Dissertation

Flexible Computation of the Well-Founded Semantics of Normal Logic Programs

by
Ulrich Zukowski

submitted to

Fakultät für Mathematik und Informatik
Universität Passau

February 2001

Abstract

The development and maintenance of modern information systems, that are getting more and more complex and, at the same time, require a high flexibility, is a great challenge for current information and communication technology. One contribution is the usage of a rule-based specification mechanism that allows the compact and declarative description of complex situations. One such rule language is given by *normal logic programs*.

The *well-founded semantics* has been accepted as the most relevant semantics for logic-based information systems. It assigns a unique (partial) model to every normal logic program, i.e., it allows specifications using any kind of recursion including cyclic positive and negative dependencies. Further, the time complexity of computing the well-founded model of a given intensional database is polynomial in the size of the extensional database.

Several evaluation methods for the well-founded semantics are known. Each of these approaches has its advantages and disadvantages. Thus, for practical applications it will not be sufficient to select one static evaluation algorithm and apply it uniformly to any kind of application. However, it is difficult to combine evaluation methods because they all are based on different data structures, applicable to different classes of programs, or whose control strategies are not compatible.

In this work we present a framework based on a set of *program transformations* that generalizes all major computation approaches using a common data structure and provides a common language to describe their evaluation strategy. We allow strategies from set-oriented bottom-up to single-answer top-down or any combination in this range just by rearranging the order of program transformations using an intuitive regular expression like specification syntax. Leveraged by our framework, we are able to analyze and compare known methods on a high level of abstraction. The advantages of the methods can be identified and new algorithms can be composed that combine the strengths of several separate approaches. For instance, we show how to optimize the alternating fixpoint procedure as the most prominent bottom-up method together with its extensions for magic set transformed programs. We can also realize the search strategies available within the *SLG* approach, the most prominent top-down method, and even extend it by allowing the activation of more than one body literal at a time.

This leads to more efficient computations for some programs and enables even the parallel evaluation of several body literals in applications where it is appropriate.

Our rewriting system gives the formal background to analyze and combine different evaluation strategies in a common framework, or to design new algorithms and prove the correctness of its implementations at a high level just by changing the order of program transformations. Further, we provide an algorithm to implement the transformation, and illustrate by experimental results, that the theoretical results carry over to the prototype implementation.

Acknowledgements

This thesis has been written during my time as a research assistant at the research group of Prof. Dr. Burkhard Freitag at the University of Passau. I would like to thank him for giving me this opportunity and for his guidance and assistence during the development of the ideas presented in this dissertatiom. His advice and encouragement were invaluable.

I would like to thank my co-supervisor Prof. David S. Warren, Ph.D., of the State University of New York at Stony Brook. He is one of the leading experts in the research area of logic programming. His work was of great importance for this dissertation. We have met at several conferences and had many fruitful discussions, not only on technical details but rather on the central questions giving the motivation for this work. He reviewed the final version and his suggestions and comments helped to improve considerably this dissertation.

A great influence on this work has come from Stefan Brass and Jürgen Dix. Starting from early discussions on program transformations a long-term cooperation leading to several joint publications could be established.

Thanks to Illka Niemelä from whom I got several inspirations concerning the optimization of my prototype implementation.

I would like to thank all my colleagues at the University of Passau, especially Alfred Fent and Carl-Alexander Wichert for proofreading the final version, and Christian Süß with whom I shared the office.

Last, but certainly not least, I have to thank my family, my parents and my wife Beatrice, for their support and care over the years.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

Modern information systems are getting more and more complex. At the same time, flexibility is very important to keep track of the requirements of modern internet and communication centered environments. In this scenario it is difficult to maintain a modern IT system based just on the classical development tools and the imperative programming languages.

In these complex applications it is required that configurations should be maintainable in some declarative way. Or there are contracts or policies that describe what information should be accessible to what group of agents involved. Or in another application, the behaviour or the way in which information is presented should be changed from time to time just to meet the current trend of the target group or just to keep an application alive.

In none of the scenarios described above, the administrator of an application would like to achieve the dynamic behaviour by changing the implementation in a programming language style. One solution would be to offer possibilities of customization that can be adapted by non-programmers. However, situations can easily be found where this kind of easy customization is not powerful enough to meet the requirements of a complex application.

Having this in mind, it becomes clear why many applications are applying some kind of *rule-based* specification mechanism. Using the appropriate kind of rule language, perhaps tailored to the current application, many complex situations can be described in a compact and declarative way without the need of a full programming language, but in a much more expressive way than by clicking some check-boxes in a dialog box.

This requirement of a rule-based specification is different from the kind of rule usage which has been popular in the Prolog programming paradigm [Llo87, MW88]. We do

not consider the rules as a programming language with many interactive components to implement a complete application. Instead, we see the usage of rule languages, possible tailored to the application, to specify in a declarative way a set of customized requirements to a host application, that interprets the rules in an appropriate way and changes its behaviour as demanded.

1.2 The Well-Founded Semantics

In the area of *logic programming* much research has been done to satisfy the requirements discussed in Section 1.1. Based on the mathematical concept of the first order theory, a rule language consisting of if-then-rules has been proposed. For sets of positive rules, in that consequences may be drawn only based on positive information, a semantics was defined that coincides with the mathematical concept of models and logical consequences for a first order theory (cf. [Llo87]). We will introduce the concept of the *least Herbrand model* for *definite logic programs* in Chapter 2.

However, the expressional power of definite programs is limited. In many practical applications rules that may draw consequences also from negative information are required. To satisfy this requirement, *normal logic programs* [AB94] have been defined that consist of rules that may contain negative conditions, i.e., consequences may be drawn if a condition is not satisfied. However, the existence of a least model for normal programs is not guaranteed. It may be that a program has several minimal models.

Several proposals for assigning models to normal logic programs have been made. Among them are the perfect model semantics [Prz88, Prz89] and the weakly perfect model semantics [PP88], that are applicable only to a restricted classes of normal programs, or the stable model semantics [GL88]. The stable model semantics assigns to every normal logic program a set of models. However, it has been shown in [MT91] that even for a propositional logic program, determining whether it has a stable model is NP-complete.

The well-founded semantics [VGRS91] has been accepted by many researchers as one of the most relevant semantics for logic-based information systems for the following reasons. It assigns a unique (partial) model to every normal logic program, i.e. with unrestricted negation as failure, and the time complexity of computing the well-founded model of a given intensional database, i.e., a set of rules, is polynomial in the size of the extensional database, i.e., the number of facts. We will introduce the concept of the well-founded semantics in Chapter 3.

The well-founded semantics is also well-suited for modern applications. It may be argued that the class of normal logic programs with unrestricted recursion and negation is not needed in practically relevant applications. Negative cyclic dependencies in a specification may look like an indicator of a specification failure, thus one could be tempted to simply forbid cyclic dependencies of a certain kind. However, for any class

of restricted programs, one will find an application that cannot be described within this class. Further, the problem to check whether a given specification satisfies a required restriction is undecidable for most conditions. Another reason to allow the full class of unrestricted programs is the usage of optimizing program transformations that may easily change the containment of a program to a certain class of programs.

1.3 Evaluation Methods

For the computation of the well-founded semantics of normal logic programs several methods have been proposed. We will compare the most prominent algorithms in the following paragraphs. Algorithms for the computation of the well-founded semantics can be divided in two major classes: *top-down* methods, that start with a query and resolve it with rules from the program until the query succeeds or fails, and *bottom-up* methods, that start with facts occurring in the program and apply rules to derive consequences, until the complete model is computed.

SLG Resolution

The SLG resolution [CW93, CSW95, SSW00] is the most prominent *top-down* method. It is a resolution based method like the SLD resolution [Llo87] for definite programs. Starting with a query, a relevant fragment of the well-founded model of a given program is derived. Thus, for query answering w.r.t. the well-founded semantics, the SLG resolution is well-suited. As the implementation [CW96, RRS$^+$99] of the SLG resolution in the XSB system [SSW94, RSS$^+$97] is based on many sophisticated optimizations, this also holds for the XSB system.

The default search strategy of the SLG implementation is a depth first search working tuple-at-a-time to derive single answers. Other search strategies that are more appropriate to process large amounts of data, e.g., provided by an external database system, have been developed [FW97, FSW98]. However, the inherent character of the SLG resolution implies a separate computation of subgoals by several separate search trees. Thus the implementation of alternative, set-oriented search strategies turns out to be difficult in a context of a resolution based evaluation.

Alternating Fixpoint Procedure

The alternating fixpoint procedure [VG89, VG93] is the most prominent *bottom-up* approach for the computation of the well-founded semantics. It computes the complete well-founded model of a given program by applying a sequence of bottom-up fixpoint iterations. Thus, the optimization techniques from deductive databases like the differential fixpoint iteration [BR86a, BR87] are applicable. However, it is well-known that

the alternating fixpoint procedure has efficiency problems in the sense that it needs quadratic evaluation time for programs that could be easily computed in linear time w.r.t. their size. In every iteration many facts have to be recomputed.

