Goal-Directed Forward Chaining
for
Logic Programs

Inauguraldissertation
zur
Erlangung der Doktorwürde
der
Mathematisch-Naturwissenschaftlichen Fakultät
der Rheinischen Friedrich-Wilhelms-Universität Bonn

vorgelegt von
Wolfram Burgard

Bonn
1991

Angefertigt mit Genehmigung der Mathematisch-Naturwissenschaftlichen Fakultät der Rheinischen Friedrich-Wilhelms-Universität Bonn.

1. Berichterstatter: Univ.-Prof. Dr. Armin B. Cremers

2. Berichterstatter: Univ.-Prof. Dr. Hans Kleine Büning

Tag der mündlichen Prüfung im Hauptfach: 18. November 1991

Tag der mündlichen Prüfung im Nebenfach: 12. November 1991

Abstract

This dissertation discusses an approach to goal-directed forward chaining for logic programs introduced by Yamamoto and Tanaka in 1986. It presents solutions to different problems with this approach. One problem is caused by the link clauses goal-directed forward chaining uses to focus on relevant clauses. Their number may be very large in the context of propositional or datalog programs and even infinite in the presence of recursive programs with function symbols. We present an approach to reduce their number in the first case and to obtain a more general but finite set of link clauses in the second case. We introduce goal-directed forward chaining as a linear resolution strategy called GDFC-resolution and show that GDFC-resolution is sound and complete for definite logic programs. We compare different properties of corresponding SLD- and GDFC-trees which lead to the conjecture that GDFC-resolution on average is more efficient than SLD-resolution for propositional logic programs. We confirm this conjecture by experimental results. We prove that GDFC-resolution is more efficient than SLD-resolution for different datalog programs such as the procedure defining the ancestor relationship. Finally, we show that GDFC-resolution terminates if SLD-resolution does and both approaches apply the left first computation rule which is also implemented in Prolog. All results suggest that goal-directed forward chaining, at least in the context of datalog programs, is a control strategy which realizes an efficient forward chaining approach and thus may be an interesting alternative to SLD-resolution.

Acknowledgements

First I would like to thank Armin B. Cremers for fruitful advice. He initiated my interest in this topic and supervised this research. Hans Kleine Büning's willingness to act as the co-referee of this dissertation and his constructive comments are highly appreciated. Special thanks are due to Lutz Plümer for his never ending engagement in many substantial discussions. Thanks also go to Elmar Eder, Michael Hanus, Peter Heusch, Ralf Hinze, Jürgen Kalinski, Udo Lipeck, Stefan Lüttringhaus, Hans Simon, Ewald Speckenmeyer and Hermann Stamm for fruitful discussions. I am grateful to the members of the 'Logic Programming Groups' at the Universities of Dortmund and Bonn for helpful comments. Finally, I want to thank Jürgen F. Münstermann for proofreading the manuscript.

CONTENTS

Introduction **1**

Goal-Directed Forward Chaining **9**

2.1 A Meta-Interpreter for Goal-Directed Forward Chaining 9
2.2 The Approach of Yamamoto and Tanaka 13
2.3 Properties of Goal-Directed Forward Chaining 16

Generating Link Clauses **21**

3.1 Finite Link Clause Programs 21
3.2 Reducing the Number of Link Clauses 37
3.3 Summary 48

GDFC-Resolution **51**

4.1 The Resolution Strategy 51
4.2 Soundness and Completeness of GDFC-Resolution 58
4.3 Summary 71

Efficiency of GDFC-Resolution **73**

5.1 Corresponding SLD- and GDFC-Trees 73
5.2 Average Complexities of Propositional Binary Programs 88
5.3 Complexities of Taxonomic Hierarchies 97
5.4 Complexities of Transitive Closures 101
5.5 Summary 113

Experimental Results **115**

6.1 Efficiency for Propositional Programs 115
6.2 Efficiency for Datalog Programs 121
6.3 Effectivity of the Link Clause Optimization 126
6.4 Summary 128

Termination of GDFC-Resolution 131

Related Work 145
8.1 Deductive Databases 145
8.2 Choosing the Rule Direction 150

Conclusions 153

References 157

Index 165

Curriculum Vitae 167

Chapter 1

INTRODUCTION

One of the basic ideas of logic programming, first expressed by Kowalski [41, 42, 43], is that an algorithm can be decomposed into two distinct components, the logic and the control. Whereas the logic describes the problem to be solved, the control specifies how it has to be solved. In its ideal form, logic programming offers the programmer the possibility only to specify the logic component of an algorithm leaving the control completely to the logic programming system.

Prolog is a widely spread programming language which borrows its basic constructs from logic, thus allowing the use of logic as a programming language. Since its first implementation in 1972, many efficient Prolog interpreters and compilers have been implemented and a lot of applications have been developed. Prolog is a logic programming system which is based on the resolution principle [62], and the kernel of standard Prolog systems realizes a specialized form of resolution called SLD-resolution [46]. SLD-resolution which is a generalization of the modus tollens rule is a goal-driven approach; starting with a given goal, SLD-refutation procedures attempt to derive a contradiction, represented by the empty clause, by repeatedly replacing atoms of the goal by the bodies of matching clauses after applying the matching substitution. Therefore, the built-in control behaviour of Prolog can roughly be described as backward chaining combined with backtracking.

From the viewpoint of expert systems tools, Prolog belongs to the programming languages with symbol manipulation capabilities in which all information is represented by facts and rules. Demonstrating how important features of expert systems such as explanation of behaviour, probabilistic reasoning and inference generated requests of data can easily be programmed in Prolog, many authors have argued that Prolog is a powerful tool for implementing knowledge-based systems [14, 64, 72]. Whenever computer programs

use facts and rules to store information the question arises which inference strategy should be chosen. In general we distinguish two main strategies: forward chaining and backward chaining. Generally, it is a difficult task to choose the optimal inference direction. Above that, only a few weak heuristics are known for that purpose [13]. In addition to efficiency aspects, the adequacy has to be taken into account, since humans often find it more natural to understand rules bottom-up.

Forward chaining as a data-driven approach [26, 27] can be applied to either condition-action systems, so called production systems, or even to logic based antecedent-consequent rules [89]. The production system approach, of which OPS5 is an efficient representative [10, 17], has been proven to be a good tool for the development of expert system applications in a wide variety of domains [88]. The efficiency of production systems has its origin in a fast inference engine which is based on the Rete match algorithm described by Forgy in [21]. Production system interpreters repeatedly perform a match-select-execute cycle [29]. In the match phase the condition parts of the rules are compared with the facts to compute the conflict set, which is the set of all executable rule instances. Subsequently, the selection strategy picks one or more rule instances out of the conflict set. Finally, in the execution step the actions of the rule instances are applied, which results in a modification of the set of facts. The process stops successfully, if a certain goal state is reached, or unsuccessfully, if no condition part of a rule can be satisfied.

Because of the great importance of rule based languages several attempts have been made to implement OPS5-like production systems on top of Prolog. In this context we have to mention the inference engine KORE/IE [66, 67], the commercially available system *flex* [85] as well as the approaches described in [22, 40, 83]. Such a linkage between production systems and logic programming leads to hybrid systems which are difficult to understand and to maintain, since they only have operational semantics and the programmer has to consider two different paradigms.

In contrast to that, logic based approaches modify only the control component; they are generally not based on a special syntax and do not change the semantics of the underlying logic program. Forward chaining approaches of this class, which often can be regarded as a generalization of the modus ponens rule, are the 'naive' and 'semi-naive' methods [4] and LLNR-resolution [77]. These strategies are a special case of unit-resolution [12] and are closely related to classical fixed point procedures based on bottom-up evaluation [11].

However, they also can be seen as a special case of production systems in which only additions to the data base are allowed.

The following Examples 1.1 and 1.3, which are due to [26], compare the efficiency of backward and forward chaining for different taxonomic hierarchies.

EXAMPLE 1.1 (Evaluating Taxonomic Hierarchies):

Consider the following set of clauses specifying the taxonomic hierarchy of animals which is illustrated in Figure 1.2. Suppose our goal is to show that zeke is an animal.

animal(X)←insect(X)
animal(X)←mammal(X)
insect(X)←ant(X)
insect(X)←bee(X)
insect(X)←spider(X)
mammal(X)←lion(X)
mammal(X)←tiger(X)
mammal(X)←zebra(X)
zebra(zeke)←

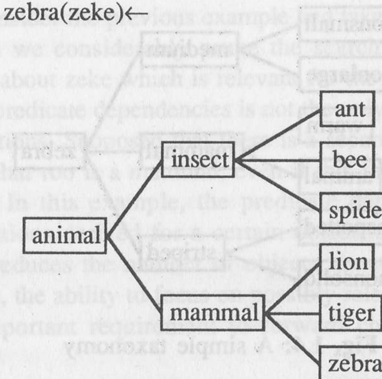

Fig. 1.2: A taxonomic hierarchy of animals

If we use forward chaining then we always need only two steps to reach the goal. Applying backward chaining, then, depending on the search strategy, we possibly have to traverse a much larger search space. ■

EXAMPLE 1.3 (Evaluating Taxonomic Hierarchies):

Now consider the following set of clauses specifying the taxonomy illustrated in Figure 1.4. Again suppose our goal is to show that zeke is an animal.

nonsmall(X)←medium(X)
nonlarge(X)←medium(X)
warm(X)←mammal(X)
animal(X)←mammal(X)
nonspotted(X)←striped(X)
nonsolid(X)←striped(X)
medium(X)←zebra(X)
mammal(X)←zebra(X)
striped(X)←zebra(X)
zebra(zeke)←

Fig. 1.4: A simple taxonomy

In comparison with the previous example, we now we have an inverse situation. Whereas backward chaining always needs only two inferences, forward chaining possibly has to traverse the complete taxonomic hierarchy. ■

In the previous example SLD-resolution as a goal-driven approach benefits from the fact that the information given with the input query (terms occurring as arguments) and the structure of the underlying program (predicate dependencies) are automatically exploited. Naive forward chaining approaches which are not goal-directed often derive more information than necessary. This situation is illustrated in Figure 1.5. In the worst case they compute the complete least fixed point even if only a small subset of it would be sufficient.

Fig. 1.5: Exhaustive forward chaining

If we again consider the previous example and take into account predicate dependencies, then we considerably prune the search space, since we only derive information about zeke which is relevant for the input query. However, the exploitation of predicate dependencies is not the only technique to eliminate unnecessary derivations. Supposed that there is a second zebra called roo, we possibly conclude that roo is a mammal, even if this fact is completely irrelevant for our goal. In this example, the predicate dependencies restrict the number of informations derived for a certain object, and the exploitation of variable bindings reduces the number of objects about which information is derived. To sum up, the ability to focus on possibly relevant clauses during the inference is an important requirement to forward chaining approaches for logic programs.

A further problem with forward chaining is that the termination problem seems to be harder than it is with backward chaining. The only inevitable source of termination problems in logic programming is the use of recursive clauses and structured terms [58, 78]. In the case of backward chaining termination is guaranteed by the simplification of terms. But if such clauses are evaluated bottom-up, then terms are constructed. If the control component

does not have any information, when it has to stop the inference, then it never terminates because it possibly produces arbitrary large terms. Figure 1.6 illustrates this situation.

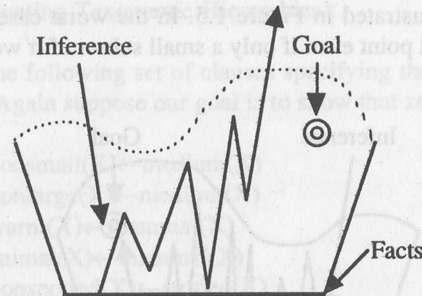

Fig. 1.6: Non-terminating forward chaining

The starting point for this dissertation is the approach to goal-directed forward chaining for logic programs which has been proposed by Yamamoto and Tanaka [90] in 1986. Considering different aspects of semantics, efficiency and termination behaviour we show that this logic based approach, at least for definite logic programs without function symbols, realizes a serious alternative to the standard evaluation strategy of Prolog.

The structure of this dissertation is as follows:

Chapter 2 introduces a meta-interpreter for goal-directed forward chaining and presents the transformation method of Yamamoto and Tanaka. It discusses the pros and cons of and identifies problems with this approach.

Chapter 3 discusses the generation of the link clauses which are used by goal-directed forward chaining to focus on possibly relevant clauses. It presents an approach to reduce the number of link clauses in order to save space. Furthermore, it gives an answer to the question how to handle definite programs with a possibly infinite number of link clauses.

Chapter 4 introduces a resolution strategy for goal-directed forward chaining, called GDFC-resolution. It shows that the meta-interpreter for goal-directed forward chaining indeed implements this resolution strategy. The

main results of this chapter are contained in two theorems concerning the soundness and completeness of GDFC-resolution for definite logic programs. This shows that goal-directed forward chaining, which can be combined with SLD-resolution in a clean way, indeed provides operational semantics which coincide with the declarative and denotational semantics. From a theoretical point of view both theorems, in combination with their proofs, build the basis for the three following chapters, since they tell us how to construct a GDFC-refutation out of an SLD-refutation and vice versa.

Chapter 5 is concerned with the efficiency of GDFC-resolution. For that purpose it compares different properties of GDFC- and SLD-trees. We show that corresponding trees contain the same number of success branches and that there is a success-branch of a certain length in an SLD-tree if and only if there is a success-branch with the same length in the corresponding GDFC-tree. If we always apply the left-first computation rule, then the failure branches in GDFC-trees are not shorter than the corresponding failure branches in SLD-trees. Moreover, a GDFC-tree contains at most as many failure branches as the corresponding SLD-tree. This leads us to the conjecture that GDFC-resolution, at least for propositional programs, on average is more efficient than SLD-resolution. We consider certain classes of propositional and datalog programs and show that indeed GDFC-resolution is more efficient than SLD-resolution for these classes. For example, for the procedure defining the ancestor relationship GDFC-resolution always needs fewer inferences than SLD-resolution to compute all answers of a goal.

Chapter 6 presents experimental results comparing the efficiency of GDFC- and SLD-resolution. It describes benchmarks confirming the conjecture expressed in the previous chapter. Further benchmarks show that GDFC-resolution is very efficient even for datalog programs. Finally it analyzes the approach to reduce the number of link clauses and presents experimental results suggesting that it may produce considerable space savings.

Chapter 7 addresses the termination problem. The main result of this chapter is that, supposed we apply the left-first computation rule and use a special approach to compute the link clauses, GDFC-resolution terminates if SLD-resolution does. It furthermore demonstrates how automatic termination proofs for logic programs proposed by Plümer [57, 58, 59] can be adopted.

Chapter 8 compares goal-directed forward chaining with other approaches providing a bottom-up evaluation of logic programs. It considers the field of deductive databases, where it already has been shown that enhanced

bottom-up approaches are more efficient than top-down strategies [80], and discusses how GDFC-resolution fits into this area. It presents an extension of goal-directed forward chaining to a set-oriented strategy which is closely related to APEX [49].

Finally, in Chapter 9 we summarize the results of this dissertation.

Chapter 2

GOAL-DIRECTED FORWARD CHAINING

This chapter introduces goal-directed forward chaining for logic programs. The first section presents a meta-interpreter implementing this control strategy. The definition of the linear resolution strategy GDFC-resolution, given in Chapter 4, is based on this meta-interpreter. The second section presents the translation method developed by Yamamoto and Tanaka. This approach allows us to transform a logic program into a forward chaining program which directly can be executed by Prolog. Finally, the third section concerns the pros and cons of and identifies the problems with this approach. The development of solutions to these problems is the main subject of this dissertation. For the foundations of logic programming we refer the reader to [2, 46]. Introductions to Prolog can be found in [15, 71].

2.1 A Meta-Interpreter for Goal-Directed Forward Chaining

The basic idea of goal-directed forward chaining, which is motivated by the approach of Yamamoto and Tanaka to translate Prolog programs into forward chaining production rules [90], can informally be described as follows: supposed G is a goal, A is an atom in G and B← is a fact possibly relevant for A, select a non-unit clause whose leftmost body literal unifies with B and whose head is relevant for A, solve the remaining body literals and take the resulting rule head as a new fact. This process is repeated until the resulting rule head is unifiable with A (success) or the evaluation of one of the remaining body literals fails (failure). Whenever a failure is encountered backtracking is invoked to choose alternative facts or rules.

Program 2.1 contains a meta-interpreter which specifies the computational model of goal-directed forward chaining more precisely. We define the

meta-interpreter at the goal reduction level. The top-level relation is gdfc_solve(Goal) which is true if Goal is true with respect to the program being interpreted.

```
gdfc_solve(true)←

gdfc_solve((A,B))←
    gdfc_solve(A),
    gdfc_solve(B)

gdfc_solve(B)←
    clause(B,true)

gdfc_solve(B)←
    clause(A,true),
    link(A,B),
    subgoal(A,B)

subgoal(A,B)←
    clause(B,(A,Body)),
    gdfc_solve(Body)

subgoal(A,B)←
    clause(C,(A,Body)),
    link(C,B),
    gdfc_solve(Body),
    subgoal(C,B)
```

Program 2.1: A meta-interpreter for goal-directed forward chaining

This meta-interpreter focuses on relevant clauses by means of link goals where link(A,B) means that A is possibly relevant for B. These link goals are evaluated w.r.t. a link clause program which is generated from the program being interpreted. Modulo variable renaming, the link clause program contains the following clauses: for each non-unit clause in the program with head B and leftmost body literal A it contains a clause link(A,B)←. It furthermore contains the transitive closure of the link clauses, that is, the clause link(A,D)θ← for each pair of link clauses link(A,B)← and link(C,D)← where θ is the most general unifier of B and C, i.e., θ=mgu(B,C).

Let us consider the procedural reading of this interpreter. The gdfc_solve/1 fact states that the empty goal, represented by the atom true, is solved. This fact is necessary, because the clause/2 goals in the two subgoal/2

clauses only match non-unit clauses with at least two body literals so that we have to append the constant true to all non-unit clauses containing exactly one body-literal. The second clause concerns conjunctions and means: 'To solve a conjunction (A,B), solve A and solve B.' Both clauses are also contained in the standard three clause meta-interpreter for pure Prolog comprising Program 2.2.

The procedural reading of the third clause is: 'To solve B, choose a fact A← such that A and B unify.' The fourth clause covers the case that A← is a possibly relevant fact for B. To solve B, choose a unit-clause A← for which the goal ←link(A,B) has a solution and show that A is a subgoal for B.

The last two clauses concern the subgoal-goal relation between atoms. To show that A is a subgoal for B, select a clause whose leftmost body literal unifies with A and whose head simultaneously unifies with B, and recursively solve the remaining part of the body.

The last clause applies to situations where the head of the selected clause is relevant for B. It states: 'To show that A is a subgoal for B, choose a clause whose leftmost body literal unifies with A and for which there is a link from the head C to B, i.e., there is a solution to ←link(C,B), recursively solve the remaining part of the body and show that C is a subgoal for B.'

```
solve(true)←

solve((A,B))←
    solve(A),
    solve(B)

solve(A)←
    clause(A,B),
    solve(B)
```

Program 2.2: A meta-interpreter for pure Prolog

EXAMPLE 2.3 (Connectivity in a Graph):

Consider the following program specifying the connectivity in a graph, i.e., its reflexive and transitive closure.

```
p(X,X)←
p(X,Z)←e(X,Y),p(Y,Z)
```

Suppose the graph is defined by the fact e(a,b)← and the link clause program contains the following link clause:

```
link(e(X,Y),p(X,Z))←
```

Figure 2.4 contains a trace of Program 2.1 computing both solutions for the query ←gdfc_solve(p(a,X)). ■

```
call gdfc_solve(p(a,X))
     call clause(p(a,X),true)
     exit clause(p(a,a),true)
exit gdfc_solve(p(a,a))                       X=a
     call clause(Y,true)
     exit clause(p(Z,Z),true)
     call link(p(Z,Z),p(a,X))                 fail
     exit clause(e(a,b),true)
     call link(e(a,b),p(a,X))
     exit link(e(a,b),p(a,X))
     call subgoal(e(a,b),p(a,X))
          call clause(p(a,X),(e(a,b),B))
          exit clause(p(a,X),(e(a,b),p(b,X)))
          call gdfc_solve(p(b,X))
               call clause(p(b,X),true)
               exit clause(p(b,b),true)
          exit gdfc_solve(p(b,b))
     exit subgoal(e(a,b),p(a,b))
exit gdfc_solve(p(a,b))                       X=b
```

Fig. 2.4: Tracing the meta-interpreter

We conclude this section discussing this meta-interpreter as a specification of a resolution strategy. The second clause specifies the selection of the leftmost goal as the goal to reduce. The third clause states that the selected literal can be dropped if there is a unifying fact. The fourth clause allows us to replace a selected literal B by subgoal(A,B), if A is a fact possibly relevant for B. Using the fifth clause we can replace a literal subgoal(A,B) by the remaining literals of a clause, after the head has been unified with B and the leftmost body literal has been unified with A. The last clause stands for cases where the head is possibly relevant for B. Then, additionally to the body literals starting with the second, the literal subgoal(C,B) is added, where C is the head of the selected clause.

2.2 The Approach of Yamamoto and Tanaka

Yamamoto and Tanaka presented an approach to transform a definite program P to a forward chaining logic program F_P [90]. This technique has been adopted from the bottom-up parsing system BUP, which is implemented in Prolog [1, 53]. Different optimizations of BUP which are based on partial evaluation are proposed in [74]. Algorithm 2.5 implements the approach of Yamamoto and Tanaka.

ALGORITHM 2.5 (Compiler Proposed by Yamamoto and Tanaka):

1) For each fact A←∈P, F_P contains the clause

   fact(A)←

2) For each n-ary predicate symbol p occurring in P, F_P contains exactly one terminate clause

   $p([X_1,\ldots,X_n],p(X_1,\ldots,X_n))\leftarrow$

   where $X_1,\ldots,X_n$ are distinct variables.

3) For each clause $p(t_1,\ldots,t_m)\leftarrow q(u_1,\ldots,u_n),A_2,\ldots,A_k\in P$ with $k\geq 1$, F_P contains the rule clause

   $q([u_1,\ldots,u_n],G)\leftarrow$
   $link(p(t_1,\ldots,t_m),G),$
   $goal(A_2),\ldots,$
   $goal(A_k),$
   $p([t_1,\ldots,t_m],G)$

4) Modulo variable renaming, F_P contains a link clause link(A_1,A)←, for each clause A←A_1,…,A_n∈P. Modulo variable renaming, for each pair of link clauses link(A,B)← and link(C,D)← with θ is an mgu for B and C, the clause link(A,D)θ← is added to F_P. Furthermore, F_P contains the clause link(X,X)←.

5) Finally, the following goal clause is added to F_P:

```
goal(G)←
    fact(H),
    link(H,G),
    H=..[Head| Args],
    Call=..[Head,Args,G],
    call(Call)
```

6) If ←A_1,…,A_n is a definite goal for P, then the corresponding goal for F_P is ←goal(A_1),…,goal(A_n). ■

The way in which goals for F_P are evaluated is specified by the meta-interpreter comprising Program 2.6.

```
gdfc_solve(true)←
gdfc_solve((A,B))←
    gdfc_solve(A),
    gdfc_solve(B)
gdfc_solve(B)←
    clause(A,true),
    link(A,B),
    subgoal(A,B)
subgoal(A,A)←
subgoal(A,B)←
    clause(C,(A,Body)),
    link(C,B),
    gdfc_solve(Body),
    subgoal(C,B)
```

Program 2.6: A meta-interpreter for the approach of Yamamoto and Tanaka

The first two clauses of this interpreter are the same as in Program 2.1. The third clause implements the goal clause. The first literal in the body of this clause selects all unit clauses, and therefore is equivalent to the first body literal in the goal clause. The second condition occurs in both clauses and checks whether the selected fact is possibly relevant for the goal. The third literal implements the last three conditions of the goal clause which are used to construct a call of the form p([t_1,…,t_m],G) where p(t_1,…,t_m) is the fact retrieved by the first literal. Whereas the subgoal/2 fact implements the terminate clauses, the last clause replaces all rule clauses.

Even if both meta-interpreters for goal-directed forward chaining have a different operational behaviour, we can show by means of unfolding [75], that they are declaratively equivalent. Let us first consider the third clause. If we unfold the link goal with the reflexive link clause, we can replace this clause by the following two clauses.

```
gdfc_solve(B)←
    clause(B,true),
    subgoal(B,B)
gdfc_solve(B)←
    clause(A,true),
    link(A,B),
    subgoal(A,B)
```

Analogous transformations can be applied to the second subgoal/2 clause. From the results described in [7, 47, 75] it follows that the obtained program is declaratively equivalent to Program 2.1 w.r.t. the top-level goal.

We now can drop the last literal of the third and fifth clause of the resulting program, since we can always reduce it to true using the first subgoal/2 clause subgoal(A,A)←. After that the fact subgoal(A,A)← is redundant, since it cannot produce additional answers to gdfc_solve/1. If we remove it from the program, we indeed obtain the meta-interpreter defined by Program 2.1. This operation is admissible if we assume that subgoal/2 is not a top-level relation. The same argument holds for the reflexive link clause link(X,X)←: after we used it as input clause for unfolding we can remove it from the link clause program.

One difference between both meta-interpreters is, that we do not need the reflexive link clause using Program 2.1. However, Program 2.1 has the disad-

vantage that it always scans the sequence of unit and non-unit clauses twice, while both parts are only scanned once with Program 2.6. Thus, from the viewpoint of runtime efficiency, the second approach can be seen as a more efficient implementation of goal-directed forward chaining.

Program 2.1, however, has an important advantage if we compare the number of resolutions needed to solve a goal with that of Program 2.6. Let us assume that the selection of a relevant clause, i.e., to choose a clause and solve the link goal takes only one resolution. Suppose ←A is a single literal goal and P contains a fact B← such that B and A are unifiable. Then Program 2.6 needs two inferences to solve ←A even if B← is selected. Whereas the first logical inference is necessary to select the unit clause, the second inference is needed to check whether it unifies with the goal. Program 2.2 as well as Program 2.1, however, need only one inference. Although Program 2.6 may be more efficient with respect to the effective runtime behaviour, we define GDFC-resolution in Chapter 4 on the basis of Program 2.1. This simplifies the construction of corresponding refutations, which is necessary to show the soundness and completeness of GDFC-resolution, as well as the comparison of the length of refutations which is necessary to judge the efficiency of GDFC-resolution. For experimental studies concerning the runtime efficiency of goal-directed forward chaining, however, we use Program 2.6.

2.3 Properties of Goal-Directed Forward Chaining

In the previous sections we presented a meta-interpreter for goal-directed forward chaining and a corresponding transformation approach. In this section we discuss some of the features of and problems with this approach.

First it is worth mentioning that the rule translation method produces efficient Prolog programs, since the head of a resulting clause is the first condition of the premise part of the original non-unit clause. Accordingly, even if there are a lot of non-unit clauses, the system can efficiently pick out the necessary rule without searching through the whole sequence of rules [90]. Furthermore, the obtained program can be compiled by a Prolog compiler in order to gain further speed up.

The meta-interpreter and the translation approach both can be enhanced in the same way as the standard interpreter for pure Prolog. Sterling [72] gives

a survey of different possible extensions of meta-interpreters which are useful for expert systems. Such enhancements can be used to implement reasoning about uncertainty, dialogue capabilities (query-the-user) or explanation facilities. Yamamoto and Tanaka already have shown how such dialogue and explanation functions can be implemented in the rule compiler. The following clause, inserted after the second, extends the behaviour of the meta-interpreter by implementing a switch to backward chaining. It assumes that a procedure backward/1 is defined, which specifies which goals should be evaluated by the interpreter for pure Prolog.

```
gdfc_solve(B)←
    backward(B),
    !,
    solve(B)
```

A similar clause can be added to Program 2.2. This extension realizes a neat integration of backward and forward chaining, because the system can switch between both inference strategies dynamically at runtime. In a first step the meta-relation backward/1 can define a fixed direction for each relation. However, further extensions may enable the system to exploit runtime information and to choose the optimal inference strategy for each particular goal depending on this information.

In Chapter 1 we identified the ability to focus on possibly relevant clauses as an important requirement to forward chaining approaches. Goal-directed forward chaining obtains this necessary feature by means of the link clauses which allow a fast selection of possibly relevant clauses exploiting variable bindings of the goal and predicate dependencies of the program.

Yamamoto and Tanaka already mentioned the problem that a huge number of link clauses may be obtained when the transitive closure of the link clauses is statically generated, i.e., at the translation process. They argued, however, that the dynamic generation of the link clauses during the inference process is a time consuming alternative which may decrease the performance considerably.

A further problem is, that the transitive closure of the link clauses may become infinite in the case of definite programs containing function symbols.

EXAMPLE 2.7 (Natural Numbers):

Consider the following procedure defining the natural numbers

```
nat(0)←
nat(s(X))←nat(X)
```

The link clause for this program is

```
link(nat(X),nat(s(X)))←
```

If we compute the transitive closure of this link clause we obtain the following sequence:

```
link(nat(X),nat(s(X)))←
link(nat(X),nat(s(s(X))))←
link(nat(X),nat(s(s(s(X)))))←...
```

Thus the transitive closure contains infinitely many link clauses of the form link(nat(X),nat(s^n(X)))← where $n \geq 1$. ■

One solution to this problem is to compute the transitive links needed to solve the actual goal dynamically at runtime. Then, however, we have to solve a termination problem. For, if there are infinitely many atoms which are possibly relevant for the actual goal or infinitely many atoms for which particular atom may be relevant for, then the process generating the transitive links may not terminate.

Concerning the termination behaviour of their approach, Yamamoto and Tanaka [90] wrote: 'In the case of first order predicate logic, recursive rules are sometimes harmless. However, we have not yet solved the problem of how to determine what kinds of recursion this system can and cannot handle.'

There are two further questions concerning goal-directed forward chaining. The first one affects the operational semantics and asks, whether each computed answer is correct (soundness), and, whether each correct answer can be computed using a fair search rule (completeness). The second one addresses the efficiency of goal-directed forward chaining: are there any programs for which the control strategy implemented by goal-directed forward chaining is more efficient than the standard strategy of Prolog?

To sum up, the following questions have to be answered:

- How can we solve the problem of infinitely many link clauses?
- How can we reduce the number of link clauses if their transitive closure is finite?
- Is goal-directed forward chaining sound and complete?
- How efficient is goal-directed forward chaining?
- Which termination behaviour does goal-directed forward chaining have?

The following chapter gives an answer to the first two questions. Chapter 4 concerns the third question and shows the soundness and completeness of GDFC-resolution, a linear resolution strategy whose definition is based on the meta-interpreter for goal-directed forward chaining. Chapter 5 and Chapter 6 show that goal-directed forward chaining may be more efficient than SLD-resolution for several classes of propositional and datalog programs, and this way give answers to the fourth question. Chapter 7 addresses the last question and presents a special procedure for the dynamic computation of the transitive links which has the effect that GDFC-resolution always terminates if SLD-resolution does.

Chapter 3

GENERATING LINK CLAUSES

The meta-interpreter for goal-directed forward chaining uses link clauses to focus on relevant clauses and to exploit variable bindings of the input query. The link clauses cause a termination problem, since their number may be infinite. But even if their number is limited, they require a large amount of storage. Thus it is an important question, how their number can be reduced. In this chapter we will be concerned with both problems. We begin with the termination problem and present an approach to generate a possibly more general but finite set of link clauses. We continue with the space problem and introduce a solution to it. Finally we discuss a further approach to optimize the link clauses.

3.1 Finite Link Clause Programs

As we have seen in the previous chapter, the number of link clauses may be infinite, if the source program contains recursive procedures defined over recursive data structures. Therefore, criteria which are necessary or sufficient for the infinity are highly desired. In addition we are interested in approaches to obtain a finite number of link clauses without affecting soundness and completeness of this approach.

DEFINITION 3.1 (Set of Link Clauses):

Let P be a definite program. Modulo variable renaming the *set of link clauses* $Link_P$ for P contains a clause $link(B_1,B)\leftarrow$ for each clause $B\leftarrow B_1,\ldots,B_n$ with $n\geq 1$ in P. We say $Link_P$ is *cyclic* (or *recursive*) if there are link clauses $l_1,\ldots,l_n \in Link_P$ $(n\geq 1)$ such that the predicate symbol of the right atom in l_i

equals the predicate symbol of the left atom in l_{i+1} and the predicate symbol of the right atom in l_n equals the predicate symbol of the left atom in l_1. ■

The link clause program, which is needed to implement goal-directed forward chaining, consists of the transitive closure of the set of link clauses.

DEFINITION 3.2 (Link Clause Program):

Let P be a definite program. The *link clause program* L_P for P is a set of link clauses containing all clauses of $Link_P$. Furthermore, for each pair of clauses link(A,C)← and link(D,B)←∈ L_P with θ is an mgu for C and D, and modulo variable renaming, L_P contains the clause link(A,B)θ←. The *reflexive link clause program* L_P^{ref} is $L_P \cup$ {link(X,X)←} ■

If we apply the approach of Yamamoto and Tanaka, i.e. Program 2.6, then indeed we need the reflexive link clause program L_P^{ref}.

Example 3.3 (Link Clauses for the append Procedure):

Consider the well known procedure for appending two lists which is

```
append([],X,X)←
append([X|Xs],Ys,[X|Zs])←
    append(Xs,Ys,Zs)
```

Whereas $Link_P$ consists of only one clause, namely

```
link(append(Xs,Ys,Zs),append([X|Xs],Ys,[X|Zs]))←
```

L_P is infinite; it contains infinitely many clauses of the form

link(append(Xs,Ys,Zs),append([X|Xs],Ys,[X|Zs]))←
link(append(Xs,Ys,Zs),append([X_1,X_2|Xs],Ys,[X_1,X_2|Zs]))←
link(append(Xs,Ys,Zs),append([X_1,X_2,X_3|Xs],Ys,[X_1,X_2,X_3|Zs]))←
... ■

The following theorem shows that the infinity of the link clause program is generally undecidable.

THEOREM 3.4 (Undecidability of the Finiteness of L_P):

It is undecidable whether or not L_P is finite if P is a definite program. ■

Proof: To show the result we apply Rice's theorem [3] which says that each non-trivial property of Turing machines is undecidable. Thus, we first have to show, how we can represent an arbitrary Turing machine program by a set of link clauses.

Let a Turing machine T be a subset of S×A×S×A×{l,r}, where S is a finite set of states, A is the finite alphabet of T and {l,r} is the set of possible directions to which the head can move. An element (s,a,t,b,r) of T is interpreted as follows: if the current state of T is s and the tape-symbol at the head position is a, then T prints b, moves the head one position to the right and changes its state to t. For each Turing machine there are two special states s_0, $s_f \in S$ where s_0 is the initial state and s_f is the final state. Let f be a Turing computable function. We say T computes f, if T starting in s_0 with tape inscription x reaches s_f with tape inscription y if and only if y=f(x).

We now construct a logic program P_T implementing T. To represent the tape of T we use two (generally incomplete) lists where the first one represents the tape left from the actual head position and the second the tape at the head position and right from it. For example, if T moves left, then we simulate this by removing the first element of the list representing the left part of the tape and inserting it at the first position of the list representing the right part of the tape. For example, if the tape is ([a|L],[b|R]) then we have the tape (L,[a,b|R]) after T moved one field left.

Accordingly, for each tuple (s_i,a_m,s_j,a_n,r) we add the clause

tm(s_i,L,[a_m|R])←tm(s_j,[a_n|L],R)

and for each tuple (s_i,a_m,s_j,a_n,l) we add the clause

tm(s_i,[S|L],[a_m|R])←tm(s_j,L,[S,a_n|R])

to P_T. Finally, for the halting state s_f we add the fact

tm(s_f,_,_)←

We start T with the goal $\leftarrow tm(s_0,left,right)$ where s_0 is the initial state and left and right are terms representing the initial tape contents. Clearly T reaches its final state if and only if there is a refutation of $P \cup \{\leftarrow tm(s_0,left,right)\}$. In this case the terms occurring at the second and third position of the last non-empty goal contain the computed output.

Let us now consider the set of link clauses $Link_{P_T}$ for P_T. For each clause representing a transition where T moves one field to the right. $Link_{P_T}$ contains a clause

$$link(tm(s_j,[a_n|L],R),tm(s_i,L,[a_m|R]))\leftarrow$$

Furthermore, for each transition where T moves left, $Link_{P_T}$ contains a link clause

$$link(tm(s_j,L,[S,a_n|R]),tm(s_i,[S|L],[a_m|R]))\leftarrow$$

Each link clause in $Link_{P_T}$ represents a one step transition between states. Accordingly, the link clauses in L_{P_T} represent all possible transitions between states. Each element

$$link(tm(s_i,left_i,right_i),tm(s_j,left_j,right_j))\leftarrow$$

of L_{P_T} means that T, starting with $tm(s_j,left_j,right_j)$, scans all fields represented by the prefixes of $left_j$, $right_j$, $left_i$ and $right_i$ until it reaches $tm(s_i,left_i,right_i)$. Otherwise, the variables L and R representing the remainders of the lists in the link clauses of $Link_{P_T}$ would not be bound to lists with non-empty prefixes during the computation of L_{P_T}. Let us first consider an example before we continue with the proof.

