Bottom-Up Query Evaluation in Extended Deductive Databases

Dem Fachbereich Mathematik
der Universität Hannover
zur Erlangung der Lehrbefugnis (venia legendi)
für das Fachgebiet Informatik
vorgelegte Habilitationsschrift

von

Stefan Brass

1996
My current address is:

Universität Hannover, Institut für Informatik,
Lange Laube 22, D-30159 Hannover, Germany,
sb@informatik.uni-hannover.de

This document was last updated on: June 12, 1996.
Contents

1 Introduction 1

2 Horn-Clause Datalog 27
   2.1 Logic and Databases ...................................... 27
   2.2 Syntax and Semantics of Datalog .......................... 38
   2.3 Bottom-Up Query Evaluation ............................... 52

3 Goal-Directed Bottom-Up Evaluation 67
   3.1 The “Magic Set” Transformation ............................ 71
   3.2 An Improved Magic Set Technique .......................... 83

4 Negation as Failure 101
   4.1 Datalog with Negations .................................... 106
   4.2 A Framework for Studying Semantics ....................... 122
   4.3 Bottom-Up Query Evaluation ............................... 149

5 Reasoning with Disjunctions 167
   5.1 Extended Bottom-Up Query Evaluation ..................... 169
   5.2 Application: Computation of Stable Models ............... 180

6 Conclusions 187

Bibliography 189

Index 203
Acknowledgements

First of all, I would like to thank my doctor father UDO LIPECK for giving me the opportunity to perform the research presented here and for numerous helpful comments. I believe that his influence can be seen in every chapter. For instance, there are notions from term rewriting used in Section 4.2, and he always recommended using meta-interpreters (which I did in Sections 3.2 and 5.1). Also Section 5.1 is based on joint research, and Section 5.2 is influenced by an earlier joint paper.

It is a pleasure for me to thank JÜRGEN DIX for two years of joint research, which has led to the results presented in Chapter 4 (and more). I really enjoyed working together with JÜRGEN, and our cooperation has been very fruitful.

I would also like to thank the students in my courses on deductive databases and logic programming. I learnt a lot by giving these lectures. DIRK HILLBRECHT and MICHAEL SALZENBERG have implemented query evaluation based on the residual program and hyperresolution.

My brother PETER found a number of typing errors and made other valuable suggestions. Last but not least, many thanks to my parents for all kinds of support and finally getting me to this point (which was not always easy).
Chapter 1

Introduction

The goal of this work is to develop query evaluation algorithms for deductive databases extended by nonmonotonic negation and disjunctions. This topic lies in between the three fields of automated theorem proving, nonmonotonic reasoning, and databases. There are three specific questions treated here:

- First, even in the standard case of Horn clauses without negation, bottom-up query evaluation has not reached the efficiency of top-down query evaluation in practice [SSW94]. What are the reasons for this, and can the situation be improved?
- Second, there is a large number of proposed semantics for nonmonotonic negation. How different do query evaluation algorithms for them have to be, and are there any connections between semantical properties and possible ways to compute them?
- Third, reasoning with disjunctive rules is currently far less efficient than reasoning with Horn clauses. In fact, it seems that the communities studying the two topics are nearly disjoint. So, how far is it possible to use standard techniques known for Horn clauses in more general query evaluation algorithms?

Some Bits of History

The field of mathematical logic has strongly influenced many branches of computer science, for instance the early results on formal languages and computability were developed by logicians (after all, there was no computer science by that time).

One of the main goals of mathematical logic is to represent knowledge in some formal way, suitable for algorithmic treatment. In computer science, databases were developed to store information. At the beginning, databases did not have much theory and were only a collection of subprograms to access files. It was a major step forward when CODD proposed 1970 the relational data model [Cod70], where a database state is nothing else than a first order interpretation (without function symbols) known in logic for a long time. This data model was first only theoretically defined, and was criticized for being un-implementable. However, a decade later first commercial relational database systems appeared, and two decades later they are the state of the art.
In the programming language community, logic lead to the development of Prolog ("PROgramming in LOGic") by COLMERAUER, ROUSSEL and others in the early 1970s. KOWALSKI theoretically explained the possibility to use predicate logic as a programming language [Kow74, vEK76].

Now the idea of deductive databases is to integrate the possibilities of relational databases and Prolog:

![Diagram of relationships between Logic, Relational Databases, Prolog, Artificial Intelligence, Object-Orientation, and Deductive Databases]

So a deductive database consists of

- a relational database, which defines a number of relations (or predicates) "extensionally", by enumerating the tuples contained in the relation, and
- a logic program, which defines a number of relations/predicates "intensionally", by giving a set of defining rules (formulas of a restricted kind).

Of course, there are also alternative possibilities to describe deductive databases:

- One can say that a deductive database is simply a relational database with a new query language (Datalog instead of SQL), and with the view mechanism extended to allow recursive definitions.
- Or one can say that a deductive database is nothing else than a logic program with a large number of facts (corresponding to the tuples in the database), possibly treated in some special way in the implementation.
- Finally, a deductive database can be seen as an automated theorem prover, which allows only special kinds of formulas, but very many of them.

Of course it is difficult to single out one point in history, where the field of deductive databases "has started". Maybe we should mention the following events:

- A milestone in the development of automated theorem proving was the invention of the resolution method by ROBINSON 1963 [Rob69]. Subsequently, so-called "question-answering systems" were developed (by GREEN [Gre69] and others [Min88b]), which tried to extract useful information (bindings of variables) from proofs. They can be seen as predecessors of deductive databases.
- VAN EMDEN and KOWALSKI introduced 1976 the minimal model of a set of definite Horn clauses [vEK76]. In logic programming it was superseded by CLARK’s
<table>
<thead>
<tr>
<th>Year</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>1963</td>
<td>Resolution-Method for Automated Theorem Proving</td>
</tr>
<tr>
<td>1969</td>
<td>Question Answering Systems</td>
</tr>
<tr>
<td>1970</td>
<td>Relational Datamodel</td>
</tr>
<tr>
<td>1973</td>
<td>Prolog</td>
</tr>
<tr>
<td>1976</td>
<td>Minimal Model / Immediate Consequence Operator</td>
</tr>
<tr>
<td>1977</td>
<td>Conference “Logic and Databases”</td>
</tr>
<tr>
<td>1979</td>
<td>Relational Algebra with Fixpoint Operator</td>
</tr>
<tr>
<td>1982</td>
<td>Fifth Generation Project</td>
</tr>
<tr>
<td>1986</td>
<td>“Magic Set” Method / Perfect Model / Systems</td>
</tr>
<tr>
<td>1989</td>
<td>Textbooks on Deductive Databases</td>
</tr>
</tbody>
</table>

Figure 1.1: How the Field of Deductive Databases has Started

completion, but it revived later as the standard semantics of deductive databases and was generalized in various ways to handle negation. In [vEK76] also the immediate consequence operator and the fixpoint semantics were introduced, which are the foundation of bottom-up evaluation. As noted in [vEK76], the immediate consequence operator is nothing else than a special case of hyperresolution, which was introduced 1965 by Robinson.

- In 1977 Gallaire, Minker, and Nicolas organized a workshop on “Logic and Databases” in Toulouse and published subsequently a book with contributions from that workshop [GM78]. Among other important contributions, also two formalizations of nonmonotonic negation were presented at this conference: The completed database by Clark [Cla78] and the closed world assumption by Reiter [Rei78].

- In 1979, a paper by Aho and Ullman on relational algebra with a fixpoint operator appeared [AU79]. They also proved that the transitive closure cannot
be expressed in standard relational algebra. Although this paper itself did not
examine the relation to logical rules, it was certainly one of the origins of bottom-
up query evaluation.

It is difficult to say when naive bottom-up evaluation of Datalog was exactly
introduced. As we already mentioned, it dates back to 1965, and there have
been a lot of papers using some form of bottom-up evaluation. Naive bottom-up
query evaluation as we know it today (based on relational algebra with fixpoint
evaluation) seems to be first described in [CGL86].

- In 1982 the Fifth Generation Project started in Japan [Fur92]. It gave an im-
  portant impetus to the development of logic programming, not only in Japan,
  but throughout the world.
- The Prolog-dialect for deductive databases is nowadays usually called “Datalog”.
  It is not easy to find out who invented that name. In [CGT90] it is said:

  The term “Datalog” was invented by a group of researchers from Oregon
  Graduate Center, MCC, and Stanford University.

In [AHV95], the following is said:

  It is difficult to attribute datalog to particular researchers because it is a
  restriction or extension of many previously proposed languages; some of
  the early history is discussed in [MW88]. The name datalog was coined
  (to our knowledge) by David Maier.

I have found no papers before 1986 which use the name “Datalog”, and 1986 it
was used in [BMSU86], which refer to an unpublished memorandum of the Ore-
gon Graduate Center 1985 (probably a predecessor of [MW88]). Also [MUVG86]
contains the name “Datalog”.

- In or before 1986 the work on three major implementations of deductive database
technology was started: The LDL system at MCC [TZ86, Zan88, NT89], the
  NAIL! system at Stanford University [MUVG86], and the Smart Data System
  (SDS) at a commercial offsprings of the Technical University of Munich [KSSD94].
- In 1986, Bancilhon, Maier, Sagiv, and Ullman developed the “Magic
  Set”-method [BMSU86], and Rohmer, Lescoeur, and Kerisit developed the
closely related “Alexander”-method [RLK86]. These methods allow to combine
the advantages of top-down and bottom-up query evaluation (see below). At
that time many query evaluation algorithms for recursive rules had been pro-
posed [BR86], and this was probably the true starting point where the field got
its own methods and results. Today, deductive database systems usually use
the Magic Set/Alexander technique, or variants of SLD-resolution with memo-
ing, of which [TS86, Vie87a, Die87] were early developments. A similar method,
which also integrates top-down and bottom-up evaluation, is the query-subquery
approach [Vie87b, Nej87].
- In 1986, Minker organized another conference on deductive databases [Min88a],
  which was important for the development of the notion of stratified databases
and their perfect model [VG86, ABW88, Prz88a, Naq89]. The treatment of
negation according to the perfect model, which became standard in deductive databases, is again a difference to logic programming, where CLARK’s completion and similar approaches were dominant.

- From 1988 to 1990 a first generation of textbooks on deductive databases appeared. They were written by ULLMAN [Ull88, Ull89b], NAQVI and TSUR [NT89] and CERI, GOTTLOB, and TANCA [CGT90]. Furthermore, in the 5th edition of his best-selling textbook on databases, DATE added a chapter on deductive databases [Dat90].

With the event of a number of dedicated textbooks, it can be really said that the field is well established. Of course, many important things have been achieved since 1989. However it is still a little difficult to put them into historical perspective, since the distance is missing.

Why Deductive Databases?

A typical database application (see Figure 1.2) consists of three parts, which are usually coded in three different languages:

- At the top is the user interface, which manages the dialog between the user and the program. It defines menus, buttons, dialog boxes, and so on. Usually, it is either written in a special language like Tcl/Tk or HTML, or is constructed with a special editor (“resource construction set”, “interface builder”).

- Below that, there is a layer of code which performs data format changes, combines and aggregates data, and ensures that the integrity of the database is not violated. This code is often written in an imperative language like “C”.

- Finally, at the bottom are the database accesses, written in SQL. The database has built-in algorithms for searching and sorting, it ensures the persistence of the data and manages concurrent accesses by different users.

Besides the problems for the user to know three different languages and to interact with three different compilers, it is well-known that between each two layers there is an “impedance mismatch problem”. For instance, the database returns a set of answers to a query, but in the programming language it is necessary to write a loop which fetches every single tuple.

Deductive databases solve this problem because they are both, a programming language and a database, and they are usually so tightly coupled that it is not clear where one thing stops and the other starts.

This wish to integrate a database with programming facilities also led to the development of object-oriented databases. However, in deductive databases the set-oriented paradigm was extended to the programming part, while in object-oriented databases the tuple-oriented approach of the programming-language has “won”.

Even commercial relational databases nowadays allow to store procedures within the database. This is a logical development, because one important goal of databases is the integration of different applications within some company. Before there were databases, every application program more or less had its own data files. Then databases allowed to share the data and avoid redundancy (and thus errors), but still
the application programs were separate. There might have been some library procedures, just as some files were shared between some applications before databases were developed. But it was necessary to explicitly relink the application programs when a library procedure was changed. There was no controlled way of sharing code. This is what deductive databases as well as object-oriented databases and procedural extensions of relational databases aim at.

An additional advantage is that all the functionality of the database, such as persistency, data dictionary, access rights, etc., apply to the procedures as well, because they are stored within the database. Furthermore, the query optimization mechanisms of the database can now make some use of their knowledge of the procedural code. Before, no “global” query optimization was possible, because the database could not know at all how the sequence of queries executed by one program would look like.

Since deductive databases are a symbiosis of Prolog and relational databases, it is natural to ask what we gain with respect to each of the partners. If we compare a deductive database with a relational one, we have also the following advantages besides the above mentioned better integration with procedures:

- The queries which can formulated in SQL are restricted. For instance the transitive closure cannot be computed by a single query in standard SQL. Of course,
since this problem is practically relevant and often cited, some commercial versions of SQL now have a special clause for computing the transitive closure. But this is only a patch on the design of the SQL language: It complicates the language and solves one specific symptom, but not the deeper cause, namely that recursion is not available in SQL. This also means that recursive data structures such as trees or already lists cannot be manipulated adequately in SQL.

It might be argued that in practice recursion is always bounded, and for instance paths of length 3 or 4 can theoretically be computed in SQL. However, such queries are ugly to write in SQL and their size grows so fast (quadratically) that one soon reaches the limitations of the system. In combination with an application program in “C” or some other computationally complete language, every kind of query can be evaluated, but then we again have the above-mentioned “impedance mismatch” problems.

• The notion of a view is not fully integrated in the relational model. Quite a number of systems restrict the queries which can be posed to views. One of the reasons for this is that views are often implemented by rewriting the query (so that it refers only to base relations), but SQL is not orthogonal, so the result is not necessarily a valid SQL query. However, in deductive databases, views are “first class citizens”: The notion of a derived predicate is so essential that any restriction in their definition and usage would be totally unacceptable. Furthermore, corresponding to their greater importance in deductive databases, the implementation of views is usually more efficient there.

• The relational model is unsuitable for highly regular data. For instance a bus schedule can be better represented by rules than by facts. In a relational database such a bus schedule would probably be stored “extensionally” by enumerating the departure times. Maybe somebody even writes a “C” program to generate all these facts. But again the database has no knowledge of this program, while it could make good usage of the rules: For instance, the rules occupy much less memory than the facts, and thus can be more quickly loaded from external storage.

In comparison with Prolog, we should mention the following differences:

• Deductive databases are “more logical” and have less control, so they are closer to a theoretically ideal logic programming language. For instance, the order of rules or body literals does not matter in a deductive database, while it is essential in Prolog. The deeper reason for this is that deductive databases and Prolog programs are used very differently: A Prolog program usually is executed by calling one main predicate, so the programmer knows in advance which arguments of a predicate are bound or free, and therefore he/she can order the rules and body literals in an optimal way. In deductive databases this is not possible, because the user might query any predicate with any binding pattern for the arguments (more or less). It is therefore the responsibility of the system to determine a good execution order. This matches the tradition of the database community that a database should have a powerful query optimizer. Of course, such optimizers work well only for sets of relatively simple rules — they cannot
optimize all Prolog programs.

- Due to the bottom-up evaluation, the termination behaviour is much better than that of Prolog. In general, we would expect that termination can be guaranteed for query evaluation, while this certainly is not possible for arbitrary programs. Modern deductive database systems allow more or less every Prolog program, so termination cannot be guaranteed in general. But termination is a major issue in databases, and large subsets of queries/programs have been defined for which termination can be guaranteed.

- Prolog has insufficient support for external memory. Naturally, Prolog was not intended to work with large sets of facts, and many Prolog implementations will become at least very inefficient if the main memory is too small. An important property of external memory is its block-orientation: In order to get a single tuple of a few bytes we must read a complete block of several kilobytes. Therefore, set-oriented computations as done in deductive databases can make much better use of external storage than the standard tuple-oriented evaluation of Prolog. Of course, also other essential database features, like support for multiple users and recovery after a system crash, are missing in Prolog.

Let us conclude this section by looking at a number of typical applications of deductive databases [KG90, Tsu90, Tsu91a, Ram95]:

- **Expert systems:** Usually the expert knowledge is formalized by means of rules (not necessarily logic programming rules). Furthermore, there are often large sets of facts needed (even an expert has to look into a book from time to time). Thus, a deductive database seems to be a good tool for developing an expert system. Of course, an expert system shell usually has better support for creating a user interface and defining a structured user dialog, and better explanation facilities. But all this would be helpful in a deductive database as well.

  In [HR95], an expert system for querying a flights database is described. Since a user has several conflicting criteria for the best flight (cheap, not too early in the morning, not too much wait time on transit, not an obscure airline, etc.), this is a quite complex task. In [KSSD94] an expert system for the public transportation system of the Munich area is mentioned.

- **Decision Support Systems:** The task of these systems is to aggregate information form large data sets, or to find interesting cases in them. Often, the temporal development of the data should be displayed. Also, the system should be flexible and allow ad-hoc queries. Furthermore, it should have the ability to reason about future plans [RH94]. All this can be well supported by deductive databases.

  In [KSSD94] the following is said on the marketing strategy of the deductive database system SDS:

  One main selling point of this technology is its strategic decision making capability. Database technology, enhanced by deductive rule-based capabilities, can assist enormously in condensing information to make
good decisions (a major key in achieving a competitive advantage). In many corporate decisions, relevant information is spread over heterogeneous databases, and such an environment has to be addressed.

In [RRS95] a system for stock market analysis is described, which is based on the deductive database system CORAL [RSSS94]. The main advantage of using a deductive database system is the easy extensibility. It is also noted that

The recursive query capabilities of CORAL are necessary for expressing many natural concepts (e.g. “bull market”, “consecutive peaks”) in this domain.

Finally, we would like to mention the following applications, which lie on the border to expert systems [RH94]:

Medical analysis and monitoring can generate large amounts of data, and an error can have disastrous consequences. A tool to carefully monitor a patient’s condition or to retrieve relevant cases during diagnosis reduces the risk of error in such circumstances. Deductive database technology allows the analysis of these data to be performed more efficiently and with lower chance of error than by ad hoc methods. Such an intelligent tool allows the human experts to concentrate on the main problems, rather than being distracted by details. A similar example may be found in mineral exploration; a large amount of data may be generated, which can then be analyzed for clues suggesting the presence of the desired mineral.

- **Hierarchical Design**: Computer aided design of hierarchically structured objects is a good application area for deductive databases, because of their special support for hierarchies (through the computation of transitive closures). Of course, currently the performance of deductive databases is a problem, and therefore object-oriented databases are preferred. However, it is a nontrivial task to write a “C++”-program which checks whether a given object directly or indirectly needs some specific part, while this is possible in three short lines of Datalog.

  The “bill of materials”-problem has been used as an example in [CGT90, KG90]; in [BK92] economists have discussed the problem and the usefulness of Datalog.

- **Complex Integrity Constraints**: In many design tasks complex integrity constraints have to be enforced. Since these integrity constraints are typically defined in logic, it is natural to use a deductive database to check them. Of course, there are again special additions to standard databases which allow the incremental checking of integrity constraints and thus make the process more efficient. But all this fits well into the framework of deductive databases while in other systems one can write only procedures or triggers which perform some checks and it is a difficult task to verify that these procedures really enforce a given set of constraints.
For instance, in [FSS+92] an application is mentioned, where the deductive database system LOLA is used for checking integrity constraints in the parts database of a car manufacturer. The deductive database systems EKS-V1 [VBK+92] and VALIDITY [FGVLV95] have special support for integrity checking.

- **Graph-Structured Data:** In general, applications of graphs (as known from combinatorics), are also good applications for deductive databases, because they often need to find paths. Some developers of deductive databases have tried to build good algorithms known from graph theory into their system (selectable with specific control statements for the optimiser). However, the graphs stored in deductive databases can be so big that they do not fit completely into main memory, so that “locality of access” is an issue here, which is usually not considered in graph theory.

- **Integration of Heterogeneous Databases:** Here the powerful view concept of deductive databases is very helpful. Currently many companies still have multiple databases and are not able to completely integrate them into a single system, but want at least a common view on all their data. As mentioned above, the deductive database system SDS [KSSD94] was specifically designed to work in such a heterogeneous database environment.

- **Parsing:** There is a strong relation between logical rules and grammar rules — in fact, Prolog was mainly invented for natural language analysis. But if we want to analyse natural language, then there are obviously large amounts of data (e.g. a lexicon), so we need database support. Definite clause grammars (DCGs) are standard material in any Prolog textbook. However, Prolog implementations of DCGs do not allow left-recursive rules, and backtracking involves the duplicate construction of syntactical structures. This is improved by using bottom-up evaluation, as done in deductive databases.

In [FSS+92, SF95] a system for the morpho-syntactical analysis of Old Hebrew texts is described, which was implemented in the deductive database system LOLA and was used to analyse the grammar of the complete old testament (an obviously quite large set of data). Also the flexibility and support for incremental design of deductive databases was helpful in this application, since the grammar had to be refined via experiments with the data.

In summary, there is a large potential of applications which could profit from deductive databases. And, vice versa, these applications might suggest some important extensions of deductive databases.

**Top-Down vs. Bottom-Up Evaluation**

We will now give a first impression how a deductive database might work. The main task of a database system is of course to answer queries. Quite different query evaluation algorithms have been developed, all with their own features and problems.

Probably the first “deductive databases” were Prolog-interpreters or later compilers. Of course, they had no specific database support, one had to load all tuples (facts)
into main memory at the beginning of the Prolog session. This severely restricts the
size of the “database”, but otherwise a Prolog system can be seen as an implementa-
tion of Datalog. Prolog uses a theorem-proving method called SLD-resolution: Given
a query $A \land \cdots$, it searches for matching rules of the form $A \leftarrow B_1 \land \cdots \land B_n$, and
then recursively tries to prove $B_1 \land \cdots \land B_n \land \cdots$ as a subquery. Of course, if $A$ is
given as a fact, it can be simply deleted. Furthermore, variables can be instantiated
by means of “unification”. So the rules are used from head to body, and the process
is goal-directed: Only rules are touched which are useful for proving the given query.
This form of query evaluation is called “top-down”, because one usually thinks of the
query being at the top, and the given facts at the bottom. The advantage of top-down
evaluation is that if it works well, then it is really fast. However, it might fail badly.
First, it might not terminate, for instance in case of a rule like

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

Of course, nobody would write such a strange rule (except perhaps a logician), but
the following program for transitive closure is really standard:

$$\text{path}(X, Y) \leftarrow \text{edge}(X, Y).$$
$$\text{path}(X, Y) \leftarrow \text{edge}(X, Z) \land \text{path}(Z, Y).$$

It assumes that a directed graph is stored in the database relation $\text{edge}$, and computes
pairs of nodes (vertices), which are connected by a path. Now this program runs into
an infinite loop if the graph contains cycles. Of course, if the programmer knows this
before, he/she can write a more complicated Prolog-program which detects cycles.
But this is extra work and makes the rules less understandable. Furthermore, even
if there are no cycles, and Prolog terminates, it might use exponential running time,
for instance in case of the following graph:

There are exponentially many paths in this graph, and Prolog follows them all. But
the number of connected node pairs is quadratic.

This is the advantage of the bottom-up query evaluation: It guarantees termina-
tion and a polynomial behaviour for any pure Datalog program (like the transitive
closure above). Bottom-up evaluation works by applying the rules to the given facts,
thereby deriving new facts, and repeating this process with the new facts until no
more facts are derivable. The query is considered only at the end, when the facts
matching the query are selected. Of course, it is very inefficient to compute a large
number of facts which are irrelevant for the given query. In contrast, top-down eval-
uation performs only “relevant work”, but might perform the same work again and
again — maybe infinitely often. So the result is that in many practical examples, top-down query evaluation runs much faster than “naive” bottom-up evaluation, but bottom-up evaluation is safer because of its guaranteed polynomial behaviour.

Naturally, people have tried to combine bottom-up and top-down query evaluation in order to have both advantages: being goal-directed and avoiding duplicate work. The standard method, used in many deductive database systems, is called the “magic set” transformation [BMSU86, RLK86, Bry90b]. It introduces new predicates which encode the queries occurring during top-down evaluation, and rewrites the rules in such a way that they are only applicable if the head literal is needed in order to answer the query (a formal definition is given in Chapter 3).

For instance, let us consider the following logic program which computes the Fibonacci numbers:

\[
\begin{align*}
\text{fib}(0, 0). \\
\text{fib}(1, 1). \\
\text{fib}(N, F) & \leftarrow N > 1 \land \\
& \quad N_1 = N - 1 \land \text{fib}(N_1, F_1) \land \\
& \quad N_2 = N - 2 \land \text{fib}(N_2, F_2) \land \\
& \quad F = F_1 + F_2.
\end{align*}
\]

Here, \( \text{fib}(N, F) \) means that \( F \) is the \( N \)-th Fibonacci number. Let us assume that the query is for instance \( \text{fib}(10, X) \) (“What is the tenth Fibonacci number?”).

Because this program contains arithmetic predicates, it is not a pure Datalog program. And in this case bottom-up evaluation does not work: It computes more and more Fibonacci numbers and does not terminate because the facts matching the query are selected only at the end. Top-down evaluation terminates, but it is also very inefficient: For instance to compute the tenth Fibonacci number, it evaluates 177 calls to \( \text{fib} \), because it does not remember which numbers it has computed already. In general, the time top-down evaluation needs to compute the \( N \)-th Fibonacci number grows exponentially, while there is a simple linear time algorithm.

Now the magic set transformation introduces a predicate \( m_{\text{fib}^\text{bf}} \), which contains the arguments, for which the Fibonacci function has to be evaluated. Then the rules are only applicable if the resulting Fibonacci-number is really needed:

\[
\begin{align*}
\text{fib}(0, 0) & \leftarrow m_{\text{fib}^\text{bf}}(0). \\
\text{fib}(1, 1) & \leftarrow m_{\text{fib}^\text{bf}}(1). \\
\text{fib}(N, F) & \leftarrow m_{\text{fib}^\text{bf}}(N) \land N > 1 \land \\
& \quad N_1 = N - 1 \land \text{fib}(N_1, F_1) \land \\
& \quad N_2 = N - 2 \land \text{fib}(N_2, F_2) \land \\
& \quad F = F_1 + F_2.
\end{align*}
\]

Of course, the predicate \( m_{\text{fib}^\text{bf}}(0) \) has to be defined in such a way that it contains those arguments to the Fibonacci function which occur during the computation:

\[
\begin{align*}
m_{\text{fib}^\text{bf}}(10). \\
m_{\text{fib}^\text{bf}}(N_1) & \leftarrow m_{\text{fib}^\text{bf}}(N) \land N > 1 \land N_1 = N - 1. \\
m_{\text{fib}^\text{bf}}(N_2) & \leftarrow m_{\text{fib}^\text{bf}}(N) \land N > 1 \land N_2 = N - 2.
\end{align*}
\]
If this “transformed” program is evaluated bottom-up, it computes only Fibonacci numbers needed for the query, and it computes every Fibonacci number only once. The goal-direction is inherited from top-down evaluation, while the memoing of already computed facts is from bottom-up evaluation. By the way, it is a standard technique in functional programming to create a list of previous calls to a function and to search in this list before really executing the function.

It is by now folklore that bottom-up evaluation after the “magic set” transformation is at least “as efficient as” top-down evaluation. This is for instance what the title “Bottom-Up beats Top-Down for Datalog” of ULLMAN’s paper [Ull89a] suggests. The result was very important because top-down evaluation was well-known in logic programming, and had been successfully used in many practical applications. With this result it seemed that deductive databases could be equally successful — or even more, because of the additional functionality. Of course, the implementation techniques of logic programming were better developed, and databases might introduce some overhead, but the result caused much optimism. It seemed achievable that good implementations of deductive databases will eventually beat Prolog systems.

However, ULLMAN in fact has not used the top-down algorithm of Prolog as a measure for comparison, instead he used an algorithm called QRGT-resolution (developed by himself). And in fact it is easy to see that the same result does not hold for Prolog’s SLD-resolution. It is already wrong for the standard transitive closure program (see above), applied to a graph which is a simple straight line:

\[ \bullet \rightarrow \bullet \rightarrow \cdots \rightarrow \bullet \rightarrow \bullet \]

Here, the running time of Prolog’s SLD-resolution is \( O(n \log(n)) \), while magic sets need more than \( O(n^2) \). To be fair, already ULLMAN noted in a footnote of [Ull89a]:

“However, Prolog implementations usually use a form of tail recursion optimization that, for certain examples, such as the right-linear version of transitive closure, will avoid rippling answer tuples up the rule/goal tree, and thus can be faster than QRGT.”

Although it is true that Prolog implementations have a tail recursion optimization, the main goal of this optimization is to save memory; the improvement of the running time is only a side effect. And in fact, the difference in performance can already be understood on the abstract level of SLD-trees, we do not have to look at the internal data structures of a Prolog system. We will investigate this example more formally in Chapter 3, and show that the problem indeed occurs only in tail-recursive programs, and that the only reason is the “materialization of lemmas” during bottom-up evaluation of the magically rewritten program. These questions have been investigated several times in the literature, but we believe that our formalizations and proofs (given in Chapter 3) are especially useful, simple, and clear. Furthermore, we will show that by using Bry’s idea of deriving magic sets from a metainterpreter [Bry90b], we can get (at least for tail-recursive programs) a rewriting technique which directly mimicks SLD-resolution.
The Need for Nonmonotonic Negation

Above, we said that a deductive database system can also be seen as an automated theorem prover for some simple subset of logic. However, there is one important difference: Negation is usually not treated as the negation of classical logic, but as some form of “negation as failure to prove”. In this way deductive databases violate the principle of monotonicity, which is fundamental to classical logic: If we add further facts to the database, it might happen that previous answers become invalid. Such a behaviour is impossible in classical logic.

Naturally, if one changes the logic in such a way, one should have good reasons for this. In order to see the need for nonmonotonic negation, let us first consider a simple example. Suppose that we want to create a small database with information about the computers of our institute. In the tabular representation usually used in relational databases, this would look as follows:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Bench</th>
<th>Date</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>wega</td>
<td>SPARCserver 330</td>
<td>89</td>
<td>11/89</td>
<td>84 425</td>
</tr>
<tr>
<td>pollux</td>
<td>SPARCstation 1</td>
<td>113</td>
<td>11/89</td>
<td>28 860</td>
</tr>
<tr>
<td>sirius</td>
<td>SPARCstation ELC</td>
<td>72</td>
<td>11/91</td>
<td>9 967</td>
</tr>
<tr>
<td>regulus</td>
<td>SPARCstation 10 Mod. 30</td>
<td>33</td>
<td>12/92</td>
<td>26 443</td>
</tr>
<tr>
<td>spica</td>
<td>SPARCstation 10 Mod. 20</td>
<td>42</td>
<td>12/92</td>
<td>18 126</td>
</tr>
<tr>
<td>polaris</td>
<td>SPARCstation 10 Mod. 20</td>
<td>36</td>
<td>12/92</td>
<td>18 126</td>
</tr>
<tr>
<td>krypton</td>
<td>IBM RS/6000 Mod. 220</td>
<td>110</td>
<td>12/92</td>
<td>15 972</td>
</tr>
<tr>
<td>prokyon</td>
<td>SPARCclassic</td>
<td>88</td>
<td>8/94</td>
<td>8 973</td>
</tr>
<tr>
<td>capella</td>
<td>Linux 486/66</td>
<td>42</td>
<td>11/94</td>
<td>4 500</td>
</tr>
<tr>
<td>deneb</td>
<td>SPARCstation 5 Mod. 110</td>
<td>32</td>
<td>10/95</td>
<td>17 478</td>
</tr>
<tr>
<td>antares</td>
<td>Linux Pentium/100</td>
<td>16</td>
<td>10/95</td>
<td>6 900</td>
</tr>
</tbody>
</table>

(The value in column Bench shows the number of seconds needed to format (with \texttt{LATEX}) this thesis, so lower values mean higher performance.)

Usually, such a table is represented in logic as a set of facts:

\[
\text{computer}(\text{wega}, \, '\text{SPARCserver 330}', \, 89, \, 11/89, \, 84\,425).
\]

\[
\ldots
\]

\[
\text{computer}(\text{antares}, \, '\text{Linux Pentium/100}', \, 16, \, 10/95, \, 6\,900).
\]

Now if we ask the query, “Does the institute have a computer called \textit{atair}?”,

\[
\text{computer}(\text{atair}, \, \ldots).
\]

we expect of course the answer “no”. But this answer is not logically correct: Given only the above set of facts, the system must logically answer “I don’t know”. The reason is that we have specified explicitly only what is true about the relation \textit{computer}, and not what is false.

The nonmonotonic logic used in deductive databases automatically assumes that everything else is false, so we get the intended answer. However, it might seem at first
that changing the logic is much too drastic. So, how can we specify this in standard first order logic? Obviously, it is impossible to explicitly enumerate all false facts like

\[ \neg \text{computer}(\text{atair}, '\text{SPARCstation IPC}', 74, 8/91, 12000). \]

(By the way, even if there were only finitely many constants, we would need a “domain closure axiom” in addition in order to conclude that the above answer is “no”.) A much better solution seems to be an explicit definition of the predicate “computer” by means of a completion-formula [Cla78] like

\[
\forall N, T, B, D, P : \\
\text{computer}(N, T, B, D, P) \\
\leftrightarrow (N = \text{wega} \land T = '\text{SPARCserver 330}' \land B = 89 \land D = 11/89 \land P = 84\,425) \\
\lor \ldots \\
\lor (N = \text{antares} \land T = '\text{Linux Pentium/100}' \land B = 16 \land D = 10/95 \land P = 6\,900).
\]

However, even with this prerequisite, the answer “no” is not justified, because it is not clear to a purely logical reasoner that atair and for instance wega are really two different objects — it might be only different names for the same object, such as “the butler” and “the murderer” in some thrillers. What we need are the “unique name axioms”, such as “atair $\neq$ wega”. Of course, if we have infinitely many constants, these axioms cannot be written down explicitly. And even if there are only finitely many constants, the set of UNA-axioms grows quadratically, so it is very impractical to work with them.

Of course, it might be possible to construct a theorem proving algorithm which uses the unique name axioms implicitly. However, this is in fact already a change of the logic. But it is not such a big change as we propose, because the logic remains monotonic. This solution would suffice for relational databases, but it does not suffice for recursive rules. For instance, the transitive closure has no explicit definition like the one given for computer above. In (nonmonotonic) Datalog, pairs of connected nodes in a graph can be defined as follows:

\[
\text{path}(X, Y) \leftarrow \text{edge}(X, Y).
\]
\[
\text{path}(X, Y) \leftarrow \text{edge}(X, Z) \land \text{path}(Z, Y).
\]

Again, the intention is that path contains only those tuples which are derivable by these rules. It might seem at first that the following axiom would constrain path in the correct way:

\[
\forall X, Y : \text{path}(X, Y) \leftrightarrow \text{edge}(X, Y) \\
\lor \exists Z : (\text{edge}(X, Z) \land \text{path}(Z, Y)).
\]

However, this does not work for cyclic relations edge, i.e. it does not enforce that path is only the transitive closure. For instance, if edge consists of the single tuple \((a, a)\), also the following relation would be a model of the above formula:

\[
\text{path} := \{(a, a), (a, b)\}.
\]
In fact, it can be proven that no set of first order formulas works if edge can be any finite relation, because the transitive closure is not first order definable (this is also the deeper reason why it is not expressible in SQL).

At this point, we could still solve our problems by simply translating the “I don’t know”-answer of the first order theorem prover into “no”. However, this works only as long as we do not use negation explicitly. For instance, suppose that the computers pollux and capella got faulty. We represent this in another relation:

<table>
<thead>
<tr>
<th>faulty</th>
</tr>
</thead>
<tbody>
<tr>
<td>pollux</td>
</tr>
<tr>
<td>capella</td>
</tr>
</tbody>
</table>

It is now natural to ask, “Which computers are currently available?”.. Of course, we could first query all computers, write the result down, and then query all faulty computers and take the set-difference. But a fundamental principle of query language design is that such simple combinations of queries should again be a valid query. For instance, in Datalog we get the available computers by means of the following query:

$$\text{computer}(X, \ldots) \land \text{not faulty}(X).$$

Note that here “not” does not denote the negation of first order logic, but “it is not provable that”. Of course, it is also possible to define a derived relation (a “view” in database terms), which contains the available computers:

$$\text{available}(X) \leftarrow \text{computer}(X, \ldots) \land \text{not faulty}(X).$$

This is not as simple as it seems at first, because now the theorem prover has to reason about something not being provable while it is performing a proof. For instance, a typical paradox is

$$p \leftarrow \text{not } p.$$

Cases like this have lead to quite a number of semantics for nonmonotonic negation, we will consider this in greater detail in Chapter 4.

Returning from this technical discussion, it is also important to note that human beings reason nonmonotonically in such a way. For instance, it is much simpler for us to write down the positive facts than to create correct $\leftrightarrow$-definitions (if at all possible). Since deductive databases are intended to be used not only by logicians, it is important to find a formalism which is similar to the way people think. For instance, it happens quite often that we say “if”, when we really mean “if and only if”. We like to concentrate on the important things and leave the rest to be understood without saying.

The field of artificial intelligence is investigating how to formalize “common sense”. For instance, JOHN McCARTHY has considered in [McC80] the well-known puzzle of missionaries and cannibals:

Three missionaries and three cannibals come to a river. A rowboat that seats two is available. If the cannibals ever outnumber the missionaries on either bank of the river, the missionaries will be eaten. How shall they cross the river?
Of course, in order to find a solution algorithmically, one usually describes the problem space as a set of states with the allowed transitions between them. But, as McCarthy argues, it is really a big step from the above problem description in natural language to the proper formalization:

The second reason why we can’t deduce the propriety of Amarel’s representation is deeper. Imagine giving someone the problem, and after he puzzles for a while, he suggests going upstream half a mile and crossing on a bridge. “What bridge,” you say. “No bridge is mentioned in the statement of the problem.” And this dunce replies, “Well, they don’t say there isn’t a bridge.” You look at the English and even at the translation of the English into first order logic, and you must admit that “they don’t say” there is no bridge. So you modify the problem to exclude bridges and pose it again, and the dunce proposes a helicopter, and after you exclude that, he proposes a winged horse or that the others hang onto the outside of the boat while two row.

You now see that while a dunce, he is an inventive dunce. Despairing of getting him to accept the problem in the puzzler’s spirit, you tell him the solution. To your further annoyance, he attacks your solution on the grounds that the boat might have a leak or lack oars. After you rectify that omission from the statement of the problem, he suggests that a sea monster may swim up the river and may swallow the boat. Again you are frustrated, and you look for a mode of reasoning that will settle this hash once and for all.

We see that the problem is again one of implicit negation: First, there are no other means of transportation besides the one mentioned in the puzzle, and second, there are no other problems, besides that the cannibals might eat the missionars.

Applications of Disjunctive Information

The above kind of deductive databases with some form of nonmonotonic negation is more or less standard now (although the allowable uses of negation are quite restricted, we will investigate this further in Chapter 4).

In this thesis, we go one step further and also allow to represent disjunctive information in the database.

Usually, a deductive database has only one intended model corresponding to the completely known state of the real world. However, there are many applications where we do not know exactly which of some possible states is the correct one. Examples are:

- Null values: For instance, an age “around 30” can be 28, 29, 30, 31, or 32.
- Legal rules: The judge always has some freedom for his decision, otherwise he/she would not be needed. So laws cannot have a unique model.
- Diagnosis: Only at the end of a fault diagnosis we know exactly which part of some machine was faulty. But as long as we are searching, there are different possibilities.
• Biological inheritance: E.g. if the parents have blood groups A and 0, the child must also have one of these two blood groups (example from [Lip79]).
• Natural language understanding: There are many possibilities for ambiguity here, and this is represented most natural in multiple intended models.
• Conflicts in multiple inheritance: If we want to keep as much information as possible, we would assume the disjunction of the inherited values [BL93].
• Reasoning about conflicts in concurrent updates: If we do not know in which sequence two processes are executed, we can assume only the disjunction of the two values they assign to some variable.

It is often possible to code these situations somehow in Horn clauses, but this is difficult, indirect, and therefore rather error-prone. At least in the first phases of database design we should use the most powerful tools available to be as near to the real world as possible. Later, it might be useful to apply some transformation to the database in order to increase efficiency. Naturally, it would be very useful if the same deductive database system could be used in every step of the transformation and we would have the possibility to stop whenever we have reached a sufficient degree of efficiency. Our goal is a system which allows gradual transitions between the disjunctive and the standard Horn case. For instance, if the database is “mostly Horn”, i.e. there are only a few disjunctive rules which are not used too heavily, then the efficiency should also be nearly that of a standard deductive database.

However, it must be said at this point that disjunctive databases are really more powerful than standard deductive databases: A translation from a set of disjunctive rules into Horn clauses is not always possible, at least not without increasing the number of objects, for instance by introducing lists. This follows from complexity-theoretic considerations: For instance, it is known that a satisfiability-test for a set of propositional clauses is NP-complete. Such a satisfiability-test can easily be done by a disjunctive deductive database. Clauses with at least one positive atom, such as \( p_1 \lor p_2 \lor \neg p_3 \lor \neg p_4 \), are written down in the form

\[
p_1 \lor p_2 \leftarrow p_3 \land p_4,
\]

while clauses without a positive atom, such as \( \neg p_1 \lor \neg p_2 \), are represented as

\[
false \leftarrow p_1 \land p_2.
\]

Now the set of clauses is unsatisfiable iff \( false \) follows from this database. So it is possible to specify this NP-complete problem in a disjunctive deductive database, while bottom-up evaluation of a standard Horn database is always polynomial.

So disjunctions really increase the expressive power of a deductive database, and make it more similar to a general automated theorem prover. However, an important difference is that we require the clauses to be “range-restricted”, i.e. every variable must appear in a positive body literal. So a deductive database can reason only about the objects explicitly known to it, usually not about all integers and so on. Such tasks, which are required for instance in program verification, need an automated theorem prover, not a deductive database.
If we wanted to relax this restriction, as even some standard deductive databases do, we would lose a lot of efficiency, because database techniques are no longer applicable. So we would have to use also the implementation techniques for theorem provers, and this is a different area, which is not treated here. Maybe, we can say that a database in contrast to a theorem prover is always able to give specific objects as answers to queries — not only the information that there is such an object, or that all objects have this property.

As can be seen from the above discussion, we have now left the realm of standard deductive databases, and it is quite a big step to disjunctive databases. In fact, some people believe that disjunctive databases will never reach an acceptable degree of efficiency, such that they can be used in practice. They say that if already standard deductive databases have efficiency problems, and are currently seldom used in real applications, why look at something more general? Well, the answer is that disjunctive databases can really solve problems for which standard deductive databases are not applicable. Naturally, disjunctive databases will always be used only in more specialized application domains (maybe niches), and it will need more time until there are powerful implementations. However, for us it is also interesting to see how techniques developed for the Horn case are really applicable in a more generalized setting. Furthermore, automated theorem provers, which are even more general, have been successfully used for solving practical problems. So there is hope that disjunctive logic programming will also become useful for applications which do not need the full power of automated theorem proving, but have a fact base of medium size, which cannot be handled by standard theorem provers.

For Horn clauses, it is by now generally accepted that top-down and bottom-up query evaluation techniques both have advantages of their own and that none is superior to the other for all applications. Furthermore, the cross-fertilization of both approaches was very successful [Bry90b]. But for disjunctive rules, up to now top-down approaches were dominant [LMR92]. Although it is known that the bottom-up immediate consequence operator $T_P$ can be directly generalized to disjunctive rules [MR90], this is usually considered only as a means to define the semantics, not as something amenable to implementation.

One important reason for this is that $T_P$ as defined in [MR90] allows very often the derivation of exponentially many disjunctive facts (although this is not made explicit in that paper, see Chapter 5 for an example). Now our contribution is an optimization of $T_P$ which makes the resolvable literal in a disjunctive fact unique. In many cases, this reduces an exponential behaviour to a polynomial one. We thereby improve an optimization which is already known for positive hyperresolution [CL73] (the theorem-proving counterpart of the disjunctive $T_P$) for quite a long time.

By applying these ideas, disjunctive rules can be naturally translated into Horn clauses with lists. This allows a direct implementation on standard deductive database systems. Of course, more specialized data structures would be very useful for a really efficient implementation. However, by the inverse translation, it is also possible to generalize standard implementation techniques developed for Horn clauses to the disjunctive case. This is our main point.
For instance, one of the very basic things a standard deductive database does is that it applies the rules in the order of the predicate dependencies and iterates only (mutually) recursive rules. However, prototype implementations for disjunctive databases usually iterate all rules until nothing changes. The reason for this is that a seemingly innocent rule like \( p(X) \leftarrow q(X) \) needs to be applied two times if there is e.g. the disjunctive fact \( q(a) \lor q(b) \). The notion of disjunction types developed in Section 5.1 does allow to determine an evaluation order for disjunctive rules.

In fact, up to now such an analysis was nearly impossible, because there are so many ways to derive a disjunctive fact. With our optimization, the possibilities to resolve with a disjunctive fact are drastically reduced. In the case of positive disjunctive databases, where one is only interested in disjunctions of answer-literals, the resolvable literal within a disjunction can be made unique. However, even in the general case (where we need more information in order to evaluate negative body literals) our technique reduces the number of different derivations of the same disjunctive fact.

By now, there is quite a lot of research on disjunctive information, e.g. [Lip79, RT88, MB88, Dem91, LMR92, SA93, EGM94]. However, there are still very few actual implementations (e.g. [SA93]), and even less using database techniques. We believe that disjunctive databases will only get away from the (early?) prototype state if we can make the best possible use of what is already known for standard deductive databases. This motivated the work presented in Chapter 5.

**The Crisis of Deductive Databases**

It seems that at the moment quite a number of deductive database researchers are a little frustrated: There is a nice theory, and there are some larger prototype implementations (see Figure 1.3 and [Ram94, RU95]), but deductive databases have not yet made their way into industry: No software company is currently developing (let alone selling) a deductive database system, and the available prototype systems are also used nearly exclusively in the universities themselves.

In fact, there has been a commercial deductive database system, called SDS [KSSD94], but it did not sell well enough. The development began 1986 and ended 1990. The system was quite ahead of the time, for instance by being especially designed for the integration of heterogeneous systems. At the beginning, it ran only in a Lisp environment, which might have deterred possible customers. Also, some companies had just moved to a relational database, and were not ready for another change. Finally text books on deductive databases were missing at that time, and application programmers did not know Datalog.

One reason, why deductive databases did not (yet) get their way is also that currently object-oriented databases are more fashionable. There are a number of companies producing object-oriented database systems, and also quite a lot of industrial users. However, one of the achievements of relational databases was their declarative query language, namely to say only what is wanted, and not how to compute it. From this point of view, object-oriented databases (in their current development state) are a step backward: It is often necessary to program in C++ to get the desired result.
Figure 1.3: Some Implementations of Deductive Database Technology

The idea of declarative programming is summarized in Kowalski’s equation: ‘Algorithm = Logic + Control’. In standard imperative programming, the control part is explicit and the logic implicit, while in declarative programming, it is the other way round. The advantages of declarative programming are:

- **Enhanced Productivity**: It is not unusual that an equivalent formulation in Prolog or Datalog is ten times shorter than in C or C++. Since there is also reason to assume that the time an experienced programmer needs for one line of code is more or less independent of the language, this can lead to drastic savings in time and money.

- **More Powerful Optimization**: Since no fixed execution algorithm has influenced the language design, the space of possible optimizations is much bigger. On the other hand, optimization is not only possible, but also necessary, since a naive evaluation algorithm would be too inefficient.

- **Simpler Parallelization**: Imperative programming languages are often influenced by classical machine models, and a not very high abstraction of the machine language. Therefore, programs written in such languages cannot make full use of new machine architectures. Thus, programs written in declarative languages will probably live longer.

- **One Logic, Many Algorithms**: The dream of declarative programming is that the system automatically selects an optimal algorithm for the specified problem. Of course, this is impossible. However, it is possible to have an extensible optimizer: If the algorithm used for the general case should be too slow, the programmer can add annotations to guide the optimizer, or integrate new algorithms. In fact, the optimizer can have a whole library of possible algorithms at its disposal, similar to the different algorithms for joins and other operations in relational databases. Another simple example for “one logic, many algorithms” is the possibility to create or delete indexes in relational databases without hav-
CHAPTER 1. INTRODUCTION

ing to change application programs. This gives a good way to adapt to changing usage profiles.

- **Simpler Verification:** Since the semantics of the language is simpler and better formalized, also the verification of programs in this language is simpler. For instance, there exists a system for proving the termination of nontrivial Prolog programs, which is able to prove its own termination [Plü90]. I do not know of an equally powerful system for imperative programming languages.

Of course, object-oriented databases have also certain advantages, for instance an extensible type system, the clustering of whole objects on external memory, and a module concept. But all this can also be integrated in deductive databases, and there is currently a lot of research on deductive and object-oriented databases (DOOD systems). This is a very promising development for the future, and it seems that also the development of prototype systems is going this way.

In 1994, it became known that BULL was developing such a deductive object-oriented database system [RH94, FGVLV95]. However, in 1995 the whole group was closed down. This management decision is hardly understandable.

It seems to me that more research is needed before deductive database technology can really redeem its big promises. It is also necessary to look more at the needs of real applications, and to extend Datalog accordingly without sacrificing its cleanness and simplicity. Probably still more patience is needed than a company can have. Some scientists believe that it is no longer a problem of research, but only of having better implementations. For instance, on ICDE’93 there was a panel discussion “Are we Polishing a Round Ball?” [Sto93], and the topic of recursive query evaluation got especially bad marks for the potential of innovative research. Also RAMAKRISHNAN writes in the preface to [Ram95]:

These limitations can mostly be addressed by more careful implementations, and are perhaps understandable in research prototypes; it does not seem that fundamental new techniques are required to resolve them.

I believe that Chapter 3 of this thesis shows that there are essential problems not adequately treated in the literature. There have been quite a number of prototype implementations. If there was still no real breakthrough, this is not only a problem of implementation.

Let us also say some words on efficiency. This is often considered as very important by industrial users, whether it is really needed for the application at hand or not. It is a shame that the currently most efficient “deductive database system”, namely XSB [SSW94], uses mainly Prolog technology, and not the algorithms developed in the field of deductive databases. Of course, we must say that the comparison is a little unfair, since XSB (as well as many other prototypes) are main memory systems. And database technology is intended for the management of large amounts of data, which can only be kept on external storage. However, we should be able to make efficient use of available main memory, and it is certainly an interesting and important research problem to find out why Prolog technology is more efficient here and how we could improve the techniques of deductive databases to catch up with it. There are also
certain very typical operations in deductive databases, such as the computation of transitive closures, where deductive databases should be as efficient as a hand-crafted implementation in C. In the preface of [Ram95] it was noted that for an application in program analysis, an equivalent program in Datalog was much shorter than a C-program (“about a hundred lines vs. a few thousand lines”), but also 5-10 times slower when executed on CORAL. Although this already saves money, because the man power for the additional programming time is much more expensive than buying a ten times faster computer, we must try to make the efficiency loss smaller.

It is also an interesting question, whether the deductive database researchers and implementors believe themselves in deductive database technology. In Prolog implementations it is very common to have a large part implemented in Prolog itself (e.g., the whole compiler into WAM-code). In current deductive database systems, this does not seem to happen at all. It is not an acceptable excuse that a deductive database system is mainly a database system, and cannot be used for programming tasks. One of the main selling arguments for deductive database technology is that it is an integrated programming and database environment. For example, since deductive database systems are especially well-suited to process graph-structured data, this should also apply the the predicate dependency graph extracted from the rules. If we have problems to get applications from the outside, we might start by looking at the deductive databases themselves. In an invited talk at DOOD’93, RAINER MANTHEY gave an overview of the possibilities to use deductive databases in their own implementation [Man93].

Currently, quite a lot of manpower has to be invested into the development of nice user interfaces. Object-oriented languages have special support for this since a long time, and it would certainly increase the acceptance of deductive databases if there would be an easy and powerful way to build user interfaces in Datalog. It seems that in order to process events triggered by the user, an integration with the concepts of active databases would be useful. This means that rules can also work as production rules, not only as deduction rules.

In this thesis, we will clarify and improve the foundations of deductive databases, since only well-understood concepts can be successfully implemented. We will return to the roots of deductive databases, namely the relation to logic and automated theorem proving, and further develop deductive databases by features which imperative languages will never have, namely powerful concepts for optimization, nonmonontic negation, and disjunctions.

What is New in This Thesis?

This thesis is structured by considering different classes of Datalog programs, one in each chapter.

Chapter 2 is devoted to the standard case of Horn-clause Datalog, but with built-in predicates (such as $<$, $sum$, and so on). Pure Datalog is well known in the literature, but it seemed necessary to review it to have a basis for later extensions and comparisons. The inclusion of the standard case makes this work more self-contained and precise, since the field has not yet reached completely canonical definitions. How-
ever, Chapter 2 is not simply a compilation of parts from other textbooks and papers. I had fun in writing it and believe that some degree of originality was reached.

**Chapter 3**, “Goal-Directed Bottom-Up Evaluation” also considers the case of pure Horn-clause programs, but is devoted to the special problem of simulating the goal-directed SLD-resolution by bottom-up evaluation. This chapter is based on my paper [Bra95], but contains an important improvement in the proposed method. In Section 3.1, we compare the efficiency of bottom-up evaluation after the standard “Magic Set” transformation with the efficiency of SLD-resolution. As shown by Ross [Ros91], bottom-up query evaluation with magic sets can be much slower than SLD-resolution for tail-recursive programs. We show that this happens only for tail-recursive programs, and that the only problem of magic sets is the “materialization of lemmas” (which is also done in variants of SLD-resolution which ensure termination). In any case, “magic sets” are “as goal-directed as” SLD-resolution. It is difficult to say how new these results are, certainly they are not very surprising. There are a number of similar approaches which compare variants of magic sets and/or variants of SLD-resolution [Ull89a, Sek89, RS91]. We compare the original magic sets with the real SLD-resolution, and give simple formalizations and proofs, suitable for classroom usage.

In Section 3.2, we show that it is possible to exactly simulate SLD-resolution for tail-recursive programs and to combine it with magic sets for other programs. A variant of magic sets with tail-recursion optimization has already been proposed by Ross [Ros91], but our goal is not only to solve this problem but to simulate SLD-resolution by bottom-up evaluation as far as we can. Formally, our method is different from the method of Ross since we work with lists of literals while he uses only pairs, and we produce range-restricted Datalog, while his method yields HiLog. However, the main reason why we believe that we can make an important contribution is that our method is based on a very simple idea, namely to simulate SLD-resolution by means of a meta-interpreter in the style of Bry’s approach [Bry90b], but it gives many optimizations for free:

- If the given program is nonrecursive, then the transformed program is nonrecursive, too. For the magic set transformation, this is not necessarily the case, and this is an important problem. For instance, the magic set transformation can also be useful in standard relational databases [GM93], where the query evaluation algorithm is unable to handle recursions.
- Values for anonymous variables never get explicitly represented.
- In contrast to the “magic set” transformation, we need no extra rectification. It is automatically included in a simulation of SLD-resolution.
- Constants are “pushed downward” as far as possible already during the transformation (at “compile time”).
- Also other constraints, such as $X < 100$, can be pushed into the called rules in order to abort inconsistent paths early.
- To some degree, also incomplete bindings can be passed to the called rules. In contrast, the standard magic set transformation usually assumes that every argument is either a ground term or a free variable. However, our transformation
also needs to “abstract” the binding information over some fixed depth limit.

- Finally, the idea to simulate SLD-resolution allows also a better understanding of the counting method and its generalizations [GZ92].

Probably Section 3.2 is the most important contribution of this thesis from the practical viewpoint, and Section 4.2 is the most important contribution from a more theoretical viewpoint.

In Chapter 4, “Negation as Failure”, we extend the class of Datalog programs to allow nonmonotonic negation. In Section 4.2, we characterize the two most important negation semantics by simply requiring that certain natural elementary program transformations preserve the semantics of the program. For example, it should be possible to delete a tautological rule like \( p(X) \leftarrow p(X) \) or to “unfold” a body literal, i.e. replace it by the bodies of the rules with matching head literal. In Section 4.3, we present a general framework for the computation of different negation semantics based on the idea of conditional facts [Bry89, Bry90a, DK89a, DK89b]. We show that with very weak assumptions on the semantics (namely the possibility to delete tautologies and to perform unfolding), the derivable conditional facts are equivalent to the original program. Our approach can be described as a source-level transformation of the program, and therefore is easy to understand and to verify. We believe that it also helps to understand other proposed query evaluation algorithms for specific semantics. The results of this chapter are based on my joint work with Jürgen Dix [BD94b, BD94a, BD95b, BD95a, BD95c]. Here, I specialized the results to the case of non-disjunctive Datalog, and was able to develop them a little further. The computation of the residual program and the well-founded and stable model semantics has been implemented by Dirk Hillbrecht and Michael Salzenberg [HS96] (even in the disjunctive case).

In Chapter 5, “Reasoning with Disjunctions”, we consider another extension of pure Datalog, namely Datalog with disjunctions in the heads. We exclude nonmonotonic negation here, since the ideas presented in Chapter 4 can easily be extended to the disjunctive case [BD94b, BD94a, BD95b, BD95a, BD95c] and it seems better to consider the problems of negation and disjunctions first separately. In Section 5.1, we propose an optimization of the immediate consequence operator \( T_P \) which resolves only with a single literal of every disjunction fact and still is complete for deriving “answer clauses”. This utilizes an idea known for “positive hyperresolution” (the theorem-proving counterpart of the disjunctive \( T_P \)) for a long time [CL73], but we show that for range-restricted clauses, the optimization can be decisively strengthened [Bra94a]. By applying this idea, disjunctive rules can be naturally translated into Horn clauses with lists. This allows a direct implementation on standard deductive database systems.

We then introduce the notion of disjunction types, which allow us to generalize techniques based on the predicate dependency graph (such as determining an evaluation order) to disjunctive rules. By specializing the meta-interpreter for disjunctive rules with respect to these disjunction types, we get a quite reasonable Horn-clause implementation of a disjunctive program. By the way, the notion of disjunctive types is very similar to the “node types” used in Section 3.2. This work has not been previously published.
In Section 5.2, we use the reasoning algorithm developed in Section 5.1 for query evaluation under the stable model semantics. The stable model semantics is inherently disjunctive, and is often used to express disjunctive knowledge in non-disjunctive logic programs. So it is quite evident that for computing stable models, we need to be able to reason with disjunctions.

