description of the relations. These tables are generated by the execution whether the trace is requested or not. They remain accessible after the execution. Thus, at debugging time the whole information related to database accesses is available without re-executing these accesses, and this at no extra cost in terms of space.

5 Driving the meta-interpreter with the trace

Thanks to the information contained in the trace non determinism can be reduced and substitution sets can be constructed.

Reducing the non determinism The trace is the image of an execution, where choices were already made. It is not necessary to remake them in the meta-interpreter.

The atom selected at a given resolution step is known thanks to the trace. The selection function then consists in retrieving this information. In the same way the type of this atom is present in the table of symbols. The case of the multi-SLD-AL resolution which will be used can now be fixed in advance. Furthermore, the rule selected for the resolution of a derived predicate is also present in the trace. In the same way, the lemmas used to solve non admissible goals can also be found in the trace. The meta-interpreter therefore does not need to manage lemmas, queries and non admissible states.

[3]http://www.nextcenturymedia.com

Reconstructing substitution sets One of the principal reasons to connect trace and meta-interpreter is to avoid re-accessing the database. The related information is stored in temporary relations accessible from within the trace, as described in the previous section.

For certain literals, it is not possible to obtain the new substitution set from the trace only. Indeed, in the case, for example, of several database predicates to be evaluated one after the other, the various operations to be carried out on the database are gathered in a single one and the obtained substitution set relates to the end of this operation. However, the meta-interpreter is able to distinguish the processing of each literal. In the current implementation, in a first stage, the result of the gathered operations is given. If the user asks for it, it will be possible in the second stage to give the detailed evaluations.

Synchronization of the meta-interpreter with the trace events The meta-interpreter and the trace must be traversed in parallel. The progression is synchronized. An event does not necessarily correspond to one literal only, but the literals to which it corresponds are well-known. The current event is changed as soon as all the related literals are treated.

6 Instrumentation of the meta-interpreter: abstract traces

Once meta-interpreter and trace are connected, the meta-interpreter can be instrumented in order to obtain different traces.

In the full article, two possible instrumentations of the meta-interpreter will be presented. The first one produces a graphical multi-SLD-AL tree. The second one produces a box-oriented trace.

The multi-SLD-AL tree A multi-SLD-AL tree is generated to be displayed by a graph editor. Nodes are associated with states of the multi-SLD-AL evaluation. The meta-interpreter creates a description file of the tree. The graph description language used is GML (Graph Modeling Language). The procedures of creation of node and edge generate GML data.

Box-oriented trace A box-oriented trace gives a sequence of events inspired by those proposed in [3]. The meta-interpreter is instrumented following the tracing methods of Prolog programs described in [14, 18]. The trace format that we chose contains eight ports including four for the non admissible sub-goals. These ports are *call, fail, exit, redo, call_na, fail_na, exit_na, redo_na*. The *_na* suffixe relates to non admissible goals. A tool such as Opium [4] can be used to filter and analyze this trace.

7 Related work

The first explanation system for deductive databases was developed for Dedex [7] by Wieland [17]. The suggested explanations are based on proof trees which can be visualized completely or partially. The execution of the initial system is not exploited since Wieland redefines another parallel system which generates information dedicated to the construction of the proof tree. This method involves a slow explanation system, disconnected from the initial system and a fixed type of trace.

The system *Explain* [1] was developed for CORAL [8]. Like the preceding system, the selected structure is the proof tree. The implementation of the system *Explain* consists in adding in CORAL a program which stores information on derivations during the evaluation of the query. This technique of generation of trace can be compared with that practised in Validity except that the information stored for Explain is completely dedicated to the construction of proof trees and does not keep any data on the operational semantics used.

The third system, designed by Specht [11] for LOLA [13], uses also the proof trees as explanations. The principle of implementation is a transformation of the user's program to insert trace generation. In opposition to other systems it does not modify the deductive engine. The transformed program is queried as usual using the deductive engine. This method, very simple to implement, makes the performances of evaluation fall with the increase of the size of the proof trees and the number of these trees to be handled (possibly thousands).

Proof trees give users declarative abstract views of executions, hiding away operational details. But sometimes, operational details can be relevant to fully understand the behavior of a system. Furthermore, proof trees do not reflect the set-oriented processing of the data.

The problems of these systems are, firstly, that they do not propose other types of trace and, secondly, that they do not exploit information which are generated during execution, then slowing down the system to generate their own information.

8 Conclusion

We presented a tracing technique which consists in integrating a relational trace with an instrumented set-oriented meta-interpreter. The relational trace reduces the non determinism of meta-interpretation and avoids at debugging stage many calculations, already performed at execution. The meta-interpreter allows different abstract views of the execution to be constructed, for example by instrumentation.

This technique has been illustrated on the Validity system. The meta-interpreter is theoretically grounded on the multi-SLD-AL operational semantics, which has been specified by extending the multi-SLD operational semantics to AL technique. The meta-interpreter is connected to the trace produced by Validity, some calculation functions and some choice functions of the meta-interpreter have been modified.

The flexibility at reasonable cost offered by our technique enables the debugger to be adapted, in particular, to end users.

Acknowledgements Alexandre Lefebvre, Laurent Vieille and Bernard Wappler sacrificed part of their time to explain the operation of Validity. Bernard Wappler has, moreover, modified the relational trace of Validity in order to facilitate integration with the meta-interpreter.

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*, LNCS 760.

[2] P. Bayer, A. Lefebvre, and L. Vieille. Architecture and design of the EKS deductive database system. *VLDB Journal*, 1993.

[3] L. Byrd. Understanding the control flow of Prolog programs. In S.-A. Tärnlund, editor, *Logic Programming Workshop*, Debrecen, Hungary, 1980.

[4] M. Ducassé. Opium: An extendable trace analyser for prolog. *The Journal of Logic programming*, 1999. To appear in the special issue on Synthesis, Transformation and Analysis of Logic Programs, A. Bossi and Y. Deville (eds).

[5] M. Ducassé and J. Noyé. Logic programming environments: Dynamic program analysis and debugging. *The Journal of Logic Programming*, 19/20:351–384, May/July 1994.

[6] A. Lefebvre. *Evaluation de requêtes dans les bases de données déductives : aspects théoriques et pratiques*. PhD thesis, Université René Descartes of Paris V, France, juin 1991.

[7] 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.

[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. *Journal of logic programming*, pages 125–149, 1995.

[10] Donald A. Smith. MultiLog: Data Or-Parallel Logic Programming. *Journal of Logic Programming*, 29:195–244, 1996.

[11] G. Specht. *Source-to-source Transformationen zur Erklärung des Programmverhaltens bei deduktiven Datenbanken*. PhD thesis, Technische Universität München, Juni 1992.

[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*, 1993. Publication IRISA.

[13] G. Specht and 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.

[14] L. Sterling and E. Shapiro. *The Art of Prolog, second edition*. MIT Press, Cambridge, Massachusetts, 1994. ISBN 0-262-19338-8.

[15] L. Vieille. *Bases de données déductives : évaluation et optimisation de programmes logiques récursifs*. PhD thesis, Université Paris 6, septembre 1988.

[16] L. Vieille, P. Bayer, V. Küchenhoff, and A. Lefebvre. EKS-V1, a short overview. In *Workshop on Knowledge Base Management System*, Boston, USA, July 1990. AAAI-90.

[17] 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.

[18] L.Ü. Yalcinalp. *Meta-programming for knowledge based systems in Prolog*. PhD thesis, Case Western Reserve University, Cleveland, Ohio 44106, August 1991. Technical Report TR 91-141.