EXAMPLE 3.5 (Turing Machines):

Consider the Turing machine T_a which simply replaces the blank symbol by a and then stops. T_a consists of the tuple (s_0,b,s_f,a,r) only. Consequently, the corresponding logic program P_a contains the following two clauses:

$$tm(s_0,L,[b|R])\leftarrow tm(s_f,[a|L],R)$$
$$tm(s_f,_,_)\leftarrow$$

$Link_{P_a}$ and L_{P_a} contain only one link clause, namely

$$link(tm(s_f,[a|L],R),tm(s_0,L,[b|R]))\leftarrow$$

Now consider T_c which replaces a word of the form a^n by c^n.

$$(s_0,a,s_0,c,r)$$
$$(s_0,b,s_f,b,r)$$

The corresponding logic program P_c contains the following clauses:

$$tm(s_0,L,[a|R])\leftarrow tm(s_0,[c|L],R)$$
$$tm(s_0,L,[b|R])\leftarrow tm(s_f,[b|L],R)$$
$$tm(s_f,_,_)\leftarrow$$

In this case $Link_{P_c}$ contains the clauses

$$link(tm(s_0,[c|L],R),tm(s_0,L,[a|R]))\leftarrow$$
$$link(tm(s_f,[b|L],R),tm(s_0,L,[b|R]))\leftarrow$$

which have an infinite transitive closure so that L_{P_c} contains infinitely many clauses of the form

$$link(tm(s_0,[c|L],R),tm(s_0,L,[a|R]))\leftarrow$$
$$link(tm(s_f,[b|L],R),tm(s_0,L,[b|R]))\leftarrow$$
$$link(tm(s_0,[c,c|L],R),tm(s_0,L,[a,a|R]))\leftarrow$$
$$link(tm(s_f,[b,c|L],R),tm(s_0,L,[a,b|R]))\leftarrow$$
$$link(tm(s_0,[c,c,c|L],R),tm(s_0,L,[a,a,a|R]))\leftarrow$$
$$link(tm(s_f,[b,c,c|L],R),tm(s_0,L,[a,a,b|R]))\leftarrow$$
$$link(tm(s_0,[c,c,c,c|L],R),tm(s_0,L,[a,a,a,a|R]))\leftarrow$$
$$link(tm(s_f,[b,c,c,c|L],R),tm(s_0,L,[a,a,a,b|R]))\leftarrow\ldots$$

The last link clause means that there is a transition where T_c, starting in s_0 with tape inscription [a,a,a,b] and head position at the leftmost a, scans all four fields until it reaches s_f. The tape inscription after this transition is [c,c,c,b] and the head is positioned right from b. L_{P_c} is infinite, since there are infinitely many such transitions. The fact that T_c scans arbitrarily long words of the form a^nb results in arbitrarily long prefixes of the lists in the second and third argument position of the atoms in the link clauses. ■

Our next goal is to show that L_{P_T} is infinite if and only if, for each natural number N, there are two states s_j and s_i and a transition from s_j to s_i during which T scans at least N+1 distinct fields of the tape.

First we show '⇒'. Suppose L_{PT} is infinite. Thus arbitrarily long prefixes of lists must occur in second and third argument positions of the atoms in the link clauses, because the set of states as well as the alphabet of T is finite. Thus there must be at least one link clause of the form

$$\text{link}(\text{tm}(s_i,\text{left}_i,\text{right}_i),\text{tm}(s_j,\text{left}_j,\text{right}_j))\leftarrow$$

such that the length of both prefixes in left_j and right_j, or left_i and right_i exceeds N. Consequently, if T starts with $\text{tm}(s_j,\text{left}_j,\text{right}_j)$ then T scans more than N different tape fields until it reaches $\text{tm}(s_i,\text{left}_i,\text{right}_i)$.

Next we show '⇐'. Assume that, for each natural number N, there is a transition from $\text{tm}(s_j,\text{left}_j,\text{right}_j)$ to $\text{tm}(s_i,\text{left}_i,\text{right}_i)$ where T scans at least N+1 distinct fields of the tape. Consequently L_{PT} must contain the link clause $\text{link}(\text{tm}(s_i,\text{left}_i,\text{right}_i),\text{tm}(s_j,\text{left}_j,\text{right}_j))\leftarrow$. Since T scans more than N fields, the length of both prefixes of left_i and right_i, or left_j and right_j must be greater than N. Since N is not fixed L_{PT} must be infinite.

Let T_∞ be the set of all Turing machines T such that, for each natural number N, there is a transition during which T scans at least N+1 distinct fields of the tape. Clearly for the programs discussed in Example 3.5 we have $T_a \notin T_\infty$ and $T_c \in T_\infty$. Hence T_∞ is a non-trivial subset of all Turing machines. It follows from Rice's theorem that T_∞ is undecidable. Consequently, it is also undecidable whether or not L_P is finite. ■

The following two theorems define program classes which have finite link clause programs.

THEOREM 3.6 (Finiteness of L_P):

L_P is finite if P is a datalog program. ■

Proof: If P is a datalog program then the Herbrand base of P is finite. Consequently, the set A of all non-ground atoms modulo variants we can build from the Herbrand base by replacing terms by variables is finite too. Thus L_P is finite, since, for each atom occurring in a link clause of L_P, there is an atom in A which is equivalent modulo variable renaming. ■

Although the number of link clauses in L_P is finite for datalog programs it may be very large. For example, the size of L_P may depend super-exponentially on the arity of the procedures in P.

EXAMPLE 3.7 (Link Clauses for Datalog Programs):

Consider the following datalog program consisting of the clauses

p(X,Y,Z)←p(Y,Z,X)
p(X,Y,Z)←p(Y,X,Z)

Whereas the first clause establishes a rotation of the arguments, the second clause exchanges the first and the second argument. Both clauses in common can be used to generate any permutation of the arguments. Therefore, L_P contains 6 link clauses which is the factorial of 3.

link(p(X,Y,Z),p(Y,Z,X))←
link(p(X,Y,Z),p(Y,X,Z))←
link(p(X,Y,Z),p(Z,X,Y))←
link(p(X,Y,Z),p(Z,Y,X))←
link(p(X,Y,Z),p(X,Y,Z))←
link(p(X,Y,Z),p(X,Z,Y))← ■

THEOREM 3.8 (Finiteness of L_P):

L_P is finite if P is not left-recursive. ■

Proof: If P is not left-recursive then the link clauses are acyclic, i.e. there is no link from any predicate symbol to itself. Consequently L_P must be finite. ■

Both conditions discussed above are rather restrictive. For example, the append procedure (see Example 3.3) is left-recursive and contains function symbols. However, both results suggest at least two approaches to obtain a finite link clause program:

1) we can transform P to a program which is not left-recursive.

2) we can modify Link_P so that L_P is finite.

Let us consider the first approach. One idea is to change the order of body literals in the clauses of P so that the transformed program is no longer left-recursive. We believe, however, that the application range of this method is rather restricted. Experienced Prolog programmers generally avoid left-recursion. Left recursive rules are inherently troublesome, since they are a source of infinite loops if they are called with inappropriate arguments [56, 71]. Moreover left-recursion is only used when it cannot be avoided, i.e., the clauses contain only one body literal and the termination is guaranteed by a simplification of terms (see Example 3.3). Thus, it is likely that this strategy can be applied very seldom only.

A second idea is to introduce unique predicate symbols which are added to left-recursive procedures as leftmost body literal so that the resulting program is no longer left-recursive. Furthermore, we also have to add corresponding facts to the program. For example, we can transform the append procedure to

```
append([],X,X)←
append([X|Xs],Ys,[X|Zs])←
        append_help,
        append(Xs,Ys,Zs)
append_help←
```

This indeed yields the desired result; the procedure is not left-recursive and the corresponding link clause program is finite. Let us consider the trace for the goal ←gdfc_solve(append([a,b],[d],L)) restricted to calls of the relations gdfc_solve/1 and subgoal/2.

```
←gdfc_solve(append([a,b],[d],L))
←subgoal(append_help,append([a,b],[d],[a|L1]))
←gdfc_solve(append([b],[d],L1))
←subgoal(append_help,append([b],[d],[b|L2]))
←gdfc_solve(append([],[d],L2))
```

The goals for gdfc_solve/1 are exactly the same as those to solve/1 evaluation the same query with Program 2.2, the standard meta-interpreter for pure Prolog. As a matter of fact, the behaviour of the interpreter for goal-directed forward chaining closely relates to backward chaining, if we transform the program this way. Although this approach is very simple, it unfortunately counteracts the effect of our interpreter.

Because we want to retain the forward chaining characteristic of our meta-interpreter, we next investigate, how we can achieve the finiteness of L_P transforming $Link_P$ instead of P.

EXAMPLE 3.9 (Natural Numbers):

Again consider Example 2.7 where we presented a procedure defining the natural numbers. As already mentioned the set of link clauses contains only one clause, but the link clause program is infinite. Now consider the clause which is obtained by renaming the first variable

```
link(nat(Y),nat(s(X)))←
```

Even though neither of the two criteria for the finiteness is met, the corresponding link clause program is finite. The difference is, that the original clause, in contrast to this one, sets up a cyclic data flow (with respect to the transitive closure) in which a structured term is involved. ■

The example above shows, that the infinity of L_P is caused by a cyclic flow of data through argument positions containing structured terms. Subsequently we consider, how we can exploit information about such data flow to formulate a more sophisticated criterion for the finiteness of L_P. The following example explains this idea more precisely.

EXAMPLE 3.10 (Cyclic Data Flow):

Consider the following two link clauses:

```
link(q(X),p(f(X)))←
link(q(f(X)),p(X))←
```

In the first link clause we have a data flow from q_1 (the first argument of q/1) to p_1. In the second link clause the data flow goes from p_1 to q_1. While the direction in the first link clause is from left to right, we have the opposite direction in the second link clause. Since both directions cannot be combined to a cyclic data flow during the computation of the transitive closure, L_P is finite. However, if we add the clause

```
link(p(f(X)),q(f(X)))←
```

L_P becomes infinite, because this clause produces a cyclic data flow in both directions. Whereas the first and the third link clause together produce arbitrarily large terms on the right side of the generated link clauses, we obtain arbitrarily large terms on the left side using the second and third link clause. Thus there is a strong relationship between the side on which arbitrarily long terms occur and the direction of the data flow. ■

The previous example shows, that we need a precise notion of cyclic data flow through argument positions containing structured terms (cyclic migration) which takes into consideration the direction of the data flow. We use the following definition to introduce *migration graphs* which are motivated by the migration sets defined in [49] (see also Section 8.1). Migration graphs represent the flow of data between structured terms in the link clauses. Because there are two possible directions of migration in the link clauses, we always have two different migration graphs representing both directions.

DEFINITION 3.11 (Migration Graphs):

Let P be a definite program. The *migration graphs* $M_{r,Link_P}$ and $M_{l,Link_P}$ for $Link_P$ are multi-graphs defined as follows:

(a) For each link clause link(p(…,t_i,…),q(…,u_j,…))←∈$Link_P$ with u_j resp. t_i is a structured term sharing a variable with t_i resp. u_j, $M_{r,Link_P}$ resp. $M_{l,Link_P}$ contains an arc (p_i,q_j) resp. (q_j,p_i).

(b) If $M_{r,Link_P}$ contains an arc (p_i,q_j), then $M_{r,Link_P}$ contains an arc (q_j,r_k) for each link clause link(q(…,u_j,…),r(…,v_k,…))←∈ $Link_P$ where u_j and v_k share a variable. If $M_{l,Link_P}$ contains an arc (q_j,p_i), then, for each link clause link(r(…,v_k,…),p(…,t_i,…))← where t_i and v_k share a variable, $M_{l,Link_P}$ contains an arc (p_i,r_k). ■

EXAMPLE 3.12 (Migration Graphs):

Consider the following set of link clauses:

link(p(X,g(Y)),q(f(X),Y))←
link(q(X,Y),p(X,Z))←

Whereas $M_{r,Link_P}$ is cyclic, $M_{l,Link_P}$ contains no cycle. Both migration graphs are shown in Figure 3.13. ■

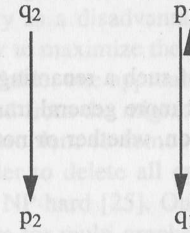

Fig. 3.13: Migration graphs $M_{l,Link_P}$ and $M_{r,Link_P}$

The following theorem establishes a link between cyclic migration and the infinity of L_P by means of migration graphs.

THEOREM 3.14 (Finiteness of L_P):

Let P be a definite program with link clauses $Link_P$. L_P is finite if $M_{r,Link_P}$ and $M_{l,Link_P}$ are acyclic. ■

Proof: Let us consider the case where $M_{r,Link_P}$ is acyclic. It follows from the definition of the migration graph, that there is no cyclic data flow from left to right through an argument position containing a structured term. Consequently it is not possible that arbitrarily large terms occur on the right side of the link clauses in L_P. Since the same argument holds for $M_{l,Link_P}$, L_P cannot contain link clauses with arbitrarily long terms. Thus L_P must be finite. ■

Theorem 3.14 says that L_P is finite if there is no cyclic data flow through argument positions containing structured terms. This result suggests an approach to tackle the problem of infinitely many link clauses. All we have to do is to eliminate cyclic data flow by replacing a sufficiently large number of migrating variables by unique variables in order to delete all cycles in $M_{r,Link_P}$ and $M_{l,Link_P}$.

The approach to replace terms by unique variables to stop possibly infinite processes is also applied in [7] to solve the problem of non-termination during partial evaluation, and in [36] to stop the computing of filters which select possibly relevant tuples during the evaluation of queries to deductive databases.

It is clear that the result of such a renaming is a set of more general link clauses which have a finite but more general transitive closure. Therefore, we next have to answer the question, whether or not such a transformation is admissible.

THEOREM 3.15 (Property of the Link Clause Program):

Let P be a definite program, L_P the link clause program for P and $\leftarrow A$ a single literal goal. If $P \cup \{\leftarrow A\}$ has a refutation, then P contains a fact $B \leftarrow$ such that either A and B are unifiable or L_P contains a link clause the head of which is unifiable with link(B,A). ■

Proof: Let us consider the SLD-refutation of $P \cup \{\leftarrow A\}$ which comes from the left-first computation rule. Suppose $C_1,\ldots,C_n$ are the input clauses used for the derivation of the first goal which is shorter than its predecessor. Let us denote the the head of each C_i by B_i, for $i=1,\ldots,n$, and the leftmost body literal of each C_i by $B_{i,1}$, for $i=1,\ldots,n-1$. Clearly $C_1,\ldots,C_n$ satisfy the following conditions:

1) B_1 and A are unifiable with mgu θ_1,

2) $B_{i,1}\theta_i$ and B_{i+1} are unifiable with mgu θ_{i+1}, for $i=1,\ldots,n-1$, and

3) C_n is a unit clause of the form $B_n \leftarrow$.

If n=1 then A and B_n are unifiable. Otherwise, if n>1, then $Link_P$ contains n-1 link clauses of $link(B_{i,1},B_i)\leftarrow$. The derivation constructed above implies that L_P contains a link clause $link(B_{n-1,1},B_1)\sigma\leftarrow$ the head of which is unifiable with $link(B_n,A)$. ■

The previous theorem implies that L_P establishes a necessary condition for the provability of a given goal. Thus, from this point of view we can use L_P^{fin} instead of L_P, because L_P^{fin} is more general than L_P. In Chapter 4 we show

that such a generalization of L_P is also admissible w.r.t. the soundness and completeness of GDFC-resolution.

Nevertheless, one could regard the fact that the resulting link clause program has a weaker selectivity as a disadvantage of this approach. Thus, we next address the question how to maximize the selectivity of L_P. If no problem dependent information is available, one approach is to minimize the number of arcs which have to be removed from the migration graphs so that they become acyclic. Unfortunately, the problem to remove a minimum number of arcs from a directed graph in order to delete all cycles denoted as the minimum *feedback arc set problem* is NP-hard [25]. Our problem exactly is the minimum feedback arc set problem for multi-graphs. The feedback arc set problem (FAS) is defined as follows :

DEFINITION 3.16 (FAS-Problem):

Input: A directed graph G=(V, A) and a positive integer $k \le |A|$.

Question: Is there a subset $A' \subseteq A$ with $|A'| \le k$ such that A' contains at least one arc from every directed cycle in G? ■

ALGORITHM 3.17 (Find Migrating Variables):

Input: A definite program P.
Output: A set V of variables occurring in the link clauses which have to be renamed so that L_P becomes finite.

1) Compute $Link_P$ (possibly after moving appropriate literals in the clause bodies to the leftmost position). If $Link_P$ contains no migrating variables set $V=\emptyset$ and stop.

2) Compute $M_{r,Link_P}$ and $M_{l,Link_P}$. Compute a feedback arc set for both graphs. Let V={v| v is a variable producing migration in a link clause which corresponds to an arc in the feedback arc set}. ■

At present there are at least two known approaches to approximate the minimum feedback vertex set [63, 69], which can also be used to solve the minimum feedback arc set problem considering the corresponding arc-graph [28, 68]. While the first one with linear runtime was developed by Rosen in

1982, the second one, presented by Speckenmeyer in 1989, has time complexity $O(F_M*|A|^3)$ where F_M is the size of the minimum feedback arc set and A is the set of arcs. Even if Speckenmeyer's approach needs more computation time, it gives promising improvements of Rosen's approach. Fortunately, we can apply both approaches without any changes as subroutines to minimize the number of variables which have to be renamed (see Algorithm 3.17), since the corresponding arc-graph of a multi-graph is a directed graph without multiple arcs.

EXAMPLE 3.18 (Arithmetic Expressions):

Consider the following contextfree grammar for arithmetic expressions defined by the productions:

expression → term
expression → term, [+], expression
term → factor
term → factor, [*], term
factor → identifier
factor → ['('], expression, [')']
identifier → [a]
identifier → [b]
identifier → [c]

A straightforward translation into definite clauses yields the following program [71]:

expression(S,S_0)←term(S,S_0).
expression(S,S_0)←term(S,[+|S_1]),expression(S_1,S_0)
term(S,S_0)←factor(S,S_0)
term(S,S_0)←factor(S,[*|S_1]),term(S_1,S_0)
factor(S,S_0)←identifier(S,S_0)
factor(['('|S],S_0)←expression(S,[')'|S_0])
identifier([a|S],S)←
identifier([b|S],S)←
identifier([c|S],S)←

$Link_P$ is

link(term(S,S_0),expression(S,S_0))←
link(term(S,[+|S_1]),expression(S,S_0))←
link(factor(S,S_0),term(S,S_0))←
link(factor(S,[*|S_1]),term(S,S_0))←
link(identifier(S,S_0),factor(S,S_0))←
link(expression(S,[')'|S_0]),factor(['('|S],S_0))←

Both migration graphs are illustrated in Figure 3.19.

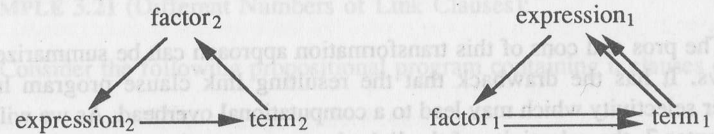

Fig. 3.19: $M_{l,LinkP}$ and $M_{r,LinkP}$ for Example 3.18

Whereas $M_{r,LinkP}$ has two parallel arcs, $M_{l,LinkP}$ contains normal edges only. Speckenmeyer's Markovian approach applied to the corresponding arc-graphs computes the optimal solution by selecting ($expression_1$,$factor_1$) from $M_{r,LinkP}$ and one arc from $M_{l,LinkP}$. If we apply Rosen's approach then, depending on the starting node, we possibly obtain two parallel arcs which have to be removed from $M_{r,LinkP}$. A straightforward extension of Rosen's method which applies Rosen's algorithm n times with each vertex as starting node (n-Rosen) also produces the optimal solution [70]. Thus, in order to make both graphs acyclic, we only have to remove one arc from each graph. Supposed that we obtained the arcs ($expression_1$,$factor_1$) and ($factor_2$,$expression_2$) which come from the last link clause, we only have to replace this clause by

link(expression(S,[')'|S_0]),factor(['('|S_1],S_2))←

For this example it suffices to rename two variables in $Link_P$ in order to obtain a finite link clause program. ■

There are different reasons why Algorithm 3.17 possibly produces sub-optimal solutions only. First, a single arc in a migration graph possibly comes from two variables producing cyclic data flow in the same argument position. Second, one single variable may be involved in both directions of data flow (see the third link clause in Example 3.10), so that we possibly make useless

renaming. One idea to reduce this problem is to split step 2) of Algorithm 3.17. Instead of computing both feedback arc sets simultaneously we simply compute them successively. After computing the first feedback arc set we remove all arcs in the second migration graph which correspond to the same migration.

Even if we do not obtain the optimal solution in all cases we regard this somewhat heuristic approach as well suited, since we can apply it with great success to non-trivial examples such as the well known procedures defined in Example 2.7, 3.3 and 3.18.

The pros and cons of this transformation approach can be summarized as follows. It has the drawback that the resulting link clause program has a weaker selectivity which may lead to a computational overhead. As we will see in Chapter 7, the selectivity of the link clause program has great influence on the termination behaviour of goal-directed forward chaining. A basic precondition of the sufficient termination criterion developed there is that the link clause program which is used has the same selectivity as L_P. In contrast to that the renaming approach has the great advantage that it allows us to deal with finite sets of link clauses so that automatic decisions, whether or not a clause is relevant, always terminate. By all means, in the remainder of this dissertation we can assume that there is a finite link clause program corresponding to the following definition.

DEFINITION 3.20 (Finite Link Clause Program):

For each definite Program P, a *finite link clause program* L_P^{fin} for P satisfies the following conditions:

1) if G is $\leftarrow$link(A,B), then the refutation of $L_P^{fin} \cup \{G\}$ terminates, and,

2) for each correct answer θ for $L_P \cup \{G\}$, there is an answer θ' for $L_P^{fin} \cup \{G\}$ such that $G\theta'$ is more general than $G\theta$.

Accordingly, the *reflexive finite link clause program* $L_P^{fin,ref}$ is obtained from L_P^{fin} by adding the link clause link(X,X)$\leftarrow$. ■

3.2 Reducing the Number of Link Clauses

This section concerns the problem to reduce the number of link clauses in L_P when it is finite. The main idea of the approach is illustrated in the following example.

EXAMPLE 3.21 (Different Numbers of Link Clauses):

Consider the following propositional program containing n clauses of the form

$$p_i \leftarrow p_{i+1}, q_i \qquad (i=1,\ldots,n)$$

Clearly, the transitive closure $Link_P$ consists of $n\cdot(n+1)/2$ link clauses. However, if we change the order of the body literals to

$$p_i \leftarrow q_i, p_{i+1} \qquad (i=1,\ldots,n)$$

then L_P, which is the same as $Link_P$, contains n clauses only. ■

This example shows that the space saving obtained by a reordering of literals in the bodies of the non-unit clause may be of order O(n). Thus it would be very useful to have an algorithm which provides a reordering of literals in the clause bodies such that the number of link clauses for the resulting program is minimal.

As a matter of fact, there is a strong relationship between link clauses and directed graphs in the context of propositional logic programs.

DEFINITION 3.22 (Link Graph):

Let P be a propositional program. The *link graph* LG_P is a directed graph containing an arc (A,B) for each link clause link(A,B)$\leftarrow$ in $Link_P$. ■

Figure 3.23 contains the link graphs for both programs discussed in Example 3.21.

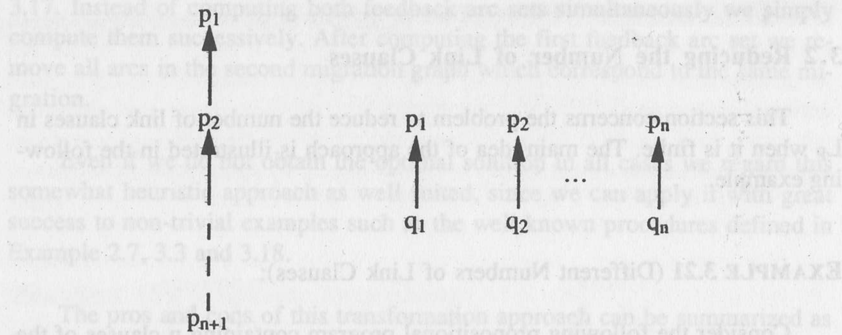

Fig. 3.23: Link graphs

DEFINITION 3.24 (Selection of Literals):

Let P be a definite program and $\sigma:=\sigma(P)$ be a selection which selects exactly one literal from each non-unit clauses in P. A corresponding program P_σ for P consists of all unit clauses in P and all non-unit clauses obtained from the non-unit clauses in P by moving the literals selected by σ to the leftmost position in the clause body. Formally, P_σ contains a clause the form

$$A_i \leftarrow A_{i,j_i}, A_{i,1}, \ldots, A_{i,j_i-1}, A_{i,j_i+1}, \ldots, A_{i,n_i}$$

for each non-unit clause

$$A_i \leftarrow A_{i1}, \ldots, A_{i,j_i-1}, A_{i,j_i}, A_{i,j_i+1}, \ldots, A_{i,n_i}$$

in P with A_{ij_i} is selected by σ, since j_i occurs in the i-th position of

$$\sigma = (j_1, j_2, \ldots, j_n)$$

where $1 \leq j_i \leq n_i$, for $i=1,\ldots,n$. ■

Subsequently we show that the problem of finding a minimum link clause program L_{P_σ} for a propositional program P is NP-hard. To prove the claim we reformulate the problem of finding the minimal link clause program as a decision problem, denoted by LCP, and show its NP-completeness by a reduc-

tion of the hitting set problem, denoted by HS. Since HS is NP-complete [25], it follows immediately that our optimization problem is NP-hard.

DEFINITION 3.25 (LCP-Problem):

Input: A propositional program P with $|Link_P|=n$ and a positive integer k with $n \leq k < n \cdot n$.

Question: Is there a selection σ of literals in P such that the number of link clauses in L_{P_σ} is less or equal k? ■

DEFINITION 3.26 (HS or Hitting-Set-Problem):

Input: Collection C of subsets of a finite set S and a positive integer $k \leq |S|$.

Question: Is there a *hitting set*, that is a subset $S' \subseteq S$ with $|S'| \leq k$ such that S' contains at least one element from each subset of C? ■

EXAMPLE 3.27 (Hitting Sets):

C={{a,b,c},{b,d,e},{a,e,f}} has hitting sets

{a,b},...,{b,e},...,{a,b,c},...,{a,b,c,d,e,f}

For k=2 there are the following solutions:

{a,b},{a,d},{a,e},{b,e},{b,f},{c,e} ■

THEOREM 3.28 (NP-Completeness of LCP):

LCP is NP-complete. ■

Proof: First, it is easy to see that LCP∈NP. For, if there is a selection σ of literals in P such that L_{P_σ} contains at most k link clauses, all we need to do is to take σ as a guess and verify that indeed L_{P_σ} has at most k link clauses. For this purpose we compute L_{P_σ} and count the number of link clauses. This can be done by a polynomially bounded algorithm.

Second we show that HS can be reduced to LCP (HS∝LCP). Let the input S, $C=\{C_1,C_2,\ldots,C_n\}$ and an integer k be given. Let c be a unique symbol not occurring in S. For each set $C_i=\{c_{i1},c_{i2},\ldots,c_{in_i}\}\in C$ produce one clause $c\leftarrow c_{i1},c_{i2},\ldots,c_{in_i}$. Obviously, this transformation is polynomially bounded. The resulting program P consists of one single non-recursive procedure defined by n clauses. Consequently, $Link_{P_\sigma}$ and L_{P_σ} are identical and contain at most n link clauses for each σ. The following example demonstrates how to carry out this transformation.

EXAMPLE 3.29 (Hitting Sets and Link Clause Programs):

Suppose C is {{a,b,c},{b,d,e},{a,e,f}} (see Example 3.27). Since p does not occur in any set in C, we construct the following propositional program.

p←a,b,c
p←b,d,e
p←a,e,f ■

If there is a hitting set S' with |S'|≤k, then compute σ by selecting only such predicates in the body of the non-unit clauses which also occur in S'. It follows from the construction of P that each non-unit clause contains a symbol in its body which is an element of S'. Obviously, $Link_{P_\sigma}$ and L_{P_σ} contain at most k link clauses.

On the other hand, if there is selection σ of literals such that the link clause program L_{P_σ} contains at most k link clauses, then the set

S'={c| c is selected by σ}

is a hitting set with |S'|≤k. Thus the answer to the input with respect to HS is 'yes' if and only if the answer to the input with respect to LCP is also 'yes'. ■

One important property of the program constructed in this proof is that some predicate symbols must occur in more than one clause body if $Link_P$ contains less link clauses than P non-unit clauses. One might argue that it is a quite exceptional case in practical applications that one predicate symbol occurs more than once in the clause bodies of a procedure. Therefore, we now consider this problem for programs in which all clauses defining one procedure contain distinct predicate symbols in their bodies. Again we use the corresponding decision problem which we call LCPD (link clause program for programs with predicate-disjoint clause-bodies).

DEFINITION 3.30 (LCPD-Problem):

Input: A propositional program P with $|Link_P|=n$, which does not contain two clauses in the definition of one procedure with the same predicate in their body, and a positive integer $n\leq k<n\cdot n$.

Question: Is there a selection σ for P such that the number of link clauses in L_{P_σ} is less or equal k? ■

THEOREM 3.31 (NP-Completeness of LCPD):

LCPD is NP-complete. ■

Proof: Based on the strong relationship between LCP and LCPD it is straightforward to see that LCPD∈NP.

Again we use the hitting set problem and show that HS∝LCPD. Let the input S, $C=\{C_1,C_2,\ldots,C_n\}$ and an integer k be given. In contrast to the proof for the NP-completeness of LCP we now need n+1 unique symbols not occurring in S. Let us denote them by $\{c,c_1,c_2,\ldots,c_n\}$. First, P contains n clauses

$c\leftarrow c_1$
$c\leftarrow c_2$
...
$c\leftarrow c_n$

Furthermore, for each set $C_i=\{c_{i,1},c_{i,2},\ldots,c_{i,n_i}\}\in C$, P contains the clause

$c_i\leftarrow c_{i,1},c_{i,2},\ldots,c_{i,n_i}$

If k is the upper bound for HS, then we set the upper bound for LCPD to 2·n+k. It is clear that this transformation is polynomially bounded.

The resulting program consists of 2·n clauses where no predicate symbol occurs in two clause bodies belonging to the same procedure. While $Link_P$ contains exactly 2·n link clauses, L_P consists of 3·n clauses in the worst case. If

there is a hitting set S' with $|S'|\leq k$, then compute σ by selecting only those predicates in the body of the last n clauses which also occur in S'. It follows from the construction of P that each of the last n clauses contains a symbol also occurring in S'. Obviously, L_{P_σ} has at most $2{\cdot}n{+}k$ link clauses.

On the other hand, if there is a selection of literals σ for P such that L_{P_σ} contains at most $2{\cdot}n{+}k$ link clauses, then

$$S'=\{c \mid c \text{ is selected by } \sigma\}\setminus\{c_1,c_2,\ldots,c_n\}$$

is a hitting set with $|S'|\leq k$.

Consequently, $|S'|\leq k$ if and only if the number of link clauses in L_{P_σ} is less than or equal $2{\cdot}n{+}k$. Hence the answer to the input with respect to HS is 'yes' if and only if the answer to the input with respect to LCPD is also 'yes'. ■

Both results presented above imply that the problem of finding a selection σ of literals in the clause bodies of P in order to minimize the number of link clauses in L_{P_σ} is intractable. So we have to be content with a polynomially bounded approximation algorithm. Our approach is based on an order of the non-unit clauses which is induced by the predicate dependency graph for P.

DEFINITION 3.32 (Predicate Dependency Graph):

Let P be a definite program. The *predicate dependency graph* D_P is a directed graph G=(V,E) where V contains all predicate symbols occurring in P and $E=\{(p,q) \mid q$ is a predicate symbol occurring in the body of a clause with head p in P$\}$. ■

If P is non-recursive, then the predicate dependency graph is acyclic and the procedures of P can be embedded into a well-founded partial order. This order is constructed out of the predicate dependency graph D_P of P. We state $p>q$ if the transitive closure of D_P contains an arc (p,q), that is, p depends on q. This order is well-founded, since P is non-recursive.

The basic idea of our approach is to scan the non-unit clauses in P bottom-up according to this order and to select that body literal from each clause for which we obtain the minimum transitive closure of all link clauses computed so far. Algorithm 3.33 realizes this approach. The output is a selection σ of literals in P and the corresponding link clause program L_{P_σ}.

ALGORITHM 3.33 (Approximation Algorithm for LCP):

Input: A non-recursive program P.

Output: A selection σ of body literals for the non-unit clauses of P and a link clause program L_{P_σ} for P_σ.

1) Set $L:=\emptyset$ and $Q:=\{C \mid C$ is a non-unit clause in P$\}$.

2) If $Q=\emptyset$, set $L_{P_\sigma}=L$ and stop. Otherwise, let $C_i=A_i\leftarrow A_{i,1},\ldots,A_{i,n_i}\in Q$ be the i-th non-unit clause in P such that A_i is one of the smallest predicate symbols occurring in a clause head in Q w.r.t. to the order induced by D_P, i.e., there is no symbol B occurring in a clause head in Q with $B<A_i$. Let $A_{i,j}$ be the body literal of C_i for which the transitive closure of $L\cup\{link(A_{i,j},A_i)\leftarrow\}$ contains the smallest number of link clauses compared with all other body literals of C_i.

3) Assign j to the i-th position of σ, the transitive closure of L to L, set $Q:=Q\setminus\{C_i\}$ and go to 2). ■

Which time complexity does this algorithm have for propositional programs? Suppose P contains n non-unit clauses, where k is the maximum length of all clause bodies in P. For each clause and each body literal occurring in a clause Algorithm 3.33 computes a transitive closure of maximally n link clauses which takes $O(n^3)$ steps. Consequently the time complexity of Algorithm 3.33 is $O(k{\cdot}n^4)$ and the costs for the optimization are limited by $O(k{\cdot}n)$, since we already need $O(n^3)$ to compute L_P without any optimization.

If we apply Algorithm 3.33 to the program discussed in Example 3.21 then indeed we obtain the optimal selection of literals. However, it is needless to say that there are cases where Algorithm 3.33 produces suboptimal solutions only, and it is easy to construct programs where we obtain the worst case. Sources for wrong selections are multiple dependencies between predicate symbols. A simple form of such dependencies are multiple occurrences of a certain predicate symbol in different clause bodies of the same procedure.

EXAMPLE 3.34 (Worst Case Ratio of Algorithm 3.33):

Consider programs of the following type:

$a \leftarrow a_1,\ldots,a_n$
$a \leftarrow a_2,\ldots,a_n$
$\ldots$
$a \leftarrow a_n$

Suppose Algorithm 3.33 scans the clauses in the order as they are listed. If the leftmost body literal is selected in each clause then we obtain n link clauses even if one would be sufficient. Consequently, the worst case ratio may depend linearly on the number of clauses. ■

Even if Algorithm 3.33 possibly produces solutions which differ linearly from the optimal solution it in practice leads to considerable space savings. In Section 6.3 we present experiments which give an impression of the effectivity of this approach.

Motivated by the strong relationship between the LCP-problem and the HS-problem, which is expressed by the reduction of the latter to the first, one could speculate that it is possible to apply heuristic approaches used to find minimum hitting sets. Johnson [32] shows that the selection of the element with the largest number of occurrences yields a worst case ratio, which can grow with the logarithm of the number of sets. Unfortunately, as the following example shows we cannot obtain the same result applying this heuristic to LCP.

EXAMPLE 3.35 (Worst Case Ratio of Johnson's Heuristic):

Consider programs of the following type

$a_0 \leftarrow a_1,b_1,c_1$
$a_0 \leftarrow a_1,b_1$
$a_1 \leftarrow a_2,b_2,c_2$
$a_1 \leftarrow a_2,b_2$
$\ldots$
$a_{n-1} \leftarrow a_n,b_n,c_n$
$a_{n-1} \leftarrow a_n,b_n$

If we apply Johnson's heuristic and select the leftmost body literal of each clause then we obtain exactly $n \cdot (n+1)/2$ clauses. However, if we always select the second body literal of each clause we have n link clauses only. Consequently, Johnson's heuristic may produce a linear worst case ratio. ■

Note that our approach would lead to a worst case ratio of order O(1) in the previous example. On the other hand, if we apply Johnson's heuristic to Example 3.34 then indeed we obtain the optimal solution. Whereas Algorithm 3.33 attempts to produce as short paths as possible in the link graph, Johnson's heuristic exploits multiple occurrences of a predicate symbol in a procedure to reduce the number of link clauses. Thus an integration of Johnson's approach may lead to a promising improvement of Algorithm 3.33.

Our approach to minimize the number of link clauses is a bottom-up approach, because the clauses in P are scanned bottom-up according to the topological order induced by D_P. It is an interesting question whether it would generally be more effective to use a top-down approach.