Many other approaches [KSS95, Mor96, SS97] are based on the alternating fixpoint procedure, and so they also suffer from the efficiency problem. An improvement of the alternating fixpoint procedure thus will imply an improvement for many bottom-up algorithms.

Magic Set Transformation

Bottom-up methods always compute the complete model of the given program. But to answer a specific query an appropriate fragment of the model would be sufficient. The magic set transformation [BR91, Ram91] is a source-to-source transformation, that adds to a given program a set of filter predicates, so called magic predicates, that control the bottom-up computation such that only relevant answers are computed.

However, it is known that the magic set transformation causes problems in the context of the well-founded semantics [Ros94, KSS95, SS97]. If facts for the magic predicates get an undefined truth value in the well-founded model of the transformed program, the computed answer may not agree with the model of the original program.

Several methods have been proposed to extend the alternating fixpoint procedure such that correct answers are computed for magic set transformed programs [KSS95, Mor96, SS97]. The key idea is to turn undefined magic facts into true facts at different stages of the computation. However, all these methods are based on the alternating fixpoint procedure and thus have the same efficiency problems. Further, it is critical to decide which undefined magic fact should be considered true at which stage.

Residual Program Approach

The Residual Program approach was suggested independently in [Bry89, Bry90] and [DK89a, DK89b], and extended in [BD97, BD98b]. This method applies a set of transformations to a ground program, until a normalform w.r.t. the transformation system, the so called *residual program* is reached. From the residual program the well-founded model of the program can be derived easily.

This method belongs to the class of bottom-up approaches. In the initial program only the facts are obviously true. By applying transformations to the program, more and more rules are turned into facts or deleted from the program, until the complete model is computed.

Obviously, this approach avoids the recomputations of the alternating fixpoint procedure. Starting from a ground program, only reductions are performed. Actually, for

many programs the residual program can be derived in linear time where the alternating fixpoint procedure needs quadratic time w.r.t. the size of the program. However, for other examples the residual program can grow to exponential size, whereas the time and space complexity of the alternating fixpoint procedure is always polynomial. Furthermore, this approach is applicable only to ground programs.

1.4 Contributions of this Thesis

In Section 1.3 we have seen that there are several evaluation methods for the well-founded semantics. Each of these approaches has its advantages and disadvantages. Thus, for practical applications it will not be sufficient to select one static evaluation algorithm and apply it uniformly to any kind of application, rather it is important to combine the advantages of several methods depending on the requirements of the current application. For instance, one application demands to find the first answer fast, whereas another application depends on large amounts of data provided by an external database system.

However, it is difficult to combine evaluation methods that are based on different data structures, applicable to different classes of programs, or whose control strategies are not compatible.

In this work we present a framework that generalizes all major computation approaches using a common data structure and provides a common language to describe their evaluation strategy. In this framework it will be possible to analyze and compare known methods on a high level of abstraction. Then the advantages of the methods can be identified and new algorithms can be composed that combine the strengths of several separate approaches.

Our framework is based on a set of program transformations. In the first part of this work we present a rewriting system for ground programs. For this ground transformation we will show the following results:

- The ground transformation system is strongly terminating and confluent. I.e., by applying the transformations to a given program in any order, the uniquely determined normalform is always reached. We call this normalform the *program remainder*.
- As opposed to the residual program, that may get exponential, we guarantee that our program remainder can be derived in polynomial time.
- As compared to the alternating fixpoint procedure, our transformation approach can avoid the recomputation of many facts. We guarantee that the computation of the remainder of any program does not need more time than the execution of the alternating fixpoint procedure for this program.

- We show that for many programs for which the alternating fixpoint procedure needs quadratic time, the remainder can be derived in linear time w.r.t. the size of the program.
- We show that magic set transformed programs are evaluated correctly by our approach, and that the extensions of the alternating fixpoint procedure for magic programs can be described and even optimized within our framework.
- We define a regular expression like syntax to declaratively describe evaluation strategies just by specifying orders of program transformations. We show that many of the known algorithms can be specified as a special case of our transformation system.
- We propose an intelligent grounding algorithm to derive relevant ground instances of a given non-ground program, such that the ground transformation can also be applied to non-ground programs.
- We provide concrete algorithms that allow one to efficiently implement the grounding algorithm and the transformation framework.
- We illustrate by experimental results that the theoretical comparisons carry over to the prototype implementation.

In the second part of this work we extend the ground transformation to a goal-directed computation scheme for non-ground programs. The following results are shown for the non-ground transformation:

- Although the non-ground transformation is not confluent, any irreducible (and non-floundered) transformation state for a program and a query contains a fragment of the well-founded model that is sufficient to answer the query correctly.
- We show that for a large class of computations of the SLG resolution the non-ground transformation is able to perform exactly the same computation.
- Further, we show that by our non-ground transformation queries can be answered, that for the SLG resolution gets floundered or does not terminate.
- We show that the non-ground transformation fully subsumes the ground transformation, i.e., all results for the ground transformation also apply to the non-ground case.
- We extend the regular strategy expression syntax to the non-ground case. This enables the specification of combined strategies mixing top-down and bottom-up, ground and non-ground, tuple-oriented and set-oriented components.

- We provide a concrete algorithm to implement the non-ground transformation. We illustrate by experimental results, that the non-ground transformation is not just a theoretical concept but can actually be implemented efficiently.

These results show that our approach is a valuable tool to analyze, compare, and optimize existing evaluation methods or to create new strategies that are automatically proven to be correct if they can be described by a sequence of transformations in our framework. As developments of top-down and bottom-up methodologies have converged in the past, this work will close the gap and serve as a basis for an integration of resolution-based, fixpoint-based, and transformation-based evaluation methods.

The ground transformation approach has been published first in [ZFB96, BFZ97, ZBF97]. Later results including the *magic reduction* and the regular strategy expressions are described in [BDFZ01]. The non-ground transformation has already been published in [ZF99].

1.5 Overview

This work is organized as follows. In Chapter 2 basic definitions for logic programs and the semantics of definite programs are provided.

The well-founded semantics is introduced in Chapter 3. We give a reformulation of the alternating fixpoint procedure and discuss the computational complexity of the well-founded semantics.

In Chapter 4 we propose the key concept of the work: the ground transformation approach. We define transformations that will be generalized to non-ground programs later, and show the confluence properties of the approach.

Chapter 5 discusses several grounding methods that are needed if we want to apply ground transformations to a non-ground program.

In Chapter 6 we give a precise comparison of the ground transformation and the alternating fixpoint procedure. We show, that the computation of the alternating fixpoint procedure can be imitated exactly by the transformation. Then we define the regular expression syntax to describe transformation strategies. On this high level of abstraction it is easy to specify a strategy that is significantly more efficient than the alternating fixpoint procedure. We will illustrate this by experimental results.

We extend the non-ground transformation by a new transformation to correctly evaluate magic set transformed programs in Chapter 7. We will, again using the regular strategy expression syntax, describe, compare and even optimize the magic extensions of the alternating fixpoint procedure of [KSS95] and [Mor96].

The prototypical implementation of the ground transformation is described in Chapter 8. We present the key algorithms that have been used for the implementation.

In Chapter 9 we define the extension of the ground transformation to non-ground programs. Therefore we generalize the ground transformation and add two transformations for a goal-directed activation and instantiation of rules.

An extension of the ground implementation to non-ground programs is presented in Chapter 10.

Chapter 11 provides a precise comparison to the SLG resolution. We prove an exact correspondence for a class of computations, but we also point out differences of the two approaches. We extend the regular strategy expression syntax to the non-ground case and give examples for several transformation strategies.

We conclude this work in Chapter 12.

Chapter 2

Normal Logic Programs

2.1 Syntax

To define the syntax of normal programs we recall the definition of a first order logic language that we adopted from [Llo87].

Definition 2.1.1 (Alphabet) An *alphabet* Σ consists of the following classes of symbols:

1. *Variables* which will be denoted by identifiers starting with a capital letter like U, V, W, X, Y, Z.
2. *Function symbols* which will be denoted by identifiers starting with a lower case letter like f, g, h. Each function symbol has an *arity* $n \in \mathbb{N}$. Function symbols with arity 0 are called *constants*. Natural numbers are also constants.
3. *Predicate symbols* which will be denoted by identifiers starting with a lower case letter like p, q, r, s. Each predicate symbol has an *arity* $n \in \mathbb{N}$.
4. *Connectives* $\mathbf{not}$, $\wedge$ (and), $\vee$ (or), $\rightarrow$ (implies), and $\leftrightarrow$ (equivalent).
5. *Quantifiers* $\forall$ (forall) and $\exists$ (exists).
6. *Punctuation symbols* "(", ")", and ",".

