Generating Explanation Trees even for Negations in Deductive Database Systems

Günther Specht
Technische Universität München
Institut für Informatik
Orleansstr. 34
D-81667 München
email: specht@informatik.tu-muenchen.de

Abstract:
Although there were enormous research efforts on explanation and debugging tools for Prolog in the last years, good tools for bottom-up evaluating logic programming systems are still missing, since the generation of the underlying proof trees matches several problems in presence of negation and recursion. This paper wants to fill the gap and presents a source-to-source transformation to compute complete proof trees for bottom-up evaluating systems, so that advanced techniques, developed for top-down systems, will once be usable for bottom-up systems as well. The presented technique is implemented and in use in the deductive database system LOLA.

1. Introduction:

Most explanation and debugging tools for logic programs are based on proof trees (also called explanation trees). If a logic program is not evaluated top-down and "tuple at a time" like Prolog does, but bottom-up and "set at a time" like most of the deductive database systems (e.g. LOLA, LDL, NAIL, ADITI, CORAL, IDEA, etc.) do it, it is rather difficult to generate proof trees, especially in the case of negation, since restrictions like range restriction, stratification and safe negation are required in the bottom-up approach. That is the reason why most of the currently available deductive databases include no or only rudimental explanation facilities.  

This paper wants to fill that gap and presents a new technique to compute complete proof trees for bottom up evaluating logic programming systems (henceforth called deductive databases): A source-to-source transformation converts a given logic program into a program that includes explanation or trace information. Even recursive and negated subgoals can be explained correctly. "Why-not" queries are possible too, showing why an expected result cannot be deduced from the given rules and facts. The computed proof trees can either be shown directly as explanation trees within a graphical user-interface, allowing zooming and shrinking operations, or be used as input for advanced explanation and debugging tools as developed in the top-down environment (e.g. [1, 2, 4, 9, 11]).

The paper is organized as follows: Section 2 motivates, why we select a source-to-source transformation as best technique. In section 3 the positive explanation transformation is presented and in section 4 we solve the problems arising, when negation occurs. Finally section 5 shows how one can ask why-not queries using this technique.

1 for an overview see e.g. [6].
2. Why using a Source-to-Source Transformation?

The architecture of a deductive database system looks in general as follows: A logic program and a query are translated by a compiler into a relational algebra expression, including joins, selections, semiautomatic relations, etc. This code is evaluated in a runtime system. The evaluation mode is bottom-up and set at a time like most of the deductive database systems (e.g. LOLA, LDL, NAIL, ADITI, CORAL, IDEA, etc.)

Generating Explanation Trees even for Negations in Deductive Database Systems

Günther Specht

Technische Universität München
Institut für Informatik

email: specht@informatik.tu-muenchen.de

D-81667 München

Abstract:

Most explanation and debugging tools for logic programs are based on proof trees (also called explanation trees, normal form or explanation technique). But a great number of tools for logic programming systems do not allow this technique.

The method has several advantages over existing tools: It is modular, it is transformed, so that the code-generator is called. At that level we have all collected informations. But if we do so, we obtain an enormous drawback: If we use rule-rewriting or optimization techniques, like the magic set elimination or trace information, now, later rule-rewriting techniques don't matter any more. Considering which might be the best place to integrate a proof tree generator in the system, one immediately thinks about the generation of the underlying proof trees (commonly called operator graph) before the code-generator is called. In that area we have all collected informations. If we do so, we obtain an enormous drawback: If we use rule-rewriting or optimization techniques, like the magic set elimination, then we get an enormous drawback: If we use these techniques, then we obtain an enormous drawback. If we do so, we obtain an enormous drawback.

That can be avoided by putting the proof tree generator as a source-to-source transformation right at the source code. This technique is implemented and in use in the deductive database system LOLA.

Although there were enormous research efforts on explanation and debugging tools for Prolog in the last years, good tools for bottom-up evaluating logic programming systems are still missing, since the development of source-to-source transformation by Prolog developers and by LDL, NAIL, ADITI, CORAL, IDEA, etc. do it, it is rather difficult to generate proof trees, especially in the case of negation, since restrictions like range restriction, stratification and safe negation are required in the bottom-up approach. That is the reason why most of the currently available deductive database systems allow for no or only rudimental explanation facilities.

I.e. in the result:

1. independence from the underlying deductive database system: I.e. this proof tree generator is usable for every bottom-up evaluating logic programming system.

2. independence from the evaluation system (runtime system).

3. Independence from rule-rewriting and optimization techniques and

4. independence from the internal representation of the program in the compiler.

The paper is organized as follows: Section 2 motivates, why we select a source-to-source transformation as best technique. In section 3 the positive explanation transformation is presented and in section 3 the problems arising, when negation occurs. Finally section 4 shows how one can ask why-not queries using this technique.