EXAMPLE 3.36 (Top-Down Approach for LCP):

Again consider the program containing n clauses

$$p_i \leftarrow p_{i+1},q_i \qquad (i=1,\ldots,n)$$

If we apply a top-down approach, i.e., we start with the clause

$$p_1 \leftarrow p_2,q_1$$

then nothing speaks against a selection of p_2 even if q_1 should be preferred. Moreover, since p_{i+1} and q_i always add the same number of link clauses, this approach possibly produces the worst case by selecting the first body literal in each clause. ■

Whenever heuristics are applied which generally lead to suboptimal solutions only, one has to answer at least two questions. The first one concerns the effectivity and asks for the worst case behaviour of the approach. With Example 3.34 we gave an answer to this question. The subject of the second one is the identification of input classes for which the optimal solution is obtained. In Example 3.34 we showed that multiple dependencies between predicate symbols are one reason for suboptimal solutions. We next investigate whether it suffices to exclude multiple dependencies so that algorithm 3.33 produces optimal solutions.

DEFINITION 3.37 (Unique Path Property for Graphs):

A graph G=(V,E) satisfies the *unique path property*, if there is at most one path between two arbitrary vertices u,v∈ V. Such a graph is called a *unique path graph.* ■

THEOREM 3.38 (Polynomial solvability of LCP):

Let P be a non-recursive, propositional program whose predicate dependency graph satisfies the unique path property. Suppose P contains no procedure such that one predicate symbol occurs in the clause bodies of two different clauses defining it. Then the minimum link clause problem for P is solvable in polynomial time and Algorithm 3.33 produces the optimal solution. ■

Proof: All we have to do is to show that Algorithm 3.33 computes the optimal selection of literals under these restrictions.

Let C be the clause currently selected and suppose π is the procedure C belongs to. Let A be the head of C and B be the body literal of C which adds the minimum number of link clauses to the transitive closure. Since each body literal of C occurs in no other clause defining π, B indeed is the optimal literal in C with respect to π. Now suppose P contains a procedure π' with head A' such that A' depends on A. The unique path property implies that A' does not depend twice on any body literal of a clause defining π. Consequently B is also optimal with respect to π'. Since Algorithm 3.33 scans the procedures bottom-up according to the topological order induced by the predicate dependency graph, B must be optimal with respect to P. ■

Although we mainly discussed the minimization of the number of the link clauses in L_P in the context of propositional programs, our approach as well can be applied to definite programs which are non-recursive. However, since the efficiency of the query evaluation process largely depends on the exploitation of variable bindings, it is not always recommended to give unrestricted preference to the space saving. Link clauses where the left and right side are variable disjoint have a minimum selectivity and negatively influence the efficiency.

EXAMPLE 3.39 (Cousin Relationship):

Consider the following datalog program defining the cousin relationship based on the parent and sibling relationships

cousin($Cousin_1$,$Cousin_2$)←
parent($Parent_1$,$Cousin_1$),
parent($Parent_2$,$Cousin_2$),
sibling($Parent_1$,$Parent_2$)

If we select the last literal in this clause, then we have no data flow between the left and the right side of the corresponding link clause which is

link(sibling($Parent_1$,$Parent_2$),cousin($Cousin_1$,$Cousin_2$))←

This link clause allows no exploitation of variable bindings which possibly produces a significant computational overhead during the evaluation of queries. ■

The previous example shows that Algorithm 3.33, from the viewpoint of efficiency, first of all should select literals sharing variables with the head of the clause which are most frequently bound by queries. Warren [87] describes a simple but effective strategy for planning a query so that the Prolog interpreter can execute it efficiently. Based on a simple cost function, which depends on statistic information about the program, this approach computes an optimal order of literals. Similar extensions of Algorithm 3.33 should allow us to select an optimum literal from the viewpoints of efficiency and space requirement.

In the context of datalog programs the efficiency of goal-directed forward chaining additionally depends on the amount of 'redundant' link clauses in L_P. Redundant link clauses are such link clauses which are an instance of other link clauses in L_P, and are a source of redundant derivations which are highly undesired.

EXAMPLE 3.40 (Ancestor Relationship):

Consider the following Program specifying the ancestor relationship:

ancestor(X,Y)←
    parent(X,Y)
ancestor(X,Z)←
    parent(X,Y),
    ancestor(Y,Z)
parent(abraham,isaac)←

$Link_P$ and L_P both contain the following link clauses:

link(parent(X,Y),ancestor(X,Y))←
link(parent(X,Y),ancestor(X,Z))←

While there is only one proof for ancestor(abraham,isaac) using the standard Prolog interpreter, there are two proofs using the meta-interpreter for goal-directed forward chaining. The reason is that there are two links from the fact to the goal. The first link clause, however, is redundant, because it is subsumed by the second. ■

This example shows that redundant link clauses are a source of redundant derivations. Since we do not alter the declarative semantics of L_P removing redundant link clauses, we can formulate the following heuristic for the optimization of link clause programs:

'Remove redundant link clauses!'

Redundant link clauses, unfortunately, are not the only source of redundant derivations. Redundant derivations are also caused by link clauses which have a common instance. We will discuss this problem and a possible solution to it in Chapter 5.

3.3 Summary

This chapter addressed the problems with the link clauses used by goal-directed forward chaining to focus on relevant clauses. It showed that it is generally undecidable whether or not the transitive closure of the set of link clauses is finite or not. It presented and approach to obtain a possibly more general but finite transitive closure of the link clauses. This allows us to deal with finite sets of link clauses so that automatic decision procedures, whether

or not a clause is relevant, always terminate. This chapter also demonstrated that the number of link clauses in the transitive closure may be reduced by moving appropriate body literals of the non-unit clauses to the leftmost position. Since the problem to find an optimal selection of literals is NP-hard in the context of propositional programs, it presented an approach to minimize the number of link clauses. For propositional programs the space saving may be of order O(n) where n is the number of non-unit clauses in the underlying program.

Chapter 4

GDFC-RESOLUTION

The main topic of this chapter is the introduction of a linear resolution strategy for goal-directed forward chaining. The calculus, which is denoted by GDFC-resolution, is based on the meta-interpreter for goal-directed forward chaining presented in 2.1. This approach has several advantages. First it enables us to define the semantics of goal-directed forward chaining and second it facilitates the efficiency comparison of SLD- and GDFC-resolution. In 4.1 we introduce GDFC-resolution. In 4.2 we show that GDFC-resolution is sound and complete for definite logic programs and consider the question, how to construct a GDFC-refutation out of a given SLD-refutation and vice versa.

4.1 The Resolution Strategy

If we define GDFC-resolution taking the meta-interpreter for goal-directed forward chaining as a basis, we first need the concept of *subgoal-goal pairs* to represent situations in which an atom of the form subgoal(A,B) is selected, i.e., we prove that A is a subgoal for B.

DEFINITION 4.1 (Subgoal-Goal Pair):

Let A and B are be arbitrary atoms. Then we denote a pair <A,B> as a *subgoal-goal pair*. A is called the subgoal of the goal B. If A and B have the same predicate symbol <A,B> is a *recursive subgoal-goal pair*. ■

DEFINITION 4.2 (GDFC-Goal):

A *GDFC-goal* is a goal $\leftarrow A_1,\ldots,A_n$ ($n\geq 1$) such that A_i is either an atom or a subgoal-goal pair, for $i=1,\ldots,n$. ■

The following definition introduces GDFC-resolution. To focus on relevant clauses GDFC-resolution uses L_P^{fin} instead of L_P. The advantage of this approach is that every check whether a clause is possibly relevant terminates so that every link goal has a finite number of solutions only. This allows us to deal with a finite number of one-step derivations for each particular goal, i.e., with GDFC-trees (see below) in which each node has finitely many successors only.

DEFINITION 4.3 (GDFC-Resolution):

Let P be a definite program, L_P^{fin} be a finite link clause program for P, and G be a GDFC-goal $\leftarrow A_1,\ldots,A_i,\ldots,A_n$. Then G' is derived from G via *GDFC-resolution* w.r.t. L_P^{fin} using a clause $C\in P$ and a substitution θ if the following conditions hold:

1) A_i is a selected element (atom or a subgoal-goal pair) in G.

2) Suppose A_i is an atom and C is a unit clause $B\leftarrow$.

   a) If A_i and B are unifiable with mgu θ, then G' is the goal

   $\leftarrow(A_1,\ldots,A_{i-1},A_{i+1},\ldots,A_n)\theta.$ (Rule 1)

   b) If θ is an answer for $L_P^{fin}\cup\{\leftarrow link(B,A_i)\}$, then G' is the goal

   $\leftarrow(A_1,\ldots,A_{i-1},<B,A_i>,A_{i+1},\ldots,A_n)\theta.$ (Rule 2)

3) Suppose A_i is a subgoal-goal pair $<A,A'>$. Let C be a non-unit clause $B\leftarrow B_1,\ldots,B_k$ such that A and B_1 are unifiable with mgu θ_1.

   a) If $B\theta_1$ and A' are unifiable with mgu θ, then G' is the goal

   $\leftarrow(A_1,\ldots,A_{i-1},B_2,\ldots,B_k,A_{i+1},\ldots,A_n)\theta.$ (Rule 3)

   b) If θ is an answer for $L_P^{fin}\cup\{\leftarrow link(B\theta_1,A')\}$, then G' is the goal

   $\leftarrow(A_1,\ldots,A_{i-1},B_2,\ldots,B_k,<B,A'>,A_{i+1},\ldots,A_n)\theta.$ (Rule 4)

G' is called the *GDFC-resolvent*. ■

Note that the conditions for an application of Rule 1 and Rule 2 respectively Rule 3 and Rule 4 are not mutually exclusive and that there are situations in which we can apply either Rule 1 or Rule 2 respectively either Rule 3 or Rule 4.

DEFINITION 4.4 (GDFC-Derivation):

Let P be a definite program and G a definite goal. A *GDFC-derivation* of $P\cup\{G\}$ w.r.t. L_P^{fin} consists of a possibly infinite sequence $G=G_0,G_1,\ldots$ of definite forward chaining goals, a sequence $C_1,C_2,\ldots$ of variants of program clauses of P, and a sequence $\theta_1,\theta_2,\ldots$ of substitutions such that each G_{i+1} is derived from G_i using C_{i+1} and θ_{i+1}. ■

DEFINITION 4.5 (GDFC-Refutation):

A *GDFC-refutation* of $P\cup\{G\}$ w.r.t. L_P^{fin} is a finite GDFC-derivation with the empty clause □ as the last goal in the derivation. ■

DEFINITION 4.6 (Computed Answer):

Let P be a definite program and G a definite goal. A *computed answer* θ for $P\cup\{G\}$ w.r.t. L_P^{fin} is the substitution obtained by restricting the composition $\theta_1\ldots\theta_n$ to the variables of G, where $\theta_1,\ldots,\theta_n$ is the sequence of substitutions used in a GDFC-Refutation of $P\cup\{G\}$ w.r.t. L_P^{fin}. ■

DEFINITION 4.7 (Failed GDFC-Derivation):

A *failed GDFC-derivation* is one that ends in a non-empty goal with the property that none of the rules 1-4 can be applied to the selected element. ■

Our next goal is to show that there is a strong relationship between the meta-interpreter for goal directed forward chaining and GDFC-resolution.

LEMMA 4.8 (Simulation-Lemma):

Let P a definite program, L_P^{fin} a finite link clause program for P and G be a GDFC-goal $\leftarrow A_1, \ldots, A_i, \ldots, A_n$. Suppose A_i is selected element in G and G' is derived from G via *GDFC-resolution* w.r.t. L_P^{fin} using a clause $C \in P$ and a substitution θ. Then we can simulate this derivation using the meta-interpreter for goal-directed forward chaining (Program 2.1). ■

Proof: All we have to do is to show that, for each derivation rule in our calculus, there is a corresponding clause in the meta-interpreter. It should be noted that, even if it is not explicitly expressed by the code, the mgu's computed at every reduction step the meta-interpreter performs are simultaneously applied to all literals of the actual goal.

Suppose the selected element A_i is an atom and we apply Rule 1 using a fact B← in P with the property that B and A_i are unifiable. We derive G' from G by removing A_i and applying $\theta = mgu(B, A_i)$ to the resulting goal. Consider the third clause which is

```
gdfc_solve(B)←
    clause(B,true)
```

This clause means that we can drop the selected atom if there is a unifying fact. Thus, it directly corresponds to Rule 1.

Now suppose A_i is an atom and we apply Rule 2 using a fact B← in P for which there is an answer θ for $L_P^{fin} \cup \{\leftarrow link(B, A_i)\}$. By means of Rule 2 we obtain G' from G by replacing A_i by the subgoal-goal pair $<B, A_i>$ in G and applying θ to the resulting goal. The fourth clause of the meta-interpreter is

```
gdfc_solve(A)←
    clause(B,true),
    link(B,A),
    subgoal(B,A)
```

It allows us to replace a selected goal A by a goal subgoal(B,A), if B is a possibly relevant fact for A. Consequently, this clause implements Rule 2.

Next suppose A_i is a subgoal-goal pair <A,A'> and P contains a non-unit clause $B \leftarrow B_1, \ldots, B_k$ such that A and B_1 are unifiable with mgu θ_1, and $B\theta_1$ and A' are unifiable with mgu θ. In this case we obtain G' from G applying Rule 3: we replace <A,A'> by $B_2, \ldots, B_k$ and apply θ to the resulting goal. Now consider the fifth clause which has the form

```
subgoal(A,B)←
    clause(B,(A,Body)),
    gdfc_solve(Body)
```

By means of this clause we can replace a goal subgoal(A,B) by the remaining literals of a clause after the head has been reduced with B and the leftmost body literal has been reduced with A. Clearly, this clause implements Rule 3.

Finally suppose that A_i is a subgoal-goal pair <A,A'>, P contains a non-unit clause $B \leftarrow B_1, \ldots, B_k$ such that A and B_1 are unifiable with mgu θ_1, and there is an answer θ for $L_P^{fin} \cup \{\leftarrow link(B\theta_1, A')\}$. In this case we apply Rule 4 and derive G' from G replacing the subgoal-goal pair by $B_2, \ldots, B_k$,<B,A'> and applying θ to the resulting goal. The last clause of the meta-interpreter is

```
subgoal(A,B)←
    clause(C,(A,Body)),
    link(C,B),
    gdfc_solve(Body),
    subgoal(C,B)
```

We apply this clause whenever the head of a non-unit clause, the leftmost body literal of which is unifiable with the already deduced subgoal A, is possibly relevant to solve B. Then, additionally to the body literals starting with the second, the atom subgoal(C,B) is added, where C is the head of the selected clause. Consequently this clause realizes Rule 4.

Since the unifications we compute in each derivation step are the same as those computed in the corresponding computation of the meta-interpreter, the derived goals must be variants. ■

The previous lemma expresses the relationship between one-step derivations of GDFC-resolution and Program 2.1. The following theorem generalizes this result to arbitrarily long derivations.

THEOREM 4.9 (Relationship between Program 2.1 and GDFC-Resolution):

Let P be a definite program and G be a definite goal. Then every GDFC-derivation of $P \cup \{G\}$ w.r.t. L_P^{fin} can be simulated by a corresponding computation of Program 2.1. ■

Proof: The proof for this theorem is straightforward. For each derivation with length n>1 we can construct the corresponding computations of Program 2.1 by repeated applications of Lemma 4.8. ■

As well as for SLD-resolution we can define GDFC-trees.

DEFINITION 4.10 (GDFC-Tree):

Let P be a definite program and G be a definite goal. A *GDFC-tree* for $P \cup \{G\}$ w.r.t. L_P^{fin} is a tree satisfying the following conditions:

1) Each node of the tree is a (possibly empty) GDFC-goal.

2) The root node is G.

3) Let $G' = \leftarrow A_1, \ldots, A_i, \ldots, A_n$ ($n \geq 1$) be a node in the tree and suppose A_i is the selected element. Then, for each GDFC-resolvent G" derived from G' using the four derivation rules, G' has a successor node G".

4) Nodes that are the empty clause have no children. ■

Each GDFC-derivation corresponds to a branch in the GDFC-tree which comes from the same selection function. Whereas refutations correspond to branches ending with the empty clause, failed derivations have branches with a non-empty goal as last node. Infinite derivations, however, correspond to infinite branches.

EXAMPLE 4.11 (GDFC-Tree):

Let us again consider the program specifying the reflexive and transitive closure of a graph:

p(X,X)← (C1)
p(X,Z)←e(X,Y),p(Y,Z) (C2)
e(a,b)← (C3)

L_P^{fin} contains only one link clause, namely

link(e(X,Y),p(X,Z))← (L1)

Figure 4.12 shows the GDFC-tree for $G = \leftarrow p(X,b)$ w.r.t. L_P^{fin} coming from the computation rule which always selects the leftmost element. A label 'Rule i (C, L)' of an arc is interpreted as follows: G' can be derived from G applying Rule i with input clause C and link clause L. Whenever Rule 1 resp. Rule 3 is applied, no link clause is needed. ■

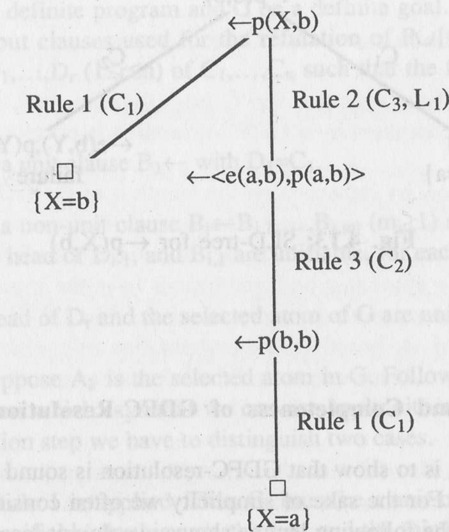

Fig. 4.12: GDFC-tree for ←p(X,b)

Note that this tree is smaller than the SLD-tree we obtain using the left-first selection function (see Figure 4.13). The SLD-tree additionally contains one failure branch. Hence, in a situation in which the complete tree has to be traversed GDFC-resolution would be more efficient than SLD-resolution. A

detailed discussion of efficiency aspects is the main subject of the two following chapters.

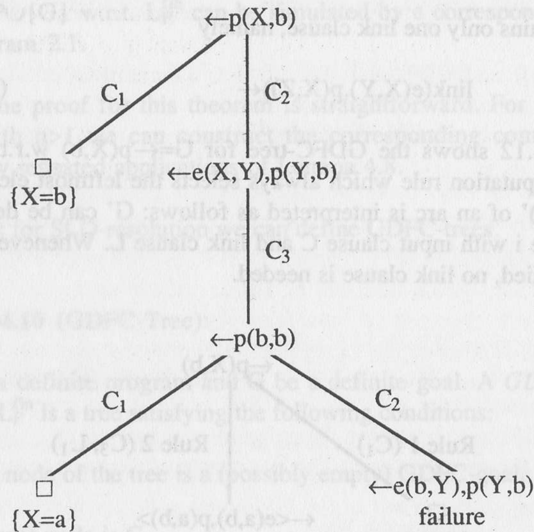

Fig. 4.13: SLD-tree for $\leftarrow p(X,b)$

4.2 Soundness and Completeness of GDFC-Resolution

Our next goal is to show that GDFC-resolution is sound and complete for definite programs. For the sake of simplicity we often consider a fixed selection rule only. As the following lemma shows, we do not lose any solutions by this restriction, since, independently from the computation rule, we always find a refutation if $P\cup\{G\}$ is unsatisfiable. We obtain this result by an extension of the switching lemma for definite goals [46] to GDFC-goals.

LEMMA 4.14 (Independence of the Computation Rule):

Let P be a definite program and G be a definite goal. Suppose there is a GDFC-refutation of $P\cup\{G\}$ w.r.t. L_P^{fin} with computed answer θ. Then, for any computation rule R, there exists a GDFC-refutation of $P\cup\{G\}$ w.r.t. L_P^{fin} via R with computed answer θ' such that $G\theta'$ is a variant of $G\theta$. ■

The following lemma illustrates the structure of GDFC-refutations. The proofs of soundness and completeness of GDFC-resolution are mainly based on this structure.

LEMMA 4.15 (Structure of GDFC-Refutations):

Let P be a definite program and G be a definite goal. Let $C_1,\ldots,C_n$ be the sequence of input clauses used for the refutation of $P\cup\{G\}$. Then there is a subsequence $D_1,\ldots,D_r$ $(1\leq r\leq n)$ of $C_1,\ldots,C_n$ such that the following conditions hold:

1) D_1 is a unit clause $B_1\leftarrow$ with $D_1=C_1$,

2) D_i is a non-unit clause $B_i\leftarrow B_{i,1},\ldots,B_{i,m_i}$ $(m_i\geq 1)$ such that B_{i-1}, which is the head of D_{i-1}, and $B_{i,1}$ are unifiable, for each $i\in\{2,\ldots,r\}$, and

3) the head of D_r and the selected atom of G are unifiable.

Proof: Suppose A_k is the selected atom in G. Following the definition of GDFC-resolution which says that we can use apply either Rule 1 or Rule 2 in the first derivation step we have to distinguish two cases.

Suppose Rule 1 is applied. Thus C_1 must be a unit clause $B_1\leftarrow$ such that B_1 and A_k are unifiable. In this case the first and the third condition are simultaneously met and we set $r=1$ and $D_1=C_1$.

Next suppose Rule 2 is applied. Thus C_1 is a unit clause $B_1\leftarrow$ which is possibly relevant for A_k so that A_k is replaced by a subgoal-goal pair of the form $\langle B_1,A_k\rangle\sigma$. Clearly C_1 satisfies the first condition and we set $D_1=C_1$. Since $P\cup\{G\}$ has a refutation, it must be possible to eliminate this subgoal-goal pair. Definition 4.3 implies that this can only be done by a finite number of applications of Rule 4 and one application of Rule 3.

Suppose we need $m \geq 0$ applications of Rule 4 which always produce subgoal-goal pairs of the form $<A,A_k>\theta$ where θ is the composition of the substitutions computed so far and $A\theta$ is an instance of the head of the input clause used do generate it. If such a subgoal-goal pair is selected, then the leftmost body literal of the next input clause must be unifiable with $A\theta$. Consequently both input clauses satisfy the second condition. We now set r to m+2. If $D_2,\ldots,D_{m+1}$ are the input clauses used for these m derivations then $D_2,\ldots,D_{r-1}$ satisfy the second condition.

Definition 4.3 implies that in the last step, when Rule 3 is applied, the head of the (m+2)-th input clause additionally must be unifiable with $A_k\gamma$ where γ is the composition of the substitutions computed so far. Let D_r be the clause used in this step. Clearly D_r satisfies the second and the third condition. Consequently, $D_1,\ldots,D_r$ is a sequence of the desired form. ■

THEOREM 4.16 (Soundness of GDFC-Resolution):

Let P be a definite program and G be a definite goal. Each answer for $P \cup \{G\}$ w.r.t. L_P^{fin} computed by a GDFC-refutation is correct. ■

Proof: We show by induction on the length n of the GDFC-refutation of $P \cup \{G\}$ w.r.t. L_P^{fin} that, for each computed answer θ, there is a corresponding SLD-refutation of $P \cup \{G\}$ which computes an answer γ such that $G\gamma$ subsumes $G\theta$. Furthermore we show that both refutations have the same length and that the same input clauses are used to compute them. The GDFC-refutations considered in this proof are based on the computation rule which always selects the subgoal-goal pair, if one exists, and the leftmost atom otherwise. An application of Lemma 4.14 yields the desired result.

Suppose n=1. Then G must be a single literal goal $\leftarrow A$. Lemma 4.15 implies that P must contain a unit clause $B\leftarrow$ such that B and A are unifiable with mgu θ so that Rule 1 can be applied. Consequently we can also use $B\leftarrow$ in a corresponding one-step SLD-refutation of $P \cup \{\leftarrow A\}$. If γ is the answer computed by the SLD-refutation, then $G\theta$ and $G\gamma$ are variants.

Next suppose the result holds for all GDFC-refutations with maximal length n-1, and G is $\leftarrow A_1,A_2,\ldots,A_m$ $(m \geq 1)$. Clearly A_1 is selected first, since G contains no subgoal-goal pair.

It follows from Lemma 4.15 that P must contain $r \geq 1$ clauses $D_1,D_2,\ldots,D_r$ such that D_1 is a unit clause of the form $B_1\leftarrow$, the head of D_i is unifiable with the first body literal of D_{i+1}, for i=1,…,r-1, and the head of D_r is unifiable with A_1, $r \in \{1,\ldots,k\}$.

We distinguish two cases. If r=1 then P contains a unit clause so that we apply Rule 1 to derive G_1 which is

$$\leftarrow(A_2,\ldots,A_m)\theta_1$$

Clearly we can use the same unit clause as input clause in an SLD-refutation which yields a variant of G_1. It follows from the induction hypothesis, that each answer for $P \cup \{G_1\}$ computed by a GDFC-refutation is correct.

On the other hand, if r>1 then Lemma 4.15 implies that D_1 is a unit clause $B_1\leftarrow$ for which there is an answer θ_1 for $L_P^{fin} \cup \{\leftarrow link(B_1,A_1)\}$ so that we apply Rule 2 producing G_1 which is

$$\leftarrow(<B_1,A_1>,A_2,\ldots,A_m)\theta_1$$

Since the computation rule always selects the subgoal-goal pair, we next use the non-unit clauses $D_2,\ldots,D_r$ as input clauses. Suppose each D_i has the form $B_i\leftarrow B_{i,1},B_{i,2},\ldots,B_{i,m_i}$, for i=2,…,r. Thus, if G_i $(1 \leq i \leq r-2)$ is

$$\leftarrow(B_{2,2},\ldots,B_{2,m_2},\ldots,B_{i,2},\ldots,B_{i,m_i},<B_i,A_1>,A_2,\ldots,A_m)\theta_i$$

we use D_{i+1} as input clause and apply Rule 4 to derive G_{i+1} which is

$$\leftarrow(B_{2,2},\ldots,B_{i,m_i},B_{i+1,2},\ldots,B_{i+1,m_{i+1}},<B_{i+1},A_1>,A_2,\ldots,A_m)\theta_{i+1}$$

Finally, if the actual goal, G_{r-1}, is

$$\leftarrow(B_{2,2},\ldots,B_{2,m_2},\ldots,B_{r-1,2},\ldots,B_{r-1,m_{r-1}},<B_{r-1},A_1>,A_2,\ldots,A_m)\theta_{r-1}$$

we use D_r applying Rule 3 to derive G_r which has the form

$$\leftarrow(B_{2,2},\ldots,B_{2,m_2},\ldots,B_{r-1,2},\ldots,B_{r-1,m_{r-1}},B_{r,2},\ldots,B_{r,m_r},A_2,\ldots,A_m)\theta_r$$

We now will construct a corresponding SLD-derivation which is based on the left-first computation rule. Since the head literal of D_r is unifiable with A_1 by mgu γ_1, we can use D_r as first input clause. Furthermore, the structure of

the GDFC-derivation and the construction of $D_1,\ldots,D_r$ imply that we can use $D_{r-1},\ldots,D_1$ as the next input clauses. Consequently the r-th goal G'_r is

$$\leftarrow(B_{2,2},\ldots,B_{2,m_2},B_{3,2},\ldots,B_{3,m_3},\ldots,B_{r,2},\ldots,B_{r,m_r},A_2,\ldots,A_m)\gamma_r$$

where γ_r is the composition of the substitutions computed so far.

Since the answers for the link goals used in the first r steps of the GDFC-derivation possibly produce additional substitutions, G_r must be an instance of G'_r.

It follows from the induction hypothesis that each answer computed by a GDFC-refutation of $P\cup\{G_r\}$ is sound. Thus, each answer computed by a GDFC-refutation of $P\cup\{G\}$ must be sound too. It furthermore follows that both refutations have the same length and that the same input clauses are needed to compute them. ■

Additionally to the soundness result a further desirable property of refutation procedures is completeness.

THEOREM 4.17 (Completeness of GDFC-Resolution):

Let P be a definite program and G be a definite goal. Then, for each correct answer θ for $P\cup\{G\}$, there is an answer γ computed by a GDFC-refutation of $P\cup\{G\}$ w.r.t. L_P^{fin} and a substitution σ such that $\theta=\gamma\sigma$. ■

Proof: By induction on the length n of the refutation we show that there is an answer γ computed by a GDFC-refutation of $P\cup\{G\}$ w.r.t. L_P^{fin} and a substitution σ for each answer θ computed by an SLD-refutation of $P\cup\{G\}$ such that $\theta=\gamma\sigma$. This GDFC-refutation needs the same number of steps and uses the same input clauses as the SLD-refutation. Throughout this proof we assume that the computation rule used for the SLD-refutation always selects the left-most literal.

Suppose n=1. In this case G must be a single literal goal $\leftarrow A$ and P must contain a fact $B\leftarrow$ such that A and B are unifiable with mgu θ. Clearly we can apply Rule 1 with $B\leftarrow$ as input clause in a GDFC-refutation of $P\cup\{G\}$. Consequently, if γ is the answer computed by the GDFC-refutation then θ and γ are variants.

Next suppose the result holds for all SLD-refutations with maximal length n-1. Suppose G is $\leftarrow A_1,A_2,\ldots,A_m$ $(1\leq m\leq n)$, $G=G_0,\ldots,G_n=\square$ is the sequence of goals, and $C_1,\ldots,C_n$ is the sequence of input clauses used for an SLD-refutation of $P\cup\{G\}$ with length n.

Let us consider the elimination of A_1, that is the SLD-refutation of $P\cup\{\leftarrow A_1\}$ with length $k\in\{1,\ldots,n\}$. Suppose C_r $(1\leq r\leq k)$ is the first unit clause in the sequence of input clauses, i.e., $r=\min\{j \mid C_j \text{ is a unit clause}\}$. Thus, G_r is the first goal in the sequence of goals which is shorter than its predecessor. Clearly $C_1,\ldots,C_r$ satisfy the following conditions:

1) The head of C_1 and A_1 are unifiable with mgu θ_1.

2) C_i is a non-unit clause $B_i\leftarrow B_{i,1},\ldots,B_{i,m_i}$ $(m_i\geq 1)$ such that $B_{i,1}$ and B_{i+1}, which is the head of C_{i+1}, are unifiable with mgu θ_{i+1}, for each $i\in\{1,\ldots,r-1\}$.

3) C_r is a unit clause.

We distinguish two cases. First, if r=1 then C_1 is a unit clause $B_1\leftarrow$ such that B_1 and A_1 are unifiable with mgu θ_1. Hence G_r is

$$\leftarrow(A_2,\ldots,A_m)\theta_1$$

Second, if r>1 then G_r is

$$\leftarrow(B_{r-1,2},\ldots,B_{r-1,m_{r-1}},B_{r-2,2},\ldots,B_{r-2,m_{r-2}},\ldots,B_{1,2},\ldots,B_{1,m_1},A_2,\ldots,A_m)\theta_1\ldots\theta_r$$

We now will construct a GDFC-derivation which uses the same number of steps and the same input clauses to derive a variant of G_r. For the sake of simplicity we use the computation rule which always selects the subgoal-goal pair or the leftmost literal if no subgoal-goal pair exists.

If r=1 we also use C_1 applying Rule 1 to derive

$$\leftarrow(A_2,\ldots,A_m)\gamma_1$$

Clearly γ_1 and θ_1 are variants.

Now suppose r>1. From the construction of $C_1,\ldots,C_r$ and the definition of L_P^{fin} it follows, that there must be r-1 link clauses in L_P^{fin} which subsume those

computed from $C_1,\ldots,C_{r-1}$. Consequently, there are answer substitutions λ_i for $L_P^{fin}\cup\{\leftarrow link(B_i,A_1)\}$ (i=2,...,r) which do not produce unnecessary substitutions. This means that we obtain no superfluous variable-bindings using these answers.

Starting with G we can use use $C_r=B_r\leftarrow$ and apply Rule 2 to derive the GDFC-goal

$$\leftarrow(<B_r,A_1>,A_2,\ldots,A_m)\gamma_1$$

with $\gamma_1=\lambda_r$. Because of the selection function we subsequently always replace the subgoal-goal pair. Consequently, if the actual goal is

$$\leftarrow(B_{r-1,2},\ldots,B_{r-1,m_{r-1}},\ldots,B_{i,2},\ldots,B_{i,m_i},<B_i,A_1>,A_2,\ldots,A_m)\gamma_1\ldots\gamma_{r-i+1}$$

for i=r,...,3, we can apply Rule 4 using C_{i-1} as input clause and λ_{i-1} to derive

$$\leftarrow(B_{r-1,2},\ldots,B_{r-1,m_{r-1}},\ldots,B_{i-1,2},\ldots,B_{i-1,m_{i-1}},<B_{i-1},A_1>,A_2,\ldots,A_m)\gamma_1\ldots\gamma_{r-i+2}$$

Finally, if i=2 and the actual goal is

$$\leftarrow(B_{r-1,2},\ldots,B_{r-1,m_{r-1}},\ldots,B_{2,2},\ldots,B_{2,m_2},<B_2,A_1>,A_2,\ldots,A_m)\gamma_1\ldots\gamma_{r-1}$$

we can apply Rule 3 using C_1 to derive G_r' which is

$$\leftarrow(B_{r-1,2},\ldots,B_{r-1,m_{r-1}},\ldots,B_{1,2},\ldots,B_{1,m_1},A_2,\ldots,A_m)\gamma_1\ldots\gamma_r$$

Since the substitutions obtained as answers for the link goals produce no unnecessary variable bindings, G_r' must be a variant G_r. It follows from the induction hypothesis that each answer for $P\cup\{G_r\}$ computed using SLD-resolution can also be computed using GDFC-resolution. Consequently, for each answer θ computed by an SLD-refutation of $P\cup\{G\}$ there is an answer γ computed by a GDFC-refutation of $P\cup\{G\}$ w.r.t. L_P^{fin} and a substitution σ such that $\theta=\gamma\sigma$. Furthermore, both refutations have the same length and the same input clauses are used to compute them. ■

Both results presented above are summarized by the following theorem.

THEOREM 4.18 (Soundness and Completeness of GDFC-Resolution):

GDFC-resolution is sound and complete for definite logic programs. ■

In Chapter 3 we presented an approach to obtain a finite number of link clauses by renaming variables in $Link_P$. In this context the question came up whether or not this operation, which leads to more general link clauses in the transitive closure, is admissible. The previous theorem shows that indeed the generalization of the link clauses has no influence on the operational semantics of GDFC-resolution.

In the remaining part of this dissertation we often speak about corresponding refutations which, in contrast to the GDFC-derivations constructed above, are computed via the left-first computation rule. The length and the number of such corresponding refutations are important in the following chapter where we compare the efficiency of GDFC-resolution with that of SLD-resolution. To put the notion of corresponding refutations onto a more formal level, we subsequently define two functions mapping a sequence of input clauses used for a GDFC- resp. SLD-refutation to the sequence of input clauses used for the corresponding SLD- resp. GDFC-refutation.

Let us first consider how an SLD-refutation can be computed out of an existing GDFC-refutation. Suppose G is $\leftarrow A_1,A_2,\ldots,A_m$ and $D_1,\ldots,D_r$ are the r clauses satisfying the conditions defined in Lemma 4.15 concerning the structure of GDFC-derivations. Suppose each D_i, $1\leq i\leq r$, has the form $B_i\leftarrow B_{i,1},\ldots,B_{i,m_i}$ and the computation rule always selects the leftmost element. Clearly we first use D_1 as input clause in the GDFC-derivation to derive G_1 which is

$$\leftarrow(<B_1,A_1>,A_2,\ldots,A_m)\theta_1$$

After that D_2 is used as input clause which results in the goal

$$\leftarrow(B_{2,2},\ldots,B_{2,m_2},<B_2,A_1>,A_2,\ldots,A_m)\theta_2$$

Subsequently the literals left from the subgoal-goal pair are eliminated, before we replace the subgoal-goal pair using D_3. Suppose Q_2 is the sequence of input clauses used to eliminate $(B_{2,2},\ldots,B_{2,m_2})\theta_2$. Thus $G_{|Q_2|+2}$ is

$$\leftarrow(<B_2,A_1>,A_2,\ldots,A_m)\theta_{|Q_2|+2}$$

where $\theta_{|Q_2|+2}$ denotes the composition of the substitutions computed so far. Generally, D_{i+1} can be used as input clause whenever $<B_i,A_1>$ is selected $(1 \le i \le r-1)$.

For each $i \in \{2,\ldots,r-1\}$, we denote the possibly empty sequences of input clauses used between D_i and D_{i+1} by Q_i. Furthermore, let Q_r be the sequence of input clauses used between D_r and the input clause used at the moment where A_2 is selected. Hence, Q_i contains the sequence of input clauses used to eliminate the elements coming from $B_{i,2},\ldots,B_{i,m_i}$, for $i=2,\ldots,r$. Suppose Σ_1 is 0 and Σ_i is $|Q_2|+\ldots+|Q_i|$, for $i=2,\ldots,r$.