Bibliographical Notes

I would like to mention a number of textbooks and survey papers relevant to the topics treated in this thesis. Of course, this thesis is sufficiently self-contained so that it should not be necessary to consult a textbook. In fact, the first four chapters of this thesis may themselves be seen as a kind of textbook. However, if the reader should be interested to look at related material or alternative presentations of common material, we give the necessary references here.

Textbooks on deductive databases (or having a substantial part on them) are [Ull88, Ull89b, NT89, CGT90, Gog90, Das92, Nus92, CGH94, AHV95]. Survey papers on deductive databases are, e.g., [Llo83, GMN84, BR86, Min88b, HPRV89, CGT89, KG90, Zan90, Con91, Tsu91b, RU95]. There is also a special issue of the VLDB journal on actually implemented deductive database systems [Ram94], and there is a collection of papers on applications of deductive databases [Ram95]. General textbooks on databases with a chapter on deductive databases are, e.g., [GV89, Dat90]. For the history of deductive databases, see, e.g., [Min88b, RU95]. Textbooks on nonmonotonic reasoning are, for example, [Rei85, Bes88, Eth88, Luk90, Bre91, MT93, GHR94]. Overview papers on the semantics of nonmonotonic negation are [She88, PP90, AB94, Prz94, Prz95, Dix95a, Dix95b, Dix95c, DF96]. There is currently only one textbook on disjunctive logic programming, namely [LMR92]. A textbook on automated theorem proving is [CL73].
Chapter 2

Horn-Clause Datalog

In this chapter, we review syntax, semantics, and query evaluation for the simplest case of pure Datalog, namely definite Horn clauses. This is more or less standard material in courses and textbooks on deductive databases, but we need it for later extensions and comparisons. Furthermore, we also consider built-in predicates (such as $<$ and $\text{sum}$).

Datalog programs are queries (or view definitions) over a relational database. Therefore, we first formalize in Section 2.1 a database (or its information contents) as a Herbrand interpretation. In Section 2.2, we review syntax and semantics of Datalog. In Section 2.3, we explain how bottom-up query evaluation works in this simplest case.

2.1 Logic and Databases

We will first informally introduce relations, and then define everything formally.

Knowledge Representation in Relations

The relational model was invented by Codd. In fact, his insight was that information can be represented in a much simpler way than it was usual before, namely in the form of tables or finite relations. For example, if we want to store which customer has ordered how many units of which ware, this would look as follows:

<table>
<thead>
<tr>
<th>order</th>
</tr>
</thead>
<tbody>
<tr>
<td>jones</td>
</tr>
<tr>
<td>smith</td>
</tr>
<tr>
<td>smith</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>milk</th>
</tr>
</thead>
<tbody>
<tr>
<td>eggs</td>
<td>milk</td>
</tr>
</tbody>
</table>

Of course, this is only a logical view of the information. We could store it directly as a file of records with the three components “customer”, “units” and “ware”, but there are also other possibilities — for example we could store each customer only once and link to it a list of his orders. The important point is that the system (perhaps with help from the administrator) decides on the physical representation of the data, the user does not have to know this for posing queries.
Most data structures can be seen as relations. For instance, an array with the names of the planets can be represented as the following relation:

<table>
<thead>
<tr>
<th>planet</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>9</td>
</tr>
</tbody>
</table>

Note that it is well possible that this relation is implemented internally by the corresponding array. So we do not necessarily lose efficiency by the relational view, but we gain uniformity. For any form of programming language, it is very important to have only a small number of basic concepts (as the success of Pascal, C, and Lisp showed).

A typical mathematical relation is the “is less than” relation <. The main difference to the above relations is that < consists of an infinite number of tuples, namely all pairs \((X, Y)\) of, say, natural numbers, where \(X < Y\). Nevertheless it can be envisaged as an “infinite table”:

\[
\begin{array}{cc}
1 & 2 \\
1 & 3 \\
\vdots & \vdots \\
2 & 3 \\
\vdots & \vdots \\
\end{array}
\]

Of course, we cannot actually store such a table as a set of records, and therefore most textbooks on relational databases exclude infinite relations. However, all existing database systems have an \(<\)-operator, so this relation can be used in queries. The only difference to the stored relations is that the access to infinite relations is limited: For instance, one can print the complete \(order\)-table, but for < this must be excluded in order to guarantee that queries can be processed in finite time. The only allowed access to the <-table is to check for given numbers \(X\) and \(Y\), whether the pair \((X, Y)\) is in the table, i.e. whether \(X < Y\). Therefore, < can be represented by procedure taking the two parameters \(X\) and \(Y\) and returning the result \(true\) or \(false\).

Mathematically, a function returning a boolean value is known as a predicate. And in fact, a predicate is isomorphic to the relation consisting of those argument tuples which yield the return value \(true\). Therefore, we can use the words “relation” and “predicate” interchangeably.

In fact, some infinite relations allow more general queries than to check only whether a given tuple is in the relation or not. For instance, all database sys-
tems allow to add two numbers. Usually, \( + \) is seen as a function of two numbers, which returns a third number. But we do not really need functions if we have relations/predicates. Consider the relation \textit{sum}, which contains all triples \((X, Y, Z)\) of numbers with \(X + Y = Z\). This is known as the graph of the function, because a function \(f\) of a single argument is often graphically represented by the points with coordinates \((X, Y)\) such that \(Y = f(X)\).

A typical query to the \textit{sum}-relation would be to find a value \(Z\) with \((3, 5, Z) \in \text{sum}\). Written in the more common predicate notation, we ask for a value \(Z\) such that \(\text{sum}(3, 5, Z)\) holds. This results only in a single value, so there are no finiteness problems, and therefore the query should be allowed. However, it should also be possible to ask for a \(Y\) such that \(\text{sum}(3, Y, 8)\) is true. So the relational view has no fixed input/output arguments and usually allows taking the inverse of a function (if this is finite).

Of course, the programmer of such a system probably has to write different procedures depending on which arguments of a relation are given in the query and which are asked. But this is an internal detail, not visible for the user. The only thing the user must ensure is that sufficiently many arguments are given, such that the result of the query is finite. We can denote this by “binding patterns”, which contain the letter \(b\) or \(f\) for every argument, where \(b\) means “bound” (input argument), and \(f\) means “free” (output argument). For instance, the predicate \textit{sum} can be evaluated for the binding patterns \(ffb\), \(bfb\), \(fbb\), and \(bbb\). The predicate < can only be evaluated for the binding pattern \(bb\). Finite relations such as \textit{order} or \textit{planet} can usually be evaluated for arbitrary binding patterns. However, depending on the available access structures, very different procedures may be executed by the system. For instance, if \textit{planet} is stored as an array, only the binding patterns \(bf\) and \(bb\) can use the efficient array indexing mechanism, for \(fb\) a complete linear search is necessary. But these are implementation details, the system automatically selects the right procedure if one is available.

The Herbrand Universe

In order to define a relational database formally, we first need a syntactic basis, i.e. a set of symbols (lexical elements, tokens, words) called the alphabet:

\textbf{Definition 2.1.1 (Alphabet):}

An alphabet \(\text{ALPH}\) is some infinite, but enumerable set, the elements of which are called symbols. It has to contain at least the logical symbols, i.e. \(\text{LOG} \subseteq \text{ALPH}\), where

\[\text{LOG} = \{(, ), \,, \wedge, \vee, \leftarrow, \neg\}\].

A subset of this alphabet can be used to denote objects in the domain of discourse, or elements of the underlying datatypes. In Prolog, sequences of letters and digits beginning with a lowercase letter are used for this purpose, plus the usual datatype constants for numbers and strings. But the theory works of course with any possible
such set, and for instance the CORAL-system allows more general “atoms” than Prolog. Note that also constants of the form 10kg, 27.05.1965, or /usr/bin could be useful.

**Definition 2.1.2 (Domain):**

Let $\text{DOM} \subseteq \text{ALPH} – \text{LOG}$ be non-empty. It is called the domain or the universe. Its elements are called constants, objects, or datatype elements.

In this work, we do not distinguish between syntax and semantics, i.e. between the names and the objects themselves. This is quite customary in the theory of logic programming. I once worked out this whole chapter with a clear distinction between syntax and semantics (as usual in mathematical logic), but it only complicated things without leading to new insights.

In mathematical logic, the special type of an interpretation which uses the syntactical elements also for the semantics is known as a Herbrand interpretation. Even if we would want to make a distinction between syntax and semantics, it is quite clear that in the database context every object must have a name (DCA, “domain closure axiom”) and distinct names must refer to distinct objects (UNA, “unique names axiom”).

First, it seems pretty clear that in the deductive database context there should be no unnamed (“anonymous”) objects: We must be able to print the query results, and it would be very unnatural to distinguish between object names in answers and constants in queries. It has been noted by Przymusinski [Prz88b] that for giving answers containing variables (representing “for all” statements), it is important to consider also non-Herbrand interpretations, since otherwise we might get a strange behaviour. However, we will consider here only range-restricted (allowed) programs, which exclude these problems. Even if we had unnamed objects, we could not refer to them in range-restricted programs. So in our context, the DCA is only natural. However, there are good reasons for not assuming it in general logic, for instance it would destroy the compactness property.

The necessity of the unique names assumption can be motivated also with the printing of answers: Assume for instance that we get the answer $X = a$ for the query $p(X)$. Then we implicitly assume that $p(X)$ is false for all other $X$. This is nothing else than the closed world assumption CWA. Without it, the positive answers on a query would not be sufficient, because they would not give us the complete information about the query result. But of course, the CWA is false if there can be synonyms, i.e. there is a $b$ with $b = a$.

However, we must note at this point that all real systems trivially violate the unique names assumption: For instance, 0, −0, and 0.00 denote the same value, as well as 1.1.94, 01.01.1994, and 01-JAN-94. However, the user knows these equivalences, and would not assume that if something is true for 1.1.94, it might be false for 01.01.1994. So if we keep one fixed and well-known interpretation of the constants, it might be possible to allow violations of the UNA. Note, however, that when we print answers, we should print only one representative of every equivalence class, because otherwise it becomes difficult for the user to see how many different results the query has. Also in our computations, we have to stick to one representative,
because otherwise things like unification would not work. So this violation of the UNA is only syntactical sugar for the input of data and queries, and can be ignored in the following.

There is one important application of UNA-violations, which we exclude in this work, namely the treatment of null-values by Skolem constants. There are quite a number of semantically different notions of null-values, but the most common is “value exists, but is presently unknown”. All standard relational database systems have some sort of null-values, but their semantical treatment is very unsatisfactory — for instance, \( X \geq 0 \text{ OR } X < 0 \) would fail if \( X \) is a null-value. It seems to be very difficult to achieve a better behaviour, and therefore we do not treat this problem here. However, if the null-value is from some known and finite domain, it can be treated with the techniques from disjunctive logic programming presented in Chapter 5. Let us finally remark that although Skolemization is natural and simple in automated theorem proving, it does not match with the closed world assumption as explained above. And the CWA is essential in deductive databases.

**Datatypes**

For simplicity, we also do not consider a type system in this work, i.e. we do not consider \( DOM \) as structured into different, possibly overlapping, types (such as \textit{string}, \textit{integer}, \textit{float}, ...).

We believe that a type system is extremely valuable for practical applications, but it is something orthogonal to the topics treated here. Again, in a previous version of this chapter, I used a type system, but it only complicated the definitions, and probably was still too simple for practical usage. It seems better to separate the two problems, and consider here the union of all types, while another chapter could specifically treat the type-checking. It is often helpful to have different possible views on the same subject, which show one or the other part in more or less detail, or which are simply more convenient for different applications.

Up to now, most implemented deductive database systems have followed Prolog’s tradition of being untyped, and this is at least an excuse for not treating this important subject. However, nearly all relational database systems are typed. One of the experiences of the CORAL deductive DBS project [RSSS94] was that a type system would have been really useful. First, type information might allow to catch some errors, and second, it is useful for certain optimizations, for instance in storage representation.

We can refer here only to the literature, for instance in [MO84, XW88, YS91, CGH94] type systems for Prolog and deductive databases have been proposed.

**Structured Terms**

Prolog and nearly all deductive database prototypes allow also structured terms as data. They can be seen as trees, where the nodes are marked by operator symbols. So Prolog terms are simply a data-structure, and correspond to variant records in pro-
gramming languages like Pascal. For instance, geometric objects could be represented in Prolog as `rectangle(Width, Height), circle(Radius), or text(String)`.

The arguments of terms are again terms, so in contrast to Pascal we do not need explicit pointers in order to create linked structures. For instance, lists are represented in Prolog with two functors: The 0-ary functor (i.e. constant) `[]` represents the empty list, and `.(Head, Tail)` creates a new list with first element `Head` and rest list `Tail`. So the list `[1,2]` would be represented in Prolog as:

```
1
  2
   []
```

Such recursive data-structures would be of little use in relational database systems, because their query language does not support recursion. Therefore, they are not contained in standard RDBMS. In fact, one important motivation for deductive databases is their ability to handle such recursive structures.

We will treat such variant records like any other datatype, i.e. we will assume that the (variable-free) terms are also elements of `DOM`, and that there are datatype predicates for constructing records/terms and for selecting their components. For instance, `cons(H, T, L)` can be called with binding pattern `bbf` in order to construct the list `L` with head `H` and tail `T`, or it can be called with binding pattern `ffb` in order to split the list `L` into head `H` and tail `T`. Of course, instead of `cons(H, T, L)` it is more common to write `L=[H|T]`. Of course, we should allow this convenient notation, but this is only “syntactical sugar”. In fact, if `L` “is equal to” `[H|T]`, we can also use the functional notation of Prolog and replace `L` by `[H|T]`. For instance, we should allow the standard definition of the concatenation of lists:

```
append([], L, L).
append([H|T], L, [H|A]) :- append(T, L, A).
```

But internally, this is translated into:

```
append([], L, L).
append(X, L, Y) :- X=[H|T], Y=[H|A], append(T, L, A).
```

Of course, the sequence of body literals must be chosen according to the allowable binding patterns, but this is a standard technique in deductive databases. The advantage of our approach is that we do not have to treat evaluable functions different from term constructors. For instance, the following would be legal:

```
factorial(0, 1).
factorial(N, N*F) :- N > 0, factorial(N-1, F).
```

In Prolog it is a common source of programming errors to write such things, and it is probably advantageous to legalize it. This has also been tried in Turbo Prolog and CORAL, but not with an automatic reordering of body literals, which is quite essential for this approach to work well.
There is one type of data in Prolog, which we will not support in our approach. These are incomplete data-structures, which have holes (variables) in them, to be filled in later. Important applications of these data-structures are meta-programming and difference-list techniques. Our reasons for excluding incomplete data-structures are as follows:

- First, such data structures complicate the theory as well as the implementation, e.g. without them, bottom-up evaluation needs only a one-directional matching, while with them, general unification (including the “occur check”) is required.
- Second, excluding them allows us to catch errors and analyse the program text more easily.
- Third, although we will temporarily use meta-programming in Chapter 3, this is very untypical for deductive databases. We believe that it is an important difference between Prolog and deductive databases that Prolog is more general in this respect. Difference-lists are hard to understand declaratively, e.g. the “append”-predicate can only concatenate “compatible” lists. So it is anyway questionable whether we should allow them.

Different system designers have decided differently in this aspect, e.g. LOLA and Aditi do not allow incomplete data structures, while CORAL allows them\(^1\). Terms with variables in them could be introduced in our approach as special datatypes, but then unification would have to be done explicitly (with new built-in predicates).

### Database Schemas and Predicates

Above, we have formalized the names of the objects and data values we want to store and process in our database as a set $\text{DOM}$. In fact, $\text{DOM}$ needs only to be a superset of the identifiers actually used in a specific application. So it is probably built into the database management system and cannot be changed.

We have also explained that the data structures and datatype operations we want to use can be seen as relations or predicates. Since a database can of course contain more than one relation, we need names for them. The names we are going to use in our application are defined by a set $\text{PREDS}$. So this set corresponds to the database schema. However, we assume that $\text{PREDS}$ contains also built-in predicates like $\text{sum}$ and $<$, which are usually not declared in the database schema because they are given by the system. So there are two types of predicates:

- Built-in predicates are defined by means of procedures within the system software. So their extensions are fixed and usually infinite.
- EDB-predicates are defined “extensionally” by enumerating the tuples in their extension. So their extensions are finite and usually can be changed by inserting and deleting tuples.

However, the distinction between built-in predicates and EDB-predicates is not so clear cut: For instance, assume that we store a set of objects with attributes (like C-structures) in a heap (either in dynamic memory or on disk). Then we can access

\(^1\)In the latest version of CORAL they have been excluded for efficiency reasons.
them only if we know their address, represented as first argument. In this way, it looks like a built-in predicate. However, tuples can be inserted or deleted, i.e. records created or their storage reclaimed. And usually only EDB-predicates are dynamic.

Another example, where the distinction is not so clear, is the system catalog (data dictionary): Although the extensions are finite, it is usually not allowed to insert or delete tuples, because this would destroy the consistency between different relations of the system. Instead, there are special commands for declaring or deleting EDB-predicates. In fact, the system might well use a special data structure for its catalog, and not the standard implementation of user-defined relations.

Therefore, and also for technical simplicity, we do not distinguish in the following between built-in and EDB-predicates. Of course, in a practical system there are important differences. For instance, it is much simpler to define a new EDB-predicate than to introduce a new built-in predicate into the system (if at all possible).

Another important difference is that some relations can be updated and others cannot. For instance, we could optimize a Datalog program by evaluating calls to predicates with a fixed interpretation (if possible), while such an optimization should be independent of the current values of updable relations. However, the “magic set” transformation which we consider in Chapter 3 does not use any knowledge about the built-in predicates. So we again need no distinction between different kinds of database predicates.

**Definition 2.1.3 (Predicates):**

Let \( \text{PREDS} \subseteq (\text{ALPH} \setminus \text{LOG}) \times \mathbb{N}_0 \). The elements are called predicates or relation names and denoted by \( p/n \), where \( n \) is called the arity (number of arguments) of \( p \).

In the tradition of Prolog, we allow that the same predicate can be used with different arities. Allowing only one arity for every predicate would not make the definitions any simpler.

In contrast to \( \text{DOM} \), which we consider as fixed, we will sometimes need to introduce new predicates, for instance during the “magic set” transformation (see Chapter 3). We will also have to distinguish the predicates defined in the database from the predicates defined by rules. Therefore, it seems natural that the set \( \text{PREDS} \) depends on the application and is not simply a sufficiently large namespace.

**Database States and Herbrand Interpretations**

The smallest information unit of the database is that some tuple \((c_1, \ldots, c_n)\) from \( \text{DOM}^n \) is contained in a relation \( p/n \). We call it a fact, a ground literal, or an elementary proposition:

**Definition 2.1.4 (Fact):**

A fact \( F \) consists of a predicate \( p/n \in \text{PREDS} \) and an \( n \)-tuple of domain elements \((c_1, \ldots, c_n) \in \text{DOM}^n \). It is usually denoted by \( p(c_1, \ldots, c_n) \), but sometimes also infix- and mixfix-notation is used (like \( c_1 < c_2 \) or \( c_3 = c_1 + c_2 \)) and in the case \( n = 0 \) we write \( p \) instead of \( p() \).
We try to distinguish here between the abstract syntax, which contains only the semantically important components, and the concrete syntax, which talks also about edge and allows some syntactical sugar. The concrete syntax is irrelevant for the theory, but it is of course essential in a practical system.

In abstract syntax, we would for instance say that a goal is a finite sequence of literals, while in concrete syntax, it is a string over \( \text{ALPH} \) of the form \( L_1 \land \cdots \land L_m \), where the \( L_i \) denote literals. The abstract syntax is better suited for mathematical definition, and corresponds also more to the internal datastructures used in an implementation.

The Herbrand base is the set of all possible facts which can be built with predicates from \( \text{PRED} \) and constants from \( \text{DOM} \):

**Definition 2.1.5 (Herbrand-Base):**
Let \( \text{BASE} \) be the set of all facts, i.e. \( \text{BASE} \) is the subset of \( \text{PRED} \times \text{DOM}^* \), where the arity of the predicate is equal to the length of the argument string. We we want to make the dependency on \( \text{PRED} \) explicit, we also write \( \text{BASE}(\text{PRED}) \).

Now a state of the database determines for every possible fact whether it is currently true or false. We call this a Herbrand interpretation, or, for short, an interpretation. Of course, the interpretation of the datatype predicates (like \( \text{sum} \) and \(<\) ) is usually fixed, but for uniformity, we view them also as part of the database state.

There are two equivalent formalizations of Herbrand interpretations. We will use both, whatever is more convenient. The first view is that a Herbrand interpretation is the set of all facts which are currently true:

**Definition 2.1.6 (Herbrand-Interpretation):**
A Herbrand-interpretation \( I \) is any subset of \( \text{BASE} \).

- If \( F \in I \), we say that the fact \( F \) is true in \( I \) and write \( I \models F \).
- If \( F \in \text{BASE} \), but \( F \not\in I \), we say that \( F \) is false in \( I \) and write \( I \not\models F \).

The second view is that a Herbrand interpretation interprets each predicate (name) by a relation, called the extension of the predicate. In this way, an interpretation assigns a meaning to the predicate names. This also corresponds to the view that a relational database stores under each relation name a relation.

**Definition 2.1.7 (Extension of a Predicate):**
The extension of a predicate \( p/n \) in an interpretation \( I \) is defined as follows:

\[
I[p/n] := \{(c_1, \ldots, c_n) \in \text{DOM}^n \mid I \models p(c_1, \ldots, c_n)\}.
\]

So we can view an interpretation equivalently as a mapping which assigns to every \( p/n \in \text{PRED} \) a relation \( I[p/n] \subseteq \text{DOM}^n \).

**Binding Patterns**
We already explained above that for instance the predicate \( \text{sum} \) corresponding to \( X + Y = Z \) can only be called with the “binding patterns” \( \text{bbf}, \text{bfb}, \text{fbb}, \) and \( \text{bbb} \).
Formally, a binding pattern simply specifies one of the two letters b and f for every argument of a predicate:

**Definition 2.1.8 (Binding Pattern):**

- A binding pattern for a predicate $p/n$ is mapping $\beta: \{1, \ldots, n\} \rightarrow \{b, f\}$. It is usually denoted by the string $\beta(1) \ldots \beta(n)$.
- Given such a binding pattern $\beta$, we will call the $i$-th argument of $p$ bound if $\beta(i) = b$, and free if $\beta(i) = f$.

Now we assume that the system designer has specified for every built-in predicate a set of allowed binding patterns. Probably there is one procedure for every binding pattern of a built-in predicate, so this specification tells us which binding patterns are implemented.

Of course, for standard EDB-relations (finite tables), every binding pattern is possible, but some binding patterns might be specifically supported by indexes. So there are also different algorithms for the access to EDB-relations, just as there are different implementations of built-in predicates, depending on the binding pattern.

Finally, it is usually required in deductive databases to specify binding patterns (or “modes”) for the IDB-predicates defined by rules. Also some Prolog systems can use such “mode” declarations in order to generate more efficient code. So we have a binding pattern specification for all predicates, not only for the built-in datatype predicates:

**Definition 2.1.9 (Binding Pattern Specification):**

A binding pattern specification is a mapping $\text{BIND}$ which assigns to every predicate $p/n \in \text{PREDS}$ a non-empty set $\text{BIND}(p/n)$ of binding patterns for $p/n$.

For instance, a lazy system designer might allow only the binding pattern $bfb$ for $\text{sum}$. However, a call of the form $\text{sum}(3, 4, 5)$ corresponding to the binding pattern $bbb$ can still be executed by first computing the output value $X$ of $\text{sum}(3, 4, X)$ and then checking whether $X = 5$. So some binding patterns imply other binding patterns. But in many cases it is better to make use of bound arguments by implementing a specialized procedure for them. For instance, the binding pattern $ff \ldots f$ is probably supported for all finite EDB-relations, but a full table scan is often not the most efficient way to access the tuples we need.

**Definition 2.1.10 (More General Binding Pattern):**

Let $\beta$ and $\beta'$ be two binding patterns for a predicate $p/n$. We say that $\beta$ is more general than $\beta'$ iff $\beta(i) = b \implies \beta'(i) = b$ for $i = 1, \ldots, n$.

There is one consistency requirement for the specification of valid binding patterns and the actual implementation: For instance, the system designer cannot allow the binding pattern $fff$ for $\text{sum}$, when $I[\text{sum}]$ is infinite. The result of a call to a built-in predicate with specific values for the bound arguments must always be finite.
2.1. LOGIC AND DATABASES

Definition 2.1.11 (Allowed Interpretation):
Given a binding pattern specification $BIND$, a Herbrand interpretation $I$ is allowed iff for every $p/n \in PREDS$ and every $\beta \in BIND(p/n)$ the following holds:

- Let us call two tuples $(a_1, \ldots, a_n)$ and $(b_1, \ldots, b_n)$ in $I[p/n]$ equivalent wrt $\beta$ iff they have the same values in the bound argument positions, i.e. $a_i = b_i$ for every $i$ with $\beta(i) = b$. Then the equivalence classes wrt this relation are finite.

When we talk about computability, we will assume the following:

- For every $p/n \in PREDS$ and every $\beta \in BIND(p/n)$ there is a terminating procedure $p^\beta$ which computes all $(c_1, \ldots, c_n) \in I[p/n]$ given the $c_i$ with $\beta(i) = b$.

Of course, there are additional interesting informations about predicates, which could be described by means of “generalized binding patterns”. For instance, sometimes it might be guaranteed that there is only a single output value. This corresponds to a key of the relation. From an implementation oriented point of view, it is also important to assign “costs” to different binding patterns. Furthermore, if there are multiple output values, it might be useful to know whether they are produced in a sorted sequence. There might also be different implementations of a predicate depending on whether we have only a single tuple of input values, or a whole input relation. Furthermore, if we do not need all output arguments, also projections could be encoded in form of binding patterns.

However, it seems that not all access structures can be defined by means of binding patterns. For instance, there might be a special support for evaluating

\[ p(X) \leftarrow \text{age}(X, Y) \land Y \geq 20 \land Y < 30 \]

(by means of a B-tree on the second argument of $\text{age}$). In an extensible database system there might also be access structures for finding points in a given rectangle. So it would not help much to treat only the special case of orders. Also the possibility of ORACLE to cluster two relations together can be seen as a specific support for conjunctions of these two relations.

Summary

So a database defines three things:

- The set of predicates $PREDS$ used to store the relevant information about the domain of application. They are also used for denoting operations on the datatypes.
- A set of possible binding patterns $BIND(p/n)$ for every predicate $p/n \in PREDS$, which describes the implemented datatype operations and the available access structures for EDB-relations.
- An interpretation $I$, which defines the current extensions of the predicates, i.e. which elementary propositions about the modeled minworld are true.

We have defined here only the information contents of the database, and not its query language. The query language “Datalog” is topic of the next section.
2.2 Syntax and Semantics of Datalog

Given a database, we define a set of derived predicates by means of rules, i.e. a logic program. These predicates are called IDB-predicates because the rules make the “intention” behind these predicates clearer than a simple enumeration of the tuples in their extension.

So our set $PRED_S$ of predicates contains now not only the database predicates, but also the rule-defined predicates. Of course, all the definitions of the preceding section remain valid, only the intuition is slightly changed. If we have to make the distinction clear, we consider $PRED_S$ as partitioned into two disjoint subsets $EDB$ and $IDB$.

There are in fact (at least) two uses of IDB-predicates in a deductive database: First, such predicates correspond to views in relational databases, which are defined permanently by the database administrator. Second, the user writes a query, which is again a logic program, and temporarily defines new predicates.

In this work, it is not necessary to distinguish between predefined view predicates and temporarily defined query predicates. We will always consider them together and assume that are defined by only one logic program. However, a practical system could make use of the information that the defined views will exist longer. By the way, queries in application programs, which are executed again and again, should be declared as views (also because of the sharing of programs, mentioned in the introduction). So there is anyway no clear border between the two applications of IDB-predicates.

An important point is that after we evaluated a logic program, we get a new database defining the extensions of an enlarged set of predicates. So whatever query language we have for databases could also be used to query the IDB-predicates. However, since the definition mechanism for IDB-predicates is so powerful, we will in fact use a very simple query language (see below).

Datalog Programs

Besides predicates and constants, Datalog-programs contain also variables, i.e. placeholders for domain elements. We only need a sufficiently big supply of names here. “Sufficiently big” means at least for SLD-resolution (see Chapter 3) that the set of variables must be infinite, because it must always be possible to replace the variables of a rule by new variables. Of course, the name space for variables is built into the system and cannot be changed. Prolog and most deductive database systems use strings of letters and digits starting with an uppercase letter as variables. Of course, these identifiers then cannot be used as elements of $DOM$, since variables and constants can both occur as arguments of predicates, and it is necessary to distinguish them.

**Definition 2.2.1 (Variables):**

Let $VARS \subseteq ALPH - (LOG \cup DOM)$ be an infinite set. The elements of $VARS$ are called variables.
2.2. SYNTAX AND SEMANTICS OF DATALOG

Next, we introduce terms. They denote domain elements and can be variables or constants. Often structured terms containing function applications (such as \( N+1 \) or \([H|T]\)) are allowed, too, but not in pure Datalog. However, as explained above, we can treat them as a form of syntactic sugar which is replaced by a new variable and a call to the corresponding built-in predicate. So this is in itself not really a restriction, but because of the range-restriction requirement (see below), we do not get full Prolog.

**Definition 2.2.2 (Term):**

A term \( t \) is either a constant from \( \text{DOM} \) or a variable from \( \text{VARS} \). So the set of terms is \( \text{TERMS} := \text{DOM} \cup \text{VARS} \).

It is also customary to allow the shorthand \( \_ \) for a variable with unspecified name (“anonymous variable”). Since we have an infinite supply of variables in \( \text{VARS} \), it is possible to envisage a preprocessor, which replaces every occurrence of the “term” \( \_ \) by a new variable, which is not otherwise used in the rule. Of course, different occurrences of \( \_ \) must be replaced by different variables.

Next on the agenda are atomic formulas (or atoms). They denote truth values determined by applying a predicate to domain elements. We call these atoms also literals, which is incompatible with the standard literature on logic and automatic theorem proving. However, it is customary to talk of head- and body-literals of a rule (see below), and at least the head-literal is in fact always an atom. But nobody says “the head-atom”, which would be more precise. Furthermore, in Chapter 4, we will use atoms also for truly negative literals. Finally, if we see a rule in clause notation (as a disjunction), the body literals are really negative literals. Therefore, we use the terms “atom” and “literal” as synonyms.

**Definition 2.2.3 (Literal):**

A literal consists of a predicate \( p/n \in \text{PREDs} \) and an \( n \)-tuple \((t_1, \ldots, t_n) \in \text{TERMS}^n\) of terms. It is usually written as \( p(t_1, \ldots, t_n) \), but we allow the same syntactic freedom as explained above for facts. Let \( \text{LITS} \subseteq \text{PREDs} \times \text{TERMS}^n \) be the set of all literals.

- We call \( \text{pred} (p(t_1, \ldots, t_n)) := p/n \) the predicate of the literal \( p(t_1, \ldots, t_n) \).
- The term \( t_i \) is called the \( i \)-th argument of the literal \( p(t_1, \ldots, t_n) \).

Most deductive databases do not use attributes, i.e. symbolic names for the relation-columns/predicate-arguments. As usual in Prolog and the domain calculus, the arguments are identified by their position. However, attributes have proven to be very useful in relational databases. They seem not to be so essential in Prolog, though, because there the predicates (like procedures in other languages) have only a small number of arguments, which usually must be completely specified in a call. This is an important difference to database relations (EDB-predicates): They can have a large number of columns (ten or even hundreded are not seldom in practical applications). And if we need only two or three of these many attributes, it would be very inconvenient if we had to fill all other columns by anonymous variables.

There are at least two possibilities to solve this problem: First, we could for
instance allow the notation

\[ \text{\texttt{product\_data}}(\text{No}: X, \text{Weight}: Y), \]

where the arguments are explicitly named. A preprocessor could replace this by a literal in which all other columns are filled with anonymous variables. Second, we could replace “\texttt{product\_data}” (without arguments) by a literal in which all columns are filled with the attribute name as a variable, and \texttt{product\_data} \( X \) by

\[ \text{\texttt{product\_data}}(X.\text{No}, X.\text{Name}, \ldots) \]

(assuming that \texttt{VARS} allows variable names containing “.”). After all, a tuple, seen as a mapping from attribute names to values, is formally nothing else than a variable assignment. Of course, then we would have to allow also literals of the form \( X = 1 \), because with this notation it is not possible to write the value directly as an argument. But this approach makes rule bodies look similar to standard relational queries, so it probably increases the acceptance of deductive databases.

Let us also note that not all deductive databases are based on the Prolog notation, one exception is DECLARE \cite{KSSD94}. Currently, it is very uncommon not to use the Prolog notation, but when standard commercial relational databases will be extended to have deductive capabilities, they will of course stick as near as possible to SQL. We should also mention the system VALIDITY \cite{FGVLV95}, which has combined Datalog with attributes and some object-oriented features.

Next, we need conjunctions as conditions of the Datalog rules. They are called “goals” because they are also used in SLD-resolution as the current goal of the prover:

\textbf{Definition 2.2.4 (Goal):}

A goal is a finite sequence of literals, i.e. the set of all goals is \( \text{GOALS} := \text{LITS}^* \). A goal is usually written as \( L_1 \land \cdots \land L_m \), the empty goal is written as “true” or “\( \Box \)”.

A Datalog-program is a set of Datalog-rules, which are if-then-rules of a particularly simple form (definite Horn-clauses):

\textbf{Definition 2.2.5 (Datalog-Rule):}

A Datalog-rule \( \phi \) consists of a literal \( p(t_1, \ldots, t_n) \) and a goal \( L_1 \land \cdots \land L_m \). It is written as \( p(t_1, \ldots, t_n) \leftarrow L_1 \land \cdots \land L_m \). We say that

\begin{itemize}
  \item \( p(t_1, \ldots, t_n) \) is the head of the rule.
  \item \( L_1 \land \cdots \land L_m \) is the body of the rule, and every \( L_i \) is called a body literal of it.
  \item The rule is called a rule about the predicate \( p/n \).
\end{itemize}

The set of rules is \( \text{RULES} := \text{LITS} \times \text{GOALS} \). It we want to emphasize the dependency on \( \text{PRED}\text{S} \), we also write \( \text{RULES}(\text{PRED}\text{S}) \).

Since variables are not explicitly declared, it might happen that a typing error in a variable name is not detected. In order to avoid this, it is common in Prolog systems to require that every non-anonymous variable occurs at least twice in a rule.
The above definition allows also rules violating the allowed binding patterns for the built-in predicates. We will below introduce the important notion of a range-restricted rule, and thereafter allow only such rules. However, the construction of the minimal model works also without this restriction, so we can assign the above rules a well-defined (albeit not practically computable) semantics.

**Definition 2.2.6 (Logic Program):**

A logic program $P$ is a set of rules, i.e. $P \subseteq RULES$.

For logic programs, there are also simple checks for avoiding obvious errors. Since the IDB-predicates are usually not explicitly declared (at least not in Prolog), it is required that for every predicate $p/n$ which occurs in the body of a rule (and which is not an EDB-predicate), there is also a rule about $p/n$. In fact, if there is not at least one non-recursive rule, the predicate can never be true (in logic programming semantics based on the minimal model, see below).

We sometimes need also infinite programs, when we consider a Herbrand interpretation as part of a logic program. By our definitions, $BASE \subseteq LITS$, and we can turn every fact $p(c_1, \ldots, c_n)$ easily into the rule $p(c_1, \ldots, c_n) \leftarrow true$. We formalize this as follows.

**Definition 2.2.7 (Union of a Herbrand Interpretation and a Logic Program):**

Let $DB$ be a Herbrand interpretation and $P$ be a logic program. Then we (re-)define the union of $DB$ and $P$ as the following logic program:

$$DB \cup P := \{ F \leftarrow true \mid F \in DB \} \cup P.$$ 

Now, finally, a deductive database consists of a database (Herbrand interpretation) $DB$, which defines the EDB-predicates, and a logic program $P$, which defines the IDB-predicates. We assume that $DOM$ and $VARS$ are built into the system, and will keep them fixed in always fixed, while $PRED$ and $BIND$ are application-dependent and should be mentioned:

**Definition 2.2.8 (Deductive Database):**

A deductive database $DDB = \langle EDB, IDB, BIND, DB, P \rangle$ consists of

- $EDB$ and $IDB$, two finite and disjoint sets of predicates,
- $BIND$, a binding pattern specification for $PRED := EDB \cup IDB$,
- $DB \subseteq BASE(EDB)$, a Herbrand-Interpretation which is allowed with respect to $BIND$,
- $P \subseteq RULES(PRED)$, a finite logic program which contains only rules about predicates in $IDB$.

The logic program corresponding to $DDB$ is defined as: $\overline{DDB} := DB \cup P$.

It is often very convenient to view a deductive database as one logic program which defines both, EDB- as well as IDB-predicates. For instance, when we investigate the semantics of logic programs, we do not need a special treatment of the EDB-predicates.
In other contexts, such as the magic set transformation, the distinction is essential, since we want this transformation to be independent of the current database.

Models and the Minimal Model

After defining the syntax of Datalog-programs, we now define their semantics, namely a special interpretation (the minimal model) given by a database $DB$ and a program $P$. So the semantics of a program is simply an extended relational database, in which the view definitions are “materialized”. In other words, we are going to define the extensions of the IDB-predicates.

First, the rules also contain variables, which are placeholders for domain elements. The intension is that the rule has to hold for all possible assignments of values to the variables occurring in the rule. In the case of Herbrand interpretations, a variable assignment is the same as a ground substitution. We need later (in Chapter 3) also general substitutions, so we introduce them first, and then specialize them to the case of ground substitutions.

A substitution replaces every variable by a constant or another variable. It simplifies further definitions to assume that substitutions are defined on the complete set $VARS$ of variables, but since we want that they can be finitely represented, we assume that they are the identity mapping for all variables except finitely many.

**Definition 2.2.9 (Substitution):**

A substitution is a mapping $\theta: VARS \rightarrow TERMS$, where the set of variables $X$ with $\theta(X) \neq X$ is finite. It is usually denoted in the form $\langle X_1/t_1, \ldots, X_n/t_n \rangle$, where $t_i = \theta(X_i)$ and $\{X_1, \ldots, X_n\} = \{X \in VARS | \theta(X) \neq X\}$.

The definition domain of substitutions can easily be extended from $VARS$ to $LITS$, $GOALS$, and $RULES$: We simply apply the substitution to all occurring variables. It is customary to use postfix notation for the application of a substitution to a term/literal/goal/rule. It is simply shorter to write, e.g., $A\theta$ instead of $\theta(A)$.

**Definition 2.2.10 (Application of a Substitution):**

The value of a term/literal/goal/rule under a substitution $\theta$ is defined as:

- $t\theta := t$ if $t \in DOM$, and $t\theta := \theta(t)$ if $t \in VARS$.
- $(p(t_1, \ldots, t_n))\theta := p(t_1\theta, \ldots, t_n\theta)$.
- $(L_1 \land \cdots \land L_m)\theta := (L_1\theta \land \cdots \land L_m\theta)$.
- $(A \leftarrow L_1 \land \cdots \land L_m)\theta := (A\theta \leftarrow L_1\theta \land \cdots \land L_m\theta)$.

Next, we need the set of variables occurring in a rule. This is exactly what the name says:
Definition 2.2.11 (Variables Occurring in a Rule):
The set of variables occurring in a term/literal/goal/rule is defined as follows:

- $\text{vars}(t) := \emptyset$ if $t \in \text{DOM}$, and $\text{vars}(t) := \{t\}$ if $t \in \text{VARS}$.
- $\text{vars}(p(t_1, \ldots, t_n)) := \text{vars}(t_1) \cup \cdots \cup \text{vars}(t_n)$.
- $\text{vars}(L_1 \land \cdots \land L_m) := \text{vars}(L_1) \cup \cdots \cup \text{vars}(L_m)$.
- $\text{vars}(A \leftarrow L_1 \land \cdots \land L_m) := \text{vars}(A) \cup \text{vars}(L_1) \cup \cdots \cup \text{vars}(L_m)$.

Now a ground substitution for a rule replaces the variables occurring in that rule by constants. The prefix “ground” is always used to indicate that there are no variables. For instance, a fact is also called a “ground literal”. A ground substitution removes all variables from the rule and yields a “ground instance” of that rule:

Definition 2.2.12 (Ground Substitution):
A substitution $\theta$ is called a ground substitution for $\mathcal{V} \subseteq \text{VARS}$ iff for every $X \in \text{VARS}$:

- If $X \in \mathcal{V}$, then $\theta(X) \in \text{DOM}$.
- If $X \notin \mathcal{V}$, then $\theta(X) = X$.

A ground substitution for a rule $\phi$ is a ground substitution for $\text{vars}(\phi)$.

We now define which Herbrand-interpretations satisfy all rules of a logic program. In this way, we assign a meaning to the rules of a program and especially formalize the following:

- “$\land$” means logical and, i.e. all body literals of a rule must be satisfied before it is applicable.
- “$\leftarrow$” means logical “if”, i.e. if the right part of the rule (the body) is true, also its left part (the head) must be true.
- All variables are implicitly universally quantified, i.e. the rules must be satisfied for all assignments of values to variables, i.e. for all ground substitutions. If a variable occurs only in the body of a rule, it is logically equivalent to quantify it existentially in the rule body.

Definition 2.2.13 (Model of a Logic Program):
A Herbrand interpretation $I$ is called a model of a logic program $P$ (written $I \models P$) iff for every rule $A \leftarrow B_1 \land \cdots \land B_m$ in $P$ and every ground substitution $\theta$ for this rule, the following holds:

- If $I \models B_i \theta$ for $i = 1, \ldots, m$, then $I \models A \theta$.

Of course, the intended interpretation given by a program $P$ must be a model of $P$. However, there are models of $P$ which contain unnecessary tuples — after all, the rules say only what must be true, and not, what must be false. For instance, suppose that the program $P$ consists of the single rule

\[ \text{price\_with\_tax}(X, Z) \leftarrow \text{price}(X, Y) \land Z = Y \times 1.15, \]
where \( \textit{price} \) is an EDB-predicate containing the single tuple \((\textit{coffee}, 1.00)\). Then \( \textit{price\_with\_tax} \) must contain the tuple \((\textit{coffee}, 1.15)\) in any model. However, the interpretation remains a model, if we add further tuples, such as \((\textit{tea}, 0.75)\). Of course, in the intended model only the tuples required by the rules hold. Thus, we say that an interpretation is preferable, if it contains fewer tuples:

**Definition 2.2.14 (Preferable Interpretation):**

We call a Herbrand interpretation \( I_1 \) preferable (or equal) to a Herbrand interpretation \( I_2 \) (written \( I_1 \preceq I_2 \)) iff \( I_1 \subseteq I_2 \).

We call \( I_1 \) (strictly) preferable to \( I_2 \) (\( I_1 \prec I_2 \)) iff \( I_1 \preceq I_2 \) and \( I_1 \neq I_2 \).

Now the most preferable interpretation obviously has empty extensions for all predicates. But if we look for the most preferable interpretation only among the models of \( P \), it turns out that we get a uniquely determined interpretation, which we consider as the semantics of a Datalog-program:

**Definition 2.2.15 (Minimal Model):**

A Herbrand interpretation \( I \) is called a minimal model of a logic program \( P \) iff

- \( I \models P \),
- there is no Herbrand interpretation \( I_0 \) with \( I_0 \prec I \) and \( I_0 \models P \).

The minimal Herbrand model of a set of Horn clauses has been introduced as the intersection of all Herbrand models in \([\nu\nu76]\). This shows that there is always exactly one minimal Herbrand model.

**Proposition 2.2.16 (Existence and Uniqueness of Minimal Model):**

For every logic program \( P \) there is exactly one minimal model of \( P \).

Now in order to assign a meaning to a deductive database \( \text{DDB} \), we can simply take the minimal model of \( \text{DDB} := \text{DB} \cup P \).

Equivalently, it is also possible to treat the extensions of the EDB-predicates as fixed, and minimize only the extensions of the IDB-predicates. After all, the database already defines the meaning of the EDB-predicates and it would be strange to consider interpretations with different extensions of predicates like \( \text{sum} \).

**Definition 2.2.17 (Preferable Interpretation wrt IDB-Predicates):**

Let \( \text{PREDs} = \text{EDB} \cup \text{IDB} \). A Herbrand interpretation \( I_1 \) is preferable (or equal) to a Herbrand interpretation \( I_2 \) (\( I_1 \preceq_{(\text{EDB},\text{IDB})} I_2 \)) iff

- \( I_1 \cap \text{BASE}(\text{EDB}) = I_2 \cap \text{BASE}(\text{EDB}) \).
- \( I_1 \cap \text{BASE}(\text{IDB}) \subseteq I_2 \cap \text{BASE}(\text{IDB}) \).

Also in variable circumscription, there is the possibility to treat some predicates as fixed while minimizing other predicates. We will need this again in Chapter 4, where we are interested in the minimal models for fixed interpretations of the negation.
predicates. They are treated there like built-in predicates, which is natural, since they are not defined by rules, but by the negation semantics.

Now if we consider only interpretations with the fixed extensions of the EDB-predicates (as given by the database $DB$), then the minimal model of the logic program $P$ defining the IDB-predicates is equal to the minimal model of $DB \cup P$.

**Proposition 2.2.18 (Minimal Model of a Deductive Database):**

Let $DDB = \langle EDB, IDB, BIND, DB, P \rangle$ be a deductive database. There is a unique $\preceq_{(EDB,IDB)}$-minimal model $I$ of $P$ which satisfies $I \cap BASE(EDB) = DB$, namely the minimal model of $DDB := DB \cup P$.

So both views lead to the same result and we can choose whatever is more convenient.

**Direct Consequences**

The minimal model can be constructed by iteratively applying the rules of the program. This is the basis of bottom-up query evaluation and also shows that the notion of the minimal model is indeed a very natural one. The direct consequence operator $T_P$ constructs for a given interpretation $I$ the interpretation $T_P(I)$, which consists of the facts derived by the rules in $P$ from the facts in $I$. So whenever a rule body is satisfied in $I$, we make the rule head satisfied in $T_P(I)$.

**Definition 2.2.19 (Direct Consequence Operator):**

Given a logic program $P$, the direct consequence operator $T_P$ is the mapping from Herbrand interpretations to Herbrand interpretations defined by

$$T_P(I) := \{ F \in BASE \mid \text{there is a rule } A \leftarrow B_1 \land \cdots \land B_m \text{ in } P \text{ and there is a ground substitution } \theta \text{ for this rule such that } A\theta = F \text{ and } B_i\theta \in I \text{ for } i = 1, \ldots, m \}. $$

Note that it does not necessarily hold that $I \subseteq T_P(I)$. However, we will later see that during the fixpoint construction this condition is in fact satisfied.

We can envisage the bottom-up rule application with $T_P$ as follows:

$$(a, b) \quad \in T_P(I)$$

$$p(a, X) \leftarrow q(X, Y) \land r(Y, a). \quad \in P$$

$$q(b, c) \quad \in I$$

$$r(c, a) \quad \in I$$

Here the substitution $\theta$ is $\theta = \langle X/b, Y/c \rangle$.

Now bottom-up evaluation starts with the empty extensions of all predicates and applies the $T_P$-operator as long as new tuples can be derived. This process may not terminate, but it is still possible to show that the “limit” of this construction is the
minimal model. It is well known that this can be done in an elegant way by applying lattice theory (see, e.g., [Llo87]).

The set of all Herbrand interpretations $I \subseteq BASE$ together with the preference relation $\preceq$ forms a complete lattice (which is nothing else than the well-known powerset lattice). If we prefer to keep the interpretations of the EDB-predicates fixed, we can also consider the complete sublattice of the interpretations $I$ with $I \cap BASE(EDB) = DB$. In both cases, the least upper bound of some set of interpretations is their union, and the greatest lower bound is their intersection.

The direct consequence operator $T_P$ has three important properties:

- **First**, $T_P$ is monotonic, i.e. $I_1 \preceq I_2 \implies T_P(I_1) \preceq T_P(I_2)$. This is obvious, since when we have more facts which can be inserted for body literals, we can derive at least all previously derivable facts. (Note that monotonicity is not $T_P(I)_!$)

- **Second**, an even stronger property is that $T_P$ is continuous, which means as usual that limit construction and function application can be interchanged, where in lattice theory the least upper bound is some form of limit (and the property is required only for so called “directed sets” of interpretations, which contain an upper bound of every finite subset). The proof is also easy and based on the fact that if we can derive $A \theta$ from the least upper bound (i.e. the union) of some set of interpretations, we can derive it also from any upper bound of the interpretations $\{I_1, \ldots, I_m\}$ containing $\{B_1 \theta, \ldots, B_m \theta\}$.

- **Third**, $I = P \iff T_P(I) \preceq I$. This is easy to see: If $F \in T_P(I)$, this means that there is a rule instance which allows to derive $F$ in $I$, and this rule instance would be violated if $F$ were not already contained in $I$.

Now a theorem of lattice theory says that a monotonic mapping $T_P$ in a complete lattice has a fixpoint, i.e. an argument $I$ with $T_P(I) = I$. Furthermore, among the many possible fixpoints there is also a least fixpoint lfp($T_P$) satisfying lfp($T_P$) $\preceq I$ for any other fixpoint $I$. This fixpoint can be constructed the greatest lower bound of all $I$ with $T_P(I) \preceq I$, i.e. of all models $I$. By the third property above, this means that the least fixpoint of $T_P$ is a lower bound of all models of $P$, i.e. lfp($T_P$) $\preceq I$ for all models $I$ of $P$. Since the least fixpoint is itself a model, we obviously have that it is the minimal model:

**Proposition 2.2.20 (Minimal Model as Least Fixpoint):**

The least fixpoint of $T_P$ exists and is the minimal model of $P$.

Now let us return to the computation of the minimal model. We already mentioned that bottom-up evaluation iteratively applies the direct consequence operator until a fixpoint is reached. It might be possible that this computation continues forever, however the “limit”, i.e. the least upper bound, is the least fixpoint of $T_P$, i.e. the minimal model. So we get better and better “approximations” of the minimal model, and any specific fact in which we are interested is derived after finite time.

That this computation really leads to the least fixpoint of $T_P$ is again a theorem of lattice theory, and holds for arbitrary continuous mappings:
2.2. SYNTAX AND SEMANTICS OF DATALOG

Proposition 2.2.21 (Computation of Minimal Model):
The minimal model of $P$ is the least upper bound of the sequence $T_P \uparrow i$, $i \in \mathbb{N}$, where $T_P \uparrow 0 := I_\bot$ and $T_P \uparrow (i + 1) := T_P(T_P \uparrow i)$.

The direct consequence operator for the well-founded semantics introduced in Chapter 4 is only monotonic and not continuous. However, it can still be seen as the “limit” of an iterative construction if we allow higher ordinal powers of $T_P$ (see, e.g., [Llo87]). For instance, it might be necessary to apply $T_P$ infinitely often before we can derive some negative fact.

Range-Restriction

The class of Datalog-programs we defined above is in fact too wide for our purposes (effective and efficient computation), and we will now introduce important restrictions.

It is quite usual in deductive databases to require that every variable in the head-literal is “bound” in a body literal. This is called in the literature “safety”, “range-restriction”, and “allowedness”. For instance, consider the rule

$\text{likes}(X, Y) \leftarrow \text{happy}(X)$.

There are several problems with this rule:

- First, $\text{DOM}$ is usually infinite, so the extension of the IDB-predicate $\text{likes}$ is also infinite, and we cannot represent it by an explicit enumeration of its tuples.
- Second, since $\text{DOM}$ is only a sufficiently large namespace built into the system, it contains many identifiers which have no meaning in the specific application. So it is very bad if the answer contains constants outside the current “active domain” (values occurring in the database or the program).
- Third, the answer of course depends on the exact set $\text{DOM}$, which might be different even between different versions of the same system.

In [Con89], it has been shown that these three problems are equivalent (under reasonable assumptions).

While it is possible to allow the above rule and to represent the result by tuples with variables in them, even more difficult problems occur with built-in predicates. For instance, consider the rule

$\text{large}_\text{enough}(X) \leftarrow X > 1000$.

Although the variable $X$ occurs in a body literal, it is not bound to a finite set of values. So the IDB-predicate $\text{large}_\text{enough}$ has again an infinite extension, but it cannot be represented simply by a variable. We must remember the condition $X > 1000$ together with the “tuple” $(X)$, for instance in order to detect that the rule

$\text{acceptable}(X) \leftarrow \text{large}_\text{enough}(X) \land X < 800$

cannot be applied. In the field of “constraint logic programming” (CLP), there are systems supporting such rules. A standard Prolog system would simply give a runtime error (“instantiation fault”).
Example 2.2.22:

It is undecidable whether diophantic equations have a solution, i.e. there is no algorithm to decide whether

\[ 0 = \sum_{i_1=0}^{m_1} \cdots \sum_{i_n=0}^{m_n} a_{i_1,\ldots,i_n} X_1^{i_1} \cdots X_n^{i_n} \]

has an integer solution \((X_1, \ldots, X_n)\). If we have built-in predicates for sum and product, such a diophantic equation can easily be coded in a single non-recursive Datalog-rule. For instance, assume that \(a_{1,0} = 3\) and \(a_{1,2} = 5\), and all other \(a_{i,j} = 0\). This is of course only an example to show how to encode Diophantic equations in Datalog, and is in fact a very simple case \((X_1 = X_2 = 0\) is a solution). But we can encode any diophantic equation in this way:

\[ p(ok) \leftarrow Y = 3 \ast X_1 \land Z_1 = 5 \ast X_1 \land Z_2 = X_2 \ast X_2 \land Z = Z_2 \ast Z_3 \land 0 = Y + Z. \]

The extension of \(p\) is empty in the minimal model if the diophantic equation has no solution, and contains the single tuple \((ok)\) otherwise. So if such rules were allowed, even a single rule application would not be effectively computable. The extension is always finite, so in this sense we do not have a termination problem, but we cannot effectively apply the rule. \(\Box\)

So we now define an important restriction for Datalog-rules called “range-restriction”, which ensures that the above mentioned problems do not occur.

It looks only locally at each rule. In itself, it does not ensure the finiteness of the extensions of the IDB-predicates, because this is a global property of the program. For instance, the following rules are acceptable in isolation, but together they compute infinite relations:

\[
\begin{align*}
even(0). \\
even(X) & \leftarrow \odd(Y) \land X = Y + 1. \\
odd(X) & \leftarrow \even(Y) \land X = Y + 1.
\end{align*}
\]

The bottom-up computation of this program does not terminate, but it is possible to apply the rules iteratively and compute “better and better approximations” of the predicate extensions.

The range-restriction requires that every variable is bound to a finite domain and every body literal is actually evaluable with the procedures we have (as specified in \(BIND\)):

**Definition 2.2.23 (Range-Restriction):**

Let a binding pattern specification \(BIND\) for \(PREDS\) be given.

- We define the set of input variables of a literal \(L = p(t_1, \ldots, t_n)\) given a binding pattern \(\beta\) for \(p/n\) as follows:

\[
\text{input}(L, \beta) := \{ X \in VARS \mid \text{there is } 1 \leq i \leq n \text{ with } X = t_i \text{ and } \beta(i) = b \}.
\]
A rule $A \leftarrow B_1 \land \cdots \land B_m$ is called range-restricted for a binding pattern $\beta$ iff there is a sequence $i_1, \ldots, i_m$ of the body literals (i.e. $\{i_1, \ldots, i_m\} = \{1, \ldots, m\}$) such that for $j = 1, \ldots, m$ there is $\beta_j \in \text{BIND}(\text{pred}(B_{i_j}))$ satisfying

$$\text{input}(B_{i_j}, \beta_j) \subseteq \text{input}(A, \beta) \cup \text{vars}(B_{i_1} \land \cdots \land B_{i_{j-1}})$$

and furthermore it holds that

$$\text{vars}(A) \subseteq \text{vars}(B_1 \land \cdots \land B_m).$$

A program $P$ is range-restricted iff every rule $A \leftarrow B_1 \land \cdots \land B_m$ in $P$ is range-restricted for every $\beta \in \text{BIND}(\text{pred}(A))$.

A program $P$ is strictly range-restricted iff every rule $A \leftarrow B_1 \land \cdots \land B_m$ in $P$ is range-restricted for the adornment $\text{ff} \ldots \text{ff}$ (all arguments free).

Naive bottom-up evaluation is only applicable for strictly range-restricted programs since it makes no use of the bound arguments of the head literal. However, the “magic set” transformation (see Chapter 3) transforms every range-restricted program into a strictly range-restricted program.

Let us now clarify that the problems mentioned above do not occur for range-restricted programs. First, we need the notion of the active domain of a program.

**Definition 2.2.24 (Active Domain):**

The set of constants occurring in a term/literal/goal/rule is defined as follows:

- $\text{dom}(t) := \{t\}$ if $t \in \text{DOM}$, and $\text{dom}(t) := \emptyset$ if $t \in \text{VARS}$.
- $\text{dom}(p(t_1, \ldots, t_n)) := \text{dom}(t_1) \cup \cdots \cup \text{dom}(t_n)$.
- $\text{dom}(L_1 \land \cdots \land L_m) := \text{dom}(L_1) \cup \cdots \cup \text{dom}(L_m)$.
- $\text{dom}(A \leftarrow L_1 \land \cdots \land L_m) := \text{dom}(A) \cup \text{dom}(L_1) \cup \cdots \cup \text{dom}(L_m)$.

The active domain $\text{dom}(P)$ of a program $P$ consists of all constants occurring in $P$, i.e. $\text{dom}(P) := \bigcup_{\phi \in P} \text{dom}(\phi)$.

First, the minimal model (and all its approximations $T_P \uparrow i$) contain only values from the active domain:

**Proposition 2.2.25 (Domain Independence):**

Let $P$ be strictly range-restricted and $I_{\text{min}}$ be its minimal Herbrand model. Then $I_{\text{min}}[p/n] \subseteq (\text{dom}(P))^n$ for every $p/n \in \text{PREDS}$.

Second, the active domain is not only sufficient to represent the minimal model, but also during its computation, we never need to instantiate any variable to a value outside the active domain. This is formalized via the “ground instantiation” of the program, a notion which will be very useful in Chapter 4:
Definition 2.2.26 (Ground Instantiation of a Program):
The ground instantiation of a program $P$ is

$$\text{ground}(P) := \{ (A \leftarrow B_1 \land \cdots \land B_m)\theta \mid A \in B_1 \land \cdots \land B_m \in P \text{ and }$$
$$\theta \text{ is a ground substitution for this rule with } \theta(X) \in \text{dom}(P)$$
$$\text{for every } X \in \text{vars}(A \in B_1 \land \cdots \land B_m) \}.$$  

Proposition 2.2.27 (Equivalence of Ground Instantiation):
For every strictly range-restricted program $P$, the minimal model of the ground instantiation $\text{ground}(P)$ is equal to the minimal model of $P$.