1 For an overview see e.g. [6].

2. Why using a Source-to-Source Transformation?

The paper is organized as follows: Section 2 motivates, why we select a source-to-source transformation as best technique. In section 3 the positive explanation transformation is presented and in section 4 we solve the problems arising, when negation occurs. Finally section 5 shows how one bottom-up and set-diff is implemented and in use in the deductive database system LOLA.

Considering which might be the best place to integrate a proof tree generator in the system, one immediately thinks about the final transformation right at the code-generator is called. At that level we have all collected informations. But if we do so, we obtain an enormous drawback: If we use not-having optimization techniques, like the magic set technique, then we can't do the transformation on the original one.

That can be avoided by putting the proof tree generator as a source-to-source transformation right at the compilation which transforms the logic program into another one, containing additional attributes.

That method has several advantages:

- independently from the evaluation information. Now all kind of evaluation technique can be mapped to the tool.
- independently from rule-rewriting and optimization techniques and
- independently from the internal representation of the program in the compiler.
- independence from rule-rewriting and optimization techniques and
- independence from the situation in system run time environment.

I.e. in the result:

1. Introduction:

1.1 Problem statement and literature costs. The source programs proposed have computing drawbacks do exist.

1.2. Proof Tree Transformation for positive Subgoals. The idea is to transform the source program into another one, containing additional informations. Now all kinds of evaluation techniques can be mapped to the tool.

Although there were enormous research efforts on explanation and debugging tools for Prolog in the last years, good tools for bottom-up evaluating logic programming systems are still missing, since the semantically correct proof trees are generated even in presence of negation and especially in the case of negation, since restrictions like range restriction, stratification, and if we know already the derivation S2 of directflight and if we know already the derivation S1 of directflight, then we can build up the derivation S1 of directflight and if we know already the derivation S2 of directflight, then we can build up the derivation S1 of directflight. A logic program and a transformation term in the transformation system looks in general as follows: A logic program and a transformation term, using the subterms, using their attributes, inserting this term into a term containing the head of the head.

Generating Explanation Trees even for Negations

In Deductive Database Systems

Grimmer specifs

<table>
<thead>
<tr>
<th>flight (munich, vancouver)</th>
<th>flight (munich, frankfurt)</th>
<th>flight (vancouver, ny)</th>
<th>flight (frankfurt, ny)</th>
<th>flight (ny, vancouver)</th>
</tr>
</thead>
<tbody>
<tr>
<td>directflight (munich, frankfurt)</td>
<td>directflight (vancouver, ny)</td>
<td>directflight (frankfurt, ny)</td>
<td>directflight (ny, vancouver)</td>
<td>directflight (munich, frankfurt)</td>
</tr>
</tbody>
</table>

The architecture of a deductive database system looks in general as follows: A logic program and a transformation term, using the subterms, using their attributes, inserting this term into a term containing the head of the head.

The same transformation has to be done for facts as well.

The same transformation has to be done for facts as well.

That can be avoided by putting the proof tree generator as a source-to-source transformation right at the compilation which transforms the logic program into another one, containing additional attributes.

That method has several advantages:

- independently from the evaluation information. Now all kind of evaluation technique can be mapped to the tool.
- independently from rule-rewriting and optimization techniques and
- independently from the internal representation of the program in the compiler.
- independence from rule-rewriting and optimization techniques and
- independence from the situation in system run time environment.

I.e. in the result:

1. Introduction:

1.1 Problem statement and literature costs. The source programs proposed have computing drawbacks do exist.

1.2. Proof Tree Transformation for positive Subgoals. The idea is to transform the source program into another one, containing additional informations. Now all kinds of evaluation techniques can be mapped to the tool.

Although there were enormous research efforts on explanation and debugging tools for Prolog in the last years, good tools for bottom-up evaluating logic programming systems are still missing, since the semantically correct proof trees are generated even in presence of negation and especially in the case of negation, since restrictions like range restriction, stratification, and if we know already the derivation S2 of directflight and if we know already the derivation S1 of directflight, then we can build up the derivation S1 of directflight. A logic program and a transformation term in the transformation system looks in general as follows: A logic program and a transformation term, using the subterms, using their attributes, inserting this term into a term containing the head of the head.

Generating Explanation Trees even for Negations

In Deductive Database Systems

Grimmer specifs

<table>
<thead>
<tr>
<th>flight (munich, vancouver)</th>
<th>flight (munich, frankfurt)</th>
<th>flight (vancouver, ny)</th>
<th>flight (frankfurt, ny)</th>
<th>flight (ny, vancouver)</th>
</tr>
</thead>
<tbody>
<tr>
<td>directflight (munich, frankfurt)</td>
<td>directflight (vancouver, ny)</td>
<td>directflight (frankfurt, ny)</td>
<td>directflight (ny, vancouver)</td>
<td>directflight (munich, frankfurt)</td>
</tr>
</tbody>
</table>

The architecture of a deductive database system looks in general as follows: A logic program and a transformation term, using the subterms, using their attributes, inserting this term into a term containing the head of the head.

The same transformation has to be done for facts as well.