Thus, if G_{Σ_i+i} is

$$\leftarrow(<B_i,A_1>,A_2,\ldots,A_m)\theta_{\Sigma_i+i}$$

for $i=1,\ldots,r-2$, we use D_{i+1} as input clause applying Rule 4 to derive G_{Σ_i+i+1} which is

$$\leftarrow(B_{i+1,2},\ldots,B_{i+1,m_{i+1}},<B_{i+1},A_1>,A_2,\ldots,A_m)\theta_{\Sigma_i+i+1}$$

However, if $i=r-1$, i.e. the actual goal is $G_{\Sigma(r-1)+r-1}$, we apply Rule 3 using D_r to derive $G_{\Sigma(r-1)+r}$ which has the form

$$\leftarrow(B_{r,2},\ldots,B_{r,m_r},A_2,\ldots,A_m)\theta_{\Sigma(r-1)+r}$$

Whenever $1 \le i \le r-2$ and $G_{\Sigma i+i+1}$ is

$$\leftarrow(B_{i+1,2},\ldots,B_{i+1,m_{i+1}},<B_{i+1},A_1>,A_2,\ldots,A_m)\theta_{\Sigma i+i+1}$$

we use the clauses contained in Q_{i+1} to eliminate the elements coming from $B_{i+1,2},\ldots,B_{i+1,m_{i+1}}$ so that $G_{\Sigma(i+1)+i+1}$ is

$$\leftarrow(<B_{i+1},A_1>,A_2,\ldots,A_m)\theta_{\Sigma(i+1)+i+1}$$

However, if $i=r-1$ and $G_{\Sigma i+i+1}=G_{\Sigma(r-1)+r}$ is

$$\leftarrow(B_{r,2},\ldots,B_{r,m_r},A_2,\ldots,A_m)\theta_{\Sigma(r-1)+r}$$

we use the sequence Q_r so that $G_{\Sigma r+r}$ is

$$\leftarrow(A_2,\ldots,A_m)\theta_{\Sigma r+r}$$

Let $Q_{i,j}$ represent input clause j in sequence Q_i. Then by our construction the sequence of input clauses is

$$D_1,D_2,Q_{2,1},\ldots,Q_{2,|Q_2|},D_3,Q_{3,1},\ldots,Q_{3,|Q_3|},\ldots,D_r,Q_{r,1},\ldots,Q_{r,|Q_r|}$$

Figure 4.19 gives a more detailed illustration for this GDFC-derivation.

Input clauses	Derived Goals
	$\leftarrow A_1,A_2,\ldots,A_m$
$D_1=C_1$	$\leftarrow(<B_1,A_1>,A_2,\ldots,A_m)\theta_1$
$D_2=C_2$	$\leftarrow(B_{2,2},\ldots,B_{2,m2},<B_2,A_1>,A_2,\ldots,A_m)\theta_2$
$Q_{2,1}$	⋮
⋮	⋮
$Q_{2,\|Q2\|}$	$\leftarrow(<B_2,A_1>,A_2,\ldots,A_m)\theta_{\Sigma 2+2}$
D_3	$\leftarrow(B_{3,2},\ldots,B_{3,m3},<B_3,A_1>,A_2,\ldots,A_m)\theta_{\Sigma 2+3}$
$Q_{3,1}$	⋮
⋮	⋮
D_{r-1}	$\leftarrow(B_{r-1,2},\ldots,B_{r-1,m_{r-1}},<B_{r-1},A_1>A_2,\ldots,A_m)\theta_{\Sigma(r-2)+r-1}$
$Q_{r-1,1}$	⋮
⋮	⋮
$Q_{r-1,\|Q_{r-1}\|}$	$\leftarrow(<B_{r-1},A_1>,A_2,\ldots,A_m)\theta_{\Sigma(r-1)+r-1}$
D_r	$\leftarrow(B_{r,2},\ldots,B_{r,m_r},A_2,\ldots,A_m)\theta_{\Sigma(r-1)+r}$
$Q_{r,1}$	⋮
⋮	⋮
$Q_{r,\|Q_r\|}=C_k$	$\leftarrow(A_2,\ldots,A_m)\theta_{\Sigma r+r}$

Fig. 4.19: A GDFC-derivation

The proof of Theorem 4.16 implies, that we can use a sequence R_i of input clauses, which is a permutation of the clauses contained in Q_i, to compute a corresponding SLD-refutation of $P \cup \{\leftarrow(B_{i,2},\ldots,B_{i,m_i})\theta_{\Sigma(i-1)+i}\}$, for each $i \in \{2,\ldots,r\}$. Consequently, the concatenation of the sequences $R_2,\ldots,R_r$ gives us the order, in which the input clauses have to be used to eliminate the literals of the goal

$$\leftarrow(B_{2,2},\ldots,B_{2,m2},B_{3,2},\ldots,B_{3,m3},\ldots,B_{r,2},\ldots,B_{r,m_r})\gamma_r$$

Thus, the corresponding SLD-derivation via the left-first computation rule, which is illustrated in Figure 4.20, uses the following sequence of input clauses

$$D_r, D_{r-1}, \ldots, D_1, R_{2,1}, \ldots R_{2,|Q_2|}, R_{3,1}, \ldots, R_{r,|Q_r|}$$

Input clauses	Derived Goals
	$\leftarrow A_1, A_2, \ldots, A_m$
D_r	$\leftarrow (B_{r,1}, B_{r,2}, \ldots, B_{r,m_r}, A_2, \ldots, A_m)\gamma_1$
D_{r-1}	$\leftarrow (B_{r-1,2}, \ldots, B_{r-1,m_{r-1}}, B_{r,2}, \ldots, B_{r,m_r}, A_2, \ldots, A_m)\gamma_2$
$\vdots$	$\vdots$
D_2	$\leftarrow (B_{2,1}, B_{2,2}, \ldots, B_{2,m_2}, \ldots, B_{r,2}, \ldots, B_{r,m_r}, A_2, \ldots, A_m)\gamma_{r-1}$
D_1	$\leftarrow (B_{2,2}, \ldots, B_{2,m_2}, \ldots, B_{r,2}, \ldots, B_{r,m_r}, A_2, \ldots, A_m)\gamma_r$
$R_{2,1}$	$\vdots$
$\vdots$	$\vdots$
$R_{2,|R_2|}$	$\leftarrow (B_{3,2}, \ldots, B_{3,m_3}, \ldots, B_{r,2}, \ldots, B_{r,m_r}, A_2, \ldots, A_m)\gamma_{\Sigma_2+r}$
$\vdots$	$\vdots$
$R_{r-1,|R_{r-1}|}$	$\leftarrow (B_{r,2}, \ldots, B_{r,m_r}, A_2, \ldots, A_m)\gamma_{\Sigma(r-1)+r}$
$R_{r,1}$	$\vdots$
$\vdots$	$\vdots$
$R_{r,|R_r|}$	$\leftarrow (A_2, \ldots, A_m)\gamma_{\Sigma_r+r}$

Fig. 4.20: The corresponding SLD-derivation

This leads us to the following definition.

DEFINITION 4.21 (Corresponding SLD-Refutation):

Suppose P is a definite program and G is a definite goal consisting of a single atom. Let $S = D_1, D_2, Q_2, D_3, Q_3, \ldots, D_r, Q_r$ be the sequence of input clauses used for a GDFC-refutation of $P \cup \{G\}$ via the left-first computation rule w.r.t. L_P^{fin} as defined above. Then the *corresponding SLD-refutation* of $P \cup \{G\}$ via the left-first computation rule uses the same input clauses in the order $f_{SLD}(S)$, where

$$f_{SLD}(S) = \begin{cases} S, \text{ if } r=1 \\ D_r, D_{r-1}, \ldots, D_1, f_{SLD}(Q_2), f_{SLD}(Q_3), \ldots, f_{SLD}(Q_r), \text{ otherwise.} \end{cases}$$ ■

Next we demonstrate how to compute a GDFC-refutation out of a given SLD-refutation. Let us consider the proof of the first literal of the goal $\leftarrow A_1, \ldots, A_m$. If we apply the left-first computation rule and G_r is the first goal which is shorter than its predecessor, i.e., C_r is the first fact in the sequence of input clauses. Suppose each C_i, $1 \le i \le r$, has the form $B_i \leftarrow B_{i,1}, \ldots, B_{m_i}$. Then the r-th goal G_r is

$$\leftarrow (B_{r-1,2}, \ldots, B_{r-1,m_{r-1}}, B_{r-2,2}, \ldots, B_{r-2,m_{r-2}}, \ldots, B_{1,2}, \ldots, B_{1,m_1}, A_2, \ldots, A_m)\theta_1 \ldots \theta_r$$

Since we use the left-first computation rule, the atoms of this goal are eliminated from left to right. Suppose Q_{r-i} is the sequence of input clauses used to eliminate all literals coming from $(B_{r-i,2}, \ldots, B_{r-i,m_{r-i}})\theta_1 \ldots \theta_{\Sigma(i-1)+r}$ $(i=1, \ldots, r-1)$, where $\Sigma_0 = 0$ and $\Sigma_i = |Q_{r-1}| + \ldots + |Q_{r-i}|$, for $i=1, \ldots, r-1$. Consequently $G_{\Sigma(i-1)+r}$ is

$$\leftarrow (B_{r-i,2}, \ldots, B_{r-i,m_{r-i}}, \ldots, B_{1,2}, \ldots, B_{1,m_1}, A_2, \ldots, A_m)\theta_1 \ldots \theta_{\Sigma(i-1)+r}$$

for $i=2, \ldots, r$, and $G_{\Sigma(r-1)+r}$ is

$$\leftarrow (A_2, \ldots, A_m)\theta_1 \ldots \theta_{\Sigma(r-1)+r}$$

The complete sequence of input clauses needed to derive $G_{\Sigma(r-1)+r}$ is

$$C_1, \ldots, C_r, Q_{r-1}, \ldots, Q_1$$

Now the corresponding GDFC-derivation which is also based on the left-first computation rule can easily be constructed. It follows from the proof of the completeness result, that we can use the sequence R_i of input clauses, which is a permutation of the clauses contained in Q_i, to eliminate all literals coming from $(B_{r-i,2}, \ldots, B_{r-i,m_{r-i}})\theta_1 \ldots \theta_{\Sigma(i-1)+r}$, for each $i \in \{1, \ldots, r-1\}$. Lemma 4.15 which concerns the structure of GDFC-refutations implies that we can use C_r as first input clause. Next we successively use C_{i-1} followed by R_{i-1}, for $i=r, \ldots, 2$. Consequently, the whole sequence of input clauses needed to eliminate the first literal of G is

$$C_r, C_{r-1}, R_{r-1}, C_{r-2}, R_{r-2}, \ldots, C_1, R_1$$

This leads to the following definition.

DEFINITION 4.22 (Corresponding GDFC-Refutation):

Suppose P is a definite program and G is a definite goal consisting of a single atom. Let $S=C_1,\ldots,C_r,Q_{r-1},\ldots,Q_1$ be the sequence of input clauses used for an SLD-refutation of $P\cup\{G\}$ via the left-first computation rule as defined above. Then each *corresponding GDFC-refutation* of $P\cup\{G\}$ via the left-first computation rule w.r.t. L_P^{fin} uses the same input clauses in the order $f_{GDFC}(S)$, where

$$f_{GDFC}(S) = \begin{cases} S, \text{ if } r=1 \\ C_r,C_{r-1},f_{GDFC}(Q_{r-1}),C_{r-2},f_{GDFC}(Q_{r-2}),\ldots,C_1,f_{GDFC}(Q_1), \text{ otherwise.} \end{cases}$$ ■

Both functions defined above allow us to compute sequences of input clauses used for SLD-refutations out of given GDFC-refutations and vice versa. The proofs of Theorem 4.16 and Theorem 4.17 imply that both functions are defined for all sequences of input clauses used in a refutation.

We conclude this section presenting a theorem concerning the relationship between both functions for the computation of corresponding sequences of input clauses. The proof is straightforward. It can be done by induction on the length of the refutation.

THEOREM 4.23 (Relationship Between f_{SLD} and f_{GDFC}):

Suppose P is a definite program and G is a definite goal consisting of a single literal. Suppose the computation rules used for SLD- and GDFC-resolution always select the leftmost element. Let S be the sequence of input clauses used for an SLD-refutation of $P\cup\{G\}$ and S' the sequence of input clauses used for the corresponding GDFC-refutation of $P\cup\{G\}$ w.r.t. L_P^{fin}. Then the following two equations hold:

$$S = f_{SLD}(f_{GDFC}(S))$$
$$S' = f_{GDFC}(f_{SLD}(S'))$$

Informally spoken, f_{SLD} is the inverse mapping of f_{GDFC} and vice versa. ■

4.3 Summary

In this chapter we introduced goal-directed forward chaining as a linear resolution strategy. We showed that the meta-interpreter for goal-directed forward chaining implements GDFC-resolution. We proved that GDFC-resolution is sound and complete for definite logic programs (Theorem 4.18). From a practical point of view this means that every answer obtained by a GDFC-refutation procedure is correct and that a GDFC-refutation procedure with a fair search rule eventually finds every success branch in a GDFC-tree. We introduced the notion of corresponding refutations which are based on the left-first computation rule. We defined two functions mapping a sequence of input clauses used for a GDFC-refutation to a sequence of input clauses used for the corresponding SLD-refutation and vice versa. We showed that each of both functions inverts the other (Theorem 4.23). This result will be important in the following chapter where we compare the efficiency of GDFC- and SLD-resolution.

Chapter 5

EFFICIENCY OF GDFC-RESOLUTION

The goal of this chapter is to analyze the efficiency of GDFC-resolution. For that purpose we compare the number of inferences needed by GDFC- and SLD-resolution to solve a goal. The first section compares different properties of SLD- and GDFC-trees. The results presented there are mainly based on Section 4.2 where we showed that GDFC-resolution is sound and complete for definite logic programs and introduced the notion of corresponding refutations. The second section compares the average case complexity of SLD- and GDFC-resolution for propositional binary Prolog programs. The third section analyzes the efficiency of GDFC-resolution for taxonomic hierarchies. Finally, the fourth section determines the efficiency of GDFC-resolution for the procedures to compute the transitive and reflexive transitive closure of directed acyclic graphs.

5.1 Corresponding SLD- and GDFC-Trees

In the previous chapter we defined GDFC-resolution as a linear resolution strategy. The problem of finding a refutation with linear resolution strategies can also be viewed as a tree-searching problem [12]. The goal is to find one or some or even all of the success branches in the corresponding search tree. Consequently the comparison of the complexity of different resolution strategies can be reduced to the question how fast a solution can be found in the corresponding refutation trees. Clearly the efficiency of a strategy depends on the size of the refutation tree.

The strongest result of a comparison of GDFC- and SLD-resolution would be that one of both is more efficient than the other in every case. The following example, however, demonstrates that this is not true.

EXAMPLE 5.1 (Corresponding SLD- and GDFC-Trees):

Suppose the computation rule used by both strategies is left-first. Let us consider the following program and suppose the goal is ←p.

p←q,r	(C_1)
q←s	(C_2)
q←t	(C_3)
s←	(C_4)
t←	(C_5)

Suppose L_P contains the following link clauses:

link(q,p)←	(L_1)
link(s,q)←	(L_2)
link(t,q)←	(L_3)
link(s,p)←	(L_4)
link(t,p)←	(L_5)

Clearly p is not a logical consequence of this program. Hence SLD-resolution and GDFC-resolution w.r.t. L_P have to traverse the complete SLD- respectively GDFC-tree. Figure 5.2 illustrates that the SLD-tree is smaller than the GDFC-tree.

Now consider the program

p←q,r	(C_1)
p←q,t	(C_2)
q←	(C_3)

Suppose the goal is ←p and L_P is

link(q,p)←	(L_1)

Figure 5.3 shows that now the SLD-tree is larger than the GDFC-tree. ■

This example shows that there are situations in which SLD-resolution is more efficient than GDFC-resolution and vice versa. Thus neither of both strategies is always better than the other.

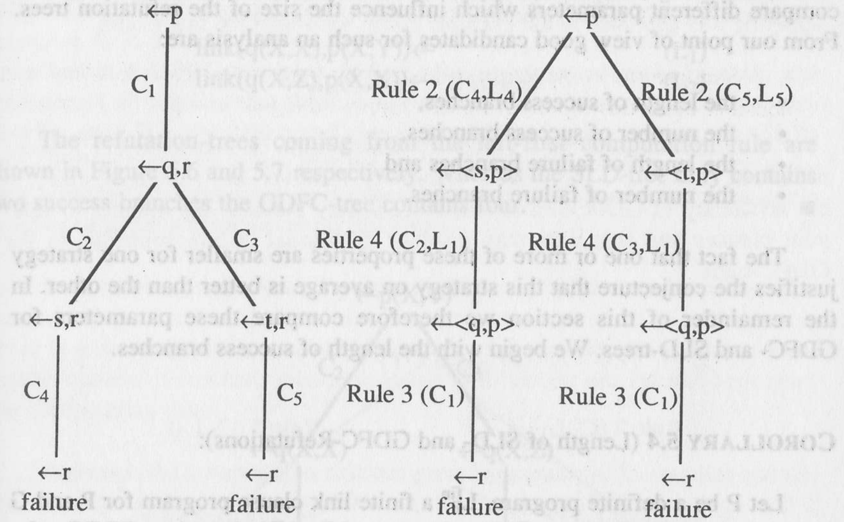

Fig. 5.2: Corresponding SLD- and GDFC-trees

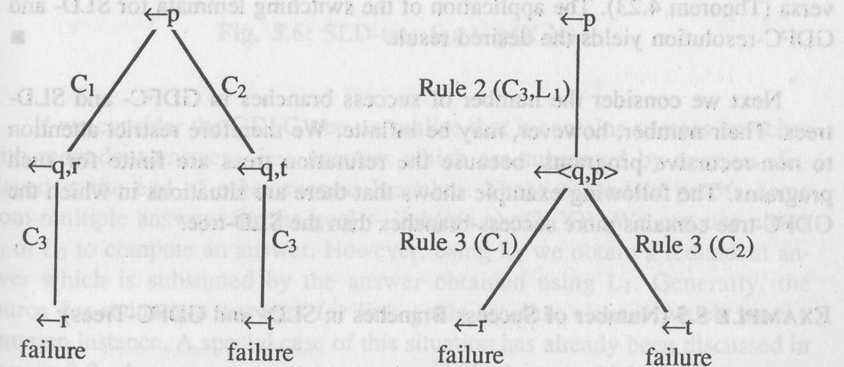

Fig. 5.3: Corresponding SLD- and GDFC-trees

In order to analyze the relative merits of both strategies, we subsequently compare different parameters which influence the size of the refutation trees. From our point of view good candidates for such an analysis are:

- the length of success branches,
- the number of success branches,
- the length of failure branches and
- the number of failure branches.

The fact that one or more of these properties are smaller for one strategy justifies the conjecture that this strategy on average is better than the other. In the remainder of this section we therefore compare these parameters for GDFC- and SLD-trees. We begin with the length of success branches.

COROLLARY 5.4 (Length of SLD- and GDFC-Refutations):

Let P be a definite program, L_P^{fin} a finite link clause program for P and G be a definite goal. For each SLD-refutation of P∪{G} there is a GDFC-refutation of P∪{G} w.r.t. L_P^{fin} which has the same length and vice versa. ■

Proof: The result follows immediately from the fact that every SLD-refutation has a corresponding GDFC-refutation with the same length and vice versa (Theorem 4.23). The application of the switching lemmata for SLD- and GDFC-resolution yields the desired result. ■

Next we consider the number of success branches in GDFC- and SLD-trees. Their number, however, may be infinite. We therefore restrict attention to non-recursive programs, because the refutation trees are finite for such programs. The following example shows that there are situations in which the GDFC-tree contains more success-branches than the SLD-tree.

EXAMPLE 5.5 (Number of Success Branches in SLD- and GDFC-Trees):

Consider the following program

q(a,a)←	(C_1)
p(X,Y)←q(X,X)	(C_2)
p(X,X)←q(X,Z)	(C_3)

Suppose the goal is ←p(X,Y) and L_P^{fin} contains the following link clauses

link(q(X,X),p(X,Y))←	(L_1)
link(q(X,Z),p(X,X))←	(L_2)

The refutation-trees coming from the left-first computation rule are shown in Figure 5.6 and 5.7 respectively. Whereas the SLD-tree only contains two success branches the GDFC-tree contains four. ■

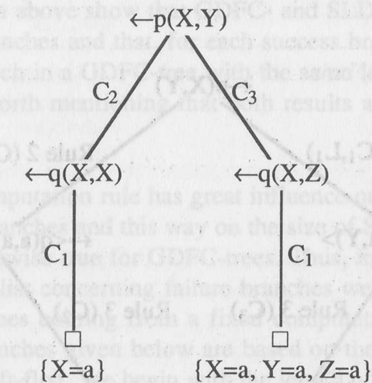

Fig. 5.6: SLD-tree for ←p(X,Y)

If we consider the GDFC-tree we realize that it contains success branches with redundant answers, i.e., answers which are subsumed by answers displayed at the end of other success branches. These redundant proofs come from multiple answers for the goal ←link(q(a,a),p(X,Y)). We can use either L_1 or L_2 to compute an answer. However, using L_2 we obtain a redundant answer which is subsumed by the answer obtained using L_1. Generally, the source for redundant answers for link goals are link clauses which have a common instance. A special case of this situation has already been discussed in Section 3.2 where we suggested to remove link clauses which are an instance of another. This approach, however, cannot be applied in the example above, because none of both is an instance of the other. One solution to this problem could be to replace all link clauses which have a common instance by their most specific generalization also denoted as the lowest common anti-unificator.

An algorithm to compute the lowest common anti-unificator is given in [44]. Applying this approach to the example above we obtain the link clause

link(q(X,Y),p(X,Z))←

A disadvantage of such a transformation could be that it further reduces the selectivity of L_P^{fin}. The weaker selectivity, however, has a negative effect to the efficiency of GDFC-resolution, since it may increase the number of relevant clauses and this way may produce additional failure branches in the GDFC-trees.

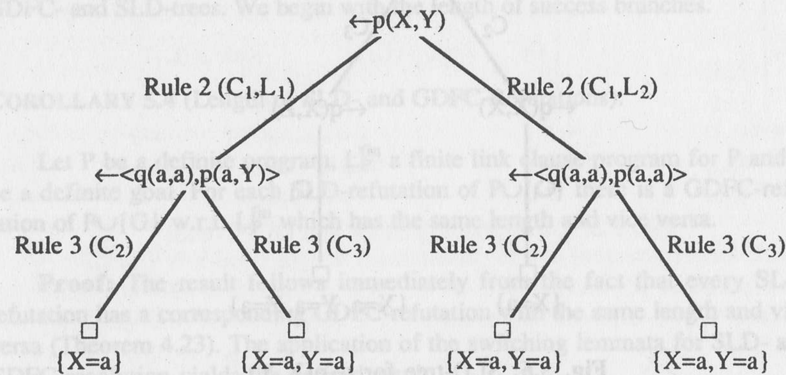

Fig. 5.7: GDFC-tree for ←p(X,Y)

In the context of propositional programs, however, L_P is finite and all link clauses in L_P are ground. Therefore, there are no link clauses which have a common instance. Since the link goals generated during the inference process are ground, we cannot obtain more than one answer for any link goal. Consequently, there are no redundant answers for link goals.

COROLLARY 5.8 (Number of Success Branches in SLD- and GDFC-Trees):

Let P be a non-recursive propositional program and G be a goal. Then each GDFC-tree for P∪{G} w.r.t. L_P contains the same number of success branches as each SLD-tree for P∪{G}. ■

Proof: Different success branches in SLD-trees differ in the sequence of input clauses used to derive the empty clause. Since P is a propositional program we have at most one answer for each link goal. Thus different success branches in a GDFC-tree also differ in the sequence of input clauses, and Theorem 4.23 implies that there is exactly one corresponding GDFC-refutation for each SLD-refutation and vice versa. The switching lemmata for SLD- and GDFC-resolution imply that the number of success branches is independent from the computation rule. Consequently, a GDFC-tree contains the same number of success branches as an SLD-tree. ■

Both results given above show that GDFC- and SLD-trees have the same number of success branches and that, for each success branch in an SLD-tree, there is a success branch in a GDFC-tree with the same length and vice versa. In this context it is worth mentioning that both results are independent from the computation rule.

However, the computation rule has great influence on the number and the length of the failure branches and this way on the size of SLD-trees [46], and it is clear that this is likewise true for GDFC-trees. Thus, in order to discuss the last two topics of our list concerning failure branches we have to restrict ourselves to refutation trees coming from a fixed computation rule. The results concerning failure branches given below are based on the assumption that the selection function is left-first. We begin with the length of failure branches and consider a small example first.

EXAMPLE 5.9 (Failed SLD- and GDFC-Derivations):

Consider the following program:

a←b	(C_1)
b←c,d	(C_2)
c←	(C_3)
d←e	(C_4)

Suppose G is ←a, L_P is

link(b,a)←
link(c,b)←
link(c,a)←
link(e,d)←

and the selection rule is left-first. Since P contains no fact B← such that link(B,d) holds, GDFC-resolution stops with failure when d is selected. SLD-resolution additionally uses the first and fourth clause so that GDFC-resolution in this case detects failure earlier than SLD-resolution. Both finitely failed trees for ←a are contained in Figure 5.10. ■

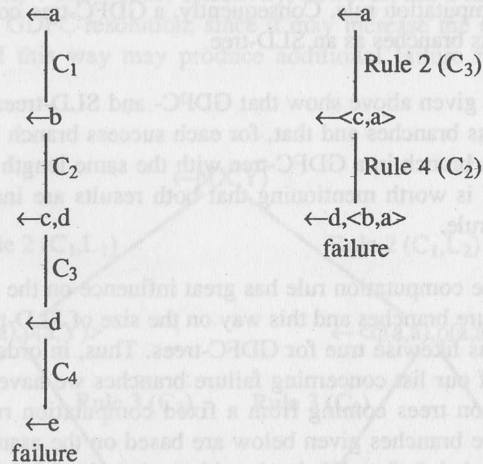

Fig. 5.10: Failed SLD- and GDFC-derivations

Example 5.9 demonstrates that failed GDFC-derivations may be shorter than failed SLD-derivations. The following theorem shows that, for each failed GDFC-derivation via the left-first computation rule, there is a failed SLD-derivation via the left-first computation rule with at least the same length.

THEOREM 5.11 (Failed SLD- and GDFC-Derivations):

Let P be a propositional program and G be a goal. Suppose the computation rules always select the leftmost element. Then, for each failed GDFC-derivation of $P \cup \{G\}$ w.r.t. L_P with length N_{GDFC}, there is a failed SLD-derivation of $P \cup \{G\}$ with length $N_{SLD} \geq N_{GDFC}$. ■

To simplify the proof, we first give the following lemma.

LEMMA 5.12 (Last Selected Element):

Let P be a propositional program and G be a goal. The last selected element of each failed GDFC-derivation of $P \cup \{G\}$ w.r.t. L_P, i.e., the selected element of the last goal of each failed GDFC-derivation is an atom. ■

Proof of Lemma 5.12: By contradiction. Suppose <A,B> is the last selected element in a failed GDFC-derivation. Hence link(A,B) must be a logical consequence of L_P. Otherwise the subgoal-goal pair <A,B> could not be generated. Since link(A,B) is true, P must contain a clause with leftmost body literal A and head A' such that either A' equals B or link(A',B) is a logical consequence of L_P. Therefore, either Rule 3 or 4 can be applied and <A,B> cannot be the last selected element. ■

Proof of Theorem 5.11: By induction on the length N_{GDFC} of the GDFC-derivation.

The inequality clearly holds if $N_{GDFC}=0$. Nevertheless we show that N_{SLD} may even be strictly greater than N_{GDFC}. Suppose G is $\leftarrow A_1,A_2,\ldots,A_m$. Since A_1 is selected in the first step, P cannot contain a unit clause B← such that either link(B,A_1) holds or B equals A_1. If P contains no clause with head A_1 then SLD-resolution directly fails. However, if P contains a set $\{E_1,\ldots,E_w\}$ ($w \geq 1$) of non-unit clauses such that the head of E_1 equals A_1 and the head of E_{i+1} equals the first body literal of E_i, for i=1,...,w-1, but there is no clause the head of which equals the leftmost body literal of E_w, SLD-resolution needs w steps.

Now suppose the result holds for each failed GDFC-derivation of maximal length N_{GDFC}-1. Suppose G is $\leftarrow A_1,A_2,\ldots,A_m$ and A_k ($1 \leq k \leq m$) is the leftmost literal whose evaluation fails. Suppose GDFC-resolution needs s steps to derive G_s which is

$$\leftarrow A_k,\ldots,A_m$$

Let us consider the failed derivation of $P \cup \{G_s\}$. We distinguish two cases. First, there is no fact B← in P such that either link(B,A_k) holds or B is unifiable with A_k. This case is similar to that discussed in the induction hypothesis. Whereas GDFC-resolution directly stops with failure, the failed SLD-derivation possibly has length $w \geq 0$.

Second, P contains a fact $B_1 \leftarrow$ such that $link(B_1,A_k)$ is a logical consequence of L_P. Note that B_1 must be distinct from A_k, since the proof of A_k would succeed otherwise. Therefore, the failed derivation consists of a finite sequence $G_s,\ldots,G_{s+t}$ $(t\geq2)$ of definite forward chaining goals with G_{s+t} as last goal. Since the last selected element is an atom, t must be greater or equal 2. Suppose $C_{s+1},\ldots,C_{s+t}$ are the input clauses used in this derivation. Because $link(B_1,A_k)$ holds, the sequence $C_{s+1},\ldots,C_{s+t}$ must contain r clauses $D_1,\ldots,D_r$ $(1\leq r\leq t)$ such that $D_1=B_1\leftarrow$, the head of D_i equals the first body literal of D_{i+1}, for $i=1,\ldots,r-1$, and the evaluation of one of the body literals of D_r fails. Let D_i be of the form $B_i\leftarrow B_{i,1},B_{i,2},\ldots,B_{i,m_i}$ $(m_i\geq1)$, for $i=2,\ldots,r$.

Clearly G_{s+1} is

$$\leftarrow\langle B_1,A_k\rangle,A_{k+1},\ldots,A_m$$

Hence $C_{s+2}=D_2=B_2\leftarrow B_{2,1},B_{2,2},\ldots,B_{2,m_2}$ with $B_1=B_{2,1}$ is used as input clause so that G_{s+2} is

$$\leftarrow B_{2,2},\ldots,B_{2,m_2},\langle B_2,A_k\rangle,A_{k+1},\ldots,A_m$$

Since the computation rule always selects the leftmost element, next the literals $B_{2,2},\ldots,B_{2,m_2}$ are eliminated. Suppose Q_2 is the sequence of input clauses used in this derivation. Consequently, after $|Q_2|$ steps, $G_{s+|Q_2|+2}$ is

$$\leftarrow\langle B_2,A_k\rangle,A_{k+1},\ldots,A_m$$

Suppose Q_i is the possibly empty sequence of input clauses used to eliminate all elements coming from $B_{i,2},\ldots,B_{i,m_i}$, i.e., between D_i and D_{i+1}, $(i=1\ldots r-1)$. Let Q_1 be $\emptyset$ and Q_r be the sequence of input clauses used after D_r.

Whenever $G_{s+|Q_i|+i}$ $(i=1,\ldots,r-2)$ is

$$\leftarrow\langle B_i,A_k\rangle,A_{k+1},\ldots,A_m$$

we use D_{i+1} as input clause to derive $G_{s+|Q_i|+i+1}$ which is

$$\leftarrow B_{i+1,2},\ldots,B_{i+1,m_{i+1}},\langle B_{i+1},A_k\rangle,A_{k+1},\ldots,A_m$$

If $i=r-1$ we have to distinguish two cases. If B_r equals A_k (Situation 1) then we apply Rule 3 to derive

$$\leftarrow B_{r,2},\ldots,B_{r,m_r},A_{k+1},\ldots,A_m$$

Otherwise, we have Situation 2 and apply Rule 4 to derive

$$\leftarrow B_{r,2},\ldots,B_{r,m_r},\langle B_r,A_k\rangle,A_{k+1},\ldots,A_m$$

For $i=1,\ldots,r-2$, whenever the actual goal, $G_{s+|Q_i|+i+1}$, is

$$\leftarrow B_{i+1,2},\ldots,B_{i+1,m_{i+1}},\langle B_{i+1},A_k\rangle,A_{k+1},\ldots,A_m$$

we eliminate the first $m_{i+1}-1$ literals using the clauses contained in Q_{i+1} to obtain $G_{s+|Q_{i+1}|+i+1}$ which is

$$\leftarrow\langle B_{i+1},A_k\rangle,A_{k+1},\ldots,A_m$$

If $i=r-1$, GDFC-resolution independently from the situation uses the clauses contained in Q_r and stops with failure. Figure 5.13 illustrates this failed GDFC-derivation of $P\cup\{G_s\}$ in more details. The length of the complete failed GDFC-derivation of $P\cup\{G\}$ is

$$N_{GDFC} = s+r+\sum_{i=1}^{r}|Q_i|$$

We now construct a failed SLD-derivation of $P\cup\{G\}$. Corollary 5.4 implies that there is a corresponding SLD-derivation of G_s needing the same number of steps and the same input clauses as the GDFC-derivation constructed above.

Let us consider the failed SLD-derivation of $P\cup\{G_s\}$. Since $link(D_r,A_k)$ holds, P must contain $v\geq0$ clauses $D_i=B_i\leftarrow B_{i,1},\ldots,B_{i,m_i}$, for $i=r+1,\ldots,r+v$, such B_{r+v} equals A_k, $B_{i+1,1}$ equals B_i, for $i=r+1,\ldots,r+v-1$, and $B_{r+1,1}$ equals B_r, i.e. the head of D_r. In Situation 1 we have $B_r=A_k$ and $v=0$.

Consequently, starting with G_s we can use the clauses $D_{r+v},\ldots,D_{r+1}$ as input clauses to derive the goal

$$\leftarrow B_{r+1,1},\ldots,B_{r+1,m_{r+1}},B_{r+2,2},\ldots,B_{r+2,m_{r+2}},\ldots,B_{r+v,2},\ldots,B_{r+v,m_{r+v}},A_{k+1},\ldots,A_m$$

Input clauses	Derived goals
	$\leftarrow A_k,A_{k+1},\ldots,A_m$
$C_{s+1}=D_1$	$\leftarrow <B_1,A_k>,A_{k+1},\ldots,A_m$
$C_{s+2}=D_2$	$\leftarrow B_{2,2},\ldots,B_{2,m2},<B_2,A_k>,A_{k+1},\ldots,A_m$
$Q_{2,1}$	⋮
⋮	⋮
$Q_{2,\lvert Q2\rvert}$	$\leftarrow <B_2,A_k>,A_{k+1},\ldots,A_m$
D_3	$\leftarrow B_{3,2},\ldots,B_{3,m3},<B_3,A_k>,A_{k+1},\ldots,A_m$
$Q_{3,1}$	⋮
⋮	⋮
D_{r-1}	$\leftarrow B_{r-1,2},\ldots,B_{r-1,m_{r-1}},<B_{r-1},A_k>A_{k+1},\ldots,A_m$
$Q_{r-1,1}$	⋮
⋮	⋮
$Q_{r-1,\lvert Q_{r-1}\rvert}$	$\leftarrow <B_{r-1},A_k>,A_{k+1},\ldots,A_m$
D_r	$\leftarrow B_{r,2},\ldots,B_{r,m_r},<B_r,A_k>,A_{k+1},\ldots,A_m$
$Q_{r,1}$	⋮
⋮	⋮
$Q_{r,\lvert Q_r\rvert}$	⋮
	failure

Fig. 5.13: Failed GDFC-derivation of $P\cup\{G_s\}$ (Situation 2)

Since $B_{r+1,1}$ equals B_r and the head of D_i equals the leftmost body literal of D_{i+1}, for each $i\in\{1,\ldots,r-1\}$, $D_r,D_{r-1},\ldots,D_1$ can be used as input clauses in the next r steps. After that the actual goal is

$$\leftarrow B_{2,2},\ldots,B_{2,m2},\ldots,B_{r,2},\ldots,B_{r,m_r},B_{r+1,2},\ldots,B_{r+v,m_{r+v}},A_{k+1},\ldots,A_m$$

Corollary 5.4 implies that there is a corresponding SLD-refutation of $P\cup\{\leftarrow B_{i,2},\ldots,B_{i,m_i}\}$ using the same input clauses contained in Q_i in a possibly other order, for $i=2,\ldots,r-1$. Let us denote the corresponding sequence for each Q_i used for the SLD-refutation by R_i. The induction hypothesis implies that there is a failed SLD-derivation, which uses at least all clauses contained in Q_r, for $P\cup\{\leftarrow B_{r,2},\ldots,B_{r,m_r}\}$. Since the last selected element is an atom, this corresponding failed SLD-derivation may be w steps longer. This situation is similar to that discussed in the induction hypothesis. If we denote this corresponding sequence by R_r then the concatenation of the sequences $R_2,\ldots,R_r$ gives us the order in which the remaining input clauses are used before the derivation

ends with failure. Note that $|R_r|=|Q_r|+w$. The corresponding failed SLD-derivation of $P\cup\{G_s\}$ is illustrated in Figure 5.14.