Finally, there are no problems with computing and representing the extensions of the IDB-predicates during the bottom-up evaluation:

Proposition 2.2.28 (Computability of Fixpoint Approximations):
Let $DDB = \langle EDB, IDB, BIND, DB, P \rangle$ be a deductive database where $DB$ is allowed wrt $BIND$ and $P$ is strictly range-restricted and finite.

- Then for every $p/n \in IDB$ and every $i \in \mathbb{N}$ the extension $(T_{DDB} \uparrow i)[p/n]$ is finite.
- Suppose that for every $q/m \in EDB$ and every $\beta \in BIND(q/m)$ there is a terminating procedure which computes all $(c_1, \ldots, c_m) \in DB[q/m]$ given the $c_i$ with $\beta(i) = b$. Then for every $p/n \in IDB$ and $i \in \mathbb{N}$ there is a terminating procedure which computes $(T_{DDB} \uparrow i)[p/n]$.

Queries and Answers
The last thing to clarify in this section is: “What is a query?” After all, the main purpose of a database is to answer queries. There are three equally powerful definitions of the notion of a query, and depending on the context, each of them can be more convenient than the others.

- In Prolog and most deductive database systems, a query is a goal $L_1 \land \cdots \land L_m$ which asks for all substitutions $\theta$ such that $(L_1 \land \cdots \land L_m)\theta$ is true in the minimal Herbrand model. Allowing goals as queries is very natural in a system based on SLD-resolution, since there goals are anyway produced as “subqueries”. Furthermore, (nonrecursive) rules can then be seen as giving a name to a query result.
- Sometimes it simplifies the definitions (e.g. of the “magic set” transformation), to allow only a single literal $L$ as a query. This is also very natural when we create a procedure for each allowed binding pattern for a predicate. Of course, the query $L$ asks for all substitutions $\theta$ such that $L\theta$ is contained in the minimal Herbrand model.
- The most rudimentary form of a query is to specify a predicate $p/n$ and to ask for its extension in the minimal Herbrand model. This is natural in the context
of naive bottom-up evaluation, where we anyway compute the complete minimal model.

The three classes of queries are equally powerful, since we allow to define new IDB-predicates by means of rules (so a query is really a logic program together with a goal/literal/predicate). If, for instance, we want a goal $L_1 \land \cdots \land L_m$ as a query, but our system allows only single literals or predicates, we define a new predicate answer by the rule

$$\text{answer}(X_1, \ldots, X_n) \leftarrow L_1 \land \cdots \land L_m,$$

where $\{X_1, \ldots, X_n\} := \text{vars}(L_1 \land \cdots \land L_m)$. Then the query answer$(X_1, \ldots, X_n)$ or answer$/n$ gives obviously the same result as the original query.

Often, we want to use the same program for slightly different queries, namely containing different input values. Since the program should not be “recompiled” in this case, it is good that a literal or goal can also contain constants. However, the same effect can be reached by using the database for storing the input values. In fact, the parameter passing from the query to the precompiled program is often implemented via such an “input relation”. And the database is of course anyway an input to the query. Sometimes a fixed number of input arguments is not sufficient, and then we also need temporary EDB-relations to hold the input values.

In contrast to Prolog, the deductive database system Aditi also allows to project the query result on a subset of the output values as, e.g., in $X, Z; p(X, Y), q(Y, Z)$. This does not increase the expressive power since in the above definition of answer, we of course do not need to include all variables in the head literal. However, it is a nice feature, since it allows to specify the important class of “select-project-join” queries without defining new IDB-predicates.

**Definition 2.2.29 (Consequences of a Logic Program):**

A ground literal $L$ is a consequence of a logic program $P$ (written $P \models L$) iff $I_{\text{min}} \models L$ for the minimal model $I_{\text{min}}$ of $P$, or, equivalently, $I \models L$ for every model $I$ of $P$.

**Definition 2.2.30 (Query):**

- A query is a literal $L$.
- A query $p(t_1, \ldots, t_n)$ is range-restricted wrt BIND iff there is $\beta \in \text{BIND}(p/n)$ such that for $i = 1, \ldots, n$: $\beta(i) = b \implies t_i \in \text{DOM}$.
- Given a logic program $P$ or deductive database DDB, a correct answer for a query $L$ is a ground substitution $\theta$ for $\text{vars}(L)$ such that $P \models L\theta$ resp. $\overline{\text{DDB}} \models L\theta$.

Of course, from a deductive database we expect that it prints all correct answers (without duplicates). In contrast, Prolog prints always one answer and asks whether more answers are wanted.
CHAPTER 2. HORN-CLAUSE DATALOG

2.3 Bottom-Up Query Evaluation

Given a database $DB$ for the EDB-predicates and a logic program $P$ for the IDB-predicates, the goal of bottom-up evaluation is to compute the extensions of the IDB-predicates in the minimal model. This is basically done by iteratively applying the immediate consequence operator $T_P$ until nothing changes.

In this section we prefer the deductive database view, since we obviously can work only with finite programs, while the database can contain infinite predicates like $<$ and $\text{sum}$. So let a deductive database $DDB = \langle EDB, IDB, BIND, DB, P \rangle$ be given, where $DB$ is allowed wrt $BIND$, and $P$ is strictly range-restricted wrt $BIND$.

In this section we assume that we need the complete minimal model in order to answer the query. This is not very typical for programs written by the user, but the “magic set” transformation introduced in Chapter 3 produces such programs. Another application might be that we need to materialize the extensions of the IDB-predicates in order to use later a query language like SQL for the minimal model.

The basic algorithm looks as follows:

\begin{verbatim}
Input: Deductive database as above;
Output: Extensions of IDB-predicates in the minimal Herbrand model;

$I := \emptyset$;
$I_{\text{new}} := T_P(DB);$
while $I_{\text{new}} \neq I$ do
  $I := I_{\text{new}}$;
  $I_{\text{new}} := T_P(DB \cup I);$ 
od

print $I$; /* Alternatively: print $I[\text{answer/n}]$; */
\end{verbatim}

Note that here $I$ and $I_{\text{new}}$ consist only of IDB-facts, therefore we add $DB$ before we apply the $T_P$ operator, so that also body literals referring to EDB-predicates can be evaluated.

Of course, we must look more into the details and improve many things. This is the subject of this section.

Naive Evaluation and Rule Order

Our goal should be to compute as few as possible facts twice. However, in the above algorithm every iteration repeats the computation of all facts derived in previous iterations.

We will now order the rules in such a way that it suffices to apply the “non-recursive” rules only once. In order to formalize this, we need the notion of the predicate dependency graph:


2.3. BOTTOM-UP QUERY EVALUATION

Definition 2.3.1 (Predicate Dependency Graph):
The predicate dependency graph of a logic program $P \subseteq RULES(PREDS)$ is the directed graph $G(P) = (V, E)$ with

- $V := PREDS$, i.e. the vertices are the IDB- and EDB-predicates, and
- $(q/m, p/n) \in E$ iff $P$ contains a rule of the form

$$p(t_1, \ldots, t_n) \leftarrow B_1 \land \cdots \land B_{i-1} \land q(u_1, \ldots, u_m) \land B_{i+1} \land \cdots \land B_k,$$

i.e. a rule about $p/n$, such that $q/m$ is the predicate of a body literal.

So the edges have the same direction as the implication arrow “$\leftarrow$” of the rules. Of course, the other direction could be used equally well.

From the predicate dependency graph, we can easily see which predicate extensions are needed in order to compute some other predicate, and which predicates are recursive or mutually recursive such that an iteration is unavoidable:

Definition 2.3.2 (Predicate-Dependencies, Recursion):
Let a logic program $P \subseteq RULES(PREDS)$ be given, and let $G(P)$ be its predicate dependency graph.

- A predicate $p/n \in PREDS$ depends directly on a predicate $q/m \in PREDS$ iff there is an edge from $q/m$ to $p/n$ in $G(P)$.
- A predicate $p/n \in PREDS$ depends on a predicate $q/m \in PREDS$ iff there is a non-empty path from $q/m$ to $p/n$ in $G(P)$.
- A predicate $p/n$ is recursive iff it depends on itself. Otherwise it is nonrecursive.
- The nodes $\{p_1/n_1, \ldots, p_k/n_k\}$ of a strongly connected component of $G(P)$ are called a recursive clique.
- Two predicates $p/n$ and $q/m$ are mutually recursive iff they are elements of the same recursive clique.

The predicate dependency graph can contain cycles. However, in order to determine an evaluation order, we need an acyclic graph. The standard solution is to merge strongly connected components into a single node:

Definition 2.3.3 (Reduced Predicate Dependency Graph):
The reduced predicate dependency graph of a logic program $P$ is the directed acyclic graph $\hat{G}(P) = (\hat{V}, \hat{E})$ with

- $\hat{V} := \{C \subseteq PREDS \mid C$ is a recursive clique of $P\}$
  $\cup \{p/n \mid p/n \in PREDS$ is nonrecursive\}.
- $(C, C') \in \hat{E} :\iff C \neq C'$ and there are $q/m \in C$ and $p/n \in C'$ such that $p/n$ depends directly on $q/m$.

Now we can sort the reduced predicate dependency graph topologically and get an evaluation order for the predicates. While paths in a graph can easily be defined in Datalog, it seems quite difficult and inefficient to specify topological sorting in
current deductive database systems. We believe that the language of a deductive
database should be sufficiently expressive so that it can be used as a basis for its own
implementation. So there is certainly room for further developments here.

**Definition 2.3.4 (Predicate Evaluation Sequence):**

Let $P$ be a logic program and $\hat{G}(P) = (\hat{V}, \hat{E})$ be its reduced predicate dependency
graph. Any sequence $C_1, \ldots, C_m$ of the nodes $V$ satisfying

$$\text{for all } i, j \in \{1, \ldots, m\}: (C_i, C_j) \in \hat{E} \implies i < j$$

(i.e. there are only “forward edges”) is called a predicate evaluation sequence for $P$.

Now what we really wish to evaluate are not the predicates, but the rules of the
program. It is possible to construct also a dependency graph for the rules, or a
combined dependency graph for rules and predicates. But these graphs are often
much larger than the predicate dependency graph and it is possible to derive the
needed information about the rules from the corresponding notions of the predicate
dependency graph:

**Definition 2.3.5 (Recursive Rule):**

Let a logic program $P \subseteq RULES(PREDS)$ be given.

- A rule $A \leftarrow B_1 \land \cdots \land B_m$ is recursive iff there is $i \in \{1, \ldots, m\}$ such that
  $\text{pred}(B_i)$ depends on $\text{pred}(A)$. Otherwise it is called nonrecursive.

- We make use of the following notation to select the recursive/nonrecursive rules
  about a predicate in some set $C \subseteq PREDS$:

\[
\text{rek}(P, C) := \{ A \leftarrow B_1 \land \cdots \land B_m \in P \mid \text{the rule is recursive and } \text{pred}(A) \in C \}.
\]

\[
\text{nrek}(P, C) := \{ A \leftarrow B_1 \land \cdots \land B_m \in P \mid \text{the rule is nonrecursive and } \text{pred}(A) \in C \}.
\]

So we can now refine the basic bottom-up evaluation algorithm by using a predicate
evaluation sequence in order to

- apply non-recursive rules only once, and to
- iterate recursive rules only locally within one recursive clique.

This leads to the algorithm shown in Figure 2.1. Of course, further optimizations are possible:

First, we can delete the extension of an IDB-predicate $p/n$ from $I$ as soon as
we have evaluated all rules which contain $p/n$ in the body (and have printed the
extension of $p/n$ if this is required). Usually, we will try to buffer the extensions
of the IDB-predicates in main memory and put them onto external storage only if
the available buffer space is insufficient. So not all predicate evaluation sequences
are equally good, since some might need more temporary space for computed and
still needed IDB-facts than others, which influences the number of disk accesses and
thus the query evaluation time. This is somehow related to register allocation during
2.3. BOTTOM-UP QUERY EVALUATION

Input: Deductive database as above;
Output: Extensions of IDB-predicates in the minimal Herbrand model;

Let $C_1, \ldots, C_m$ be a predicate evaluation sequence for $P$;
$I := \emptyset$;
for $i := 1$ to $m$ do
  if $C_i$ consists of a nonrecursive predicate $p/n$ then
    $I := I \cup T_{nrek}(P, C_i)(DB \cup I)$;
  else /* $C_i$ is a recursive clique */
    $I_{\text{new}} := T_{nrek}(P, C_i)(DB \cup I)$;
    while $I_{\text{new}} \neq \emptyset$ do
      $I := I \cup I_{\text{new}}$
      $I_{\text{new}} := (T_{rek}(P, C_i)(DB \cup I)) - I$
    od
  fi
od

print $I$; /* Alternatively: print $I[answer/n]$; */

Figure 2.1: Bottom-Up Evaluation Algorithm with Rule Order

the compilation of arithmetic expressions (or basic blocks). There it is also a costly operation to swap a value which we have in a register to main memory. However, in contrast to values of arithmetic expressions, IDB-relations can have very different sizes, and it is not simply the number of IDB-relations we need to keep in the buffer. By the way, it is also difficult to guess at compile time how big the IDB-relations will be and in the case of recursions the size might vary a lot between different iterations. Therefore, the Glue-Nail system uses a runtime optimizer [DMP94].

Above, we have talked about complete IDB-relations as units. However, every tuple can be deleted from main memory as soon as it has been used for the last time. This leads to a view of the predicates as coroutines, which immediately pass the produced tuples to the coroutines which consume them. For instance, in query evaluation of standard relational databases it is also usual to combine operations like selection and projection, and to create a temporary relation only for those intermediate results which we need to pass several times (e.g. for sorting or join computation). A method for compiling Datalog into coroutines was described in [Ben94].

Related to this is the question of duplicate elimination. It is a very costly operation, and furthermore it forces us to keep produced tuples in the buffer as long as there is the possibility that we can compute them a second time. So we should try to analyze the program in order to find out where duplicate elimination is needed and where not. For this purpose, acyclicity-, exclusion-, and key-constrains on the EDB-relations would be very useful. Also, sometimes there is a tradeoff where duplicate elimination would filter out some tuples but might not be worth the effort. However, the set difference in the processing of recursive predicates in Figure 2.1 is
really needed since otherwise we could not guarantee termination.

**Semiaive or Differential Evaluation**

Our goal was to compute as few as possible facts twice. More precisely, we want to consider every rule instance $A\theta \leftarrow B_1\theta \land \cdots \land B_m\theta$ only once. Of course, it is possible that the same fact can be derived with two different rule instances, and in this case we do not try to exclude that we compute it twice.

For nonrecursive rules, we have reached our goal to consider each of their instances only once. But for recursive rules, each iteration still repeats the computations done by the previous iterations. It is the goal of seminaive or differential evaluation to avoid this. As a simple example, consider the following program to compute pairs of connected nodes in a directed graph:

\[
\text{path}(X, Y) \leftarrow \text{edge}(X, Y).
\]
\[
\text{path}(X, Z) \leftarrow \text{edge}(X, Y) \land \text{path}(Y, Z).
\]

Here, it suffices of course to insert only the tuples produced in the last iteration (i.e. contained in $I_{\text{new}}[\text{path}]$) for the recursive body literal $\text{path}(Y, Z)$. The next version of our bottom-up evaluation algorithm will ensure that these tuples are contained in a predicate called $\text{path\_diff}$. So we rewrite this rule to

\[
\text{path\_new}(X, Z) \leftarrow \text{edge}(X, Y) \land \text{path\_diff}(Y, Z).
\]

Semi-naive evaluation can neither be explained as a pure rewriting method nor by using the $T_P$-operator for the old rules. But it can be understood as rewriting with the possibility to refer to new and old tuples by means of special system-defined predicates. Here, the system will do the following after every iteration:

\[
\text{path\_diff} := \text{path\_new} - \text{path}.
\]
\[
\text{path} := \text{path} \cup \text{path\_diff}.
\]
\[
\text{path\_new} := \emptyset.
\]

Of course, the system does not need to really keep three different versions of the $\text{path}$-relation if it has a special treatment of such “layered relations” (or “relations with deltas”).

With two recursive body literals it is important that we need also combinations of old tuples and tuples derived in the last iteration:
**Input:** Recursive Rule $A \leftarrow B_1 \land \cdots \land B_m$ and Predicate-Dependencies;

**Output:** Rules for Seminaive Evaluation;

Let $p(t_1, \ldots, t_n) := A$;
Let $q_i(u_{i,1}, \ldots, u_{i,k_i}) := B_i$ (for $i = 1, \ldots, m$);

for $i := 1$ to $m$ do
  if $q_i/k_i$ is recursive then
    print $p_{new}(t_1, \ldots, t_n)$;
    print “−”; 
    for $j := 1$ to $i - 1$ do
      if $j \neq 1$ then print “∧” fi;
      if $q_j/k_j$ is recursive then
        print $q_{j-old}(u_{j,1}, \ldots, u_{j,k_j})$;
      else
        print $q_j(u_{j,1}, \ldots, u_{j,k_j})$;
      fi
    od;
  if $i \neq 1$ then print “∧” fi;
  print $q_{i-diff}(u_{i,1}, \ldots, u_{i,k_i})$;
  for $j := i + 1$ to $m$ do
    print “∧”;
    print $q_j(u_{j,1}, \ldots, u_{j,k_j})$;
  od;
  print “.”;
od

Figure 2.2: Seminaive Rewriting

Note that we can avoid combinations between old tuples, and this is important since there are usually much more old tuples than tuples from the last iteration (the old tuples have been produced during possibly many iterations).

In general, we need to ensure that a tuple from the last iteration is inserted for at least one body literal. We have four versions of every recursive predicate $p$:

- $p_{old}$ contains the tuples which were already derived before the last iteration.
- $p_{diff}$ contains the tuples which were newly derived in the last iteration. So $p_{old} \cap p_{diff} = \emptyset$.
- $p$ contains all tuples which were derived up to and including the last iteration, i.e. $p = p_{old} \cup p_{diff}$.
- $p_{new}$ contains the tuples which are derived in this iteration.

Figure 2.2 shows the algorithm for rewriting the rules in order to ensure that at least one tuple from $p_{diff}$ is used.
Example 2.3.6:
Consider for instance the rule

\[ p(X) \leftarrow q(X) \land r(X) \land s(X), \]

where all predicates are mutually recursive. This rule is rewritten into:

\[
\begin{align*}
p_{\text{new}}(X) & \leftarrow q_{\text{diff}}(X) \land r(X) \land s(X). \\
p_{\text{new}}(X) & \leftarrow q_{\text{old}}(X) \land r_{\text{diff}}(X) \land s(X). \\
p_{\text{new}}(X) & \leftarrow q_{\text{old}}(X) \land r_{\text{old}}(X) \land s_{\text{diff}}(X).
\end{align*}
\]

Note that for a rule with \( n \) recursive body literals we need only \( n \) rules, although there are \( 2^n - 1 \) possible combinations of “old” and “diff”. The trick is that the original version of every predicate contains the union of its “old” and “diff” parts, for instance \( r = r_{\text{old}} \cup r_{\text{diff}} \).

The rewritten rules are to be executed by the bottom-up evaluation algorithm of Figure 2.3, where \( \text{semi} \) denotes the seminaive rewriting.

Since we have different versions of the recursive predicates, it now suffices to work with a single interpretation \( I \). Interpretations correspond to relational databases, so this is an important refinement from the practical viewpoint. All applications of the \( T_P \)-operator have the form \( I := I \cup T_P(DB \cup I) \), i.e. are insertions of tuples into the database. Note that by the rewriting all rules have been made “nonrecursive”, so the resulting tuples can never interfere with the tuples used in the rules. Thus, the resulting tuples can immediately be inserted into the database, which is an important simplification.

Of course, it would be good to use these tuples also immediately for further derivations. For instance, consider the following program:

\[
\begin{align*}
path_1(X, Y) & \leftarrow edge(X, Y). \\
path_2(X, Y) & \leftarrow edge(X, Y) \land path_3(Y, Z). \\
path_2(X, Y) & \leftarrow path_1(X, Y). \\
path_3(X, Y) & \leftarrow path_2(X, Y).
\end{align*}
\]

Here it takes always three iterations between two successful applications of the recursive rule for \( \text{path}_1 \). In all other iterations, this rule does nothing because there are no new tuples for \( \text{path}_1 \). It is possible to use the tuples produced by one rule immediately in the next rule, however this complicates the bookkeeping for seminaive evaluation — we need to remember for each rule which tuples already have been inserted for the body literals. Furthermore, the order of the rules within a recursive clique then becomes relevant — for instance in the example, we would gain nothing if we applied the rules in the opposite order. So we must try to find an order which preserves the cycles in the rule dependencies as far as possible [RSS94].

At least in the above example it seems simpler to check before each rule application whether it can possibly produce new tuples. Such a check can be done very efficiently and we can avoid the overhead of trying to apply the rule.
2.3. BOTTOM-UP QUERY EVALUATION

Input: Deductive database as above;

Output: Extensions of IDB-predicates in the minimal Herbrand model;

Let \( C_1, \ldots, C_m \) be a predicate evaluation sequence for \( P \);
\[ I := \emptyset; \]
for \( i := 1 \) to \( m \) do

\[ \text{if } C_i \text{ consists of a nonrecursive predicate } p/n \text{ then} \]
\[ I := I \cup T_{\text{reck}}(P, C_i)(DB \cup I); \]

\[ \text{else } /* C_i \text{ is a recursive clique */} \]
\[ I := I \cup T_{\text{reck}}(P, C_i)(DB \cup I); \]
\[ P_i := \text{semi}(\text{reck}(P, C_i)); \]
foreach \( p/n \in C_i \) do

\[ I[p\_old/n] := \emptyset; \]
\[ I[p\_diff/n] := I[p/n]; \]
\[ I[p\_new/n] := \emptyset; \]
enddo;

while there is \( p/n \in C_i \) with \( I[p\_diff/n] \neq \emptyset \) do

\[ I := I \cup T_{P_i}(DB \cup I); \]
foreach \( p/n \in C_i \) do

\[ I[p\_old/n] := I[p\_old/n] \cup I[p\_diff/n]; \]
\[ I[p\_diff/n] := I[p\_new/n] - I[p/n]; \]
\[ I[p/n] := I[p/n] \cup I[p\_new/n]; \]
enddo;
enddo;
fi

do
print \( I \); /* Alternatively: print \( I[\text{answer/n}] \); */

Figure 2.3: Seminaive Bottom-Up Evaluation Algorithm
It is interesting to note that bottom-up evaluation can also be described as an iterative method to solve systems of algebraic equations [CGT90]. For instance, the program for computing pairs of connected nodes in a directed graph corresponds to the following equation:

\[ \text{path} = \text{edge} \cup \pi_{\$1, \$4}(\text{edge} \Join_{\$2=\$1} \text{path}) \].

Then the bottom-up method we have described above is formally identical to the Jacobi-method for solving systems of equations in numerical analysis, while the method to use derived tuples immediately corresponds to the Gauss-Seidel method.

Finally, we should note that there are special variants of seminaive evaluation if we are not interested in all derivable tuples, but only the in some sense “optimal” tuples. For instance, if the edges in the graph have lengths assigned to them, and we want to find the shortest paths, then we can immediately delete any connections which we know already to be suboptimal. For a discussion of this topic see [KKTG93].

How to Compute the Direct Consequences of Rules

Above, we have explained the overall control structure of bottom-up query evaluation. Continuing our successive refinement steps, we now need to explain how to implement the statements of the form \( I := I \cup T_p(DB \cup I) \).

Many systems are based on some variant of relational algebra, because there is a large body of knowledge how to implement relational algebra efficiently. It is possible to translate every rule into a relational algebra expression which yields the tuples derivable by this rule. For instance, consider the rule

\[ p(X, a) \leftarrow q(b, Y) \land r(Y, X). \]

We translate

- constants in body literals into selections (also variables which appear more than once in a body literal are treated in this way),
- common variables of different body literals into joins, and
- the head literal into a projection.

In the example, we get the relational algebra expression

\[ \pi_{\$4, a}(\sigma_{\$1=b}(q) \Join_{\$2=\$1} r) \].

The produced tuples can be immediately inserted into the relation \( p \), since our seminaive algorithm ensures that no predicate appears in a head and in a body of some program component.

Standard relational algebra has no concept of “built-in” predicates, so we have to extend it slightly. Of course, predicates like \(<\), which are called with all arguments bound, can be treated as generalized selection conditions. Predicates like \(\text{sum}\), which correspond to computable functions, can easily be integrated into projections. However, if we call for instance \(\text{cons}(\text{Head}, \text{Tail}, \text{List})\) with \(\text{List}\) bound, this acts as
both, a selection (List must be non-empty) and a construction of head and tail of the list. So this is of course nothing else than a join with an infinite relation, which is implemented like a "hash join": We loop over all tuples of the finite input relation and look up the corresponding tuple in the cons-relation for the given value of List. It makes no difference whether there is a hash table in case of a finite, explicitly stored relation, or whether the relation is only virtual, and the needed tuples are computed on the fly. In both cases we access it with a specific binding pattern.

Since the objects of relational algebra are finite relations, we should denote the join with an infinite relation in an asymmetric way, for instance $\alpha_{p(\xi_1, \ldots, \xi_n)}(R)$, where the following two conditions are satisfied:

- Let $m$ be the number of attributes of $R$. Then there is a $k \in \mathbb{N}_0$ such that
  \[ \{i_1, \ldots, i_n\} - \{1, \ldots, m\} = \{m+1, \ldots, m+k\}, \]
  i.e. new attributes are introduced consecutively.
- There is a binding pattern $\beta \in \text{BIND}(p/n)$ such that for $j = 1, \ldots, n$: If $\beta(j) = b$, then $i_j \leq n$.

The result of this operation consists of all tuples $(c_1, \ldots, c_{n+k}) \in \text{DOM}^{n+k}$ such that $(c_1, \ldots, c_m) \in R$ and $(c_1, \ldots, c_n) \in DB[p]$. Of course, instead of attribute numbers $\xi_j$ we can also allow constants. This is only a shorthand, since we can introduce the constants into the argument, e.g. as in $\alpha_{\xi_1 < \xi(n+1)}(R \times \{(20)\})$. Then a translation of a rule $p(t_1, \ldots, t_n) \leftarrow B_1, \ldots, B_m$ into this extended relational algebra becomes nearly trivial. Since the rule is strictly range-restricted, there is a sequence of the body literals such that each literal is evaluable given values for the variables occurring in the body literals to the left of it. Let us assume that the body literals are ordered already in such a way. Furthermore, let us assume that the variables have been numbered in the order of their first occurrence from left to right and have been called $\xi_1, \xi_2, \ldots$. Then the following expression computes the derived tuples:

$$\pi_{t_1, \ldots, t_n}(\alpha_{B_m}(\ldots(\alpha_{B_1}(\{()\})\ldots))).$$

Of course, in this way the evaluation order is very strictly determined, leaving not much space for optimizations. It is probably better to use the standard relational algebra operations where possible.

If we have bindings for some variables occurring in the head, the "magic" relation containing these bindings would replace the innermost $\{()\}$. Furthermore, it is usually important to start the evaluation with these bindings, at least if we want to simulate SLD-resolution (see Chapter 3).

It is also interesting to think a little of the interface of the algorithms for a predicate with given binding pattern. We have seen that we have an input relation for some of the bound arguments, and constants for others. However, it seems possible that there can also be independent input relations for different arguments, for instance in case of the rule

$$p(X) \leftarrow q_1(a, Y_1) \land q_2(b, Y_2) \land r(Y_1, Y_2, X).$$
Here the bindings for $Y_1$ and $Y_2$ are independent of each other and it would be good to avoid computing the cartesian product. This would be possible by using NF$^2$-relations for representing the bindings.

Let us finally mention that there are often connections between different rules, so it is a bit too simple to translate every single rule in isolation. For instance, the following pattern occurs quite often (usually coded in Prolog with a cut)$^2$:

\[
\begin{align*}
p(X, Y) &\leftarrow m_p^{bf}(X) \land q(X) \land r(X, Y). \\
p(X, Y) &\leftarrow m_p^{bf}(X) \land \text{not } q(X) \land s(X, Y).
\end{align*}
\]

Here it would be good to compute the join and the antijoin of $m_p^{bf}(X)$ and $q$ at the same time and pass the $X$-values either to $r$ or $s$.

**Bottom-Up Evaluation in Some System Prototypes**

Some of the early deductive database prototypes (including the NAIL! system) were using a standard relational database via an SQL-interface. This turned out to be quite inefficient for the following reasons (among others):

- Relational databases usually do not make the concept of a “temporal relation” available to the user (although they certainly have it internally): It is an important simplification that IDB-relations need not to be made persistent. In case of a sudden power failure or system crash they are no longer needed, and it is known that they will be deleted at the end of the computation.

- Furthermore, there are usually no deletions of single tuples from IDB-relations (let alone updates), there are only insertions. This knowledge could also be used to avoid certain overheads in the management of standard relations.

- The size of the IDB-relations is unknown at compilation time and can vary a lot between different iterations. This is a problem for the relational query optimizer.

- For seminaive evaluation, we need a “layered relation”, where we have special access paths to the tuples produced in the last iteration. The usual ways to simulate this with multiple relations or an “iteration counter” in every tuple are not very efficient.

- On a lower level, it is a very simple operation to check whether a tuple is already contained in a relation, and if not, to insert it and to set a flag. However, in SQL, this is difficult to express and leads to inefficiencies.

- Finally, there are all kinds of inefficiencies related to double parsing, double optimization, and different data formats.

So it is much better to use a lower level interface and to have access to the internals of the database software.

$^2$Negation will be introduced only in Chapter 4, but $m_p^{bf}(X) \land \text{not } q(X)$ can be understood as the set-difference of the two relations. The relation $m_p^{bf}$ is a “magic set” containing the given bindings for $X$, see Chapter 3.
Many systems translate Datalog in some intermediate language which contains imperative constructs and statements for database access. For instance, Aditi\textsuperscript{3} [VRK+94] translates the program

\begin{align*}
\text{path}(X, Y) & \leftarrow \text{edge}(X, Y). \\
\text{path}(X, Z) & \leftarrow \text{edge}(X, Y) \land \text{path}(Y, Z).
\end{align*}

into the code shown in Figure 2.4. Aditi constructs for every IDB-predicate \( p \) of arity \( n \) and binding pattern number \( i \) a procedure \( p_{n,i} \) with one input relation and one output relation. The input relation contains values for the bound arguments and the output relation must contain all tuples in the extension of \( p/n \) which agree with one of the tuples of the input relation for the bound arguments. The operations of Aditi RL have the following meaning:

- \texttt{settrel}: Initialization of a temporary relation.
- \texttt{setbrel}: Initialization of a relation with a B-tree index allowing to find duplicate tuples efficiently.
- \texttt{setprel}: Initialization of a relation variable with a pointer to a “permanent” EDB-relation.
- \texttt{btmerge}(\( R, S, T \)): This corresponds to \( T := S - R \) and \( R := R \cup S \), i.e. the tuples of \( S \) are inserted into \( R \), and if they were really new, they are also inserted into \( T \).
- \texttt{join}(\( R, S, A, X, T, B \)): This corresponds to \( T := \pi_B(R \bowtie S) \) (\( X \) is used only for arithmetical operations). Aditi RL uses the notation \#(\( i, j \)) to refer to attribute number \( j \) of relation number \( i \).
- \texttt{flatten}: This copies the tuples from a B-Tree into a more compact format.

Figure 2.5 shows the same example in the IGlue language of the Glue-Nail system [DMP94]. The most important type of IGlue-statement is the \_\texttt{FORALL}. It is possible to understand this as a “nested loop join”, however, it uses also hash-indexes for the relation access and might choose a different sequence of the “body literals”. A “!” in front of a relation means that the corresponding tuple may not be contained in it, “++” denotes insertion, “--” deletion, and “~~” means to delete all tuples (“clear”). “\_\texttt{LOCAL}”, “\_\texttt{EDB}” etc. denote the type of a relation.

Finally, there is also a system using a language based on coroutines, see [Ben94].

\textsuperscript{3}Beta-Test version from 1993
Figure 2.4: Aditi RL (Relational Language) Code for path-example
PROCEDURE path
  LOCALDECL changed/0
  FORALL(\~ LOCAL(changed))
  FORALL(EDB(edge(X,Y),
    ++ OUT(path(X,Y),
      ++ LOCAL(changed)))
  repeat:
    IF EXISTS(LOCAL(changed)) GOTO end
    FORALL(\~ LOCAL(changed))
      FORALL(EDB(edge(X,Y),
        OUT(path(Y,Z)),
        OUT(path(X,Z)),
        OUT(path(X,Z)),
        ++ LOCAL(changed)))
    GOTO repeat
  end:
  RETURN

Figure 2.5: IGlue Code for path-example
Chapter 3

Goal-Directed Bottom-Up Evaluation

Pure bottom-up evaluation as presented in Chapter 2 computes the complete minimal model of a Horn program. This is necessary for some applications, but often we only want answers for some specific query. Then top-down query evaluation can be more efficient, because it does only computations relevant to the given query. However, pure top-down query evaluation has also problems of its own, namely it can do the same work again and again, and maybe even not terminate.

Therefore, deductive database researchers tried to develop new query evaluation methods, which combine the advantages of top-down and bottom-up evaluation. In fact, a large percentage of the work in deductive databases was (and still is) devoted to query evaluation. It was a major step forward when in 1986, BANCILHON, MAIER, SAGIV, and ULLMAN developed the “magic set”-method [BMSU86]. A closely related method was independently developed by ROHMER, LESCOEUR, and KERISIT [RLK86]. The “magic set”-method is a program transformation which allows bottom-up evaluation to simulate top-down evaluation. We will define and analyze it in Section 3.1.

In contrast to many other query evaluation methods proposed at that time, the “magic set” method has been very successful and is today a standard component of most deductive database systems (e.g. CORAL [RSSS94] and Aditi [VRK+94]). An important reason for the success of the “magic set” technique was the claim that it is as least as efficient as the “top-down evaluation” known from logic programming. An often cited formalization of this has been proven by ULLMAN in a paper entitled “Bottom-Up beats Top-Down for Datalog” [Ull89a]. Logic programming had proven to be efficient enough for many practical applications, and with such a result it seemed that deductive databases could inherit this efficiency in addition to their own strengths in dealing with large sets of data. Of course, this was only an asymptotic result, but it seemed that the efficiency problems of deductive databases were only a question of better low-level implementation techniques.

While the general idea of this result is by now folklore, it is less well known that ULLMAN has used for comparison a top-down algorithm called QRG-T-resolution, and not the the standard top-down algorithm, namely Prolog’s SLD-resolution. And
in fact, the corresponding result for SLD-resolution is false. This was probably first explicitly noted in the literature by Ross [Ros91]. To be fair, already Ullman stated in a footnote of [Ull89a]:

“However, Prolog implementations usually use a form of tail recursion optimization that, for certain examples, such as the right-linear version of transitive closure, will avoid rippling answer tuples up the rule/goal tree, and thus can be faster than QRGT.”

But this is a little misleading, since the difference in performance can already be understood on the abstract level of SLD-trees; we do not have to look at the internal data structures of a Prolog system.

Let us consider the standard program for computing the transitive closure, mentioned in the above citation. We use it to compute the ancestor relation from given parents, but there are many other applications of transitive closures, and we believe that it is representative of quite a large class of tail-recurrences.

\[
\begin{align*}
\text{anc}(X, Y) & \leftarrow \text{parent}(X, Y). \\
\text{anc}(X, Z) & \leftarrow \text{parent}(X, Y) \land \text{anc}(Y, Z).
\end{align*}
\]

To make the example as simple as possible, let the parent-relation be one straight line of length \( n \). For example, we might know only the father within some ancient dynasty of kings:

\[
\text{parent} := \{(i - 1, i) \mid 1 \leq i \leq n\}.
\]

Represented as a directed graph, the parent relation looks as follows:

\[
\begin{array}{c}
0 \\
1 \\
\vdots \\
n
\end{array}
\quad \quad \quad \quad \quad \quad
\]

Let us for example pose the query \( \text{anc}(0, X) \), i.e. we ask for the ancestors of person number 0. This problem should obviously be solvable in linear time. The SLD-tree for this query looks as shown in Figure 3.1. It has \( 4n + 3 \), i.e. \( O(n) \) nodes. The application of the two rules is possible in constant time, whereas looking up a value in the parent-relation may take \( O(\log(n)) \)-time (assuming that there is an index over the first argument). So we can say that a reasonable implementation of SLD-resolution should have the running time \( O(n \log(n)) \).

In this particular example, the magic set transformation does not improve the program at all: We can derive \( m_{-\text{anc}}(i) \) corresponding to the query \( \text{anc}(i, X) \) for all \( 0 \leq i \leq n \). This is intuitively understandable, since when we want to determine the ancestors of person number 0, we have to determine recursively the ancestors of person number 1, and so on. But when we have \( m_{-\text{anc}}(i) \), we also get \( \text{anc}(i, j) \) for all \( 0 \leq i < j \leq n \) (i.e. the “answer facts” to the above queries). Thus the behaviour is at least \( O(n^2) \) (not counting the computations of joins, duplicate elimination, and so on).

So the original “magic set” method is not always as efficient as SLD-resolution, not even asymptotically. This rises two questions, which we treat in this chapter.
First, if we cannot prove that “Magic Sets beat SLD-Resolution”, is there something which we still can prove on the efficiency of magic sets? After all, it is an important and often used method, so it would be much too hasty to simply throw it away. But we of course want some performance guarantees. This is the subject of Section 3.1.

The second question is of course to ask for improvements of the “magic set” technique, which can simulate SLD-resolution better, especially on tail-recursive programs. We propose such a method in Section 3.2.

Both questions have already been treated in the literature, but we still feel that we can make important contributions.

In [Sek89], Seki compares the efficiency of “Alexander Templates” (a variant of magic sets) with a version of SLD-resolution with memoing (tabulation). But precisely because of this memoing, which is also used in Ullman’s QRGT-resolution, the efficiency advantages of SLD-resolution are destroyed. It means that the IDB-facts (lemmas) proven in an SLD-tree are explicitly stored (“materialized”), while in case of tail-recursive programs (and only there, as we will see), a single node in an SLD-tree can complete the proofs of an unbounded number of lemmas.

In [Bry90b], Bry shows how top-down evaluation can be simulated by evaluating a meta-interpreter bottom-up. Many known query-evaluation methods, and especially the “magic sets”, can be derived from this “backward fixpoint procedure”. However, the memoing is implicit in this approach, so the special advantages of SLD-resolution are again not considered. But in Section 3.2, we will generalize this meta-interpreter in such a way that it can simulate SLD-resolution much more directly. Our approach
owes much to Bry’s work.

Ross has proposed in [Ros91] a variant of the magic-set technique with a tail-recursion optimization. His idea was to add to a tail-recursive literal $A$ the literal $B$, which is really proven when the proof of $A$ is finished. Ramakrishnan and Sudarshan [RS91, Sud92] have compared the efficiency of this method with Prolog-evaluation. And the result was that this method is as efficient as Prolog (on an abstract level). So this is already a solution to the problem of magic sets.

However, we will show in Section 3.2 that it is possible to go further in simulating SLD-resolution. The goal of Ross was to solve the problem of tail-recursion, our goal is to implement SLD-resolution on top of a deductive database. The tail-recursion optimization is only a consequence of our approach, and it gives also a number of other known optimizations “for free”. Furthermore, our method can be described simply and elegantly by adapting Bry’s idea of using a meta-interpreter to derive the “magic set” method [Bry90b]. So it might help to understand and integrate other optimizations. While the method of Ross works with pairs of literals, we simulate lists of literals $A_1 \land \cdots \land A_n$ as used in SLD-resolution. Our method has advantages also in the case of non-recursive programs, whereas the method of Ross reduces to the standard method in this case. Finally, we try to produce range-restricted Datalog-clauses, which can be more efficiently evaluated than the HiLog-rules used by Ross.

Let us also note that the currently fastest deductive database system, namely the XSB-system, does not use bottom-up evaluation with magic sets, but a variant of SLD-resolution with memoing. This means that it has the same problems with tail-recursive programs, and yet it is faster than bottom-up evaluation with magic sets [SSW94]. So probably any contribution to simulate SLD-resolution more directly could help to make “real” deductive database systems catch up with the XSB-system, and fully utilize the theoretical advantages of bottom-up evaluation.

The results in Section 3.1 were published in [Bra95], the results of Section 3.2 will appear in [Bra96].
3.1 The “Magic Set” Transformation

In this section, we formally define the “magic set” transformation, prove its correctness, and analyze its efficiency. We believe that we can explain the efficiency of this method much clearer than it has previously been done in the literature.

In summary, we show that the only problem of magic sets is the materialization of “lemmas” (derived IDB-facts), and that this has an essential effect only in case of tail-recursive programs. And although “magic sets” are sometimes not “as efficient as” SLD-resolution, they are in some precise sense always “as goal-directed as” SLD-resolution (Theorems 3.1.17 and 3.1.18).

We also give rather simple proofs which can be demonstrated to students. For instance, ULLMAN’s efficiency comparison is a lot more detailed than our one (besides relating to another algorithm). But this of course makes his proofs quite complicated. In contrast, we will consider a very simple (“high level”) cost measure, namely the number of nodes of the SLD-tree on one side, and the number of applicable rule instances on the other side. Certainly, a more detailed analysis is very valuable, but it is good to have a simple result first.

In this chapter, we take the deductive database view and assume that the set of predicates \(\text{PRED}\) is disjointly partitioned into EDB-predicates \(\text{EDB}\), defined by a set of facts \(\text{DB}\), and IDB-predicates \(\text{IDB}\), defined by a range-restricted program \(P\).

Given such a program \(P\) and a query \(Q\), the goal of the magic sets transformation is to construct a new program and query which are “more efficiently evaluable” and answer-equivalent:

**Definition 3.1.1 (Answer Equivalent):**

Let \(P_1\) and \(P_2\) be two Datalog-programs and let \(Q_1\) and \(Q_2\) be two queries with \(\text{vars}(Q_1) = \text{vars}(Q_2)\). We call the pairs \((P_1, Q_1)\) and \((P_2, Q_2)\) answer-equivalent iff for all \(\text{DB} \subseteq \text{BASE}(\text{EDB})\) and all ground substitutions \(\theta\) the following holds:

\[
\text{DB} \cup P_1 \models Q_1 \theta \iff \text{DB} \cup P_2 \models Q_2 \theta.
\]

It is essential to make a distinction between program and database, and to allow only transformations which are independent of the database. Otherwise we could transform every program simply into facts defining values for the answer variables.

We must note at this point that the result of the “magic set” transformation is not always more efficiently evaluable than the original program. It is possible to construct examples where the transformation makes the program worse, see [Sag90].

**SLD-Resolution**

In this Chapter, we compare the efficiency of magic sets with SLD-resolution. For precision and completeness, let us quickly review its definition. We consider here only the “first literal” selection function (as known from Prolog). Other selection functions would be possible, but they have to correspond to the SIP-rule used in the “magic set” transformation (see below). We can assume for simplicity that the programmer or the system has already ordered the body literals in a good way.
Definition 3.1.2 (SLD-Tree):
Let \( P \) be a Datalog-program, \( Q \) be a query and \( DB \subseteq BASE(EDB) \). Then the SLD-tree for \( P \cup \{ \neg Q \} \) wrt \( DB \) is a tree with nodes marked by goals and edges by substitutions defined as follows:

- The root node is marked with the goal \( Q \).
- If a node is marked with the goal \( A_1 \land \cdots \land A_n, n \geq 1 \), and the predicate of \( A_1 \) is in \( IDB \), then the child nodes are constructed as follows: For every rule \( A \leftarrow B_1 \land \cdots \land B_m \) in \( P \), let \( A' \leftarrow B'_1 \land \cdots \land B'_m \) be a variant with new variables, which do not occur in \( A_1 \land \cdots \land A_n \) and the nodes and unifiers above. If \( A_1 \) and \( A' \) are unifiable, let \( \theta \) be a most general unifier. Then there is a child node marked with the goal 
  \[(B'_1 \land \cdots \land B'_m \land A_2 \land \cdots \land A_n)\theta,\]
  the edge is marked with the unifier \( \theta \). For every rule \( A \leftarrow B_1 \land \cdots \land B_m \) there is at most one child node (the variant of the rule and the mgu can be arbitrarily chosen).
- If a node is marked with the goal \( A_1 \land \cdots \land A_n, n \geq 1 \), and the predicate of \( A_1 \) is in \( EDB \), the child nodes are determined as follows: For every ground substitution \( \theta \) for \( A_1 \) such that \( A_1\theta \in DB \) there is a node \((A_2 \land \cdots \land A_n)\theta\). The edge to this node is marked with \( \theta \).
- A node marked with the empty goal \( \Box \) has no child nodes. Such a node is called a success node.

A path starting at the root node and ending in a success node
\[Q \xrightarrow{\theta_1} G_1 \xrightarrow{\theta_2} \cdots \xrightarrow{\theta_{k-1}} G_{k-1} \xrightarrow{\theta_k} \Box\]
is called an SLD-refutation. The composition of the substitutions \( \theta_1, \ldots, \theta_k \), restricted to the variables of \( Q \), is called a computed answer substitution.

It is well known that SLD-resolution is sound and complete for Datalog, i.e. every ground substitution \( \theta \) with \( DB \cup P \vdash Q\theta \) is also a computed answer substitution.

Next, we need the notion of a “lemma” in SLD-resolution. With the “first literal” selection rule, the current goal operates as a stack of literals. So we work on the first literal of a given goal and do not touch the remaining literals (except for applying substitutions) until the first literal is “resolved away”. At this point we have proven the corresponding instance of the first literal as a “lemma”:

Definition 3.1.3 (Lemma Proven in an SLD-Tree):
Let
\[(A_1 \land \cdots \land A_n) \xrightarrow{\theta_1} G_1 \xrightarrow{\theta_2} \cdots \xrightarrow{\theta_{k-1}} G_{k-1} \xrightarrow{\theta_k} G_k\]
be a path in an SLD-tree, where \( G_k \) has the form \((A_2 \land \cdots \land A_n)\theta_1 \ldots \theta_k\), and every goal \( G_i, i = 1, \ldots, k - 1 \), consists of at least \( n \) literals. Then we say that \( A_1\theta_1 \ldots \theta_k \) has been proven as a lemma during SLD-resolution.
By the completeness of SLD-resolution, if a ground instance of the first literal in a node is logically implied, it is actually proven as a lemma:

**Lemma 3.1.4:**

Let \( A_1 \land \cdots \land A_n \) be a node in an SLD-tree, and let \( \theta \) be a ground substitution for \( A_1 \) such that \( DB \cup PA_1 \theta \). Then this node has the descendant node \((A_2 \land \cdots \land A_n)\theta\).

**Proof:**

By the completeness of SLD-resolution, there is an SLD-refutation

\[
A_1 \overset{\theta_1}{\rightarrow} G_1 \overset{\theta_2}{\rightarrow} \cdots \overset{\theta_{k-1}}{\rightarrow} G_{k-1} \overset{\theta_k}{\rightarrow} \Box
\]

such that \( \theta_1 \circ \cdots \circ \theta_k \) restricted to the variables of \( A_1 \) is \( \theta \) (because of the range-restriction, the computed answer substitution must be a ground substitution). The applied rule variants can be chosen in such a way, that the \( \theta_i \) do not substitute any variables in \( A_2 \land \cdots \land A_n \) except those occurring also in \( A_1 \). Now we can “append” \( A_2 \land \cdots \land A_n \) to this SLD-refutation: \( A_1 \land \cdots \land A_n \) has the child node \( G_1 \land (A_2 \land \cdots \land A_n)\theta_1 \) and so on until finally the descendant \( \Box \land (A_2 \land \cdots \land A_n)\theta_1 \circ \cdots \circ \theta_k \) is reached, which is exactly \((A_2 \land \cdots \land A_n)\theta\). \( \square \)

**Rectification**

As noted by ULLMAN [Ull89a], the magic set approach can be much less efficient than SLD-resolution because it cannot handle variable-to-variable bindings. His example was recursive, but the following simpler example also demonstrates the problem:

**Example 3.1.5:**

Consider the program consisting of the following two rules:

\[
p(Y_1, Y_2, Y_3) \leftarrow q(X, X, Y_1, Y_2, Y_3).
q(a, b, Y_1, Y_2, Y_3) \leftarrow r(Y_1) \land r(Y_2) \land r(Y_3).
\]

Let the query be \( p(Y_1, Y_2, Y_3) \) and \( r \) be an EDB-predicate with \( n \) facts. In this case the magic set transformation treats \( q \) as called without any arguments bound, so the bottom-up evaluation of the transformed program really constructs \( n^3 \) facts about \( q \). However, the SLD-tree for the same query contains only 2 nodes. \( \square \)

This is in fact no problem, because any program can be “rectified”:

**Definition 3.1.6 (Rectified Program):**

A logic program \( P \) is rectified iff no body atom contains the same variable twice, i.e. for every atom \( p(t_1, \ldots, t_n) \) in the body of a rule holds: If \( t_i = t_j \) for \( i \neq j \), then \( t_i \) is a constant.

We also call a query \( p(t_1, \ldots, t_n) \) rectified, if it does not contain the same variable twice.

In [Ull89a, Ull89b], a “subgoal-rectified” program is also not allowed to contain constants in the rule bodies. For our efficiency result, however, this is not important.
Note that our condition also does not exclude multiple occurrences of the same variable in the rule head — because of the range-restriction, the variable will be bound to a constant before it can do any harm.

It is not difficult to rectify any given program. The idea [Ull89a, Ull89b] is to introduce new predicates where equal arguments are merged into one argument. In the example, we replace \( q(X, X, Y_1, Y_2, Y_3) \) by \( q^{1234}(X, Y_1, Y_2, Y_3) \). In general, the \( j \)-th argument of a predicate \( p \) of arity \( n \) is represented as the \( i_j \)-th argument of \( p^{i_1 \ldots i_n} \). Then the rules about \( p \) are specialized to \( p^{i_1 \ldots i_n} \) by unifying the head with \( p(X_1, \ldots, X_n) \).

We will not give the algorithm here (the algorithm in [Ull89a, Ull89b] produces an even stricter rectification than we need), but note the important properties:

- The resulting program \( P_{\text{rect}} \) and query \( Q_{\text{rect}} \) are rectified.
- \( P_{\text{rect}} \) and \( Q_{\text{rect}} \) are answer-equivalent to the original program and query.
- The SLD-tree for \( P_{\text{rect}} \cup \{ \leftarrow Q_{\text{rect}} \} \) has the same number of nodes as the SLD-tree for the original program \( P \) and query \( Q \). In fact, one can simply replace all literals of the form \( p^{i_1 \ldots i_n}(t_1, \ldots, t_k) \) by their old counterparts \( p(t_{i_1}, \ldots, t_{i_n}) \) in order to get the SLD-tree for \( P \cup \{ \leftarrow Q \} \).

In general, the rectification can create a lot of new predicates and really “blow up” the program. However, it is unclear whether this happens also in practical applications, or only in theoretically constructed examples. Of course, if there should be many different versions of one predicate, the naive bottom-up evaluation of the rectified program takes much more time than the evaluation of the original program. However, we will show that the magic set evaluation of the rectified program is as fast as SLD-resolution for the rectified program, and we already know that this is as fast as the SLD-resolution for the original program. So for our theorem the possible blowup due to the rectification does no harm, whereas not doing the rectification is harmful, as Example 3.1.5 showed.

### Explicit Binding Patterns (Adornments)

There are many variations of the magic set technique in the literature. In order to be precise and self-contained, we give a quick review of the magic set transformation in this section and the next. We also prove all necessary properties, which are needed as lemmas for our efficiency result.

Before the magic set transformation itself can be applied, we have to make the binding patterns explicit which would occur during SLD-resolution. We have talked about binding patterns already in Chapter 2, and what we do here is simply to create a different version of every IDB-predicate \( p \) for every possible binding pattern \( \beta \). We denote this predicate by \( p^\beta \). So we now extend the set of Predicates to

\[
PRED_{\text{adorn}} := EDB \cup \{ p^\beta/n \mid p/n \in IDB, \; \beta: \{1, \ldots, n\} \rightarrow \{b, f\} \}.
\]

For simplicity of the presentation, we always assume that the body literals are called from left to right. Much research has been done on “sideways information passing (SIP)” strategies which select a better sequence. For instance, one can choose to evaluate next the literal with a maximal number of bound arguments, or a minimal
number of unbound ones, and one can also use possible knowledge about functional dependencies (keys), existing indexes, or relation sizes. It is usually assumed that a standard relational query optimization will be done later, but in fact, the SIP strategy already determines a lot. However, for our purposes, we can ignore these problems. We simply assume that the system already has ordered the body literals in such a way that the DB-literals are called with valid binding patterns if the rule body is executed from left to right. As we noted before, the “selection function” of SLD-resolution must of course be compatible with the SIP-strategy, so we consider only the “first literal” selection function (known from Prolog).

Definition 3.1.7 (Adorned Program):

- Given a rule \( A \leftarrow B_1 \land \cdots \land B_m \) and a binding pattern \( \beta \) for \( A \) (i.e. for its predicate), the binding patterns \( \gamma_i \) for the body literals \( B_i \equiv q_i(t_{i,1}, \ldots, t_{i,n_i}) \) with \( q_i \in IDB \) are determined as follows: If \( t_{i,j} \) is a constant or a variable which appeared already in a \( B_k \) with \( k < i \) or in a bound position of \( A \), then \( \gamma_i[j] = b \), else \( \gamma_i[j] = f \). The literals with predicates in \( EDB \) are unchanged.

- Given a program \( P \), its adorned version \( P_{adorn} \) consists of the adorned versions of all rules in \( P \) for every possible binding pattern for the head literal.

- The adornment \( \beta \) of a query \( p(t_1, \ldots, t_n) \) with \( q \in IDB \) is defined by: \( \beta[i] = b \) if \( t_i \) is a constant, and \( \beta[i] = f \) otherwise.

Theoretically, it is simplest to assume first that all possible binding patterns for all predicates are constructed — so there will be \( 2^n \) versions of a rule about a predicate \( p \) of arity \( n \). Of course, given a specific query \( p^\beta(\ldots) \), only rules about predicates on which \( p^\beta \) depends are needed, and all other rules can obviously be eliminated without changing the answer. Therefore, an implementation of adorning will do the two steps together and never construct the completely adorned program. But for the complete version \( P_{adorn} \) it is especially simple to see that adorning does not change the set of correct answers:

Lemma 3.1.8:

- If \( DB \cup P \vdash p(c_1, \ldots, c_n) \), then \( DB \cup P_{adorn} \vdash p^\beta(c_1, \ldots, c_n) \) for all \( \beta \).

- If \( DB \cup P_{adorn} \vdash p^\beta(c_1, \ldots, c_n) \) for some \( \beta \), then \( DB \cup P \vdash p(c_1, \ldots, c_n) \).

Proof:

- This is proven by induction on the number of iterations in the fixpoint construction needed to derive \( p(c_1, \ldots, c_n) \). Consider the last derivation step and let \( p(t_1, \ldots, t_n) \leftarrow B_1 \land \cdots \land B_m \) be the applied rule and \( \theta \) be the instantiation. Then \( P_{adorn} \) contains a rule of the form \( p^\beta(t_1, \ldots, t_n) \leftarrow B_1^{\gamma_1} \land \cdots \land B_m^{\gamma_m} \).

But if \( DB \cup P \vdash B_i \theta \), then \( DB \cup P_{adorn} \vdash B_i^{\gamma_i} \theta \) by the induction hypothesis.
• This is trivial, one only has to delete the adornments in a derivation of \( p^3(c_1, \ldots, c_n) \) from \( P_{\text{adorn}} \) to get a derivation of \( p(c_1, \ldots, c_n) \) from \( P \).

\[ \square \]

**Lemma 3.1.9:**
The SLD-tree for the adorned program \( P_{\text{adorn}} \) and query \( Q_{\text{adorn}} \) has the same number of nodes as the SLD-tree for \( P \cup \{ \leftarrow Q \} \).

**Proof:**
This is again trivial: Simply remove the adornments to get the SLD-tree for \( P \cup \{ \leftarrow Q \} \).

\[ \square \]

For the preceding lemmas it was completely irrelevant which binding patterns are assigned to the body literals. In order to understand the real meaning of the adornments, we must prove that they correspond to the binding patterns occurring during SLD-resolution.

To formalize this, we assume that also SLD-resolution is applied to the adorned program. This is slightly unusual, because the adornments were only introduced as a preprocessing step for the magic set transformation. But due to Lemma 3.1.9 adorning does not influence SLD-resolution. Furthermore, the same was true for the rectification. So it suffices to consider in the following rectified and adorned programs, because these properties can be achieved without decreasing the efficiency of SLD-resolution.

**Definition 3.1.10 (Correct Adornment):**
A literal \( p^3(t_1, \ldots, t_n) \) has the correct adornment iff the following holds:

- If \( \beta[i] = b \), then \( t_i \) is a constant.
- If \( \beta[i] = f \), then \( t_i \) is a variable.
- No variable appears twice in \( p^3(t_1, \ldots, t_n) \).

**Theorem 3.1.11 (Correct Adornments in SLD-Tree):**
Let \( P \) and \( Q \) be rectified and adorned. Then the first literal of every node in the SLD-tree of \( P \cup \{ \leftarrow Q \} \) has the correct adornment.

**Proof:**
The proof is by induction on the length of the SLD-derivation to that node (i.e. the distance from the root). In order to make the induction work, we prove the following statements about nodes \( A_1 \wedge \cdots \wedge A_n \) of the SLD-tree:

- Let \( X \) be a variable which appears in \( A_i \), but in no \( A_j \) with \( j < i \) (i.e. this occurrence is the first). Then \( X \) appears in a free argument position in \( A_i \). Furthermore, \( X \) appears only once in \( A_i \).
- Let \( X \) be a variable which appears in \( A_i \) and in some \( A_j, j < i \) (i.e. this occurrence is not the first). Then the argument position in \( A_i \) is bound.
- Constants appear only in bound argument positions.

For the first literal \( A_1 \), this directly implies the statement of the theorem.

For the given query \( Q \), the conditions hold by definition of the adornment (and because the query is rectified). Before we treat the general resolution with a rule, let us consider
the evaluation of literals defined in the database \(DB\). This yields \((A_2 \land \cdots \land A_n)\theta\), where \(\theta\) substitutes the variables occurring in \(A_1\) by constants. Our conditions were satisfied in \(A_1 \land \cdots \land A_n\), so let us consider the differences:

- The substitution \(\theta\) replaces some variable occurrences by constants. But these variables occurred also in \(A_1\), i.e. were not first occurrences, and are thus marked “bound”.
- Since the literal \(A_1\) is removed, some variable occurrences would become first occurrences which were not first occurrences before. But this means that the variable also occurred in \(A_1\) and thus is replaced by a constant.

