A Framework for Goal-Directed Query Evaluation with Negation

Stefan Brass

Martin-Luther-Universität Halle-Wittenberg

Germany

Inhalt

Motivation (1)

- SLDMagic (developed by the author) is a method for goal-directed bottom-up query evaluation in deductive DBs.
- It is a competitor to the well-known Magic Set Method.

  Sometimes, SLDMagic is better, e.g. for tail recursions. + more advantages.
- Using SLDMagic and new ideas for bottom-up evaluation, we recently reached a factor 700 speedup over XSB in the tc-bf benchmark (part of OpenRuleBench).

  Small print: The system is still under development so that some parts were manually crafted for executing the example. For other parts, already an automatic translation from Datalog to C++ was used.
- Thus SLDMagic is still interesting.
- But: SLDMagic cannot handle negation.

Motivation (2)

- Together with Jürgen Dix, I also investigated negation semantics based on elementary program transformations.
- SLDMagic was not defined based on program transformations, but as SLD-resolution, it can be seen as doing unfolding.
- Now the (long-term) goal is to combine these two approaches to get a fast goal-directed query evaluation method based on such program transformations.

  This approach directly computes only WFS, it might be a useful precomputation step for other semantics permitting the transformations.

- My previous attempts failed because I wanted a source-to-source transformation like Magic Sets/SLDMagic.

  But the result would still have negation, and the translation might even make the evaluation of negation more difficult. Idea: Extend target language!

Inhalt

Program Transformations for Query Evaluation (1)

- The elementary program transformations studied in our papers on negation semantics worked on ground programs.
- In order to use them for directly computing answers, we must lift the transformations to the non-ground level.
- Most modern semantics are based on the ground instantiation of the program, i.e. $\mathcal{S}(P) = \mathcal{S}\big(ground(P)\big)$.
- Therefore, if a non-ground transformation corresponds to (possibly multiple) transformations on the ground instantiation, and the semantics permits the ground case, also the non-ground version is equivalence-preserving.

Program Transformations for Query Evaluation (2)

- The query is represented as a rule with a special predicate in the head:

$$answer(X) \leftarrow p(X).$$

- The real query goal is the body of the rule.
- Because the query variables appear in the head, no separate bookkeeping of the substitution for them is needed.
- The program transformation "unfolding" corresponds to SLD-Resolution.
  - E.g. unfolding with $p(X) \leftarrow q(a, X)$:

$$answer(X) \leftarrow q(a, X).$$

  - E.g. unfolding with $q(a, b)$ gives a solution:

$$answer(b).$$

Program Transformations for Query Evaluation (3)

- For goal-directed query evaluation, we cannot work with the entire program.
- The “relevance” property studied by Dix and Müller ensures that it is sufficient to look only at literals which are reachable from the query via the call-graph.

  WFS has relevance, the stable model semantics has not, but it might be possible to treat “odd loops over negation” separately ($\rightarrow$ Galliwasp).

- Instead of starting with all rules and removing irrelevant ones, we start with only the query and ensure that transformations (using rules from the program) remain applicable as long as there is a relevant rule.

  So relevance is formally applied at the end, after our other transformations have removed many edges from the call graph.

The Framework (1)

- A rule is variable-normalized iff it contains only the variables $X_1, X_2, \ldots$, numbered in the order of first occurrence. The function $std(\ldots)$ normalizes the variables.

  We do not want multiple rules which differ only in the names of the variables.

- A computation state is a pair (R, D) of sets of variable-normalized rules such that $D \subseteq R$. A rule in $R - D$ is called active, a rule in D is called deleted.

  By keeping deleted rules, we avoid non-termination by entering a rule again.

  Some transformations also need to know that a rule was previously considered.

- Let the query Q be $answer(X_1, \ldots, X_m) \leftarrow B_1 \wedge \cdots \wedge B_n$. The initial computation state is (R_0, D_0) with $R_0 := \{std(Q)\}$ and $D_0 := \emptyset$.