Input clauses	Derived Goals
	$\leftarrow A_k,A_{k+1},\ldots,A_m$
D_{r+v}	$\leftarrow B_{r+v,1},\ldots,B_{r+v,m_{r+v}},A_{k+1},\ldots,A_m$
⋮	⋮
D_{r+1}	$\leftarrow B_{r+1,1},\ldots,B_{r+1,m_{r+1}},B_{r+2,2},\ldots,B_{r+v,m_{r+v}},A_{k+1},\ldots,A_m$
D_r	$\leftarrow B_{r,1},B_{r,2},\ldots,B_{r,m_r},B_{r+1,2},\ldots,B_{r+v,m_{r+v}},A_{k+1},\ldots,A_m$
⋮	⋮
D_1	$\leftarrow B_{2,2},\ldots,B_{r,m_r},B_{r+1,2},\ldots,B_{r+v,m_{r+v}},A_{k+1},\ldots,A_m$
$R_{2,1}$	⋮
⋮	⋮
$R_{2,\lvert R2\rvert}$	$\leftarrow B_{3,2},\ldots,B_{r,m_r},B_{r+1,2},\ldots,B_{r+v,m_{r+v}},A_{k+1},\ldots,A_m$
⋮	⋮
$R_{r-1,\lvert R_{r-1}\rvert}$	$\leftarrow B_{r,2},\ldots,B_{r,m_r},B_{r+1,2},\ldots,B_{r+v,m_{r+v}},A_2,\ldots,A_m$
$R_{r,1}$	⋮
⋮	⋮
$R_{r,\lvert R_r\rvert}$	⋮
	failure

Fig. 5.14: Corresponding failed SLD-derivation of $P\cup\{G_s\}$

Based on this construction we can estimate the number N_{SLD} of input clauses used in the failed SLD-derivation by

$$N_{GDFC}=s+r+\sum_{i=1}^{r}|Q_i|\le s+r+\sum_{i=1}^{r}|R_i|\le s+r+v+\sum_{i=1}^{r}|R_i|=N_{SLD}$$

since $|R_r|=|Q_r|+w$ $(w\ge 0)$. ■

Note that it may be the case that there is more than one such corresponding failed SLD-derivation for a given failed GDFC-derivation. This is due to the fact that there is possibly more than one sequence of input clauses we can use to construct R_r out of Q_r. Furthermore, there may be different sequences with the same properties as $D_{r+v},\ldots,D_{r+1}$. Since there is exactly one answer for any link goal generated in the derivation ('yes' or 'no'), each failed SLD-

derivation can have at most one corresponding failed GDFC-derivation in the context of propositional programs. The notion of corresponding failed derivations is specified more precisely in the following definition.

DEFINITION 5.15 (Corresponding Failed SLD- and GDFC-Derivations):

Suppose P is a propositional program and G is a goal. Then the relation between corresponding failed SLD- and GDFC-derivations w.r.t. L_P via the left-first computation rule contains all tuples (S,S') where S and S' are sequences of clauses contained in P satisfying the following conditions:

1) S has the form

$$C_1,\ldots,C_s,D_1,D_2,Q_2,D_3,Q_3,\ldots,D_{r-1},Q_{r-1},D_r,Q_r$$

and contains all input clauses used by a failed GDFC-derivation of $P\cup\{G\}$ via the left-first computation rule w.r.t. L_P as defined in the proof of Theorem 5.11.

2) S' has the form

$$f_{SLD}(C_1,\ldots,C_s),D_{r+v},\ldots,D_{r+1},D_r,\ldots,D_1,f_{SLD}(Q_2),\ldots,f_{SLD}(Q_{r-1}),R_r$$

as defined in the proof of Theorem 5.11 where $|R_r|\geq|Q_r|$. ■

The next two corollaries directly follow from the proof of Theorem 5.11 and the definition of corresponding failed derivations given above. The first one compares the length of failure branches and the second one the number of failure branches.

COROLLARY 5.16 (Length of Failed SLD- and GDFC-Derivations):

Let P be a propositional program and G be a goal. Suppose we always apply the left-first computation rule. Then each failed SLD-derivation of $P\cup\{G\}$ is at least as long as its corresponding failed GDFC-derivation of $P\cup\{G\}$ w.r.t. L_P. ■

COROLLARY 5.17 (Number of Failed SLD- and GDFC-Derivations):

Let P be a non-recursive, propositional program, G be a goal and suppose we always apply the left-first computation rule. Then there are at least as many failed SLD-derivations of $P\cup\{G\}$ as failed GDFC-derivations of $P\cup\{G\}$ w.r.t. L_P. ■

Figure 5.18 sums up the results obtained in this section. If we consider non-recursive, propositional programs then, independently from the computation rule, number and length of success branches in GDFC- and SLD-trees are equal. If we consider only such trees coming from the left-first computation rule, then GDFC-trees contain at least as many failure branches as corresponding SLD-trees. Furthermore, for each failure branch in a GDFC-tree there is a failure branch in the corresponding SLD-tree which has at least the same length.

Topic	GDFC ↔ SLD
Length of success branches	=
Number of success branches	=
Length of failure branches	≤
Number of failure branches	≤

Fig. 5.18: Comparison of different tree properties for non-recursive propositional programs

This result suggests that GDFC-resolution on average is more efficient than SLD-resolution for propositional logic programs, because it can be expected that GDFC-trees are very often smaller than their corresponding SLD-trees. From a practical point of view it is important to know whether there is an inevitable overhead in an implementation of GDFC-resolution for propositional programs which could equalize the reduced number of inferences. The crucial operation which could cause such an overhead is the evaluation of link goals, i.e., the selection of possibly relevant clauses. For SLD-resolution powerful compilation techniques are known, which allow an access of the procedure relevant for the actual goal in constant time [52]. But also for GDFC-resolution the list of clauses relevant for a given atom or subgoal-goal pair can be accessed in constant time. This is achieved by indexing schemes which are

based on the link clauses. For every atom resp. subgoal-goal pair, we store the list of facts resp. non-unit clauses which are possibly relevant. In Section 6.2 we present an interpreter for goal-directed forward chaining (Program 6.7) which realizes such an indexing on the basis of the first argument indexing implemented in most of the commercially available Prolog systems.

The following sections of this chapter are concerned with the question how much the number of necessary inferences needed with GDFC-resolution to find one or all solutions of a goal differs from that of SLD-resolution.

5.2 Average Complexities of Propositional Binary Programs

The analysis of the average case complexity of a resolution strategy is a complex problem, since it requires to argue about a possibly large class of programs. In addition the result of such a study may be contestable from a pragmatical point of view, because programs are considered which are irrelevant in practice. Both problems may be one of the reasons why 'so far only few theoretical work has been done on average complexity of algorithms in logic' [37]. From a theoretical viewpoint, however, analytical results concerning the average complexity are important for several reasons. They help to identify program classes in which a certain behaviour can be expected and they can be used to explain a behaviour observed in practice.

To confirm our conjecture that GDFC-resolution on average is more efficient than SLD-resolution for propositional logic programs, we therefore concentrate on cases where results for SLD-resolution have been achieved. We consider tree-like, binary, propositional programs, analyze the average case complexity of finding the first solution of a goal with GDFC-resolution and compare it with the results presented by Kleine Büning and Löwen [37] for SLD-resolution. The following definitions are partly taken from this publication. Throughout this and the following sections the notion complexity denotes the number of inferences needed.

DEFINITION 5.19 (Binary Program):

A definite program is *binary* if each of its clauses contains at most one body literal. ■

In this section we consider programs as sequences (and not as sets) allowing that a clause occurs twice in a program. As already mentioned in [37], programs containing clauses twice generally do not play a role in practice and most of all programs are not tree-like. But the behaviour of many programs which are not tree-like can be simulated by tree-like programs containing clauses twice.

DEFINITION 5.20 (Program Graph):

Let P be a binary propositional program. The *program graph* G_P for P is an ordered, directed multi-graph containing an arc (A,B) for each clause $A \leftarrow B \in P$. ■

Note that program graphs, in contrast to predicate dependency graphs, are multi-graphs. Furthermore, the edges going out of a vertex are ordered, since programs are considered as ordered sequences of clauses.

The previous definition shows that there is a strong relationship between binary programs and directed graphs. Based on this equivalence we often speak about graphs instead of programs in the remainder of this section. The program graphs must be multi-graphs, since we allow clauses to occur twice in a program. In this context we say (A,B) is an *unary arc*, if $A \leftarrow B$ occurs once in P, and a *binary arc*, if $A \leftarrow B$ occurs twice in P.

DEFINITION 5.21 (Tree-Like Program):

A definite program P is *tree-like* if the predicate dependency graph D_P of P is a tree. ■

In this section we consider tree-like, binary programs only. Furthermore, we restrict the procedures to contain at most two clauses with a different predicate symbol in the body. This is not a strong restriction, since every binary program can be transformed to a program satisfying this condition [37]. Following the approach to transform arbitrary trees into binary trees we simply replace clauses of the form $A \leftarrow B_i$, $1 \leq i \leq n$, in P by the $2 \cdot n - 2$ clauses

$A \leftarrow B_1$
$A \leftarrow X_1$
$X_{i-1} \leftarrow B_i$ $\quad 2 \leq i \leq n-1$
$X_{i-1} \leftarrow X_i$ $\quad 2 \leq i \leq n-2$
$X_{n-2} \leftarrow B_n$

where X_i, $1 \leq i \leq n-2$, are new symbols not occurring in P. Therefore, we subsequently consider only such trees in which each node has an out-degree of at most 2. We say a vertex respectively atom A is *unary*, if there is exactly one atom B such that $A \leftarrow B$ occurs in P. A is called *binary*, if there are exactly two different atoms B and B' with $A \leftarrow B$ and $A \leftarrow B'$ occur in P.

Subsequently we consider three different classes of binary, tree-like programs which are introduced by the following definition.

DEFINITION 5.22 (Chain, Full Binary Tree and Binary Tree):

Let B be the set of all tree-like, binary, propositional programs modulo renaming. Since each vertex in the program graph G_P for a program $P \in B$ is unary, binary or has an out-degree 0, we say B is the set of all *binary trees*. $C \subseteq B$ is the set of all *chains*, i.e., $P \in C$ if all vertices in G_P are either unary or have out-degree 0. $F \subseteq B$ is the set of *full binary trees*, that is, $P \in F$ if each vertex in G_P is either binary or has out-degree 0.

For each $\Phi \in \{C,F,B\}$, $\Phi(n) \subseteq \Phi$ denotes the class of all programs containing n vertices. With root(P) we denote the root of the tree. ■

The fact that we consider each element of B as a representative of the equivalence class, containing all programs which are equivalent modulo renaming, allows us to omit the labels of vertices in the program graphs. Figure 5.23 shows one example for each of the three program classes considered here.

There is a strong relation between the process of finding an SLD-refutation of $P \cup \{G\}$ and a search through the program graph G_P, if G is a single literal goal. Let us suppose the facts are linked to the corresponding vertices by additional edges. Thus SLD-resolution corresponds to a left-first depth-first search with backtracking through G_P starting from the node corresponding to G. Accordingly, applying GDFC-resolution is to start with a fact node from which there is a path to G and traversing G_P bottom-up until G is reached. The number of inferences needed equals the number of edges traversed.

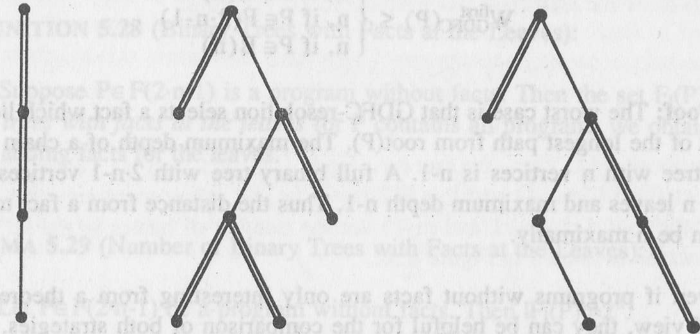

Fig. 5.23: Chain, full binary tree and binary tree

THEOREM 5.24 (Worst Case Complexity of SLD-Resolution):

Suppose P is a chain, binary or full binary tree. The worst case complexity $W^{first}_{SLD}(P)$ of finding the first solution of root(P) with SLD-resolution is

$$W^{first}_{SLD}(P) \leq \begin{cases} 2^n-2, \text{ if } P \in C(n) \\ n \cdot 2^n, \text{ if } P \in F(2 \cdot n-1) \\ 3 \cdot 2^n-2, \text{ if } P \in B(n) \end{cases}$$ ■

Proof: For chains we show by induction on n that $W^{first}_{SLD}(P) \leq 2^n-2$. Clearly the worst case is that P contains no facts. The inequation is correct if n=1. Now suppose the result holds for n-1. If $P \in C(n)$ then

$$W^{first}_{SLD}(P) \leq 2 + 2 \cdot (2^{n-1}-2) = 2 \cdot 2^{n-1} - 4 + 2 = 2^n - 2$$

The results for binary and full binary trees have been shown in [37]. ■

THEOREM 5.25 (Worst Case Complexity of GDFC-Resolution):

Suppose P is a chain, binary or full binary tree. The worst case complexity $W^{first}_{GDFC}(P)$ of finding the first solution of root(P) with GDFC-resolution is

$$W^{first}_{GDFC}(P) \leq \begin{cases} n, \text{ if } P \in C(n) \\ n, \text{ if } P \in F(2 \cdot n-1) \\ n, \text{ if } P \in B(n) \end{cases}$$ ■

Proof: The worst case is that GDFC-resolution selects a fact which lies at the end of the longest path from root(P). The maximum depth of a chain or a binary tree with n vertices is n-1. A full binary tree with 2·n-1 vertices has exactly n leaves and maximum depth n-1. Thus the distance from a fact to the root can be n maximally. ■

Even if programs without facts are only interesting from a theoretical point of view, they can be helpful for the comparison of both strategies. The following two theorems concern the average case complexity of SLD- and GDFC-resolution for programs without facts.

THEOREM 5.26 (SLD-Resolution for Programs without Facts):

Let $\Phi \in \{C,F,B\}$, $\Phi(n) \subseteq \Phi$ be the set of all programs containing n vertices and $\varphi(n)$ be the subset of $\Phi(n)$ containing all programs without any facts. Then the average case complexity $A^{first}_{SLD}(\varphi(n))$ of finding the first solution for the root with SLD-resolution is exponential. ■

Proof: See [37]. ■

THEOREM 5.27 (GDFC-Resolution for Programs without Facts):

Let $\Phi \in \{C,F,B\}$, $\Phi(n) \subseteq \Phi$ be the set of all programs containing n vertices and $\varphi(n)$ be the subset of $\Phi(n)$ containing all programs without facts. Then the average case complexity $A^{first}_{GDFC}(\varphi(n))$ to find the first solution for the root with GDFC-resolution is 0. ■

Proof: Since the programs in $\varphi(n)$ contain no facts, neither Rule 1 nor Rule 2 can be applied to the first goal. Thus GDFC-resolution performs no inferences. ■

Next we consider binary trees allowing that facts occur at the leaves only. First we compute the average case complexity for the set of programs which only differ from a full binary tree without facts in the set of facts added for the leaves.

DEFINITION 5.28 (Binary Trees with Facts at the Leaves):

Suppose $P \in F(2 \cdot n-1)$ is a program without facts. Then the set $F_l(P)$ of *binary trees with facts at the leaves* for P contains all programs we obtain from P by adding facts for the leaves.

LEMMA 5.29 (Number of Binary Trees with Facts at the Leaves):

Let $P \in F(2 \cdot n-1)$ be a program without facts. Then $|F_l(P)|=2^n$.

Proof: The result follows from the fact that P has n leaves [37]. ■

THEOREM 5.30 (Average Case Complexity of SLD-Resolution for $F_l(P)$):

Suppose $P \in F(2 \cdot n-1)$ is a program without facts. The average case complexity of $F_l(P)$ using SLD-resolution is

$$A^{first}_{SLD}(F_l(P)) \leq (2 \cdot n-1)^2$$ ■

Proof: See [37]. ■

THEOREM 5.31 (Average Case Complexity of GDFC-Resolution for $F_l(P)$):

Suppose $P \in F(2 \cdot n-1)$ is a program without facts. The average case complexity of $F_l(P)$ using GDFC-resolution is

$$A^{first}_{GDFC}(F_l(P)) \leq \frac{(n+3)}{2}$$ ■

Proof: It is easy to verify that there are 2^{n-1} programs in $F_l(P)$ containing a certain fact. Since there are n facts, we have to compute the average over $n \cdot 2^{n-1}+1$ programs. Let D_i be the distance from fact i to the root, that is the length of the path from the root to the corresponding fact plus one. Thus

$$A^{first}_{GDFC}(F_l(P)) = \frac{2^{n-1}}{n \cdot 2^{n-1}+1} \sum_{i=1}^{n} D_i \leq \frac{1}{n} \sum_{i=1}^{n} D_i$$

To show the result we prove by induction on n that

$$\sum_{i=1}^{n} D_i \le \frac{n\cdot(n+3)}{2} - 1$$

Suppose n=1. Then the fact has depth 1 so that $D_1=1\le1$. Now suppose $n\ge2$. Let $P_1\in F(2\cdot m_1-1)$ and $P_2\in F(2\cdot m_2-1)$ be the left respectively right subtree of the root, i.e., $m_1\ge1$ and $m_2\ge1$ are the number of leaves in the left resp. right subtree as shown in Figure 5.32.

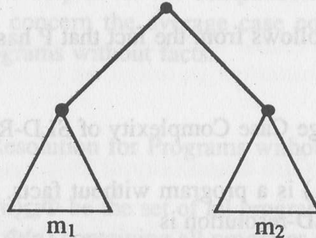

Fig. 5.32: Left and right subtree

Furthermore, let $D_{l,i}$ be the depth of fact i ($1\le i\le m_1$) in the left subtree and $D_{r,j}$ the depth of fact j ($1\le j\le m_2$) in the right subtree. Thus

$$\sum_{i=1}^{n} D_i = \sum_{i=1}^{m_1} D_{l,i} + \sum_{i=1}^{m_2} D_{r,i} + n \le$$

$$\frac{m_1\cdot(m_1+3)}{2} - 1 + \frac{m_2\cdot(m_2+3)}{2} - 1 + n =$$

$$\frac{n\cdot(n+3)}{2} + (n-1-m_1\cdot m_2) \le \frac{n\cdot(n+3)}{2} - 1$$

since $n-1-m_1\cdot m_2\le0$. ■

The previous theorem shows that there is a linear upper bound for the average case complexity of GDFC-resolution for binary trees with facts. This upper bound is given by the average depth of the facts. It is easy to verify that

the average depth is linear in the worst case. For example, the left and right 'snakes' illustrated in Figure 5.33 both have a linear average depth of facts and thus represent this worst case. But which upper bound can we expect on average?

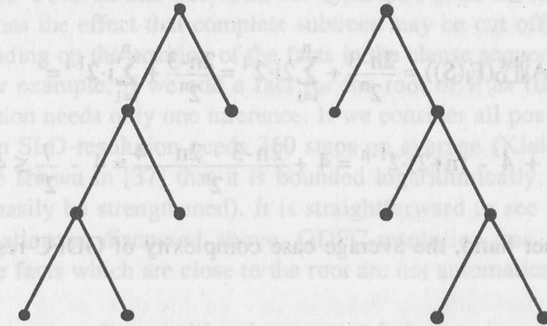

Fig. 5.33: Left and right 'snake'

THEOREM 5.34 (Average Upper Bound for Full Binary Trees):

The average value of $A^{first}_{GDFC}(F_l(P))$ taken over all Programs $P\in F(2\cdot n-1)$ which contain no facts is bounded by $O(\sqrt{n})$. ■

Proof: The result follows from the fact that the average depth over all binary trees with $2\cdot n-1$ vertices is bounded by $O(\sqrt{n})$ [39]. ■

The fact that the upper bound for the average case complexity of GDFC-resolution for $F_l(P)$ is smaller than that of SLD-resolution does not mean that this is also true for each binary tree P. Moreover, it is possible to characterize subclasses of F for which SLD-resolution on average is more efficient than GDFC-resolution.

EXAMPLE 5.35 (Average Complexities for Right Snakes):

Let $S\in F(2\cdot n-1)$ be a right snake without facts which has unary edges only. Then

$$A^{first}_{SLD}(F_l(S)) \leq 4$$

$$A^{first}_{GDFC}(F_l(S)) \geq \frac{n^2+3\cdot n-2}{2\cdot(n+1)}$$

For the average case complexity of SLD-resolution we have

$$A^{first}_{SLD}(F_l(S)) = \frac{2n-3}{2^n} + \sum_{i=1}^{n} 2i\cdot 2^{-i} = \frac{2n-3}{2^n} + \sum_{i=1}^{n} i\cdot 2^{1-i} =$$

$$\frac{2n-3}{2^n} + 4 - (n+2)\cdot 2^{1-n} = 4 + \frac{2n-3-2n-4}{2^n} = 4 - \frac{7}{2^n} \leq 4$$

On the other hand, the average case complexity of GDFC-resolution for $F_l(S)$ is

$$A^{first}_{GDFC}(F_l(P)) = \frac{2^{n-1}}{n\cdot 2^{n-1}+1}\sum_{i=1}^{n} D_i$$

Thus

$$A^{first}_{GDFC}(F_l(P)) \geq \frac{2^{n-1}}{2^{n-1}\cdot(n+1)}\sum_{i=1}^{n} D_i = \frac{2^{n-1}}{2^{n-1}\cdot(n+1)}\cdot\frac{n^2+3n-2}{2} = \frac{n^2+3n-2}{2\cdot(n+1)}$$

Whereas the upper bound for the average case complexity of SLD-resolution is 4, we have a linear lower bound for GDFC-resolution. ■

This result, however, only has negligible influence on the general comparison of both approaches for propositional binary programs, since the right snakes with unary vertices are a very small subset only. Moreover, if we allow binary edges, then the average number of inferences needed with SLD-resolution may significantly increase even for the right snake. In contrast to that, the average case complexity of GDFC-resolution remains unchanged, because it does not depend on the type of the edges.

The previous example demonstrates a weakness of GDFC-resolution. The reason for the better average case complexity of SLD-resolution for right snakes with unary edges is that SLD-resolution always visits all leaves in the order of their distance from the root. Facts which are close to the root are used first and therefore carry more weight than the others. For GDFC-resolution, however, all facts have equal weights so that not always the fact with the smallest depth is selected.

This matter of fact also plays a role if we consider binary trees and allow to add facts not only for the leaves of the tree. In contrast to the situation discussed above we even can add facts for symbols which are also defined by rules. This has the effect that complete subtrees may be cut off for SLD-resolution depending on the position of the facts in the clause sequences of the procedures. For example, if we add a fact for the root of P as first clause, then SLD-resolution needs only one inference. If we consider all possible assertions of facts then SLD-resolution needs 360 steps on average (Kleine Büning and Löwen have shown in [37] that it is bounded logarithmically, but this upper bound can easily be strengthened). It is straightforward to see that we have a similar situation as discussed above. GDFC-resolution has a linear upper bound, since facts which are close to the root are not automatically preferred.

In the context of propositional programs, however, it seems not to be a realistic situation from a practical point of view that facts may be added for each procedure. In contrast, it is more reasonable to expect that the number of facts to be added or the number of procedures to which they can be added is restricted. In this case the amount of subtrees cut off is decreased, so that the average case complexity of SLD-resolution increases. The complexity of GDFC-resolution, however, remains unchanged under these restrictions.

5.3 Complexities of Taxonomic Hierarchies

In the previous section we considered binary, tree-like, propositional programs and analyzed the average case complexity of SLD- and GDFC-resolution to find the first solution of the root of the tree. For this problem we obtained a linear upper bound for the average case complexity of GDFC-resolution. However, it is not less important to consider how both approaches behave if one is interested in all solutions.

For that purpose we consider a special subclass of datalog programs used for the representation of taxonomic hierarchies. In many applications, the possibly structured objects denoting individuals or sets form a taxonomic hierarchy [55]. The tree-like taxonomic divisions of animals discussed in Chapter 1 are a common example. Such hierarchies play an important role in the context

of frame systems in which they are used to inherit properties of structured objects along specializations [8].

According to Montini [54] there is a straightforward top-down approach frequently used to describe taxonomies in logic programs. A class is represented as an unary predicate symbol. For each membership of an object to a class, there is one clause

class_name(object_name)←

Furthermore, there is one clause

class_name(C)←
subclass_name(C)

for each class-subclass relation (for an instance of this scheme see Example 1.1).

Subsequently we consider the average case complexity of SLD- and GDFC-resolution to find the first and all solutions of a query for the root of the hierarchy. For the sake of simplicity we consider tree-like, binary taxonomic hierarchies only. Furthermore we assume that the trees are full and all leaves have the same depth, that is, the trees are balanced. In the remainder of this section P_n denotes a program implementing a full, binary and balanced taxonomic tree consisting of 2^n-2 rules.

THEOREM 5.36 (Average Complexities of Finding the First Solution):

Let P_n be a program implementing a full, binary and balanced taxonomic tree consisting of 2^n-2 rules and G be a goal for the root of the hierarchy which has at least one solution. Then SLD-resolution always needs at least as many inferences as GDFC-resolution to find the first solution of G, since the following relationships hold for the best, average and worst case complexities of both approaches:

$$B^{first}_{GDFC}(P_n) = A^{first}_{GDFC}(P_n) = W^{first}_{GDFC}(P_n) = n$$

$$n \leq B^{first}_{SLD}(P_n) \leq A^{first}_{SLD}(P_n) \leq W^{first}_{SLD}(P_n) = 2^n-1$$ ■

Proof: If we use GDFC-resolution we start with a relevant fact for G which must exist since G has a solution. The depth of the taxonomic tree is n-1. Since all leaves have the same depth and we need one additional inference, we have n inferences in the best, average and worst case. Now let us consider what happens if we use SLD-resolution. Clearly the best case is that the left-most branch in the SLD-tree is a success branch. Since this leaf has depth n, the best case for SLD-resolution is n. In the worst case the rightmost branch is the only success branch so that we have to traverse the whole SLD-tree which has exactly 2^n-1 arcs. ■

THEOREM 5.37 (Average Complexities of Finding All Solutions):

Let P_n be a program implementing a full, binary and balanced taxonomic tree consisting of 2^n-2 rules and let G be a goal for which there are $m \geq 1$ solutions. Then the best, average and worst case complexities of finding all solutions with GDFC- and SLD-resolution are

$$B^{all}_{GDFC}(P_n) = A^{all}_{GDFC}(P_n) = W^{all}_{GDFC}(P_n) = m \cdot n$$

$$B^{all}_{SLD}(P_n) = A^{all}_{SLD}(P_n) = W^{all}_{SLD}(P_n) = 2^n-2+m$$ ■

Proof: Since G has m solutions, the GDFC-tree for $P_n \cup \{G\}$ contains exactly m success branches which have length n. Furthermore, the tree contains no failure branches. Thus GDFC-resolution needs m·n inferences to find all solutions of G, in the best, average and worst case.

To find all solutions with SLD-resolution we have to traverse the complete taxonomic tree which requires 2^n-2 steps. For each solution we additionally need one inference. Thus the total number of inferences needed to find all solutions in the best, average and worst case is 2^n-2+m. ■

Based on these results we now can give an answer to the question in which situations we should use GDFC-resolution to compute all solutions for the root of the tree. In order that GDFC-resolution is more efficient than SLD-resolution m·n must be less than 2^n-2+m. This is equivalent to the following inequation:

$$m < \frac{2^n-2}{n-1}$$

In the context of datalog programs, however, we have to respect that there may be an inherent overhead for GDFC-resolution coming from the increased unification costs and the costs for the selection of relevant facts. Let us again consider the last four clauses of the meta-interpreter for goal-directed forward chaining (Program 2.1). In the third clause we unify the head of the unit clause with the selected literal. In the fourth clause, however, we have to unify two atoms, since we have to solve ←link(A,B). In the fourth clause we unify the atoms in the subgoal-goal pair with the leftmost body literal and the head of the input clause. Finally, in the last clause we have to unify three atoms. Thus, we on average have to unify two atoms per clause whereas we only have to unify one atom in the third clause of the standard interpreter for pure Prolog (Program 2.2). One therefore could roughly estimate the unification overhead of GDFC-resolution by a factor of two. However, the total amount of the additional unifications in practice depends on the size of the atoms to be unified, and we will see in Section 6.2 that the effective degree of this overhead may be so small that it can be ignored. Still, in order to respect that both approaches require a different amount of unifications let us suppose that c on average is the factor by which the unification costs of GDFC-resolution differ from that of SLD-resolution. Thus, the total costs of GDFC-resolution are c·m·n which results in

$$m < \frac{2^n - 2}{c \cdot n - 1}$$

Accordingly, if N is the number of rules then GDFC-resolution is more efficient than SLD-resolution if

$$m < \frac{N}{c \cdot \log(N)}$$

approximately.

A further interesting question is, how these result change if we consider inverted trees such as Example 1.3. An inverted tree is obtained from a tree by inverting all arcs. Suppose we are interested in all objects belonging to a certain top level class. Clearly the number of inferences needed with SLD-resolution is bounded by m+n where n is the depth of the tree and m is the number of solutions. In contrast to that the upper bound for GDFC-resolution is m·(n+1), since GDFC-resolution traverses the whole path from the leaf to the top-level class for each solution. Even if GDFC-resolution does no unnecessary

inferences it is always less efficient than SLD-resolution if the query has more than one solution.

5.4 Complexities of Transitive Closures

A procedure commonly used to compare the efficiency of different strategies is the ancestor relationship [4]. We discussed the corresponding procedure in Example 3.40. This procedure equals the procedure computing the transitive closure of a directed graph:

p(X,Y)←e(X,Y) (C_1)
p(X,Z)←e(X,Y),p(Y,Z) (C_2)

The relation to compute the reflexive and transitive closure of a directed graph is very similar:

p(X,X)← (D_1)
p(X,Z)←e(X,Y),p(Y,Z) (D_2)

Throughout this section, we assume that the link clause program in both cases is

link(e(X,Y),p(X,Z))←

The link clause coming from C_1 is useless, because it is subsumed by that coming from C_2. We furthermore assume that GDFC- and SLD-resolution both apply the left-first computation rule.

To analyze the complexity of finding all solutions with GDFC- and SLD-resolution we have to respect all four binding patterns of goals for the transitive and reflexive, transitive closure of e/2, namely ff, fb, bf and bb [81]. The term fb means that the first argument of the atom is free and the second is bound.

To avoid the problem of infinite loops, we assume that all tuples in e/2 are ground graphs are acyclic. Furthermore, we assume that all tuples in e/2 are ground and that each arc occurs only once in e/2.

With $d_{out}(v)$ we denote the out-degree of each node in the graph, that is, the number of facts in the relation e containing v in the first argument position. Accordingly, $d_{in}(v)$ is the in-degree of v, i.e., the number of unit clauses in e/2 with v in the second position.

LEMMA 5.38 (Binding Pattern bf for the Transitive Closure):

Let v be a node in the directed acyclic graph specified by e/2. Suppose $d_{out}(v)$ is the out-degree and $v_1,\ldots,v_{d_{out}(v)}$ are the successor nodes of v. Suppose the goal for the procedure to compute the transitive closure of e/2 is ←p(v,X), which has the binding pattern bf. Then the number of inferences needed to find all solutions with SLD- resp. GDFC-resolution, C^{bf}_{SLD} resp. C^{bf}_{GDFC}, via the left-first computation rule is determined as follows:

$$C^{bf}_{SLD}(v) = \begin{cases} 2, \text{ if } d_{out}(v) = 0 \\ 2\cdot d_{out}(v) + 2 + \sum_{i=1}^{d_{out}(v)} C^{bf}_{SLD}(v_i), \text{ otherwise} \end{cases}$$

$$C^{bf}_{GDFC}(v) = \begin{cases} 0, \text{ if } d_{out}(v) = 0 \\ 3\cdot d_{out}(v) + \sum_{i=1}^{d_{out}(v)} C^{bf}_{GDFC}(v_i), \text{ otherwise} \end{cases}$$ ■

Proof: Consider the SLD-tree for ←p(v,X). Since there are only two clauses in P which can be used as input clauses, the root node has exactly two successor nodes. While the left one contains the goal ←e(v,X) we have the goal ←e(v,Z),p(Z,X) in the right one. If v has out-degree 0 then both nodes are failure nodes. Otherwise, if $d_{out}(v)>0$ then each of these goals has $d_{out}(v)$ successor nodes. Whereas the left node has nodes representing the empty clause as successor nodes, the successor nodes of the right node are of the form ←p(v_i,X), where $i \in \{1,\ldots,d_{out}(v)\}$. Figure 5.39 shows the first three levels of this SLD-tree.

Now consider the GDFC-tree for ←p(v,X). If the out-degree of v is 0, then there are no relevant facts for G so that the root node has no successor nodes. Otherwise, we have $d_{out}(v)$ relevant facts and the root has $d_{out}(v)$ successor nodes of the form ←<e(v,v_i),p(v,X)>, for $i=1,\ldots,d_{out}(v)$. Since there are two non-unit clauses fitting to each subgoal-goal pair, each of these nodes has two successor nodes, the left one containing the empty clause and the right one containing the goal ←p(v_i,X). Thus the GDFC-tree contains $3\cdot d_{out}(v)$ arcs

plus the number of arcs needed to solve each of the goals ←p(v_i,X), for $i=1,\ldots,d_{out}(v)$. The first three levels of this tree are illustrated in Figure 5.40. ■

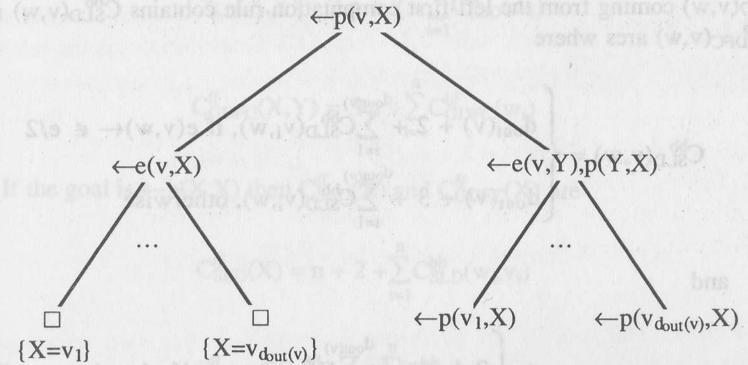

Fig. 5.39: The first three levels of an SLD-tree for the binding pattern bf

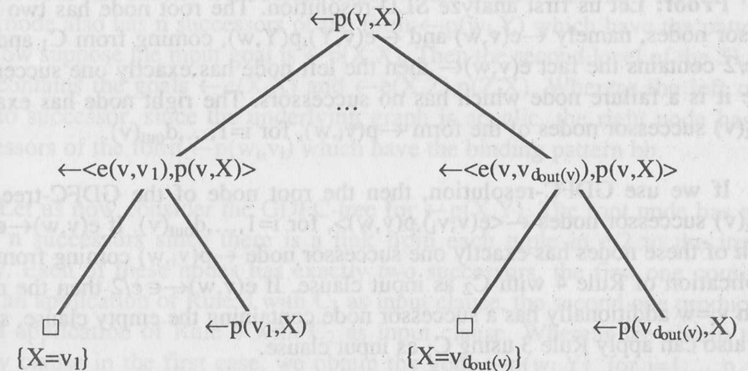

Fig. 5.40: The first three levels of a GDFC-tree for the binding pattern bf

LEMMA 5.41 (Binding Pattern bb for the Transitive Closure):

Let v and w be two arbitrary nodes of the directed acyclic graph defined by the binary relation e/2. Suppose $d_{out}(v)$ is the out-degree and $v_1,\ldots,v_{d_{out}(v)}$ are the successor nodes of v. Then the SLD- resp. GDFC-tree for the goal $\leftarrow p(v,w)$ coming from the left-first computation rule contains $C^{bb}_{SLD}(v,w)$ resp. $C^{bb}_{GDFC}(v,w)$ arcs where

$$C^{bb}_{SLD}(v,w) = \begin{cases} d_{out}(v) + 2 + \sum_{i=1}^{d_{out}(v)} C^{bb}_{SLD}(v_i,w), & \text{if } e(v,w)\leftarrow\ \notin e/2 \\ d_{out}(v) + 3 + \sum_{i=1}^{d_{out}(v)} C^{bb}_{SLD}(v_i,w), & \text{otherwise} \end{cases}$$

and

$$C^{bb}_{GDFC}(v,w) = \begin{cases} 2\cdot d_{out}(v) + \sum_{i=1}^{d_{out}(v)} C^{bb}_{GDFC}(v_i,w), & \text{if } e(v,w)\leftarrow\ \notin e/2 \\ 2\cdot d_{out}(v) + 1 + \sum_{i=1}^{d_{out}(v)} C^{bb}_{GDFC}(v_i,w), & \text{otherwise} \end{cases}$$ ■

Proof: Let us first analyze SLD-resolution. The root node has two successor nodes, namely $\leftarrow e(v,w)$ and $\leftarrow e(v,Y),p(Y,w)$, coming from C_1 and C_2. If e/2 contains the fact $e(v,w)\leftarrow$ then the left node has exactly one successor, else it is a failure node which has no successors. The right node has exactly $d_{out}(v)$ successor nodes of the form $\leftarrow p(v_i,w)$, for $i=1,\ldots,d_{out}(v)$.