Now let a rule

\[
p^\beta(u_1, \ldots, u_k) \leftarrow B_1 \land \cdots \land B_m
\]

be applied to \(A_1 \equiv p^\beta(t_1, \ldots, t_k)\). As usual, we assume that the variables of the rule are disjoint from the variables of \(A_1 \land \cdots \land A_n\). By the induction hypothesis, we know which \(t_i\) are variables, so we can compute the most general unifier \(\theta\) as follows:

- If \(\beta[i] = f\), we substitute \(t_i\) by \(u_i\).
- If \(\beta[i] = b\), and \(u_i\) is not yet substituted, we substitute \(u_i\) by \(t_i\). (If \(u_i\) is a constant, it must be equal to \(t_i\). If \(u_i\) is a variable which was already substituted, i.e. \(u_i = u_j\), with \(j < i\) and \(\beta[j] = b\), then \(t_i = t_j\) must hold.)

Now we have to verify our conditions for the child node

\[
(B_1 \land \cdots \land B_m \land A_2 \land \cdots \land A_n)\theta.
\]

Since the variables of the \(B_i\) and \(A_j\) were disjoint, the \(B_i\) are effected only by the second kind of substitution. So exactly the variables in bound argument positions of the head were substituted by constants, and the conditions directly follow from the construction of the adorned rule and the rectification.

Now let us consider the \(A_j\). Since the conditions were satisfied before the derivation step, it suffices to look at the variables \(t_i\) which were substituted. Occurrences of \(t_i\) in \(A_j\), \(j \geq 2\), are not the first occurrence of these variables (the first occurrence was in \(A_1\)). So they occur only in bound argument positions. If \(t_i\) was substituted by a constant, the condition is obviously satisfied. If \(t_i\) was substituted by a variable \(u_i\), this variable appears also in \(B_1 \land \cdots \land B_m\) (due to the range restriction), so occurrences in \(A_j\theta\), \(j \geq 2\), are again not the first occurrence.

\[\square\]

The Magic Set Transformation

Now the magic set transformation itself can be done. The idea is to introduce for every IDB-predicate \(p\) and binding pattern \(\beta\) a new predicate \(m_p^\beta\) which contains the bindings for which we want to compute all corresponding \(p\)-facts. Then we make the rules applicable only if we really need their result.

So we again extend the set of predicates:

\[
PRED\text{S}_{\text{magic}} := PRED\text{S}_{\text{adorn}} \cup \{m_p^\beta/n | p/n \in IDB, \beta \text{ binding pattern for } p/n\}.
\]

Literals with the new predicates \(m_p^\beta\) are called magic literals, literals with the original predicates \(p^\beta\) are called non-magic literals. The following notation is very useful:
Definition 3.1.12 (Binding Information of a Literal):

Let

\[ \text{magic} [p^\beta(t_1, \ldots, t_n)] := m_p^\beta(t_{i_1}, \ldots, t_{i_k}), \]

where \( 1 \leq i_1 < \cdots < i_k \leq n \) are the indices with \( \beta[i_j] = b \).

Now the magic set transformation itself is very simple:

Definition 3.1.13 (Magic Set Transformation):

Let \( P \) and \( Q \) be rectified and adorned. Then the result \( P_{\text{magic}} \) of the magic set transformation consists of:

- For every rule \( A \leftarrow B_1 \land \cdots \land B_m \) from \( P \) the “modified rule”
  \[ A \leftarrow \text{magic}[A] \land B_1 \land \cdots \land B_m. \]

- For every rule \( A \leftarrow B_1 \land \cdots \land B_m \) from \( P \) and every \( B_i \) with IDB-predicate the “magic rule”
  \[ \text{magic}[B_i] \leftarrow \text{magic}[A] \land B_1 \land \cdots \land B_{i-1}. \]

- For the query \( Q \) the “seed”:
  \[ \text{magic}[Q]. \]

The “modified rules” correspond to the old program, however, they are restricted to “useful” applications, which contribute to our goal. The “magic rules” formalize that if we want to know the head, we also want to know the body literals, as long as there is still the chance that the rule is applicable. The “seed” finally states that we want to know all derivable instances of the query.

The modified rule and the magic rules resulting from one rule \( A \leftarrow B_1 \land \cdots \land B_m \) of the original program contain common subexpressions. This can be optimized with so-called “supplementary predicates” [Ull89b]. However, we believe that it is better to consider the pure “magic set” transformation separately from further optimizations.

The correctness of the magic sets transformation can be verified as follows:

Lemma 3.1.14:

Let \( A \) be a non-magic literal.

- If \( DB \cup P_{\text{magic}} \vdash A \), then \( DB \cup P \vdash A \).
- If \( DB \cup P \vdash A \) and \( DB \cup P_{\text{magic}} \vdash \text{magic}[A] \), then \( DB \cup P_{\text{magic}} \vdash A \).

Proof:

- This is trivial: The modified rules are only restricted versions of the original rules. So if they are applicable, also the original rules can be applied to derive \( A \).
- The proof is by induction on the number of iterations in the fixpoint computation needed to derive \( A \) from \( DB \cup P \). Let \( A \) be derived by the rule instance
  \[ A \leftarrow B_1 \land \cdots \land B_m. \]
3.1. THE “MAGIC SET” TRANSFORMATION

Let $B_1, \ldots, B_k$ the body literals with IDB-predicate (from left to right). Since $DB \cup P_{\text{magic}} \vdash magic[A]$, we can for $j = 1, \ldots, k$ first apply the rule instance

$$magic[B_{ij}] \leftarrow magic[A] \land B_1 \land \cdots \land B_{ij-1}$$

to get $DB \cup P_{\text{magic}} \vdash magic[B_{ij}]$ and then apply the inductive hypothesis to get $DB \cup P_{\text{magic}} \vdash B_{ij}$ (The literals with EDB-predicates are trivial, since the database is equal in both cases.). Now we only have to apply

$$A \leftarrow magic[A] \land B_1 \land \cdots \land B_m$$

to get $DB \cup P_{\text{magic}} \vdash A$. \hfill \Box

The following corollary, the correctness of the “magic set” transformation is well-known from the literature:

**Corollary 3.1.15:**

The result of the magic set transformation is answer-equivalent to the original program.

**Proof:**

- If $DB \cup P \vdash Q\theta$, then $DB \cup P_{\text{magic}} \vdash Q\theta$ since $magic[Q\theta] = magic[Q]$ is given as a fact.
- Conversely, if $DB \cup P_{\text{magic}} \vdash Q\theta$, then $DB \cup P \vdash Q\theta$. \hfill \Box

**Efficiency of Magic Sets**

We are now in a position to state and prove our results on the efficiency of the magic set method. As explained above, a comparison between SLD-resolution and the original “magic set” method seems to be missing in the literature, although there are quite a number of comparisons between “top-down” and “bottom-up”. So here is one more version, which, however, treats the in my view most relevant case. Other comparisons have tried to show that “bottom-up” is at least as efficient as “top-down”, we clarify what exactly the problem is.

First, as intuitively expected, the magic facts really correspond to selected literals in the SLD-tree. So this is not the source of the problem:

**Definition 3.1.16 (Magic Facts vs. Goal Literals):**

A literal $p^\beta(t_1, \ldots, t_n)$ matches a magic fact $m_{p^\beta}(c_1, \ldots, c_k)$ iff

- $p^\beta(t_1, \ldots, t_n)$ has the correct adornment and
- $magic[p^\beta(t_1, \ldots, t_n)] = m_{p^\beta}(c_1, \ldots, c_k)$.

**Theorem 3.1.17 (Magic Facts Correspond to Nodes in the SLD-Tree):**

Let $P$ and $Q$ be a rectified and adorned. Then for every magic fact $m_{p^\beta}(c_1, \ldots, c_n)$ derivable from $P_{\text{magic}}$ there is a node in the SLD-tree of $P \cup \{ \leftarrow Q \}$ with matching first literal.
Proof:
The proof is by induction on the number of derivation steps needed for $m_p^\beta(c_1, \ldots, c_n)$. The “seed” fact directly corresponds to the root of the SLD-tree. Now let $m_p^\beta(c_1, \ldots, c_n)$ be derived by a magic rule

$$ magic[B_i] \leftarrow magic[A] \land B_1 \land \cdots \land B_{i-1} $$

instantiated by $\theta$. So

$$ m_p^\beta(c_1, \ldots, c_n) = magic[B_i] \theta = magic[B_0]. $$

Let $\theta_0$ be the restriction of $\theta$ to the variables in $magic[A]$, $\theta_1$ be the restriction of $\theta$ to the variables which first appear in $B_1$, and so on. So $\theta$ can be written as $\theta = \theta_0 \circ \cdots \circ \theta_{i-1}$.

Since $magic[A\theta_0]$ was derived with fewer steps, we can apply the inductive hypothesis to conclude that there is a node $A_1 \land \cdots \land A_k$ in the SLD-tree where $A_1$ matches $magic[A\theta_0]$.

Now we of course resolve $A_1$ with $A \leftarrow B_1 \land \cdots \land B_m$. We compute the most general unifier as in the proof of Theorem 3.1.11, i.e. we substitute variables at bound argument positions in $A$ by the corresponding constants in $A_1$, this gives exactly $\theta_0$, and variables in free argument positions of $A_1$ by the corresponding values in $A$, let this substitution be called $\sigma$. Because $\theta_0$ is a ground substitution, the mgu can be written as $\sigma \theta_0$. Then the child node in the SLD-tree is

$$(B_1 \land \cdots \land B_m) \theta_0 \theta_1 \land (A_2 \land \cdots \land A_k) \sigma \theta_0$$

(Since the variables were disjoint before the resolution, $B_1 \land \cdots \land B_m$ is not affected by $\sigma$.)

But now, since we know that $P_{magic} \vdash B_1 \theta_0 \theta_1$, we get $P \vdash B_1 \theta_0 \theta_1$ by Lemma 3.1.14, and then Lemma 3.1.4 ensures the existence of a descendant node

$$(B_2 \land \cdots \land B_m) \theta_0 \theta_1 \land (A_2 \land \cdots \land A_k) \sigma \theta_0 \theta_1.$$ 

Continuing this, we finally get to

$$(B_i \land \cdots \land B_m) \theta_0 \theta_1 \cdots \theta_{i-1} \land (A_2 \land \cdots \land A_k) \sigma \theta_0 \theta_1 \cdots \theta_{i-1}.$$ 

But now $B_i \theta_0 \theta_1 \cdots \theta_{i-1}$ i.e. $B_i \theta$ matches $m_p^\beta(c_1, \ldots, c_n)$: By Theorem 3.1.11 it has the correct adornment, and

$$ magic[B_i \theta] = m_p^\beta(c_1, \ldots, c_n) $$

holds by construction. □

Next, let us look at the non-magic facts, which are derivable after the “magic set” transformation. We will show that they are proven as “lemmas” in the SLD-resolution:

**Theorem 3.1.18 (Non-Magic Facts Are Lemmas in the SLD-Tree):**

Let $P$ and $Q$ be a rectified and adorned. Then every non-magic fact $p^\beta(c_1, \ldots, c_n)$ derivable from $P_{magic}$ is proven as a lemma during SLD-resolution.
3.1. THE “MAGIC SET” TRANSFORMATION

Proof:
Let \( p^3(c_1, \ldots, c_n) \) be derived by the rule

\[
A \leftarrow \text{magic}[A] \land B_1 \land \cdots \land B_m
\]

instantiated with \( \theta \). So \( \text{magic}[A] \theta \) is derivable from \( P_{\text{magic}} \), and Theorem 3.1.17 tells us that there is a node \( A_1 \land \cdots \land A_k \) in the SLD-tree with matching first literal. We now proceed as in the proof to Theorem 3.1.17, i.e. compute the unifier \( \sigma \theta_0 \), and then use \( P_{\text{magic}} \vdash B_i \theta \) together with Lemma 3.1.14 and Lemma 3.1.4 to resolve the \( B_i \) away. This finally leads to a descendant node

\[
(A_2 \land \cdots \land A_k)\sigma \theta_0 \cdots \theta_m.
\]

But since already \( A_1 \sigma \theta_0 = A \theta_0 \), we can conclude that the proven lemma \( A_1 \sigma \theta_0 \cdots \theta_m \) is in fact \( A \theta \).

This proves our claim that bottom-up evaluation after the magic set transformation is “as goal-directed as” SLD-resolution. This is important, since it is usually considered to be the main problem of pure bottom-up evaluation that it is not goal-directed.

But let us now look at the efficiency results. As explained above, we compare the number of applicable rule instances in \( P_{\text{magic}} \) to the number of nodes in the SLD-tree. Both are very high level (simple) cost measures. First, the magic rules never pose a problem, they correspond directly to nodes of the SLD-tree:

Theorem 3.1.19 (Number of Applicable Instances of Magic Rules):
Let \( P \) and \( Q \) be a rectified and adorned. Then the number of applicable instances of magic rules in \( P_{\text{magic}} \) is \( \leq \) the number of nodes in the SLD-tree of \( P \cup \{ \leftarrow Q \} \).

Proof:
Let us modify for the moment SLD-resolution in such a way that when it resolves with \( A \leftarrow B_1 \land \cdots \land B_m \), it attaches

\[
\text{magic}[B_i] \leftarrow \text{magic}[A] \land B_1 \land \cdots \land B_{i-1}
\]

to \( B_i \) (and also applies all substitutions to these attachments). We now assign to each node the rule instance attached to its first literal.

Now we of course have to verify that all applicable magic rule instances are actually assigned to some node of the SLD-tree, i.e. the mapping is surjective. Let us consider

\[
\text{magic}[B_i] \leftarrow \text{magic}[A] \land B_1 \land \cdots \land B_{i-1}
\]

instantiated by \( \theta \). If this rule instance is applicable, \( \text{magic}[A] \theta \) is derivable, so by Theorem 3.1.17 there is a node with matching first literal. It can be resolved with the rule \( A \leftarrow B_1 \land \cdots \land B_m \). At this point, the magic rule in question is assigned to \( B_i \). Now if we do the following derivation as in the proof of Theorem 3.1.17, i.e. utilize \( P_{\text{magic}} \vdash B_j \theta \) \((j = 1, \ldots, i-1)\) together with Lemma 3.1.14 and Lemma 3.1.4, \( B_i \theta \) finally gets to the front.

As the example at the beginning of this chapter showed, a corresponding statement for non-magic facts does not hold. The problem is that one node in the SLD-tree
can prove an unbounded number of lemmas. For instance, consider the success node computing the answer \( X = n \). It completes the proofs of the lemmas \( \text{anc}(i, n) \) for all nodes \( \text{anc}(i, X) \) above it. However, it is not difficult to see that this can happen only if there is a tail-recursive rule. Otherwise, by resolving away a single literal, we can complete only a restricted number of rule bodies above it (namely, as many as we have predicates).

**Theorem 3.1.20 (Number of Applicable Instances of Non-Magic Rules):**

> Let \( P \) and \( Q \) be a rectified and adorned. If \( P \) contains no tail-recursive rules (i.e. the last literal of every rule body does not depend on the head), the number of applicable instances of non-magic ("modified") rules in \( P_{\text{magic}} \) is

\[
O\left(\text{number of nodes in the SLD-tree of } P \cup \{ \leftarrow Q \}\right).
\]

**Proof:**

We can again visualize the applied rule instances in the SLD-tree. This time, when we apply \( A \leftarrow B_1 \land \cdots \land B_m \) to a node \( A_1 \land \cdots \land A_k \) in the SLD-tree, we put the corresponding modified rule as a comment into the child node (similar to the framed literals of OL-resolution):

\[
(B_1 \land \cdots \land B_m \land [A \leftarrow \text{magic}[A] \land B_1 \land \cdots \land B_m]) \theta_0 \land (A_2 \land \cdots \land A_k) \sigma \theta_0.
\]

As explained above, if the rule instance is applicable, the \( B_i \) will be resolved away and the "framed" portion will come to the beginning of a descendant node. We delete it from the node at that time, because we need to access literals behind it, but we assign to every node the framed rule instances deleted at that node. There can be several such framed rule instances before the beginning of proper (non-framed) literals, but not more than there are predicates: Because we remove the framed rule instances once a literal has been completely proven, the situation

\[
\ldots [A' \leftarrow \ldots] \land [A \leftarrow \text{magic}[A] \land B_1 \land \cdots \land B_m] \land \ldots
\]

can only occur if \( B_m \) has called \( A' \) (i.e. they have the same predicate). So if we had a chain which is longer than the number of predicates, we would have a tail recursion.

So this verifies that the only problem with magic sets is the materialization of "lemmas" in the case of tail recursive programs.
3.2 An Improved Magic Set Technique

In this section we present a rewriting method for Datalog-programs which simulates SLD-resolution more closely than the ordinary “magic set” method does. This is especially advantageous in the case of tail-recursive programs, but already in non-recursive programs we can often save a number of joins. Furthermore, we get some other known optimizations “for free” when we simulate SLD-resolution in this way.

We use Bry’s idea to start with a meta-interpreter, which describes top-down evaluation, but is intended for bottom-up evaluation [Bry90b]. However, Bry’s meta-interpreter corresponds to the QRGT-algorithm, therefore he got the standard magic-set transformation. In fact, [Bry90b] also contains a meta-interpreter which corresponds more directly to SLD-resolution, but it still implicitly uses the critical memoing of derived lemmas.

So we present a bottom-up meta-interpreter which corresponds exactly to SLD-resolution. This is in itself simple, but the partial evaluation of this meta-interpreter needs some thoughts. Our contribution is to show that the partial evaluation is indeed possible for tail-recursive programs. We introduce for this purpose the notion of “node types”. Of course, if we simulate SLD-resolution bottom-up, we can guarantee termination (for tail-recursive Datalog programs without built-in predicates).

We then explain the main differences between this bottom-up implementation of SLD-resolution and magic sets. In short, SLD-resolution gives the called literals more context information, namely, how the proof will continue after the literal is proven. In contrast, magic sets usually need a join to get this context back. The disadvantage of SLD-resolution is that it might happen that the same literal is proven more than once if it is called in different contexts. This is avoided by magic sets, but they might introduce new recursions by merging different computations. So both methods have advantages of their own, and the real optimum will be some combination.

Thus, we extend the meta-interpreter in such a way that it can do both, either simulate SLD-resolution or magic sets (selectable for every literal by annotations in the program). The idea for combining the two methods is that we get magic sets from SLD-resolution if we evaluate IDB-literals in a subproof, and explicitly get back the answer. Something similar happens with negative literals in SLDNF-resolution: they are also evaluated by a recursive call to SLDNF-resolution. Another possibility to understand our extended meta-interpreter is that it describes SLD-resolution with tabulation.

Simulating SLD-Resolution via Bottom-Up Evaluation

Our goal in this subsection is to present a bottom-up meta-interpreter which describes SLD-resolution.

As usual for meta-interpreters, the rules of the given program are treated as input data, so they are stored in the database. We represent rules as facts about a binary predicate rule, where the first argument is the head literal and the second argument is the list of body literals. For instance,

\[ \text{anc}(X, Z) \leftarrow \text{parent}(X, Y) \land \text{anc}(Y, Z) \]
is stored as

\[ \text{rule}(\text{anc}(X, Z), [\text{parent}(X, Y), \text{anc}(Y, Z)]). \]

Of course, this fact is neither Datalog (the old predicates are now function symbols) nor range-restricted (it contains variables, but does not have a rule body). Meta-interpreters always need a more general rule language, but this is only temporary, because we will later try to remove this by partial evaluation.

Besides the rules of the program, of course also the facts in the database are input to our meta-interpreter. We assume that the tuples in the database predicates are represented in a common relation \( db \), namely in the form

\[ db(p(c_1, \ldots, c_n)). \]

In this way, the meta-interpreter does not depend on the set of EDB-predicates.

Finally, the query \( Q \) (a single positive literal) is represented by the fact

\[ \text{query}(Q). \]

Our goal now is to compute the nodes of the SLD-tree, more precisely, the goals attached to these nodes. But we need of course also the computed substitution for the answer variables. We solve this problem by attaching to each node the given query literal. It shares variables with the goal at the node, and all substitutions are also applied to the query literal. So when we have derived the empty goal, the corresponding instance of the query has been proven. For example, the following node-fact might be derivable:

\[ \text{node}(\text{anc}(0, X), [\text{parent}(3, X)]). \]

This means that a parent \( X \) of person number 3 is also an ancestor of person number 0. There are of course also other possibilities to remember the substitution for the answer variables, but this encoding of the nodes is especially useful when we later have to manage more than one SLD-tree.

The meta-interpreter consists of four rules. First, we need to create the root node of the SLD-tree. Of course, the goal in the root node is the given query literal:

\[
\begin{align*}
\text{node}(\text{Query}, [\text{Query}]) & \leftarrow \\
\text{query}(\text{Query}).
\end{align*}
\]

The second rule describes the SLD-resolution step. It is the main rule of the meta-interpreter, which does the “real work”. As usual in SLD-resolution, we replace the first goal literal by the body of a rule with a matching head literal:

\[
\begin{align*}
\text{node}(\text{Query}, \text{Child}) & \leftarrow \\
\text{node}(\text{Query}, [\text{Lit}|\text{Rest}] & \leftarrow \\
\text{rule}(\text{Lit}, \text{Body}) & \leftarrow \\
\text{append}(\text{Body}, \text{Rest}, \text{Child}).
\end{align*}
\]

Note that bottom-up evaluation with non-ground facts does the necessary unification, and renames the variables of the used “facts” before that in order to avoid name
3.2. AN IMPROVED MAGIC SET TECHNIQUE

/* Initialization (Root Node): */
node(Query, [Query]) ← query(Query).

/* SLD-Resolution: */
node(Query, Child) ←
node(Query, [Lit|Rest]) ∧
rule(Lit, Body) ∧
append(Body, Rest, Child).

/* Evaluation of DB-Literal: */
node(Query, Rest) ←
node(Query, [Lit|Rest]) ∧
db(Lit).

/* Turn Proven Query into Answer: */
answer(Query) ←
node(Query, []).

Figure 3.2: A Bottom-Up Meta-Interpreter Describing SLD-Resolution

clashes. The variables in the derived facts are again normalized, so that facts which differ only in the variable names are detected as duplicates. This is important for the termination.

The third rule does the evaluation of EDB-literals by finding matching facts in the database:

node(Query, Rest) ←
node(Query, [Lit|Rest]) ∧
db(Lit).

The last rule derives the ground instances of the query which are proven once the empty goal is reached:

answer(Query) ←
node(Query, []).

The complete meta-interpreter is shown in Figure 3.2. It computes exactly the nodes of the SLD-tree.

Example 3.2.1:
This meta-interpreter can be evaluated bottom-up by systems like CORAL\(^1\) which allow structured terms and non-ground facts. For instance, consider again the transitive closure program:

\[
\begin{align*}
anc(X, Y) & \leftarrow parent(X, Y). \\
anc(X, Z) & \leftarrow parent(X, Y) \land anc(Y, Z).
\end{align*}
\]

\(^1\)Version 1.2, which is still available. In version 1.5, non-ground facts are no longer supported (for efficiency reasons).
For simplicity, let the parent-relation contain the single tuple (0, 1). The SLD-tree for the query \( \text{anc}(0, X) \) then looks as follows:

\[
\begin{align*}
\text{anc}(0, X) \\
\text{par}(0, X) & \quad \text{par}(0, Y) \land \text{anc}(Y, X) \\
X = 1 & \quad \text{anc}(1, X) \\
\text{par}(1, X) & \quad \text{par}(1, Y) \land \text{anc}(Y, X)
\end{align*}
\]

CORAL answers the query “\text{?node}(\text{Query}, \text{Node})” with

- Query = \( \text{anc}(0, _X0) \), Node = [\text{anc}(0, _X0)].
- Query = \( \text{anc}(0, _X0) \), Node = [\text{par}(0, _X0)].
- Query = \( \text{anc}(0, _X0) \), Node = [\text{par}(0, _X1), \text{anc}( _X1, _X0)].
- Query = \( \text{anc}(0, 1) \), Node = [].
- Query = \( \text{anc}(0, _X0) \), Node = [\text{anc}(1, _X0)].
- Query = \( \text{anc}(0, _X0) \), Node = [\text{par}(1, _X0)].
- Query = \( \text{anc}(0, _X0) \), Node = [\text{par}(1, _X1), \text{anc}( _X1, _X0)].

(Number of Answers = 7)

The query “\text{?answer}(A)” returns “A=\text{anc}(0, 1)”. So this approach directly simulates SLD-resolution.

Since we do not compute the nodes themselves, but only the goals attached to them, the termination behaviour is better than that of SLD-resolution. For instance, the rule \( p(X) \leftarrow p(X) \) poses no problem at all, since it does not yield new goals. In general, we can guarantee the termination for all tail-recursive Datalog-programs using only finite database predicates. We do not suggest to simulate SLD-resolution for predicates which are recursive, but not tail-recursive. For programs containing different kinds of recursion, we will later present a combined method which allows to use the “magic set” behaviour (tabulation) for calling some literals.

**Example 3.2.2:**

But let us have a look at a non tail-recursive program, for instance the “same generation” example:

\[
\begin{align*}
\text{sg}(X, X) & \leftarrow \text{par}(X, _X).
\text{sg}(X, X) & \leftarrow \text{par}( _X, X).
\text{sg}(X, Y) & \leftarrow \text{par}(X, Xp) \land \text{sg}(Xp, Yp) \land \text{par}(Y, Yp).
\end{align*}
\]

Let the query be \( \text{sg}(a, Y_0) \). If the parent-relation should be cyclic, e.g. consist of the single tuple parent(a, a), longer and longer goals of the form

\[
\text{sg}(a, Y_n) \land \text{parent}(Y_{n-1}, Y_n) \land \ldots \land \text{parent}(Y_1, Y_2) \land \text{parent}(Y_0, Y_1)
\]
3.2. AN IMPROVED MAGIC SET TECHNIQUE

are constructed.

This problem also occurs with the “counting” method [BR86]. One can say that this method also tries to simulate SLD-resolution in its limited application area. It only uses the special encoding \( sg_{down}(a, n) \) for the above node of the SLD-tree. In fact, many known optimizations can be understood by the simple idea to simulate SLD-resolution (see below).

As mentioned above, we can guarantee termination if all recursions are tail-recursions (and the database is finite):

Definition 3.2.3 (Tail-Recursive Program):
A program \( P \) is at most tail-recursive iff for every rule \( A \leftarrow B_1 \wedge \cdots \wedge B_m \) in \( P \) the predicates of \( B_1, \ldots, B_{m-1} \) do not depend on the predicate of \( A \), i.e. no body literal except possibly the last is recursive.

Note that this class of programs is much larger than the class for which the “right recursion optimization” of [Ull89b] is applicable. Since the only requirement is that the recursive literal is the last literal, we could of course make the “same generation”-program tail-recursive by moving the recursive body literal to the end of the rule. But with this sequence of body literals, SLD-resolution becomes very inefficient for a query of the form \( sg(a, Y) \), because \( parent \) is called with both arguments free. So we would not want the behaviour of SLD-resolution in this case. But we would be able to simulate it with guaranteed termination:

Lemma 3.2.4:
Let \( P \) be an at most tail-recursive program and \( Q \) be a query. Then there is an \( n \in \mathbb{N} \), such that for every DB, no goal in the SLD-tree for \( P \cup \{ \leftarrow Q \} \) wrt DB has more than \( n \) literals.

Proof:
In at most tail-recursive programs, a rule can only applied recursively on its last body literal, so iterative recursive rule applications cannot make the stack of “to be proven” literals longer. Therefore, the length (number of literals) of nodes in the SLD-tree is bounded. An upper bound would be the sum of the lengths of all rule bodies.

So if there are only finitely many possible constants occurring in the goals (e.g. because the program contains only finite database predicates), then the number of possible goals is bounded and we can guarantee termination.

Efficiency Comparison
An important difference between SLD-resolution and magic sets is that SLD-resolution passes more context (the continuation) into the sub-computations. For instance, when a predicate \( p \) is called in two different rules, then the two computations for \( p \) are kept disjoint in SLD-resolution, while they are merged in the magic set method. If the arguments for which \( p \) is called are more or less disjoint, then SLD-resolution
is superior, because we do not need to filter the appropriate results later by means of a join. If, however, \( p \) is called for the same arguments at different places, then SLD-resolution repeats the computation, and magic sets are superior (at least if the computation of \( p \) is difficult).

**Example 3.2.5:**
Consider for instance the following program computing “grandparent” relation, where we prefer to use the shorthand \( p \) for \( \text{parent} \):

\[
\begin{align*}
gp(X, Z) & \leftarrow p(X, Y) \land p(Y, Z). \\
p(X, Y) & \leftarrow \text{parent}(X, Y).
\end{align*}
\]

Suppose that the query is \( gp(a, Z) \). The “magic set” transformation leads to the following program:

\[
\begin{align*}
gp(X, Z) & \leftarrow \_\_\_gp^{bf}(X) \land p(X, Y) \land p(Y, Z). \\
m_p^{bf}(X) & \leftarrow \_\_\_gp^{bf}(X). \\
m_p^{bf}(Y) & \leftarrow \_\_\_gp^{bf}(X) \land p(X, Y). \\
p(X, Y) & \leftarrow \_\_\_p^{bf}(X) \land \text{parent}(X, Y). \\
m_gp^{bf}(a).
\end{align*}
\]

(Since there is only one binding pattern for \( gp \) and \( p \), we suppressed the adornments.)

Now if \( \text{parent} \) does not contain \((a, a)\), the two rules about \( \_\_\_p^{bf} \) yield in fact disjoint bindings for \( p \). But these are collected together in one set, and when we get the resulting \( p \)-tupels back, we need two joins to match them with the calling body literal. In total, four joins are needed to evaluate this program (and probably even more, see below).

In contrast, SLD-resolution computes the following sequence of goals (encoded in our \textit{node}-facts):

- \texttt{node(gp(a, Z), [gp(a, Z)])}.
- \texttt{node(gp(a, Z), [p(a, Y) \land p(Y, Z)])}.
- \texttt{node(gp(a, Z), [parent(a, Y) \land p(Y, Z)])}.
- \texttt{node(gp(a, Z), [p(b, Z)])} for every \( b \) with \( \text{parent}(a, b) \).
- \texttt{node(gp(a, Z), [parent(b, Z)])}.
- \texttt{node(gp(a, c), [])} for every \( c \) with \( \text{parent}(b, c) \).

So the results of the calls to \( p \) appear directly at the place where they are needed, no additional filtering is necessary. Only when \( \text{parent} \) is evaluated, a join or a selection is needed, which gives two joins in total. We will see later how the intermediate steps can be optimized away.

If \( \text{parent} \) contained the tuple \((a, a)\), SLD-resolution would repeat the computation of \( p(a, Y) \), once with continuation \([p(Y, Z)]\), and once with continuation \([]\). Of course, since the computation of \( p \) is easy, this is no problem. But if it would take a long time to compute \( p \), magic sets might be better.

The merging of computations done by the “magic set” method can sometimes introduce recursions in the magically rewritten program, and indeed this happens in
the above example ($m\_p^{bf}$ depends on $p$, and $p$ depends on $m\_p^{bf}$). Since the overhead for evaluating recursions is relatively large, doing some duplicate computations during SLD-resolution is probably preferable.

\begin{example}
Another source of multiple contexts for one goal literal are bindings for variables not occurring in this literal. For instance, consider the rule

\[
\text{uncle}(X, Y) \leftarrow p(X, Z) \land \text{brother}(Z, Y),
\]

where $p$ is again a computed parent-relation (e.g. the union of $\text{mother}$ and $\text{father}$). Suppose that $\text{uncle}$ is called with both arguments bound. The magic set method translates this to:

\[
\text{uncle}(X, Y) \leftarrow m\_\text{uncle}^{bb}(X, Y) \land p(X, Z) \land \text{brother}(Z, Y) \land m\_p^{bf}(X).
\]

So when $p$ is called, the given values of $Y$ are projected away, but after $p$ is computed, we must join the result again with $m\_\text{uncle}^{bb}$ in order to get $Y$ back. SLD-resolution retains $Y$ during the computation of $p$, and thus avoids this join. Of course, there is a tradeoff: If there are many different $Y$-values for a single $X$-value, magic sets can be faster.

It is interesting to note that we might get both advantages if we use NF$^2$-relations, and group the $(X, Y)$-tuples by $X$ when we execute $p$. Such a “nesting” operation is not much more expensive than the projection (with duplicate elimination) done by the magic set method. There seem to be many cases in which NF$^2$-relations would allow a more compact representation and fewer copying operations, resulting in a more efficient query evaluation.

While for tail-recursive programs, SLD-resolution is nearly always preferable, we should choose magic sets for other forms of recursion. There we would need further assumptions about acyclicity of database predicates in order to guarantee termination of SLD-resolution. Also partial evaluation would become more difficult, and we will not handle this case below.

Let us conclude this comparison by mentioning some optimizations which our approach yields as a byproduct:

- Duplicate variables in body literals pose no problem, since we always have the complete information about the structure of the goals. In contrast, the magic set method needs an extra “rectification” phase.
- Values for anonymous variables are never represented explicitly.
- Constants from the rules are pushed into the called rules, corresponding to a kind of “compile time” magic set transformation.
- If the given program is non-recursive, the resulting program is also non-recursive.

Of course, each of these optimizations is already known in the field of deductive databases, but they have been introduced separately. Our common framework can explain the deeper reasons behind them.
Adding “Magic Set” Behaviour to the Meta-Interpreter

So both, magic sets and our bottom-up simulation of SLD-resolution have advantages of their own, and it depends on the application, which one is superior. Therefore, we introduce in this subsection a parameterized approach which allows to choose between both possibilities on a single literal basis.

In order to simulate the “magic set” behaviour, we allow that SLD-resolution can call itself recursively for evaluating certain literals. Standard SLD-resolution does a similar thing for negative literals. We believe that this idea of starting a subproof and getting back the explicitly represented result is the key to understand the difference between magic sets and the real SLD-resolution.

For our meta-interpreter, we assume that the user has included literals to be evaluated via magic sets in a special predicate \( \text{call}(\ldots) \) (not otherwise used in the program). Another possibility would be to have “table” declarations for the predicates to be evaluated in subproofs. In fact, our meta-interpreter can also be understood as describing a form of SLD-resolution with tabulation. Of course, the best option would be an intelligent optimizer which automatically determines (guesses) where it is better to use SLD-resolution and where “magic sets” are superior.

Our meta-interpreter is generalized by adding two additional rules to handle \( \text{call}(\ldots) \)-literals. First, such a literal creates a new query:

\[
\text{query}(\text{Lit}) \leftarrow \text{node}(\text{Lit}, \text{call}(\text{Lit})).
\]

So we now construct not a single SLD-tree, but a set of SLD-trees, one for each recursive call. Since every derived \( \text{node} \)-fact contains the root of its tree as the first argument, we can always identify the tree(s) to which a node belongs.

If the subproof started by the above rule is successful, one or more instances of the query are derived in the \( \text{answer} \)-relation. Now the following rule gets these results back into the main proof by resolving the \( \text{call}(\ldots) \)-literal with the derived \( \text{answer} \)-literals:

\[
\text{node}(\text{Query}, \text{Rest}) \leftarrow \text{node}(\text{Query}, \text{call}(\text{Lit}|\text{Rest})), \\
\text{answer}(\text{Lit}).
\]

The complete meta-interpreter is shown in Figure 3.3.

If no IDB-literals is classified as \( \text{call}(\ldots) \), we obviously compute exactly the nodes of the single SLD-tree as in the above meta-interpreter. If all literals are classified as \( \text{call}(\ldots) \), we get something very similar to BRY’s meta-interpreter [Bry90b] (only the representation of conjunctions is a little different). In fact, we will argue below that partial evaluation yields exactly the magic set method with supplementary predicates, where \( \text{query} \)-facts correspond to magic facts, \( \text{answer} \)-facts correspond to derived IDB-facts, and \( \text{node} \)-facts correspond to facts of the supplementary predicates.

Note that the first new rule can also be understood as entering a goal literal into a table \( \text{query} \), and the second rule then allows resolution with proven lemmas from the table \( \text{answer} \). In this way, our meta-interpreter also describes some form of SLD-resolution with tabulation.
/* Initialization (Root Node): */
node(Query, [Query]) ←
    query(Query).

/* SLD-Resolution: */
node(Query, Child) ←
    node(Query, [Lit|Rest]) ∧
    rule(Lit, Body) ∧
    append(Body, Rest, Child).

/* Set up Recursive Call (Derive Magic Literal): */
query(Lit) ←
    node(_, [call(Lit)|.]).

/* Get Result of Recursive Call: */
node(Query, Rest) ←
    node(Query, [call(Lit)|Rest]),
    answer(Lit).

/* Evaluation of DB Literals: */
node(Query, Rest) ←
    node(Query, [Lit|Rest]),
    db(Lit).

/* Turn Proven Query into Answer: */
answer(Query) ←
    node(Query, []).

Figure 3.3: A Meta-Interpreter for SLD-Resolution and Magic Sets

Computation of Node Types

The above meta-interpreter can be executed in deductive database systems which allow bottom-up evaluation with non-ground facts. Of course, the use of lists and non-ground facts significantly decreases the performance. If we are given a Datalog-program consisting of range-restricted rules without function symbols, we of course want that the result of the transformation also has these nice properties. The approach suggested by [Bry90b] is partial evaluation with respect to the given rules and the query. However, simply unfolding the call to the rule-predicate does not help much. The problem is of course the relation node — we need more information about the derivable node-facts in order to continue the partial evaluation. To this end, we have introduced in [Bra95] the notion of a node type. It results from a node-fact by abstracting from the occurring constants and the names of the variables.
Definition 3.2.7 (Node Type):
Let $X_i$ and $C_i$ ($i \in \mathbb{N}_0$) be disjoint sequences of variables. We will call the $C_i$ in the following parameters. Let $a_1, \ldots, a_k$ be the constants appearing in the given program $P$ or query $Q$. A node type $N$ is a node-literal of the form $\text{node}(Q, [A_1, \ldots, A_n])$, where

- $Q$ is a literal and the $A_i$ are possibly call($\ldots$)-enclosed literals.
- $Q$ and the $A_i$ contain only variables from $\{X_0, X_1, \ldots\}$ and constants from $\{a_1, \ldots, a_k\}$.
- If $X_i$ appears somewhere in $N$, then $X_0, \ldots, X_{i-1}$ appear already to the left (and the same for $C_i$).

A node-literal $L$ has type $N$ if there is a mapping $\theta$ from the $C_i$ to constants and from the $X_i$ to variables such that $N\theta = L$ and $\theta(X_i) \neq \theta(X_j)$ for $i \neq j$.

Query types and answer types are defined completely analogously.

Example 3.2.8:
Consider again the “transitive closure” program for computing ancestors. In case of the query $\text{anc}(0, X)$, the following node types can appear:

\[
\begin{align*}
\text{node}(\text{anc}(0, X_0), [\text{anc}(0, X_0)]) \\
\text{node}(\text{anc}(0, X_0), [\text{parent}(0, X_0)]) \\
\text{node}(\text{anc}(0, X_0), [\text{parent}(0, X_1), \text{anc}(X_1, X_0)]) \\
\text{node}(\text{anc}(0, C_0), [\text{true}]) \\
\text{node}(\text{anc}(0, X_0), [\text{anc}(C_0, X_0)]) \\
\text{node}(\text{anc}(0, X_0), [\text{parent}(C_0, X_0)]) \\
\text{node}(\text{anc}(0, X_0), [\text{parent}(C_0, X_1), \text{anc}(X_1, X_0)])
\end{align*}
\]

This set of node types corresponds to the nodes computed in Example 3.2.1. However, these seven node types suffice for any possible parent-relation, since $C_0$ can be any ancestor of person number 0.

Of course, it is possible to replace the constant 0 from the query by a parameter. This is usually done in the magic set method, since in this way the program does not have to be recompiled when the same query is posed later with a different constant. However, if we use the constants from the query and the rules, the resulting program can be slightly faster. \hfill $\blacksquare$

The computation of the node-types can be done by executing the meta-interpreter bottom-up, but with the following differences:

- Since the actual database is unknown at compile time, we use a “generic EDB-fact” $p(C_1, \ldots, C_n)$, when we have to resolve with an EDB-literal. Here the parameters $C_i$ act like variables, but if we have to unify a “real variable” $X_j$ and a parameter $C_i$, we replace $X_j$ by $C_i$ and not vice versa. Thus, we know at what places constants appear.

- When we have derived a literal, we normalize the variables to $X_0, X_1, \ldots$ and the parameters to $C_0, C_1, \ldots$. In this way, we can detect when we have derived the same node type again. It is optionally possible to replace known constants (from the query or the rules) by parameters $C_i$. 

This results in the algorithm of Figure 3.4.

For simplicity, we have only shown how the second rule of the meta-interpreter (performing the SLD-resolution step) is used to generate node types. The application of the other rules is similar. The procedures called in this algorithm do the following:

- **normalize_lit** and **normalize_goal** rename variables and parameters as required in Definition 3.2.7.
- **max_var_index** returns one above the maximal index \( i \) of a variable \( X_i \) occurring in a node type.
- **fresh_var** adds the given offset to all numbers of variables occurring in a rule. This ensures the disjointness of variables before unification.
- **unify** is the standard unification procedure, only variable-to-variable bindings are done in such a way that never a \( C_i \) is replaced by an \( X_j \).
- **append** is the list concatenation.

As explained above, it is possible that longer and longer goals occur during SLD-resolution. In that case, our algorithm will not terminate. However, we can guarantee termination if every recursive body literal, which is not the last literal of the rule, is evaluated in a subproof. (In fact, it suffices if every recursive loop containing a non-tail-recursive literal is “broken” somewhere by requesting a subproof.) As stated in Lemma 3.2.4, in this case the number of nodes in a goal occurring in the SLD-tree is bounded (literals evaluated in a subproof are treated like EDB-literals.). But since node types can contain only the finitely many constants occurring in the program or the query, and since the variables/parameters are normalized, there is only a finite number of possible node types.

**Partial Evaluation**

When we know the structure of the occurring node-, query-, and answer-literals, it is a simple task to partially evaluate the rules of the meta-interpreter. In fact, the specialized rules can already be constructed during the computation of the node types, since this computation simply simulates the rule application.

In the rewritten program, the node types become predicates with the actual values of the parameters \( C_i \) as arguments. In this way, the \( C_i \) and \( X_i \) generalize the usual \( b/f \)-binding patterns. For instance, the node type

\[
\text{node}(\text{anc}(0, X_0), [\text{parent}(C_0, X_1), \text{anc}(X_1, X_0)])
\]

is used as a predicate of arity one (because it contains one parameter). We denote the predicate corresponding to \( \text{node}(Q, [A_1, \ldots, A_n]) \) by \( \text{node}(Q : A_1, \ldots, A_n) \). Then the fact

\[
\text{node}(\text{anc}(0, X_0) : \text{parent}(C_0, X_1), \text{anc}(X_1, X_0)) (3)
\]

corresponds to the node \( \text{parent}(3, Y) \land \text{anc}(Y, X) \) in the SLD-tree with root \( \text{anc}(0, X) \).

It is actually very simple to modify the program shown in Figure 3.4 in such a way that it additionally computes the specialized rule instances. All we need is a proce-
**Input:** Program $P$, Query $Q$.

**Output:** Node Types $\mathcal{N}$, Query Types $\mathcal{Q}$, Answer Types $\mathcal{A}$.

```plaintext
/* Initialization: */
\mathcal{N} := \emptyset; /* No node types so far */
\mathcal{A} := \emptyset; /* No answer types so far */
\mathcal{Q} := \{ \text{normalize}_\text{lit}(Q) \};
\text{changed} := \text{true};

/* Bottom-Up Computation Loop: */
while changed do
    changed := false;
    /* Meta-Interpreter Rule #1: */
    ...
    /* Meta-Interpreter Rule #2: */
    foreach node type $N_1$ in $\mathcal{N}$ do
        let $N_1 = \text{node}(Q, [A_1, \ldots, A_n])$;
        foreach rule $A \leftarrow B$ in $P$ do
            $i := \text{max}_{\text{var}_\text{index}}(N_1)$;
            $A' \leftarrow B' := \text{fresh}_{\text{var}}(i, A \leftarrow B)$;
            $\theta := \text{unify}(A_1, A')$;
            if $\theta \neq \text{nil}$ then
                $G := \text{append}(B', [A_2, \ldots, A_n])$;
                $N_2 := \text{node}(Q\theta, G\theta)$;
                $N_2' := \text{normalize}_\text{node}(N_2)$;
                if $N_2' \notin \mathcal{N}$ then
                    $\mathcal{N} := \mathcal{N} \cup \{N_2'\}$;
                    changed := true;
                fi;
            fi;
        od;
    od;
    /* Further Meta-Interpreter Rules: */
    ...

/* End of Bottom-Up Computation Loop: */
```

Figure 3.4: Algorithm to Compute Node Types
/* Meta-Interpreter Rule #2 (SLD-Resolution): */

```c
foreach node type \( N_1 \) in \( \mathcal{N} \) do
  let \( N_1 = \text{node}(Q, [A_1, \ldots, A_n]) \);
  \( C_1 := \text{node\_params}(N_1) \);
  foreach rule \( A \leftarrow B \) in \( P \) do
    \( i := \text{max\_var\_index}(N_1) \);
    \( A' \leftarrow B' := \text{fresh\_var}(i, A \leftarrow B) \);
    \( \theta := \text{unify}(A_1, A') \);
    if \( \theta \neq \text{nil} \) then
      \( G := \text{append}(B', [A_2, \ldots, A_n]) \);
      \( N_2 := \text{node}(Q\theta, G\theta) \);
      \( C_2 := \text{node\_params}(N_2) \);
      \( N'_2 := \text{normalize\_node}(N_2) \);
      \( \mathcal{R} := \mathcal{R} \cup \{ \text{node}(N'_2)(C_2) \leftarrow \text{node}(N_1)(C_1\theta) \} \);
      if \( N'_2 \notin \mathcal{N} \) then
        \( \mathcal{N} := \mathcal{N} \cup \{ N'_2 \} \);
        \( \text{changed} := \text{true} \);
      fi;
    fi;
  od;
od;
```

Figure 3.5: Specialization of Meta-Interpreter Rule #2

dure \text{node\_params} which constructs an argument list consisting of the parameters \( C_i \) contained in a given node type\(^2\).

As an example, the program to partially evaluate the second rule

```c
/* SLD-Resolution: */

\text{node}(Query, Child) \leftarrow
  \text{node}(Query, [Lit|Rest]) \land
  \text{rule}(Lit, Body) \land
  \text{append}(Body, Rest, Child).
```

of the meta-interpreter is shown in Figure 3.5. There are two points worth noting:

- It is important to select the parameters \( C_1 \) of the given node type \( N_1 \) before the unification with the head literal, and to apply later the unifier \( \theta \). Otherwise, if \( \theta \) replaces a parameter \( C_i \) by a constant \( a \) from the head literal \( A' \), the argument list will not be correctly constructed.

- The selection of the parameters of the resulting node type must be done before the normalization, otherwise we would loose the connection between head and body of the rule.

\(^2\)It has to work also with non-normalized node types, and creates the argument list in the sequence of occurrence, not in the sequence of their indices (of course, every parameter is represented only once in this list).
Example 3.2.9:
Let us for example consider the node type

\[
\text{node}(p(C_1), [q(C_2, C_3, C_1, X_1)])
\]

and the rule (with fresh variables)

\[
q(a, X_2, X_2, X_3) \leftarrow r(X_2, X_3).
\]

The argument list of the given node type is \((C_1, C_2, C_3)\). The unifier \(\theta\) replaces \(C_2\) by \(a\), \(X_2\) by \(C_3\), then \(C_1\) by \(C_3\) (or the other way round), and finally \(X_1\) by \(X_3\) (or vice versa). Before normalization, the resulting "node type" is

\[
\text{node}(p(C_3), [r(C_3, X_3)]).
\]

Its parameter list is \((C_3)\). Normalization yields \(\text{node}(p(C_1), [r(C_1, X_1)])\). So we finally get the following rule:

\[
\text{node}(p(C_1):r(C_1, X_1))(C_3) \leftarrow \text{node}(p(C_1):q(C_2, C_3, C_1, X_1))(C_3, a, C_3).
\]

It is optionally possible to normalize also the argument lists, i.e. to replace as a last step \(C_3\) in the argument lists by \(C_1\). But this is only cosmetics. \(\square\)

Example 3.2.10:
For the transitive closure example, the complete program resulting from our transformation is shown in Figure 3.6. Obviously, some optimizations are needed:

- The first four rules are propositional and can already be evaluated during the compilation phase, since they do not depend on the given data.
- Rules number 7 and 8 are pure copying rules and can be eliminated. It is always advantageous to unfold a call to a predicate \(p\) if this is the only call to \(p\) and there is only one rule about \(p\). In the example, \(\text{node}(\text{anc}(0, X_0):\text{parent}(C_0, X_0))\) is defined only in rule 7 and called only in rule 9, and the same happens with rules 8 and 10.

The state after these optimizations is:

\[
\begin{align*}
\text{node}(\text{anc}(0, C_0):\text{true}) & \leftarrow (C_0) \quad /* 5 */ \\
\text{parent}(0, C_0). & \\
\text{node}(\text{anc}(0, X_0):\text{anc}(C_0, X_0)) & \leftarrow (C_0) \quad /* 6 */ \\
\text{parent}(0, C_0). & \\
\text{node}(\text{anc}(0, C_0):\text{true}) & \leftarrow (C_1) \quad /* 9 */ \\
\text{node}(\text{anc}(0, X_0):\text{anc}(C_0, X_0)) & \leftarrow (C_0) \wedge \\
\text{parent}(C_0, C_1). & \\
\text{node}(\text{anc}(0, X_0):\text{anc}(C_0, X_0)) & \leftarrow (C_1) \quad /* 10 */ \\
\text{node}(\text{anc}(0, X_0):\text{anc}(C_0, X_0)) & \leftarrow (C_0) \wedge \\
\text{parent}(C_0, C_1). & \\
\text{anc}(0, C_0) & \leftarrow (C_0) \quad /* 11 */ \\
\text{node}(\text{anc}(0, C_0):\text{true}) & \\
\end{align*}
\]
3.2. AN IMPROVED MAGIC SET TECHNIQUE

Figure 3.6: Rewritten Program (Without Optimizations)
Now the predicates $node(\text{anc}(0, C_0) : \text{true})$ and $node(\text{anc}(0, X_0) : \text{anc}(C_0, X_0))$ have identical definitions. We can assume that some form of common subexpression elimination detects this. Let us replace the two synonyms by a predicate $\text{anc}_0$.

With the help of these optimizations, we get a very reasonable program for computing the reachable nodes:

$$\begin{align*}
\text{anc}_0(C_0) &\quad \leftarrow \text{parent}(0, C_0). \quad \text{/* 5/6 */} \\
\text{anc}_0(C_1) &\quad \leftarrow \text{anc}_0(C_0) \land \text{parent}(C_0, C_1). \quad \text{/* 9/10 */} \\
\text{anc}(0, C_0) &\quad \leftarrow \text{anc}_0(C_0). \quad \text{/* 11 */}
\end{align*}$$

This seems to be even slightly superior to the standard optimized version of this program [Ull89b, KRS90]:

$$\begin{align*}
\text{m_anc}^{bf}(0). \\
\text{m_anc}^{bf}(Z) &\quad \leftarrow \text{m_anc}^{bf}(X) \land \text{parent}(X, Z). \\
\text{ans}(Y) &\quad \leftarrow \text{m_anc}^{bf}(X) \land \text{parent}(X, Y).
\end{align*}$$

But even without any optimizations, bottom-up evaluation of the transformed program in Figure 3.6 is already as efficient as SLD-resolution in the sense that any applicable rule instance corresponds to an edge in the SLD-tree. And since we have seen how much we really still can optimize, there are good chances that bottom-up evaluation really can beat SLD-resolution. \hfill \square

**Special Case: Magic Sets with Supplementary Predicates**

Let us now consider the special case that all IDB-literals are evaluated in subproofs.

A small inefficiency in the above meta-interpreter is that we first must turn the query into a goal and only then can apply the rules. In order to improve this, we can treat already $\text{query}(\text{Query})$ as representing the root node. This is done by replacing the first rule of the meta-interpreter by the following optimized version:

$$\begin{align*}
&\text{/* Initialization (Select Applicable Rules for Query): */} \\
&\text{node}(\text{Query}, \text{Body}) \leftarrow \text{query}(\text{Query}), \\
&\quad \text{rule}(\text{Query}, \text{Body}).
\end{align*}$$

Now all node types have the form $node(A\theta, [B_1, \ldots, B_n]\theta)$, where

$$A \leftarrow B_1 \land \cdots \land B_n$$

is a rule of the given program and $\theta$ replaces variables, which are bound in the call to $A$ or in $B_1 \land \cdots \land B_{i-1}$, by parameters (and normalizes the variable names). Note that only values of variables occurring in $B_i \land \cdots \land B_n$ or $A$ are represented by parameters. So the arguments correspond exactly to the arguments of the $i$-th supplementary predicate of this rule. Furthermore, if we compare the resulting rules with the magic set transformation of [Ull89b], we see a striking similarity: The rules for the zeroth supplementary predicates are generated by the first rule of our meta-interpreter, the
rules for the other supplementary predicates are generated by the fourth and fifth rule, the rules for the magic predicates are generated by the third rule, and the rules for IDB-predicates correspond to our last rule. To be precise, there are only two very minor differences:

- As presented in [Ull89b], there is no $n$-th supplementary predicate for a rule with $n$ body literals. This would correspond to nodes of the form $\text{node}(A\theta, [])$. It is a simple optimization to eliminate it by combining two rules of our meta-interpreter.
- Second, if two rules have the same head $A$ and body rest $B_1 \land \cdots \land B_n$, we merge the two supplementary predicates. But this can only improve the performance.

Conclusions

This work has tried to clarify the relation between SLD-resolution and magic sets, and has introduced a joint generalization through a meta-interpreter. It was the goal of the magic set transformation to combine all advantages of bottom-up and top-down evaluation (i.e. SLD-resolution). However, this goal was not fully reached, and our approach goes further in this direction.

Our rewriting method allows to decide between SLD-resolution and magic sets for every literal, and choose whatever seems better. If there is only one call of a predicate, or we can see that the arguments are disjoint, or the computation is so simple that it does no harm to repeat it, then we should choose SLD-resolution. Also, the potential savings for tail-recursive literals are so big that they probably outweigh the cost of duplicate computations. On the other hand, recursive calls which are not tail-recursive can only be processed via magic sets.

A first prototype implementation of the computation of node types and the partially evaluated meta-interpreter is available from

ftp://ftp.informatik.uni-hannover.de/software/sldmagic/sldmagic.html

An especially nice feature of our method is that it gives many known optimizations “for free”, namely:

- No extra rectification is needed.
- Values for anonymous variables are never explicitly represented.
- Non-recursive input programs yield non-recursive output programs (if all literals are evaluated via SLD-resolution).
- Already at compile-time, constants are pushed “downward” into the called rules.
- Our method pushes also non-equality conditions, such as $X < 20$ into the called literals. These conditions are represented in the “context” or “continuation” of a literal, and a good selection function would evaluate them as soon as $X$ becomes bound. In the standard magic set transformation, such optimizations are not easy [MFPR90], while in our framework, we only need to choose another selection function. Note that the selection function of SLD-resolution has much more freedom than the SIP-strategy of magic sets, because the SIP-strategy can
choose only locally within the current rule. So \( X < 20 \) is only an example of an “easy to evaluate” literal, we do not need to treat orders in any way special.

- Generalizing this idea, we can check already at compile time the consistency of goals in the generated node types (as far as possible). For instance, nodes containing the two conditions \( X < 20 \) and \( X > 100 \) can immediately be deleted. Also a hierarchical type system could be used to eliminate paths in the SLD-tree which can never yield answers because of inconsistent types for some variable.

At the moment, we have either passed the complete context with the called literal or no context at all. This seemed reasonable, since as soon as we have not the complete context, we need a join to get back to the caller. However, incomplete contexts can still help to delimit the computations within the called literal. For instance, we would like to pass the condition \( X < 20 \) even to a left-recursive literal, where we cannot represent all possible continuations completely.

Let us also note that many deductive database systems have some procedure call mechanism for the predicates, and do not use the pure “magic set” transformation. This leads to some double computations, but avoids the merging of different “calls” to a predicate. Our approach can be understood as going further in this direction.

It seems that future improvements need to look at the internal data structures. We still have considered only a very abstract level and talked about SLD-trees and applicable rule instances. Now the time has come to look at the lower implementation levels. Currently, Prolog implementations are still much faster than deductive databases (for instance, the ECLiPSe Prolog-System is \( 10^{-20} \) times faster than the CORAL system, if the problems mentioned above do not occur). This probably can be improved, or else it would be useful to know the theoretical reasons why set-oriented evaluation is inherently slower than tuple-oriented evaluation (what I do not believe).
Chapter 4

Negation as Failure

In this chapter, we extend pure Datalog by negative body literals of the form \texttt{not }\texttt{B}. The intuitive semantics of \texttt{not }\texttt{B} is “\texttt{B} cannot be proven”, hence the name “negation as failure” (and some Prolog systems use the notation \texttt{\neg B} because of the similarity to \texttt{\neg B}). Note that this negation is very different from the one of classical logic, which means “it can be proven that \texttt{B} is false”. For instance, consider the program \texttt{P_1} := \{\texttt{p \leftarrow not q}\}. In deductive databases and logic programming, it is clear that from this program \texttt{q} cannot be proven (by SLD-resolution or a similar proof-procedure), because there is no rule about \texttt{q}. So \texttt{not q} is true and thus \texttt{p} is implied. Now if \texttt{not} were the negation of classical logic, the program \texttt{P_2} := \{\texttt{q \leftarrow not p}\} would have to be equivalent. However, here the situation is reversed: \texttt{not p} and \texttt{q} are true.

It is impossible to appreciate negation as failure as long as one believes that it really is logical negation, together with perhaps some minimization. The existence of such logically equivalent programs with totally different semantics always calls for “some additional, non-logical information”. I have tried to isolate this in my work on modular default specifications [Bra94b], which were intended to integrate logic programming and some generalization of circumscription. However, not all logic programs can be automatically translated into a modular default specification. So it is at least difficult (if not impossible) to separate logical and additional information if we view \texttt{not} as the negation known from classical logic.