□

Inductive Definition 2.1.2 (Term) A *term* is defined inductively as follows:

1. A variable is a term.

2. A constant is a term.

3. If f is a function symbol with arity n and $t_1, \ldots, t_n$ are terms, then $f(t_1, \ldots, t_n)$ is a term.

We will denote the *syntactical equality* of terms by the symbol $\equiv$. □

Definition 2.1.3 (Atom) Let p be a predicate symbol with arity $n \in \mathbb{N}$. Let $t_1, \ldots, t_n$ be terms. Then $p(t_1, \ldots, t_n)$ is an *atomic formula* or *atom*. □

Inductive Definition 2.1.4 (Well-Formed Formula) A *(well-formed) formula* is defined inductively as follows:

1. An atom is a formula.

2. If F and G are formulas, then so are $(\mathbf{not}\ F)$, $(F \wedge G)$, $(F \vee G)$, $(F \rightarrow G)$, and $(F \leftrightarrow G)$.

3. If F is a formula and X is a variable, then $(\forall X F)$ and $(\exists X F)$ are formulas.

It will often be convenient to write the formula $(F \rightarrow G)$ as $(G \leftarrow F)$. We will denote the *syntactical equality* of formulas by the symbol $\equiv$. □

Definition 2.1.5 (Precedences) We will omit parenthesis where possible, using the following precedences of connectives and quantifiers:

1. $\mathbf{not}$, $\forall$, $\exists$

2. $\wedge$, $\vee$

3. $\rightarrow$, $\leftrightarrow$

□

Definition 2.1.6 (First Order Language) The *first order language* $\mathcal{L}$ for an alphabet Σ consists of the set of all formulas that can be constructed according to Definition 2.1.4 using symbols of Σ. □

Definition 2.1.7 (Bound and Free Occurrences of Variables) Let X be a variable and F be a formula. The *scope* of $\forall X$ in $\forall X F$ and of $\exists X$ in $\exists X F$ is F. A *bound occurrence* of a variable in a formula is an occurrence immediately following a quantifier or an occurrence within the scope of a quantifier, which has the same variable immediately after the quantifier. Any other occurrence of a variable is *free*. □

Definition 2.1.8 (Closed Formula) A *closed formula* is a formula with no free occurrence of any variable. □

Definition 2.1.9 (Literal) A *literal* is a positive literal or a negative literal. A *positive literal* is an atom A. A *negative literal* is the negation $\mathbf{not}\, A$ of an atom A. □

Definition 2.1.10 (Complement) Let L be a literal. The *complement* $\sim L$ of L is defined as follows:

$$\sim L \;:\equiv\; \begin{cases} \mathbf{not}\, B & \text{if } L \equiv B \\ B & \text{if } L \equiv \mathbf{not}\, B \end{cases}$$

For a set S of literals, $\sim S$ denotes the set of the complements of the literals in S:

$$\sim S \;:\equiv\; \{\sim L \mid L \in S\}$$

□

Definition 2.1.11 (Positive and Negative Atoms) For a set S of literals we define the set $pos(S)$ of *positively occurring atoms* in S and the set $neg(S)$ of *negatively occurring atoms* in S as follows:

$$\begin{array}{lcl} pos(S) & := & \{A \mid A \in S \text{ is a positive literal in } S\}, \\ neg(S) & := & \{A \mid \mathbf{not}\, A \in S \text{ is a negative literal in } S\}. \end{array}$$

□

Definition 2.1.12 (Clause) A *clause* is a formula of the form

$$\forall X_1 \ldots \forall X_n (L_1 \vee \ldots \vee L_m)$$

where each L_i is a literal and $X_1, \ldots, X_n$ are all the variables occurring in $L_1 \vee \ldots \vee L_m$. □

Definition 2.1.13 (Rule) The *program clause* or *rule*

$$A_1, \ldots, A_k \leftarrow B_1, \ldots, B_l, \mathbf{not}\, C_1, \ldots, \mathbf{not}\, C_m$$

or equivalently

$$A_1 \vee \ldots \vee A_k \leftarrow B_1 \wedge \ldots \wedge B_l \wedge \mathbf{not}\, C_1 \wedge \ldots \wedge \mathbf{not}\, C_m$$

is a convenient notation for the clause

$$\forall X_1 \ldots \forall X_s (A_1 \vee \ldots \vee A_k \vee \mathbf{not}\, B_1 \vee \ldots \vee \mathbf{not}\, B_l \vee C_1 \vee \ldots C_m)$$

where $A_1, \ldots, A_k, B_1, \ldots, B_l, C_1, \ldots, C_l$ are atoms and $X_1, \ldots, X_s$ are all the variables occurring in these atoms. Thus, in a program clause, all variables are assumed to be universally quantified, the commas in the antecedent denote conjunction and the commas in the consequent denote disjunction. □

Definition 2.1.14 (Query) A *query* is a rule with $k = 0$, i.e., a rule of the form

$$\leftarrow B_1, \ldots, B_l, \mathbf{not}\, C_1, \ldots, \mathbf{not}\, C_m$$

having no atom in the consequent. □

Definition 2.1.15 (Goal) A *goal* is a conjunction of literals denoted by

$$B_1 \wedge \ldots \wedge B_l, \mathbf{not}\, C_1 \wedge \ldots \wedge \mathbf{not}\, C_m$$

or equivalently

$$B_1, \ldots, B_l, \mathbf{not}\, C_1, \ldots, \mathbf{not}\, C_m.$$

□

Remark 2.1.16 (Atomic Queries and Goals) A query or a goal is called *atomic* if it consists of a single atom B_1, i.e., $l = 1$ and $m = 0$. For atomic queries, we will often write only B_1 to denote the query $\leftarrow B_1$ if the meaning is clear from the context. □

Definition 2.1.17 (Normal Rule) A *normal rule* is a rule with $k = 1$, i.e., a rule of the form

$$A \leftarrow B_1, \ldots, B_l, \mathbf{not}\, C_1, \ldots, \mathbf{not}\, C_m$$

having exactly one atom in the consequent. □

Definition 2.1.18 (Normal Program) A normal program is a finite set of normal rules. □

Remark 2.1.19 In this work we consider only normal programs. Thus, by speaking of rules and programs we always mean normal rules and normal programs. □

Definition 2.1.20 (Language of a Program) Let P be a program. Then the language $\mathcal{L}_P$ induced by the program is the first order language built from the predicate symbols, function symbols and constants occurring in P. □

Definition 2.1.21 (Rule Head and Rule Body) Let

$$A \leftarrow B_1, \ldots, B_l, \mathbf{not}\, C_1, \ldots, \mathbf{not}\, C_m$$

be a (normal) rule. Then A is called the *rule head* which consists of the *head atom* A. Further, $B_1, \ldots, B_l, \mathbf{not}\, C_1, \ldots, \mathbf{not}\, C_m$ is called the *rule body* and the literals in it are called the *body literals*. The $B_1, \ldots, B_l$ are called *positive body literals*, the $\mathbf{not}\, C_1, \ldots, \mathbf{not}\, C_m$ are called *negative body literals*. □

Definition 2.1.22 (Set Notation for Rules) We frequently treat the body of a rule as a set of literals. Instead of the rule

$$A \leftarrow B_1, \ldots, B_k, \mathbf{not}\, C_1, \ldots, \mathbf{not}\, C_l.$$

we write

$$A \leftarrow \mathcal{B}$$

with $\mathcal{B} = \{B_1, \ldots, B_k\ \mathbf{not}\, C_1, \ldots, \mathbf{not}\, C_l\}$ being the set of body literals. We also use the notation

$$A \leftarrow \mathcal{B} \wedge \mathbf{not}\, \mathcal{C}$$

with $\mathcal{B} = \{B_1, \ldots, B_k\}$ and $\mathcal{C} = \{C_1, \ldots, C_l\}$ being sets of atoms occurring positively or negatively in the rule body, respectively. □

Note, that these definitions of the set notation are legitimate because the conjunction is commutative.

Definition 2.1.23 (Fact) A *fact* is a (normal) rule without any body literals. We will denote a fact just by its head atom A, or in the set notation by $A \leftarrow \emptyset$. □

Remark 2.1.24 (Unconditional Fact) We call this classical type of fact *unconditional* since we will introduce conditional facts with non-empty rule bodies later in this work (cf. Definition 4.3.1). □

Definition 2.1.25 (Definite Rule) A *definite rule* or *positive rule* is a normal rule of the form

$$A \leftarrow B_1, \ldots, B_l$$

having no negative body literals. □

Definition 2.1.26 (Definite Program) A *definite program* or *positive program* is a finite set of definite rules. For a normal program P, we use P^+ to denote the (sub-)set of definite rules in P. □

Definition 2.1.27 (Facts, Heads, Body Literals) Let P be a program. We define the *set of facts* in P, $facts(P)$ and the *set of heads* in P, $heads(P)$ as follows:

$$\begin{array}{lcl} facts(P) & := & \{A \mid (A \leftarrow \emptyset) \in P\}, \\ heads(P) & := & \{A \mid \text{there is a } \mathcal{B} \text{ such that } (A \leftarrow \mathcal{B}) \in P\}. \end{array}$$

□

Note, that for every P, $facts(P) \subseteq heads(P)$ holds.