If we use GDFC-resolution, then the root node of the GDFC-tree has $d_{out}(v)$ successor nodes $\leftarrow \langle e(v,v_i),p(v,w)\rangle$, for $i=1,\ldots,d_{out}(v)$. If $e(v,w)\leftarrow\notin e/2$ each of these nodes has exactly one successor node $\leftarrow p(v_i,w)$ coming from the application of Rule 4 with C_2 as input clause. If $e(v,w)\leftarrow\in e/2$ then the node with $v_i=w$ additionally has a successor node containing the empty clause, since we also can apply Rule 3 using C_1 as input clause. ■

LEMMA 5.42 (Binding Pattern ff for the Transitive Closure):

Suppose e/2 defines a directed acyclic graph and consists of n facts. Suppose v_i and w_i, $i\in\{1,\ldots,n\}$, are the nodes occurring in the first respectively

second argument position of the i-th fact. Then the number of inferences needed to find all solutions of the goal $\leftarrow p(X,Y)$ with SLD- resp. GDFC-resolution via the left-first computation rule, $C^{ff}_{SLD}(X,Y)$ resp. $C^{ff}_{GDFC}(X,Y)$, is:

$$C^{ff}_{SLD}(X,Y) = 2\cdot n + 2 + \sum_{i=1}^{n} C^{bf}_{SLD}(w_i)$$

$$C^{ff}_{GDFC}(X,Y) = 3\cdot n + \sum_{i=1}^{n} C^{bf}_{GDFC}(w_i)$$

If the goal is $\leftarrow p(X,X)$ then $C^{ff}_{SLD}(X)$ and $C^{ff}_{GDFC}(X)$ are

$$C^{ff}_{SLD}(X) = n + 2 + \sum_{i=1}^{n} C^{bb}_{SLD}(w_i,v_i)$$

$$C^{ff}_{GDFC}(X) = 2\cdot n + \sum_{i=1}^{n} C^{bb}_{GDFC}(w_i,v_i)$$ ■

Proof: Suppose the goal is $\leftarrow p(X,Y)$. Again the root of the SLD-tree has exactly two successors, namely $\leftarrow e(X,Y)$ and $\leftarrow e(X,Z),p(Z,Y)$. Since $|e/2|=n$, the left node has exactly n successor nodes which contain the empty clause. The right node also has n successors of the form $\leftarrow p(w_i,Y)$ which have the pattern bf. Now suppose the input goal is $\leftarrow p(X,X)$. Then the second level of the SLD-tree contains the goals $\leftarrow e(X,X)$ and $\leftarrow e(X,Z),p(Z,X)$. Whereas the left one has no successor, since the underlying graph is acyclic, the right node has n successors of the form $\leftarrow p(w_i,v_i)$ which have the binding pattern bb.

Let us now consider the GDFC-tree for $\leftarrow p(X,Y)$. The root node has exactly n successors since there is a link from each tuple in e/2 to the input query. Each of these nodes has exactly two successors, the first one coming from an application of Rule 3 with C_1 as input clause, the second one produced by an application of Rule 4 with C_2 as input clause. Whereas we obtain the empty clause in the first case, we obtain the goal $\leftarrow p(w_i,Y)$, for $i=1,\ldots,n$, in the second case. If the goal is $\leftarrow p(X,X)$, then the root has n successors of the form $\leftarrow\langle e(v_i,w_i),p(v_i,v_i)\rangle$ each of which has exactly one successor of the form $\leftarrow p(w_i,v_i)$. ■

LEMMA 5.43 (Binding Pattern fb for the Transitive Closure):

Suppose the goal is $\leftarrow p(X,v)$ where v is an arbitrary node occurring in the directed acyclic graph specified by the binary relation e/2. Suppose e/2 consists of n facts $e(u_i,w_i)\leftarrow$, for i=1,...,n. Suppose v has in-degree $d_{in}(v)$ and $v_1,\ldots,v_{d_{in}(v)}$ are the direct predecessors v. Then the number of inferences needed to find all solutions with SLD- resp. GDFC-resolution via the left-first computation rule, $C^{fb}_{SLD}(v)$ resp. $C^{fb}_{GDFC}(v)$, is:

$$C^{fb}_{SLD}(v) = 2 + d_{in}(v) + n + \sum_{i=1}^{n} C^{bb}_{SLD}(w_i,v)$$

$$C^{fb}_{GDFC}(v) = d_{in}(v) + 2\cdot n + \sum_{i=1}^{n} C^{bb}_{GDFC}(w_i,v)$$

■

Proof: The root of the SLD-tree for $\leftarrow p(X,v)$ has two successors, namely $\leftarrow e(X,v)$ and $\leftarrow e(X,Y),p(Y,v)$. The left node has exactly $d_{in}(v)$ successors, one for each direct predecessor of v. The right node, however, has n successors, since e(X,Y) unifies with all unit clauses in e/2. The resulting goals are $\leftarrow p(w_i,v)$, for i=1,...,n, which have the binding pattern bb.

The root $\leftarrow p(X,v)$ of the corresponding GDFC-tree has n successor nodes of the form $\leftarrow$<$e(u_i,w_i),p(u_i,v)$>. Since v has $d_{in}(v)$ direct predecessors it $d_{in}(v)$ times must be possible to apply Rule 3 with C_1 as input clause which produces the empty clause. Furthermore we n times can apply Rule 4 with C_2 as input clause. This yields n successors of the form $\leftarrow p(w_i,v)$ with binding pattern bb.

■

THEOREM 5.44 (Complexities for the Transitive Closure):

Suppose P is a program containing the specification of a directed acyclic graph and the two clauses C_1 and C_2 defining the transitive closure of the graph. Furthermore suppose we always apply the left-first computation rule. Then GDFC-resolution is more efficient than SLD-resolution, that is, the GDFC-tree for $P\cup\{G\}$ w.r.t. {link(e(X,Y),p(X,Z)←)} contains less arcs than the corresponding SLD-tree for each goal G for p/2. ■

Proof: To prove the theorem we show by induction that GDFC-resolution is more efficient than SLD-resolution for all four binding patterns.

Let us begin with the binding pattern bf. Suppose v is a terminal node, i.e., $d_{out}(v)=0$. It follows from Lemma 5.38 that

$$0 = C^{bf}_{GDFC}(v) < C^{bf}_{SLD}(v) = 2$$

Now suppose $d_{out}(v)>0$ and the inequation holds for all direct successors of v. Thus

$$C^{bf}_{GDFC}(v) = 3\cdot d_{out}(v) + \sum_{i=1}^{d_{out}(v)} C^{bf}_{GDFC}(v_i)) \leq$$

$$3\cdot d_{out}(v) + \sum_{i=1}^{d_{out}(v)} (C^{bf}_{SLD}(v_i)-1) < 2\cdot d_{out}(v) + 2 + \sum_{i=1}^{d_{out}(v)} C^{bf}_{SLD}(v_i) = C^{bf}_{SLD}(v)$$

Next we consider the binding pattern bb. Suppose the query is $\leftarrow p(v,w)$. If we assume that v is a terminal node, i.e., $d_{out}(v)=0$, then Lemma 5.41 implies that

$$0 = C^{bb}_{GDFC}(v,w) < C^{bb}_{SLD}(v,w) = 2$$

Next suppose the inequation is true for all direct successors of v. Suppose s=0 if $e(v,w)\leftarrow \notin e/2$ and s=1, otherwise. Consequently

$$C^{bb}_{GDFC}(v,w) = 2\cdot d_{out}(v) + s + \sum_{i=1}^{d_{out}(v)} C^{bb}_{GDFC}(v_i,w) \leq$$

$$2\cdot d_{out}(v) + s + \sum_{i=1}^{d_{out}(v)} (C^{bb}_{SLD}(v_i,w)-1) <$$

$$d_{out}(v) + 2 + s + \sum_{i=1}^{d_{out}(v)} C^{bb}_{SLD}(v_i,w) = C^{bb}_{SLD}(v,w)$$

The next binding pattern is ff. Here we have to distinguish two cases:

$$C^{ff}_{GDFC}(X,Y) = 3\cdot n + \sum_{i=1}^{n} C^{bf}_{GDFC}(w_i) \leq 3\cdot n + \sum_{i=1}^{n} (C^{bf}_{SLD}(w_i)-1) < C^{ff}_{SLD}(X,Y)$$

and

$$C^{ff}_{GDFC}(X) = 2\cdot n + \sum_{i=1}^{n} C^{bb}_{GDFC}(w_i,v_i) \le 2\cdot n + \sum_{i=1}^{n}(C^{ff}_{SLD}(w_i,v_i)-1) < C^{ff}_{SLD}(X)$$

The last case concerns the pattern fb:

$$C^{fb}_{GDFC}(v) = d_{in}(v) + 2\cdot n + \sum_{i=1}^{n} C^{bb}_{GDFC}(w_i,v) \le$$

$$d_{in}(v) + 2\cdot n + \sum_{i=1}^{n}(C^{bb}_{SLD}(w_i,v)-1) < C^{fb}_{SLD}(v)$$

■

This result is very important since it suggests that GDFC-resolution, independently from the structure of the underlying directed acyclic graph, always needs fewer inferences than SLD-resolution to compute all answers for a query to the ancestor relation. In the following example we address the question which scale the difference between both strategies may have.

EXAMPLE 5.45 (Transitive Closure for Full, Balanced, Binary Trees):

Suppose e/2 specifies a full and balanced binary tree consisting of depth n. Furthermore suppose v is the root and the input query is $\leftarrow p(v,X)$. Lemma 5.38 implies that

$$C_{SLD}(n) = \begin{cases} 2, & \text{if } n = 0 \\ 6 + 2\cdot C_{SLD}(n-1), & \text{otherwise} \end{cases}$$

$$C_{GDFC}(n) = \begin{cases} 0, & \text{if } n = 0 \\ 6 + 2\cdot C_{GDFC}(n-1), & \text{otherwise} \end{cases}$$

This is equivalent to

$$C_{SLD}(n) = 8\cdot 2^n - 6$$

$$C_{GDFC}(n) = 6\cdot 2^n - 6$$

Thus

$$\lim_{n\to\infty} \frac{C_{SLD}(n)}{C_{GDFC}(n)} = \lim_{n\to\infty} \frac{8\cdot 2^n-6}{6\cdot 2^n-6} = \frac{4}{3}$$

Since

$$\frac{C_{SLD}(n)}{C_{GDFC}(n)} \ge \frac{4}{3}$$

for n≥1, GDFC-resolution is at least 25% better than SLD-resolution. ■

In the remainder of this section we compare the efficiency of both strategies for the procedure to compute the reflexive, transitive closure. Again we consider trees coming from the left-first computation rule and begin with the binding pattern bf.

LEMMA 5.46 (Binding Pattern bf for the Reflexive, Transitive Closure):

Let v be a node in the directed acyclic graph specified by e/2 where $d_{out}(v)$ is the out-degree and $v_1,\ldots,v_{d_{out}(v)}$ are the successor nodes of v. Suppose the goal for the procedure to compute the reflexive, transitive closure of e/2 is $\leftarrow p(v,X)$, which has the binding pattern bf. Then the following two formulas specify the size of the SLD- resp. GDFC-tree coming from the left-first computation rule, D^{bf}_{SLD} resp. D^{bf}_{GDFC}, for this goal:

$$D^{bf}_{SLD}(v) = d_{out}(v) + 2 + \sum_{i=1}^{d_{out}(v)} D^{bf}_{SLD}(v_i)$$

$$D^{bf}_{GDFC}(v) = 2\cdot d_{out}(v) + 1 + \sum_{i=1}^{d_{out}(v)} D^{bf}_{GDFC}(v_i)$$

■

Proof: Again we first analyze the behaviour of SLD-resolution. Since the goal is $\leftarrow p(v,X)$, the root of the SLD-tree has two successor nodes, the left one coming from D_1 and the right one coming from D_2. While the left node contains the empty clause and therefore has no children, the right node has the form $\leftarrow e(v,Y),p(Y,Z)$ which has exactly $d_{out}(v)$ children corresponding to the direct successors of v in the underlying graph.

On the other hand the root of the GDFC-tree for $\leftarrow p(v,X)$ has $1+d_{out}(v)$ children coming from D_1 and the $d_{out}(v)$ direct successors of v. Whereas the

first one contains the empty clause, the other nodes contain the goal $\leftarrow<e(v,v_i),p(v,X)>$, for $i=1,\ldots,d_{out}(v)$. Except the first, each of these nodes has exactly one successor of the form $\leftarrow p(v_i,X)$ which comes from an application of Rule 4 using D_2 as input clause. ■

Since the proofs of the following lemmata directly correspond to that of the previous lemma, we omit them for the sake of brevity.

LEMMA 5.47 (Binding Pattern bb for the Reflexive, Transitive Closure):

Let v and w be two arbitrary nodes of the directed acyclic graph defined by the binary relation e/2. Suppose $d_{out}(v)$ is the out-degree and $v_1,\ldots,v_{d_{out}(v)}$ are the successor nodes of v. Then the SLD- resp. GDFC-tree for the goal $\leftarrow p(v,w)$ coming from the left-first computation rule contains $D^{bb}_{SLD}(v,w)$ resp. $D^{bb}_{GDFC}(v,w)$ arcs where

$$D^{bb}_{SLD}(v,w) = \begin{cases} d_{out}(v) + 1 + \sum_{i=1}^{d_{out}(v)} D^{bb}_{SLD}(v_i,w), \text{ if } v \neq w \\ d_{out}(v) + 2 + \sum_{i=1}^{d_{out}(v)} D^{bb}_{SLD}(v_i,w), \text{ otherwise} \end{cases}$$

and

$$D^{bb}_{GDFC}(v,w) = \begin{cases} 2\cdot d_{out}(v) + \sum_{i=1}^{d_{out}(v)} D^{bb}_{GDFC}(v_i,w), \text{ if } v \neq w \\ 2\cdot d_{out}(v) + 1 + \sum_{i=1}^{d_{out}(v)} D^{bb}_{GDFC}(v_i,w), \text{ otherwise} \end{cases}$$ ■

LEMMA 5.48 (Binding Pattern ff for the Reflexive, Transitive Closure):

Suppose e/2 specifies a directed acyclic graph and consists of n facts where v_i and w_i, $i\in\{1,\ldots,n\}$ are the nodes occurring in the first respectively second argument position of the i-th fact. Then the SLD- resp. GDFC-tree for the goal $\leftarrow p(X,Y)$ coming from the left-first computation rule has $D^{ff}_{SLD}(X,Y)$ respectively $D^{ff}_{GDFC}(X,Y)$ arcs, where

$$D^{ff}_{SLD}(X,Y) = n + 2 + \sum_{i=1}^{n} D^{bf}_{SLD}(w_i)$$

$$D^{ff}_{GDFC}(X,Y) = 2\cdot n + 1 + \sum_{i=1}^{n} D^{bf}_{GDFC}(w_i)$$

If X and Y are the same variable, i.e., the goal is $\leftarrow p(X,X)$, then $D^{ff}_{SLD}(X)$ and $D^{ff}_{GDFC}(X)$ are

$$D^{ff}_{SLD}(X) = n + 2 + \sum_{i=1}^{n} D^{bb}_{SLD}(w_i,v_i)$$

$$D^{ff}_{GDFC}(X) = 2\cdot n + 1 + \sum_{i=1}^{n} D^{bb}_{GDFC}(w_i,v_i)$$ ■

LEMMA 5.49 (Binding Pattern fb for the Reflexive, Transitive Closure):

Suppose the goal is $\leftarrow p(X,v)$ where v is an arbitrary node occurring in the directed acyclic graph specified by the binary relation e/2. Suppose e/2 consists of n facts $e(u_i,w_i)\leftarrow$, for $i=1,\ldots,n$. Let $D^{fb}_{SLD}(v)$ and $D^{fb}_{GDFC}(v)$ be the number of inferences needed to find all predecessors of v with SLD- resp. GDFC-resolution applying the left-first computation rule. Then

$$D^{fb}_{SLD}(v) = 2 + n + \sum_{i=1}^{n} D^{bb}_{SLD}(w_i,v)$$

$$D^{fb}_{GDFC}(v) = 1 + 2\cdot n + \sum_{i=1}^{n} D^{bb}_{GDFC}(w_i,v)$$ ■

THEOREM 5.50 (Complexities for the Reflexive, Transitive Closure):

Suppose P is a program containing a specification of a directed acyclic graph in the binary relation e/2 and the two clauses D_1 and D_2 defining the reflexive, transitive closure of the graph. Furthermore suppose we always apply the left-first computation rule. Then GDFC-resolution is more efficient than SLD-resolution, because, for each goal G for p/2, the GDFC-tree for $P\cup\{G\}$

w.r.t. {link(e(X,Y),p(X,Z)←} is exactly by one arc smaller than the corresponding SLD-tree. ■

Proof: Again we show this result by induction.

Let us begin with bf. Suppose v is a terminal node, i.e., $d_{out}(v)=0$. Lemma 5.46 implies that

$$D^{bf}_{SLD}(v) = 2 = D^{bf}_{GDFC}(v) + 1$$

Now suppose the result holds for all successor nodes of v. Consequently

$$D^{bf}_{SLD}(v) = d_{out}(v) + 2 + \sum_{i=1}^{d_{out}(v)} D^{bf}_{SLD}(v_i) = d_{out}(v) + 2 + \sum_{i=1}^{d_{out}(v)} (D^{bf}_{GDFC}(v_i)+1) =$$

$$2 \cdot d_{out}(v) + 2 + \sum_{i=1}^{d_{out}(v)} D^{bf}_{GDFC}(v_i) = D^{bf}_{GDFC}(v) + 1$$

Now suppose the binding pattern is bb, v and w are two arbitrary nodes and $d_{out}(v)$ is the out-degree v, where $v_1, \ldots, v_{d_{out}(v)}$ are its successor nodes. Let s=0 if v≠w and s=1, otherwise. Let us assume that v is a terminal node, i.e., $d_{out}(v)=0$. Lemma 5.47 implies that

$$D^{bb}_{SLD}(v,w) = 1 + s = D^{bb}_{GDFC}(v,w) + 1$$

Now suppose the result holds for all successor nodes of v. Thus

$$D^{bb}_{SLD}(v,w) = d_{out}(v) + 1 + s + \sum_{i=1}^{d_{out}(v)} D^{bb}_{SLD}(v_i,w) =$$

$$d_{out}(v) + 1 + s + \sum_{i=1}^{d_{out}(v)} (D^{bb}_{GDFC}(v_i,w)+1) = D^{bb}_{GDFC}(v,w) + 1$$

Next we consider the binding pattern fb. In the case where both variables in the input query are different, Lemma 5.49 suggests that

$$D^{bf}_{SLD}(X,Y) = n + 2 + \sum_{i=1}^{n} D^{bf}_{SLD}(w_i) =$$

$$n + 2 + \sum_{i=1}^{n} (D^{bf}_{GDFC}(w_i)+1) = D^{ff}_{GDFC}(X,Y) + 1$$

On the other hand, if both variables are the same, then

$$D^{ff}_{SLD}(X) = n + 2 + \sum_{i=1}^{n} D^{bb}_{SLD}(w_i,v_i) =$$

$$n + 2 + \sum_{i=1}^{n} (D^{bb}_{GDFC}(w_i,v_i)+1) = D^{ff}_{GDFC}(X) + 1$$

Finally, we analyze the binding pattern fb. As a consequence of Lemma 5.49 we compute

$$D^{fb}_{SLD}(v) = 2 + n + \sum_{i=1}^{n} D^{bb}_{SLD}(w_i,v) =$$

$$2 + n + \sum_{i=1}^{n} (D^{bb}_{GDFC}(w_i,v)+1) = D^{fb}_{GDFC}(v) + 1$$ ■

Thus, GDFC-resolution also is more efficient than SLD-resolution for the reflexive transitive closure. Since we did not assume anything about the structure of the underlying graph, this result holds for arbitrary directed acyclic graphs.

5.5 Summary

In this chapter we considered the efficiency of GDFC-resolution comparing it with SLD-resolution. We showed that corresponding success branches in GDFC- and SLD-trees have the same length (Corollary 5.4). We considered propositional programs and proved that GDFC- and SLD-trees contain the same number of success branches (Corollary 5.8), that the failure branches in GDFC-trees are not longer than the corresponding failure branches in SLD-trees (Corollary 5.16), and that GDFC-trees contain not more failure branches than SLD-trees (Corollary 5.17). However, the fact that the length and the number of the failure branches may be smaller in GDFC-trees than in SLD-

trees gives rise to the conjecture that GDFC-resolution on average is more efficient than SLD-resolution.

We considered tree-like, binary, propositional programs and analyzed the complexity of finding the first solution with SLD- and GDFC-resolution for certain subclasses. Even if GDFC-resolution is not always better than SLD-resolution it is worth applying GDFC-resolution, since it has a linear worst case complexity (Theorem 5.25). SLD-resolution, in contrast, may need exponentially many steps on average (Theorem 5.24). The reason for this significant improvement is that each fact in the program produces a solution for the root of the tree, so that no backtracking is needed using GDFC-resolution. SLD-resolution, however, possibly has to traverse the whole tree which contains exponentially many paths from the root to the leaves.

Furthermore we considered taxonomic hierarchies which can be implemented by binary datalog programs. We compared the efficiency of GDFC- and SLD-resolution for full, binary and balanced taxonomic trees which are closely related to the tree-like propositional programs. Accordingly, we showed that GDFC-resolution is optimal for computing the first solution for the root of the taxonomic tree; the number of inferences needed depends linearly on the depth of the tree (Theorem 5.36). In contrast to that SLD-resolution may need exponentially many steps. We then analyzed the complexity of computing all solutions for the root of the tree. We showed that GDFC-resolution is more efficient than SLD-resolution if the query has less than approximately $N/(c \cdot \log(N))$ solutions, where c is the unification overhead of GDFC-resolution and N is the number of arcs in the taxonomic tree. However, if the taxonomic hierarchy corresponds to an inverted tree, then SLD-resolution is better than GDFC-resolution if there are at least two solutions.

Finally we analyzed the complexity of both strategies for the well known procedures to compute the transitive respectively reflexive transitive closure of directed acyclic graphs. We proved that GDFC-resolution is always more efficient than SLD-resolution if we consider the size of the corresponding refutation trees (Theorems 5.44 and 5.50).

To sum up, the results obtained in this section imply that GDFC-resolution gives promising improvements with respect to the number of inferences needed to evaluate several types of propositional and even datalog programs. A comparison of the effective runtime behaviour of the meta-interpreters for pure Prolog and goal-directed forward chaining is the subject of the following chapter.

Chapter 6

EXPERIMENTAL RESULTS

The previous chapter presented theoretical results concerning the complexity of goal-directed forward chaining for several classes of propositional and datalog programs. The results suggest in the first instance, that GDFC-resolution on average is better than SLD-resolution for propositional programs, and in the second, that it may even be more efficient for some standard datalog programs. The aim of this chapter is to present the results of experimental studies made to find out the scale of the difference between SLD- and GDFC-resolution. The first section presents experimental results confirming the conjecture that GDFC-resolution is more efficient than SLD-resolution for propositional programs. The second section considers the ancestor relationship and the specification or the reflexive transitive closure of directed acyclic graphs. It demonstrates by means of experiments that the meta-interpreter for goal-directed forward chaining with respect to the effective runtime behaviour is more efficient than the interpreter for pure Prolog. This shows that GDFC-resolution is a serious alternative to SLD-resolution even in the context of datalog programs. The last section investigates the space saving obtained by the link clause optimization for propositional programs and its influence to the efficiency of both strategies. It presents experimental results indicating that the link clause optimization produces considerable space savings. A further interesting result is that the reordering of body literals has a positive effect to the efficiency of GDFC- and SLD-resolution.

6.1 Efficiency for Propositional Programs

In order to confirm the conjecture that GDFC-resolution on average is more efficient than SLD-resolution for propositional logic programs, we randomly generated a set of sample programs and counted the number of logical

inferences needed to find all solutions for a certain set of goals, i.e., we counted the number of arcs of corresponding SLD- and GDFC-trees. Clearly it is essential to consider a sufficiently large number of programs to obtain reliable results. As well as for analytical results, however, one could argue that some or even many of the programs occurring as samples are irrelevant in practice. One way to weaken this argument is to generate only such programs which are 'reasonable' from their syntactic structure.

First, we consider non-recursive programs only, because recursion is undesired and unnecessary in the context of propositional program; it leads to infinite loops on one hand and does not extend the expressiveness on the other hand. Second, we consider only programs without predicates whose solution requires too much redundant derivations. The reason for this restriction is, that it is worth using extensions such as lemma generation or even other resolutions strategies eliminating redundant derivations whenever there are a lot of them. Therefore, we allow to specify a limit defining the maximum number of inferences which may be needed to solve a goal.

For technical reasons we only consider programs which can be decomposed into a set of predicates defined by non-unit clauses only (intentional data base symbols or IDB symbols) and a set of predicates defined by unit-clauses only (extensional data base symbols or EDB symbols). Put another way, we do not allow that a procedure is simultaneously defined by facts and rules. Fortunately, it is possible to rewrite each program as an 'equivalent' program which can be decomposed w.r.t. the EDB and IDB symbols [4].

The syntactic structure of a propositional program additionally depends on the following parameters: the number of EDB and IDB symbols as well as the length and number of the rules used to define a procedure. Therefore, the program generator we have implemented allows us to adjust the following parameters:

- the number N_{EDB} of EDB symbols,
- the number N_{IDB} of IDB symbols,
- the distribution of the number of rules per IDB symbol,
- the distribution of the length of the rules and
- the limit for the number of inferences.

Whereas the first distribution specifies the probability that an IDB symbol is defined by a certain number of rules, the second defines the probability that a rule has a specific length. We considered three distinct samples consisting of

programs with normal, long and many rules. The specific values of the corresponding distributions are contained in Figure 6.1.

normal	1	2	3	4	5	6	7	8
rule length	0.1	0.5	0.3	0.08	0.02			
rule number	0.2	0.4	0.3	0.08	0.02			

long rules	1	2	3	4	5	6	7	8
rule length	0.15	0.2	0.2	0.15	0.12	0.09	0.06	0.03
rule number	0.2	0.4	0.3	0.08	0.02			

many rules	1	2	3	4	5	6	7	8
rule length	0.1	0.5	0.3	0.08	0.02			
rule number	0.15	0.18	0.2	0.18	0.13	0.09	0.05	0.02

Fig. 6.1: Distributions for normal, long and many rules

We proceeded as follows: for different ratios N_{EDB}/N_{IDB} we generated a sample of programs. For each program, we stepwise increased the number of facts, i.e., the number of EDB symbols set to true, and measured the size of the GDFC- and SLD-tree for each IDB symbol. We then computed the relative improvement R of GDFC-resolution w.r.t. SLD-resolution. Suppose $|T_{SLD}|$ is the size of the SLD- and $|T_{GDFC}|$ the size of the GDFC-tree, then R is computed by the following formula

$$R = \frac{|T_{SLD}|-|T_{GDFC}|}{|T_{SLD}|}$$

Figure 6.2 shows the mean values over 85.000 goals for N_{EDB}=40 and N_{IDB}=10 with a limit of 150 inferences. The X-axis represents the number of provable IDB symbols in percent and the Y-axis $\bar{R}$, the mean value of relative improvement in percent obtained using GDFC-resolution. The horizontal bars represent the interval of the standard deviation.

This shows that GDFC-resolution on average is more efficient than SLD-resolution in nearly the whole range. For example, if no IDB symbol is provable, then the speed-up is nearly 90%, and if 50% of the IDB symbols can be

solved, then GDFC-resolution needs about 25% fewer inferences than SLD-resolution to find all solutions.

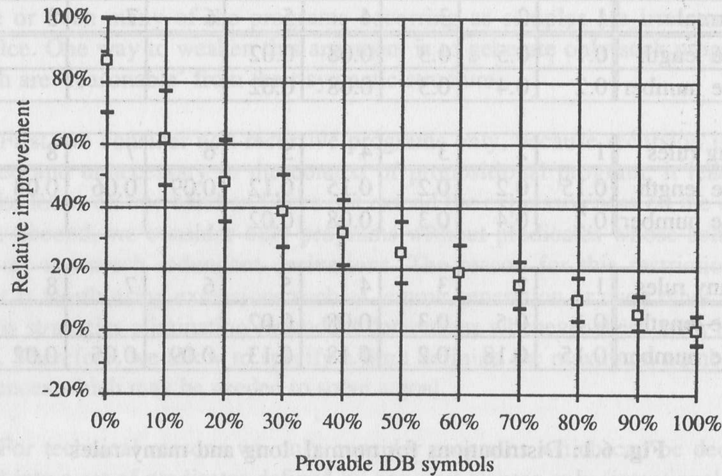

Fig. 6.2: Average relative improvement

The function describing the relationship between the number of provable IDB symbols and the number of facts has a slightly sigmoid shape. Figure 6.3 illustrates this dependency for the distribution normal with N_{EDB}=40 and N_{IDB}=10.

The X-axis represents the number of facts and the Y-axis the average number of provable IDB symbols in percent. Thus, if 50% of the EDB symbols are true, then on average 40% of the IDB symbols are provable and GDFC-resolution on average is 25% faster than SLD-resolution.

An interesting question concerning the function which describes the relative improvement is, how its shape changes if we modify the input parameters of our program generator. Here we have the quite interesting result that it is largely independent from the ratio N_{EDB}/N_{IDB} and different numbers N_{EDB} and N_{IDB} with a fixed ratio N_{EDB}/N_{IDB} (see Figures 6.4 and 6.5).

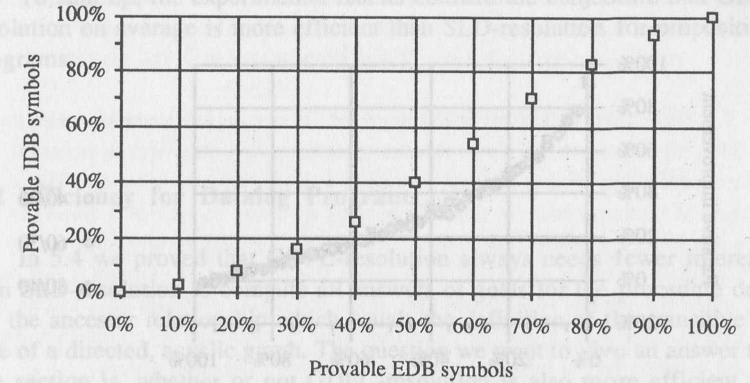

Fig. 6.3: Dependency between provable EDB and IDB symbols

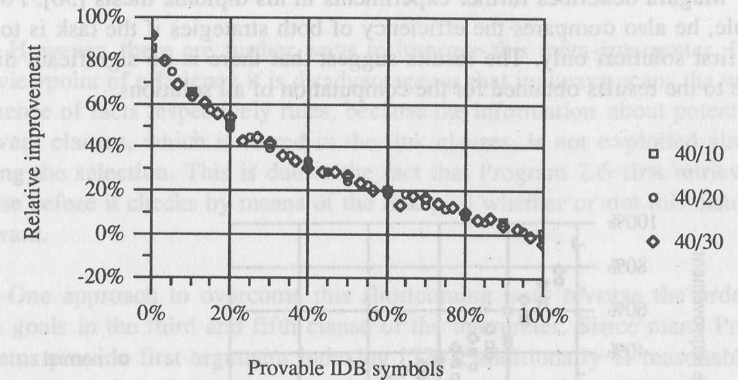

Fig. 6.4: Relative improvement depending on the ratio N_{EDB}/N_{IDB}

However, the improvement slightly depends on the distributions defining the expected length and number of rules. Figure 6.6 shows that the relative improvement increases with the number of rules and decreases with the length of the rules for the ratio N_{EDB}/N_{IDB}=40/20 and a limit of 250 inferences.

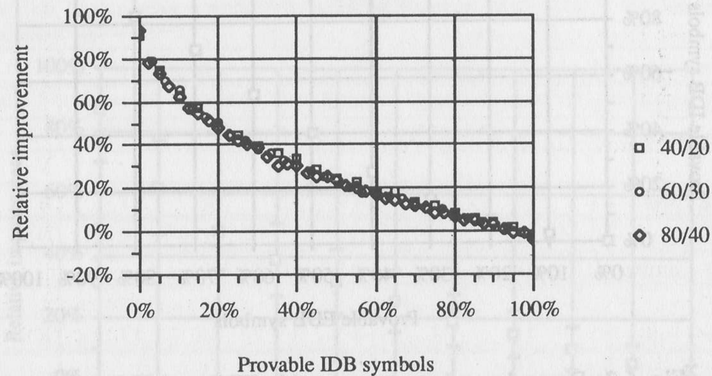

Fig. 6.5: Relative improvement depending on N_{EDB} and N_{IDB}

Magura describes further experiments in his diploma thesis [50]. For example, he also compares the efficiency of both strategies if the task is to find the first solution only. The results suggest that there is no significant difference to the results obtained for the computation of all solutions.

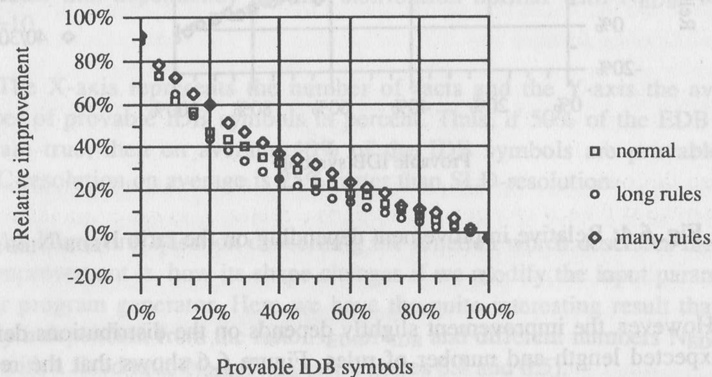

Fig. 6.6: Influence of the distributions to the relative improvement

To sum up, the experimental results confirm the conjecture that GDFC-resolution on average is more efficient than SLD-resolution for propositional programs.

6.2 Efficiency for Datalog Programs

In 5.4 we proved that GDFC-resolution always needs fewer inferences than SLD-resolution to compute all answers of goals for the procedure defining the ancestor relationship which equals the definition of the transitive closure of a directed, acyclic graph. The question we want to give an answer to in this section is, whether or not GDFC-resolution is also more efficient than SLD-resolution w.r.t. the effective runtime behaviour.

Since there is no efficient implementation of GDFC-resolution like Prolog for SLD-resolution, we compare the performance of the standard interpreter for pure Prolog with that of Program 2.6 which realizes a more efficient implementation of GDFC-resolution than Program 2.1 (see Section 2.2).

However, there are further ways to improve this meta-interpreter. From the viewpoint of efficiency it is disadvantageous that it always scans the whole sequence of facts respectively rules, because the information about potentially relevant clauses, which is stored in the link clauses, is not exploited already during the selection. This is due to the fact that Program 2.6 first retrieves a clause before it checks by means of the link goal whether or not this clause is relevant.

One approach to overcome this shortcoming is to reverse the order of both goals in the third and fifth clause of the interpreter. Since many Prolog systems provide first argument indexing [52] it additionally is reasonable to reverse the order of the arguments in the link clauses. However, using this scheme we still need two goals to retrieve a relevant clause. Fortunately there is a straightforward approach to transform the program so that a join of the clauses in P and the link clauses in $L_P^{fin,ref}$ is realized.

1) For each fact A← in P and each link clause link(B,C)← in $L_P^{fin,ref}$ with θ=mgu(A,B), P' contains the clause fact(C,A)θ←.

2) For each non-unit clause $A \leftarrow A_1,\ldots,A_n$ in P and each link clause link(B,C)← in $L_P^{fin,ref}$ with θ=mgu(A,B), P' contains the unit-clause rule(C,A,($A_1,\ldots,A_n$))←. If n=1 then we set n=2 and A_2=true.

A clause fact(A,B)← is interpreted as: 'B is true and possibly relevant for A'. This allows us to retrieve relevant clauses in one step and in constant time if the underlying Prolog system provides first argument indexing. The disadvantage of this approach is that P' contains as many copies of a clause in P as there are corresponding link clauses. However, we can assume that, in an efficient implementation of GDFC-resolution, the copying of clauses is not necessary since the access is supported by index structures realized on a low level of the implementation. A further speed-up may be obtained by extending this indexing scheme to the leftmost body literal of each non-unit clause.