Prolog uses a variant of SLD-resolution to evaluate programs with negation. If it were logical negation, we would be forced to use more powerful theorem proving algorithms, which also look at the contrapositions of the rules. But exactly this is not happening, and it would yield a totally different language. An example of a knowledge representation formalism which uses contrapositions is circumscription. And sometimes these contrapositions are in fact not intended and lead for instance to the “Yale shooting problem”. Of course, it depends on the application, which kind of knowledge representation formalism is better suited.

The well-founded semantics, which is probably the most widely accepted negation semantics, sometimes makes \texttt{p} and \texttt{not p} false at the same time. This is impossible in classical logic, and three-valued logic is quite exotic for many people. Nevertheless, sometimes the same people regard the well-founded semantics as quite reasonable. It is only possible to understand the well-founded semantics in two-valued logic if we do
not insist that \texttt{not} \(p\) is logical negation.

Therefore, we suggest to think of \texttt{not} \(p\) simply as a new predicate, which of course has some relation to \(p\), but is not necessarily the complement. This view has been introduced in [Prz91]. Then it is of course also possible to view \texttt{not} \(p\) as a shorthand for some modal formula, as it was already done in the definition of the stable model semantics [GL88], and later very successfully used in the semantical framework of [Prz95].

Now let us explain why negation as failure is important in deductive databases. First, without negation not every query of relational algebra (or SQL) is expressible in Datalog. And of course, a deductive database system should be an extension of a standard relational system, otherwise there would be no hope that users change to a deductive database.

But even without reference to a particular query language, it seems that every good query language should be closed under certain simple operations, which the user would otherwise apply with a paper-and-pencil method. For instance, it might happen that the user poses one query, then another query, and then he or she wants the set-difference of the two results. It should be possible to specify this in a single query. This also explains why \texttt{not} \(A\) should be the failure to proof \(A\), i.e. the absence in the query result. As long as there is only one model, also logical negation could be used (but the difference then lies in the definition of this model, see above). But for instance the query language of a first-order logic theorem prover is not closed under set-difference, because it might be that neither \(A\) nor \(\neg A\) follows from the given theory (which is in fact very common). By the way, also the well-founded semantics does not satisfy this closure property, which shows that there should be a way to refer to the undefined literals. Also every semantics with multiple models (such as the stable model semantics) fails this test, unless one allows to use the skeptical as well as the credulous view in queries. Nevertheless, such a closure can be seen as a motivation for the introduction of negation as failure, and for the subclass of stratified programs (see below) the goal was reached.

Note that with the closure under set-difference, we have only explained that it should be possible to use \texttt{not} \(A\) (or really \texttt{not}(\(A_1 \land \cdots \land A_n\))) in queries. But another important closure property requires that whatever can be the result of a query, can also be defined as the extension of a predicate (i.e. given a name). Thus, we must allow \texttt{not} also in the bodies of rules\footnote{In fact, with these arguments we have only motivated stratified programs, since they already have the required closure properties. We will explain below why stratified programs are sometimes insufficient.}. This aspect of logic programming rules to give a name to a query result also explains why the rules \(p \leftarrow \texttt{not} \ q\) and \(q \leftarrow \texttt{not} \ p\) are so different.

Now the problem with negation as failure is first, to select the precise semantics of \texttt{not}, and second, to evaluate programs efficiently under this semantics. In order to understand the semantical problems, let us consider the program \(P := \{ p \leftarrow \texttt{not} \ p \}\).

There are (at least) three possible intuitions about this program:

- This program is a paradox. We always view \(\leftarrow\) as \(\leftarrow\), and it is impossible to
say that \( p \) holds iff \( p \) is not provable. So this program is inconsistent. This is the intuition behind the stable models.

- A less drastic solution is to say that this rule makes it impossible to prove that \( p \) cannot be proven — we would get a contradiction if we would make \( \text{not } p \) true. But if \( \text{not } p \) is false, there is no reason why \( p \) should be true. So both are false. This is the view of the well-founded model.

- In classical logic, the above rule is of course equivalent to the fact \( p \) (this is some form of indirect proof). This would make \( p \) true and \( \text{not } p \) false.

So we do not always have a clear intuition, what a program should mean. This is the reason, why there are different possible semantics, at the moment about fifteen semantics have been proposed for Datalog\textsuperscript{neg} (the extension of Datalog by a “negation as failure” operator).

We might be tempted at first to simply define this problem away and single out a subclass of “acceptable” Datalog\textsuperscript{neg}-programs, where such problems do not appear and everybody agrees on the semantics. In fact, this has been done, but it has not led to a completely satisfactory solution.

A milestone in the development of nonmonotonic negation was the introduction of stratified programs, which exclude recursion through negation [VG86, ABW88, Prz88a, Naq89]. This means that the critical example \( P := \{ p \leftarrow \text{not } p \} \) and variants of it are not permissible. And for stratified programs, everybody agrees (at least in the field of deductive databases) that the only sensible semantics is the perfect model [Prz88a]. We will give in Section 4.2 some very simple and intuitive properties, which for stratified programs are only satisfied by the perfect model. So there are also more abstract reasons why the perfect model is so strongly determined.

However, it turned out that in quite a number of practical applications, stratified negation is not sufficient [Tsu91a, ZAO93, KSSD94]. Negation is closely related to aggregation, and one of the first and most often cited examples for aggregation is the “bill-of-materials” problem, which is not stratified. A variant of this problem working with negation has been proposed by Ross [Ros90]: Suppose that we have a complex mechanism which is constructed from a number of components, each of which may itself have smaller components. The mechanism is known to be working either if it has been successfully tested or if all its components are known to be working. I.e. we have a tree of component parts and beginning from the leaves we want to assign to every node an “ok”-label iff all its child nodes already have an “ok”-label or else it is marked as “tested”. This results in the following Datalog\textsuperscript{neg}-program:

\[
\begin{align*}
ok(X) & \leftarrow \text{tested}(X). \\
ok(X) & \leftarrow \text{child}(X,_) \land \text{not has_bad_child}(X). \\
\text{has_bad_child}(X) & \leftarrow \text{child}(X,Y) \land \text{not ok}(Y).
\end{align*}
\]

The construction with \( \text{has_bad_child} \) and the double \( \text{not} \) only encodes a “for all” expression, i.e. all childs \( Y \) are “ok”. It is necessary to check in the second rule that \( X \) really has a child, because otherwise all leaves (and thus all nodes) would be “ok”. The important point about this example is that here the \( \text{child} \)-relation determines the order of evaluation, i.e. we must work from the leaves to the root
because we can evaluate a negative literal like \texttt{not ok}(Y) only to true once we are sure that there is no further possibility to derive \texttt{ok}(Y) later in the proof. Such a dynamic evaluation order is beyond the possibilities of stratified programs.

For the above example there is in fact an equivalent stratified program (we must check that there is no path to a leaf which does not pass through a tested node), but this is less natural to write down and less efficient to evaluate. Furthermore, there are examples for which it is known that there is no equivalent stratified program. An example of this kind is the “winning states” program:

\[
\text{winning}(X) \leftarrow \text{move}(X, Y) \land \text{not winning}(Y).
\]

For example, consider the simplest case of the NIM-game, where a valid move is to take 1 or 2 matches from a pile on the desk. Again, this program can be understood best by working backwards. A pile with 0 matches is certainly not a winning state, because there is no valid move. Once we have \texttt{not winning}(0), we can derive \texttt{winning}(1) and \texttt{winning}(2), but then from state 3 the only valid moves lead to winning states, i.e. the opponent can force to win. So we get \texttt{not winning}(3).

Most modern deductive database systems (like, e.g., CORAL [RSSS94]) allow non-stratified programs, but they require that during runtime no fact depends negatively on itself (as formalized by weak stratification [PP88] and modular stratification [Ros90]). The advantage of this solution is that for such programs, the well-founded and stable (and most other) semantics coincide, so there is still no semantical problem. However, there are other problems:

- The exact definition, which programs are allowed, is quite complicated, and there are subtle variations between different systems or even different versions of the same system.

- The system cannot help the programmer to check whether his or her program is acceptable. It might run with the input data he or she checks, but it might fail in the first presentation to the customer.

- If a program cannot be evaluated with certain data, the error message might not be very good. Furthermore, otherwise in logic programming it is possible to debug a program by posing queries. For instance, if I believe that \texttt{A} should follow from some rule instance \texttt{A} \leftarrow \texttt{B}_1 \land \cdots \land \texttt{B}_n, but it does not, then I can check every \texttt{B}_i in turn. However, if this kind of “dynamic stratification” is violated, the system is not able to answer any further questions\textsuperscript{2}.

In summary, the situation is not very satisfactory. There are also a number of applications which heavily make use of non-stratified programs under the stable model semantics [SZ90, SZ95, CMMT95]. These programs have multiple stable models, and in this way some form of “disjunction” can be formulated.

Therefore, we are interested in semantics and query evaluation algorithms for the complete class of Datalog\textsuperscript{neg}-programs.

In Section 4.1, we define the language Datalog\textsuperscript{neg} formally, and also define the two most important semantics, namely the well-founded and stable models of a program.

\textsuperscript{2}This argument was put forward by François Bry on the Workshop on Logic Programming 1994.
Section 4.2 is devoted to the formal study of negation semantics and their properties. So we try to answer questions like

- Which semantics are good, which are bad? What are possible criteria for a comparison? Some semantics have been defined only on the basis of examples, where they removed unintended behaviour of other semantics. But later it turned out that they behave even worse than the semantics they tried to improve. So how can we be sure that a semantics is somehow sensible?

- Are there possible semantics, which behave good, but have not been discovered yet? Is it simply by chance that we know certain semantics? Or, as a frustrated referee might ask, “How many semantics will still be proposed?”

This shows that we must look at the space of all possible “abstract semantics”. We introduce a nice theoretical framework based on elementary program transformations. We were able to give characterizations of the supported, the stable, and the well-founded semantics. Furthermore, our framework allows the computation of normal forms, which can also be used for query evaluation.

In Section 4.3, we show how to efficiently compute such a normal form, called the residual program, and how to use it for computing the well-founded semantics. The residual program can also be used for computing other semantics, such as the stable model semantics (see Section 5.2). So we describe general principles of computation, which are possible for all semantics satisfying certain properties. It is an important feature of our algorithms that they can be explained on the source program level. We also discuss specialized data structures, which help to speed up the computation, but they are not essential for the correctness of the algorithms. Thus, we believe that our algorithms are easily understandable and might help to understand other algorithms proposed in the literature.

The results presented in Section 4.2 are based on joint work together with JÜRGEN DIX [BD94b, BD95b, BD95a, BD95c]. I have changed the framework a little, and developed at least two new results, but the approach itself as well as the most important theorems have been developed together, and I appreciate JÜRGEN’s help very much. I have also learnt a lot through recent joint work with TEODOR PRZYMUSINSKI [BDP96]. This has at least influenced my treatment of the negation predicates and the presentation of the well-founded semantics.

Overview papers on negation semantics in deductive databases and logic programming are [She88, PP90, AB94, Prz94, Prz95, Dix95a, Dix95b, Dix95c, DF96].
4.1 Datalog with Negations

In this section we formally define an extended language Datalog$^{\text{neg}}$, which allows to use “negation as failure”.

Syntax of Datalog$^{\text{neg}}$

We have motivated above that \texttt{not} is not logical negation, and that a very general and yet simple approach is to view \texttt{not p} as a new predicate [Prz91]. In this way, we can also use much of the theory introduced in Chapter 2 — we formally do not leave the class of Horn programs, we only introduce a new kind of built-in predicates. This also corresponds closely to the approach taken in Prolog, and is the reason why it makes sense to try using variants of SLD-resolution for Datalog$^{\text{neg}}$.

Definition 4.1.1 (Negation Predicates, Positive and Negative Literals):

Let a set \textsc{preds} of predicates be given.

- Let $\text{NEG} := \{(\text{not } p)/n \mid p/n \in \textsc{preds}\}$, where \texttt{not} is an injective function$^3$ from predicate names to $\text{ALPH} - \text{LOG}$ such that $\text{NEG} \cap \textsc{preds} = \emptyset$. Then we define the extended set of predicates as $\textsc{preds}^{\text{neg}} := \textsc{preds} \cup \text{NEG}$.
- We call a literal $L$ positive, iff its predicate is from $\textsc{preds}$, and negative, iff its predicate is from $\text{NEG}$. We write $\text{BASE}^+$ for the set of all positive ground literals and $\text{BASE}^-$ for the set of negative ground literals.
- We call the predicates $p$ and $\text{not } p$ complementary, and we call two literals of the form $p(t_1, \ldots, t_n)$ and $\text{not } p(t_1, \ldots, t_n)$ complementary. For given literal $L$, we denote the complementary literal by $\sim L$. If we know $L$ to be positive, we also write $\text{not } L$ for the complementary literal.

Now Datalog$^{\text{neg}}$-rules are simply Datalog-rules with respect to this extended set of predicates. However, it is impossible to have a negative literal in a rule head. The system has a built-in semantics for “not”, and this cannot be redefined by means of rules. There are approaches allowing another kind of negation, called “strong” or “classical” negation, which can be explicitly defined (see, e.g., [GL91, Jon95]). But this is not the negation as failure which we investigate here. The exclusion of certain predicates from the rule heads is nothing new, we have done this above already for built-in predicates and EDB-predicates.

Definition 4.1.2 (Datalog$^{\text{neg}}$):

- A Datalog$^{\text{neg}}$ rule with respect to $\textsc{preds}$ is a Datalog-rule wrt $\textsc{preds}^{\text{neg}}$, where the head of the rule is a positive literal.
- A Datalog$^{\text{neg}}$-program is a set of Datalog$^{\text{neg}}$-rules. In this chapter, by a (logic) program we mean a Datalog$^{\text{neg}}$-program.

$^3$We need this construction since predicates are required to be elements of the alphabet $\text{ALPH}$. An alternative would be to allow just any set of predicates and not to care about the syntactic representability.
Next, we have to extend the valid binding patterns \( \text{BIND} \) to the negation predicates in order to generalize the notion of range-restriction. Like many other built-in predicates (such as \(<\)\), negation predicates can only be called with all arguments bound. One reason for this is that the extensions of negation predicates are usually infinite. For instance, consider an EDB-predicate \( p \), which is a finite set of tuples over domains such as strings or integers. Obviously the negation \( \text{not} \ p \) should be the complement, and this is infinite.

Furthermore, there are semantical problems if we allow to evaluate non-ground body literals, even if the domain is finite. For instance, consider the following program:

\[
\begin{align*}
p & \leftarrow \text{not} \ q(X). \\
q(a). 
\end{align*}
\]

Does \( p \) follow from this program?

- On the one hand, we can say “yes”, because \( X \) can be bound to some other constant \( b \), and \( q(b) \) obviously cannot be proven. However, it is not very nice to rely on new constants in this way, which have no meaning in the application domain of the database. This was one of the main motivations for the range-restriction. And checking all constants which occur in the database is at least very inefficient and often impossible (if the database uses, e.g., numbers and contains built-in predicates like \( \text{sum} \)).

- On the other hand, we can say that \( p \) is not implied, because the proof of \( q(X) \) succeeds (with \( X \) bound to \( a \)), thus the proof of \( \text{not} \ q(X) \) must fail. This interpretation is easily implementable, but we loose the simple “for all” quantification of all variables over the complete rule. Furthermore, if there were no \( q \)-fact, \( \text{not} \ q \) would succeed without \( X \) bound to a value, so it is not clear what a rule like \( p \leftarrow \text{not} \ q(X) \land \text{not} \ r(X) \) should mean.

As a simple theoretical solution, we just exclude such programs from our considerations in this chapter. However, from experience with practical applications, it seems sensible to allow anonymous variables within negative body literals, treated with the second semantics. This is not really needed, because one can always write a projection rule like \( q' \leftarrow q(X) \), and then replace \( \text{not} \ q(X) \) by \( \text{not} \ q' \). But it is a useful form of “syntactical sugar”, and catches the intuition of the user in most cases.

By the way, in standard Prolog implementations, which allow to execute negative literals containing free variables, it is possible that the two rules \( p \leftarrow q(X) \land \text{not} \ r(X) \) and \( p \leftarrow \text{not} \ r(X) \land q(X) \) assign complementary truth values to \( p \). Deductive databases can choose the evaluation order of the body literals themselves, only restricted by the allowable binding patterns. So we must ensure that negative literals are called with all arguments bound.

**Definition 4.1.3 (Extended Binding Pattern Specification):**

Given a binding pattern specification \( \text{BIND} \) for \( \text{PREDs} \), we define an extended binding pattern specification \( \text{BIND}^{\text{neg}} \) for \( \text{PREDs}^{\text{neg}} \) by

- \( \text{BIND}^{\text{neg}}(p/n) := \text{BIND}(p/n) \) for \( p/n \in \text{PREDs} \), and
- \( \text{BIND}^{\text{neg}}((\text{not} \ p)/n) := \{b \ldots b\} \), where the length of the string is the arity \( n \).
Now the definition of range-restricted rules from Chapter 2 can be applied.

Next, we generalize the predicate dependency graph to programs with negation. If we simply used the definition from Chapter 2 with the new negation predicates, this would not express the obvious dependency of \( \text{not } p \) on \( p \). Thus, we introduce a new graph, where the nodes are only the positive predicates \( \text{PREDS} \), but there are two kinds of edges, positive and negative ones. Using a more standard graph-theoretic notion, this is a directed graph where the edges are labelled “+” or “−” or both.

**Definition 4.1.4 (Extended Predicate Dependency Graph):**
Let \( P \) be a Datalog\(^{neg}\)-program wrt \( \text{PREDS} \). The extended predicate dependency graph of \( P \) is the following graph \( G_{\text{neg}}(P) = (V, E_+, E_-) \):

- \( V := \text{PREDS} \),
- \((p/n, q/m) \in E_+ \) iff \( P \) contains a rule of the form
  \[ p(t_1, \ldots, t_n) \leftarrow L_1 \land \cdots \land L_{i-1} \land q(u_1, \ldots, u_m) \land L_{i+1} \land \cdots \land L_k. \]
- \((p/n, q/m) \in E_- \) iff \( P \) contains a rule of the form
  \[ p(t_1, \ldots, t_n) \leftarrow L_1 \land \cdots \land L_{i-1} \land \text{not } q(u_1, \ldots, u_m) \land L_{i+1} \land \cdots \land L_k. \]

Edges in \( E_+ \) are called positive edges, and edges in \( E_- \) are called negative edges.

**Definition 4.1.5 (Positive and Negative Dependencies):**
Let \( P \) be a Datalog\(^{neg}\)-program wrt \( \text{PREDS} \) and let \( G_{\text{neg}}(P) = (V, E_+, E_-) \) be its extended predicate dependency graph.

- We say that a predicate \( p/n \) depends positively on a predicate \( q/m \) iff there is a path in \( G_{\text{neg}}(P) \) passing only through positive edges.
- A predicate \( p/n \) depends negatively on a predicate \( q/m \) iff there is a path in \( G_{\text{neg}}(P) \) passing through at least one negative edge.

We already mentioned that for a restricted class of Datalog\(^{neg}\)-programs, namely the stratified programs, the semantics is not questionable. The idea of stratification is that it should be possible to compute the complete extension of a predicate before its negation is used to define other predicates. The following definition requires that there is such an evaluation sequence. For finite \( \text{PREDS} \), it is equivalent to require simply that no predicate depends negatively on itself.

**Definition 4.1.6 (Stratified Program):**
A Datalog\(^{neg}\)-program \( P \) wrt \( \text{PREDS} \) is stratified iff there is \( \ell : \text{PREDS} \rightarrow \mathbb{N} \), called a level mapping, such that for every rule \( A \leftarrow L_1 \land \cdots \land L_n \) in \( P \) the following holds:

- \( \ell (\text{pred}(L_i)) \leq \ell (\text{pred}(A)), \) if \( L_i \) is a positive literal,
- \( \ell (\text{pred}(L_i)) < \ell (\text{pred}(A)), \) if \( L_i \) is a negative literal.

The extended predicate dependency graph is also useful for determining the evaluation order in bottom-up query evaluation. We need the following notion:
Definition 4.1.7 (Recursive Clique):
Let a Datalog\textsuperscript{neg}-program \( P \) wrt \( \textsc{PreDS} \) be given, and let \( \mathcal{G}_{\text{neg}}(P) = \langle V, E_+, E_- \rangle \) be its extended predicate dependency graph. The nodes \( C \subseteq \textsc{PreDS} \) of a strongly connected component in \( \langle V, E_+ \cup E_- \rangle \) are called a recursive clique.

Semantical Foundations

Using an extended signature to introduce negation has the advantage that the notions of interpretation and model carry over unchanged from the Datalog-case. But sometimes also the view of three-valued logic is useful. The correspondence is as follows:

<table>
<thead>
<tr>
<th>( p )</th>
<th>not ( p )</th>
<th>“Truth Value”</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>false</td>
<td>“( p ) is true”</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>“( p ) is false”</td>
</tr>
<tr>
<td>false</td>
<td>false</td>
<td>“( p ) is undefined”</td>
</tr>
</tbody>
</table>

Three-valued logic excludes the case that \( p \) and not \( p \) can be true at the same time (this would be the truth value “inconsistent” in a four-valued logic). So we should give interpretations corresponding to three-valued logic a special name. I would have liked to call them “three-valued”, but since all our interpretations are formally two-valued, this seems to be problematic.

Definition 4.1.8 (Semi-Normal Interpretation):
A Herbrand-interpretation \( I \subseteq \textsc{BASE}(\textsc{PreDS}^{\text{neg}}) \) is called semi-normal iff it does not contain complementary literals, i.e. for every \( p/n \in \textsc{PreDS} \):

\[
I[p/n] \cap I[\text{not } p/n] = \emptyset.
\]

Our construction of the well-founded model uses interpretations which are not semi-normal, although the well-founded model itself is of course semi-normal. So also non-semi-normal interpretations can be useful.

Of course, interpretations in which the extensions of \( p \) and \( \text{not } p \) are complements also deserve a special name. We call them “normal”:

Definition 4.1.9 (Normal Interpretation):
A Herbrand interpretation \( I \subseteq \textsc{BASE}(\textsc{PreDS}^{\text{neg}}) \) is called normal iff

\[
A \in I \iff \lnot A \not\in I
\]

for every \( A \in \textsc{BASE}(\textsc{PreDS}) \).

There are different variations of three-valued logic, but the version most often used in logic programming has slightly stricter requirements on models than we get by reusing the notion of a model of two-valued logic. For instance, consider the rule \( p \leftarrow q \). If \( q \) is false, then \( p \) of course can also be false. However, in our approach it might happen that although \( q \) is false, not \( q \) is also false corresponding to the third truth-value “undefined”. Then it is usually expected that \( p \) should also be at least undefined,
i.e. that \textbf{not} \( p \) is false. We call models satisfying this additional requirement “strong models”. Note, however, that in this kind of three-valued logic \( p \leftarrow q \) is not equivalent to \( p \lor \neg q \). This is one of the reasons why we prefer a treatment based on two-valued logic, but we will prove that the models we define as a semantics are really strong models.

**Definition 4.1.10 (Strong Model):**
A Herbrand interpretation \( I \subseteq \text{BASE}(\text{PREDS}^{\text{neg}}) \) is called a strong model of a logic program \( P \) iff it is a model of \( P \) and in addition for every rule \( A \leftarrow L_1 \land \cdots \land L_n \) in \( P \) and every ground substitution \( \theta \) the following holds:

\[
I \not \models \neg (L_i \theta) \text{ for every } i = 1, \ldots, n, \text{ then } I \not \models \neg (A \theta).
\]

It is necessary to require that a strong model is a model, because otherwise the interpretation which makes \( q \) true and \( p \) undefined would be a strong model of \( p \leftarrow q \).

Note that the strong model property gives us some form of contrapositions: A strong model of \( p \leftarrow q \) must satisfy \textbf{not} \( q \leftarrow \textbf{not} \ p \). Since \( p \) and \textbf{not} \( p \) are different predicates, this is not a logical consequence.

Of course, for normal interpretations, the notions of model and strong model coincide:

**Lemma 4.1.11:**
If \( I \) is a model of \( P \), and \( I \) is a normal interpretation, then \( I \) is a strong model.

**Proof:**
Let \( A \leftarrow L_1 \land \cdots \land L_n \) be any rule in \( P \) and \( \theta \) be any ground substitution for this rule. If \( I \not \models \neg (L_i \theta) \) for \( i = 1, \ldots, n \), then \( I \models L_i \theta \) (because \( I \) is normal), and thus \( I \models A \theta \) (because \( I \) is a model), and finally \( I \not \models \neg (A \theta) \) (because \( I \) is normal).

By a minimal model of a Datalog\textsuperscript{neg}-program we mean a minimal model with fixed interpretation of the negation predicates. Otherwise we would always get empty extensions for these predicates. We have discussed this preference relation above when we treated the extensions of the EDB- and built-in predicates as fixed. It is very natural to treat the negation predicates as built-in predicates, and we already excluded them from rule rule heads for this reason.

**Definition 4.1.12 (Minimal Model):**
Let \( I_1 \) and \( I_2 \) be two Herbrand-interpretations wrt \( \text{PREDS}^{\text{neg}} \). We call \( I_1 \) preferable to \( I_2 \) (\( I_1 \preceq I_2 \)) iff

- \( I_1[[\textbf{not} \ p)/n]] = I_2[[\textbf{not} \ p)/n] \) for every \( p/n \in \text{PREDS} \), and
- \( I_1[p/n] \subseteq I_2[p/n] \) for every \( p/n \in \text{PREDS} \).

A minimal model of a Datalog\textsuperscript{neg}-program \( P \) is a model \( I \) of \( P \) such that there is no model \( I_0 \) of \( P \) with \( I_0 \preceq I \) and \( I_0 \neq I \).

Note that for any given interpretation of the negation predicates there is always one and only one minimal model.
Lemma 4.1.13:
For any model $I_1$ of logic program $P$, there is a unique minimal model $I_0$ of $P$ with $I_0 \leq I_1$.

Proof:
$I_0$ is constructed as the intersection of all models $I$ of $P$ with $I \leq I_1$: For every predicate $p/n \in \text{PRED}$, let
\[ I_0[p/n] := \bigcap_{I \models P \text{ and } I \leq I_1} I[p/n], \]
i.e. a positive ground literal $p(c_1, \ldots, c_n)$ is true in $I_0$ iff it is true in every model $I$ of $P$ with $I \leq I_1$. Furthermore let $I_0[\text{not } p/n] := I_1[\text{not } p/n]$. Note that all interpretations $I$ with $I \leq I_1$ agree in the interpretation of the negation predicates.

We now show that $I_0$ is a model of $P$. Suppose this would not be the case. So there were a rule $A \leftarrow L_1 \land \cdots \land L_m$ in $P$ and a ground substitution $\theta$ with $I_0 \models L_i \theta$ ($i = 1, \ldots, m$) and $I_0 \not\models A \theta$. By the construction of $I_0$, there must be a model $I$ of $P$ with $I \leq I_1$ and $I \not\models A \theta$. Furthermore, since $I_0 \models L_i \theta$, it follows that the $L_i \theta$ are true in every model participating in the intersection, and especially in this model $I$. Thus, we get a contradiction, since $I$ also does not satisfy $A \leftarrow L_1 \land \cdots \land L_m$, but was assumed to be a model.

The construction immediately ensures that $I_0$ is minimal: If $I \models I_0$ for a model $I$ of $P$, then also $I \leq I_1$, and $I$ participates in the intersection, so $I_0 \models I$, and thus $I = I_0$.

The uniqueness is also obvious, since any other minimal model $I$ of $P$ with $I \leq I_1$ would again participate in the intersection.

The Supported Model Semantics
Let us now consider three important semantics for Datalog\textsuperscript{neg}-programs: the supported, the stable and the well-founded semantics. As mentioned above, many more semantics have been proposed in the literature, but this selection gives already a good overview of the possibilities. In some sense, the well-founded semantics is the weakest possible semantics, while the stable semantics is very strong (to a degree that it is sometimes inconsistent).

In general, a semantics selects some of the models of every logic program $P$ as the “intended models” of $P$. As in the Horn case, simply selecting all models does not capture the intentions of the user, because more atoms can be true in model than are implied by the program. For programs with negation, also the set of minimal models is too large: We have allowed all possible interpretations of the negation atoms, but the user of course intended some specific interpretation (probably the complement of the truth values of the corresponding positive atoms).

The historically first approach to give programs with negation a semantics was CLARK’s completion [Cla78]. Its intuition is to turn $\leftarrow$ into $\leftrightarrow$, i.e. to require that for every true atom $A$ there is a reason in form of a rule which allows to derive $A$. The details of this syntactic transformation are a little complicated, but fortunately it has a model-theoretic counterpart, the supported models. Supported models have been introduced in [ABW88], and independently (under the name “causal models”) in [BH86].
Defnition 4.1.14 (Supported Model):

$I$ is a supported model of $P$ iff

- $I$ is a normal model of $P$, and
- for every $A \in \text{BASE}^+$ with $I \models A$ there is a rule $A_0 \leftarrow L_1 \land \cdots \land L_n \in P$ and a ground substitution $\theta$ such that $A = A_0\theta$ and $I \models L_i\theta$ for $i = 1, \ldots, n$.

Let $\text{SUPP}(P)$ be the set of supported models of $P$.

Supported models have been used as semantics in logic programming, but for deductive databases they are not suitable, because they depend on tautological rules, which is incompatible with bottom-up evaluation. For instance, the program

$$p \leftarrow p.$$  

has two supported models: $I_1 = \{\text{not } p\}$ and $I_2 = \{p\}$. This corresponds somehow to SLD-resolution, where this rule creates an infinite loop and $p$ is neither proven nor finitely failed. But during bottom-up evaluation, a tautological rule can never yield anything new: We must have $p$ already in order to apply the rule.

Another problem with the supported semantics is that there are programs without supported models, for instance:

$$p \leftarrow \text{not } p.$$  

If $p$ is true, it does not have support, and if $\text{not } p$ is true, $p$ would also have to be true, so the model is not even semi-normal.

We certainly do not suggest to use supported models as a semantics for Datalog$^{\text{neg}}$, but nevertheless they are a useful tool for many proofs. Because of their historic importance, they are also an interesting measure for comparison.

The Stable Model Semantics

The stable model semantics was introduced by Gelfond and Lifschitz [GL88, GL91]. A stable model is simply a minimal model which is normal. Given some interpretation of the negation predicates, we of course want that the IDB-predicates have their minimal extension, i.e. contain only those tuples which they must contain because of the rules. Also, if possible, we would that the negation predicates are interpreted as the complements of their positive counterparts. Both conditions together define which interpretations of the negation predicates would be acceptable (although this gives no direct way to compute them).

Defnition 4.1.15 (Stable Model):

An interpretation $I$ is a stable model of a Datalog$^{\text{neg}}$-program $P$ iff

- $I$ is a minimal model of $P$, and
- $I$ is normal.
Let us consider a few simple examples. The following program is stratified and should pose no problems:

$$p \leftarrow \text{not } q.$$ 

If we start with the interpretation \{\textit{not } p: \text{false}, \textit{not } q: \text{true}\}, we get that \( p \) is true and \( q \) is false in the minimal model \( I \). So \( I \) is normal and thus a stable model.

However, if we start with \{\textit{not } p: \text{true}, \textit{not } q: \text{false}\}, then \( p \) and \( q \) are both false in the minimal model. Since \( q \) and \textit{not } \( q \) are not assigned complementary truth values, this model is not stable.

There are programs which have no stable models. We have already mentioned the standard example, namely the program consisting of the following rule:

$$p \leftarrow \text{not } p.$$ 

Here, if we assume that \textit{not } \( p \) is true, we get that \( p \) is also true, yielding an interpretation which is not even semi-normal. But if we assume that \textit{not } \( p \) is false, we get that \( p \) is also false, which is again not a normal interpretation. So there is no stable model of this program.

Another interesting example is the program consisting of the two rules

$$p \leftarrow \text{not } q.$$  

$$q \leftarrow \text{not } p.$$ 

It has two stable models, based on the following interpretations of the negative literals: \{\textit{not } \( p \): \text{true}, \textit{not } \( q \): \text{false}\} and \{\textit{not } \( p \): \text{false}, \textit{not } \( q \): \text{true}\}. Both assumptions about the negation predicates can “reproduce” themselves, i.e. yield complementary truth values for the positive predicates. So this program has two stable models.

This possibility to have multiple models is useful for a number of applications [SZ90, SZ95, CMMT95]. It gives some way to encode disjunctive information or a nondeterministic choice. However, it also complicates the notions of “query” and “correct answer”. As long as there is a single canonical model of a program, it is clear how to formulate formulas in it (it is in fact nothing else than a relational database). But with multiple models, queries can be answered “skeptically” (true in all models), “credulously” (true in at least one model), or “nondeterministically” (the system somehow chooses one model and sticks to it). In many applications, more or less the complete model is needed, so we do not have to evaluate queries, but only to compute (one or all) stable models.

A goal-directed query evaluation is anyway impossible under the stable semantics, since apparently unrelated rules can change the set of stable models. For instance, consider the program, which is an extension of the last program:

$$p \leftarrow \text{not } q.$$  

$$q \leftarrow \text{not } p.$$  

$$r \leftarrow \text{not } r.$$  

$$r \leftarrow \text{not } p.$$ 

The first rule about \( r \) ensures that \textit{not } \( r \) must be false, otherwise we get an inconsistency. But the rule does not make \( r \) true. Here only the second rule about \( r \) can help,
namely \texttt{not }$p$ must be true in a stable model. So these two rules have eliminated one of the two stable models of the preceding program, although they only define a new predicate $r$. The query $q$ would now be answered with “yes”, but if we look only at the relevant rules for $q$, we get a different answer. This supports our claim that for applications which make really use of the specialities of the stable semantics, we in general need to compute the complete models. In Section 4.3 we given an algorithm for this.

The Well-Founded Semantics

The well-founded semantics was introduced by Van Gelder, Ross, and Schlipf [VGRS88, VGRS91]. In contrast to the stable model semantics, which can yield more than one model or no model at all, the well-founded semantics yields always a unique model. However, this model is in general non-normal, i.e. there can be ground atoms $A$ such that neither $A$ nor \texttt{not} $A$ is true.

Being based on a canonical model makes the well-founded semantics easier to integrate into existing systems. Probably the next generation of deductive database systems will support the well-founded semantics — this is already announced for the XSB system and the LOLA system [Zuk96]. In contrast, the stable model semantics will probably be supported only by smaller prototype implementations for specialized applications. However, most users will not detect the difference, since their programs are (more or less) stratified, and the two semantics agree. Furthermore it seems that when the well-founded model really contains undefined literals, they contain little information and furthermore cannot be queried or further processed. The stable models have much more structure in this case (if they exist). We can see the holes in the truth valuation of a well-founded model as some form of “localized error message”, something which we probably will not use consciously in applications\footnote{An exception is the “winning state” example, where they correspond to games ending in a tie. However, if undefined literals really have a semantics in an application, it is strange that we cannot refer to them in a program.}. However, as we argued before, it is an advantage to assign a semantics to every logic program, even questionable ones, because then debugging via queries becomes possible. Other advantages of the well-founded model are that it can be computed in polynomial time (in contrast to the stable semantics), and that it allows goal-directed query-evaluation.

We already mentioned that the well-founded semantics is a rather weak semantics, which only makes true or false what is absolutely certain. It is defined by an iterative fixpoint computation. At the beginning, nothing is known about the negation predicates. So we have to consider all minimal models with arbitrary interpretations of the negation predicates. We certainly have to make true, what is true in all of these models, and false, what is false in all of these models. But with this knowledge about true and false facts, not all interpretations of the negation predicates remain possible. So in the second round, we have to look only at a subset of the minimal models, and thus get more surely true and false facts (since there are less models, their “intersection” is bigger). This computation continues until a fixpoint is reached. For instance, a stratified program with $n$ strata (predicate levels) needs at most $n$ iterations.
4.1. DATALOG WITH NEGATIONS

This construction of the well-founded model is a bit different from the definition given in [VGRS88, VGRS91]. We have especially used ideas from the definition of the static semantics [Prz95]. It is, however, not difficult to check that the negations of the “greatest unfounded set” [VGRS88] are minimally implied, and vice versa. Our handling of positive facts is symmetric to the negative ones, while in [VGRS88] the immediate consequence operator from Chapter 2 was used, which is much “slower” than the derivation of negative facts via greatest unfounded sets.

Let us now give the formal definitions. The well-founded model is defined by a fixpoint construction in a lattice based on the “information order” between Herbrand interpretations. It considers true and false both “better than” undefined. This is different from the order used to determine minimal models.

Definition 4.1.16 (Information Order):
Let \( I_1, I_2 \subseteq \text{BASE(PREDS}^\text{neg}) \) be Herbrand interpretations. We say that \( I_1 \) has less information than \( I_2 \) (written \( I_1 \sqsubseteq I_2 \)) iff
- \( I_1[p/n] \subseteq I_2[p/n] \) for all \( p/n \in \text{PRED}S \),
- \( I_1[\text{not } p]/n \subseteq I_2[\text{not } p]/n \) for all \( p/n \in \text{PRED}S \).

Least upper bounds and greatest lower bounds are constructed in the usual way by taking the union resp. intersection of the predicate extensions. The bottom element of this lattice, with which we start the computation of the well-founded model, is the interpretation \( I_\bot \) which makes all facts undefined, i.e. \( I_\bot[p/n] = \emptyset \) and \( I_\bot[\text{not } p]/n = \emptyset \) for all \( p/n \in \text{PRED}S \).

Now given some knowledge in form of an interpretation \( I_0 \), which interpretations \( I \) of the negation predicates are still possible? Of course, we would require \( I_0[\text{not } p]/n \subseteq I[\text{not } p]/n \), i.e. what we already know to be false, should be false in every interpretation we consider. On the other hand, what we already know to be true, should not be false, i.e. \( I[\text{not } p]/n \subseteq \text{DOM}^n - I_0[p/n] \), or, equivalently, \( I[\text{not } p]/n \cap I_0[p/n] = \emptyset \). So this constrains the possible interpretations of the negation predicates we have to consider as shown in Figure 4.1. Of course, the more we know, the less interpretations we have to consider, i.e. if \( I_0 \sqsubseteq I_1 \) and \( I \) is possible wrt \( I_1 \), then \( I \) is of course also possible wrt \( I_0 \). So when we gain knowledge during the well-founded computation, we restrict the possible interpretations \( I \) from above and from below.

In fact, it suffices to consider only the two extreme cases, namely the interpretation \( I_\bot \) with \( I_\bot[\text{not } p]/n = I_0[\text{not } p]/n \) and the interpretation \( I_{\text{top}} \) with \( I_{\text{top}}[\text{not } p]/n = \text{DOM}^n - I_0[p/n] \). The reason is that the larger the extensions of the negation predicates are, the larger the extensions of the positive predicates will be in the minimal model (because more rule instances can be applied). So what is surely true in all minimal models \( I \) which we have to consider, is what is true in the minimal model \( I_\bot \) with the smallest possible extensions of the negation predicates. And what is surely false in every minimal model, is what is false in the minimal model \( I_{\text{top}} \) with the largest possible extensions of the negation predicates.

Thus, we get to the following definition of the WFS operator, which given a knowledge state \( I_0 \) produces an improved knowledge state \( I_1 \):
CHAPTER 4. NEGANATION AS FAILURE

Definition 4.1.17 (WFS Operator):
Given a Herbrand interpretation $I_0$, let $T_{P}^{\text{wf}}(I_0)$ be the Herbrand interpretation $I_1$ with

- For all $p/n \in \text{PREDS}$: $I_1[p/n] := I_{\text{bot}}[p/n]$, where $I_{\text{bot}}$ is the minimal model of $P$ with $I_{\text{bot}}[[\text{not } p]/n] := I_0[[\text{not } p]/n]$.
- For all $p/n \in \text{PREDS}$: $I_1[[\text{not } p]/n] := (\text{DOM}^n - I_{\text{top}}[p/n])$, where $I_{\text{top}}$ is the minimal model of $P$ with $I_{\text{top}}[[\text{not } p]/n] := (\text{DOM}^n - I_0[p/n])$.

Lemma 4.1.18:
$T_{P}^{\text{wf}}$ is monotonic wrt the information order, i.e. if $I \subseteq I'$, then $T_{P}^{\text{wf}}(I) \subseteq T_{P}^{\text{wf}}(I')$.

Proof:
Let $I \subseteq I'$.

- So we have $I[[\text{not } p]/n] \subseteq I'[[\text{not } p]/n]$, i.e. $I_{\text{bot}}[[\text{not } p]/n] \subseteq I'_{\text{bot}}[[\text{not } p]/n]$ for all $p/n \in \text{PREDS}$. Since larger extensions of the negation predicates yield larger extensions of the positive predicates, we get $I_{\text{bot}}[p/n] \subseteq I'_{\text{bot}}[p/n]$, i.e.

$$T_{P}^{\text{wf}}(I)[p/n] \subseteq T_{P}^{\text{wf}}(I')[p/n]$$

for all $p/n \in \text{PREDS}$.
- Since $I \subseteq I'$, we also have $I[p/n] \subseteq I'[p/n]$, i.e. $I_{\text{top}}[[\text{not } p]/n] \supseteq I'_{\text{top}}[[\text{not } p]/n]$ for all $p/n \in \text{PREDS}$. Using again that larger extensions of the negation predicates yield larger extensions of the positive predicates, we get $I_{\text{top}}[p/n] \supseteq I'_{\text{top}}[p/n]$, i.e.

$$\left( T_{P}^{\text{wf}}(I) \right)[[\text{not } p]/n] \subseteq \left( T_{P}^{\text{wf}}(I) \right)[[\text{not } p]/n]$$

for all $p/n \in \text{PREDS}$. □
4.1. DATALOG WITH NEGATIONS

Figure 4.2: A Fixpoint of the WFS-Operator

So the least fixpoint of $T_P^{wf}$ exists, and we call it the well-founded model of $P$:

**Definition 4.1.19 (Well-Founded Model):**

The $\Box$-least fixpoint of $T_P^{wf}$ is the well-founded model of $P$.

If $I_0$ is a fixpoint of $T_P^{wf}$, then it is equal to the minimal model $I_{bot}$ occuring in the definition of $T_P^{wf}$. So we immediately get from the definition that the well-founded model is really a model of $P$, and furthermore a minimal model. So if it should be normal, then it is a stable model of $P$. When we set $I_1 = I_0$ in Definition 4.1.17, we get the following characterization of fixpoints of $T_P^{wf}$:

**Proposition 4.1.20 (Fixpoints of WFS Operator):**

A Herbrand interpretation $I$ is a fixpoint of $T_P^{wf}$ iff $I$ is a minimal model of $P$, and furthermore the Herbrand interpretation $\hat{I}$ with

$$\hat{I} \models L :\iff I \not\models \neg L \quad (\text{for all ground literals } L)$$

is also a minimal model of $P$.

In other words, if we assume that all undefined facts in the well-founded model would be false, then we are able to derive that exactly these facts are true. So the situation looks as shown in Figure 4.2. Note also that if $I$ is normal (i.e. there are no undefined facts), then $\hat{I} = I$. So any normal minimal model, i.e. any stable model, is a fixpoint of $T_P^{wf}$. Since the well-founded model is the least fixpoint, every stable model is $\Box$-larger, i.e. makes at least true or false what is true or false in the well-founded model. We already mentioned that if the well-founded model is normal, it is a stable model. But then there can be no $\Box$-larger semi-normal interpretation, thus the well-founded model is the only stable model in this case. All this follows trivially from our definitions:
Proposition 4.1.21 (WFS vs. Stable Models):
Let \( I_{WFS} \) be the well-founded model of \( P \). Then \( I_{WFS} \subseteq I \) holds for every stable model \( I \) of \( P \). Furthermore, if \( I_{WFS} \) is normal, then it is the uniquely determined stable model of \( P \).

From our characterization of the fixpoints of \( T_P^{\text{wf}} \) (Proposition 4.1.20) it is also easy to see that any such fixpoint has the strong model property: Let \( A \leftarrow L_1 \land \cdots \land L_n \) be any rule in \( P \) and \( \theta \) be a ground substitution. When none of the \( \sim(L_i \theta) \) is true in \( I \), this means that every \( L_i \theta \) is true in \( I \). Thus \( A \theta \) is also true in \( I \), but then \( \sim A \theta \) cannot be true in \( I \). In fact, the strong model property simply requires that \( I \) is also a model of \( P \).

Proposition 4.1.22 (Strong Model Property):
Any fixpoint \( I \) of \( T_P^{\text{wf}} \) is a strong model of \( P \).

Finally, we have to check that the well-founded model is semi-normal. This is an easy proof by transfinite induction.

Proposition 4.1.23 (Consistency of Well-Founded Model):
The well-founded model of any program \( P \) is a semi-normal interpretation.

Proof:
- If the input interpretation \( I_0 \) to the operator \( T_P^{\text{wf}} \) is semi-normal (and \( I_\bot \) certainly is), then \( I_{\text{top}}[\text{not}(p)/n] = (\text{DOM}^n - I_0[p]) \supseteq I_0[\text{not}(p)/n] \) for all \( p/n \in \text{PRED} \). Since \( I_{\text{bot}}[\text{not}(p)/n] = I_0[\text{not}(p)/n] \subseteq I_{\text{top}}[\text{not}(p)/n] \) for all \( p/n \in \text{PRED} \), it follows that \( I_{\text{bot}}[p/n] \subseteq I_{\text{top}}[p/n] \) for all \( p/n \in \text{PRED} \). By definition of \( I_1[\text{not}(p)/n] \), it is the complement of \( I_{\text{top}}[p/n] \), thus also \( I_1[p/n] = I_{\text{bot}}[p/n] \) and \( I_1[\text{not}(p)/n] \) are disjoint.
- For a limit ordinal \( \alpha \), we know that all \( T_P^{\text{wf}} \uparrow \beta \) for all \( \beta < \alpha \) are semi-normal and have to show that \( \text{lfp}(T_P^{\text{wf}} \uparrow \beta) \) is semi-normal. Suppose it were not. Then there would be \( p/n \in \text{PRED} \) and a tuple \( \tau \) with \( \tau \in \text{lfp}(T_P^{\text{wf}} \uparrow \beta)[p/n] \) and \( \tau \in \text{lfp}(T_P^{\text{wf}} \uparrow \beta)[\text{not}(p)/n] \). This means that there are \( \beta_1, \beta_2 < \alpha \) such that \( \tau \in T_P^{\text{wf}} \uparrow \beta_1[p/n] \) and \( \tau \in T_P^{\text{wf}} \uparrow \beta_2[\text{not}(p)/n] \). But then \( T_P^{\text{wf}} \uparrow \max(\beta_1, \beta_2) \) is already not semi-normal, contradicting the induction hypothesis.

Let us again consider a few examples. We start with the stratified program consisting of the single rule

\[
p \leftarrow \text{not} \ q.
\]

In the first round, we do not know whether \( \text{not} \ q \) is true or false, so there is a minimal model in which \( p \) is true, and one in which \( p \) is false. But in both minimal models, \( q \) is false. So in the next round, we already know that \( \text{not} \ q \) is true, and can conclude that \( p \) is true.

Next, let us consider the standard example for a program without stable models:

\[
p \leftarrow \text{not} \ p.
\]
If \textbf{not} \(p\) is true, \(p\) is true in the minimal model, and if \textbf{not} \(p\) is false, \(p\) is false in the minimal model. So we do not get any new information, and have reached the fixpoint in which both \(p\) and \textbf{not} \(p\) are false (i.e. \(p\) is “undefined”).

The well-founded semantics is sometimes criticized because it is not able to do a case analysis on undefined literals. This can be demonstrated by the following example:

\[
\begin{align*}
p & \leftarrow \textbf{not} \ p. \\
q & \leftarrow \ p. \\
q & \leftarrow \textbf{not} \ p.
\end{align*}
\]

In the well-founded model of this program, \(q\) is undefined. This is not astonishing, because \(p\) is undefined, and thus neither \(p\) nor \textbf{not} \(p\) are true. This also supports our claim that the undefined truth value can be seen as some form of error message, and of course the error message is inherited to literals depending on the problematic literal \(p\).

Sometimes, a third truth value is used when one really has multiple two-valued models and wants to “represent” them in a single three-valued model (which of course usually is a loss of information). But the well-founded semantics cannot be understood in this way: The program \(\{p \leftarrow \textbf{not} \ p\}\) has a unique two valued model (in which \(p\) is true), but this is different from the well-founded model.

Finally, the following example shows that the converse of the second part of Proposition 4.1.21 does not hold. It is possible that there is only a single stable model, but this is not equal to the well-founded model:

\[
\begin{align*}
p & \leftarrow \textbf{not} \ q. \\
q & \leftarrow \textbf{not} \ p. \\
r & \leftarrow \textbf{not} \ r. \\
r & \leftarrow \textbf{not} \ p.
\end{align*}
\]

We explained above that this program has a single stable model, in which \(q\) and \(r\) are true and \(p\) is false, but in the well-founded model of this program all three literals are undefined.

\section*{Instantiation of Programs}

It is quite usual in the literature to define semantics only for propositional programs and to “lift” this definition to arbitrary programs by replacing the variables by elements from the Herbrand universe \(\text{DOM}\) in all possible ways. Of course, the resulting programs can in general be infinite.\par

Having to consider only propositional programs sometimes drastically simplifies the definitions and proofs by removing a lot of technical overhead, and thus allows to concentrate on the essential ideas. So although propositional programs are too restricted for most applications, they are very useful for developing a theory of logic programming, and via this “instantiation”, the theoretical results can be applied to practical programs. We have seen already several times that different views on the same subject can be very helpful. And seeing a program only as a shorthand to denote
its ground instantiation is very convenient for semantical investigations. After all, the definitions of a Herbrand model and the $T_P$-operator also refer only to the ground instances of the rules, not to the rules themselves.

**Definition 4.1.24 (Ground Instantiation):**

The ground instantiation $P^*$ of a program $P$ is

$$P^* := \{ A \leftarrow L_1 \theta \land \cdots \land L_n \theta \mid A \leftarrow L_1 \land \cdots \land L_n \text{ is a rule in } P \text{ and } \theta \text{ is a ground substitution for it} \}.$$  

As noted above, it is immediate from the definitions that the “is model of”-relationship is invariant under instantiation. So, as long as we are interested only in Herbrand models, we do not lose something by instantiating the program. Of course, this also applies to the minimal model and all other notions based on Herbrand models.

**Proposition 4.1.25 (Instantiation and Models):**

A Herbrand interpretation $I$ is a (minimal, stable, well-founded) model of $P$ iff it is a (minimal, stable, well-founded) model of $P^*$.

Note, however, that this works only as long as it suffices to consider Herbrand models. If we wanted to conclude non-ground facts like $p(X)$ from a program $P$, we would have to consider also non-Herbrand models of $P$. For instance, $p(X)$ would follow from $P_1 := \{ p(X) \}$, but not from $P_2 := \{ p(a) \}$, although both have the same Herbrand model and ground instantiation if $\text{DOM} = \{ a \}$. It is a well-known theorem from logic that it in fact suffices to consider only Herbrand models if we check the implication by testing the satisfiability of $P \cup \{ \neg p(c) \}$ with a new constant $c$. So the difference is that we treat $\text{DOM}$ as fixed here. For this reason, [VGRS88] have introduced sufficiently many “dummy terms” in their construction of the well-founded model.

But when we consider only strictly range-restricted programs, and are interested in minimal models, we have already seen that it suffices to look only at the active domain of the program (see Proposition 2.2.27). No matter how much we would extend $\text{DOM}$, the minimal model is unchanged. This also shows that it is never possible that a strictly range-restricted program implies a literal containing variables. So for strictly range-restricted programs and notions based on minimal models (like the well-founded and stable models) it is justified to concentrate on Herbrand models, and then taking the ground instantiation makes no difference.

**Proposition 4.1.26 (Instantiation and Models II):**

A Herbrand interpretation $I$ is a minimal/stable/well-founded model of $P$ iff it is a minimal/stable/well-founded model of $\text{ground}(P)$, the ground instantiation of $P$ restricted to the active domain of $P$.

This has also the practical advantage that the active domain is often finite, in which case $\text{ground}(P)$ is also finite (if $P$ itself is finite). We will need this finiteness in Section 4.2.
Of course, built-in predicates like $<$ or $\text{sum}$ immediately lead to an infinite “active domain” according to Definition 2.2.24. However, the set of values which can ever be touched during terminating bottom-up evaluation is certainly finite. We can, for instance, assume that the built-in predicates have been restricted to the “really needed” tuples in order to get a finite ground instantiation. This is not very elegant, and a formalization of it is either very operational or partially cyclic (by referring to a minimal model).

So maybe it is clearer and more honest simply to say that we have nice, elegant, and simple results for finite propositional programs, and essential parts of them do not hold for infinite programs. And we believe that the finite case is very relevant from a more practical viewpoint.
4.2 A Framework for Studying Semantics

Above, we presented two important semantics for the class of Datalog\textsuperscript{neg}-programs: The stable models and the well-founded model. However, about fifteen semantics have been proposed in the literature. So we should look at the space of all possible “abstract semantics” and try to classify them by means of their properties. This is the subject of this section.

In this section, we mean by a logic program always a finite and variable-free (ground) program. It is very convenient to analyze negation semantics only for propositional programs. This is again the general principle to separate the problems. The results can be lifted to range-restricted Datalog-programs by considering the ground instantiation (see above). This approach is very common in the literature.

For technical reasons, we must assume that there is always one further ground atom, i.e. that the set $BASE^+$ is infinite.

Abstract Semantics

Many concrete semantics have been defined by selecting either a specific model of every logic program (such as the well-founded model) or a set of models (such as the stable models). Furthermore, also semantics which define a program completion $\text{comp}(P)$, simply select a subset of the models of $P$ by adding formulas to $P$. So it seems that the following definition is very wide:

Definition 4.2.1 (Semantics):
A semantics is a mapping $S$, which assigns to every ground program $P$ a set $S(P)$ of interpretations, such that for every $P$ and every $I \in S(P)$:

- $I$ is a model of $P$, and
- if $A \in BASE^+$ does not occur in $P$, then $I \not\models A$ and $I \models \text{not } A$.

We call $S(P)$ the semantics of the program $P$. If $S(P_1) = S(P_2)$, we call $P_1$ and $P_2$ equivalent under $S$.

The constraint that ground atoms $A$ which do not occur in $P$ are interpreted as false ensures that there are only finitely many different semantics of a given program.

Next, the following main classes of semantics seem to be interesting:

Definition 4.2.2 (Classes of Semantics):

- A semantics $S$ is called normal/semi-normal iff the semantics $S(P)$ of every program $P$ is a set of normal/semi-normal interpretations.
- A semantics $S$ has the strong model property iff for every program $P$, its semantics $S(P)$ is a set of strong models of $P$.
- A semantics $S$ is called consistency-preserving iff $S(P) \neq \emptyset$ for every program $P$.
- A semantics $S$ is called definite iff $S(P)$ consists of exactly one interpretation (for every program $P$).
We have defined a semantics as a mapping from programs to sets of models. However, some semantics can be defined in a more uniform way as a selection function from sets of models to subsets. Such functions and their properties have been studied in social choice theory [Mou85], which lies on the border of mathematics, economics, and social sciences. It investigates the dependence of decisions on the candidate set. For instance, given any set of possible computers to buy, I can select a subset of “good” computers (within this subset I have no specific preferences). The set from which I can choose depends on how much money I have and other external circumstances.

Now suppose that a computer which I considered “bad” (i.e. I have not chosen it) is no longer produced. Then if my selection function behaves “logically”, the set of chosen computers should not change if I exclude a non-chosen computer from the set of possibilities. Such and other properties are studied in social choice theory, and I have shown in [Bra90] that there is in fact a strong connection to non-monotonic reasoning (The above property is equivalent to cumulation, and was introduced in social choice theory long before cumulation was considered in non-monotonic reasoning.).

We cannot continue this discussion here, but it is obviously very useful if a semantics looks only at the logical contents of the given program and not at its syntax:

**Definition 4.2.3 (Equivalence-Preserving Semantics):**

A semantics $S$ is equivalence-preserving iff $S(P_1) = S(P_2)$ for all programs $P_1, P_2$ which are logically equivalent, i.e. satisfying

$$I |= P_1 \iff I |= P_2.$$ 

Note that the equivalence used in the above definition is not the equivalence of classical logic, since we allow to assign $A$ and $\text{not } A$ independent truth values. For instance, the programs $P_1 := \{ p \leftarrow \text{not } q \}$ and $P_2 := \{ q \leftarrow \text{not } p \}$ are not considered as equivalent here, although they of course would be equivalent in classical logic. In fact, no sensible logic programming semantics would be equivalence-preserving with respect to classical logic, whereas for instance the stable and the well-founded semantics have the above property.

A slightly stronger property, also satisfied by the stable and the well-founded semantics, is that only the minimal models of the given program are important for selecting the intended models. This gives us even more freedom for changing the syntax of the program without changing its semantics. For instance, then the important “unfolding” transformation (see below) becomes possible.

**Definition 4.2.4 (Strictly Equivalence-Preserving Semantics):**

A semantics $S$ is strictly equivalence-preserving iff $S(P_1) = S(P_2)$ for all programs $P_1$ and $P_2$ such that for all interpretations $I$:

$$I \text{} \text{is a minimal model of } P_1 \iff I \text{ is a minimal model of } P_2.$$ 

Of course, if a semantics $S$ is strictly equivalence-preserving, it is also equivalence-preserving.
Program Transformations

In order to analyze a given abstract semantics, we require that it allows certain elementary program transformations, i.e. has a good (surprise-free) “behaviour” when we change the input program a little. For instance, we would expect that deleting a tautological rule like

\[ p \leftarrow p \land q \land \text{not } r \]

does not change the semantics of a given program. First, deleting a tautology is a logical equivalence transformation. Second, if we do bottom-up query evaluation, such a rule is obviously useless, because we must have \( p \) already before we can apply it, so it can never yield anything new. However, if we do top-down query evaluation, a tautological rule can lead to an infinite loop. So semantics which try to model SLD-resolution very closely (such as the supported models) depend on the existence of tautological rules. Thus, it is an interesting property of a semantics whether it allows the deletion of tautological rules or not.

Formally, a program transformation is a relation on logic programs. It is not a mapping, because there are programs in which the transformation is not applicable (for instance, programs containing no tautologies), and there are programs, in which the result of the transformation is not uniquely determined (e.g., programs containing more than one tautology). The deletion of tautologies could be made a mapping by simply deleting all tautologies, but deleting a single tautology is a more elementary concept. Furthermore, there are transformations (e.g. the unfolding below), where the sequence of the single operations is important and termination could not be guaranteed if we simply iterate it as long as possible.

**Definition 4.2.5 (Program Transformation):**

- A program transformation is a relation \( \triangleright \) between ground programs.
- A semantics \( S \) allows a program transformation \( \triangleright \) iff for all ground programs \( P_1 \) and \( P_2 \): If \( P_1 \triangleright P_2 \), then \( S(P_1) = S(P_2) \), i.e. \( P_1 \) and \( P_2 \) are equivalent under \( S \).