Definition 2.1.28 (Groundness) A term, atom, rule or program with no variables occurring in it is called *ground*. □

Definition 2.1.29 (Range-Restriction) A program P is called *range-restricted* if each rule $r \in P$ satisfies the following condition: every variable occurring in r occurs (also) in a positive body literal of r. □

2.2 Substitution and Unification

For non-ground logic programs we need the concept of substitution and unification. We will recall the definitions that are most important for this work.

Definition 2.2.1 (Substitution [Llo87]) Let $\mathcal{L}$ be a first order language. A *substitution* θ in $\mathcal{L}$ is a finite set of the form $\{V_1/t_1, \ldots, V_n/t_n\}$, where each V_i is a variable in $\mathcal{L}$, each t_i is a term in $\mathcal{L}$ distinct from V_i and the variables $V_1, \ldots, V_n$ are pairwise distinct. Each element V_i/t_i is called a *binding* for V_i. θ is called a *ground substitution* if the t_i are all ground terms. θ is called a *variable-pure substitution* if the t_i are all variables. We will omit the language $\mathcal{L}$ when it is clear from the context. □

Definition 2.2.2 (Expression) An *expression* is either a term, a literal or a conjunction or disjunction of literals. A *simple expression* is either a term or an atom. □

Definition 2.2.3 (Instance [Llo87]) Let $\theta = \{V_1/t_1, \ldots, V_n/t_n\}$ be a substitution and E be an expression. Then $E\theta$, the *instance* of E by θ, is the expression obtained from E by simultaneously replacing each occurrence of the variable V_i in E by the term t_i for $i = 1, \ldots, n$. If $E\theta$ is ground, then $E\theta$ is called a *ground instance* of E. □

Definition 2.2.4 (Variant [Llo87]) Let E and F be expressions. We say E and F are *variants* if there exist substitutions θ and σ such that $E = F\theta$ and $F = E\sigma$. We say E is a variant of F or F is a variant of E. □

Definition 2.2.5 (Renaming Substitution [Llo87]) Let E be an expression and S be the set of variables occurring in E. A *renaming substitution* for E is a variable-pure substitution $\{X_1/Y_1, \ldots, X_n/Y_n\}$ such that $\{X_1, \ldots, X_n\} \subseteq S$, the Y_i are pairwise distinct and $(S \backslash \{X_1, \ldots, X_n\}) \cap \{Y_1, \ldots, Y_n\} = \emptyset$. □

Definition 2.2.6 (Composition [Llo87]) Let $\theta = \{U_1/s_1, \ldots, U_m/s_m\}$ and $\sigma = \{V_1/t_1, \ldots, V_n/t_n\}$ be substitutions. Then the *composition* $\theta\sigma$ of θ and σ is the substitution obtained from the set

$$\{U_1/s_1\sigma, \ldots, U_m/s_m\sigma, V_1/t_1, \ldots, V_n/t_n\}$$

by deleting any binding $U_i/s_i\sigma$ for which $U_i = s_i\sigma$ and deleting any binding V_j/t_j for which $V_j \in \{U_1, \ldots, U_m\}$. □

Definition 2.2.7 (Most General Unifier [Llo87]) Let S be a finite set of simple expressions. A substitution θ is called a *unifier* for S if $S\theta$ is a singleton. A unifier for S is called *most general unifier (mgu)* for S if, for each unifier σ of S, there exists a substitution γ such that $\sigma = \theta\gamma$. S is called *unifiable* if there exists a unifier for S. □

Remark 2.2.8 A most general unifier for a set of expressions, if it exists, is unique modulo renaming. □

2.3 Interpretations and Models

In this section we recall the definitions of interpretations and models.

Definition 2.3.1 (Pre-Interpretation [Llo87]) A *pre-interpretation* of a first order language L consists of the following:

1. A non-empty set D, called the *domain* of the pre-interpretation.
2. For each constant in L, the assignment of an element in D.
3. For each function symbol with arity n in L, the assignment of a mapping from D^n to D.

□

Definition 2.3.2 (Interpretation [Llo87]) An *interpretation* I of a first order language L consists of a pre-interpretation J with domain D of L together with the following:

4. For each predicate symbol in L with arity n, the assignment of a mapping from D^n into $\{true, false\}$ (or equivalently, a relation on D^n).

We say I is *based on* J. □

Definition 2.3.3 (Variable Assignment [Llo87]) Let J be a pre-interpretation of a first order language L. A *variable assignment* w.r.t. J is an assignment of an element in the domain of J to each variable in L. □

Inductive Definition 2.3.4 (Term Assignment [Llo87]) Let J be a pre-interpretation of a first order language L with domain D. Let V be a variable assignment. The *term assignment* w.r.t. J and V of the terms in L is defined as follows:

1. Each variable is given its assignment according to V.
2. Each constant is given its assignment according to J.
3. If $t'_1, \ldots, t'_n$ are the term assignments of $t_1, \ldots .t_n$ and f' is the assignment of the function symbol f with arity n, then $f'(t'_1, \ldots, t'_n) \in D$ is the term assignment of $f(t_1, \ldots, t_n)$.

□

Inductive Definition 2.3.5 (Truth Values [Llo87]) Let I be an interpretation of a first order language L with domain D and let V be a variable assignment. Then a formula F in L can be given a *truth value*, *true* or *false* (w.r.t. I and V) as follows:

1. If the formula is an atom $p(t_1, \ldots, t_n)$, then the truth value is obtained by calculating the value of $p'(t'_1, \ldots, t'_n)$ where p' is the mapping assigned to p by I and $t'_1, \ldots, t'_n$ are the term assignments of $t_1, \ldots, t_n$ w.r.t. I and V.
2. If the formula has the form $\mathbf{not}\ F$, $F \wedge G$, $F \vee G$, $F \rightarrow G$, $F \leftrightarrow G$, then the truth value of the formula is given by the following table:

F	G	$\mathbf{not}\ F$	$F \wedge G$	$F \vee G$	$F \rightarrow G$	$F \leftrightarrow G$
true	true	false	true	true	true	true
true	false	false	false	true	false	false
false	true	true	false	true	true	false
false	false	true	false	false	true	true

3. If the formula has the form $\exists X F$, then the truth value of the formula is true if there exists $d \in D$ such that F has truth value true w.r.t. I and $V(X/d)$, where $V(X/d)$ is V except that X is assigned d; otherwise, its truth value is false.
4. If the formula has the form $\forall X F$, then the truth value of the formula is true if, for all $d \in D$, we have that F has truth value true w.r.t. I and $V(X/d)$; otherwise, its truth value is false.

□

Clearly the truth value of a *closed formula* does not depend on the variable assignment. Consequently, we can speak unambiguously of the *truth value of a closed formula w.r.t. an interpretation*. According to its truth value, we say a closed formula is true or false w.r.t. an interpretation.

Definition 2.3.6 (Model [Llo87]) Let I be an interpretation of a first order language L. Then I is a model of a closed formula F, if F is true w.r.t. I. Further, I is a model of a set S of closed formulas, if I is a model of each formula of S. □

2.4 Herbrand Models

Definition 2.4.1 (Herbrand Universe [Llo87]) Let $\mathcal{L}$ be a first order language with at least one constant. The *Herbrand universe* $U_{\mathcal{L}}$ of L is the set of all ground terms, which can be formed out of the constants and function symbols of $\mathcal{L}$. □

Remark 2.4.2 For the case that a given logic language contains no constant, we add a new constant c_0 to it and consider the resulting language $\mathcal{L}$. □

Definition 2.4.3 (Herbrand Base [Llo87]) Let $\mathcal{L}$ be a first order language. The *Herbrand base* $BASE(\mathcal{L})$ of $\mathcal{L}$ is the set of all ground atoms which can be formed by using predicate symbols from $\mathcal{L}$ with ground terms from $U_{\mathcal{L}}$ as arguments. □

Remark 2.4.4 We will also assign a Herbrand universe U_P and a Herbrand base $BASE(P)$ to a program P by assuming that the underlying first order language $\mathcal{L}_P$ consists of exactly the constants, function symbols and predicate symbols occurring in the program. □

Definition 2.4.5 (Herbrand Instantiation [Llo87]) Let $\mathcal{L}$ be a first order language. The *Herbrand instantiation* $ground_{\mathcal{L}}(P)$ of P consists of all ground instances (w.r.t. the Herbrand universe $U_{\mathcal{L}}$) of all rules in P. We will omit the language $\mathcal{L}$ when it is clear from the context. □

Definition 2.4.6 (Herbrand Pre-Interpretation [Llo87]) Let L be a first order language. The *Herbrand pre-interpretation* for L is given by the following:

1. The domain of the pre-interpretation is the Herbrand universe U_L.
2. Constants in L are assigned themselves in U_L.
3. If f is a function symbol in L with arity n, then the mapping $f' : U_L^n \mapsto U_L$ assigned to f is defined by $f'(t_1, \ldots, t_n) := f(t_1, \ldots, t_n)$.