The meta-interpreter needed to interpret the programs obtained by this transformation comprises Program 6.7.

```
gdfc_solve(true)←
gdfc_solve((A,B))←
        gdfc_solve(A),
        gdfc_solve(B)
gdfc_solve(G)←
        fact(G,F),
        subgoal(F,G)
subgoal(G,G)←
subgoal(F,G)←
        rule(G,G1,(F,B)),
        gdfc_solve(B),
        subgoal(G1,G)
```

Program 6.7: GDFC-meta-interpreter used for the benchmarks

Accordingly, we use a variant of the standard meta-interpreter for pure Prolog which comprises Program 6.8. To avoid the re-compilation of compiled clauses, we store all clauses in a binary relation cl/2 where the head of each clause is stored in first argument position. By means of first argument indexing possible input clauses can be accessed very fast.


```
solve(true)←
solve((A,B))←
        solve(A),
        solve(B)
solve(A)←
        cl(A,B),
        solve(B)
```

Program 6.8: SLD-meta-interpreter used for the benchmarks

To compare the efficiency of both interpreters, we randomly generated directed acyclic graphs following the constant density model which is one of the most frequently encountered probability models for random graphs [9]. For a fixed number n of nodes and a fixed density 0≤d≤1, each graph only contains arcs of the form (u,v) with u<v where each arc exists with probability d. We then measured the runtime needed to find all solutions of the following five goals:

←p(X,Y) ←p(X,X) ←p(0,n-1) ←p(0,X) ←p(X,n-1)

Figure 6.9 shows the average runtime $\bar{t}_{SLD}$ and $\bar{t}_{GDFC}$ in milliseconds obtained for the evaluation of these five goals with SLD- and GDFC-resolution for 100 randomly generated graphs with n=100 and d=0.05. This and all following benchmarks described in this section were carried out with Quintus Prolog 3.0 on a SUN 4/470 under SunOS Release 4.1. In Quintus Prolog, procedures are indexed on their first argument [60]. $\bar{D}$ denotes the mean value of the runtime difference, $|\bar{T}_{SLD}|$ and $|\bar{T}_{GDFC}|$ the average tree sizes and $\bar{S}$ the average number of solutions for the particular query.

In order to analyze the reasons for this improvement, which lies in the range between 45 and 55 percent, we have to consider the behaviour of Program 6.8. The problem is that it always scans the sequence of the arcs in e/2 twice, once in the first and once in the second p/2 clause. Program 6.7, in contrast, scans this relation only once and then attempts to apply one of the two rules in p/2. Whereas the first clause checks whether the edge found is a solution, the second clause is used to produce longer paths out of every edge. Thus, by using Program 6.7 we halve the number of selections.

	←p(X,Y)	←p(X,X)	←p(0,99)	←p(0,X)	←p(X,99)
$\overline{t}_{SLD}$	3264	3248	153	155	3236
$\overline{t}_{GDFC}$	1749	1458	69	86	1438
$\overline{D}$	1515	1790	84	69	1798
$\lvert\overline{T}_{SLD}\rvert$	10468	7852	380	498	7977
$\lvert\overline{T}_{GDFC}\rvert$	7850	5233	254	372	5358
$\overline{S}$	2617	0	6	124	125

Fig. 6.9: Runtime for the transitive closure

In order to obtain the same behaviour with the standard Prolog interpreter we have to specify the transitive closure based on the procedure defining the reflexive transitive closure (see Program 6.10) which results in a more efficient but less natural variant. Thus, goal-directed forward chaining, in contrast to Prolog, realizes an efficient evaluation of the natural specification of the transitive closure.

```
p(X,Z)←e(X,Y),q(Y,Z)
q(X,X)←
q(X,Z)←e(X,Y),q(Y,Z)
```

Program 6.10: Efficient implementation of the transitive closure

Figure 6.11 summarizes the results we obtained comparing the evaluation of Program 6.10 with SLD-resolution and the evaluation of the standard definition of the transitive closure with GDFC-resolution for 100 graphs of the same type as in the previous benchmark. Even if SLD-resolution always needs only one inference more than GDFC-resolution, the latter is considerably better for the second third and fifth query. A comparison of the efficiency of both strategies for the reflexive, transitive closure leads to similar values. The result of a sample consisting of 100 graphs with density 0.05 is illustrated in Figure 6.12. Whereas this effect has not been observed with LPA MacProlog 3.5 on an Apple Macintosh using the optimizing compiler [31], where SLD-resolution indeed is faster for the reflexive, transitive closure, it is stronger with the optimizing compiler of LPA Prolog Professional [48] on an IBM PC. Nevertheless, the benchmarks demonstrate that the possible unification over-

head of GDFC-resolution discussed in Section 5.3 indeed can be ignored for these examples.

	←p(X,Y)	←p(X,X)	←p(0,99)	←p(0,X)	←p(X,99)
$\overline{t}_{SLD}$	1761	1735	83	85	1733
$\overline{t}_{GDFC}$	1811	1477	73	91	1482
$\overline{D}$	-50	258	10	-6	251
$\lvert\overline{T}_{SLD}\rvert$	8127	5418	269	394	5541
$\lvert\overline{T}_{GDFC}\rvert$	8126	5417	268	393	5540
$\overline{S}$	2709	0	6	131	123

Fig. 6.11: Comparing the performance of SLD-resolution for the optimized transitive closure with that of GDFC-resolution for the standard version

	←p(X,Y)	←p(X,X)	←p(0,99)	←p(0,X)	←p(X,99)
$\overline{t}_{SLD}$	1714	1687	79	85	2028
$\overline{t}_{GDFC}$	1664	1399	68	81	1495
$\overline{D}$	50	288	11	4	533
$\lvert\overline{T}_{SLD}\rvert$	7880	5254	259	379	5382
$\lvert\overline{T}_{GDFC}\rvert$	7879	5253	258	378	5381
$\overline{S}$	2627	1	6	127	129

Fig. 6.12: Runtime for the reflexive, transitive closure

GDFC-resolution, however, is not always more efficient for datalog programs than SLD-resolution. Figure 6.14 shows the mean values taken for the same generation procedure over 100 directed acyclic graphs defined by parent/2 with 70 nodes and density 0.05. This procedure is defined in Program 6.13. Again we have the situation that SLD-resolution always needs one inference more than GDFC-resolution, but now SLD-resolution on average is about 6% faster than GDFC-resolution.

```
sg(X,X)←
sg(X,Y)←
    parent(X,Xp),
    sg(Xp,Yp),
    parent(Y,Yp).
```

Program 6.13: The same generation procedure

	←sg(X,Y)	←sg(X,X)	←sg(0,69)	←sg(0,X)	←sg(X,69)
$\overline{t}_{SLD}$	4404	3774	252	303	3510
$\overline{t}_{GDFC}$	4739	4092	269	327	3728
$\overline{D}$	-335	-318	-17	-24	-218
$\lvert\overline{T}_{SLD}\rvert$	13440	10022	633	824	9322
$\lvert\overline{T}_{GDFC}\rvert$	13439	10021	632	823	9321
$\overline{S}$	4119	702	0	191	1

Fig. 6.14: Runtime for the same generation procedure

6.3 Effectivity of the Link Clause Optimization

In this section we are going to analyze the degree of the space saving obtained by our approach to reduce the number of link clauses (Algorithm 3.33). For that purpose we again used the program generator described in the first section of this chapter. We generated a set of sample programs and counted the number of link clauses before and after the optimization. Figure 6.15 illustrates how the number of link clauses and rules depends on the number N_{IDB} of IDB symbols, if N_{EDB}=100. We used the normal distribution for the rule length and rule number.

For example, if N_{IDB}=200 then we have about 460 rules and 1500 link clauses on average. However, if we apply Algorithm 3.33 then the average number of link clauses reduces to fewer than 650, so that we have a space saving of more than 55%. This indeed is a considerable reduction of the required space. If we consider that the number of IDB symbols is a lower bound

for the optimum number of link clauses, then the average number of link clauses obtained by Algorithm 3.33 must be closer to the optimum than to the average number obtained without optimization.

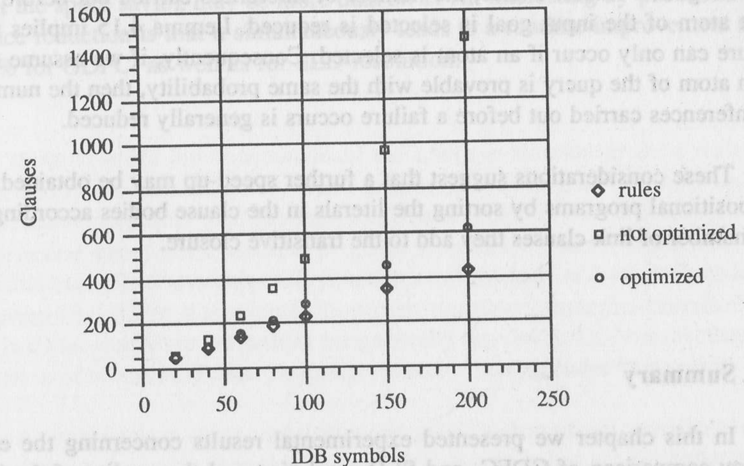

Fig. 6.15: Average number of link clauses before and after optimization

An important by-product of Algorithm 3.33 is that the programs produced as output often are more efficient than the original ones. Magura demonstrates that the evaluation of the resulting programs may require about 10% fewer inferences. This holds not only for GDFC-resolution but also for SLD-resolution [50]. There may be different reasons for this speed-up. One of them is easy to understand, if we remember the basic idea of Algorithm 3.33: we always select such literals which produce the fewest link clauses in the transitive closure. The size of the transitive closure, however, largely depends on the length of the paths in the set of link clauses. Thus the reduction of the number of link clauses in the transitive closure corresponds to the selection of literals producing short paths.

If we consider the behaviour of SLD-resolution in the context of propositional programs, then the only source of failure is that there is no fact for the selected EDB symbol. Because the link clause optimization reduces the length

of the derivation up to the first goal the leftmost literal of which is an EDB symbol, a failure occurs earlier in the derivation.

For GDFC-resolution the link clause optimization moves such literals to the leftmost position which only need a small number of iterations of the second subgoal/2 clause. Hence, the number of inferences carried out before the next atom of the input goal is selected is reduced. Lemma 4.15 implies that failure can only occur if an atom is selected. Consequently, if we assume that each atom of the query is provable with the same probability, then the number of inferences carried out before a failure occurs is generally reduced.

These considerations suggest that a further speed-up may be obtained for propositional programs by sorting the literals in the clause bodies according to the number of link clauses they add to the transitive closure.

6.4 Summary

In this chapter we presented experimental results concerning the efficiency comparison of GDFC- and SLD-resolution and the quality of the link clause minimization procedure. The results for propositional programs confirm the conjecture that GDFC-resolution on average is more efficient than SLD-resolution. The improvements are promising, especially if the amount of deducible information is relatively small. This is largely due to the results presented in Section 5.1 where we showed that GDFC-trees contain fewer and smaller failure branches than their corresponding SLD-trees.

Concerning the computation of the transitive of directed acyclic graphs we presented experiments demonstrating that the meta-interpreter for GDFC-resolution may be nearly twice as fast as the standard three-clause interpreter for pure Prolog. But even if we specify the transitive closure so that SLD-resolution only needs one inference more than GDFC-resolution for each query, then the meta-interpreter for pure Prolog remains slower than the meta-interpreter for goal-directed forward chaining. We obtained similar results for the reflexive transitive closure. Even if there are procedures for which SLD-resolution is better than GDFC-resolution, the results are very promising, since they demonstrate that the intrinsic unification overhead of GDFC-resolution sometimes can be ignored. Moreover, they give rise to expect that GDFC-resolution may be very efficient for a possibly large class of datalog programs.

Finally we analyzed the efficiency of our approach to minimize the number of link clauses for propositional programs. The experiments demonstrate that the space saving may be more than 50%. An interesting by-product of this space reduction is that it simultaneously leads to a runtime improvement up to 10% for GDFC- as well as for SLD-resolution.

Chapter 7

Termination of GDFC-Resolution

If we identify the termination of the query evaluation process with the finiteness of the refutation tree, then it is generally undecidable in the presence of recursive programs with function symbols whether or not the evaluation of a particular query for a definite program terminates. This follows from the fact that every Turing computable function can be computed by a definite logic program [3, 73]. In this context, however, top-down strategies benefit from the fact that procedure definitions are generally top-down. This means, that the structure of terms simplifies top-down, whereas it complicates bottom-up.

An exception to this is the somewhat artificial bottom-up specification of the digits given with Program 7.1. In this example bottom-up procedures terminate, while top-down approaches generally go into an infinite loop. Cohen and Feigenbaum [16] as well as Kifer and Lozinskii [36] use similar examples to disuss the pros and cons of forward and backward chaining.

```
digit(X)←
      digit(s(X))
digit(s(s(s(s(s(s(s(s(s(0))))))))))←
```

Program 7.1: Bottom-up specification of the digits

Bottom-up specified procedures, which generally have a finite fixed point, do not occur frequently in practice. Moreover, most of the recursive procedures, such as the append procedure, have an infinite fixed point and are specified top-down. Yamamoto and Tanaka [90] already mentioned, that their approach sometimes does not terminate. One therefore could speculate that GDFC-resolution as well as other bottom-up approaches cannot deal with top-down specified procedures.

Our goal in this chapter is to develop a variant of GDFC-resolution which always terminates when SLD-resolution does. Since the structure and finiteness of the refutation trees largely depends on the computation rule we restrict attention to the left-first computation rule in this chapter. A further precondition is that GDFC-resolution uses the link clause program L_P instead of a finite link clause program L_P^{fin}. Since L_P may be infinite, we have to compute the transitive closure $Link_P$ dynamically at runtime. Following the definition of Yamamoto and Tanaka this can be done by Program 7.2. Consequently, we have to replace the link goals in our meta-interpreter by bottom_up_link/2 goals which are evaluated after adding the link clauses in $Link_P$.

```
bottom_up_link(S,G)←
    link(S,G)
bottom_up_link(S,G)←
    link(S,S1),
    bottom_up_link(S1,G)
```

Program 7.2: Bottom-up computation of the transitive links

The disadvantage of this definition is that it computes all atoms for which a particular atom may be relevant for - and there may be infinitely many of them - in a bottom-up manner. At runtime the first argument of each call of the bottom_up_link/2 procedure is bound, so that terms occurring in the first argument of the recursive bottom_up_link/2 call are generally more complicated for top-down specified procedures. Consequently the termination of the evaluation of bottom_up_link/2 goals cannot be guaranteed for programs such as the append procedure.

```
top_down_link(S,G)←
    link(S,G)
top_down_link(S,G)←
    link(S1,G),
    top_down_link(S,S1)
```

Program 7.3: Top-down computation of transitive links

If we consider the fact that at runtime also the second argument of each bottom_up_link/2 goal is bound, namely to the actual goal, then this problem can be solved easily. The idea is to compute all atoms that are possibly relevant

for the goal in a top-down manner. This approach, which is declaratively equivalent to the bottom-up approach, is implemented with Program 7.3.

LEMMA 7.4 (Termination of top_down_link/2):

Let P be a program and A be an atom such that the SLD-tree for $P \cup \{\leftarrow A\}$ w.r.t. the left-first computation rule is finite. Then each goal for the procedure top_down_link/2 where the second argument is bound to A terminates too. ■

Proof: It follows from the definition of top_down_link/2 that the evaluation of each goal where the second argument is bound to A equals that part of the SLD-derivation from the root to the first derivation where the derived goal is shorter than its predecessor. Since the SLD-tree for $P \cup \{\leftarrow A\}$ is finite, there are no infinite derivations so that the top-down computation of the transitive links also terminates. ■

```
gdfc_solve(true)←
gdfc_solve((A,B))←
    gdfc_solve(A),
    gdfc_solve(B)
gdfc_solve(B)←
    clause(B,true)
gdfc_solve(B)←
    clause(A,true),
    top_down_link(A,B),
    subgoal(A,B)
subgoal(A,B)←
    clause(B,(A,Body)),
    gdfc_solve(Body)
subgoal(A,B)←
    clause(C,(A,Body)),
    top_down_link(C,B),
    gdfc_solve(Body),
    subgoal(C,B)
```

Program 7.5: A variant of the meta-interpreter based on top-down links

A disadvantage of this top-down computation is that it possibly does not terminate for bottom-up specified procedures such as Example 7.1. In this case the goal ←top_down_link(digit(s(0)),digit(0)) first produces the answer 'yes' and then loops forever. From our point of view this is no important disadvantage since, as already mentioned, bottom-up specified procedures occur rarely in practical applications. In this chapter we consider a variant of Program 2.6 which calls top_down_link/2 goals instead of link/2 goals. This meta-interpreter is defined in Program 7.5.

THEOREM 7.6 (Termination of GDFC-Resolution):

Suppose P is a definite program and G is a goal. Suppose SLD- and GDFC-resolution both apply the left-first selection function. If the evaluation of P∪{G} with SLD-resolution terminates then it also terminates with GDFC-resolution using the top-down computation of possibly relevant atoms. ■

Proof: To prove this theorem we show that the meta-interpreter for goal-directed forward chaining based on top-down computed transitive links (Program 7.5) cannot go into an infinite loop if the meta-interpreter for pure Prolog (Program 2.2) does not.

Since SLD-resolution terminates, every top_down_link/2 goal occurring in a derivation terminates. Consequently, the procedures gdfc_solve/1 and subgoal/2 are the only possible source of non-termination. Our goal is to show that there is a corresponding recursive call of the solve/1 procedure for each recursive call of the gdfc_solve/1 respectively subgoal/2 procedure.

Suppose A is an atom and ←gdfc_solve(A) is the actual goal. We consider such recursive calls of gdfc_solve/1 which come from either $n-1 \geq 0$ applications of the sixth followed by one application the fifth clause, or $n \geq 1$ applications of the sixth clause. Let $C = B \leftarrow B_1, \ldots, B_m$ be the input clause used in the n-th step and suppose the recursive call is $\leftarrow$gdfc_solve$(B_2, \ldots, B_m)\theta$.

If we have the first situation then A and B are unifiable so that, starting with ←solve(A), we reach $\leftarrow$solve$(B_1, \ldots, B_m)\gamma$ after a finite number of steps, since SLD-resolution terminates. Next B_1 is selected. Because $B_1\theta$ is a logical consequence of P and SLD-resolution terminates we after a finite number of steps derive the goal $\leftarrow$solve$(B_2, \ldots, B_m)\sigma$ with the property that $(B_2, \ldots, B_m)\theta$ is an instance of $(B_2, \ldots, B_m)\sigma$.

Now suppose we have the second situation. Hence there must be a transitive link from B to A. Since SLD-resolution terminates, we need a finite number of steps to derive $\leftarrow$solve$(B_1, \ldots, B_m)\gamma$ using the clauses also used to construct the transitive link. Clearly $B_1\theta$ is an instance of $B_1\gamma$. Consequently, there is an answer σ for $\leftarrow$solve$(B_1)\gamma$ such that $(B_2, \ldots, B_m)\theta$ is an instance of $(B_2, \ldots, B_m)\sigma$.

Thus, in both situations there is a recursive call $\leftarrow$solve$(B_2, \ldots, B_m)\sigma$ such that $(B_2, \ldots, B_m)\theta$ is an instance of $(B_2, \ldots, B_m)\sigma$. Consequently, for each recursive call of gdfc_solve/1 there is a corresponding recursive call of solve/1.

Let us now consider the recursive subgoal/2 procedure and suppose the non-termination is caused by a sequence of recursive calls of this relation. Starting with the goal $\leftarrow$subgoal$(B,A)\theta_0$ where B← is a fact in P, for which there is a transitive link to A producing the answer θ_0, all recursive calls have the form $\leftarrow$subgoal$(B_i, A\theta_i)$, $i \geq 1$, where $A\theta_i$ is more specific than $A\theta_{i-1}$ and each B_i is a logical consequence of P. Because SLD-resolution terminates applying the left-first computation rule and since there is a transitive link from each B_i to A, it must be possible to derive a goal $\leftarrow$solve(A_i) where A_i is an instance of B_i for each B_i. This can be done using the non-unit clauses in P used to construct the corresponding transitive link. Consequently, if there is an infinite sequence of recursive subgoal/2 calls then there also must be an infinite sequence of solve/1 calls.

Consequently, if the evaluation of ←solve(A) terminates, then the evaluation of ←gdfc_solve(A) using Program 7.5 also terminates. ■

With respect to this termination behaviour, there is a strong parallel to the system graph approach proposed by Kifer and Lozinskii in [36]. Since the filters used to select relevant clauses are pushed top-down through the system graphs, this approach terminates for many top-down specified procedures.

From the previous theorem it follows that the termination of SLD-resolution is a sufficient condition for the termination of GDFC-resolution using top-down computed transitive links. The following example demonstrates that it is not admissible to replace this implication by an equivalence, i.e., the termination of GDFC-resolution is not a sufficient criterion for the termination of SLD-resolution.

EXAMPLE 7.7 (Termination of SLD- and GDFC-Resolution):

Suppose P consists of only one clause, namely

$$p \leftarrow p$$

and suppose G is $\leftarrow p$. Whereas every SLD-tree for $P \cup \{G\}$ contains an infinite branch, every GDFC-tree is finite since P contains no fact. ■

This leads to the following theorem.

THEOREM 7.8 (Termination of GDFC- and SLD-Resolution):

If we apply the left-first computation rule, then the class of programs for which SLD-resolution terminates is a proper subset of the class for which GDFC-resolution using top-down computed links terminates. ■

Proof: The result follows from Theorem 7.6 and Example 7.7. ■

An interesting by-product of Theorem 7.6 is that termination proofs for SLD-resolution can be adopted for GDFC-resolution. We explain this at the example of the automatic termination proof technique for Prolog recently developed by Plümer in [57, 58, 59]. His approach can be applied to definite programs which are well-moded, normalized and free of mutual recursion. A program is normalized if no variable occurs more than once in a literal. Whereas the last two properties are syntactic restrictions only, the well-modedness is a dynamic property. Plümer adopted the definition of well-modedness from Dembinski and Maluszynski [18]. Well-modedness informally means, that a certain set of input arguments is guaranteed to be ground before a procedure is called, and that a set of further arguments is guaranteed to be ground after a successful evaluation of a procedure. Thus, the well-modedness always is only valid for a fixed and static computation rule (for example for Prolog's left-first strategy).

The basic idea of Plümer's approach is to prove that the size of the terms occurring in certain argument positions of recursive body literals always is smaller than in the head of the clause. The size of the terms is measured by linear term norms. In the context of bottom-up evaluation the presence of

termination proofs implies that the size of the terms in the head always must be greater than in the recursive body literals.

THEOREM 7.9 (Termination Proofs for GDFC-Resolution):

Suppose P is well-moded, normalized and free of mutual recursion and G is a goal satisfying the modes. Suppose there is a termination proof for each procedure occurring in G. Then the evaluation of $P \cup \{G\}$ with GDFC-resolution using the left-first computation rule and top-down computed transitive links terminates. ■

Proof: Since SLD-resolution terminates for $P \cup \{G\}$, GDFC-resolution terminates too. ■

The price for this termination behaviour of GDFC-resolution for top-down specified programs is that we have to compute the necessary link clauses of the transitive closure dynamically at runtime. In order to optimize the dynamic computation of the transitive links it clearly is possible to use memo tables. But even if we store solutions of top_down_link/2 goals and reuse them when they are needed this remains a time consuming process.

An interesting question is whether there is any approach which allows us to use L_P^{fin} instead of L_P for top-down specified programs. Unfortunately, using L_P^{fin} the termination of SLD-resolution is no longer a sufficient condition for the termination of GDFC-resolution. This has two reasons: first the goals derived with GDFC-resolution w.r.t. L_P^{fin} are possibly more general than those derived with SLD-resolution and second there may be infinitely many atoms which are relevant for the selected element.

EXAMPLE 7.10 (Non-Termination of GDFC-Resolution using L_P^{fin}):

Suppose P is

nat(0)←
nat(s(X))←nat(X)

L_P^{fin} is

link(nat(X),nat(s(Y)))←

and G is $\leftarrow$nat(s(0)). Whereas SLD-resolution terminates, GDFC-resolution goes into an infinite loop repeatedly calling subgoal/2 goals after giving the answer 'yes'. The reason is that each atom $nat(s^n(0))$ ($n \geq 0$) is a possibly relevant subgoal for nat(s(0)). Thus we can eliminate the body literal of the second nat/1 clause by each of these atoms since the resulting head $nat(s^{n+1}(0))$ also is possibly relevant. ■

One solution to this problem could be to exploit the bounded term size property which says that there is a function f(n) such that, for each input with size n, no terms with size greater than f(n) occur during the evaluation. Since each program for which there is a termination proof satisfies the bounded term size property [58, 84], the size of the atoms possibly relevant for the goal must be limited. Thus, if termination proofs are available, the extension of the meta-interpreter which forces a failure whenever the linear predicate inequalities are violated guarantees the termination. The approach described in Heidelbach's diploma thesis [30], which is based on dynamic size checks, builds a first step towards this direction.

For bottom-up defined procedures, however, we still can use L_P^{fin} instead of L_P, because the termination of the derivation is guaranteed by the simplification of the terms from the recursive body literals to the head.

The main disadvantage of the approaches discussed above is that they require additional operations at runtime, which both, the dynamic computation of the transitive links and the operations needed to stop the inference if the size of the derived terms exceeds a certain level, are very time consuming. Practical experiments show that indeed this overhead decreases the efficiency of GDFC-resolution so that SLD-resolution is generally more efficient (if it terminates). The optimization of these approaches or the development of alternative strategies to improve GDFC-resolution for recursive programs containing function symbols is an important field of future research.

In this context it is worth mentioning that Theorem 7.6 also holds if L_P is finite and we use L_P instead of top-down computed transitive links. Consequently, if L_P is finite the meta-interpreter for goal-directed forward chaining (Program 2.1) terminates whenever the meta-interpreter for pure Prolog (Program 2.2) does. Thus, especially in the context of datalog programs the efficiency of GDFC-resolution is combined with an improved termination behaviour.

Motivated by the fact that the class of programs for which SLD-resolution terminates is a proper subset of the class of programs for which GDFC-resolution terminates, one now could ask if it is easier to decide whether GDFC-resolution goes into an infinite loop than it is for SLD-resolution. Clearly in the context of definite programs this problem is undecidable. However, in the context of propositional programs there are problems concerning the termination of the query evaluation process which are decidable. Let us consider the following problem denoted as the special loop-test problem:

DEFINITION 7.11 (Special Loop-Test Problem for SLD-Resolution):

Input: A be a propositional program P and a goal G.

Question: Does a depth-first search through the SLD-tree for $P \cup \{G\}$ coming from the left-first computation rule go into an infinite loop without failing finitely or reaching a success node? ■

Kleine Büning, Löwen and Schmitgen addressed this problem in [38] where they showed that it is decidable in O(n) where n is the number of clauses in P. The following lemma addresses the special loop-test problem for GDFC-resolution.

LEMMA 7.12 (Special Loop-Test Problem for GDFC-Resolution):

Let P be a propositional program and G be a goal. Suppose we apply the left-first computation rule. Then it is decidable in polynomial time whether or not a depth-first search through the GDFC-tree for $P \cup \{G\}$ w.r.t. L_P goes into an infinite loop before it finds the first node containing the empty clause or finitely fails. ■

Proof: To show the result we modify the interpreter for goal-directed forward chaining (Program 2.1) so that each clause in P is used at most m+1 times as input clause where m is the number of predicate symbols in P.

For that purpose we introduce an extension table, similar to that one used in the ET algorithm [19], containing two lists. The first list contains the elements selected but not solved in the branch currently traversed. The second list contains the elements already evaluated together with their truth values. Whenever we select an element not contained in any of the two lists we simply

add it to the list of selected elements and proceed as usual. At the moment where it is solved or even failed we remove it from the list of selected elements and add it together with its truth value to the list of evaluated elements. Since left-first computation rule is a local selection rule [86] which always selects one of the most recently introduced literals, GDFC-resolution is in a loop and we can stop the derivation reporting that GDFC-resolution loops, if we select an element which is contained in the list of selected elements. Suppose we select an element contained in the list of evaluated goals. If its status is true, then nothing remains to be done with it and we proceed with the next goal. Otherwise, if its status is false then we force backtracking. Hence, if the evaluation of the input query with this approach succeeds or finitely fails then the depth-first search through the GDFC-tree cannot go into an infinite loop.

Let us now consider the complexity of this loop-checking. Suppose n ($m \leq n$) is the number of clauses in P. Since we store the result of the evaluation of each selected element in the extension table, every element is evaluated at most once. Each non-unit clause can be used as input clause for at most m subgoal-goal pairs. In the case where the head of the clause unifies with the right atom of the subgoal-goal pair we, supposed that L_P contains the corresponding link clause, can apply Rule 3 as well as Rule 4. The same holds for the unit clauses in P: there are at most m atoms each unit clause may be relevant for. However, if the fact and the selected atom are the same and L_P contains a link clause where this atom occurs on both sides then we can apply Rule 1 as well as Rule 2. Consequently each clause is used at most m+1 times as input clause. Consequently, the complexity of the special loop-test problem for GDFC-resolution is bounded by $O(n^2)$. ■

Kleine Büning et al. furthermore considered the following problem denoted as the general loop-test problem:

DEFINITION 7.13 (General Loop-Test Problem for SLD-Resolution):

Input: A be a propositional program P without facts.

Question: Is there a set of facts F and a goal $\leftarrow A$ such that a depth-first search through the SLD-tree for $P \cup F \cup \{\leftarrow A\}$ coming from the left-first computation rule goes into an infinite loop without failing finitely or reaching a success node? ■

By a reduction of the 1-in-3SAT Problem Kleine Büning et al. showed that the general loop-test problem is NP-complete for SLD-resolution in the context of propositional programs [38]. The following theorem shows that it is also NP-complete for GDFC-resolution.

THEOREM 7.14 (General Loop-Test Problem for GDFC-Resolution):

Let P be a propositional program and G be a goal. Suppose we apply the left-first computation rule. Then it is NP-complete to decide whether there is a set F of facts and an atom A such that a depth-first search through the GDFC-tree for $P \cup F \cup \{\leftarrow A\}$ w.r.t. L_P goes into an infinite loop before it finds the first success branch or fails finitely. ■

In order to prove that the general loop-test problem is NP-hard we borrow the idea of Kleine Büning et al. to reduce the 1-in-3SAT problem of propositional calculus formulas which unfortunately is NP-hard [65].

DEFINITION 7.15 (1-in-3SAT-Problem):

Input: A propositional formula φ in conjunctive normal form consisting of n clauses with three literals per clause where each literal is positive.

Question: Is there a truth assignment τ for φ such that exactly one literal is true in each clause? ■

Proof of Theorem 7.14: First it is easy to verify that general loop-test problem is in NP. We simply choose A and F non-deterministically and solve the special loop-test problem for $P \cup F \cup \{\leftarrow A\}$. Lemma 7.12 implies that this can be done by a polynomially bounded algorithm.

Let $\varphi = (A_{1,1} \vee A_{1,2} \vee A_{1,3}) \wedge \ldots \wedge (A_{n,1} \vee A_{n,2} \vee A_{n,3})$ with $A_{i,j}$ positive literals be given. Let $s, s_0, \ldots, s_n$ be n+2 distinct symbols not occurring in φ. We now construct the corresponding program P_φ in the following way: for each clause $(A_{i,1} \vee A_{i,2} \vee A_{i,3})$ $(1 \leq i \leq n)$, P_φ contains a procedure

$s_{i-1} \leftarrow A_{i,1}, A_{i,2}$	C_1
$s_{i-1} \leftarrow A_{i,1}, A_{i,3}$	C_2
$s_{i-1} \leftarrow A_{i,2}, A_{i,1}$	C_3
$s_{i-1} \leftarrow A_{i,2}, A_{i,3}$	C_4
$s_{i-1} \leftarrow A_{i,3}, A_{i,1}$	C_5
$s_{i-1} \leftarrow A_{i,3}, A_{i,2}$	C_6
$s_{i-1} \leftarrow A_{i,1}, s_i$	C_7
$s_{i-1} \leftarrow A_{i,2}, s_i$	C_8
$s_{i-1} \leftarrow A_{i,3}, s_i$	C_9

Furthermore P_φ contains the clauses

$$s_n \leftarrow s, s_0$$
$$s \leftarrow$$

Clearly this construction can be done by a polynomially bounded algorithm. We now show that $\varphi \in$ 1-in-3SAT if and only if there is a goal G and a set of facts F such that the depth-first search through the GDFC-tree for $P_\varphi \cup F \cup \{G\}$ via the left-first computation rule and w.r.t. L_{P_φ} goes into an infinite loop before it finitely fails or reaches a leaf containing the empty clause.

Let us first consider the link clause program L_{P_φ} for P_φ. For each $i \in \{1,\ldots,n\}$ and each $j \in \{1,\ldots,3\}$ it contains one link clause $link(A_{i,j}, s_{i-1}) \leftarrow$. Furthermore, it contains the clause $link(s, s_n) \leftarrow$.

Now suppose there is a truth assignment $\tau = \{A_1, \ldots, A_m\}$ $(1 \le m \le n)$ for φ such that exactly one literal is true in each clause of φ. Then we set $A := s_0$ and $F := \{A_1 \leftarrow, \ldots, A_m \leftarrow\}$.

Let us consider the GDFC-derivation of a goal $\leftarrow s_{i-1}$, $1 \le i \le n$. Since τ is a truth assignment which contains exactly one literal from each clause there can be exactly one $j \in \{1,\ldots,3\}$ with the property that $A_{i,j} \leftarrow$ is in F. Since $link(A_{i,j}, s_{i-1})$ is true, the root node $\leftarrow s_{i-1}$ of the GDFC-tree for $P_\varphi \cup F \cup \{\leftarrow s_{i-1}\}$ has exactly one successor node, namely $\leftarrow <A_{i,j}, s_{i-1}>$. Since the clauses are scanned top-down, this node has exactly three successor nodes coming from the clauses $C_{2 \cdot j-1}$, $C_{2 \cdot j}$ and C_{6+j} (from left to right) of the procedure defining s_{i-1}. Since only one literal is true in each clause the successor nodes coming from the first two clauses directly lead to failure. However, using C_{6+j} we derive the goal $\leftarrow s_i$. Figure 7.16 shows the first three levels of this GDFC-tree for j=2.

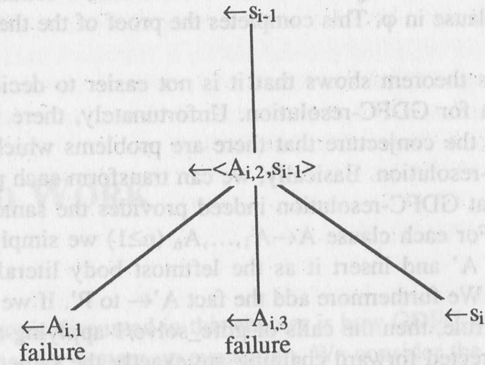

Fig. 7.16: The first three levels of a GDFC-tree for $P_\varphi \cup F \cup \{\leftarrow s_{i-1}\}$

Let us now consider what happens if the actual goal is $\leftarrow s_n$. The only fact relevant for s_n is $s \leftarrow$. Consequently we apply Rule 2 which results in the goal $\leftarrow <s_n, s>$. Next we can apply only Rule 3 using the clause $s_n \leftarrow s, s_0$ to derive the goal $\leftarrow s_0$.

Consequently, if τ is a truth assignment such that exactly one literal is true in every clause, then a depth-first search through the GDFC-tree for $P_\varphi \cup F \cup \{\leftarrow s_0\}$ goes into an infinite loop.

Now suppose there is a goal $\leftarrow A$ and a set of facts F such that a depth-first search through the GDFC-tree for $P_\varphi \cup F \cup \{\leftarrow A\}$ goes into an infinite loop before it reaches a success branch or finitely fails. Clearly the answer for a refutation of $P_\varphi \cup F \cup \{\leftarrow A\}$ for each $A \notin \{s_0, s_1, \ldots, s_n\}$ would always be 'yes' or 'no', since the corresponding GDFC-tree is finite. Therefore, let us assume that $A \in \{s_0, s_1, \ldots, s_n\}$. If F is $\{A_1 \leftarrow, \ldots, A_m \leftarrow\}$ then we define the truth assignment as $\tau := \{B \mid B$ occurs in φ and $B \leftarrow$ is contained in F$\}$.