We now introduce seven specific program transformations, which we later use to analyze and characterize semantics. They also allow to compute normal forms of programs which are very useful for semantical characterizations as well as for query evaluation.

First, we already motivated that a good semantics amenable to bottom-up query evaluation should allow to delete tautologies:

**Definition 4.2.6 (Deletion of Tautologies):**

Program \( P_2 \) results from program \( P_1 \) by deletion of tautologies \( (P_1 \triangleright_T P_2) \) iff there is \( A \leftarrow L_1 \land \cdots \land L_n \in P_1 \) such that \( A \in \{L_1, \ldots, L_n\} \) and \( P_2 = P_1 - \{A \leftarrow L_1 \land \cdots \land L_n\} \).

**Lemma 4.2.7:**

If a semantics \( S \) is equivalence-preserving, it allows the deletion of tautologies.
4.2. A FRAMEWORK FOR STUDYING SEMANTICS

Proof:
This is trivial: Let $P_1 \rightarrow_T P_2$. Every interpretation $I$ is a model of a tautological rule, so $I \models P_1 \iff I \models P_2$. But then $S(P_1) = S(P_2)$ since $S$ is equivalence-preserving. \qed

Another very important program transformation is to “unfold” a body literal. For instance, consider the rule

$$p \leftarrow q \land \text{not } r.$$

Suppose that there are two rules about $q$, namely

$$q \leftarrow s_1 \land \text{not } t_1,$$

$$q \leftarrow s_2 \land \text{not } t_2.$$

Then during bottom-up evaluation, the only possibility to derive $q$ is by applying one of these two rules. So we can replace the body literal $q$ by the bodies of the rules about $q$:

$$p \leftarrow s_1 \land \text{not } t_1 \land \text{not } r.$$

$$p \leftarrow s_2 \land \text{not } t_2 \land \text{not } r.$$

This is a very powerful transformation, as we shall see.

However, we must note that it is not a logical equivalence transformation. Of course, it is no problem to add the combined rules, but deleting the original rule

$$p \leftarrow q \land \text{not } r$$

would allow more models (because classical logic does not enforce that $q$ is only true if one of its rule bodies is true). So it is not surprising that “unfolding” has a strong connection to supported and minimal models (see below).

The formal definition of unfolding is a little lengthy, but not complicated:

Definition 4.2.8 (Unfolding):
Program $P_2$ results from program $P_1$ by unfolding $(P_1 \rightarrow_U P_2)$ iff

- there is a rule $A \leftarrow L_1 \land \cdots \land L_n \in P_1$, and an atom $B$ such that $L_i$ is the positive literal $B$,
- and

$$B \leftarrow M_{i,1} \land \cdots \land M_{1,m_1}$$

$$\vdots$$

$$B \leftarrow M_{k,1} \land \cdots \land M_{k,m_k}$$

are all rules in $P_1$ about $B$,

- and

$$P_2 = P_1 - \{ A \leftarrow L_1 \land \cdots \land L_{i-1} \land B \land L_{i+1} \land \cdots \land L_n \}$$

$$\cup \{ A \leftarrow L_1 \land \cdots \land L_{i-1} \land M_{i,1} \land \cdots \land M_{1,m_1} \land L_{i+1} \land \cdots \land L_n, \}$$

$$\vdots$$

$$A \leftarrow L_1 \land \cdots \land L_{i-1} \land M_{k,1} \land \cdots \land M_{k,m_k} \land L_{i+1} \land \cdots \land L_n \}.$$
Lemma 4.2.9:
If a semantics $S$ is strictly equivalence-preserving, it allows unfolding.

Proof:
We show that unfolding does not change the set of minimal models, i.e. that if $P_1 \equiv^U P_2$, then $P_1$ and $P_2$ have the same minimal models. If $S$ is strictly equivalence-preserving, this implies $S(P_1) = S(P_2)$.

In order to show that $P_1$ and $P_2$ have the same minimal models, we first show that a minimal model of $P_1$ is also a model of $P_2$ and vice versa:

- Let $I$ be a model of $P_1$ (we do not need the minimality in this direction). Since the combined rules are logical consequences of the old rules, $I$ is also a model of $P_2$.
- Let $I$ be a minimal model of $P_2$. The only rule of $P_1$ which it could possibly violate is the deleted rule

  $$A \leftarrow L_1 \land \cdots \land L_{i-1} \land B \land L_{i+1} \land \cdots \land L_n.$$ 

  Suppose that $I$ would violate this rule. Then every $L_j$ and $B$ were true in $I$, but $A$ were false. Consider the interpretation $I_0$ with $I_0 \not= B$, but otherwise agrees with $I$ (i.e. $I_0 \models L \iff I \models L$ for all literals $L$ except $B$). Since $I_0 \not= I$, $I_0 \not= I$, and $I$ is a minimal model of $P_2$, $I_0$ cannot be a model of $P_2$. Since $I_0$ differs only in the truthvalue of $B$ from the model $P_2$, it must violate a rule with $B$ in the head. This rule must already be contained in $P_1$, if $A = B$, the above rule would be a tautology and could not be violated in $I$. So the rule which $I_0$ violates is one of the rules about $B$ in $P_1$:

  $$B \leftarrow M_{j,1} \land \cdots \land M_{j,m_j}.$$ 

  This means that $M_{j,1}, \ldots, M_{j,m_j}$ are true in $I_0$ and thus in $I$. But then $I$ violates the corresponding combined rule

  $$A \leftarrow L_1 \land \cdots \land L_{i-1} \land M_{j,1} \land \cdots \land M_{j,m_j} \land B \land L_{i+1} \land \cdots \land L_n.$$ 

  This is impossible, since $I$ was assumed to be a model of $P_2$.

Now we show that the minimal models agree:

- Let $I$ be a minimal model of $P_1$. We have shown above that it is a model of $P_2$. If there were a smaller model $I'$ of $P_2$, there would be also a minimal model $I''$ of $P_2$ with $I'' \preceq I'$ (by Lemma 4.1.13). Thus, we also have $I'' \preceq I$ and $I'' \not= I$. But as shown above, $I''$ is a model of $P_1$, which contradicts the assumed minimality of $I$.
- The other direction is completely analogous (with $P_1$ and $P_2$ exchanged).

The next transformation which helps to classify semantics is the deletion of contradictions. In classical two-valued logic, a rule like

$$p \leftarrow q \land \textbf{not} \ r$$

would also be a tautology: The condition $r \land \textbf{not} \ r$ is always false, so the rule can never be applied, and thus is never violated. However, if we consider non-normal strong models, it is possible that the body is “undefined”, and thus $p$ must also be at least “undefined” (i.e. $\textbf{not} \ p$ may not be true). So not all semantics will allow the deletion of contradictions. In fact, we can prove that (together with the strong model property) it already implies that the semantics is normal. But for normal semantics, it is a very natural property:
4.2. A FRAMEWORK FOR STUDYING SEMANTICS

Definition 4.2.10 (Deletion of Contradictions):
Program $P_2$ results from program $P_1$ by deletion of contradictions ($P_1 \leftarrow^c P_2$) iff there is a rule $A \leftarrow L_1 \land \cdots \land L_n \in P_1$ and there are $i, j \in \{1, \ldots, n\}$ such that $L_i = \neg L_j$ and $P_2 = P_1 - \{A \leftarrow L_1 \land \cdots \land L_n\}$.

Lemma 4.2.11:
If $S$ has the strong model property and $S$ allows the deletion of contradictions, then $S$ is a normal semantics.

Proof:
Let $P$ be any logic program, and $I \in S(P)$. Let $\bot \in BASE^+$ be an atom which does not occur in $P$ (such an atom exists because logic programs are finite and $BASE^+$ is infinite). By the definition of a semantics, we know that $I \models \text{not} \bot$. Now suppose that there were $A \in BASE^+$ with $I \not\models A$ and $I \not\models \text{not} A$. Consider the program $P' := P \cup \{\bot \leftarrow A \land \text{not} A\}$. Since $I$ is not a strong model of the new rule, $I \notin S(P')$. But obviously we can get $P$ from $P'$ be the deletion of contradictions, i.e. $P' \leftarrow^c P$, so that $S(P) = S(P')$, since $S$ allows this transformation. But this is a contradiction to $I \in S(P)$. \hfill \Box

The next two transformation describe the evaluation of negative body literals in trivial cases. For instance, if there the is no rule with $p$ in the head, there is obviously no way to derive $p$ by bottom-up evaluation, so $\text{not} p$ should be true. But then it should be possible to delete the condition $\text{not} p$ from the body of a rule, since it is anyway true:

Definition 4.2.12 (Positive Reduction):
Program $P_2$ results from program $P_1$ by positive reduction ($P_1 \leftarrow^p P_2$) iff there is a rule $A \leftarrow L_1 \land \cdots \land L_n \in P_1$ and $i \in \{1, \ldots, n\}$ such that

- $L_i$ has the form $\text{not} B$,
- there is no rule about $B$ in $P_1$, and
- $P_2 = P_1 - \{A \leftarrow L_1 \land \cdots \land L_n\} \cup \{A \leftarrow L_1 \land \cdots \land L_{i-1} \land L_{i+1} \land \cdots \land L_n\}$.

On the other hand, if $p$ is given as a fact in the program, a condition of the form $\text{not} p$ can never be true (if the semantics yields only semi-normal interpretations). So a rule with $\text{not} p$ in its body is useless and it should be possible to delete it:

Definition 4.2.13 (Negative Reduction):
Program $P_2$ results from program $P_1$ by negative reduction ($P_1 \leftarrow^N P_2$) iff

- there is a rule $A \leftarrow L_1 \land \cdots \land L_{i+1} \land \text{not} B \land L_{i+1} \land \cdots \land L_n$ in $P_1$, and
- $B \leftarrow \text{true} \in P_1$, and
- $P_2 = P_1 - \{A \leftarrow L_1 \land \cdots \land L_{i+1} \land \text{not} B \land L_{i+1} \land \cdots \land L_n\}$.

Negative reduction is an equivalence transformation of classical logic, however, it is not an equivalence transformation as long as we consider also non semi-normal interpretations, which can make $B$ and $\text{not} B$ true at the same time.
The next transformation, the deletion of subsumed rules, preserves again logical equivalence. The idea is that we can delete “logically weaker” rules. Consider for instance the two rules
\[
\begin{align*}
  p & \leftarrow q \land r \land \text{not } s \land \text{not } t. \\
  p & \leftarrow q \land \text{not } s.
\end{align*}
\]
Whenever the first rule is applicable (i.e. each body literal is satisfied), also the second rule is applicable. So we should be able to delete the first rule, because it does not give us any additional information. In the following, we formalize this by requiring that the body of the weaker rule contains all literals of the stronger rule in the same sequence:

**Definition 4.2.14 (Subsumed Rule):**

We call a rule \( A \leftarrow L_1 \land \cdots \land L_n \) subsumed by a rule \( B \leftarrow M_1 \land \cdots \land M_m \) iff \( A = B \), \( m < n \), and there are \( 1 \leq i_1 < i_2 < \cdots < i_m \leq n \) with \( M_j = L_{i_j} \) for \( j = 1, \ldots, m \).

**Definition 4.2.15 (Deletion of Subsumed Rules):**

Program \( P_2 \) results from program \( P_1 \) by deletion of subsumed rules \( (P_1 \rightarrow S P_2) \) iff
- there is a rule \( A \leftarrow L_1 \land \cdots \land L_n \) in \( P_1 \) which is subsumed by another rule in \( P_1 \), and
- \( P_2 = P_1 - \{ A \leftarrow L_1 \land \cdots \land L_n \} \).

A more powerful transformation would be to require that the set of body literals of the stronger rule is a subset of the body literals of the weaker rule. But for our characterizations, the above transformation suffices, and the characterizations are the stronger, the weaker the used transformations are.

However, for query evaluation algorithms, it is useful to throw away as many rules as possible. Therefore we also introduce the following transformation, which formalizes that the order and multiplicity of body literals are not important, so that the rule bodies can be treated as sets. This is very common in deductive databases, where the optimizer has the freedom to change the order of body literals, if this allows a more efficient evaluation.

**Definition 4.2.16 (Normalization of Rule Bodies):**

Let \( < \) be any fixed linear order on the set of literals. Program \( P_2 \) results from program \( P_1 \) by normalization of rule bodies \( (P_1 \rightarrow_B P_2) \) iff there is a rule \( A \leftarrow L_1 \land \cdots \land L_n \) in \( P_1 \) and a rule \( A \leftarrow M_1 \land \cdots \land M_m \) in \( P_2 \) such that
- \( \{L_1, \ldots, L_n\} = \{M_1, \ldots, M_m\} \), but \( L_1 \land \cdots \land L_n \neq M_1 \land \cdots \land M_m \),
- \( M_1 < M_2 < \cdots < M_m \),
- \( P_1 - \{ A \leftarrow L_1 \land \cdots \land L_n \} = P_2 - \{ A \leftarrow M_1 \land \cdots \land M_m \} \).

It will later be important that our transformations are only applicable if they really simplify something, i.e. that there is no program \( P \) with \( P \rightarrow P \) (because otherwise there would be no “normalform”, see below). Therefore it is necessary to add the condition \( L_1 \land \cdots \land L_n \neq M_1 \land \cdots \land M_m \).

The deletion of subsumed rules and the normalization of rule bodies are satisfied by all semantics we consider. So they are indeed very natural:
Lemma 4.2.17:
If a semantics $\mathcal{S}$ is equivalence-preserving, it allows the deletion of subsumed rules and the normalization of rule bodies.

Proof:
This is trivial: If an interpretation satisfies a rule $A \leftarrow M_1 \land \cdots \land M_m$, it also satisfies any rule $A \leftarrow L_1 \land \cdots \land L_n$ with $\{M_1, \ldots, M_m\} \subseteq \{L_1, \ldots, L_n\}$. So if $P_1 \xrightarrow{S} P_2$ or $P_1 \xrightarrow{B} P_2$, the programs $P_1$ and $P_2$ are logically equivalent, and thus $\mathcal{S}(P_1) = \mathcal{S}(P_2)$.

Normalforms and the Residual Program
If a semantics allows unfolding and the deletion of tautologies, we can transform every program into an equivalent program without positive body literals. This simplifies semantical characterizations and can also be seen as a first step of query evaluation. Rules without positive body literals have already been studied by Bry [Bry89, Bry90a], who called them “conditional facts”. He also introduced bottom-up evaluation with conditional facts (see below) where the negative body literals are “delayed” and attached as conditions to the derived facts. The concept of conditional facts has independently been introduced by Dung and Kanchansut [DK89a, DK89b]. It is a very useful and natural notion and also implicitly used in several query evaluation algorithms.

Definition 4.2.18 (Conditional Fact):
A conditional fact is a rule where all body literals are negative, i.e. of the form
$$A \leftarrow \lnot B_1 \land \cdots \land \lnot B_n.$$

Theorem 4.2.19 (Normalform I: No Positive Body Literals):
Let $\rightarrow_1$ be the union of the transformations “unfolding” and “deletion of tautologies”, and $\rightarrow_1^*$ be the reflexive and transitive closure.
• For every program $P$ there is a set $F$ of conditional facts with $P \xrightarrow{\rightarrow_1^*} F$.

Proof:
The proof is by induction on the number $n$ of atoms $A \in \text{BASE}^+$ which occur in $P$ as positive body literals. This number is finite since $P$ is finite.
• $n = 0$: In this case the given program $P$ is already a set of conditional facts.
• $n \rightarrow n + 1$: Let $A \in \text{BASE}^+$ occur as a body literal in $P$. We first eliminate all tautological rules about $A$ (containing $A$ in head and body). Let the resulting program be $P_1$.

We now apply unfolding to all occurrences of $A$ in rule bodies as long as possible. Of course, we must show that this process terminates. Since there are no tautological rules about $A$, unfolding replaces $A$ by a sequence of literals different from $A$. However, if the original rule contained more than one occurrence of $A$, the resulting rules still contain $A$, so that the total number of occurrences of $A$ can even temporarily increase. But if we assign the programs “costs” as follows, it is simple to see that
the cost decreases in every step: Let \( m \) be the number of rules about \( A \) in \( P_1 \). Let a rule with \( k \) occurrences of \( A \) as a body literal have the cost \( (m+1)^k \). An unfolding step (if \( k > 0 \)) replaces this rule by \( m \) rules with cost \( (m+1)^{k-1} \), having the total cost \( m \cdot (m+1)^{k-1} = (m+1)^k - (m+1)^{k-1} \). Since \( (m+1)^{k-1} > 0 \), this is less than the original cost \( (m+1)^k \).

Let the resulting program be \( P_2 \). Note that all atoms occurring in \( P_2 \) as body literals already occurred in \( P \) as body literals, and furthermore, \( A \) does no longer occur. So we have reduced their number by at least one and can apply the inductive hypothesis: \( P \xrightarrow{\star} P_1 \xrightarrow{\star} P_2 \xrightarrow{\star} F \). \( \square \)

In the theory of term-rewriting (see, e.g., [EGL89]) a term \( t \) is called irreducible if no rewriting rule is applicable to \( t \). This corresponds in our approach to sets of conditional facts, which are the “irreducible programs” to which \( \rightarrow_1 \) is not applicable.

Given a term \( t \), any irreducible term \( t' \) with \( t \xrightarrow{\star} t' \) is called a normalform of \( t \). In Theorem 4.2.19 we have shown a property which is known as “termination” in term rewriting: Every term \( t \) has at least one normalform. Note, however, that this does not exclude infinite sequences of rewriting steps. For instance, we can apply unfolding to \( p \leftarrow p \) as long as we wish without changing it.

**Definition 4.2.20 (Normalform):**

A program \( P' \) is a normalform of a program \( P \) wrt a transformation \( \rightarrow \) iff

- \( P \xrightarrow{\star} P' \), and
- there is no program \( P'' \) with \( P' \xrightarrow{\star} P'' \).

If normalforms are simpler than general programs, they can be useful for query-evaluation, because they have the same semantics as the original program. It is interesting that also conversely, if it suffices to define the semantics on the normalforms, the semantics automatically allows the transformation:

**Lemma 4.2.21:**

Suppose that every program \( P \) has a normalform wrt \( \rightarrow \). Then the following two conditions are equivalent:

- \( S \) allows \( \rightarrow \).
- \( S(P) = S(P') \) for every program \( P \) and every normalform \( P' \) of \( P \).

**Proof:**

- If \( P' \) is a normalform of \( P \), we have \( P = P_0 \leftarrow P_1 \leftarrow \cdots \leftarrow P_n = P' \). If \( S \) allows \( \rightarrow \), it follows that \( S(P) = S(P_0) = S(P_1) = \cdots = S(P_n) = S(P') \).
- Let \( P_1 \leftarrow P_2 \) and let \( P' \) be a normalform of \( P_2 \). Then \( P' \) is also a normalform of \( P_1 \). Thus \( S(P_1) = S(P') = S(P_2) \). \( \square \)

Contradicting slightly the intuitive understanding of the word “normalform”, there can be more than one normalform of a given program \( P \). Consider for instance the following program:

\[
p \leftarrow p \land \textbf{not} \ q.
\]

\[
p.
\]
Of course, if we simply delete the tautology, we get the program

\[ p. \]

But if we first unfold the body literal and then delete the resulting tautology, we get

\[ p \leftarrow \text{not } q. \]
\[ p. \]

In fact, by unfolding \( n \) times and then deleting the tautology, we can get rules with \( 0, 1, \ldots, n \) copies of \( \text{not } q \) in the body.

A very useful property in term-rewriting is the confluence of the rewriting system:

**Definition 4.2.22 (Confluence):**
A transformation \( \rightsquigarrow \) is called confluent iff, whenever \( P \rightsquigarrow^* P_1 \) and \( P \rightsquigarrow^* P_2 \), there is a program \( P_3 \) with \( P_1 \rightsquigarrow^* P_3 \) and \( P_2 \rightsquigarrow^* P_3 \).

Confluence and termination imply that every program \( P \) has exactly one normalform: If \( P \) had two normalforms \( P_1 \) and \( P_2 \), this would contradict confluence, because \( P_1 \) and \( P_2 \) are irreducible, so we cannot get to a common program \( P_3 \).

We can make our rewriting system confluent by adding the deletion of subsumed rules. This is also motivated by the above example, where the derivable normalforms only differ in subsumed rules. However, the proof of the confluence needs some preparation.

We first give a specific algorithm for computing from a program \( P \) some set of conditional facts \( F \), which we will later show to be the unique normalform. The idea is to generalize bottom-up evaluation to conditional facts, and to take the least fixpoint of this immediate consequence operator. As mentioned above, this has already been done by Bry [Bry90a], but the relation to the elementary transformations of unfolding and tautology elimination was investigated first in [BD95b, BD95c].

An example of a derivation step with conditional facts is:

\[
\begin{align*}
p & \leftarrow \text{not } s \quad \land \quad \text{not } r \\
q_1 \land q_2 \quad \land \quad \text{not } r \\
q_1 \quad q_2 & \leftarrow \text{not } s
\end{align*}
\]

So the heads of the conditional facts are matched with body literals as usual, and their conditions are attached to the derived fact (together with the negative body literals of the rule itself). This formalizes the delayed evaluation of negative body literals: We simply remember their presence in all derived facts that depend on them, but we do not check whether they are true or false. Of course, a practical implementation will evaluate those conditions which are easy to evaluate and only delay the “hard cases” (see Section 4.3). Sometimes such delaying is unavoidable, since we can
evaluate a negative body literal only to true if we are sure that the corresponding positive fact cannot be derived, also not later in the proof. For instance, in the program \{p ← \texttt{not} p\}, the negative body literal \texttt{not} p will stay “forever delayed”, since we can evaluate it neither to true nor to false.

Bottom-up evaluation with conditional facts is in fact nothing else than hyper-resolution \cite{CL73}, if we view a conditional fact like \(q_2 ← \texttt{not} s\) as the “electron” \(q_2 ∨ \neg \texttt{not} s\) and a rule like \(p ← q_1 ∧ q_2 ∧ \texttt{not} r\) as the “nucleus” \(p ∨ \neg \texttt{not} r ← q_1 ∧ q_2\). The formal definition is as follows:

**Definition 4.2.23 (Bottom-Up Evaluation with Conditional Facts):**
Given a program \(P\), we define the following operator \(T_P\) on sets of conditional facts:

\[
T_P(F) := \{ A ← B_1 ∧ \cdots ∧ B_n \mid \text{there is a rule } A ← L_1 ∧ \cdots ∧ L_n \text{ in } P \\
\quad \text{and for } i = 1, \ldots, n: \\
\quad \quad \text{if } L_i \text{ is a positive literal then } \\
\quad \quad \quad L_i ← B_i \text{ is a conditional fact in } F \\
\quad \quad \text{else } L_i \text{ is a negative literal and } \\
\quad \quad \quad B_i = L_i \}. 
\]

Bottom-up query evaluation as defined above would often not terminate, since it may derive longer and longer rule bodies, for instance in the above example

\[
p ← p ∧ \texttt{not} q. \\
p.
\]

Obviously, we should add the elimination of subsumed rules:

**Definition 4.2.24 (Bottom-Up Evaluation/Subsumed Rule Elimination):**
Given a program \(P\), we define the following operator \(\hat{T}_P\) on sets of conditional facts:

\[
\hat{T}_P(F) := \{ A ← L_1 ∧ \cdots ∧ L_n ∈ T_P(F) \mid \text{there is no } A ← M_1 ∧ \cdots ∧ M_m \text{ in } T_P(F) \\
\quad \text{subsuming } A ← L_1 ∧ \cdots ∧ L_n \}. 
\]

In contrast to \(T_P\), the new operator \(\hat{T}_P\) is not monotonic with respect to the inclusion order on sets of conditional facts. However, it is natural to use an order where subsumed facts are ignored:

**Definition 4.2.25 (Order Between Sets of Conditional Facts):**
We define the following order \(⊆\) between sets of conditional facts:

\[
F_1 ⊆ F_2 :⇔ \text{ for every } A ← L_1 ∧ \cdots ∧ L_n \text{ in } F_1 \\
\quad \text{there is } A ← M_1 ∧ \cdots ∧ M_m \text{ in } F_2 \\
\quad \quad \text{which is equal to or subsumes } A ← L_1 ∧ \cdots ∧ L_n.
\]

**Lemma 4.2.26:**
The operator \(\hat{T}_P\) is monotonic wrt \(⊆\).
4.2. A FRAMEWORK FOR STUDYING SEMANTICS

Proof:
First, the operator $T$ is obviously monotonic wrt $\sqsubseteq$: If $L_i \leftarrow B_i$ subsumes $L_i \leftarrow B'_i$, then also $A \leftarrow B_1 \land \cdots \land B_n$ subsumes $A \leftarrow B'_1 \land \cdots \land B'_n$. So $F_1 \sqsubseteq F_2 \implies T_P(F_1) \sqsubseteq T_P(F_2)$.

Second, the order $\sqsubseteq$ depends only on the minimal facts in both sets (since the “subsumes (or equal to)” relation is transitive). So $T_P(F_1) \sqsubseteq T_P(F_2) \implies \hat{T}_P(F_1) \sqsubseteq \hat{T}_P(F_2)$. \quad $\square$

Note that we must allow also infinite sets of conditional facts, since the lattice otherwise would not be complete. However, we will show that the least fixpoint of $\hat{T}_P$ is finite.

Definition 4.2.27 (Weak Residual Program):
The weak residual program $wres(P)$ of a program $P$ is the least fixpoint of $\hat{T}_P$:
\[
wres(P) := \text{lfp}(\hat{T}_P).
\]

Lemma 4.2.28:
Let $P_2$ result from $P_1$ by one of the three transformations
- “deletion of tautologies” ($P_1 \rightarrow_T P_2$),
- “unfolding” ($P_1 \rightarrow_U P_2$), or
- “deletion of subsumed rules” ($P_1 \rightarrow_S P_2$).
Then $wres(P_1) = wres(P_2)$.

Proof:
- Let $P_1 \rightarrow_T P_2$ and let
  \[
  A \leftarrow L_1 \land \cdots \land L_{i-1} \land A \land L_{i+1} \land \cdots \land L_n
  \]
  be the deleted tautological rule. Whenever this rule is used in the $T_P$-operator, the derived conditional fact
  \[
  A \leftarrow B_1 \land \cdots \land B_{i-1} \land B \land B_{i+1} \land \cdots \land B_n
  \]
  is subsumed by the conditional fact $A \leftarrow B$ matched with the body literal $A$. Since subsumed conditional facts are deleted in $\hat{T}_P$, and during the fixpoint computation we have $F \sqsubseteq \hat{T}_P(F)$, the derived conditional fact has no effect on the least fixpoint, and thus $wres(P_1) = wres(P_2)$.

- Let $P_1 \rightarrow_U P_2$ and let the body literal $B$ in the rule
  \[
  A \leftarrow L_1 \land \cdots \land L_{i-1} \land B \land L_{i+1} \land \cdots \land L_n
  \]
  be unfolded, and let $B \leftarrow M_{j,1} \land \cdots \land M_{j,m_j}, j = 1, \ldots, k$ be all rules about $B$ in $P_1$. 
  - If a conditional fact $A \leftarrow B_1 \land \cdots \land B_{i-1} \land B \land B_{i+1} \land \cdots \land B_n$ was derived by the unfolded rule, then the conditional fact $B \leftarrow B$ inserted for $B$ must have been derived by one of the rules $B \leftarrow M_{j,1} \land \cdots \land M_{j,m_j}$ in the previous step (or else this rule is itself a conditional fact). So $B$ has the form $B'_1 \land \cdots \land B'_{m_j}$, where $B'_1 = M_{j,1}$ or $M_{j,1} \leftarrow B'_1$ was a conditional fact one step ago. It might have been replaced in the meantime by a stronger conditional fact $M_{j,1} \leftarrow B''_1$. But then we can derive with the unfolded rule
    \[
    A \leftarrow B_1 \land \cdots \land B_{i-1} \land M_{j,1} \land \cdots \land M_{j,m_j} \land B_{i+1} \land \cdots \land B_n
    \]
  the same (or a possibly stronger) conditional fact.
- Vice versa, if a conditional fact was derived by using the unfolded rule
  \[ A \leftarrow B_1 \land \cdots \land B_{i-1} \land M_{j,1} \land \cdots \land M_{j,m_j} \land B_{i+1} \land \cdots \land B_n, \]
  it can also be derived by using first the rule \( B \leftarrow M_{j,1} \land \cdots \land M_{j,m_j} \) and then
  the rule \( A \leftarrow L_1 \land \cdots \land L_{i-1} \land B \land L_{i+1} \land \cdots \land L_n. \)

- If \( P_1 \leftarrow_S P_2 \), then conditional facts derived by using the weaker rule are obviously
  subsumed by conditional facts derived using the stronger rule. \( \Box \)

**Theorem 4.2.29 (Normalform II: Weak Residual Program):**

Let \( \rightarrow_2 \) be the union of the transformations “unfolding”, “deletion of tautologies”,
and “deletion of subsumed rules”. Then \( \rightarrow_2 \) is

- terminating, i.e. every program \( P \) has a normalform wrt \( \rightarrow_2 \), and
- confluent, i.e. for all programs \( P, P_1, P_2 \) with \( P \leftarrow_2^* P_1 \) and \( P \leftarrow_2^* P_2 \), there is a
  program \( P_3 \) with \( P_1 \leftarrow_2^* P_3 \) and \( P_2 \leftarrow_2^* P_3 \).

The weak residual program \( \text{wres}(P) \) is the unique normalform of \( P \). This also proves
that \( \text{wres}(P) \) is indeed a program in our sense (i.e. finite).

**Proof:**

- The existence of a normalform follows easily from Theorem 4.2.19: After we have
  computed the normalform wrt \( \rightarrow_1 \), we apply the elimination of subsumed rules \( \rightarrow_S \) as
  long as possible. This terminates, since the number of rules is reduced in each step.
  Furthermore, \( \rightarrow_T \) and \( \rightarrow_U \) remain non-applicable.
- For an irreducible program \( P \) (without positive body literals or subsumed rules) it is clear that
  \( \text{wres}(P) = P \).
- Now if \( P \leftarrow_2^* P' \), i.e. \( P \rightarrow_2 P_1 \rightarrow_2 P_2 \rightarrow_2 \cdots \rightarrow_2 P_n \rightarrow_2 P' \) then Lemma 4.2.28 implies
  \( \text{wres}(P) = \text{wres}(P_1) = \text{wres}(P_2) = \cdots = \text{wres}(P_n) = \text{wres}(P') \).
- Thus, if \( P' \) is a normalform of \( P \), then \( \text{wres}(P) = \text{wres}(P') = P' \). Since every pro-
  gram \( P \) has a normalform \( P' \), we can conclude that \( \text{wres}(P) \) is a normalform, and in
  fact the only normalform (since \( P' = \text{wres}(P) \) for every normalform \( P' \) of \( P \)).
- Finally, if \( P \leftarrow_2^* P_1 \) and \( P \leftarrow_2^* P_2 \), we have that \( \text{wres}(P_1) = \text{wres}(P) = \text{wres}(P_2) \).
  As shown above, \( P_1 \leftarrow_2^* \text{wres}(P_1) \) and \( P_2 \leftarrow_2^* \text{wres}(P_2) \). Thus, there is a program \( P_3 \),
  namely \( \text{wres}(P) \), with \( P_1 \leftarrow_2^* P_3 \) and \( P_2 \leftarrow_2^* P_3 \). \( \Box \)

The above rewriting system is interesting, because it allows an algorithmic elimination
of positive body literals under very weak conditions on the semantics. Further
investigations can then concentrate on the meaning and evaluation of the negative
body literals, they do not have to treat also the positive part of the program, which
is a considerable simplification.

If we view the rule bodies as sets (i.e. the semantics allows normalization), we
in fact do not need the elimination of subsumed rules: In [BD95c], we have shown
that already elimination of tautologies and unfolding (plus the implicit assumption
of normalization) are sufficient for \( S(\text{lfp}(T_P)) = S(P) \). However, these properties are
not sufficient for the confluence, as the above example

\[
p \leftarrow p \land \text{not } q.
\]
\[p.\]
4.2. A FRAMEWORK FOR STUDYING SEMANTICS

shows.

With all four transformations we get exactly the strictly equivalence-preserving semantics. We have already proven above that if a semantics looks only at the minimal models of a program, then it allows these transformations. However, the converse is also true, which shows that strict equivalence-preservation is a very natural property.

Theorem 4.2.30 (Normalform III):

Let $\rightarrow_3$ be the union of the transformations “unfolding”, “deletion of tautologies”, “deletion of subsumed rules”, and “normalization of rule bodies”.

- Every program $P$ has a normalform $P'$ wrt $\rightarrow_3$, i.e. $\rightarrow_3$ is terminating.
- If $P'$ is a normalform of $P$ wrt $\rightarrow_3$, then $P$ and $P'$ have the same set of minimal models.
- If two programs $P_1$ and $P_2$ have the same set of minimal models, they have identical normalforms.
- $\rightarrow_3$ is confluent.

Proof:

- From Theorem 4.2.19 it is obvious that normalforms wrt $\rightarrow_3$ exist: We first compute a set of conditional facts, and then apply the deletion of subsumed rules and the normalization of rule bodies as long as possible. If we assign every non-normalized rule the cost 2, and every normalized rule the cost 1, the total cost of the program is reduced in each step, so this process comes to an end.
- From Lemma 4.2.7, Lemma 4.2.9, and Lemma 4.2.17, it is also clear that the normalform $P'_i$ has the same minimal models as $P_i$ ($i = 1, 2$).
- We now show that two normalforms $P'_1$ and $P'_2$ wrt the above $\rightarrow$ can only have the same minimal models if they are equal. Together with the last proposition this immediately implies the property stated in the theorem. Suppose that $P'_1 \neq P'_2$. Let for instance $P'_1$ contain a conditional fact $A \leftarrow \text{not } B_1 \land \cdots \land \text{not } B_n$, which is not contained in $P'_2$, and for which $P'_2$ does not contain a subsuming conditional fact $A \leftarrow \text{not } B_{i_1} \land \cdots \land \text{not } B_{i_m}$ with $m < n$. (If $P'_2$ contained such a subsuming fact, we could consider the inverse situation, since $P'_1$ cannot contain a conditional fact subsuming another conditional fact in $P'_1$.) Now consider the interpretation $I$ with $I \models \text{not } B_i$, $i = 1, \ldots, n$, $I \not\models A$, and which makes any other negative literal false and positive literal true. It obviously is a model of $P'_2$, but not a model of $P'_1$. Now by Lemma 4.1.13 there is a minimal model $I_0$ of $P'_2$ with $I_0 \preceq I$. But $I_0$ is still not a model of $P'_1$. So $P'_1$ and $P'_2$ do not have the same minimal models.
- If $P_1 \rightarrow_3^* P_2$ and $P_1 \rightarrow_3^* P_3$, then $P_2$ and $P_3$ have the same minimal models. But then, by the last part of this theorem, their normalforms must be identical. 

Theorem 4.2.31 (Characterization of Strict Equivalence-Preservation):

A semantics $S$ is strictly equivalence-preserving iff it allows the deletion of tautologies, unfolding, normalization of rule bodies, and the elimination of subsumed rules.
Proof:
It follows from Lemma 4.2.7, 4.2.9, and 4.2.17 that if $S$ is strictly equivalence-preserving (and thus also equivalence-preserving), then it allows the four transformations.

Now suppose that $S$ allows the four transformations and that programs $P_1$ and $P_2$ have the same minimal models. Then by the third part of Theorem 4.2.30, their normalforms $P_1'$ and $P_2'$ are identical, so we have $S(P_1) = S(P_1') = S(P_2) = S(P_2)$.

The normalform wrt $\rightarrow_3$ can easily be computed from the weak residual program by normalizing the rule bodies and deleting then newly subsumed conditional facts:

**Definition 4.2.32 (Normalization and Deletion of Subsumed Facts):**
Let $\text{norm}$ be the following “normalization” operator on sets on conditional facts:

$$\text{norm}(F) := \{ A \leftarrow M_1 \land \cdots \land M_m \mid M_1 \prec M_1 \prec \cdots \prec M_m, $$
$$\text{there is } A \leftarrow L_1, \ldots, L_n \text{ in } F$$
$$\text{such that } \{L_1, \ldots, L_n\} = \{M_1, \ldots, M_m\},$$
$$\text{there is not } A \leftarrow L'_1 \land \cdots \land L'_k \text{ in } F$$
$$\text{such that } \{L'_1, \ldots, L'_k\} \subseteq \{M_1, \ldots, M_m\}.\$$

**Corollary 4.2.33:**
The unique normalform of a program $P$ wrt $\rightarrow_3$ is $\text{norm}(\text{wres}(P))$.

The previous normalforms contained still many negative literals with an obvious truth value. They can be eliminated if we add positive and negative reduction. Since any sensible semantics should allow these transformations, this does not really restrict the space of possible semantics.

Our goal is that only the hard cases of negative body literals remain in the resulting “residual program” (called the “islands of complexity” in [MD93]). They have to be evaluated by some other means, and this is where the different semantics really differ. But usually the remaining negative body literals are only very few compared with the size of the original program.

The following operator obviously corresponds to the application of positive and negative reduction and the elimination of subsumed rules:

**Definition 4.2.34 (Reduction of Conditional Facts):**
We define the following reduction operator on sets of conditional facts:

$$R(F) := \{ A \leftarrow \textbf{not} B_{i_1} \land \cdots \land \textbf{not} B_{i_k} \mid A \leftarrow \textbf{not} B_1 \land \cdots \land \textbf{not} B_n \in F, $$
$$B_i \leftarrow \text{true} \notin F \text{ for } i = 1, \ldots, n, $$
$$\text{there is no } A \leftarrow \textbf{not} C_1 \land \cdots \land \textbf{not} C_m \text{ in } F$$
with $\{C_1, \ldots, C_m\} \subset \{B_1, \ldots, B_n\}$,
$$\{B_{i_1}, \ldots, B_{i_k}\} = \{B_1, \ldots, B_n\} \cap \text{delay}(F),$$
$$1 \leq i_1 < i_2 < \cdots < i_k \leq n\},$$

where $\text{delay}(F)$ is the set of negative literals not yet known to be true:

$$\text{delay}(F) := \{ \textbf{not} A \mid \text{there is a rule in } F \text{ about } A\}.$$
Note that positive reduction can yield subsumed facts and make negation reduction applicable again, whereas negative reduction can cause positive reduction to be applicable once more. So we need again a fixpoint computation. The iteration of the operator \( R \) terminates, since it reduces the total number of body literals. We call the result of this reduction the “residual program”, since it is similar to the “residual program” computed in [CW93a]. However, our version also contains the true facts, not only conditions for the undefined facts. Thus, our residual program is still equivalent to the original program (for semantics allowing the discussed elementary transformations):

**Definition 4.2.35 (Residual Program):**

The residual program \( \text{res}(P) \) of a program \( P \) is defined as follows:

- Let \( P_0 := \text{norm}(\text{lfp}(\bar{T}_P)) \).
- Let \( P_i := R(P_{i-1}) \) for \( i \in \mathbb{N} \), and let \( n \in \mathbb{N} \) with \( P_n = P_{n-1} \).
- Then \( \text{res}(P) = P_n \).

**Theorem 4.2.36 (Normalform IV):**

Let \( \rightarrow^*_4 \) be the union of the transformations “unfolding”, “deletion of tautologies”, “deletion of subsumed rules”, “normalization of rule bodies”, and “positive and negative reduction”.

- Every program \( P \) has a normalform wrt \( \rightarrow^*_4 \), namely the residual program \( \text{res}(P) \).
- \( \rightarrow^*_4 \) is confluent, so \( \text{res}(P) \) is the only normalform of \( P \).

**Proof:**

- By Corollary 4.2.33, \( P \rightarrow^*_3 \text{norm}(\text{wres}(P)) \), and since \( \rightarrow^*_3 \subseteq \rightarrow^*_4 \), it follows that \( P \rightarrow^*_4 \text{norm}(\text{wres}(P)) \). Furthermore, it is clear that \( F \rightarrow^*_4 R(F) \) for every set \( F \) of conditional facts. Thus, \( P \rightarrow^*_4 \text{res}(P) \). Finally, \( \rightarrow^*_4 \) is obviously no longer applicable in the residual program.
- We first show that if \( P_1 \rightarrow^*_4 P_2 \), then \( \text{res}(P_1) = \text{res}(P_2) \):
  - If \( P_1 \rightarrow^*_3 P_2 \), we have \( \text{wres}(P_1) = \text{wres}(P_2) \) by Corollary 4.2.33, and thus also \( \text{res}(P_1) = \text{res}(P_2) \).
  - If \( P_1 \rightarrow^*_P P_2 \), then \( \text{wres}(P_2) \) differs from \( \text{wres}(P_1) \) only by additional conditions of the from \( \text{not} \, B \), where \( B \) occurs in no rule head. But these conditions will be deleted by the \( R \)-operator, so they do not influence the residual program.
  - If \( P_1 \rightarrow^*_N P_2 \), then \( \text{wres}(P_1) \) differs from \( \text{wres}(P_2) \) only by additional rules containing a condition of the form \( \text{not} \, B \), where \( B \leftarrow \text{true} \) is also contained in \( \text{wres}(P_1) \). These rules will be deleted by the \( R \)-operator, so they again have no influence on the residual program.

Now if \( P \rightarrow^*_4 P_2 \) and \( P \rightarrow^*_4 P_2 \), we get \( \text{res}(P_1) = \text{res}(P) = \text{res}(P_2) \), thus there is a program \( P_3 \), namely \( \text{res}(P) \), with \( P_1 \rightarrow^*_4 P_3 \) and \( P_2 \rightarrow^*_4 P_3 \).
Defining Semantics on the Residual Program

Up to now, we have used Lemma 4.2.21 only to conclude that if a semantics $S$ allows a transformation $\rightarrow$, then $S(P') = S(P)$ for the normalform $P'$ of $P$. For instance, if $S$ allows $\rightarrow_4$, then $S(\text{res}(P)) = S(P)$.

However, the converse is also true, i.e. if a semantics satisfies $S(\text{res}(P)) = S(P)$ for all programs $P$, then it allows $\rightarrow_4$. This gives us a way to construct semantics by defining them on the residual program and then lifting the definition to all programs by $S(P) := S(\text{res}(P))$. Such a semantics would have all the nice properties studied in this section.

However, $S(P) := S(\text{res}(P))$ does not ensure that $S$ selects only models of $P$ (which is required in our definition of an abstract semantics). So we cannot select arbitrary models of the residual program. In this subsection we will define a large class of models which guarantee that a semantics defined on residual programs can be lifted to all programs.

This result shows that our transformations are not very restrictive, and still allow many possible semantics. This also demonstrates the limits of our approach and calls for additional properties, since the semantics can behave quite strange on the residual programs. It is certainly not true that any semantics which allows $\rightarrow_4$ is already a good semantics. More research is needed here. But for these investigations, it is very useful to have a method for constructing semantics satisfying $\rightarrow_4$. This also gives us a partial characterization of this class of semantics.

We have shown above that some of our transformations do not change the set of models of the program, or not the set of minimal models. Here we need a class of models, which can only decrease when we apply our transformations. In this case we know that such a model of the residual program is also a model of the original program. The following “weakly supported models” have this property:

**Definition 4.2.37 (Weakly Supported Model):**
A model $I$ of a logic program $P$ is weakly supported iff

- For every $A \in \text{BASE}^+$ with $I \models A$, there is a rule $A \leftarrow L_1 \land \cdots \land L_n$ in $P$ with $I \models L_i$, $i = 1, \ldots, n$.
- If $A \leftarrow \text{true}$ is contained in $P$, then $I \not\models \text{not } A$.
- If $A$ does not occur in any rule head, then $I \not\models \text{not } A$.

**Theorem 4.2.38 (Weakly Supported Models):**
If $P_1 \rightarrow_4 P_2$, and $I$ is a weakly supported model of $P_2$, then $I$ is also a weakly supported model of $P_1$.

---

5It is not clear at the moment whether this characterization is complete, i.e. whether every semantics satisfying $\rightarrow_4$ can be constructed in this way.
4.2. A FRAMEWORK FOR STUDYING SEMANTICS

Proof:

- Let $P_1 \rightarrow_T P_2$ and let $I$ be a weakly supported model of $P_2$. Of course, $I$ is also a model of $P_1$, since every interpretation satisfies a tautological rule. Furthermore, no support for an atom can be lost, and there are no new requirements from the third condition, since $P_2 \subseteq P_1$. Finally, a tautological rule does not have the form $A \leftarrow true$, so it does not yield additional constraints on $I$ by the second condition.

- Let $P_1 \rightarrow_U P_2$ and let $I$ be a weakly supported model of $P_2$. Let the unfolded rule be

$$A \leftarrow L_1 \land \cdots \land L_{i-1} \land B \land L_{i+1} \land \cdots \land L_n.$$  

If this rule were not satisfied in $I$, this would mean $I \models B$, $I \models L_j$, and $I \models A$. But since $I \models B$, there is a supporting rule for $B$ in $P_2$ (and thus in $P_1$), having the form

$$B \leftarrow M_{j,1} \land \cdots \land M_{j,m_j}.$$  

But then the unfolded rule

$$A \leftarrow L_1 \land \cdots \land L_{i-1} \land M_{j,1} \land \cdots \land M_{j,m_j} \land L_{i+1} \land \cdots \land L_n$$  

is contained in $P_2$, but violated in $I$, which is a contradiction.

- Suppose that one of the unfolded rules, namely

$$A \leftarrow L_1 \land \cdots \land L_{i-1} \land B \land L_{i+1} \land \cdots \land L_n$$

was used to support $A$ in $I$. Then $I \models M_{j,1} \land \cdots \land M_{j,m_j}$. If $B \neq A$, the rule

$$B \leftarrow M_{j,1} \land \cdots \land M_{j,m_j}$$

is also contained in $P_2$, thus $I \models B$. If $B = A$, $B$ is of course also true. So the rule

$$A \leftarrow L_1 \land \cdots \land L_{i-1} \land B \land L_{i+1} \land \cdots \land L_n$$

can be used to support $A$ in $P_1$.

Since the unfolded rule contained a body literal, it does not yield new constraints on $I$ by the second condition. Also the third condition is no problem, since $A$ remains present in the rule heads.

- If $P_1 \rightarrow_S P_2$, then any weakly supported model of $P_2$ is also a weakly supported model of $P_1$ since it first is a model, second no support is lost ($P_2 \subseteq P_1$), third the deleted rules were no facts, and fourth, the set of atoms occurring in rule heads is unchanged.

- If $P_1 \rightarrow_B P_2$, then any weakly supported model of $P_2$ is also a weakly supported model of $P_1$ since the definition does not refer to the order or multiplicity of body literals.

- Let $P_1 \rightarrow_P P_2$, and let $I$ be a weakly supported model of $P_2$. Let the strengthened rule be

$$A \leftarrow L_1 \land \cdots \land L_{i-1} \land \text{not } B \land L_{i+1} \land \cdots \land L_n.$$  

Since $I$ satisfies even the stronger rule, it is of course a model of $I_1$. Also, if the stronger rule was used to support $A$, the weaker rule can also be used, since $I \models \text{not } B$ by the third condition. From $P_2$ to $P_1$, the set of facts can only decrease, and the set of atoms occurring in rule heads is unchanged, so the second and third condition remain satisfied.
• Let \( P_1 \leftarrow_N P_2 \), let \( I \) be a weakly supported model of \( P_2 \), and let

\[
A \leftarrow L_1 \land \cdots \land L_{i-1} \land \text{not} B \land L_{i+1} \land \cdots \land L_n
\]

be the deleted rule, where \( B \leftarrow \text{true} \) occurs in \( P_1 \) (and thus in \( P_2 \)). Since \( I \not\models \text{not} B \), \( I \) satisfies the deleted rule. Since \( P_2 \subseteq P_1 \), no support can be lost and the third condition cannot be violated. The deleted rule was no fact, so the second condition also poses no problem.

So if \( S \) selects any subset of the weakly supported models of the residual programs, we can extend it to all programs by \( S(P) := S(\text{res}(P)) \). Note that such weakly supported models really exist:

**Lemma 4.2.39:**

Let \( I \) be a minimal model of a program \( P \) satisfying:

- If \( A \leftarrow \text{true} \) is contained in \( P \), then \( I \not\models \text{not} A \).
- If \( A \) does not occur in any rule head, then \( I \models \text{not} A \).

Then \( I \) is a weakly supported model of \( P \).

**Proof:**

Suppose that \( A \in \text{BASE}^+ \) were not supported, i.e. there were no rule \( A \leftarrow L_1 \land \cdots \land L_n \) in \( P \) with \( I \models L_i, i = 1, \ldots, n \). Then \( I_0 \), which differs from \( I \) only in \( I_0 \not\models A \), would also be a model of \( P \), contradicting the assumed minimality of \( I \).

This means that we can choose any valuation of the negative literals still remaining in the residual program, and then compute the (unique) minimal model in order to determine the truth values of the positive literals. In fact, all weakly supported models of the residual program can be constructed in this way:

**Lemma 4.2.40:**

If \( I \) is a weakly supported model of \( \text{res}(P) \), then it is also minimal.

**Proof:**

Suppose that \( I \) were not minimal, so there were a model \( I_0 \) of \( \text{res}(P) \) with \( I_0 \preceq I \) and \( I_0 \not\models A \). Then there would be \( A \in \text{BASE}^+ \) with \( I \models A \) and \( I_0 \not\models A \). Let \( A \leftarrow L_1 \land \cdots \land L_n \) be the supporting rule for \( A \), so \( I \models L_i, i = 1, \ldots, n \). All body literals in a residual program are negative, so \( I \) and \( I_0 \) cannot differ in their truth values. Thus, \( I_0 \) violates \( A \leftarrow L_1 \land \cdots \land L_n \).

However, for defining a semantics satisfying \( \rightarrow_4 \), we can not only select any subset of the weakly supported models of the residual program. If we make negative literals false, a model of \( P \) remains a model of \( P \). So we can select also such models of the residual program which differ from a weakly supported model \( I \) by making some of the negative literals, which were true in \( I \), false. Of course, if \( A \) does not occur in \( \text{res}(P) \), our definition of a semantics requires that \( \text{not} A \) is true.
Theorem 4.2.41 (Lifting Semantics from the Residual Program):
Let $I$ be a weakly supported model of $\text{res}(P)$, and let $I_0$ be an interpretation satisfying:

- For all $A \in \text{BASE}^+$: $I_0 \models A \iff I \models A$.
- For all $A \in \text{BASE}^+$: $I_0 \models \text{not } A \implies I \models \text{not } A$.

Then $I_0$ is a model of $P$.

Example 4.2.42:
Consider the following residual program:

\[
p \leftarrow \text{not } p.
\]
\[
q \leftarrow \text{not } p.
\]

If $\text{not } p$ is true, also $p$ and $q$ must be true, otherwise the interpretation would be no model. If $\text{not } p$ is false, either both $p$ and $q$ can be false, or both can be true. The interpretation making $p$ and $q$ both true although $\text{not } p$ is false is no weakly supported model, but it can be derived from the weakly supported model in which $\text{not } p$ is true.

Note that it is not possible that only one of $p$ and $q$ is true. Suppose that $p$ is true, and $q$ and $\text{not } p$ are false. Then consider the program $P$ which consists of the above two rules plus

\[
q \leftarrow p.
\]

It has the above residual program, but the interpretation in question is no model.

The truth value of $\text{not } q$ can be arbitrarily chosen in each of the cases. \hfill \Box

The Supported Model Semantics

After we have introduced all these properties, let us investigate and characterize the three semantics defined above. We will start with the supported model semantics SUPP. It allows all of the above transformations except the deletion of tautologies:

Theorem 4.2.43 (Properties of Supported Models):
The semantics SUPP allows unfolding, deletion of contradictions, positive and negative reduction, deletion of subsumed rules, and the normalization of rule bodies.

Proof:
Note that a supported model is a weakly supported model and that a normal weakly supported model is also supported. So if $P_1 \rightarrow_U P_2$ for one of these transformations (except the deletion of contradictions), we know by Theorem 4.2.38 that a supported model of $P_2$ is also a supported model of $P_1$. So it suffices to prove only the other direction:

- Let $P_1 \rightarrow_U P_2$, and let $I$ be a supported model of $P_1$. Since the unfolded rules are logical consequences of $P_1$, $I$ is of course a model of $P_2$. Now suppose that

\[
A \leftarrow L_1 \land \cdots \land L_{i-1} \land B \land L_{i+1} \land \cdots \land L_n
\]
was used to support $A$. This means that all body literals including $B$ are true in $I$, thus there is also a rule $B \leftarrow M_{j,1} \land \cdots \land M_{j,m_j}$ supporting $B$. But then the corresponding unfolded rule
\[
A \leftarrow L_1 \land \cdots \land L_{i-1} \land M_{j,1} \land \cdots \land M_{j,m_j} \land L_{i+1} \land \cdots \land L_n
\]
can also be used to support $B$.

- Let $P_1 \rightarrow_C P_2$ and let $I$ be a supported model of $P_1$. Since $P_2 \subseteq P_1$, $I$ is also a model of $P_2$. Furthermore, a rule containing two complementary body literals has always a false body in a normal interpretation, so it was not used to support an atom in $P_1$.

- Let $P_1 \rightarrow_C P_2$ and let $I$ be a supported model of $P_2$. Any normal interpretation satisfies a rule containing two complementary body literals, and since $P_2 \subseteq P_1$, no support can be lost.

- Let $P_1 \rightarrow P_2$ and let $I$ be a supported model of $P_1$. Let
\[
A \leftarrow L_1 \land \cdots \land L_{i-1} \land \text{not } B \land L_{i+1} \land \cdots \land L_n
\]
be the rule where the condition $\text{not } B$ was deleted because $B$ occurs in no rule head. This means that $I \models \text{not } B$, and since $I$ satisfies the above rule, it also satisfies the rule without the condition $\text{not } B$. If the rule was used to support $A$, the stronger rule can also be used.

- Let $P_1 \rightarrow_N P_2$ and let $I$ be a supported model of $P_1$. Let the deleted rule be
\[
A \leftarrow L_1 \land \cdots \land L_{i-1} \land \text{not } B \land L_{i+1} \land \cdots \land L_n,
\]
where $B \leftarrow \text{true}$ occurs in $P_1$. Since $P_2 \subseteq P_1$, $I$ is of course a model of $P_2$. No support can be lost, since $I \not\models \text{not } B$, so the deleted rule anyway cannot support $A$.

- Let $P_1 \rightarrow_S P_2$. Of course, $P_1$ and $P_2$ are logically equivalent, so any model $I$ of $P_1$ is also a model of $P_2$. Furthermore, whenever the weaker rule was used to support an atom $A$, also the stronger rule can be used.

- Since the definition of supported models does not depend on the sequence of body literals, normalization of rule bodies has no effect on them. \(\Box\)

The supported model semantics does not allow the deletion of tautologies, as the program $P := \{p \leftarrow p\}$ shows: Here, $p$ may be true in a supported model, while in the resulting empty program, $p$ must be false. Thus, SUPP is also not equivalence-preserving. Furthermore, the residual program is not equivalent to the original program for the supported model semantics. Nevertheless, there is a nice characterization:

**Theorem 4.2.44 (Characterization of Supported Models):**

- Let $S$ be a normal semantics which allows unfolding and the deletion of contradictions. Then $S(P) \subseteq \text{SUPP}(P)$ holds for all programs $P$.

- The supported model semantics SUPP has itself these properties, so it is the maximal normal semantics which allows unfolding and the deletion of contradictions.
4.2. A FRAMEWORK FOR STUDYING SEMANTICS

Proof:
Suppose that there were a program $P$ and an interpretation $I \in S(P)$ which is not a supported model of $P$. So there would be an $A \in BASE^+$ which has no support in $I$, but still is true in $I$. Let $B_1 \land \cdots \land B_n$ be the atoms occurring in $P$ and being true in $I$, and $C_1, \ldots, C_m$ be the atoms occurring in $P$ and being false in $I$ (note that we require all programs to be finite). Let $\bot \in BASE^+$ be an atom not occurring in $P$, and thus being false in $I$ ($A$ semantics must make all atoms not used in a program false, and there is always a new atom, because we require $BASE^+$ to be infinite\(^6\)). Then consider the following program $P'$:

$$P' := P \cup \{ \bot \leftarrow A \land B_1 \land \cdots \land B_n \land \text{not } C_1 \land \cdots \land C_m \}.$$  

Now apply unfolding to the first body literal $A$ (note that $A$ occurs again among the $B_i$). All rules of the form

$$A \leftarrow L_1 \land \cdots \land L_k$$

contain a body literal $L_i$ with $I \not\models L_i$, so $\sim L_i \in \{ B_1, \ldots, B_n, \text{not } C_1, \ldots, \text{not } C_m \}$, thus the rules resulting from unfolding can be removed by the deletion of contradictions. So we have $P' \rightarrow_U P'' \rightarrow_C P$. But this means $S(P') = S(P)$, which is impossible, since $I \in S(P)$, but $I$ does not satisfy the new rule in $P'$. \hfill \Box

As explained above, the supported model semantics is not consistency preserving, e.g. for $P := \{ \text{not } p \}$ we have $\text{SUPP}(P) = \emptyset$. Since every semantics with the above properties satisfies $S(P) \subseteq \text{SUPP}(P)$, there is no semantics with all four nice properties. We must give up at least one of them:

Corollary 4.2.45:
There is no normal and consistency-preserving semantics, which allows unfolding and the deletion of contradictions.

The Stable Model Semantics

The stable semantics allows all our transformations, so in this sense it is very well behaved. However, above we have already seen that even if we do not require the deletion of tautologies, we do not always get a model. So satisfying all these properties means also that the stable model semantics is not consistency-preserving.

Theorem 4.2.46 (Properties of the Stable Model Semantics):

STABLE is a strictly equivalence-preserving semantics. Thus, STABLE allows deletion of tautologies, unfolding, deletion of subsumed rules, and the normalization of rule bodies. STABLE also allows the deletion of contradictions as well as positive and negative reduction.

\(^6\)Note that this is really necessary. If $p$ would be the only atom, it is not difficult to see that there is a semantics with these properties which allows the model $\langle p \text{ true} \rangle$ for $P := \{ p \leftarrow \text{not } p \}$. This is not a supported model.
CHAPTER 4. NEGIATION AS FAILURE

Proof:

- Obviously, STABLE is a semantics in the above sense: If an atom $A$ does not appear in $P$, it is false in all minimal models $I$ of $P$, and since $I$ is normal, we also have $I \models \neg A$.
- Since the definition of the stable model semantics only refers to the minimal models of $P$ and not to the syntax of $P$, STABLE is strictly equivalence-preserving.
- Let $P_1 \hookrightarrow_C P_2$.