□

Definition 2.4.7 (Herbrand Interpretation [Llo87]) A *Herbrand interpretation* is an interpretation based on a Herbrand pre-interpretation. □

Remark 2.4.8 Since, for Herbrand interpretations, the assignment to constant and function symbols is fixed, it is possible to identify a Herbrand interpretation with a subset of the Herbrand base. For any Herbrand interpretation, the corresponding subset of the Herbrand base is the set of all ground atoms which are true w.r.t. the interpretation. □

Definition 2.4.9 (Herbrand Model [Llo87]) Let L be a first order language and S a set of closed formulas of L. A *Herbrand model* of S is an Herbrand interpretation of L which is a model of S. □

The intended meaning of a logic program is that a formula should be true if it is a logical consequence of the program, i.e., it is true in all models of the program. For *definite* programs this intention leads to a semantics that coincides with the intuition because of the following property.

Proposition 2.4.10 (Model Intersection Property [Llo87]) Let P be a *definite* program. Let M be a non-empty set of Herbrand models for P. Then the intersection $\bigcap M$ of all models is an Herbrand model for P. □

Remark 2.4.11 Since every program P has $BASE(P)$ as an Herbrand model, the set of all Herbrand models for P is always non-empty. □

Definition 2.4.12 (Least Herbrand Model [Llo87]) Let P be a definite program. Then the *least Herbrand model* M_P of P is the intersection of all Herbrand models for P. □

The least Herbrand model M_P is considered as the natural interpretation of a definite program.

2.5 Immediate Consequences

To compute the least model of a definite program constructively, we define the following *immediate consequence operator*.

Definition 2.5.1 (Immediate Consequence Operator) Let P be a definite program. Let $I \subseteq BASE(P)$ be a set of atoms. The set of *immediate consequences of I w.r.t. P* is defined as follows:

$$T_P(I) := \{A \mid \text{there is } A \leftarrow \mathcal{B} \in ground(P) \text{ with } \mathcal{B} \subseteq I\}.$$

□

Remark 2.5.2 (Lattices and Fixpoints) A detailed description of the theory of lattices and fixpoints can be found in [Llo87]. We recall some definitions that are needed for this work. We restrict ourselves to the special case of set inclusion lattices where the bottom element is the empty set and the lub and glb operations are given by the set union and set intersection, respectively. □

The T_P operator is monotonic in the following sense.

Definition 2.5.3 (Monotonic Mapping) Let $T : L \mapsto L$ be a mapping. We say T is *monotonic* if $T(x) \subseteq T(y)$, whenever $x \subseteq y$. □

We start with the empty set and then derive more and more consequences by iteratively applying the T_P operator.

Definition 2.5.4 (Ordinal Powers of T) Let $T : L \mapsto L$ be a monotonic mapping. Then we define

$$\begin{array}{lcll} T \uparrow 0 & = & \emptyset & \\ T \uparrow \alpha & = & T(T \uparrow (\alpha - 1)) & \text{if } \alpha \text{ is a successor ordinal} \\ T \uparrow \alpha & = & \bigcup\{T \uparrow \beta \mid \beta < \alpha\} & \text{if } \alpha \text{ is a limit ordinal} \end{array}$$

□

Definition 2.5.5 (Fixpoint) Let $T : L \mapsto L$ be a mapping. An element $a \in L$ is called *fixpoint* of T if and only if $T(a) = a$ holds. □

Definition 2.5.6 (Least Fixpoint) Let $T : L \mapsto L$ be a mapping. An element $a \in L$ is called *least fixpoint* of T if and only if a is a fixpoint of T and for all fixpoints b of T, we have $a \subseteq b$. We will denote this least fixpoint, if it exists, by $\mathrm{lfp}(T)$. □

It is known that for monotonic mappings the least fixpoint does exist and will be reached for some ordinal.

Proposition 2.5.7 (Fixpoints of Monotonic Mappings [Llo87]) Let $T : L \mapsto L$ be a monotonic mapping. Then T has a least fixpoint, $lfp(T)$. For every ordinal α, $T \uparrow \alpha \subseteq lfp(T)$. Furthermore, there exists an ordinal β such that $\gamma \geq \beta$ implies $T \uparrow \gamma = lfp(T)$. □

The T_P operator is not only monotonic but also continuous (in the sense of [Llo87, Chap. 1, §5]. This leads to the following result, stating that the least fixpoint of the T_P operator is always reached not later than at the first limit ordinal.

Theorem 2.5.8 (Fixpoint Characterization of the Least Herbrand Model [Llo87]) Let P be a definite program. Then $M_P = \mathrm{lfp}(T_P) = T_P \uparrow \omega$. □

Chapter 3

The Well-Founded Semantics

In this chapter we will recall the definition of the well-founded semantics and its most prominent computation algorithm, the alternating fixpoint procedure.

3.1 Well-Founded Models

Definition 3.1.1 (Partial and Total Interpretation [VGRS91]) Let P be a normal program. A *partial interpretation* I is a set of ground literals such that for no atom A both A and $\mathbf{not}\ A$ are contained in I, i.e.

$$pos(I) \cap neg(I) = \emptyset,$$

and whose atoms are contained in the Herbrand base $BASE(P)$ of P, i.e.

$$pos(I) \cup neg(I) \subseteq BASE(P).$$

I is a *total interpretation*, if I is a partial interpretation and for every atom $A \in BASE(P)$ it contains A or $\mathbf{not}\ A$, i.e.

$$pos(I) \cup neg(I) = BASE(P).$$

□

Definition 3.1.2 (Truth Values [VGRS91]) Let I be a partial interpretation. A ground literal L is *true* in I, if L is contained in I, i.e. $L \in I$. L is *false* in I, if its complement $\sim L$ is contained in I, i.e. $\sim L \in I$. L is *undefined* in I, if it is neither true nor false in I. □

Definition 3.1.3 (Restricted Interpretation) Let I be a partial interpretation. Let Q be an atom. Then the interpretation I restricted to Q is defined by

$$I|_Q \ := \ I \cap (ground(Q) \cup ground(\mathbf{not}\, Q))$$

□

An interpretation I restricted to Q contains the information as to what ground instances (w.r.t. a given first order language) of Q are true or false in I.

Definition 3.1.4 Two partial interpretations I_1 and I_2 are said to agree on an atom Q, if they contain the same ground instances of Q and $\mathbf{not}\, Q$, i.e., if $I_1|_Q = I_2|_Q$ holds. □

Definition 3.1.5 (Partial and Total Model [VGRS91]) Let P be a normal program. Let I be a partial or total interpretation. A ground rule $A \leftarrow \mathcal{B}$ is *satisfied* in I, if its head A is true in I or if at least one body literal $L \in \mathcal{B}$ is false in I. I is a *total model* of P, if it is a total interpretation and every ground instance $A \leftarrow \mathcal{B} \in ground(P)$ of a rule of P is satisfied in I. I is a *partial model* of P, if it is a partial interpretation that can be extended (by adding literals to I) to a total model of P. □

Remark 3.1.6 (Partial Model [VGRS91]) A partial model is a partial interpretation such that some instantiated rules may not be satisfied, but there is a (possibly empty) set of literals whose addition to the partial interpretation will satisfy all rules. □

Definition 3.1.7 (Unfounded Set [VGRS91]) Let P be a normal program. Let I a partial interpretation. Let $\mathcal{A} \subseteq BASE(P)$ be a set of ground atoms. $\mathcal{A}$ is an *unfounded set* of P w.r.t. I, if for every atom $A \in \mathcal{A}$ and every ground rule instance $A \leftarrow \mathcal{B} \in ground(P)$ at least one of the following conditions holds:

1. at least one body literal $L \in \mathcal{B}$ is false in I,
2. at least one positive body literal $B \in \mathcal{B}$ is contained in $\mathcal{A}$.