Suppose A is s_n. In this case we use the clauses $s \leftarrow$ and $s_n \leftarrow s, s_0$ to derive $\leftarrow s_0$. Now suppose A is s_{i-1}, for an arbitrary $i \in \{1,\ldots,n\}$. Clearly we only go into an infinite loop using one of the clauses C_7, C_8 and C_9. Consequently, there must be at least one $j \in \{1,2,3\}$ for each $i \in \{1,\ldots,n\}$ such that $A_{i,j} \leftarrow$ is contained in F. Since we scan the clauses top-down, it cannot be possible that one of the clauses $C_1, \ldots, C_6$ can be evaluated successfully. Otherwise, the evaluation of $\leftarrow s_{i-1}$ would succeed. Hence, there can be at most one $j \in \{1,2,3\}$ for

each $i \in \{1,\ldots,n\}$ such that $A_{i,j} \leftarrow$ is contained in F. Thus τ contains exactly one atom from each clause in φ. This completes the proof of the theorem. ■

The previous theorem shows that it is not easier to decide the general loop-test problem for GDFC-resolution. Unfortunately, there is a strong argument opposing the conjecture that there are problems which are easier to decide for GDFC-resolution. Basically, we can transform each program P to a program P' so that GDFC-resolution indeed provides the same behaviour as SLD-resolution. For each clause $A \leftarrow A_1,\ldots,A_n$ $(n \geq 1)$ we simply introduce an auxiliary symbol A' and insert it as the leftmost body literal which yields $A \leftarrow A', A_1,\ldots,A_n$. We furthermore add the fact $A' \leftarrow$ to P'. If we apply the left-first computation rule, then the calls of gdfc_solve/1 applying the meta-interpreter for goal-directed forward chaining are exactly the same as the calls of solve/1 applying the standard interpreter for pure Prolog. In Section 3.1 we already demonstrated that, using this approach, GDFC-resolution simulates the behaviour of SLD-resolution. If we compare the proof above with that for SLD-resolution given in [38] then s is such an auxiliary predicate. Thus, the introduction of auxiliary predicates seems to be a good and general means to show that decision problems such as the general loop-test problem have the same complexity for GDFC- and SLD-resolution.

Chapter 8

RELATED WORK

The first topic discussed in this chapter is how GDFC-resolution relates to other approaches for bottom-up evaluation. We consider the field of deductive databases and analyze the applicability and efficiency of GDFC-resolution for this area. The second section concerns an approach to choose the optimal rule direction for LLNR-resolution.

8.1 Deductive Databases

The design and implementation of efficient inference algorithms, which are more efficient than resolution for large programs, is one of the most important research areas in the field of deductive database systems [24, 82]. Gallaire [23] comprehensively discusses the notion of deductive databases which are motivated by the desire to integrate database technology and logic. Possible application areas of deductive databases deal with massive amounts of data and need a query facility more powerful than that of typical query languages [79, 81].

One rough classification of the query processing strategies (see [4] for an excellent overview) is based on the direction of the rule application: top-down vs. bottom-up. Many of the top-down approaches such as QRGT [81], OLD-resolution [76] and ET* [19], which is similar to QSQR [4], are an extension of SLD-resolution; they store encountered goals and computed answers and re-use them when they are needed. These strategies avoid redundant derivations and, in contrast to Prolog, are complete for datalog programs.

Most of the bottom-up approaches are based on the 'semi-naive' fixed point procedure which likewise is an optimized version of the 'naive' strategy. The naive approach for definite logic programs, realized by Algorithm 8.2,

implements the mapping T_S defined on S-Herbrand interpretations which are introduced in [20, 45]. In contrast to the standard definition of Herbrand interpretations which are based on ground atoms, S-Herbrand interpretations possibly contain non-ground atoms. Accordingly, B_V and U_V are the non-ground counterparts of the standard definition of the Herbrand base and universe. The following definition is due to Falaschi et al. [20].

DEFINITION 8.1 (The Mapping T_S):

Let P be a definite program. The mapping T_S on the set of S-Herbrand interpretations, associated to P, is defined as $T_S(I) = \{A\theta \mid \exists A \leftarrow B_1,\ldots,B_n \in P, \exists B'_1,\ldots,B'_n \in I, \text{ and } \exists\theta = mgu((B_1,\ldots,B_n),(B'_1,\ldots,B'_n))\}$. ■

Falaschi et al. show that T_S is monotonic and continuous which implies that T_S has a least fixed point. They furthermore show that this denotational characterization is equivalent to its model-theoretic pendant, the least S-model.

ALGORITHM 8.2 (Naive Evaluation):

Input: A definite program P.
Output: The least fixed point which is the minimal S-Model M_S of P, if it is finite. If the least fixed point is infinite, then the output is an infinite sequence of approximations which approach the least fixed point as a limit.

1) $F=\emptyset$.

2) Repeat $F:=F\cup\Delta F$, where $\Delta F=\{A\theta \mid$ there is a clause $A\leftarrow B_1,\ldots,B_n$ in P, a sequence of atoms $B'_1,\ldots,B'_n$ in F and a most general unifier $\theta=mgu((B_1,\ldots,B_n),(B'_1,\ldots,B'_n))$ such that $A\theta$ is not subsumed by an atom if F$\}$, until $\Delta F=\emptyset$. ■

This definition of the naive strategy is more general than that given in [81]. Ullman assumes that P is safe so that all atoms in the fixed point are ground. In this case the naive approach implements the function T_P defined in [46] which is a mapping between standard Herbrand interpretations. For safe programs the naive and semi-naive approaches can be implemented more efficiently, since unification can be replaced by matching and the subsumption check can be replaced by a duplicate check.

Maher and Ramakrishnan describe an improvement of the subsumption check for the semi-naive procedure. In [51] they show that, for many programs, it suffices to check whether any atom was produced in the previous iteration, and they develop techniques to prove this property.

The main drawback of the naive approach lies in the duplication of work; in every round, we have to evaluate each rule body against all facts. The basic principle of the semi-naive approach is motivated by the observation that, at each iteration, new atoms can only be produced using at least one atom obtained in the previous iteration. Therefore, the semi-naive approach exploits the differential ΔF, using in every round at least one B'_i which was derived in the previous round.

Compared with top-down approaches, the pure semi-naive approach lacks a feature to select relevant clauses. In the last years many researchers addressed this problem to the effect that very promising approaches to eliminate this shortcoming were developed. Some of these optimizations are the system graph approach of Kifer and Lozinskii, described in [33, 34, 36], magic sets, introduced by Bancilhon et al. in [5], generalized supplementary magic sets, presented by Beeri et al. in [6] and magic templates described by Ramakrishnan in [61]. The last three approaches are rewriting strategies which transform a logic program P into a program P', which is equivalent to P w.r.t. the top-level goal and can be evaluated efficiently with the semi-naive approach. Recently Ullman [80] compared the result of the magic set optimization with the QRGT algorithm. In his paper with the title

'BOTTOM-UP BEATS TOP-DOWN FOR DATALOG'

he shows that the magic set approach is never worse than any top-down approach developed so far by more than a constant factor. Thus, all the advantages so far associated with top-down evaluations only can also be offered by bottom-up evaluation.

In this context it is an interesting question how GDFC-resolution compares to these bottom-up approaches. First we have to mention, that GDFC-resolution is a tuple at a time approach which possibly does a lot of redundant derivations. This excludes GDFC-resolution (as well as SLD-resolution) from being an efficient strategy for very large databases. In his diploma thesis Heidelbach [30] demonstrates how extension tables implemented in ET* can be used to reduce the number of redundant derivations and to make GDFC-resolution sound and complete for datalog programs if the left-first computation

rule with depth-first search is applied. But, as the following example shows, this approach does not completely eliminate redundant derivations.

EXAMPLE 8.3 (Elimination of Redundant Derivations):

Suppose P is

$$p \leftarrow r,q$$
$$r \leftarrow s$$
$$q \leftarrow r$$
$$s \leftarrow$$

and G is $\leftarrow p$. Let us consider the GDFC-derivation for $P \cup \{G\}$ in which all solved atoms are stored in an extension table and reused when they are needed. Clearly the next goal in the derivation is $\leftarrow <s,p>$. Using the second clause we derive $\leftarrow <r,p>$. The next derived goal is $\leftarrow q$. Again we use the fact $s \leftarrow$ and derive $\leftarrow <s,q>$. In the next step we use the second clause which yields $\leftarrow <r,q>$. Finally, we derive the empty empty clause using the third clause. The fact that r occurs twice on the left side of a subgoal-goal pair indicates that r is proved twice in this derivation. Thus, even the use of extension tables does not prevent GDFC-resolution from making redundant derivations. In contrast to that, using the ET^* algorithm r is proved only once. ■

An interesting observation is that the query processing strategy APEX for deductive databases, which is due to Lozinskii [49], is strongly related to GDFC-resolution, because a slightly modified version of it realizes a set oriented extension of GDFC-resolution. APEX uses migration sets to focus on relevant clauses where all terms occurring as arguments are assumed to be distinct variables. Thus, the rules in P have the form

$$A \leftarrow A_1,\ldots,A_n,E_1,\ldots,E_m$$

where $A,A_1,\ldots,A_n$ are atoms containing distinct variables as arguments and $E_1,\ldots,E_m$ are equalities between the variables in $A,A_1,\ldots,A_n$ or between variables and constants. A migration set of X, mig(X), is defined by all terms to which X can migrate:

DEFINITION 8.4 (Migration Sets):

If P contains a clause

$$p(\ldots X\ldots) \leftarrow \ldots,q(\ldots Y\ldots),\ldots,X{=}Y,\ldots$$

then $Y \in mig(X)$. If there are two atoms such that their unifier contains a substitution $X{=}Y$ then $Y \in mig(X)$. If $Y \in mig(X)$ and $Z \in mig(Y)$ then $Z \in mig(X)$. ■

A fact $p(Y_1{=}c_1,\ldots,Y_n{=}c_n)$ with variables $Y_1,\ldots,Y_n$ and constants $c_1,\ldots,c_n$ is relevant to $q(\ldots,X_i{=}d_i,\ldots)$ if there is a $j \in \{1,\ldots,n\}$ such that $Y_j \in mig(X_i)$ and $c_j{=}d_i$. In contrast to the link clauses this notion of relevance directly depends on the data flow and not on the order of literals.

Bancilhon and Ramakrishnan argue in [4] that APEX which is illustrated with Algorithm 8.5 is a mixed approach. This is motivated by the fact that step 2) is a classical bottom-up step, while the recursive calls to APEX in step 3) also occur in top-down approaches such as SLD-resolution.

ALGORITHM 8.5 (APEX):

Input: A program P and a single literal goal $G{=}\leftarrow A$.
Output: All answers to G, if APEX terminates.

1) Compute R, the set of all facts relevant for A.

2) For each rule C in P and each combination of facts in R satisfying the body of C, compute the resulting rule head and insert it into R. Terminate if no new facts can be produced.

3) If $C{=}B \leftarrow B_1,\ldots,B_n$ is a non-unit clause in P and $B' \leftarrow$ is a fact in R which is unifiable with B_i ($1 \le i \le n$) by mgu θ, then solve all auxiliary queries $\leftarrow B_j\theta$, for $j{=}1\ldots n$ and $j \ne i$, using the facts in P. After generating all answers to each goal $\leftarrow B_j\theta$, execute C augmenting R by the newly produced facts and go to 2). ■

The parallelism between APEX and GDFC-resolution is obvious. To extend GDFC-resolution to a relation at a time approach, we make the following

changes: in step 1) we apply the link clauses to compute relevant facts, and in step 3) we always have j=1, since each relevant fact B'← can only be unifiable with B_1. In both steps we additionally have to check whether R is augmented by any facts, since we can stop if this is not the case. Note that the data flow between the head and the leftmost body literal represented by the equalities has to be made explicit in order to obtain link clauses with optimal selectivity.

Bancilhon and Ramakrishnan demonstrate in different benchmarks that APEX is dominated by the magic set approach for the transitive closure and has a comparable performance for the same generation procedure. This demonstrates that GDFC-resolution, even if it is not designed especially for this purpose, can be extended to an efficient evaluation strategy for deductive databases. In this context it is important to mention that the supplementary magic set approach is more efficient than this relation oriented extension of GDFC-resolution. The disadvantage of the set-oriented variant of GDFC-resolution is that it weakly supports sideways information passing; the answers obtained for $\leftarrow B_j\theta$ are not exploited to restrict the search space for $B_{j+1}\theta$. In the supplementary magic set approach sideways information passing is realized by supplementary relations and in the system graph approach by dynamic filtering [35]. However, it is worth mentioning that the link clauses are query independent. A further interesting property is that we do not place the frequently occurring restriction, that the program is safe, i.e., that all variables occurring in the head also occur in the body. Moreover, if we implement sideways information passing then we even can deal with built-in predicates for arithmetic.

8.2 Choosing the Rule Direction

Treitel and Genesereth [77] address the question which rule direction should be chosen. They consider LLNR-resolution which is a restricted form of LL-resolution. LL-resolution is a specialized version of the standard binary resolution; complementary literals, which are resolved away, must each be the first or leftmost in their respective clauses. In LLNR-resolution additionally the order of the literals appearing in the resolvent is specified. At each derivation, the relative positions of two literals from the same parent clause are inherited. The literals of the parent clause whose leftmost literal is negative are put in the place of the negative literal which was resolved away.

If all clauses are Horn and all non-unit clauses have their positive literal at the leftmost or rightmost position, then LLNR-resolution behaves very similar to backward or forward chaining: if all positive literals are in the leftmost position, then LLNR-resolution realizes backward chaining and otherwise forward chaining.

They address the question which set of non-unit clauses R_f, where R is the set of rules, is the optimal set to be used forward, i.e., re-written with the positive literal at the end of the clause. R_f must realize a coherent strategy which means that if a rule is evaluated forward then all its predecessors are. The coherence property is motivated by the fact that under a coherent strategy all of the forward inferences can be done before any of the backward ones. The resulting facts of the forward chaining process can be stored and used as the basis for the backward inferences. Optimality is defined by the sum of the times taken by all the forward and backward inferences.

The authors show that it is NP-hard to find the optimal R_f if P is a datalog program. If there are no rules generating duplicate answers, then the optimal set can be found in polynomial time [77]. Simultaneously they present an polynomially bounded algorithm to choose an optimal coherent strategy for datalog programs without duplicate answers.

One of the benefits available from this approach is that it provides an automatic choice of the optimal strategy. In most of the cases the efficiency of different strategies is compared considering particular procedures such as the transitive closure (ancestor) or the same generation example only. However, since GDFC- and SLD-resolution differ in several points from their corresponding LLNR-resolution variants, this approach cannot easily be adopted to choose between GDFC- and SLD-resolution. For example, the forward chaining component of LLNR-resolution contains no mechanism to focus on relevant clauses. In contrast to GDFC-resolution, LLNR-resolution stores all deduced facts and can re-use them when they are needed. From our point of view it is an important direction for future research to investigate automatic approaches to choose the optimal strategy between GDFC- and SLD-resolution statically as well as dynamically on the basis of the results for LLNR-resolution.

Chapter 9

CONCLUSIONS

This dissertation discusses an approach to goal-directed forward chaining for logic programs which Yamamoto and Tanaka introduced in 1986. In their paper they proposed a translation approach and mentioned two open problems with it. The first one concerns the termination problem: 'In the case of first order predicate logic, recursive rules are sometimes harmless. However, we have not yet solved the problem of how to determine what kinds of recursion this system can and cannot handle.'

Goal-directed forward chaining focuses on relevant clauses by means of link clauses which are generated out of the program being interpreted. These link clauses cause the second problem: 'Another problem is the huge number of link clauses that are obtained when the transitive relations are enumerated at the time of translation. Although it is possible to determine the transitive relation at the time of execution, this makes the system slower.'

We first addressed the problem with the link clauses. We showed that it is generally undecidable for a definite program whether or not the transitive closure of the link clauses is finite. We presented an approach to obtain a finite number of more general link clauses for recursive programs with function symbols. By renaming variables in the link clauses, which excludes cyclic data flow through structured terms, we guarantee that the process to generate the transitive data structure at compile time terminates. We furthermore showed that, especially for propositional programs, the number of link clauses in the transitive closure can be reduced by moving appropriate literals to the leftmost position in the body of non-unit clauses. Concerning the complexity of this problem we proved that it is NP-hard to find an optimal selection of literals. We presented a simple algorithm which approximates the optimum. In experimental studies we found out that this approach on average leads to considerable space savings.

Based on a meta-interpreter we introduced GDFC-resolution as linear resolution strategy for goal-directed forward chaining. We showed that GDFC-resolution is sound and complete for definite logic programs. This result is important from the viewpoint of the semantic integration of goal-directed forward chaining for logic programs. It suggests that the operational semantics of SLD-resolution is not affected by the transformation which realizes this approach on top of Prolog.

A large part of this dissertation is concerned with the efficiency of GDFC-resolution. We compared the efficiency of GDFC-resolution with SLD-resolution and showed that neither of both approaches is always better than the other. We compared different properties of SLD- and GDFC-trees for propositional programs. We obtained the result that the number and size of success branches are the same and that an SLD-tree contains at least as many failure branches as the corresponding GDFC-tree, where each failure branch in an SLD-tree is at least as long as the corresponding failure branch in the GDFC-tree. These results give rise to the conjecture that GDFC-resolution on average is more efficient than SLD-resolution for propositional logic programs. We described experimental studies confirming this conjecture.

We furthermore analyzed and compared the average case complexity of GDFC-resolution with that of SLD-resolution for propositional binary Prolog programs. We showed that the number of resolutions needed by GDFC-resolution in the worst case depends linearly on the number of non-unit clauses, whereas SLD-resolution on average may need exponentially many steps. Even if GDFC-resolution is not always better than SLD-resolution for this program class, it provides the great benefit that it guarantees sufficient efficiency for the whole class.

We also considered different datalog programs and showed that GDFC-resolution is very efficient for taxonomic hierarchies. This result is important since it demonstrates that GDFC-resolution in fact offers the efficiency which generally is expected for the bottom-up evaluation of taxonomic hierarchies.

We considered the procedure defining the ancestor relationship and showed that GDFC-resolution always needs fewer inferences than SLD-resolution. We also presented experiments, in which we compared the runtime efficiency of the meta-interpreters for GDFC- and SLD-resolution. The scale of the improvement goes beyond that what could be expected based on the theoretical results. The reason for this efficiency of goal-directed forward chaining is that the facts representing the parent relationship are scanned half as often as

it is done using Prolog. We also considered the procedure defining the reflexive transitive closure of directed acyclic graphs. We proved that GDFC-resolution likewise is better for this procedure, since it always needs one inference less than SLD-resolution.

These results suggest that GDFC-resolution even for datalog programs may be more efficient than SLD-resolution. Since until now goal-directed forward chaining has only been implemented on top of Prolog, an efficient implementation of this calculus providing a similar or even better performance than Prolog may be one field of future research.

We furthermore compared GDFC-resolution with bottom-up approaches coming from the field of deductive databases. We demonstrated that there is a strong relationship between the APEX method and GDFC-resolution which is expressed by the fact that a slightly modified variant of APEX realizes a set-oriented extension of goal-directed forward chaining. Since not all possibilities for an improvement such as sideways information passing are exploited in this approach, it is less efficient than the supplementary magic set approach. However, the fact that it is query independent sets up a strong motivation for future research in this area.

Assuming that GDFC- and SLD-resolution both apply the left-first computation rule we showed that GDFC-resolution using top-down computed links always terminates if SLD-resolution does. Moreover, the class of definite programs for which SLD-resolution terminates is a proper subset of the class of programs for which GDFC-resolution terminates. However, the price for this termination behaviour is that, at least for recursive programs with function symbols, we have to compute the necessary transitive links at runtime. Unfortunately this computational overhead generally makes SLD-resolution more efficient for such programs. However, if we consider datalog programs only, then the transitive closure of the link clauses is finite so that we can compute it statically. Consequently, at least for datalog programs the better termination behaviour can be achieved without loosing efficiency.

To sum up, this dissertation analyzes goal-directed forward chaining from different viewpoints which concern aspects of semantics, efficiency and verification. It shows that goal-directed forward chaining combines many advantages generally associated to top-down approaches with bottom-up evaluation. The semantically clean integration of GDFC-resolution into SLD-resolution is accompanied with the performance of a bottom-up approach. Simultaneously,

GDFC-resolution provides a better termination behaviour than SLD-resolution.

In the field of deductive databases the question which is the optimal inference direction has intensively been discussed in the last decade. One result of this research is that bottom-up approaches are more efficient than top-down approaches for large datalog programs. Now this dissertation demonstrates that GDFC-resolution at least in the context of datalog programs can be a serious alternative to SLD-resolution. This suggests that SLD-resolution is not always the resolution strategy of choice and that it is worth to intensify the research on alternative control strategies for logic programs.

REFERENCES

1. Abramson, H. and Dahl, V. *Logic Grammars*, Springer Verlag (1989).
2. Apt, K.R. Introduction to Logic Programming. Tech. Rept. CS-R8826, Centre for Mathematics and Computer Science, Amsterdam, Netherlands, 1988.
3. Börger, E. *Berechenbarkeit, Komplexität, Logik*, Vieweg (1985).
4. Bancilhon, F. and Ramakrishnan, R. An Amateur's Introduction to Recursive Query Processing Strategies. In *Proceedings of the Fifth ACM SIGACT SIGMOD SIGART Symposium on Principles of Database Systems*, ACM-Press, 1986, pp. 16-52.
5. Bancilhon, F., Maier, D., Sagiv, Y., and Ullman, J.D. Magic Sets and Other Strange Ways to Implement Logic Programs. In *Proceedings of the Fifth ACM SIGACT SIGMOD SIGART Symposium on Principles of Database Systems*, ACM-Press, 1986, pp. 1-15.
6. Beeri, C. and Ramakrishnan, R. On the Power of Magic. In *Proceedings of the Sixth ACM SIGACT SIGMOD SIGART Symposium on Principles of Database Systems*, ACM-Press, 1987, pp. 269-283.
7. Benkerimi, K. and Lloyd, J.W. A Procedure for the Partial Evaluation of Logic Programs. Tech. Rept. TR-89-04, Department of Computer Science, University of Bristol, 1989.
8. Binot, J.L., Burgard, W., De Zegher, I., Donner, D., and Michaux, G. Integration of a Frame Based Extension in a Prolog Environment. In *Proceedings of the First PROTOS Workshop, Morcote, Switzerland*, Appelrath, H.J., Cremers, A.B., and Schiltknecht, H., 1989, pp. 10-28.
9. Bollobás, B. *Random Graphs*, Academic Press (1985).
10. Brownston, L., Farrell, R., Kant, E., and Martin, N. *Programming Expert Systems in OPS5: An Introduction to Rule-Based Programming*, Addison-Wesley Publ. Comp., Inc. (1985).
11. Bry, F. Query Evaluation in Recursive Databases: Bottom-up and Top-down Reconciled. In *Proceedings First International Conference on Deductive and Object-Oriented Databases, Kyoto, Japan*, 1989.
12. Chang, C.L. and Lee, R.C.T. *Symbolic Logic and Mechanical Theorem Proving*, Academic Press (1973).

13. Charniak, E. and McDermott, D. *Introduction to Artificial Intelligence*, Addison-Wesley Publ. Comp., Inc. (1985).

14. Clark, K.L. and Mc Cabe, F.G. Prolog: A Language for Implementing Expert Systems. *Machine Intelligence 10* (1982), pp. 455-475.

15. Clocksin, W.F. and Mellish, C.S. *Programming in Prolog*, Springer Verlag (1981).

16. Cohen, P.R. and Feigenbaum, E.A. *The Handbook of Artificial Intelligence*, William Kaufmann Inc., Vol. 3 (1982).

17. Cooper, T. and Wogrin, N. *Rule-Based Programming with OPS5*, Morgan Kaufmann Publishers, Inc., Los Altos, California (1988).

18. Dembinski, P. and Maluszynski, J. And-Parallelism with Intelligent Backtracking for Annotated Logic Programs. In *Proceedings of the International Symposium on Logic Programming, Boston*, IEEE Computer Society Press, 1985, pp. 29-38.

19. Dietrich, S.W. Extension Tables: Memo Relations in Logic Programming. In *Proceedings of the International Symposium on Logic Programming, San Francisco, California*, Warren, D.S. and Haridi, S., IEEE Computer Society Press, 1987, pp. 264-272.

20. Falashi, M., Levi, G., Martelli, M., and Palamidessi, C. Declarative Modeling of the Operational Behaviour of Logic Languages. *Theoretical Computer Science 69* (1989), pp. 289-318.

21. Forgy, C.L. Rete: A Fast Algorithm for the Many Pattern / Many Object Pattern Match Problem. *Artificial Intelligence 19* (1982), pp. 17-37.

22. Furukawa, K. and Fujita, H. Deriving an Efficient Production System by Partial Evaluation. In *Proceedings of the North American Conference on Logic Programming, Cleveland, Ohio*, Lusk, E.L. and Overbeek, R.A., The MIT Press, 1989, pp. 661-674.

23. Gallaire, H., Minker, J., and Nicolas, J.M. Logic and Databases: A Deductive Approach. *ACM Computing Surveys 16*, 2 (June 1984), pp. 153-185.

24. Gardarin, G. and Valduriez, P. *Relational Databases and Knowledge Bases*, Addison-Wesley Publ. Comp., Inc. (1989).

25. Garey, M.R. and Johnson, D.S. *Computers and Intractability: A Guide to NP-Completeness*, W. H. Freeman and Company, San Francisco (1979).

26. Genesereth, M.R. and Nilsson, N.J. *Logical Foundations of Artificial Intelligence*, Morgan Kaufmann Publishers, Inc., Los Altos, California (1987).

27. Graham, I. Inside the Inference Engine. In *Expert Systems - Principles and Case Studies*. Chapman and Hall Computing, Forsyth, R., pp. 57-83, 1989.

28. Harary, F. *Graph Theory*, Addison-Wesley Publ. Comp., Inc. (1969).

29. Hayes-Roth, F. Rule-Based Systems. *Communications of the ACM 28*, 9 (1985), pp. 921-932.

30. Heidelbach, M. *Behandlung spezieller Probleme bei der forwärtsgerichteten Auswertung logischer Programme*, Diploma thesis, in German, University of Dortmund, 1990.

31. Johns, N. and Spenser, C., *LPA MacProlog Reference Manual*, Logic Programming Associates Ltd., London.

32. Johnson, D.S. Approximation Algorithms for Combinatorial Problems. *Journal of Computer and System Sciences 9* (1974).

33. Kifer, M. and Lozinskii, E. Query Optimization in Logical Databases. Tech. Rept. 85/16, Department of Computer Science, SUNY at Stony Brook, 1985.

34. Kifer, M. and Lozinskii, E.L. Filtering Data Flow in Deductive Databases. In *International Conference on Database Theory, Rome, Italy*, Ausiello, G. and Atzeni, P., Springer Verlag, 1986, pp. 186-202.

35. Kifer, M. and Lozinskii, E.L. A Framework for an Efficient Implementation of Deductive Database Systems. In *Proceedings of the Sixth Advanced Database Symposium, Tokyo, Japan*, 1986.

36. Kifer, M. and Lozinskii, E.L. Implementing Logic Programs as a Database System. In *Third International Conference on Data Engineering*, Computer Society Press, 1987, pp. 375-385.

37. Kleine Büning, H. and Löwen, U. Towards Average Complexity of Propositional Binary Prolog Programs. *Fundamenta Informaticae 13* (1990), pp. 387-399.

38. Kleine Büning, H., Löwen, U., and Schmitgen, S. Equivalence of Propositional Prolog Programs. *Journal of Automated Reasoning 6* (1990), pp. 319-335.

39. Knuth, D.E. *The Art of Computer Programming - Fundamental Algorithms*, Addison-Wesley Publ. Comp., Inc., Vol. 1, 2 (1973).

40. Koseki, Y. Amalgamating Multiple Programming Paradigms in Prolog. In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence, Milan, Italy*, McDermott, J., Morgan Kaufmann Publishers, Inc., Los Altos, California, 1987.

41. Kowalski, R.A. Predicate Logic as a Programming Language. In *Proc. IFIP Congress*, Stockholm, North Holland, 74, pp. 556-574.

42. Kowalski, R.A. Algorithm = Logic + Control. *Communications of the ACM 22* (1979), pp. 424-436.

43. Kowalski, R.A. *Logic for Problem Solving*, North-Holland, Amsterdam New York Oxford (1979).

44. Lassez, J.L., Maher, M.J., and Mariott, K. Unification Revisited. In *Deductive Databases and Logic Programming*. Morgan Kaufmann Publishers, Inc., Los Altos, California, Minker, J., pp. 587-626, 1987.

45. Levi, G. Models, Unfolding Rules and Fixpoint Semantics. In *Proceedings of the Fifth International Conference on Logic Programming*, Kowalski, R.A. and Bowen, K., 1988, pp. 1649-1665.

46. Lloyd, J.W. *Foundations of Logic Programming, Second, Extended Edition*, Springer Verlag (1987).

47. Lloyd, J.W. and Shepherdson, J.C. Partial Evaluation in Logic Programming. Tech. Rept. TR-87-09, Department of Computer Science, University of Bristol, 1987.

48. *LPA Prolog Professional Reference Manual*, Logic Programming Associates Ltd., London.

49. Lozinskii, E.L. Evaluating Queries in Deductive Databases by Generating. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, Aravind, J., Morgan Kaufmann Publishers, Inc., Los Altos, California, 1985, pp. 173-177.

50. Magura, N. *Experimentelle Untersuchungen zur Beurteilung der Effizienz von zielgerichtetem forward-chaining*, Diploma thesis, in German, University of Dortmund, 1991.

51. Maher, M.J. and Ramakrishnan, R. Déjà Vu in Fixpoints of Logic Programs. In *Proceedings of the North American Conference on Logic Programming, Cleveland, Ohio*, Lusk, E.L. and Overbeek, R.A., The MIT Press, 1989, pp. 963-980.

52. Maier, D. and Warren, D.S. *Computing with Logic - Logic Programming with Prolog*, Benjamin/Cummings (1989).

53. Matsumo, Y., Tanaka, H., Hirakawa, H., Miyoshi, H., and Yasukawa, H. BUP: A Bottom-Up Parser Embedded in Prolog. *New Generation Computing 1*, 2 (1983), pp. 145-158.

54. Montini, G. Efficiency Considerations on Build-In Taxonomic Reasoning in Prolog. In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence, Milan, Italy*, McDermott, J., Morgan Kaufmann Publishers, Inc., Los Altos, California, 1987.

55. Nilsson, N.J. *Principles of Artificial Intelligence*, Springer Verlag (1982).

56. O'Keefe, R.A. *The Craft of Prolog*, The MIT Press (1990).

57. Plümer, L. *Termination Proofs for Logic Programs*, Ph.D. dissertation, University of Dortmund, 1989.

58. Plümer, L. *Termination Proofs for Logic Programs*, Springer Verlag, Lecture Notes in Artificial Intelligence, 446 (1990).

59. Plümer, L. Termination Proofs for Logic Programs Based on Predicate Inequalities. In *Proceedings of the Seventh International Conference on Logic Programming*, Szeredi, P. and Warren, D.H.D., The MIT Press, 1990.

60. *Quintus Prolog Reference Manual*, Quintus Computer Systems, Inc., Mountain View, California.

61. Ramakrishnan, R. Magic Templates: A Spellbinding Approach to Logic Programs. In *Proceedings of the Fifth International Conference on Logic Programming*, Kowalski, R.A. and Bowen, K., 1988, pp. 140-159.

62. Robinson, J.A. A Machine-Oriented Logic Based on the Resolution Principle. *Journal of the ACM 12*, 1 (1965), pp. 23-41.

63. Rosen, B.K. Robust Linear Algorithms for Cutsets. *Journal of Algorithms 3* (1982), pp. 205-217.

64. Rossi, G. Uses of Prolog in Implementation of Expert Systems. *New Generation Computing 4* (1986), pp. 321-329.

65. Schaefer, T.J. The Complexity of Satisfiability Problems. In *Proceedings of the Tenth ACM Symposium on Theory of Computing*, 1978, pp. 216-226.

66. Shintani, T. An Approach to Speeding Up the Prolog-based Inference Engine KORE/IE. In *Logic Programming: Proceedings of the Fourth International Conference*, Lassez, J.L., The MIT Press, 1987, pp. 284-297.

67. Shintani, T. A Fast Prolog-Based Production System KORE/IE. In *Proceedings of the Fifth International Conference on Logic Programming*, Kowalski, R.A. and Bowen, K., 1988, pp. 26-41.

68. Speckenmeyer, E. On Feedback Problems in Digraphs. Tech. Rept. 264, Department of Computer Science, University of Dortmund, 1988.

69. Speckenmeyer, E. On Feedback Problems in Digraphs. In *Proceedings of the Fifteenth Workshop on Theoretic Concepts in Computer Science, Rolduc, Netherlands*, Springer Verlag, 1989.

70. Stamm, H., private communication, 1991.

71. Sterling, L. and Shapiro, E.Y. *The Art of Prolog: Advanced Programming Techniques*, The MIT Press (1986).

72. Sterling, L. and Beer, R.D. Metainterpreters for Expert System Construction. *Journal of Logic Programming 6* (1989), pp. 163-178.

73. Tärnlund, S. Horn Clause Computability. *BIT 17* (1977), pp. 215-226.

74. Takeuchi, A. and Fujita, H. Competitive Partial Evaluation - Some Remaining Problems of Partial Evaluation. *New Generation Computing 6* (1988), pp. 259-277.

75. Tamaki, H. and Sato, T. Unfold/Fold Transformations of Logic Programs. In *Proceedings of the Second International Conference on Logic Programming*, Tärnlund, S., 1984, pp. 127-138.

76. Tamaki, H. and Sato, T. OLD Resolution with Tabulation. In *Proceedings of the Third International Conference on Logic Programming*, 1986, pp. 84-98.

77. Treitel, R. and Genesereth, M.R. Choosing Directions for Rules. *Journal of Automated Reasoning 3* (1987), pp. 395-431.

78. Ullman, J.D. and Van Gelder, A. Efficient Tests for Top-Down Termination of Logical Rules. *Journal of the ACM 35*, 2 (1988), pp. 345-373.

79. Ullman, J.D. *Principles of Database and Knowledge-Base Systems*, Computer Science Press, Vol. 1 (1988).

80. Ullman, J.D. Bottom-Up Beats Top-Down for Datalog. In *Proceedings of the Eighth ACM SIGACT SIGMOD SIGART Symposium on Principles of Database Systems*, ACM-Press, 1989, pp. 140-149.

81. Ullman, J.D. *Principles of Database and Knowledge-Base Systems*, Computer Science Press, Vol. 2 (1989).

82. Ullman, J.D. The Theory of Deductive Database Systems. In *COMPCON*, IEEE Computer Society Press, 1990, pp. 496-502.

83. *Logische Programmierung und Wissensrepräsentation: Implementierung einer Entwicklungsumgebung*, University of Dortmund, Project Group 128 (PEPP), Final Report, 1988, in German.

84. Van Gelder, A. Negation as Failure Using Tight Derivations for General Logic Programs. *Journal of Logic Programming 6* (1989), pp. 109-133.

85. Vasey, P., *flex Expert System Toolkit*, Logic Programming Associates Ltd., London.

86. Vieille, L. Recursive Query Processing: The Power of Logic. *Theoretical Computer Science 69* (1989), pp. 1-53.

87. Warren, D.H.D. Efficient Processing of Interactive Relational Database Queries Expressed in Logic. In *Proceedings of the Seventh International Conference on Very Large Data Bases*, Morgan Kaufmann Publishers, Inc., Los Altos, California, 1981.

88. Waterman, D.A. *A Guide to Expert Systems*, Addison-Wesley Publ. Comp., Inc. (1986).

89. Winston, P.H. *Artificial Intelligence*, Addison-Wesley Publ. Comp., Inc. (1984).

90. Yamamoto, A. and Tanaka, H. Translating Production Rules into a Forward Reasoning Prolog Program. *New Generation Computing 4* (1986), pp. 97-105.

Index

1-in-3SAT 141
APEX 149
arc
  binary 89
  unary 89
backward chaining 1
chain 90
computed answer 53
feedback arc set 33
forward chaining 2
GDFC-
  derivation 53
    failed 53
  goal 52
  refutation 53
    corresponding 70
  resolution 52; 54
  resolvent 53
  tree 56
goal-directed forward chaining
  a meta-interpreter for 10; 14; 131
  a transformation approach for 13
hitting set 39
link clause program 22
  finite 36
  reflexive 22
  reflexive finite 36
link clauses 21
  bottom-up computed 132
  top-down computed 132
link graph 37
literals
  selection of 38
LL-resolution 150
LLNR-resolution 2; 150
loop-test problem
  general 140; 141
  special 139
migration graphs 30
migration sets 149
naive evaluation 2; 146
predicate dependency graph 42
production system 2
program
  binary 88
  normalized 136
  tree-like 89
  well-moded 136
program graph 89
Prolog 1
S-Herbrand interpretations 146
semi-naive evaluation 2; 147
SLD-
  refutation
    corresponding 68
  resolution 1
subgoal-goal pair 51
  recursive 51
tree
  binary 90
  full binary 90
unique path property 46
unit-resolution 2
vertex
  binary 90
  unary 90

Curriculum Vitae

Name:	Wolfram Burgard
Date of birth:	8 February 1961
Birthplace:	Gelsenkirchen

7.9.67 - 2.7.71	Primary school
16.8.71 - 12.5.80	Secondary school
12.5.80	Examination at the Gymnasium Hammonense, Hamm
13.10.81 - 23.4.87	Study of computer science at the University of Dortmund
23.4.87	Diploma in computer science
27.4.87- 30.9.90	Scientific employee at the department of computer science of the University of Dortmund
since 1.10.90	Scientific employee at the department of computer science of the University of Bonn