  If $I$ is a stable model of $P_1$, it is also a model of $P_2$ (since $P_2 \subseteq P_1$). If there were a preferable model $I_0$ of $P_2$, this would also satisfy the deleted rule: Only an interpretation which is not semi-normal could violate it, but $I$ is normal and $I_0$ makes even less positive literals true. So $I_0$ would be a model of $P_1$, contradicting the assumed minimality of $I$.

  If $I$ is a stable model of $P_2$, it is also a model of $P_1$ (since every normal interpretation satisfies a rule with complementary body literals). But if there were a preferable model $I_0$ of $P_1$, this would also be a model of $P_2$ (since $P_2 \subseteq P_1$), and $I$ could not be minimal as a model of $P_2$.

- Let $P_1 \hookrightarrow_P P_2$ an let the simplified rule be

  $$A \leftarrow L_1 \land \cdots \land L_{i-1} \land \neg B \land L_{i+1} \land \cdots \land L_n,$$

  where $B$ occurs in no rule head.

  Let $I$ be a stable model of $P_1$. Since obviously $I \models \neg B$, $I$ is a model of $P_2$. Since $P_2 \subseteq P_1$, every model of $P_2$ is also a model of $P_1$, so if there were a preferable model of $P_2$, $I$ would not be minimal as a model of $P_1$.

  Conversely, let $I$ be a stable model of $P_2$. It is clear that $I$ is also a model of $P_1$. Furthermore, $I \models \neg B$, so if there were a preferable model of $P_1$, this would also be a model of $P_2$, contradicting again the minimality of $I$.

- Let $P_1 \hookrightarrow_N P_2$ and let the deleted rule be

  $$A \leftarrow L_1 \land \cdots \land L_{i-1} \land \neg B \land L_{i+1} \land \cdots \land L_n,$$

  where $B \leftarrow \text{true}$ is contained in $P_1$ (and thus in $P_2$).

  Let $I$ be a stable model of $P_1$. Since $P_2 \subseteq P_1$, $I$ is also a model of $P_2$. If there were a preferable model $I_0$ of $P_2$, this would also be a model of $P_1$, since $I \not\models \neg B$ and thus $I_0 \not\models \neg B$. This would contradict the assumed minimality of $I$.

  Let $I$ be a stable model of $P_2$. Then $I \not\models \neg B$, thus $I$ is also a model of $P_1$. If there were a preferable model $I_0$, this would be also a model of $P_2$ (since $P_2 \subseteq P_1$), contradicting again the minimality of $I$.

Our characterization of the stable models can be derived from the characterization of the supported models by adding the deletion of tautologies. So we first prove that stable models are also supported:

Lemma 4.2.47:

- For arbitrary programs $P$ holds: $\text{STABLE}(P) \subseteq \text{SUPP}(P)$.
- If $P$ is a set of conditional facts, then $\text{STABLE}(P) = \text{SUPP}(P)$.
4.2. A FRAMEWORK FOR STUDYING SEMANTICS

Proof:

- Suppose that \( I \models A \), but there is no rule \( A \leftarrow L_1 \land \cdots \land L_n \) in \( P \) with \( I \models L_i \) \((i = 1, \ldots, n)\). Let \( I_0 \) the interpretation with \( I_0 \not\models A \), and \( I_0 \models L \) \( \iff I \models L \) for all other literals \( L \). Then \( I_0 \) obviously does not violate any rule of \( P \), so it is a model of \( P \), and \( I_0 \not\models I \), \( I_0 \not= I \), thus \( I \) is not minimal.

- We only have to show \( \text{SUPP}(P) \subseteq \text{STABLE}(P) \), where \( P \) is a set of conditional facts. Let \( I \) be a supported model of \( P \), and suppose that \( I \) were not minimal, so there would be a model \( I_0 \) of \( P \) with \( I_0 \not\models I \) and \( I_0 \not= I \). Thus, there is \( A \in \text{BASE}^+ \) with \( I \models A \) and \( I_0 \not\models A \). Since \( I \) is a supported model of \( P \), there is a conditional fact of the form \( A \leftarrow L_1 \land \cdots \land L_n \) with \( I \models L_i \), \( i = 1, \ldots, n \). However, since the \( L_i \) are negative literals, they have the same truth value in \( I \) and \( I_0 \). But then \( I_0 \) violates the rule \( A \leftarrow L_1 \land \cdots \land L_n \), i.e. is no model of \( P \).

Theorem 4.2.48 (Characterization of Stable Models):

- Let \( S \) be a normal semantics which allows unfolding and the deletion of contradictions and tautologies. Then \( S(P) \subseteq \text{STABLE}(P) \) holds for all programs \( P \).

- The stable model semantics \( \text{STABLE} \) has itself these properties, so it is the maximal normal semantics which allows unfolding and the deletion of contradictions and tautologies.

Proof:

- Let \( P \) be any logic program. By Theorem 4.2.19, there is a set of conditional facts \( F \) with \( P \rightarrow^* F \). Since \( S \) allows the transformation \( \rightarrow^* \) (the union of unfolding and deletion of tautologies), it follows that \( S(P) = S(F) \). Since \( S \) allows unfolding and the deletion of contradictions, and is a normal semantics, Theorem 4.2.44 implies \( S(F) \subseteq \text{SUPP}(F) \). Now Lemma 4.2.47 gives \( \text{SUPP}(F) = \text{STABLE}(F) \). But \( \text{STABLE} \) also allows unfolding and the deletion of tautologies (Theorem 4.2.46), thus Theorem 4.2.19 yields \( \text{STABLE}(F) = \text{STABLE}(P) \).

- That \( \text{STABLE} \) has these properties was already shown in Theorem 4.2.46.

The Well-Founded Model

The well-founded semantics satisfies all our properties except the deletion of contradictions. This is not surprising, since the well-founded model is in general not normal, i.e. can have “undefined” atoms. On the other hand, the well-founded semantics yields always a unique model, i.e. it is consistency-preserving and definite.

Theorem 4.2.49 (Properties of the Well-Founded Semantics):

The well-founded semantics is strictly equivalence-preserving, and thus allows unfolding, deletion of tautologies and subsumed rules, and the normalization of rule bodies. It also allows positive and negative reduction.
CHAPTER 4. NEGATION AS FAILURE

Proof:

- Since the WFS operator depends only on the minimal models of $P$, it is clear that the well-founded semantics is strictly equivalence-preserving.
- Positive reduction: Let $P_1 \rightarrow_P P_2$ and let

\[ A \leftarrow L_1 \land \cdots \land L_{i-1} \land \text{not } B \land L_{i+1} \land \cdots \land L_n \]

be the rule in which the condition \text{not } $B$ was deleted since $B$ does not occur in any rule head. We now show that a semi-normal interpretation $I$ is a fixpoint of $T_{P_1}^{\text{wf}}$ if it is a fixpoint of $T_{P_2}^{\text{wf}}$. The well-founded model is semi-normal by Proposition 4.1.23, and if there were a $\subset$-smaller fixpoint, it would also be semi-normal. Thus, this implies that the least fixpoints of $T_{P_1}^{\text{wf}}$ and $T_{P_2}^{\text{wf}}$ agree.

Let $I$ be a semi-normal fixpoint of $T_{P_1}^{\text{wf}}$. Since \text{not } $B$ is immediately derived by $T_{P_1}^{\text{wf}}$ (it is true in all minimal models, no matter which valuation of the negative literals we use), it follows that $I \models \text{not } B$, and thus $I \not\models B$. So $I_{\text{bot}}$ and $I_{\text{top}}$ used in the construction of $T_{P_1}^{\text{wf}}(I)$ both make not $B$ true. The same valuation for the negative literals is used in the construction of $T_{P_2}^{\text{wf}}(I)$. Since $I_{\text{bot}}$ and $I_{\text{top}}$ are models of $P_1$ and make \text{not } $B$ true, they are also models of $P_2$. Furthermore $P_2 \models P_1$, thus they are also minimal models (any preferable model would also be a model of $P_1$ contradicting their minimality). So the minimal models $I_{\text{bot}}$ and $I_{\text{top}}$ used in the construction of $T_{P_1}^{\text{wf}}(I)$ agree with the corresponding models used in the construction of $T_{P_2}^{\text{wf}}(I)$, thus $T_{P_1}^{\text{wf}}(I) = T_{P_2}^{\text{wf}}(I) = I$.

The opposite direction is completely analogous.

- The proof for negative reduction is very similar (using $I_{\text{bot}} \not\models \text{not } B$ and $I_{\text{top}} \not\models \text{not } B$).

$\square$

Once we have the residual program, the computation of the well-founded model is trivial, since it makes all “hard cases” undefined:

**Theorem 4.2.50 (Computation of the Well-Founded Model):**

Let $I$ be the well-founded model of a logic program $P$. Then for all $A \in \text{BASE}^+$, the following holds:

- $I \models A$ iff $A \leftarrow \text{true} \in \text{res}(P)$,
- $I \models \text{not } A$ iff $A$ does not occur in any rule head of $\text{res}(P)$.

**Proof:**

Since the well-founded semantics allows $\rightarrow_1$, the well-founded model of the residual program $\text{res}(P)$ is equal to the well-founded model of the original program $P$. But for the residual program, it is easy to see that the interpretation $I$ defined by

- $I \models A$ iff $A \leftarrow \text{true} \in \text{res}(P)$,
- $I \models \text{not } A$ iff $A$ does not occur in any rule head of $\text{res}(P)$.

is indeed the well-founded model of $\text{res}(P)$. First, $I$ is a fixpoint of $T_{\text{res}(P)}^{\text{wf}}$:

- If we assume only the negative literals true in $I$, we can only derive the positive literals true in $I$, since all rule bodies in $\text{res}(P)$ are false in $I$ (a negative literal true in $I$ occurs in no rule body).
Now consider $I_{\text{top}}$, which makes all negative literals $\text{not } A$ true except when $A$ is true in $I$, i.e. $A$ occurs as a fact $A \leftarrow \text{true}$ in $\text{res}(P)$. Then we can derive all positive literals $B$ except when $\text{not } B$ is true in $I$: Every atom $B$ such that $\text{not } B$ is not true in $I$ occurs in a rule head, where every negative literal $\text{not } A$ in the body is not excluded by a fact $A \leftarrow \text{true}$.

Since already the first iteration of $T_{\text{res}(P)}$ yields $I$, it is the least fixpoint, i.e. the well-founded model. □

This also explains that the well-founded semantics is the weakest under all semantics allowing the normalization to the residual program:

**Theorem 4.2.51 (Characterization of the Well-Founded Model):**

Let $S$ be a semantics which allows unfolding, deletion of tautologies, positive and negative reduction, deletion of subsumed rules, and normalization of rule bodies.

- If $A \in \text{BASE}^+$ is true in the well-founded model of a logic program $P$, then $I \models A$ for all $I \in S(P)$.
- If $\text{not } A$ is true in the well-founded model of a logic program $P$, then $I \models \text{not } A$ for all $I \in S(P)$.

**Proof:**

For any semantics $S$ allowing $\rightarrow_4$ we have $S(P) = S(\text{res}(P))$. But for residual programs, the well-founded semantics is obviously the weakest possible semantics:

- $A \in \text{BASE}^+$ is only true in the well-founded model if $A \leftarrow \text{true}$ occurs as a fact, so any other semantics must also make $A$ true.
- $\text{not } A$ is only true in the well-founded model if $A$ does not occur at all in the residual program, so again any semantics must make $\text{not } A$ true. □

**Summary**

The properties of the semantics investigated in this section are summarized in the following table:

<table>
<thead>
<tr>
<th>Property</th>
<th>SUPP</th>
<th>STABLE</th>
<th>WFS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal Interpretations</td>
<td>●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Equivalence-Preserving</td>
<td></td>
<td>●</td>
<td></td>
</tr>
<tr>
<td>Consistency-Preserving</td>
<td></td>
<td></td>
<td>●</td>
</tr>
<tr>
<td>Definite</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unfolding</td>
<td>●</td>
<td>●</td>
<td></td>
</tr>
<tr>
<td>Deletion of Tautologies</td>
<td></td>
<td>●</td>
<td></td>
</tr>
<tr>
<td>Deletion of Contradictions</td>
<td>●</td>
<td>●</td>
<td></td>
</tr>
<tr>
<td>Positive Reduction</td>
<td>●</td>
<td>●</td>
<td></td>
</tr>
<tr>
<td>Negative Reduction</td>
<td>●</td>
<td>●</td>
<td></td>
</tr>
<tr>
<td>Deletion of Nonminimal Rules</td>
<td>●</td>
<td>●</td>
<td></td>
</tr>
<tr>
<td>Normalization of Rule Bodies</td>
<td>●</td>
<td>●</td>
<td></td>
</tr>
</tbody>
</table>
We believe that our approach can also be used to analyze and characterize other important semantics such as the three-valued stable models. This is subject of our future research.
4.3 Bottom-Up Query Evaluation

In this section, we explain how to compute the residual program and how to use it for query evaluation under the well-founded semantics. As explained above, the residual program is still equivalent to the original program for a wide range of semantics, so it might also be useful for computing other semantics. In Section 5.2, we will show how to compute the stable semantics based on the residual program. This has to be delayed until Chapter 5 since it needs disjunctions.

We assume in this section that we are given a finite and strictly range-restricted program $P$ (including the tuples of the database as facts). Of course, in contrast to Section 4.2, we do not assume that $P$ is propositional or ground. It would be very inefficient to instantiate the program before query evaluation.

The Residual Program

Let us start with a quick review of the residual program and its properties (and thereby generalize it to non-propositional programs). As explained in Section 4.2, the residual program is a set of conditional facts, i.e. facts depending on a set of “delayed” negative literals. They are formally nothing else the rules with a purely negative body, such as

$$\text{winning}(2) \leftarrow \text{not winning}(1).$$

Note that because of the range-restriction, conditional facts are always ground. Conditional facts have been introduced by BRY [Bry89, Bry90a], and independently by DUNG and KANCHANSUT [DK89a, DK89b]. They are a very useful and natural notion and also implicitly used in several query evaluation algorithms.

In order to have a stronger duplicate elimination, we treat the rule bodies as sets. This is also formally simpler:

**Definition 4.3.1 (Normalized Conditional Fact):**

Let $\prec$ be any fixed linear order on the set of ground literals. A normalized conditional fact is a ground rule $A \leftarrow L_1 \land \cdots \land L_n$ where all body literals are negative, and

- the body literals are ordered according to $\prec$, i.e. $L_i \prec L_j \implies i < j$,
- there are no duplicate body literals, i.e. $L_i = L_j \implies i = j$.

Thus we can identify the rule body with the set $B = \{L_1, \ldots, L_n\} \subseteq \text{BASE}($NEG$)$.

The immediate consequence operator for conditional facts is very similar to the standard $T_P$ operator, it only does not evaluate the negative literals, but instead attaches them to the derived facts. For example, the following is a typical derivation step:

$$
p(a) \leftarrow \text{not } s(b) \land \text{not } r(b).
\uparrow
\uparrow
\uparrow
\{ p(X) \leftarrow q_1(X) \land q_2(X, Y) \land \text{not } r(Y). \}
\uparrow
\uparrow
\uparrow
q_1(a) \quad q_2(a, b) \leftarrow \text{not } s(b).
$$
In general, the derivation of conditional facts can be formalized as follows. Note that it refers only to the ground instances of program rules, so the same conditional facts are derivable from $P$ and $\text{ground}(P)$. This is important in order to apply the results of Section 4.2. However, the implementation of the derivation operator for $P$ and $\text{ground}(P)$ is of course very different. Since $\text{ground}(P)$ is usually much larger than $P$, it is essential to work with $P$ and to compute only the needed ground instances (having body literals for which we can insert matching conditional facts).

**Definition 4.3.2 (Bottom-Up Evaluation with Conditional Facts):**
Given a program $P$, we define the following operator $T_P$ on sets of normalized conditional facts:

$$T_P(F) := \{ A \theta \leftarrow \bigcup_{i=1}^{n} B_i \mid \text{there is a rule } A \leftarrow L_1 \land \cdots \land L_n \text{ in } P \text{ and a ground substitution } \theta \text{ for this rule} \}
\begin{align*}
&\text{such that for } i = 1, \ldots, n:
&\quad \text{if } L_i \text{ is a positive literal then } \\
&\quad \quad L_i \theta \leftarrow B_i \text{ is a conditional fact in } F \\
&\quad \text{else } L_i \text{ is a negative literal and } B_i = \{ L_i \theta \}.
\end{align*}$$

Since we have assumed that the given program $P$ is finite, there are only finitely many predicates and constants which can occur in the conditional facts derivable from $P$. Thus, the least fixpoint of this operator is reached after a finite number of iterations.

From this set of derivable conditional facts we get the residual program by iteratively applying an operator performing the following three reductions:

- Delete a conditional fact $A \leftarrow B$ if it is non-minimal, i.e. there is a conditional fact $A \leftarrow B'$ with $B' \subset B$.
- Delete a conditional fact $A \leftarrow B$ if $B$ contains a condition $\text{not } p(c_1, \ldots, c_n)$ which is known to be false because $p(c_1, \ldots, c_n) \leftarrow \text{true}$ is already derived.
- Remove a condition $\text{not } p(c_1, \ldots, c_n)$ from a conditional fact if $p(c_1, \ldots, c_n)$ occurs in no head of a derived conditional fact, i.e. the condition is known to be true.

**Definition 4.3.3 (Reduction of Conditional Facts):**
We define the following reduction operator on sets of normalized conditional facts:

$$R(F) := \{ A \leftarrow (B \cap \text{delay}(F)) \mid A \leftarrow B \in F, \text{ there is no } A \leftarrow B' \text{ in } F \text{ with } B' \subset B, \text{ and } \}
\begin{align*}
&\quad (\neg L) \leftarrow \text{true} \notin F \text{ for every } L \in B, \}
\end{align*}$$

where $\text{delay}(F) := \{ \text{not } A \mid A \leftarrow B \in F \}$ is the set of negative literals not currently known to be true, because they occur in a rule head in $F$.

All changes done by this reduction operator strictly reduce the total number of literals. So we reach a fixpoint after finitely many steps. This is the residual program of $P$:...
Definition 4.3.4 (Residual Program):
Let a program \( P \) be given and let \( F_0 := \text{lfp}(T_P) \) and \( F_i := R(F_{i-1}) \). The residual program of \( P \) is \( \text{res}(P) := F_n \) with \( n \in \mathbb{N} \) such that \( F_n = F_{n-1} \).

This generalizes Definition 4.2.35, from propositional programs to arbitrary Datalog\(^\text{neg}\) programs. Furthermore, the residual program is invariant under instantiation, so with the above construction, we get the same result as with the old Definition applied to \( \text{ground}(P) \). Furthermore, we know by Proposition 4.1.26, that \( P \) and \( \text{ground}(P) \) have the same well-founded and stable models. This allows us to apply the results of Section 4.2.

As shown in Theorem 4.2.50, the well-founded model can be directly read from the residual program. The well-founded semantics is so weak that it makes a fact \( A \) only true if \( A \) absolutely must be true because it is contained unconditionally in the residual program. And a negative literal \( \text{not} \ A \) is only true in the well-founded model if \( A \) does not occur in the residual program. For all other facts \( A \), both \( A \) and \( \text{not} \ A \) are false, i.e. \( A \) is “undefined” in the view of three-valued logic.

Proposition 4.3.5 (Computation of the Well-Founded Model):
Let a program \( P \) be given, and let \( I \) be its well-founded model.
- For every positive literal \( A \): \( I \models A \ iff \ A \leftarrow true \in \text{res}(P) \).
- For every negative literal \( \text{not} \ A \): \( I \models \text{not} \ A \ iff \ there \ is \ no \ conditional \ fact \ of \ the \ form \ A \leftarrow B \ in \ \text{res}(P) \).

Also for the stable model semantics, the computation of the residual program is a useful preprocessing step, since it is still equivalent to the original program, but much simpler, because
- it is ground,
- it contains only negative body literals, and
- the truth value of most literals can already be read from the residual program, only small “islands of complexity” [MD93] remain.

The following proposition follows from Theorems 4.2.36 and 4.2.46:

Proposition 4.3.6 (Stable Models of Residual Program):
A Herbrand interpretation \( I \) is a stable model of \( P \) iff it is a stable model of \( \text{res}(P) \).

In order to use the results of Section 4.2, it was necessary to allow only finite programs. However, it seems well possible that the above propositions hold also in the infinite case. We must leave this as a topic for future research.

Efficient Computation of the Residual Program
The computation of the derivable conditional facts \( \text{lfp}(T_P) \) can be done with the standard techniques for Horn clauses presented in Section 2.3. Formally, we can
simply encode the conditions in an additional set-valued argument. For instance, the rule
\[ p(X) \leftarrow q(a) \land r(X, Y) \land \textbf{not} \ s(Y) \]
can be translated to
\[ p(X, B) \leftarrow q(a, B_1) \land r(X, Y, B_2) \land \text{union}(B_1, B_2, B_3) \land \text{union}(B_3, \{ \textbf{not} \ s(Y) \}, B). \]
The CORAL system allows set-valued arguments and it is easily possible to compute \( \text{lfp}(T_P) \) in this way. In fact, if we know the structure of the conditions, we sometimes do not even need set-valued arguments. For instance the classical “winning state” example
\[ \text{winning}(X) \leftarrow \text{move}(X, Y) \land \textbf{not} \ \text{winning}(Y) \]
can be translated into
\[ \text{winning}_{\text{if_not_winning}}(X, Y) \leftarrow \text{move}(X, Y), \]
where the head represents the conditional fact \( \text{winning}(X) \leftarrow \textbf{not} \ \text{winning}(Y) \) (for given values of \( X \) and \( Y \)).

We will not repeat the material from Section 2.3 here, but instead concentrate on the reduction operator \( R \). There are three main possibilities to improve the efficiency of the computation of the residual program:

First, the deletion of non-minimal conditional facts should be done already during the derivation of conditional facts. This is possible since any conditional fact derived by using a non-minimal conditional fact is again non-minimal, so it will later anyway be deleted by the reduction operator. But it is of course better to avoid deriving it in the first place. The same applies to conditional facts with conditions already known to be false. So we suggest to replace the immediate consequence operator for conditional facts by a version which already performs the first two reductions:

**Definition 4.3.7 (Derivation of Conditional Facts with Weak Reduction):**
The weak reduction operator on sets of normalized conditional facts is:
\[
R_0(F) := \left\{ A \leftarrow B \mid \begin{array}{l}
A \leftarrow B \in F, \\
\text{there is no } A \leftarrow B' \text{ in } F \text{ with } B' \subseteq B, \text{ and} \\
(\sim L) \leftarrow \text{true} \notin F \text{ for every } L \in B
\end{array} \right\};
\]
then the restricted immediate consequence operator is \( \hat{T}_P := R_0 \circ T_P \).

Deleting unnecessary facts makes the immediate consequence operator nonmonotonic \( \subseteq \); however, it is monotonic in the lattice based on
\[
F_1 \subseteq F_2 :\iff \text{ for every } A_1 \leftarrow B_1 \text{ in } F_1 \text{ there is } A_2 \leftarrow B_2 \text{ in } F_2 \text{ such that } (A_1 = A_2 \text{ and } B_2 \subseteq B_1) \text{ or } (B_2 = \emptyset \text{ and } \sim A_2 \subseteq B_1).
\]
Since the suppressed conditional facts and any conditional facts derived with them would have been anyway deleted through the reduction phase, the following is clear:
Proposition 4.3.8 (Restricted Immediate Consequences):
Let a program $P$ be given and let $F_0 := \text{lfp}(\bar{T}_P)$, $F_i := R(F_{i-1})$, and $n \in \mathbb{N}$ such that $F_n = F_{n-1}$. Then $F_n = \text{res}(P)$.

Note that in the second phase we still need the full reduction operator, because by evaluating negative literals to true (i.e. deleting them from the conditions), the other two reductions can become applicable again.

The second possibility to speed up the computation of the residual program is to make use of a rule order as introduced in Section 2.3. We would of course like to evaluate a negative literal $\text{not } p(c_1, \ldots, c_n)$ to true already during the derivation of conditional facts. But for this to be possible, we must know that no conditional fact of the form $p(c_1, \ldots, c_n) \leftarrow B$ can be derived, also not later in the proof. This is possible if we do not use a single iteration to derive all conditional facts, but instead order the rules according to the predicate dependencies and iterate only “recursive cliques”. If $p$ is not defined in the current recursive clique, then all rules about $p$ have already been evaluated when the condition $\text{not } p(c_1, \ldots, c_n)$ occurs in a rule body. So, if there is no conditional fact of the form $p(c_1, \ldots, c_n) \leftarrow B$, we know that it also cannot be derived later. Thus, we can safely evaluate $\text{not } p(c_1, \ldots, c_n)$ to true.

For stratified programs, all negative literals $\text{not } A$ can immediately be evaluated, either because there is the positive fact $A \leftarrow \text{true}$ or because there is no conditional fact $A \leftarrow B$ and we know that there also will be no such conditional fact. So we never have to delay any negative literals and get the standard bottom-up computation of the perfect model.

But even for non-stratified programs, there are often only very few and small non-stratified recursive cliques, and we need to delay only their literals. However, when we derive later facts depending on them, these facts can also have conditions. But the reduction can be done immediately since we already know all derivable conditional facts about predicates which are not mutually recursive to the head of the current rule.

The third thing to do in order to improve the efficiency of the reduction is to study specialized data structures. We will do so in the rest of this section. Suppose that we have derived a new conditional fact $A \leftarrow B$ and we want to insert it into the current set $F$ of conditional facts, which we always keep reduced according to $R_0$. So we assume that $R_0(F) = F$, and we want to compute $R_0(F \cup \{A \leftarrow B\})$.

The elimination of conditional facts containing a condition already known to be false is straightforward. If $B = \emptyset$, we delete from $F$ all conditional facts containing $\text{not } A$ in the body. So we need an efficient implementation of

$$\text{depends}_{\text{on}}(A) := \{A' \leftarrow B' \in F \mid (\text{not } A) \in B'\}.$$  

This can of course be done with an index over the conditional facts. If we can keep the complete set $F$ in main memory, it is probably advantageous to translate all ground literals occurring in $F$ into array indices/pointers (e.g., via a hash table), and then to represent the literals in the conditional facts only by these indices.
If $B \neq \emptyset$, we need to check whether one of the negative literals $L \in B$ contradicts an unconditional fact ($\sim L \leftarrow true$) in $F$. In fact, if conditional facts are immediately deleted as soon as their condition is known to be false, this test is only needed for those $L \in B$ which were produced by negative literals in the rule itself, since the conditions of the facts inserted for positive body literals are known to be free of definitely false literals. In either case, we need an efficient implementation of

$$known\_true(A) := (A \leftarrow true) \in F.$$  

If we represent ground atoms internally as indices or pointers, this can be implemented by a boolean attribute in the ground atom table.

Now let us consider the minimality check. It is possible that the new fact $A \leftarrow B$ is subsumed by a conditional fact $A \leftarrow B'$ in $F$ (i.e. $B' \subseteq B$), or that $A \leftarrow B$ itself subsumes an older fact $A \leftarrow B'$ in $F$ (i.e. $B \subseteq B'$).

The straightforward algorithm is to look at all conditional facts $(A \leftarrow B') \in F$ with the given head literal $A$, and then to check whether each literal $L \in B$ is contained in $B'$ or vice versa. If we store the conditions as ordered lists, this is possible in $O(|B| + |B'|)$ time. Ordered lists are anyway useful for computing the condition of a derived conditional fact as the union of the conditions of the facts inserted for body literals.

It is possible to find nonminimal conditional facts quicker if we have a hash table which allows efficient access to

$$contains\_dependency(A, L) := \{(A \leftarrow B) \in F \mid L \in B\}.$$  

Let us define the “overlap” between two conditional facts $A \leftarrow B$ and $A \leftarrow B'$ as the number of common negative literals, i.e. $|B \cap B'|$. Furthermore, let the length of a conditional fact $A \leftarrow B$ be its number of negative literals, i.e. $|B|$. Then we utilize

$$B \subseteq B' \iff |B \cap B'| = |B|$$  

(and symmetrically for $B' \subseteq B$). With our hash table contains\_dependency it is easy to determine the overlap of every $A \leftarrow B'$ in $F$ with the new conditional fact $A \leftarrow B$ (whenever this is at least 1).

In order to compute the overlap, we need a counter overlap$[A \leftarrow B']$ for every conditional fact in $F$. Now, for every literal $L \in B$, we look at all conditional facts $(A \leftarrow B') \in contains\_dependency(A, L)$ and increment their overlap counter. If we have reached the length of either $A \leftarrow B$ or $A \leftarrow B'$, the other conditional fact is a duplicate. Of course, it is not possible to initialize all overlap counters to 0 before every test. This problem is solved by storing together with every overlap counter the time (test number) at which it was last set (this is a kind of “lazy initialization”).

Of course, the special cases $B = \emptyset$ and $B' = \emptyset$ must be treated separately (which is anyway advantageous, because it is especially simple and probably occurs very often). For this we need the above mentioned function known\_true plus an efficient access to all conditional facts in $F$ with a given head literal:

$$defines(A) := \{(A' \leftarrow B) \in F \mid A' = A\}.$$  

Figure 4.4 shows the data we have to manage in ER-notation.
var $F := \emptyset$;
var $test\_no := 0$;

procedure insert($A \leftarrow B$):

  /* Check whether already known as fact: */
  if \texttt{known\_true}(A) then return fi;

  if $B = \emptyset$ then
    /* Make $A \leftarrow true$ the only fact about $A$: */
    $F := (F - \texttt{defines}(A)) \cup \{A \leftarrow true\};$

    /* Delete facts containing condition not $A$: */
    foreach ($A' \leftarrow B'$) \in \texttt{depends\_on}(A) do
      $F := F - \{A' \leftarrow B'\};$
    od;
  else
    /* Are the conditions known to be false? */
    foreach $L \in B$ do
      if \texttt{known\_true}($\sim L$) then return fi;
    od;

    /* Check Minimality */
    $test\_no := test\_no + 1;$
    foreach $L \in B$ do
      foreach ($A \leftarrow B'$) \in \texttt{contains\_dependency}(A, L) do
        /* Increment or initialize overlap counter: */
        if \texttt{last\_set}[$A \leftarrow B'$] = $test\_no$ then
          $\texttt{overlap}[A \leftarrow B'] := \texttt{overlap}[A \leftarrow B'] + 1;$
        else
          \texttt{last\_set}[$A \leftarrow B'$] := $test\_no$;
          $\texttt{overlap}[A \leftarrow B'] := 1;$
        fi;
      /* Check Inclusion: */
      if $\texttt{overlap}[A \leftarrow B'] = |B'|$ then return fi;
      if $\texttt{overlap}[A \leftarrow B'] = |B|$ then
        $F := F - \{A \leftarrow B'\};$
      fi;
    od;
  fi;

  /* Insert it: */
  $F := F \cup \{A \leftarrow B\};$
fi;

Figure 4.3: Efficient Implementation of Reductions
With this knowledge about the implementation of $R_0$ and the material of Section 2.3, it should be possible to compute $\text{lfp}(\hat{T}_P)$ efficiently. We now must look at the second phase of the computation of the residual program, where negative literals are evaluated to true. For this purpose, we need an efficient access to the ground atoms which occur in the conditions but in no rule head:

$$\text{known\_false} := \{ A \mid \text{there is } A' \leftarrow B' \text{ in } F \text{ with } \textbf{not} A \in B', \text{ but there is no conditional fact of the form } A \leftarrow B \text{ in } F \}.$$ 

This set is updated during the insertions and deletions of conditional facts in/from the set $F$. In order to simplify this, it is useful to maintain for every ground atom $A$ occurring in $F$ a counter $\text{num\_pos}[A]$ for the number of conditional facts about $A$. Then it is easy to do the reductions:

```plaintext
procedure reduce:
    while $\text{known\_false} \neq \emptyset$ do
        let $A \in \text{known\_false}$;
        foreach $(A' \leftarrow B') \in \text{depends\_on}(A)$ do
            $F := F \setminus A' \leftarrow B'$;
            $\text{insert} \left( A' \leftarrow (B' - \text{known\_false}) \right)$;
        od;
        $\text{known\_false} := \text{known\_false} \setminus \{A\}$;
    od;
```

Figure 4.4: Information about Conditional Facts
Note that we call the above procedure \textit{insert} in order to do the other reductions while the simplified fact \( A' \leftarrow (E' - \text{known\_false}) \) is inserted into \( F \). This might also lead to new facts inserted into \text{known\_false}, but of course, the termination is ensured because no fact is inserted twice into \text{known\_false}. Once \( A \) has been processed by the above procedure \textit{reduce}, it is completely removed from all conditional facts in \( F \).

In summary, with all of the optimizations discussed here, the computation of the residual program should be efficiently possible. Especially, the mentioned access paths ensure that we never have to go through the complete set \( F \). We also believe that our algorithm to eliminate subsumed conditional facts has a good performance. This is important, because experiments of SEIPEL have shown that the elimination of duplicate and subsumed disjunctive facts (which is the same problem of finding supersets and subsets) can take more than half of the total running time of query evaluation [Sei94]. Further improvements are subject of future research (for instance, it might be possible to generalize data structures for partial match queries on relations).

**Comparison with Alternating Fixpoint Computation of WFS**

In [KSS91], Kemp, Stuckey, and Srivastava have proposed a bottom-up algorithm for computing the well-founded model. It is based on Van Gelder’s alternating fixpoint construction [VG89]. Let us compare the this method with our\(^{7}\) approach based on the residual program.

The alternating fixpoint method computes possibly true and surely true facts in an alternating sequence. This is quite similar to the construction of the well-founded model presented in Section 4.1. Given a Herbrand interpretation \( I_{\text{sure}} \) containing positive ground atoms known to be true (at the beginning \( I_{\text{sure}} = \emptyset \)), we can compute a set of possibly true facts by assuming that all negative ground literals not \( A \) with \( A \notin I_{\text{sure}} \) are satisfied (i.e. in the beginning all negative conditions are simply removed from the rules). Remember that the more negative ground literals we assume, the more positive ground literals are derivable. So in this way we get an upper bound of the really true facts. Given an interpretation \( J \) for determining the truth values of the negative literals (e.g., \( J = I_{\text{sure}} \)), we can compute the positive ground literals true in the minimal model by the following immediate consequence operator:

\[
T_{P,J}(I) := \{ A\theta \mid \text{there is a rule } A \leftarrow L_1 \land \cdots \land L_n \text{ in } P \\
\text{and a ground substitution } \theta \text{ for this rule such that for } i = 1, \ldots, n:\n\text{if } L_i \text{ is a positive literal then } L_i\theta \in I \\
\text{else } L_i \text{ is a negative literal and } \neg L_i\theta \notin J \};
\]

Now we get a first set \( I_{\text{poss}} := \text{lfp}(T_{P,I_{\text{sure}}}) \) of possibly true facts. Its complement is a lower bound of the negative ground literals satisfied in the well-founded model. So if we use this set to determine the truth of the negative literals, we get a lower bound

---

\(^{7}\)As mentioned before, François Bry has introduced conditional facts as well as the immediate consequence and reduction operators in [Bry90a]. We have shown that this allows to compute the well-founded model and many other semantics.
of the true facts: \( I_{\text{sure}} := \text{lfp}(T_P, I_{\text{poss}}) \). This alternating computation of \( I_{\text{poss}} \) and \( I_{\text{sure}} \) is repeated until a fixpoint is reached. Then \( I_{\text{sure}} \) contains the positive ground literals true in the well-founded model and \( I_{\text{poss}} \) contains the complement of the negative ground literals satisfied by the well-founded model. So the difference \( I_{\text{poss}} - I_{\text{sure}} \) contains the “undefined” literals.

It is easily possible to implement this computation with the standard techniques for bottom-up evaluation (see Section 2.3) if we introduce two versions \( p_{\text{poss}} \) and \( p_{\text{sure}} \) of every predicate. This was suggested in [KSS91] (they also introduced a “magic set” transformation for the well-founded semantics). In fact, [KSS91] start their computation with \( I_{\text{sure}} \) (by considering only the rules without negative body literals), but this is only an optimization of the basic approach, which we will ignore in the following.

Now let us consider the relation to the residual program. When \( I_{\text{poss}} \) is computed for the first time, we assume that all negative literals are satisfied (because \( I_{\text{sure}} = \emptyset \)). So in effect the negative conditions are removed from the rules. During the computation of the derived conditional facts, also no negative condition prevents a rule application, but we remember the negative literals on which a derived fact depends. So the heads of our derived conditional facts are exactly the atoms contained in the first version of \( I_{\text{poss}} \).

Now in the second round, when \( I_{\text{sure}} \) is computed, the negative literals \( \text{not } A \) with \( A \not\in I_{\text{poss}} \) are evaluated to true. But exactly this also happens when our reduction operator \( R \) is applied. So the set \( I_{\text{sure}} \) is the set of unconditional atoms \( A \leftarrow \text{true} \) contained in \( R(\text{lfp}(T_P)) \).

In general, our reduction operator \( R \) is a little “faster”, because it does positive and negative reduction at the same time (the elimination of nonminimal facts is not important here, see below). But if we would do positive and negative reduction in an alternating sequence, we get an exact correspondence to the alternating fixpoint approach.

So the alternating fixpoint approach can be understood as reducing the information contained in the conditions of our conditional facts to single bit: Is there a nontrivial condition (i.e. \( B \neq \emptyset \)) or not?

- The relations \( p_{\text{sure}} \) contain the conditional facts with condition true.
- The relations \( p_{\text{poss}} \) represent the heads of all conditional facts.

This is of course a loss of information, and must be compensated by recomputing the conditional facts during each reduction step:

- We evaluate negative literals \( \text{not } A \) in the conditional facts to true (i.e. delete them from the conditions) if \( A \) does not occur in any rule head. In the alternating fixpoint approach, this corresponds to a recomputation of all derived facts under the assumption that \( \text{not } A \) is true iff \( A \not\in I_{\text{poss}} \).

- We evaluate negative literals \( \text{not } A \) in the conditional facts to false (i.e. delete the conditional fact) if \( A \leftarrow \text{true} \) is a conditional fact. The alternating fixpoint approach must again recompute all consequences under the assumption that \( \text{not } A \) is true iff \( A \not\in I_{\text{sure}} \).
Example 4.3.9:
The following example suggested by Ulrich Zukowski [Zuk96] shows that our method can indeed be substantially faster than the alternating fixpoint approach. Consider the rule
\[
odd(X) \leftarrow succ(Y, X) \land \text{not } odd(Y)
\]
(i.e. “\(X\) is odd iff \(Y = X - 1\) is not odd”) together with the following finite successor relation (for some fixed \(n \in \mathbb{N}\)):
\[
succ := \{(i - 1, i) \mid 1 \leq i \leq n\}.
\]
This is nothing else than the “winning” state example with an especially boring “move” relation (there is always only one possible move or no move at all). So \(odd\) contains the odd numbers less or equal than \(n\).

With the access paths discussed above, it is easy to compute the residual program and thus the well-founded semantics in time \(O(n)\). This is also the complexity we would intuitively expect for computing the odd numbers \(\leq n\).

However, the alternating fixpoint approach needs \(n\) iterations, each costing \(O(n)\). So the total cost is \(O(n^2)\). \(\square\)

Example 4.3.10:
On the other hand, the alternating fixpoint approach needs always polynomial time, whereas the residual program can grow to exponential size. An example is
\[
\begin{align*}
p(0), \\
p(X) & \leftarrow p(Y) \land succ(Y, X) \land \text{not } q(Y), \\
p(X) & \leftarrow p(Y) \land succ(Y, X) \land \text{not } r(Y), \\
q(X) & \leftarrow succ(X, \_.) \land \text{not } q(X), \\
r(X) & \leftarrow succ(X, \_.) \land \text{not } r(X).
\end{align*}
\]
The last two rules are only needed to ensure that \(\text{not } q(Y)\) and \(\text{not } r(Y)\) cannot be deleted by the reduction operator. The relation \(succ\) is again a path of length \(n\):
\[
succ := \{(i - 1, i) \mid 1 \leq i \leq n\}.
\]
In this example, conditional facts of the following form are derivable:
\[
p(i) \leftarrow \text{not } q(0) \land \text{not } q(1) \land \cdots \land \text{not } q(i - 1),
\]
where any subset of the \(q\)-literals can be replaced by the corresponding \(r\)-literals. \(\square\)

We will present in the next section a partial solution to this problem, but it seems that there is a price to pay and there is not (yet) an algorithm which is equally optimal for all programs.

Let us also note that the residual program contains important information which is not provided by the alternating fixpoint method. As we motivated before, the “undefined” truth value is something unusual. So if it happens that one or more atoms become undefined, the user probably wants to see the reason for this. In a system based on the residual program, we can show the user immediately the dependencies between the delayed facts. In the alternating fixpoint method, this information is not available.
A Polynomial Encoding of the Residual Program

As we have seen in Example 4.3.10, the residual program can grow to exponential size. Since there are algorithms for computing the well-founded model in polynomial time (such as the alternating fixpoint method), an exponential algorithm would not be acceptable. Note, however, that the power of the residual program lies in its generality: It can also be used for computing other, more complex semantics, such as the stable models.

Of course, it is possible to develop combined methods, which use the residual program for only those parts of a given program where we can guarantee polynomial complexity at compile-time. Otherwise we would use the alternating fixpoint method, i.e. represent the conditions only by a single bit. We can also choose to give a little more information, e.g. in Example 4.3.10 we could represent at least the predicates of the delayed literals.

Another idea (developed jointly with Ulrich Zukowski and Carl-Alexander Wichert) is to use other data structures for representing the residual program. In Example 4.3.10, we would for instance store the conditional facts in the form

\[ p(i) \leftarrow (\neg q(0) \lor \neg r(0)) \land \cdots \land (\neg q(i-1) \lor \neg r(i-1)). \]

A possibility to formalize this is to use ideas of SLG-resolution [CW93b], which delays not only negative literals, but also positive literals which depend on delayed negative literals. So the main difference is that we now do not replace a positive body literal \( A \) by the conditions \( B_i \) of conditional facts \( A \leftarrow B_i \), \( 1 \leq i \leq n \), about \( A \), but instead delay \( A \). In this way, \( A \) represents the disjunction \( (B_1 \lor \cdots \lor B_n) \) of the conditions of all conditional facts about \( A \).

**Definition 4.3.11 (Immediate Consequences with Delayed Literals):**

The following immediate consequence operator works on sets of ground rules (containing both, positive and negative delayed literals in the body):

\[ T_P(F) := \{ A \theta \leftarrow L_1 \theta \land \cdots \land L_n \theta \mid A \leftarrow L_1 \land \cdots \land L_n \text{ is a rule in } P \]
\[ \text{and } \theta \text{ is a ground substitution for this rule} \]
\[ \text{such that for every } 1 \leq i \leq n, \text{ if } L_i \text{ is positive,} \]
\[ \text{then there is a rule instance about } L_i \theta \text{ in } F \}. \]

Note that we do not delay arbitrary positive literals (then we would simply get the ground instantiation of the program), but only literals which are possibly derivable (because there is already such an “extended conditional fact” for them). Since negative literals are delayed without precondition, the first iteration of \( T_P \) yields exactly the conditional facts contained in \( P \).

By an induction on the number of derivation steps we can easily prove that the derived rule instances contain only constants occurring in the program: Because of the strict range-restriction, every variable is bound in a positive body literal, and this is matched with already derived rule instances. But then it is obvious that the set of derived rule instances can be computed in polynomial time.
4.3. BOTTOM-UP QUERY EVALUATION

It is also easy to see that \( \text{lfp}(\mathcal{T}_P) \) still has the same minimal models as \( P \). So it is equivalent to the original program under every strictly equivalence preserving semantics, including the well-founded and the stable model semantics:

Lemma 4.3.12:
A Herbrand interpretation \( I \) is a minimal model of a program \( P \) iff \( I \) is a minimal model of \( \text{lfp}(\mathcal{T}_P) \).

Proof:
Since \( \text{lfp}(\mathcal{T}_P) \subseteq \text{ground}(P) \), every model \( I \) of \( P \) is also a model of \( \text{lfp}(\mathcal{T}_P) \).

Now let \( I \) be a minimal model of \( \text{lfp}(\mathcal{T}_P) \), but suppose that \( I \) violated a ground instance \( A\theta \leftarrow L_1\theta \land \cdots \land L_n\theta \) of a rule in \( P \). This means especially that every positive literal \( L_i\theta \) is true in \( I \). But since \( I \) is a minimal model of \( \text{lfp}(\mathcal{T}_P) \), only literals can be true which occur at least once in a rule head. But then, by the definition of \( \mathcal{T}_P \), also the critical rule instance would be contained in \( \text{lfp}(\mathcal{T}_P) \), which is impossible, since it is not satisfied in \( I \).

So we now know that every minimal model of \( \text{lfp}(\mathcal{T}_P) \) is also a model of \( P \) and vice versa. But then it must also be minimal, since a smaller minimal model of \( P \) would also be a model of \( \text{lfp}(\mathcal{T}_P) \) (and vice versa).

Next, we again define a reduction operator which evaluates delayed literals if their truth value is obvious. To simplify the definition, we treat the rule body as a set of literals (as in the case of normalized conditional facts). It is possible to include also the deletion of non-minimal rules, but we probably get very few of them as long as we work only with instances of rules explicitly written by the user.

Definition 4.3.13 (Reductions of Delayed Literals):
The following reduction operator evaluates delayed positive and negative literals as far as possible:

\[
\hat{R}(F) := \{ A \leftarrow (B - \text{known}(F)) \ | \ (A \leftarrow B) \in F \ \text{and for every } L \in B: \sim L \notin \text{known}(F) \},
\]

where \( \text{known}(F) \) is the set of positive and negative ground literals with known truth value:

\[
\text{known}(F) := \{ L \in \text{BASE} \ | \ L \text{ is a positive literal and } (L \leftarrow \text{true}) \in F \\
\text{or } L \text{ is a negative literal } \text{not } A, \\
\text{and } F \text{ contains no rule about } A \}.
\]

Of course, we can and should do the reductions which depend on the presence of positive facts \( A \leftarrow \text{true} \) already during the computation of \( \text{lfp}(\mathcal{T}_P) \). Reductions which depend on the absence of rules about \( A \) are possible if we know that such rules also cannot be derived later.

Note that the two reductions of positive literals are both extreme cases of unfolding, where there is either no rule about the unfolded literal, or the unfolded literal is given as a fact. So the reduction \( \hat{R} \) is an equivalence transformation for all semantics allowing unfolding and positive and negative reduction. This includes the well-founded and stable model semantics.
Example 4.3.14:
However, we do not get the polynomial complexity for free. Delaying positive literals can introduce loops, something usually not possible during bottom-up evaluation. For instance, consider the following program: $P$:

\[
\begin{align*}
p. \\
qu & \leftarrow \text{not } p. \\
q & \leftarrow q.
\end{align*}
\]

Here, $\text{lfp}(\tilde{T}_P)$ is the complete program $P$: The rule $q \leftarrow q$ is derived since we already have the delayed rule $q \leftarrow \text{not } p$ about $q$. During the reduction phase, it turns out that $\text{not } p$ is false, so $q \leftarrow \text{not } p$ is deleted. But the above reduction operator does not delete $q \leftarrow q$. \hfill $\square$

So our reductions are still to weak. In order to have a useful encoding of the residual program, we need the following property: If we unfold the positive body literals, we get for every delayed rule $A \leftarrow B$ at least one conditional fact $A \leftarrow B'$. This condition is violated in the above example, because we will never get a conditional fact about $p$.

Of course, we cannot actually do the unfolding, because this could be exponential. But we can mark in the other direction those ground literals for which there will be at least one conditional fact:

**Definition 4.3.15 (Deletion of Unfounded Atoms):**
Let $P$ be a ground program. The following operator computes the possibly true atoms:

\[
\tilde{T}_P(I) := \{A \mid \text{there is a ground rule } A \leftarrow L_1 \land \cdots \land L_n \text{ about } A \text{ in } P \text{ such that } L_i \in I \text{ for every positive literal } L_i\}.
\]

Then the following operator deletes rules depending on unfounded atoms:

\[
\tilde{R}(P) := \{A \leftarrow L_1 \land \cdots \land L_n \in P \mid L_i \in \text{lfp}(\tilde{T}_P) \text{ for every positive literal } L_i\}.
\]

Again, this operation does not change the set of minimal models, so it is an equivalence transformation for a wide range of possible semantics:

**Lemma 4.3.16:**
For every ground program $P$ and Herbrand interpretation $I$: $I$ is a minimal model of $P$ iff $I$ is a minimal model of $\tilde{R}(P)$.

**Proof:**
Since $\tilde{R}(P) \subseteq P$, if $I$ is a model of $P$, it is also a model of $\tilde{R}(P)$.

Next, we claim that every minimal model $I$ of $P$ satisfies $I \subseteq \text{lfp}(\tilde{T}_P)$. Suppose that this were not the case. Then the interpretation $J := I \cap \text{lfp}(\tilde{T}_P)$ would be strictly preferable to $I$, so $J$ could not be a model of $P$. So there must be a ground rule $A \leftarrow L_1 \land \cdots \land L_n$ violated by $J$. This would mean that every $L_i$ is true in $J$, so if $L_i$ is positive, it is contained in $\text{lfp}(\tilde{T}_P)$. Furthermore, every $L_i$ is true in $I$, thus also $A$ is true in $I$. But since $A$ is false in $J$, we have $A \not\in \text{lfp}(\tilde{T}_P)$. By the definition of $\tilde{T}_P$, this is impossible.
Now it is easy to see that a minimal model $I$ of $\bar{R}(P)$ is also a model of $P$. Suppose that $I$ violated a ground rule $A \leftarrow L_1 \land \cdots \land L_n$ in $P$. This would mean that especially every positive literal $L_i$ is satisfied in $I$, and thus, as just shown, $L_i \in \text{lfp}(\bar{T}_P)$. But this is impossible since then $A \leftarrow L_1 \land \cdots \land L_n$ would also be contained in $\bar{R}(P)$. □

**Example 4.3.17:**

After we have applied $\bar{R}$, the above reduction operator $\bar{R}$ can become applicable again, and vice versa:

$$
\begin{align*}
p_1 & \leftarrow \text{not } q_0. \\
q_1 & \leftarrow \text{not } p_1. \\
q_1 & \leftarrow q_1. \\
p_2 & \leftarrow \text{not } q_1. \\
q_2 & \leftarrow \text{not } p_2. \\
q_2 & \leftarrow q_2.
\end{align*}
$$

Here all rules result from the delaying. The reduction operator $\bar{R}$ first evaluates $\text{not } q_0$ to true and then $\text{not } p_1$ to false. Then we need to apply $\bar{R}$ in order to get rid of the rule $q_1 \leftarrow q_1$. After we have done this, the reduction operator $\bar{R}$ can evaluate $\text{not } q_1$ to true and then $\text{not } p_2$ to false. Then $\bar{R}$ allows to delete the rule $q_2 \leftarrow q_2$. □

So we must iterate both reduction operators. Since the total number of literals is reduced in every step, it is clear that that only polynomially many iterations are needed. And of course, each single reduction is also computable in polynomial time.

**Definition 4.3.18 (Delayed Program Rest):**

Let a program $P$ be given and let $F_0 := \text{lfp}(\bar{T}_P)$, $F_i := \bar{R}(\bar{R}(F_{i-1}))$, and $n \in \mathbb{N}$ such that $F_n = F_{n-1}$. Then delayed($P$) := $F_n$ is called the delayed program rest of $P$.

So it follows that the delayed program rest is equivalent to the original program under the well-founded and stable model semantics, and in fact under a large class of semantics which allow the elementary transformations introduced in Section 4.2:

**Theorem 4.3.19 (Equivalence of Delayed Program Rest):**

Let $S$ be any semantics which allows unfolding, deletion of tautologies, positive and negative reduction, deletion of subsumed rules, and normalization of rule bodies and furthermore satisfies $S(\text{ground}(P)) = S(P)$. Then $S(\text{delayed}(P)) = S(P)$.

**Proof:**

By Theorem 4.2.31, a semantics $S$ with the above properties looks only at the minimal models of a given program. By Lemma 4.3.12, $\text{lfp}(\bar{T}_P)$ still has the same minimal models as the original program $P$, and by Lemma 4.3.16, the operator $\bar{R}$ also does not change the minimal models. The operator $\bar{R}$ performs only positive and negative reduction and unfolding, so it also cannot change the semantics of the program. □
Theorem 4.3.20 (Computation of Well-Founded Semantics):
For every program $P$, the well-founded model $I$ of $P$ satisfies only those positive and negative literals which are immediately obvious in $\text{delayed}(P)$, i.e.

$$I = \text{known}(\text{delayed}(P)).$$

Proof:
By Theorem 4.2.49 and Proposition 4.1.26, the well-founded semantics has the properties required in Theorem 4.3.19, so $\text{delayed}(P)$ has the same well-founded model as $P$. Thus, it suffices to prove that $I = \text{known}(\text{delayed}(P))$ is the well-founded model of $\text{delayed}(P)$. We do this by showing that $I$ is a fixpoint of $T_{\text{delayed}(P)}^\text{wf}$. Since every fixpoint must obviously contain at least the literals in $\text{known}(\text{delayed}(P))$, $I$ then is the least fixpoint of $T_{\text{delayed}(P)}^\text{wf}$, i.e. the well-founded model.

First, we have to show that $I = \text{known}(\text{delayed}(P))$ is a minimal model of $\text{delayed}(P)$. Since $\text{delayed}(P)$ is a fixpoint of the reduction operator $R$, the literals in $\text{known}(\text{delayed}(P))$ do not occur in any rule body. So every non-trivial rule body contains only literals which are not satisfied in $I$, thus $I$ is a model of $\text{delayed}(P)$. Obviously, there can be no strictly preferable model, since it would immediately violate an unconditional fact $A \leftarrow \text{true}$ in $\text{delayed}(P)$.

Second, we have to show that the interpretation $J$ defined by $J \models L \iff I \not\models \sim L$, is also a minimal model of $\text{delayed}(P)$. So suppose that $J$ violated a rule $A \leftarrow L_1 \land \cdots \land L_n$ in $\text{delayed}(P)$. This would mean $J \models A$, i.e. $I \models \text{not} A$. But $I$ satisfies only those negative literals which occur in no rule head, so this situation is impossible.

Thus, $J$ is a model of $\text{delayed}(P)$. We finally must check that there is no preferable model. Since $\text{delayed}(P)$ is a fixpoint of $R$, all negative literals which occur in rule bodies are satisfied in $J$: $J \models \text{not} A$ means $I \not\models A$. This holds because $I$ satisfies only the positive literals $A$ explicitly given, and then $\text{not} A$ cannot occur in a rule body in $\text{delayed}(P)$. But then, since $\text{delayed}(P)$ is also a fixpoint of $R(P)$, all literals occurring in rule bodies are derivable. Thus, every interpretation which agrees with $J$ on the truth values of the negative literals must satisfy all literals occurring in rule heads in $\text{delayed}(P)$. But $J$ makes exactly these literals true, so there can be no preferable model. 

In summary, we have given a polynomial time algorithm for computing the well-founded semantics. It is based on the elementary program transformations studied in Section 4.2. If we unfolded all positive body literals in the delayed program rest (and deleted occurring tautologies), we would get the residual program (plus some non-minimal conditional facts). So the delayed program rest can be seen as an efficient encoding of the residual program. It is equivalent to the original program under a wide range of semantics. For instance, the computation of $\text{delayed}(P)$ is also a useful preprocessing step for query evaluation under the stable model semantics.

While the computational complexity of the delayed program rest is decisively lower than that of the residual program, the “intellectual complexity” seems to be higher. One of the strengths of the residual program was its simplicity, and it seems that this has to be partially sacrificed, since we now have to cope with positive loops. But our method still operates on the source program level, and can be understood as a sequence of elementary program transformations.

The method is very similar to SLG-resolution [CW93b], maybe it can be described as a bottom-up version of SLG-resolution. We believe that in the “intellectual com-
plexity hierarchy", it is at least one level below SLG-resolution. Of course, our method is not goal-directed, but this can be regained via a "magic set" transformation, and it is certainly good to separate the issues.

We claim that our method never has to do more work than the alternating fixpoint approach, but a careful analysis of this still has to be done. Of course, there are examples (such as Example 4.3.9), where our method is superior to the alternating fixpoint approach.

Note also that in contrast to the residual program, where we had to encode the conditions in set-valued arguments, which might possibly be inefficient, we now can encode the delayed rule rests by means of new predicates (similar to the node types introduced in Section 3.2). It is certainly an advantage that we know at compile time the structure of the conditions.

We have explained above which data structures are needed to compute the residual program efficiently. Many of these ideas can be reused for computing the delayed program rest. However, a new problem is the elimination of positive loops with the $\bar{R}$-operator. Of course, we would like to avoid recomputing $\text{lfp}(\bar{T}_P)$ from scratch in every iteration. Techniques known for computing induced updates on materialized IDB-relations can be used for this purpose. For instance, whenever the reduction operator $\bar{R}$ deletes a rule instance in the current program, we mark recursively all literals depending on the head of this rule instance as "possibly deleted" from $\text{lfp}(\bar{T}_P)$. Then we have to repeat the computation of $\text{lfp}(\bar{T}_P)$ only for the possibly deleted literals.

The example programs 4.3.14 and 4.3.17 are in fact stratified, and we should certainly use the standard method. In general, because of the problems with loops, we should delay positive literals only if there is the risk that the residual program may have exponential size. If the program is stratified, we can immediately evaluate negative literals (by applying our reductions), so we will delay neither positive nor negative literals. We believe that it is a strength of the proposed framework that the discussed methods can be nicely integrated, and it is possible to select for every predicate the optimal method.
Chapter 5

Reasoning with Disjunctions

The topic of this chapter is query evaluation for positive disjunctive databases, i.e. deductive databases with rules of the form

\[ A_1 \lor \cdots \lor A_n \leftarrow B_1 \land \cdots \land B_m. \]

We mentioned in the introduction that there are quite a lot of applications where we have only incomplete knowledge about the world represented in the database. If this incompleteness has the form that there is only a finite (and not too large) set of possibilities, and we do not want to talk about probabilities, then a representation by means of disjunctions seems to be most natural.

Choosing a sufficiently powerful and natural knowledge representation formalism is very important — at least in the early phases of knowledge base design, where the problem is to understand the application domain by talking with the users and experts. Later it might be necessary to transform the initial specification into a formalism which allows more efficient evaluation.