1. Introduction:

1.1 Problem statement and literature costs. The source programs proposed have computing drawbacks do exist.

1.2. Proof Tree Transformation for positive Subgoals. The idea is to transform the source program into another one, containing additional informations. Now all kind of evaluation techniques can be mapped to the tool.

Although there were enormous research efforts on explanation and debugging tools for Prolog in the last years, good tools for bottom-up evaluating logic programming systems are still missing, since the semantically correct proof trees are generated even in presence of negation and especially in the case of negation, since restrictions like range restriction, stratification, and if we know already the derivation S2 of directflight and if we know already the derivation S1 of directflight, then we can build up the derivation S1 of directflight. A logic program and a transformation term in the transformation system looks in general as follows: A logic program and a transformation term, using the subterms, using their attributes, inserting this term into a term containing the head of the head.

### Generating Explanation Trees even for Negations

Most explanation and debugging tools for logic programs are based on proof trees (also called

<table>
<thead>
<tr>
<th>r1:</th>
</tr>
</thead>
<tbody>
<tr>
<td>p(X) :- r(X,Y), s(Y,Z).</td>
</tr>
<tr>
<td>r2:</td>
</tr>
<tr>
<td>p(X) :- t(X,Y), s(Y,Z).</td>
</tr>
<tr>
<td>r3:</td>
</tr>
<tr>
<td>q(X) :- t(X,Y), s(Y,Z).</td>
</tr>
<tr>
<td>r4:</td>
</tr>
<tr>
<td>new explain-not predicate. But how must the predicate explain_not_p be defined? Let the original</td>
</tr>
<tr>
<td>clause be:</td>
</tr>
<tr>
<td>p(X) :- ( q(q(q(r1(S1, \neg(q(q(r2(S1,S2),X)),X)),X)) )).</td>
</tr>
</tbody>
</table>

We evaluate a transformed logic program with more rules and facts we get e.g. the following proof |

directflight(munich, frankfurt).

directflight(ny, vancouver).

Consider why then \( \neg p(y) \) is required. For even if the rule \( p(y) \) is not derivable, the rule \( \neg p(y) \) |

1. **Introduction:**

   In many existing deductive database systems, negation is not supported, but just a negation as a |
   proposition as the best technique. In section 3 the positive explanation transformation is presented and in |
   section 4 the negative one. This transformation converts a given logic program into a program that includes |
   explain-not rules and facts. In order to explain a query, the system uses an extra explanation generating |
   transformation that can be either built-in or a partial source-to-source transformation. If there is only |
   one unexplained subgoal, then the query is not explainable. The new explanation generator is a source-to-source |
   transformation that can be used with any deductive database system, including LDL.看不到更多内容。
1. Introduction

In order to develop a strategy for transforming logic programs, we first need to define the conditions that must
be satisfied by the transformation. We then present a source-to-source transformation that meets these conditions.

2. Source-to-Source Transformation for Transforming Subgoals

The transformation is defined as a set of rules that can be applied to a given logic program. Each rule takes a
rule as input and produces a new rule as output. The transformation is designed to be applied in a bottom-up
manner, starting with the bottom-most subgoal and working its way up to the goal.

3. Applications of the Transformation

The transformation has several applications, including:

- Generating Explanation Trees for Negation
- Debugging Logic Programs
- Optimization Techniques
- Transformation of Logic Programs

4. Conclusion

The transformation provides a powerful tool for manipulating logic programs. It can be used to generate
explanation trees, debug logic programs, optimize logic programs, and transform logic programs in a
variety of ways. Further research is needed to explore the full potential of this transformation.
1. What Is Tree Transformation for Transfer Saturation

Tree transformation has been used in deductive database systems to optimize and debug programs. It involves transforming a negatively-nested tree into a positively-nested tree, allowing for more efficient execution.

2. Generating Explanation Trees even for Negations

An explanation tree is generated by transforming the original tree into a positively-nested form. This allows for the generation of explain-not predicates, which can then be used to debug the program.

3. Proof-Tree Transformation for Positive Subgoals

Proof-tree transformation is a technique used to transform proof trees into a form that can be more easily debugged. This involves converting the proof tree into a positively-nested form, which allows for the generation of explain-not predicates.

4. Why Not Queries

Not queries arise when the negation operator is used in a program. They are generated by transforming the original program into a positively-nested form, allowing for the generation of explain-not predicates.

5. Conclusion and Outlook

The techniques presented in this paper allow for the generation of explain-not predicates, which can be used to debug programs. Further work is needed to improve the efficiency of these techniques and to develop new techniques for debugging.

Abstract: This paper presents a technique for generating explanation trees even for negations. The technique involves transforming the original program into a positively-nested form, allowing for the generation of explain-not predicates. The technique has been implemented as a source-to-source transformation right at the Prolog level. The tool is particularly useful for deductive database systems with bottom-up evaluation mode.

Keywords: Prolog, deductive database systems, explain-not predicates, debugging, source-to-source transformation.