The Framework (2)

- We define transformations between computation states: $(R, D) \mapsto (R', D')$.
- Then an implementation can follow any sequence of computation states
$$(R_0, D_0) \ \mapsto \ (R_1, D_1) \ \mapsto \ \cdots \ \mapsto \ (R_n, D_n)$$
from the initial state to a final state, i.e. a state where no further transformation is applicable.
- The computed answers are then the tuples $(c_1, \ldots, c_m)$, such that $answer(c_1, \ldots, c_m) \in R_n$.
- The transformation system is not confluent, i.e. one can arrive at different final states, but they all contain the correct answers.

Termination

- For each transformation $(R, D) \mapsto (R', D')$ it holds that $R \subseteq R'$ and $D \subseteq D'$, and at least one inclusion is proper.
- The length of the occurring rules is bounded:
  - Unfolding can be applied to a recursive body literal only if it is the last/only positive body literal.

    Negative body literals cannot have additional variables because all occurring rules are range-restricted.
  - If a rule contains more than one recursive positive body literal, or this execution sequence seems sub-optimal, one can mark the positive body literal as $call(B)$.
  - Such body literals are solved in a subproof.

    Just as SLDNF-resolution calls itself recursively for negative body literals.

Transformation List

- Positive body literals:
  - Unfolding
  - Deletion after complete unfolding
- Negative body literals:
  - Complement call
  - Positive reduction
  - Negative reduction
- Call literals (simpler version):
  - Start of subproof
  - Return
  - End of subproof (includes loop check)

Inhalt

An Example (1)

- Program:

$$\begin{array}{l} odd(Y) \leftarrow succ(X, Y) \land \neg odd(X). \\ succ(0, 1). \\ succ(1, 2). \\ \quad \vdots \\ succ(999999, 1000000). \end{array}$$

- Initial state (with query):

$$R_0 := \{answer(yes) \leftarrow odd(1).\},\ D_0 := \emptyset$$

- Unfolding with rule about *odd*:

$$\begin{array}{lll} R_1 - D_1 & := & \{answer(yes) \leftarrow succ(X_1, 1) \land \neg odd(X_1).\} \\ D_1 & := & \{answer(yes) \leftarrow odd(1).\} \end{array}$$

Actually, unfolding with one program rule/fact, and removing the original rule after complete unfolding are different steps to give more flexibility.

An Example (2)

- Unfolding with $succ(0, 1)$:

$$R_2 - D_2 := \{answer(yes) \leftarrow \neg odd(0).\}$$
$$D_2 := \{answer(yes) \leftarrow odd(1).$$
$$answer(yes) \leftarrow succ(X_1, 1) \land \neg odd(X_1).\}$$

- Complement Call (setting up a subquery):

$$R_3 - D_3 := \{answer(yes) \leftarrow \neg odd(0).$$
$$odd(0) \leftarrow odd(0).\}$$
$$D_3 := D_2$$

- Unfolding with rule about *odd*:

$$R_4 - D_4 := \{answer(yes) \leftarrow \neg odd(0).$$
$$odd(0) \leftarrow succ(X_1, 0) \land \neg odd(X_1).\}$$
$$D_4 := D_3 \cup \{odd(0) \leftarrow odd(0)\}$$

An Example (3)

- Unfolding deletes the rule since there is no fact $succ(X_1, 0)$:

$$R_5 - D_5 := \{answer(yes) \leftarrow \neg odd(0).\}$$
$$D_5 := D_4 \cup \{odd(0) \leftarrow succ(X_1, 0) \land \neg odd(X_1)\}$$

- Positive Reduction evaluates the negative body literal to "true" (deletes it), since there is no rule with matching head in in active part:

$$R_6 - D_6 := \{answer(yes).\}$$
$$D_6 := D_5 \cup \{answer(yes) \leftarrow \neg odd(0).\}$$

- $answer(yes)$ has been proven.
- No further transformation is applicable.

Inhalt

Conclusions

- This is work in progress. An implementation will have to
  - choose a transformation when several can be applied for a given program, and
  - use data structures for representing rules or rule sets so that the transformations can be applied efficiently.
- With certain choices, one probably arrives at known algorithms, such as SLG-resolution.

  That is not necessarily bad (improved understanding, comparison of different methods, faster tail recursion).
- In deductive databases, there is an important distinction between a (usually small) set of rules and a large set of facts. My focus will be on precomputing as much as possible at "compile time", when only the rules are known (→SLDMagic).