□

Definition 3.1.8 (Greatest Unfounded Set [VGRS91]) Let P be a normal program. Let I be a partial interpretation. The *greatest unfounded set* of P w.r.t. I is the union of all unfounded sets of P w.r.t. I. □

Definition 3.1.9 (Positive and Negative Immediate Consequences [VGRS91]) Let P be a normal program. Let I be a partial interpretation. The three operators T_P, U_P, and W_P are defined as follows:

$$\begin{array}{lll} T_P(I) & := & \{A \in BASE(P) \mid \exists (A \leftarrow \mathcal{B}) \in ground(P) : \mathcal{B} \subseteq I\} \\ U_P(I) & := & \text{the greatest unfounded set of } P \text{ w.r.t. } I \\ W_P(I) & := & T_P(I) \cup \sim U_P(I) \end{array}$$

□

Remark 3.1.10 Note, that the T_P operator in Definition 3.1.9 is defined for normal programs and is applicable to sets of literals. This generalizes the T_P operator of Definition 2.5.1 which is defined only for definite programs and is applicable only to sets of atoms. □

Lemma 3.1.11 ([VGRS91]) T_P, U_P, and W_P are monotonic operators. □

Theorem 3.1.12 ([VGRS91]) Let P be a normal program. For every countable ordinal α, $W_P \uparrow \alpha$ is a partial model of P. □

Definition 3.1.13 (Partial and Total Well-Founded Model [VGRS91]) Let P be a program. The *well-founded (partial) model* of P, W_P^*, is the least fixpoint of W_P. If W_P^* is a total interpretation, it is called the *well-founded total model* of P. □

Definition 3.1.14 (Semantics) A *semantics* is a mapping $\mathcal{S}$, which assigns to every logic program P a set $\mathcal{S}(P)$ of (partial) models of P such that $\mathcal{S}$ is "instantiation invariant", i.e. $\mathcal{S}(P) = \mathcal{S}(ground(P))$. □

Now we can define the *Well-Founded Semantics*.

Definition 3.1.15 (Well-Founded Semantics) The *Well-Founded Semantics* assigns to every logic program P the well-founded partial model W_P^* of P:

$$\mathrm{WFS}(P) := \{W_P^*\}$$

□

The well-founded semantics is defined as a set of ground literals. However, we will assign a truth value to a non-ground literal L w.r.t. the well-founded semantics, if any ground instance of L has the same truth value in W_P^*. With any ground instance we mean, that any first order language $\mathcal{L}$ can be used for the grounding and that the truth value of the instances do not depend on the constants occurring in P.

Definition 3.1.16 (Truth Values for Non-Ground Literals) Let P be a normal program. Let L be a non-ground literal. We say that L is true or false or undefined w.r.t. the well-founded semantics of P, if for any first order language $\mathcal{L}$ including the language $\mathcal{L}_P$ of the program, all ground literals in $ground_{\mathcal{L}}(L)$ are true or false or undefined, respectively, in the well-founded model of $ground_{\mathcal{L}}(P)$, where $ground_{\mathcal{L}}$ denotes that for the grounding operation all ground terms of the language $\mathcal{L}$ are used (cf. Definition 2.4.5). □

3.2 Alternating Fixpoint Procedure

Let us recall the definition of the alternating fixpoint procedure. We introduce an extended version of the immediate consequence operator that uses two different sets of facts for positive and negative subgoals, respectively. Actually, this is an adaption of the stability transformation found in [VG89, VG93] to our purposes. This operator has been introduced and investigated by Przymusinski in [Prz89, Prz90, Prz91].

Definition 3.2.1 (Extended Immediate Consequence Operator) Let P be a normal logic program. Let I and J be sets of ground atoms. The set $T_{P,J}(I)$ of *immediate consequences of I w.r.t. P and J* is defined as follows:

$$T_{P,J}(I) \;:=\; \{A \mid \text{there is } A \leftarrow \mathcal{B} \in ground(P) \text{ with } pos(\mathcal{B}) \subseteq I \text{ and } neg(\mathcal{B}) \cap J = \emptyset\}.$$

If P is definite, the set J is not needed and we obtain the standard immediate consequence operator T_P of Definition 2.5.1, i.e., $T_P(I) = T_{P,\emptyset}(I)$. □

$T_{P,J}$ checks negative subgoals against the set of possibly true atoms that are supplied by the argument J. This allows the following elegant formulation of the alternating fixpoint procedure.

Definition 3.2.2 (Alternating Fixpoint Procedure) Let P be a normal logic program. Let P^+ denote the subprogram consisting of only the definite rules of P. Then the sequence $(K_i, U_i)_{i\geq 0}$ with sets K_i of true (known) facts and U_i of possible (unknown) facts is defined by:

$$\begin{array}{lrcl} & K_0 & := & \mathrm{lfp}(T_{P^+}) \\ & U_0 & := & \mathrm{lfp}(T_{P,K_0}) \\ i > 0: & K_i & := & \mathrm{lfp}(T_{P,U_{i-1}}) \\ i > 0: & U_i & := & \mathrm{lfp}(T_{P,K_i}). \end{array}$$

The computation terminates when the sequence becomes stationary, i.e., when a fixpoint is reached in the sense that

$$(K_j, U_j) = (K_{j+1}, U_{j+1}).$$

This computation schema is called the *Alternating Fixpoint Procedure* (AFP). □

Theorem 3.2.3 (Termination [VG93]) Let P be a normal program based on a first order language without function symbols. Let the sequence $(K_i, U_i)_{i\geq 0}$ be defined as above. Then there is a $j \geq 0$ such that $(K_j, U_j) = (K_{j+1}, U_{j+1})$. □

Theorem 3.2.4 (Partial Correctness of AFP [VG93]) Let P be a normal program. Let the sequence $(K_i, U_i)_{i\geq 0}$ be defined as above. If there is a $j \geq 0$ such that $(K_j, U_j) = (K_{j+1}, U_{j+1})$, then the well-founded model W_P^* of P can be directly derived from the fixpoint (K_j, U_j), i.e.,

$$W_P^* = \{L \mid L \text{ is a positive ground literal and } L \in K_j \text{ or}$$
$$L \text{ is a negative ground literal } \mathbf{not}\, A \text{ and } A \in BASE(P)\backslash U_j\}. \quad \square$$

The following results which follow easily from the anti-monotonicity of $T_{P,J}$ w.r.t. J are also stated in [KSS95].

Lemma 3.2.5 (Monotonicity) Let the sequence $(K_i, U_i)_{i\geq 0}$ be defined as above. Then $K_i \subseteq K_{i+1},\ U_i \supseteq U_{i+1},\ K_i \subseteq U_i$ holds for $i \geq 0$. $\square$

Whereas the sets K_i of true facts can be computed incrementally, the sets U_i of possible facts have to be recomputed in every iteration step. It is well-known that this leads to many unnecessary re-computations even for simple programs.

Example 3.2.6 (Even Numbers) Consider the following logic program $EvenNum$

$$\begin{array}{lcl} even(0). & & \\ even(1) & \leftarrow & \mathbf{not}\, even(0). \\ even(2) & \leftarrow & \mathbf{not}\, even(1). \\ even(3) & \leftarrow & \mathbf{not}\, even(2). \\ & \vdots & \\ even(n) & \leftarrow & \mathbf{not}\, even(n-1). \end{array}$$

It defines the even numbers between 0 and some fixed even number n. We would expect that it is possible to compute the well-founded model of this program in linear time w.r.t. n. The alternating fixpoint procedure derives the following sequence:

$$\begin{array}{rcl} K_0 & = & \{even(0)\} \\ U_0 & = & \{even(0), even(2), even(3), \ldots, even(n)\} \\ K_1 & = & \{even(0), even(2)\} \\ U_1 & = & \{even(0), even(2), even(4), even(5), \ldots, even(n)\} \\ K_2 & = & \{even(0), even(2), even(4)\} \\ U_2 & = & \{even(0), even(2), even(4), even(6), even(7), \ldots, even(n)\} \\ & \vdots & \\ K_{n/2} = U_{n/2} & = & \{even(0), even(2), even(4), \ldots, even(n)\}. \end{array}$$

Apparently, the alternating fixpoint approach needs n iterations, each deriving a number of facts that is linear in n. So the total time complexity is at least quadratic in n. $\square$

3.3 Fitting Reduction

In [Fit85] an operator is introduced that is relevant for this work. We will recall its definition in this section.

Definition 3.3.1 (Fitting Reduction) Let P be a normal program. Let I be a partial interpretation. The three operators T_P (true consequences), F_P (obviously false consequences), and the *Fitting operator* Φ_P are defined as follows:

$$\begin{array}{lcl} T_P(I) & := & \{A \in BASE(P) \mid \exists (A \leftarrow \mathcal{B}) \in ground(P) : \mathcal{B} \subseteq I\} \\ F_P(I) & := & \{A \in BASE(P) \mid \forall (A \leftarrow \mathcal{B}) \in ground(P) : \exists L \in \mathcal{B} : \sim L \in I\} \\ \Phi_P(I) & := & T_P(I) \cup \sim F_P(I) \end{array}$$

□

The T_P operator is defined exactly as in Definition 3.1.9. The F_P operator is a simplified version of the U_P operator of Definition 3.1.9 in the following sense.

Lemma 3.3.2 Let P be a normal program. Let I be a partial interpretation. Then $F_P(I) \subseteq U_P(I)$ holds.