Other problems are inherently so complex that they cannot be encoded in Datalog, because this allows always polynomial time query evaluation (if we use well-founded negation). For instance, with disjunctions it is easily possible to encode the question whether a graph is four-colorable:

\[
\text{color}(X,1) \lor \text{color}(X,2) \lor \text{color}(X,3) \lor \text{color}(X,4) \leftarrow \\
\text{vertex}(X).
\]

\[
\text{impossible} \leftarrow \\
\text{color}(X_1, Y_1) \land \text{edge}(X_1, X_2) \land \text{color}(X_2, Y_2) \land Y_1 \neq Y_2.
\]

If this program implies \textit{impossible} the graph stored in the relations \textit{vertex} and \textit{edge} cannot be colored with four colors. Note how simple and \textit{logical} it is to specify this problem in Datalog\(^\lor\). Allowing disjunctions gives already quite a lot of the power of general theorem proving, however, we require that the clauses are range-restricted and contain no function symbols.

In contrast to the Horn case, where a general consensus about the basic bottom-up and top-down algorithms has been reached, there are still quite a lot of query evaluation proposed for disjunctive logic programs:
• The Prolog technology theorem prover [Sti88, Sti92].
• SLI-resolution [LMR92].
• Near Horn Prolog [Lov91].
• Model Elimination without Contrapositives [BF94]
• SATChMO [MB88], MGTP [HKF92].
• Hyperresolution [CL73], disjunctive consequence operator [MR90, LMR92].

All claim to be somehow natural generalizations of the standard algorithms for Horn programs. We do not aim here at a comparison or overview of all these algorithms. Our goal in Section 5.1 is to show that for one specific such algorithm, namely hyper-resolution, a substantial part of the implementation techniques developed for Horn clauses can be reused. In [BL92], we showed how hyperresolution can be implemented on top of an NF²-database system, in [Bra92] I gave an implementation using only a standard relational database system. The main contribution in Section 5.1 is an optimization to hyperresolution which makes the resolvable literal unique [Bra94a]. We use this idea in a Horn clause meta-interpreter for disjunctive rules. In order to partially evaluate it, we introduce the notion of disjunction types, which are related to the “node types” introduced in Section 3.2. They also allow to determine an evaluation order for the rules (in a way similar to the predicate dependency graph).

In Section 5.2, we use the reasoning algorithms developed in Section 5.1 for computing the stable models. As explained in Chapter 4, even nondisjunctive logic programs can have multiple stable models. Furthermore, the stable semantics supports case analysis, which is typical in disjunctive reasoning. The applications proposed in the literature [SZ90, SZ95, CMMT95] make heavily use of the disjunctive reasoning capabilities inherent in the stable semantics. So it is only natural to use a disjunctive theorem prover such as the one proposed in Section 5.1 for reasoning under the stable semantics. This section is based on joint work with JÜRGEN DIX [BD95c] and also uses ideas developed together with UDO LIPECK [BL93]. Other algorithms for computing stable models are, e.g., [CW93a, BNNS93, PAA91]. Closest to our approach is probably [CW93a]: They also compute a “residual program”, but this is done top-down, and their processing of the residual program is quite different.
5.1 Extended Bottom-Up Query Evaluation

In this Section, we discuss query evaluation for disjunctive deductive databases. It is based on the theorem-proving algorithm “hyperresolution” [CL73], which seems to be a natural generalization of the standard immediate consequence operator $T_P$.

In [BL92], we have shown that hyperresolution can be implemented with database techniques using NF²-relations.

We will present here important further optimizations, which allow to generalize even more implementation techniques known from standard deductive databases to the disjunctive case. In order to demonstrate the applicability of standard techniques, we will give a useful translation from disjunctive databases into Horn clauses (with lists, but only if really needed).

We consider here only positive databases, i.e. without “negation as failure”. As shown in [BD95a, BD95b, BD95c], it is possible to generalize the material from Chapter 4 to the disjunctive case. Of course, we need a disjunctive reasoning algorithm as the basis, and this is what we will study here. Separating the two problems of nonmonotonic negation and disjunctions makes our approach easier to understand, and we believe that the combination is (for the most part) not difficult.

A reasoning algorithm for disjunctive rules is also needed as the basis of query evaluation algorithms for other knowledge-representation formalisms. For instance, in our method for evaluating prioritized default theories [BL93], the main work is to compute consequences of a positive disjunctive database.

Syntax and Semantics of Disjunctive Rules

Let us start by quickly reviewing the syntax and declarative semantics of disjunctive deductive databases.

**Definition 5.1.1 (Disjunctive Rule):**

A disjunctive rule consists of two lists of positive literals, written

$$A_1 \lor \cdots \lor A_n \leftarrow B_1 \land \cdots \land B_m,$$

where $n \geq 0$ and $m \geq 0$. The left hand side $A_1 \lor \cdots \lor A_n$ is called the head of this rule, and the right hand side $B_1 \land \cdots \land B_m$ is its body.

Note that we allow the head to be empty. This is interpreted as “logically false”, i.e. it may not happen that all body literals $B_i$ are satisfied together. Allowing empty heads does not really increase the expressive power of disjunctive databases, we could as well introduce a special atom $false$ and delete it from all derived disjunctions.

**Definition 5.1.2 (Model of Disjunctive Rules):**

A Herbrand interpretation $I$ is a model of a rule $A_1 \lor \cdots \lor A_n \leftarrow B_1 \land \cdots \land B_m$ iff for all ground substitutions $\theta$ such that $I \models B_i \theta$ for $i = 1, \ldots, m$, there is $j \in \{1, \ldots, n\}$ with $I \models A_j \theta$. 
Range-restriction is defined as before. Since there is not yet a “magic set” transformation (but see [Roy90, Dem91]), we need only the strict version of range restriction:

**Definition 5.1.3 (Range-Restriction for Disjunctive Rules):**
A disjunctive rule \( A_1 \lor \cdots \lor A_n \leftarrow B_1 \land \cdots \land B_m \) is (strictly) range-restricted wrt a binding pattern specification \( \text{BIND} \) iff

1. \( \text{vars}(A_1) \cup \cdots \cup \text{vars}(A_n) \subseteq \text{vars}(B_1) \cup \cdots \cup \text{vars}(B_m) \), and
2. there is a permutation \( \pi : \{1, \ldots, m\} \to \{1, \ldots, m\} \) and there are binding patterns \( \beta_i \in \text{BIND}(\text{pred}(B_i)) \) such that for \( i = 1, \ldots, m \)

\[
\text{input}(B_{\pi(i)}, \beta_{\pi(i)}) \subseteq \text{vars}(B_{\pi(1)}) \cup \cdots \cup \text{vars}(B_{\pi(i-1)}).
\]

Of course, the second condition is empty if we have only finite EDB-predicates which allow the binding pattern \( \mathbf{f} \ldots \mathbf{f} \).

For simplicity, we take here the logic programming view, which considers the given facts in the database as part of the logic program. If there are built-in predicates like \(<\) or \(\text{sum}\), this means that we need also infinite logic programs.

**Definition 5.1.4 (Disjunctive Logic Program):**
A disjunctive (logic) program \( P \) is a set of range-restricted disjunctive rules.

**Definition 5.1.5 (Model of a Disjunctive Logic Program):**
A Herbrand interpretation \( I \) is model of a disjunctive program \( P \) \( (I \models P) \) iff \( I \) is model of every rule in \( P \).

Now the goal of bottom-up query evaluation is to compute all implied disjunctive facts. A disjunctive fact corresponds to a rule with empty body, as in the Horn case. Of course, facts cannot contain variables, this also follows from the range-restriction. We treat facts here as sets of literals, not as lists. This simplifies the definitions and allows a stronger duplicate elimination:

**Definition 5.1.6 (Disjunctive Fact):**
A disjunctive fact is a set \( F = \{A_1, \ldots, A_n\} \) of positive ground literals. It is usually written as \( A_1 \lor \cdots \lor A_n \) (in any order, without duplicates).

**Definition 5.1.7 (Implied Disjunctive Fact):**
A Herbrand interpretation \( I \) is model of a disjunctive fact \( F \) \( (\text{written } I \models F) \) iff there is \( A \in F \) with \( I \models A \).

A disjunctive program \( P \) implies a disjunctive fact \( F \) \( (\text{written } P \models F) \) iff every model \( I \) of \( P \) is also a model of \( F \).

Whenever a disjunctive fact \( F \) is implied, also every superset of \( F \) is implied. So what we need are only the \( \subseteq \)-minimal disjunctive facts which are implied by the given program \( P \).
Definition 5.1.8 (Minimal Implied Disjunctive Facts):
We write $I(P)$ for the set of minimal disjunctive facts logically implied by a disjunctive program $P$:

$$I(P) := \{ F \mid F \text{ is a disjunctive fact with } P \vdash F \text{ and such that } P \not\vdash F' \text{ for all } F' \subset F \}.$$ 

Example 5.1.9:
Suppose we want to find paths through graphs where intermediate nodes may be blocked. If we want to give a definition without knowing already which nodes are blocked, we arrive at the following:

$$\text{path}(X, Y) \leftarrow \text{edge}(X, Y).$$
$$\text{path}(X, Z) \lor \text{blocked}(Y) \leftarrow \text{edge}(X, Y) \land \text{path}(Y, Z).$$

Suppose for simplicity that the $\text{path}$-relation consists of a single straight line, i.e.

$$\text{path} := \{(i - 1, i) \mid 1 \leq i \leq n\}.$$ 

Now if we want to find paths starting in node 0, we get disjunctions of the form

$$\text{path}(0, i) \lor \text{blocked}(1) \lor \cdots \lor \text{blocked}(i - 1).$$

This is very intuitive and it is often interesting to know the nodes through which a path leads. Usually this is implemented by an additional list- or set-valued argument:

$$\text{path}(X, Y, []) \leftarrow \text{edge}(X,Y).$$
$$\text{path}(X, Z, [Y \mid Ys]) \leftarrow \text{edge}(X, Y) \land \text{path}(Y, Z, Ys).$$

In fact, our transformation from disjunctive programs to Horn programs will yield exactly this.

Immediate Consequences of Disjunctive Rules
As explained in Chapter 2, Horn-clauses are usually evaluated by inserting matching facts for the body literals and deriving the fact corresponding to the head literal, e.g.:

$$p(a) \quad \uparrow$$
$$\left( p(X) \leftarrow q_1(X) \land q_2(X, Y). \right) \quad \uparrow$$
$$q_1(a) \quad q_2(a, b) \quad \uparrow$$

Now, if we work with disjunctive facts instead of simple facts, the idea is to split the disjunction into the "active literal" and the "context". The active literal participates in the resolution as before, i.e. it is matched with a body literal. The context is directly passed into the resulting disjunction:
Here, \( p(a) \lor p(b) \lor r(c) \) is obviously a logical consequence of the rule and the two disjunctive facts: We do not know whether the active literal \( q_2(a, b) \) or the context \( r(c) \) is true. But if the context is true, the resulting disjunction is trivially satisfied. So assume that \( r(c) \) is false. Then the active literal \( q_2(a, b) \) must be true, the rule is applicable, and we can derive \( p(a) \lor p(b) \).

The following is the standard immediate consequence operator for disjunctive facts [MR90]:

**Definition 5.1.10 (Immediate Consequences of Disjunctive Rules):**

Given a disjunctive program \( P \), we define the following immediate consequence operator on sets of disjunctive facts:

\[
T_P^\delta(\mathcal{F}) := \{ F \mid \text{there are} \\
\text{a rule } A_1 \lor \cdots \lor A_n \leftarrow B_1 \land \cdots \land B_m \text{ in } P, \\
\text{a ground substitution } \theta \text{ for this rule, and} \\
\text{disjunctive facts } F_1, \ldots, F_m \in \mathcal{F}, \\
\text{such that} \\
B_i\theta \in F_i \text{ for } i = 1, \ldots, m, \text{ and} \\
F = \{A_1\theta, \ldots, A_n\theta\} \cup \bigcup_{i=1}^{m}(F_i - \{B_i\theta\}).
\]

So the extended bottom-up evaluation requires that a disjunctive fact with \( n \) literals is split in the \( n \) possible ways into the active literal and the context. In fact, the literature leaves this point a bit unclear. One might get the feeling from [MR90] that disjunctive facts are lists of literals and resolution is allowed only with the first literal. But this is obviously not intended, since then \( p \) would not be derivable from \( p \leftarrow q \) and \( p \lor q \) (contradicting their Theorem 2). This approach would not even be complete for deriving the empty clause, which is interesting from the theorem proving point of view.

**Example 5.1.11:**

Let the following rules and disjunctive facts be given:

\[
\begin{align*}
\leftarrow p \land q. \\
\leftarrow q \land r. \\
\leftarrow r \land p.
\end{align*}
\]

\[
\begin{align*}
p \lor q. \\
q \lor r. \\
r \lor p.
\end{align*}
\]
This set of formulas is inconsistent, i.e. we should be able to derive the empty clause. But if we treat the disjunctive facts as lists and allow only resolution with the first literal, no new disjunctive facts are derivable. □

But allowing the resolution with any literal in a disjunctive fact can be very inefficient. Even the simplest possible computation on the \textit{blocked}-literals, for instance renaming them by means of a rule

\[ \text{problem}(X) \leftarrow \text{blocked}(X) \]

immediately yields an exponential behaviour, because we can apply this rule to any subset of the literals of

\[ \text{path}(0, n) \lor \text{blocked}(1) \lor \cdots \lor \text{blocked}(n - 1). \]

Disjunctions are so expressive that in general, an exponential behaviour is unavoidable (their greater expressivity is one of the reasons for using them). However, often it is possible to improve the efficiency of bottom-up evaluation for disjunctive rules by means of an important optimization which we are going to present now.

\section*{Making the Resolvable Literal Unique}

In fact, the extended bottom-up evaluation is nothing else than a special case of positive hyperresolution [CL73]. And it is known that positive hyperresolution remains complete if one restricts the active literal to have the minimal predicate with respect to some order (e.g., the lexicographical one) of the predicates [CL73, BL92]. This is a big step in the right direction, but unfortunately disjunctions often consist of many literals with the same predicate, so the active literal is still far from being uniquely determined. But in contrast to the theorem-proving community, we consider only range-restricted rules, so all variables will be bound during a resolution step. Therefore, theoretically there is no difference whether we work with the original program or its ground instantiation. This allows us to extend the order from the predicates to the ground literals, so that the minimality requirement for the active literal uniquely determines it.

\textbf{Definition 5.1.12 (Optimized Immediate Consequence Operator):}

Let $\prec$ be a linear order on the ground atoms. Given a disjunctive program $P$, the optimized immediate consequence operator for sets of disjunctive facts is:

\[
\mathcal{T}_P^{\prec}(\mathcal{F}) := \{ F \mid \text{there are} \}
\]

\[
\text{a rule } A_1 \lor \cdots \lor A_n \leftarrow B_1 \land \cdots \land B_m \text{ in } P, \\
\text{a ground substitution } \theta \text{ for this rule, and} \\
\text{disjunctive facts } F_1, \ldots, F_m \in \mathcal{F}, \\
\text{such that} \\
B_i\theta \text{ is the } \prec\text{-minimal element of } F_i \ (i = 1, \ldots, m), \text{ and} \\
F = \{A_i\theta, \ldots, A_n\theta\} \cup \bigcup_{i=1}^{m} (F_i - \{B_i\theta\}).
\]
The order $<$ on the ground literals can be any order, for instance the lexicographic one. However, for our completeness result we need that atoms which we expect in the answers are $\leftrightarrow$-maximal, so that they do not interfere with the resolution steps (see below). Also, some orders yield more efficient computations than others. It is still an open research problem to automatically determine an “optimal” $<$.  

The operator $\hat{T}_P$ is monotonic and even continuous. So in order to compute its least fixpoint, this operator is iteratively applied until nothing changes (as usual). If the facts and rules contain no function symbols (or built-in predicates), the fixpoint is reached after a finite number of iterations. Of course, we are only interested in minimal derivable disjunctions:

**Definition 5.1.13 (Derivable Disjunctive Facts):**

Then the set of minimal disjunctive facts derivable from a disjunctive program $P$ is

$$D(P) := \{ F \in \text{lfp}(\hat{T}_P) \mid \text{there is no } F' \subset F \text{ with } F' \in \text{lfp}(\hat{T}_P) \}.$$  

In the theorem proving community, one is only interested in the completeness for deriving the empty clause. So we cannot directly use their completeness result. And in fact, there is a problem. In the example where we rename `blocked` into `problem`, exponentially many disjunctions are logical consequences of the given formulas. Of course, we do not want to derive all of these. Unusually, we are only interested in disjunctions consisting entirely of literals with the special predicate `answer`, which is defined by the user and does not appear in rule bodies. If we choose the order $<$ on the ground atoms in such a way that the `answer`-atoms follow all other atoms, then the conditions of the following theorem are satisfied for the subset $A$ of atoms of the form $\text{answer}(c_1, \ldots, c_n)$:

**Theorem 5.1.14 (Soundness and Completeness):**

Let a logic program $P$ and a linear order $<$ on the ground atoms be given. Let $A$ be a set of ground atoms satisfying the following conditions:

- $A$ is upwards closed wrt $<$, i.e. if $A \in A$ and $A < A'$, then $A' \in A$.
- No atom in $A$ matches a rule body, i.e. for every rule $A_1 \lor \cdots \lor A_n \leftarrow B_1 \land \cdots \land B_m$ in $P$ and every ground substitution $\theta$: $\{B_1\theta, \ldots, B_m\theta\} \cap A = \emptyset$.

Let $A^\forall$ be the disjunctive facts consisting only of atoms from $A$. Then $\hat{T}_P^\forall$ is sound and complete for $A^\forall$, i.e. $D(P) \cap A^\forall = I(P) \cap A^\forall$.

**Proof:**

The correctness is easy: Consider a derivation step with $\hat{T}_P^\forall$: A model of the rule

$$A_1 \lor \cdots \lor A_n \leftarrow B_1 \land \cdots \land B_m$$

and the disjunctive facts $F_1, \ldots, F_m$ must satisfy one of the contexts $F_i - \{B_i\theta\}$ or all the $B_i\theta$, and therefore one of the $A_i\theta$. But this means that the resulting disjunction is satisfied.

Now we have to show that if $F$ is a minimal disjunction with $P \vdash F$, then it is derivable by iterated application of $\hat{T}_P^\forall$. We first prove this for finite sets of ground rules only, and later lift the argument to the general case. The proof is by induction on the number $n$ of
ground atoms appearing in $P \cup \{F\}$. The case $n = 0$ is trivial since $F$ can only be the empty disjunction $\square$, and then $F$ must include $\square$.

In the inductive step, let us first assume that $F$ is not $\square$. Then let $A$ be any atom in $F$, and let $F' := F - \{A\}$. Let $P'$ the result of evaluating $A$ as false, i.e. remove $A$ from every rule head, and delete rules containing $A$ in the body. Then we have:

- $P' \cup \{F'\}$ contains (at least) one fewer ground atom than $P \cup \{F\}$,
- $P' \vDash F'$ (a model of $P'$ not satisfying $F'$ can be extended to a model of $P$ not satisfying $F$),
- $F'$ is still $\subseteq$-minimal with this property (if $F'' \subset F'$ would follow from $P'$, then $F'' \lor A$ would follow from $P$).

So by the inductive Hypothesis, $F' \in \mathcal{D}(P')$. Now the same derivation steps can be performed from $P$, with the only difference that the derived facts can contain $A$ in addition. But the conditions on $A$ ensure that $A$ is always contained in the context and never prevents a resolution step. And since $F$ was $\subseteq$-minimal and the correctness of $T^\cup_p$ is already proven, we can conclude that $F$ is derived from $P$, and not $F'$.

The second case in the inductive step is $F = \square$. Let $A$ be the $\prec$-maximal ground atom appearing in $P$. As before, we construct a reduced set $P'$ by interpreting $A$ as false. Since $P$ has no model, it especially has no model in which $A$ is false, so $P' \not\vDash \square$, and the inductive hypothesis yields a derivation of $\square$ from $P'$. Now, because $A$ was the $\prec$-maximal atom, it does not prevent any of the previous resolution steps if we put it back into the rule heads and disjunctive facts. So we are able to derive $\square$ or $A$ from $P$. In the first case, we are done. In the second case, we construct $P''$ by interpreting $A$ as true (i.e. we remove $A$ from the rule bodies and delete rules containing it in the head). The inductive hypothesis now gives us a derivation of $\square$ from $P''$. In order to turn this into a derivation from $P$, we may need $A$ in some rule bodies. But we know already that $A$ is derivable.

This completes the induction for the ground case. In the general case, we can conclude by Herbrand's theorem [CL73] that there is a finite set $P_0$ of ground instances of rules in $P$ such that $P_0 \vDash F$. But because of the range-restriction every variable is bound in each derivation step, so the “lifting” of a proof from the ground instances is trivial here. 

\[\square\]

**Completeness for More Disjunctive Facts**

But what can we do if we need more than only answer-disjunctions, for instance, because we have delayed negative body literals? An easy trick is to double every predicate $p$ which we want to have in the answer and which does not fulfill the above condition, and to add the rule $p'(X_1, \ldots, X_n) \leftarrow p(X_1, \ldots, X_n)$. Now if we choose the order $\prec$ in such a way that all atoms with primed predicates come after all the other atoms, the condition of the above theorem is satisfied for the set $A$ of primed atoms.

Of course, if in the extreme case we do this for every predicate, we cannot gain something with respect to the number of derivable disjunctions — as any sound and complete inference procedure. However, we can reduce the number of ways in which a disjunctive fact is derivable. By priming a predicate, we permanently delay its atom: It will never participate again in a resolution step. For instance, consider again a disjunction of the form $\text{path}(0,i) \lor \text{blocked}(1) \lor \cdots \lor \text{blocked}(i-1)$ and let us assume that $\text{path} \prec \text{blocked}$. So we must work first at the $\text{path}$-literal. It is possible to prime it, but then we can never touch it again, so we must continue with $\text{blocked}(1)$.
Without our optimization, resolution would be totally unstructured and we could work on every literal in every order.

Finally note that we of course not really propose to have explicit priming rules. Lower level data structures would allow a much more efficient implementation. For instance, we would certainly have indexes over the atoms within a disjunctive fact, which can be matched with a body literal. If we had a complete set of priming rules (which is usually not necessary), we can in the beginning resolve with every literal within a disjunctive fact (just as the standard approach). But once we resolve with a literal which is not -minimal, all -less atoms cannot participate in the resolution any more. So the generated disjunctive fact does not get an index entry for them.

A Meta-Interpreter for Disjunctive Rules

Let us now apply this result in a Horn-clause bottom-up metainterpreter for disjunctive rules. This can serve as a basis for generalizing existing implementation techniques for deductive databases to the disjunctive case.

We represent disjunctive facts as \(\leftarrow\)-sorted lists. The given disjunctive facts are stored in \texttt{db\_disfact}. The proper rules are stored as \texttt{rule\_\_m}-facts (depending on their number \(n/m\) of head/body literals). The distinction between rules based on their number of literals seems to be an inherent problem of bottom-up meta-interpreters, in [Bry90b] a built-in predicate \texttt{evaluate} was invented for this purpose. We prefer to use one rule in the meta-interpreter for every occurring \(n\) and \(m\):

\[
\texttt{disfact(Stored) :-} \\
\quad \texttt{db\_disfact(Stored)}. \\
\]

\[
\texttt{disfact(Derived) :-} \\
\quad \texttt{rule\_1\_2(A1, B1, B2),} \quad % A1 \leftarrow B1 \land B2. \\
\quad \texttt{disfact([B1\_C1]),} \\
\quad \texttt{disfact([B2\_C2]),} \\
\quad \texttt{merge(C1, C2, C1C2),} \\
\quad \texttt{merge([A1], C1C2, Derived)}. \\
\]

The built-in predicate \texttt{merge} merges two \(\leftarrow\)-sorted lists and removes duplicate elements. Of course, the next step is to partially evaluate the metainterpreter with respect to the given rules. For instance, we would transform the second rule of example 5.1.9 into:

\[
\texttt{disfact(Derived) :-} \\
\quad \texttt{disfact([edge(X,Y)\_C1]),} \\
\quad \texttt{disfact([path(Y,Z)\_C2]),} \\
\quad \texttt{merge(C1, C2, C1C2),} \\
\quad \texttt{merge([\text{path}(X,Z), \text{blocked}(Y)], C1C2, Derived)}. \\
\]

We assume in this example that all \texttt{path}-facts are \(\leftarrow\) all \texttt{blocked}-facts. Therefore, we know the sequence of the two head literals already at compile-time.
In order to continue the partial evaluation, we can replace \texttt{disfact} by specialized predicates. For instance, \texttt{edge} never has a disjunctive context, so \texttt{C1} is always empty, and we can replace \texttt{disfact([edge(X,Y)|C1])} simply by \texttt{edge(X,Y)}. In order to do this, we need to analyze the structure of the disjunctions really occurring. This is very similar to the “node types” introduced in Section 3.2.

**Disjunction Types**

Let us first look very abstractly at the application of rules. Given a base set \( \hat{F} \) of all (disjunctive) facts, a rule with \( m \) body literals can be seen as \( m \)-ary partial function on \( \hat{F} \) — yielding a derived fact for every \( m \)-tuple of facts (if the given facts match the rule body). Now we can classify the facts into fact types. Formally, a fact type \( \tau \) denotes a set \(|\tau|\) of facts. In the Horn case, facts are usually grouped together according to their predicates.

Now we say that a rule \( \phi \) is applicable to fact types \( \tau_1, \ldots, \tau_m \) yielding \( \tau \) iff there are facts \( F_1, \ldots, F_m \) and \( F \) with \( F_i \in |\tau_i| \) and \( F \in |\tau| \) such that \( F = \phi(F_1, \ldots, F_m) \). In this case, we will say that \( \tau \) depends directly on every \( \tau_i \). Of course, we will represent this information in a graph corresponding to the usual predicate dependency graph.

Now there are of course good and bad classifications of facts into types. The reason why predicates have been so useful in the Horn case is that in this case a rule determines the input and output fact types uniquely. This we cannot get in the disjunctive case. However, the “fan-out” of a rule application should still be small, i.e. given fact types for the body literals, there should be only very few (ideally one) resulting fact types. So in the disjunctive case it seems necessary to look also at the disjunctive context of the facts matched with body literals. For instance, if we have a rule \( z(X) \leftarrow a(X) \) then every possible predicate can be in the context of the body literal and becomes the active literal in the result (assuming an alphabetical `←`). So what results from this rule could possibly be input to every other rule. Therefore, all rules yielding \( a(X) \) together with all they depend on would immediately be classified as recursive. So we need to have as much as possible information about the disjunctive facts resulting from a rule application. This is also important for a magic sets transformation where we want to restrict rule applications to those yielding “useful” facts.

Of course, we always have to find a compromise between the complexity of the analysis and the usefulness of the information we gain from it. The minimal information about the disjunctive fact types we need is which predicates they contain. We distinguish here only between zero, one, and an arbitrary number of occurrences of a predicate:

**Definition 5.1.15 (Disjunction Type):**

- A disjunction type \( \tau \) is a mapping of the set \( \text{PREDS} \) of predicates into \( \{0, 1, +\} \).
  We write disjunction types as a “disjunction” of the predicates having count 1 or+, e.g. \( p \lor q^+ \).
- The set of facts denoted by a disjunction type is defined as follows: \( F \in |\tau| \) iff for all \( p \in \text{PREDS} \):
- If $p$ does not occur in $F$, then $\tau(p) = 0$.
- If $p$ occurs exactly once in $F$, then $\tau(p) = 1$ or $\tau(p) = +$.
- If $p$ occurs more than once in $F$, then $\tau(p) = +$.

Now we want to compute the occurring disjunction types and the dependencies between them. In the Horn case it is usually assumed that facts about all predicates can be derived (as far as we can determine by looking only at the rules). This is not true in the disjunctive case — a computation with all of the above disjunction types is certainly not feasible (there are $3^n$ disjunction types with $n$ predicates). What we do instead is that we start with the disjunction types stored in the database, and compute the types of derivable facts bottom-up by a usual fixpoint construction. We simply use a version of the $\hat{T}_\approx$-operator which ignores the arguments and looks only at the occurring predicates.

The types of facts stored in the database have to be declared by the user, so that we do not have to look actually into the database. This generalizes the usual declaration of EDB-predicates in the Horn case. It is an important database design decision and seems also necessary for choosing efficient storage structures.

**Example 5.1.16:**

Let us assume that the rules of example 5.1.9 are given, together with a rule generated from the query:

\begin{align*}
\text{path}(X, Y) & \leftarrow \text{edge}(X, Y). \quad (1) \\
\text{path}(X, Z) \lor \text{blocked}(Y) & \leftarrow \text{edge}(X, Y) \land \text{path}(Y, Z). \quad (2) \\
\text{answer}(X) & \leftarrow \text{path}(0, X). \quad (3)
\end{align*}

We also assume that $\prec$ is based on the following order among the predicates:

$$
\text{edge} \prec \text{path} \prec \text{blocked} \prec \text{answer}.
$$

In this case theorem 5.1.14 guarantees that $\hat{T}_\approx$ is complete for deriving disjunctions consisting of $\text{answer}$ and $\text{blocked}$-literals.

In the database, only $\text{edge}$-tuples are stored (without disjunctive context). So we first can only apply rule (1) and get the disjunction type $\text{path}$. Now we can apply rule (2) to get the disjunction type $\text{path} \lor \text{blocked}$, and another application of rule (2) yields $\text{path} \lor \text{blocked}^+$. Rule (3) allows to transform $\text{path}$ into $\text{answer}$ in each of these disjunction types. So we get the “disjunction type dependency graph” depicted in Figure 5.1. Of course, we remove subsumed types, i.e. disjunction types $\tau_1$ such that there is $\tau_2$ with $|\tau_1| \subset |\tau_2|$. In this case, $\text{path} \lor \text{blocked}$ and $\text{answer} \lor \text{blocked}$ are superfluous. If we would introduce disjunction types like $\text{path} \lor \text{blocked}^*$, which allow also 0 occurrences of $\text{blocked}$-literals, we could compress the graph even further. However, it depends on the implementation of rule evaluation, whether this is really useful. We anyway have to iterate the recursive rule, and it might be advantageous to know that in the first iteration there is no $\text{blocked}$-literal.

If it turns out that too many disjunction types are generated, it is possible to merge some by introducing $\ast$-counts. The extreme case would be a single “disjunction type” $p_1^* \ldots p_n^*$ corresponding to the disfact-predicate in our meta-interpreter.
Further note that as usual in the Horn case, we reduce the dependency graph to the nodes on which the goal type depends. For instance, if we are interested only in pure answer-facts (without a blocked-context), we would eliminate the upper half of the graph and never use the recursive rule.

**Example 5.1.17:**
Let us continue example 5.1.16 and specialize our meta-interpreter of section 5.1 with respect to the occurring disjunction types. We assume that we performed the compression of the dependency graph and eliminated the subsumed type path ∨ blocked.

\[
\begin{align*}
path(X,Y) & :- \tag{1} 
edge(X,Y).
p\_o_r\_b_l\_o\_k\_d\_P(X, Z, \text{[blocked}(Y)]) & :- \tag{2a}
edge(X,Y),
path(Y,Z).
p\_o_r\_b_l\_o\_k\_d\_P(X, Y, \text{Blocked}) & :- \tag{2b}
edge(X,Y),
p\_o_r\_b_l\_o\_k\_d\_P(Y, Z, \text{Blocked1}),
merge([\text{blocked}(Y)], \text{Blocked1}, \text{Blocked}).
\end{align*}
\]

\[
\begin{align*}
\text{answer}(X) & :- \tag{3a}
path(0,X).
\text{answer\_or\_blockedP}(X, \text{B}) & :- \tag{3b}
p\_o_r\_b_l\_o\_k\_d\_P(0, X, \text{B}).
\end{align*}
\]

Since blocked has only a single argument, we can simplify \text{[blocked}(Y)] to \text{[Y]}. □
5.2 Application: Computation of Stable Models

In Chapter 4, we have introduced the stable model semantics, and explained that it is often used to encode disjunctive information in standard logic programs. For instance, the program

\[
\begin{align*}
p & \leftarrow \text{not } q. \\
q & \leftarrow \text{not } p. \\
r & \leftarrow p. \\
r & \leftarrow q.
\end{align*}
\]

has two stable models. The disjunction \( p \lor q \) holds in both models, and the stable model semantics allows a case analysis to conclude \( r \). In contrast, the well-founded model simply makes all literals undefined.

So in order to compute the stable model semantics, we need to reason with disjunctions, even though we treat only non-disjunctive programs here (the generalization to the disjunctive case is simple, see [BD95c]). Therefore, we had to delay this topic until this point. The computation of the stable semantics is also a nice application of the hyperresolution introduced in Section 5.1.

Of course, our algorithm is based on the residual program introduced theoretically in Section 4.2 and practically in Section 4.3. Let us repeat the two important properties of the residual program:

- It is equivalent to the original program for a wide range of semantics, namely all semantics allowing the elementary transformations of Theorem 4.2.36. By Theorem 4.2.46, this includes the stable model semantics.
- It is much simpler than the original program, because it is ground, it contains no positive body literals, and usually only very few proper rules.

The truth value of most literals is already decided by the residual program, because they either are given as an unconditional fact or do not appear at all in it. We now have to treat the few difficult literals, about which conditions remained after the reductions. In this section, we assume again that the given program \( P \) is finite. This is necessary, because CLARK’s completion is defined only for finite programs (it would otherwise yield “infinite disjunctions”).

The results of this section are joint work with JÜRGEN DIX and have been published in [BD95c]. In [BL93], UDO LIPECK and I used a similar syntactical completion to compute the consequences of prioritized default theories. The relation between circumscription and CLARK’s completion was investigated in [Rei82, Lif85].

A Syntactical Characterization of Stable Models

In order to compute the stable models of the residual program, we make use of a result of FAGES and BEN-ELIYAHU [AB94] that if there are no positive loops, the stable models coincide with the supported models. This is certainly the case for residual programs because they have no positive body literals at all. We have given a direct proof of this in Lemma 4.2.47.
So it suffices to compute supported models. But for the supported models, a nice syntactic characterization is known for long time, namely CLARK’s completion \( \text{comp}(P) \) [Cla78] (for the equivalence to supported models see [BH86]). The idea behind CLARK’s completion is to turn “\( \leftarrow \)” into “\( \rightarrow \)”. This corresponds to the idea of supported models that for every true ground literal \( A \) there must be a reason in form of a rule instance which allows to derive \( A \). For instance, in the above example the residual program is:

\[
\begin{align*}
  p & \leftarrow \text{not } q. \\
  q & \leftarrow \text{not } p. \\
  r & \leftarrow \text{not } q. \\
  r & \leftarrow \text{not } p.
\end{align*}
\]

CLARK’s completion of the residual program is:

\[
\begin{align*}
  p & \leftarrow \neg q. \\
  q & \leftarrow \neg p. \\
  r & \leftarrow \neg q \lor \neg p.
\end{align*}
\]

Note that we now use the logical negation \( \neg \), since the completion is a set of first-order formulas: After all, its purpose is to assign a specific meaning to the “\( \text{not} \)”-operator.

In the general case, the definition of CLARK’s completion is a little complex, but it is much simpler for residual programs. We also prefer to use not the “\( \leftarrow \)”-notation, but a logically equivalent form, which can directly be used for hyperresolution:

**Definition 5.2.1 (Completion of a Residual Program):**

Let \( P \) be a finite ground program without positive body literals. Then we define

\[
\begin{align*}
  \text{comp}^+(P) := & \{ A \lor B_1 \lor \cdots \lor B_n \mid A \leftarrow \text{not } B_1 \land \cdots \land \text{not } B_n \in P \}. \\
  \text{comp}^{-}(P) := & \{ \leftarrow A \land B_{1,j_1} \land \cdots \land B_{m,j_m} \mid A \text{ is a ground atom, } \\
  & A \leftarrow \text{not } B_{i,1} \land \cdots \land \text{not } B_{i,n_i} , \\
  & i = 1, \ldots, m, \text{ are all rules about } A \text{ in } P, \\
  & \text{every } n_i > 0 \text{ and } 1 \leq j_i \leq n_i \}. \\
  \text{comp}(P) := & \text{comp}^+(P) \cup \text{comp}^{-}(P).
\end{align*}
\]

In standard \( \leftarrow \)-notation, CLARK’s completion of the atom \( A \) with conditional facts \( A \leftarrow \text{not } B_{i,1} \land \cdots \land \text{not } B_{i,n_i}, 1 \leq i \leq m \), would be:

\[
A \iff (\neg B_{1,1} \land \cdots \land \neg B_{1,n_1}) \lor (\neg B_{2,1} \land \cdots \land \neg B_{2,n_2}) \lor \cdots \lor (\neg B_{m,1} \land \cdots \land \neg B_{m,n_m})
\]

The disjunctive facts in \( \text{comp}^+(P) \) correspond to the direction \( \leftarrow \), and the headless rules \( \leftarrow A \land B_{j_1} \land \cdots \land B_{j_m} \) in \( \text{comp}^{-}(P) \) are the clausal (“multiplied out”) form the direction \( \rightarrow \). We could also have written them as purely negative disjunctions \( \neg A \lor \neg B_{j_1} \lor \cdots \lor \neg B_{j_m} \), but in the above way \( \text{comp}(P) \) is a disjunctive logic program,
and they will indeed be used as rules for hyperresolution. Of course, the clausal form may be much larger than CLARK’s completion in \( \iff \) -notation. It might be better to work directly with the original form, if a theorem proving allows this (hyperresolution does not).

As we mentioned before, it is known that CLARK’s completion corresponds to supported models [BH86]. In our special case, this is easy to see:

**Lemma 5.2.2:**

Let \( P \) be a finite ground program without positive body literals. A normal interpretation \( I \) is a supported model of \( P \) iff \( I \models \text{comp}(P) \).

**Proof:**

- Let \( I \) be a supported model of \( P \). Of course, it does satisfy the positive disjunctions in \( \text{comp}^+(P) \), since these are logically equivalent to the rules in \( P \) if we interpret “not” as “\( \neg \)”.

  A supported model is a normal interpretation, so this is indeed possible.

  Now suppose that \( I \) does not satisfy a rule \( \leftarrow A \land B_{i_1} \land \cdots \land B_{i_m} \) in \( \text{comp}^+(P) \).

  Then \( I \models A \), but \( A \) is not supported, since \( I \models B_{i_j} \) makes \( A \leftarrow \text{not } B_{i_1} \land \cdots \land \text{not } B_{i_{n_i}} \) not applicable as a support for \( A \).

- Let \( I \models \text{comp}(P) \), but suppose that \( I \) were not a supported model of \( P \). Since \( I \models \text{comp}^+(P) \) and \( I \) is normal, \( I \) obviously is a model of \( P \). So it is only possible that the support for some ground atom \( A \) is missing in \( I \). Let \( A \leftarrow \text{not } B_{i_1} \land \cdots \land \text{not } B_{i_{n_i}} \), \( 1 \leq i \leq m \), be all rules about \( A \) in \( P \). Since none of these supports \( A \), in every rule a body literal \( \text{not } B_{i_{j_i}} \) must be false in \( I \). But then \( I \) does not satisfy \( \leftarrow A \lor B_{1,j_1} \land \cdots \land B_{m,i_m} \) (because the body is satisfied, but the empty head can never be satisfied).

In summary, if we take CLARK’s completion of the residual program, we get a syntactical characterization of the stable models:

**Theorem 5.2.3 (Completion Characterizes Stable Models):**

A normal interpretation \( I \) is a stable model of \( P \) iff \( I \models \text{comp}(\text{res}(P)) \).

**Proof:**

By Theorem 4.2.36 and Theorem 4.2.46, \( I \) is a stable model of \( P \) iff it is a stable model of \( \text{res}(P) \). By Lemma 4.2.47, \( I \) is a stable model of \( \text{res}(P) \) iff it is a supported model of \( \text{res}(P) \). By Lemma 5.2.2, \( I \) is a supported model of \( \text{res}(P) \) iff \( I \models \text{comp}(\text{res}(P)) \).}

**Skeptical Consequences of the Stable Semantics**

So an answer \( \theta \) to a query \( Q \) is (skeptically) correct wrt the stable model semantics iff \( \text{comp}(\text{res}(P)) \models Q\theta \). Any theorem prover can be used to check that (or generate answers during the proof of \( Q \)). It is also possible to use standard model generating algorithms on \( \text{comp}(\text{res}(P)) \) to construct the stable models of \( P \). Of course, here we will use the theory developed in Section 5.1. Note also that \( \text{comp}(\text{res}(P)) \) has a very special structure: It consists only of purely positive and purely negative clauses.
In order to continue our computation bottom-up, we compute all minimal positive disjunctions derivable from \( \text{comp}(\text{res}(P)) \). It is possible that the empty disjunction is computed, which is interpreted as logically false. This happens if the completion is inconsistent, because there is no stable model, e.g. for the program \( p \leftarrow \text{not} \ p \).

**Definition 5.2.4 (Positive Consequences of Completion):**

For a finite ground program \( P \) without positive body literals let \( \text{comp}^+(P) \) be the set of disjunctive facts \( F \) with

- \( \text{comp}(P) \vdash F \), and
- \( \text{comp}(P) \not\vdash F' \) for every \( F' \subseteq F \).

Let us illustrate this with an example. Consider the following residual program \( P \) and its completion \( \text{comp}(P) \) written as facts and rules:

\[
\begin{align*}
P & : p \lor q. \\
r & \leftarrow \text{not} \ p. & \quad \text{comp}^+(P) & : p \lor q. & \quad \text{comp}^-(P) & : \leftarrow p \land q. \\
r & \leftarrow \text{not} q. & \quad r \lor p. & \quad \leftarrow q \land p. \\
s & \leftarrow \text{not} r. & \quad s \lor r. & \quad \leftarrow s \land r.
\end{align*}
\]

The second and the third rule are of course superfluous. We can derive \( r \) (with the first rule applied to the second and third disjunctive fact), and after deleting non-minimal disjunctions we get \( \{p \lor q, r\} \). This directly describes the two stable models \( I_1 := \{p, r\} \) and \( I_2 := \{q, r\} \). In general, the minimal models of \( \text{comp}^+(\text{res}(P)) \) are the stable models of \( P \). Since the minimal models can often easily be “read off” a set of positive disjunctions, it makes sense to present \( \text{comp}^+(\text{res}(P)) \) to the user.

**Lemma 5.2.5:**

Let \( P \) be a finite ground program without positive body literals. An interpretation \( I \) is a model of \( \text{comp}(P) \) iff it is a minimal model of \( \text{comp}^+(P) \).

**Proof:**

- Let \( I \) be a minimal model of \( \text{comp}^+(P) \), and let \( I_1, \ldots, I_m \) be all models of \( \text{comp}(P) \). We have to show that \( P \) is among the \( I_i \). Suppose it were not. Since \( I \) is a minimal model of \( \text{comp}^+(P) \), and the \( I_i \) are also models of \( \text{comp}^+(P) \) (since \( \text{comp}(P) \vdash \text{comp}^+(P) \)), there must be ground atoms \( A_i \) with \( I_i \models A_i \) and \( I \not\models A_i \) (if for some \( I_i \) there were no such atom, we would have \( I_i \preceq I \), and together with \( I_i \not= I \) this would imply \( I_i \prec I \)). Now consider the disjunction \( A_1 \lor \cdots \lor A_m \). It (or a subdisjunction of it) is contained in \( \text{comp}^+(P) \), because all the models of \( \text{comp}(P) \) satisfy it, i.e. \( \text{comp}(P) \vdash A_1 \lor \cdots \lor A_m \). However, \( I \) does not satisfy it, which is a contradiction.

- Now let \( I \) be a model of \( \text{comp}(P) \). Since \( \text{comp}(P) \vdash \text{comp}^+(P) \), \( I \) is also a model of \( \text{comp}^+(P) \). Now suppose that there were a smaller model \( I_1 \) of \( \text{comp}^+(P) \), and let \( I_0 \) be a minimal model of \( \text{comp}^+(P) \) with \( I_0 \preceq I_1 \), i.e. \( I_0 \prec I \). By the already proven other direction, we get \( I_0 \models \text{comp}(P) \). Let \( A \) be a ground atom with \( I \models A \) and \( I_0 \not\models A \). By Lemma 5.2.2, \( I \) is a supported model of \( P \), so let \( A \leftarrow \text{not} \ B_1 \land \cdots \land \text{not} \ B_m \) be the supporting rule for \( A \). Since \( I \models \text{not} \ B_1 \land \cdots \land \text{not} \ B_m \), and \( I_0 \prec I \), it follows that \( I_0 \models \text{not} \ B_1 \land \cdots \land \text{not} \ B_m \). Finally, we know \( I_0 \not\models A \). But this means \( I_0 \not\models A \leftarrow \text{not} \ B_1 \land \cdots \land \text{not} \ B_m \), which is a contradiction. \( \Box \)
Theorem 5.2.6 (Meaning of Implied Positive Disjunctions):

\( I \) is a stable model of \( P \) iff \( I \) is a minimal model of \( \text{comp}^*(\text{res}(P)) \).

Proof:

This follows immediately from Theorem 5.2.3 and Lemma 5.2.5.

Obviously, a positive disjunction is skeptically implied by the stable model semantics iff it or a subdisjunction of it is contained in \( \text{comp}^*(\text{res}(P)) \). A negative ground atom is implied iff it does not occur in \( \text{comp}^*(\text{res}(P)) \). Arbitrary negative disjunctions require a little more computation, but still can be answered efficiently from \( \text{comp}^*(\text{res}(P)) \):

Theorem 5.2.7 (Computation of Negative Disjunctions):

A negative disjunction \( \lnot A_1 \lor \cdots \lor \lnot A_n \) is skeptically implied by STABLE iff there are not \( F_1, \ldots, F_n \in \text{comp}^*(\text{res}(P)) \) with \( A_i \in F_i \) (\( i = 1, \ldots, n \)), and such that neither \( \bigcup_{i=1}^n (F_i - \{A_i\}) \) nor a subdisjunction of it is contained in \( \text{comp}^*(\text{res}(P)) \).

Proof:

- Let \( \lnot A_1 \lor \cdots \lor \lnot A_n \) be skeptically implied by STABLE, i.e. true in every minimal model of \( \text{comp}^*(\text{res}(P)) \). Let \( F_1, \ldots, F_n \in \text{comp}^*(\text{res}(P)) \) with \( A_i \in F_i \) (\( i = 1, \ldots, n \)). Since one of the \( A_i \) is false in every minimal model of \( \text{comp}^*(\text{res}(P)) \), the corresponding rest disjunction \( F_i - \{A_i\} \) must be true in that minimal model, so \( \bigcup_{i=1}^n (F_i - \{A_i\}) \) is true in every minimal model of \( \text{comp}^*(\text{res}(P)) \), and since it is a positive disjunction, it is also true in every model. By the transitivity of \( \vdash \), all positive disjunctions which follow from \( \text{comp}^*(\text{res}(P)) \), are already represented by a subdisjunction in \( \text{comp}^*(\text{res}(P)) \).

- Now let \( \lnot A_1 \lor \cdots \lor \lnot A_n \) be not skeptically implied by STABLE, so there is a stable model \( I \) of \( \text{res}(P) \) with \( I \models A_i \) (\( i = 1, \ldots, n \)). By Lemma 5.2.5, \( I \) is a minimal model of \( \text{comp}^*(\text{res}(P)) \), therefore \( I_i := I - \{A_i\} \) cannot be a model of it, i.e. there must be a disjunction \( F_i \) with \( I_i \not\models F_i \). Since \( I \) satisfies it, \( A_i \in F_i \) follows. However, \( \bigcup_{i=1}^n (F_i - \{A_i\}) \) is false in \( I \), so neither it nor a subdisjunction of it can be contained in \( \text{comp}^*(\text{res}(P)) \).

It is interesting to note that also the dual approach is possible: If we compute all implied negative disjunctions, the maximal models of the result are the stable models of the original program.

A Model-Generation Algorithm

Let us now consider the computation of the stable models themselves. In the special case that there is only a single stable model, \( \text{comp}^*(\text{res}(P)) \) is the set of atoms true in it, so no further computation is needed. If there are multiple models, we construct them by a case analysis. Note that any atom contained in a proper disjunction of \( \text{comp}^*(\text{res}(P)) \) can really be true or false (of course, the decision on different atoms is not independent). So we also need no further computation if we are only interested in credulous truth of ground literals. In the general case, we choose a truth value for a ground atom, and then compute the implied positive disjunctions which
follow from this new information. We repeat this process until no proper disjunctions remain:

**Input:** Datalog\textsuperscript{neg}-Program $P$;

**Output:** Stable models of $P$;

$\hat{P} := \text{res}(P)$;

$\text{model}_\text{generation}(\text{comp}^+(\hat{P}), \text{comp}^-(\hat{P}))$;

**procedure** $\text{model}_\text{generation}(P^+, P^-)$:

$P^* := \mathcal{D}(P^+ \cup P^-)$;

if $P^*$ contains no proper disjunction then

print $P^*$;

else

let $A$ be contained in a proper disjunction;

$\text{model}_\text{generation}(P^+ \cup \{A\}, P^-)$;

$\text{model}_\text{generation}(P^+, P^- \cup \{\neg A\})$;

Note that by the completeness of hyperresolution, we can be sure that if we choose $A$ to be true or false, there is really a stable model in which $A$ has this truth value. So our algorithm never runs into “dead ends”. This is an important difference to the algorithm used in [CW93a] for propagating guessed truth values of atoms. For instance, the following residual program has only a single stable model (namely $\{p, \neg q, r\}$), but it is reduced in the sense of [CW93a], so their algorithm has to guess a truth value of some atom:

\begin{align*}
p & \leftarrow \neg p. \\
p & \leftarrow \neg q. \\
q & \leftarrow \neg r. \\
r & \leftarrow \neg q.
\end{align*}

Of course, this example is quite pathological, and their reduction algorithm works well in the more usual simpler cases.

Of course, in our algorithm it is essential to compute $P^* := \mathcal{D}(P^+ \cup P^-)$ incrementally, i.e. only compute derivations based on the new fact $A$ or rule $\neg A$. In [BD95c], we have shown how hyperresolution can be implemented by “pushing the facts through the rules” (whereas in Section 5.1, the rules had the control). This gives immediately a seminaive behaviour.
Chapter 6

Conclusions

The two most important contributions of this thesis are the new “SLDMagic” transformation from Section 3.2 and the framework for negation from Sections 4.2 and 4.3.

The idea to implement SLD-resolution on top of a deductive database by means of a new program transformation might well become a foundation of the next generation of deductive database systems. It is simple and elegant and has given us a whole bunch of optimization techniques within a common framework. This idea has also led to a much better understanding of the differences between SLD-resolution and the “magic set” method.

Our framework for negation is a nice combination of theory and practice: We have studied properties of abstract semantics, and have given characterizations of the most important concrete semantics. Furthermore, our properties have the form of elementary program transformations, and together they allow to compute normalforms which led to algorithms for bottom-up query evaluation. Especially the algorithm for computing the well-founded semantics discussed at the end of Section 4.3 seems to be very relevant for actual systems.

One question which I am aware of is the wish for an integrated approach: We of course would like a bottom-up query evaluation algorithm, which is goal directed and supports nonmonotonic negation as well as disjunctions. The combination of negation and disjunctions is possible: In [BD95a, BD95b, BD95c] we have documented an approach to semantics of disjunctive logic programs which is a generalization of the work presented in Chapter 4 and Section 5.2 (with some minor differences, since I of course have tried to strengthen and simplify the results in the special case treated here). We also believe that the combination of goal-directedness and our approach to negation semantics (in the non-disjunctive case) should not be difficult, but we have not yet worked out the details. However, goal-directed bottom-up evaluation for disjunctive rules seems to be quite difficult. The approaches which come closest are a goal-directed version of hyperresolution proposed in [Dem91] and a magic set transformation for the SATCHMO-prover [MB88, SSI95]. However, in both cases it seems not yet clear how far standard implementation techniques from Horn deductive databases can be used. The translation to Horn clauses proposed in Section 5.1 solves this problem, and we could in fact use the known magic set transformation on the result, but the use of lists limits the amount of goal-directedness which we get. The
transformation method proposed in Section 3.2 already yields better results, but more research is needed here.

Another topic which deserves further work is the relation of our treatment of negations to default reasoning (see, e.g., [BL93]) and abductive reasoning. There seem to be many similarities, but a common framework is still missing.

In the end, we of course want systems which do some useful work for human beings — in a reliable, robust, easy-to-use, and sufficiently efficient way. This can only be based on a solid, clear, and simple theory. We have contributed to a better understanding of deductive databases, and we believe that they in turn have the potential to support reliable and understandable applications better than current technology. So we must look now at the applications and maybe modify the language Datalog in order to support them better, and we must also look at the systems, and improve their efficiency and robustness. However, this will also lead to new challenges for theoretic research. I certainly do not believe that we have already reached the state where it suffices to add some “syntactical sugar” to Datalog and to invest more manpower in implementation projects.
Bibliography


[Fur92] K. Furukawa: Logic programming as the integrator of the fifth generation computer systems project. Communications of the ACM (CACM) 35:3 (1992), 82–92. Some software developed during this project is available from “http://www.icot.or.jp”.


Index

abstract semantics, 122
active domain, 49
Aditi, 33, 51, 63, 67
adorned program, 75
allowed, 30
allowed interpretation, 37
alphabet, 29
anonymous variable, 39
answer, 51
answer equivalent, 71
application
  of a substitution, 42
argument
  bound, 29, 36
  free, 29, 36
  of a literal, 39
arity, 34
atomic formula, 39

binding pattern, 29, 36
  more general, 36
binding pattern specification, 36
  extended, 107
body literal, 40
body of a rule, 40
bottom-up evaluation, 45, 46, 52
  with conditional facts, 132, 149
bound argument, 29, 36
built-in predicate, 36
built-in predicates, 33
Clark’s completion, 111, 179
closed world assumption, 30
complementary literal, 106
completion
  of a residual program, 179
conditional fact, 129
  normalized, 148

confluence, 131
conjunction, 40
consequence, 51
consistency-preserving semantics, 122
constants, 30
constraint logic programming, 47
CORAL, 30–33, 67, 85, 86, 100, 104, 151
correct answer, 51
CWA, 30
database
deductively, 41
Datalog
  with Negations, 106
Datalog-rule, 40
datatype elements, 30
DCA, 30
deductive database, 41
definite semantics, 122
delayed program rest, 162
deletion
  of contradictions, 127
  of subsumed rules, 128
  of tautologies, 124
  of unfounded atoms, 161
dependency graph
  extended, 108
depends on, 53
direct consequence operator, 45
disjunctive fact, 168
disjunctive logic program, 168
disjunctive rule, 167
domain, 30
  active, 49
domain closure, 30

EDB-predicates, 33
elimination of subsumed rules, 132
 equivalence-preserving semantics, 123
    strictly, 123
 equivalent programs, 122
 extended binding pattern specification, 107
 extension of a predicate, 35

 fact, 34
    conditional, 129
    disjunctive, 168
 fixpoint, 46
    least, 46
 free argument, 29, 36

 Glue-Nail, 55, 63
 goal, 40
 ground instantiation, 50
 ground substitution, 43

 head of a rule, 40
 Herbrand-base, 35
 Herbrand-interpretation, 35

 IDB-predicate, 38
 immediate consequences
    of disjunctive rules, 170
    optimized, 171
    restricted, 151
    with delayed literals, 159
 information order, 115
 interpretation, 35
    allowed, 37
    normal, 109
    semi-normal, 109
 irreducible, 130

 key, 37

 lemma in SLD-tree, 72
 literal, 39
    complementary, 106
    magic, 77
    negative, 106
    non-magic, 77
    positive, 106
 logic program, 41

 disjunctive, 168
 LOLA, 33, 114

 magic literals, 77
 magic set transformation, 78
 matching magic fact, 79
 minimal model, 44
    for Datalog\textsuperscript{neg}, 110
 model, 43
    minimal, 44
    of a disjunctive program, 168
    stable, 112
    supported, 112
    well-founded, 117
 more general binding pattern, 36

 negation predicate, 106
 negation semantics, 122
 negative literal, 106
 node type, 91
 non-magic literals, 77
 normal interpretation, 109
 normal semantics, 122
 normalform, 130
 normalization
    of rule bodies, 128
    normalized conditional fact, 148
 null-value, 31

 objects, 30
 order
    between sets of cond. facts, 132

 positive consequences
    of completion, 181
 positive literal, 106
 predicate, 28, 34
    built-in, 36
    dependencies, 53
    extension, 35
    IDB-, 38
    negation, 106
    of a literal, 39
    recursive, 53
 predicate dependency graph, 53
    extended, 108
reduced, 53
predicate evaluation sequence, 54
predicates
   built-in, 33
   EDB-, 33
preferable interpretation, 44
   wrt IDB-Predicates, 44
preference relation
   information order, 115
   with negation, 110
program, 41
   adorned, 75
   range-restricted, 49
   rectified, 73
   residual, 137
   stratified, 108
   strictly range-restricted, 49
   tail-recursive, 87
program transformation, 124
   deletion of contradictions, 127
   deletion of subsumed rules, 128
   deletion of tautologies, 124
   negative reduction, 127
   normalization of rule bodies, 128
   positive reduction, 127
   unfolding, 125
query, 51
   range-restricted, 30
   range-restricted program, 49
   range-restriction, 48
   rectified program, 73
   recursive clique, 53
   recursive predicate, 53
   recursive rule, 54
   reduced predicate dep. graph, 53
reduction
   negative, 127
   positive, 127
reduction operator, 136, 149, 160
   weak, 151
relation, 27
relational model, 27
residual program, 137, 150
   weak, 133
rule, 40
   about a predicate, 40
   body, 40
   body literal, 40
   disjunctive, 167
   head, 40
   range-restricted, 48
   recursive, 54
   subsumed, 128
   variables, 43
semantics, 122
   allows transformation, 124
   classes, 122
   consistency-preserving, 122
   definite, 122
   equivalence-preserving, 123
   normal, 122
   semi-normal, 122
   strictly equivalence-preserving, 123
   strong model property, 122
   semi-normal interpretation, 109
   semi-normal semantics, 122
   Skolem constant, 31
   SLD-resolution, 72
   SLD-tree, 72
   lemma, 72
social choice theory, 123
stable model, 112
stratification, 108
stratified program, 108
strictly equivalence-preserving, 123
strictly range-restricted program, 49
strong model property, 122
substitution, 42
   application, 42
   ground, 43
subsumed rule, 128
supported model, 112
symbol, 29
table, 27
tail-recursive program, 87
term, 39
   in Prolog, 31
term-re-wriing, 130
termination, 130
three-valued logic, 109
transformation, 124
    confluent, 131

UNA, 30
unfolding, 125
union
    of Herbrand interpretation and program, 41
unique names, 30
universe, 30

variable, 38
    anonymous, 39
variables occurring in a rule, 43

weak reduction operator, 151
weak residual program, 133
well-founded model, 117
WFS, 117

XSB, 70, 114