Proof: Let $A \in F_P(I)$ be any atom in $F_P(I)$. Then for any ground rule $(A \leftarrow \mathcal{B}) \in ground(P)$ at least one literal $L \in \mathcal{B}$ is false in I, by Definition 3.3.1. By Definition 3.1.7 the set $\{A\}$ is an unfounded set. It follows $\{A\} \subseteq U_P(I)$ and thus $A \in U_P(I)$. □

The $F_P(I)$ operator determines all atoms A that are obviously false because all rule instances that could derive A have already at least one body literal that is false in I. The $U_P(I)$ operator of Definition 3.1.9 contains the same atoms because of the first condition of Definition 3.1.7. But additionally, it contains atoms depending on each other through cyclic positive dependencies. This detection of positive loops is what makes the computation of the well-founded semantics expensive. We will discuss this point more deeply later in this work.

The Φ_P operator is obviously monotonic. Thus its least fixpoint $\mathrm{lfp}(\Phi_P)$ exists.

Lemma 3.3.3 (Fitting Reduction) Let P be a normal program. Then $\mathrm{lfp}(\Phi_P) \subseteq W_P^*$ holds.

Proof: From Definition 3.3.1 and Lemma 3.3.2 it follows that $\Phi_P(I) \subseteq W_P(I)$ holds for any partial interpretation I. From this the lemma follows immediately. □

For any normal program, $\mathrm{lfp}(\Phi_P)$ is a subset of the well-founded model W_P^* of P. Thus it is a possible to start the computation of W_P^* with the computation of $\mathrm{lfp}(\Phi_P)$.

3.4 Computational Complexity

To measure the computational complexity of the well-founded semantics we define the following two attributes of a program.

Definition 3.4.1 (Number of Atoms) Let P be a program. Then $At(P)$ denotes the set of all atoms occurring in P. Consequently, $|At(P)|$ denotes the number of distinct atoms occurring in P. □

Definition 3.4.2 (Size) Let P be a program. Then the *size* of P, denoted by $size(P)$, is the number of literals in P, i.e., the number of rule heads plus the total number of body literals in all rules including duplicates. □

Now we can give an upper bound for the computation time of the well-founded semantics.

Lemma 3.4.3 (Polynomial Time [BSF95]) Let P be a ground program. Then the well-founded model W_P^* of P can be computed in polynomial time, and more specifically, in $O(|At(P)| \times size(P))$ steps.

Proof: This bound is achieved by the alternating fixpoint procedure (cf. Definition 3.2.2). The AFP needs at most $|At(P)|$ iterations until a fixpoint is reached. Each iteration consists of finding the least models of two definite programs derived from P. Since finding the least model of a definite program is linear in the size of the program (cf. Section 8.2), this gives the stated complexity. □

Remark 3.4.4 (Quadratic Time [BSF95]) As a consequence of Lemma 3.4.3, the *worst case* time needed to compute the well-founded model of a ground program is *quadratic* in the size of the program. □

Remark 3.4.5 (Ground Programs) The results in this section are discussed only for ground programs. For many practical purposes, ground programs are not especially interesting by themselves. But semantics for first order logic programs (IDB) over finite extensional databases (EDB) are generally defined by translating, via Herbrand instantiation, these logic programs and databases into ground programs. Also the alternating fixpoint procedure considers, by Definition 3.2.1, only ground rule instances. □

Despite the importance of the well-founded semantics, the question of how fast it can be computed has not attracted significant attention [LT00]. Improving the quadratic bound given in Lemma 3.4.3 turned out to be difficult. There are some approaches that

achieve a better result for special classes of programs. However, to our knowledge, until now no algorithm has been found that breaks the quadratic worst case bound in general.

Berman, Schlipf, and Franco have described algorithms that achieve a speedup for two classes of programs.

Lemma 3.4.6 ([BSF95]) Let P be a ground program whose rules contain at most two occurrences of atoms in their bodies. Then W_P^* can be computed in time $O(|At(P)|^{4/3} size(P)^{2/3})$. □

Lemma 3.4.7 ([BSF95]) Let P be a ground program whose rules have no more than one positive atom in their bodies. Then W_P^* can be computed in time $O(|At(P)|^{3/2} size(P)^{1/2})$. □

Lonc and Truszczynski have described another algorithm for the same class of programs.

Lemma 3.4.8 ([LT00]) Let P be a ground program whose rules have no more than one positive atom in their bodies. Then W_P^* can be computed in time $O(|At(P)|^2 + size(P))$. □

If we analyze how these complexity results have been achieved, it turns out that the cost for computing the well-founded model can be divided into two major parts. The easier part is the reduction of obviously true or false body literals by means of the Fitting reduction (cf. Section 3.3). This reduction is known to be computable in time $O(size(P))$ (cf. Section 8.2). The more difficult part is the detection and removal of cyclic positive dependencies, or positive loops. To break these positive loops, the alternating fixpoint procedure needs at most $|At(P)|$ iterations each of which needs $O(size(P))$ computation steps. Consequently, the alternating fixpoint procedure has a worst case running time of

$$O(size(P) \ + \ |At(P)| \times size(P)).$$

The algorithm of Lonc and Truszczynski takes advantage of the fact that in each rule there is at most one positive body literal. For this special case they describe a loop detection algorithm that needs time $O(|At(P)|)$ to detect one single loop. Thus they achieve an overall running time of

$$O(size(P) \ + \ |At(P)| \times |At(P)|).$$

In this notation it is clearly visible that they reduced the last factor from $size(P)$ to $|At(P)|$ which will be a an advantage for most programs.

In this work we will not break the quadratic worst case bound for normal programs in general. However, we will provide a framework which allows the specification of evaluation strategies that achieve a significant speedup, up to computation in linear time for a large class of programs. As we have seen in the discussion in this section, the detection and removal of positive loops plays a major role in this context. We will reduce the *number* of loop detections needed significantly as compared to the alternating fixpoint for a large class of programs. Further, we will discuss in Section 8.3 how the loop detection can be implemented more efficiently, such that any single loop detection needs sub-linear time leading to an overall time that is sub-quadratic for many programs, even if a linear number of loop detections is needed.

Chapter 4

Ground Transformation

We have seen that the alternating fixpoint procedure needs quadratic time for some programs whose model could be easily computed in linear time. In this chapter we introduce the key concept of this work: model computation by program transformation. We will show that by handling rule instances instead of sets of atoms the computation efficiency can be improved for many programs.

First, we define the program transformation for ground programs only. Later in this work we will generalize the definitions for non-ground programs.

4.1 Basic Definitions

Definition 4.1.1 (Program Transformation) A *program transformation* is a relation $\mapsto$ between ground logic programs. A semantics $\mathcal{S}$ (cf. Definition 3.1.14) allows a transformation $\mapsto$ iff $\mathcal{S}(P_1) = \mathcal{S}(P_2)$ for all P_1 and P_2 with $P_1 \mapsto P_2$. □

Definition 4.1.2 (Transitive Reflexive Closure) Let $\mapsto$ be a program transformation. Then $\mapsto^*$ denotes the transitive reflexive closure of $\mapsto$, i.e., denotes 0 or any positive number of applications of $\mapsto$. □

If a ground atom appears as a fact in a ground program P then this fact is obviously true in this program. Further, if a ground atom does not appear in any rule head in the program, then there is no way to derive this atom and thus the atom is obviously false in this program.

Definition 4.1.3 (True and False Atoms) Let P be a ground program. Let S be a set of ground atoms. Then the set of atoms that are *true* in P w.r.t. S and the set of atoms that are *false* in P w.r.t. S are defined as follows:

$$\begin{array}{lcl} true_S(P) & := & \{A \in S \mid A \in facts(P)\} \\ false_S(P) & := & \{A \in S \mid A \notin heads(P)\} \end{array}$$

Usually the set of S is given by the Herbrand base $BASE(P)$ of the program P. In this case we will omit the subscript S. □

Based on this observation, we assign to every ground program a partial interpretation which contains these obviously true or false atoms.

Definition 4.1.4 (Known Literals) Let P be a ground program and let S be a set of ground atoms. The set of positive and negative ground literals with atoms in S having an obvious truth value in P is denoted by $known_S(P)$:

$$known_S(P) \ := \ true_S(P) \cup \sim false_S(P)$$

□

Lemma 4.1.5 (Soundness of Known Literals) Let P be a ground program. Then $known_{BASE(P)}(P) \subseteq W_P^*$ holds.

Proof: This result is obvious from the definition of the well-founded semantics. From Definition 3.1.9 is follows that

$$true_{BASE(P)}(P) = facts(P) \subseteq T_P(\emptyset) \subseteq W_P(\emptyset) \subseteq W_P^*.$$

According to Definition 3.1.7 $false_{BASE(P)}(P)$ is an unfounded set of P w.r.t. the empty interpretation, and thus we have

$$false_{BASE(P)}(P) \subseteq U_P(\emptyset)$$

and as a consequence

$$\sim false_{BASE(P)}(P) \subseteq W_P(\emptyset) \subseteq W_P^*.$$

□

The main idea of the program transformation is to start with a given ground program P and apply program transformations to it until the set of known literals $known_{BASE(P)}(P')$ of the resulting program P' contains not only a subset of the well-founded model as stated in Lemma 4.1.5 but the complete model.

4.2 The Residual Program Approach

Brass and Dix [BD94, BD97, BD98a, BD98b] have introduced a framework for studying and computing negation semantics by means of elementary program transformations. This framework, called the *residual program approach*, defines a set of program transformations in the sense of Definition 4.1.1. The first transformation is the deletion of tautological rules like $p \leftarrow p \wedge q$ where the head atom also occurs as a positive body literal.

Definition 4.2.1 (Deletion of Tautologies) Let P_1 and P_2 be ground programs. Program P_2 results from program P_1 by *deletion of tautologies* ($P_1 \ \mapsto_T \ P_2$) iff there is $A \leftarrow \mathcal{B} \ \in \ P_1$ such that $A \in \mathcal{B}$ and $P_2 = P_1 \backslash \{A \leftarrow \mathcal{B}\}$. □

Another program transformation is the "unfolding" of a body literal. For instance, consider the program

$$p \leftarrow q \wedge \mathbf{not}\, r.$$

Suppose that there are two rules for q, namely

$$\begin{array}{lll} q & \leftarrow & s_1 \wedge \mathbf{not}\, t_1. \\ q & \leftarrow & s_2 \wedge \mathbf{not}\, t_2. \end{array}$$

q can only be derived by applying one of these two rules. Thus we can replace the body literal q by the bodies of the rules for q. This leads to the program

$$\begin{array}{lll} p & \leftarrow & s_1 \wedge \mathbf{not}\, t_1 \wedge \mathbf{not}\, r. \\ p & \leftarrow & s_2 \wedge \mathbf{not}\, t_2 \wedge \mathbf{not}\, r. \end{array}$$

Definition 4.2.2 (Unfolding) Let P_1 and P_2 be ground programs. Program P_2 results from program P_1 by *unfolding* ($P_1 \ \mapsto_U \ P_2$) iff there is a rule $A \leftarrow \mathcal{B}$ in P_1 and a positive literal $B \in \mathcal{B}$ such that

$$P_2 = (P_1 \backslash \{A \leftarrow \mathcal{B}\}) \ \cup \ \{A \leftarrow ((\mathcal{B} \backslash \{B\}) \cup \mathcal{B}') \mid B \leftarrow \mathcal{B}' \ \in \ P_1\}.$$

□

Unfolding is a very powerful transformation that has been studied by many researchers, e.g. [AD95, DM94, SS94, Sek93].

Let $\mapsto_{TU} := \mapsto_T \cup \mapsto_U$ be the rewriting system consisting of the two transformations introduced above. This system is interesting because already *unfolding* and *deletion of tautologies* allow to eliminate all positive body literals [BD95]. Thus the resulting

program is guaranteed to contain no cyclic positive dependencies, or positive loops, since it contains no positive dependencies at all. This is remarkable since the detection and removal of positive loops is the most difficult part of the computation of the well-founded semantics. However, we will show that this kind of loop detection has another disadvantage.

The next transformation is called deletion of non-minimal rules. Consider a program that contains the rule $p \leftarrow q$ and another rule $p \leftarrow q \wedge r$. Then it should be possible to delete the second rule, since it is logically weaker than the first rule.

Definition 4.2.3 (Deletion of Non-minimal Rules) Let P_1 and P_2 be ground programs. Program P_2 results from the program P_1 by *deletion of non-minimal rules* ($P_1 \mapsto_M P_2$) iff there are rules $A \leftarrow \mathcal{B}$ and $A \leftarrow \mathcal{B}'$ in P_1 such that $\mathcal{B} \subset \mathcal{B}'$ and $P_2 = P_1 \backslash \{A \leftarrow \mathcal{B}'\}$. □

The next two transformations describe the evaluation of negative body literals in trivial cases. For instance, if there is no rule with p in the head, there is obviously no way to derive p, so $\mathbf{not}\, p$ should be true. But then it should be possible to delete the condition $\mathbf{not}\, p$ from the body of a rule, since it is true anyway:

Definition 4.2.4 (Positive Reduction) Let P_1 and P_2 be ground programs. P_2 results from P_1 by *positive reduction* ($P_1 \mapsto_P P_2$) iff there is a rule $A \leftarrow \mathcal{B}$ in P_1 and a negative literal $\mathbf{not}\, B \in \mathcal{B}$ such that B is false in P_1 (i.e., $B \notin heads(P_1)$), and $P_2 = (P_1 \backslash \{A \leftarrow \mathcal{B}\}) \cup \{A \leftarrow (\mathcal{B} \backslash \{\mathbf{not}\, B\})\}$. □

On the other hand, if p is given as a fact in the program, a condition of the form $\mathbf{not}\, p$ can never be true. So a rule with $\mathbf{not}\, p$ in its body is useless and it should be possible to delete the complete rule:

Definition 4.2.5 (Negative Reduction) Let P_1 and P_2 be ground programs. P_2 results from P_1 by *negative reduction* ($P_1 \mapsto_N P_2$) iff there is a rule $A \leftarrow \mathcal{B}$ in P_1 and a negative literal $\mathbf{not}\, B \in \mathcal{B}$ such that B is true in P_1 (i.e., $B \in facts(P_1)$), and $P_2 = P_1 \backslash \{A \leftarrow \mathcal{B}\}$. □

Definition 4.2.6 Let $\mapsto_R$ be the rewriting system consisting of the above five transformations, i.e. $\mapsto_R := \mapsto_T \cup \mapsto_U \cup \mapsto_M \cup \mapsto_P \cup \mapsto_N$. □

As shown in [BD98a, BD97] $\mapsto_R$ has some nice properties which we summarize in the following.

Definition 4.2.7 (Normal Form) A program P' is a *normal form* of a program P w.r.t. a transformation $\mapsto$ iff

1. $P \mapsto^* P'$, and
2. P' is irreducible, i.e. there is no program P'' with $P' \mapsto P''$.

□

For $\mapsto_R$ we have the following strong normal form theorem.

Theorem 4.2.8 (Residual Program [BD98b, BD98a]) The rewriting system $\mapsto_R$ is

1. terminating, i.e. every ground program P has a normal form P', and
2. confluent, i.e. for all ground programs P, P_1, P_2 with $P \mapsto_R^* P_1$ and $P \mapsto_R^* P_2$, there is a program P_3 with $P_1 \mapsto_R^* P_3$ and $P_2 \mapsto_R^* P_3$.

□

Definition 4.2.9 (Residual Program) As a consequence of Theorem 4.2.8 every ground program P has a unique normal form, which we call *residual program* $res(P)$ of P. □

In fact, we have an even stronger result.

Definition 4.2.10 (Strong Termination) A rewriting system is *strongly terminating*, iff every chain of transformations is terminating for fair sequences of transformations. □

Lemma 4.2.11 ([BD98a]) The rewriting system $\mapsto_R$ is strongly terminating. □

As shown in [BD98b, BD94], the well-founded semantics is the weakest semantics which allows the above transformations.

Proposition 4.2.12 ([BD98b, BD94]) Let P be any normal program. Let P' be any program such that $P \mapsto_R^* P'$ holds. Then $WFS(P) = WFS(P')$ (cf. Definition 3.1.15). □

We can directly "read off" the well-founded model from the residual program:

Theorem 4.2.13 (Computation of WFS [BD97, BD94, BD98b]) The well-founded model W_P^* of a program P consists of those positive and negative ground literals which have an obvious truth value in the residual program, i.e.,

$$W_P^* = known_{BASE(P)}(res(P)).$$

□

As the following example shows, the residual program approach needs only a linear number of transformation steps, whereas the alternating fixpoint procedure needs a quadratic number of steps in this example.

Example 4.2.14 (Even Numbers revisited) Consider the logic program $EvenNum$ of Example 3.2.6 for some fixed number n. With the program transformations defined above, $EvenNum$ can be transformed into the residual program $res(EvenNum)$ as follows:

$$
\begin{array}{lll}
EvenNum & = & \{ \quad even(0), \\
 & & \quad even(1) \leftarrow \mathbf{not}\, even(0), \\
 & & \quad even(2) \leftarrow \mathbf{not}\, even(1), \\
 & & \quad even(3) \leftarrow \mathbf{not}\, even(2), \\
 & & \quad \ldots, \\
 & & \quad even(n) \leftarrow \mathbf{not}\, even(n-1)\ \} \\
\mapsto_N & \{ & \quad even(0), \\
 & & \quad even(2) \leftarrow \mathbf{not}\, even(1), \\
 & & \quad even(3) \leftarrow \mathbf{not}\, even(2), \\
 & & \quad \ldots, \\
 & & \quad even(n) \leftarrow \mathbf{not}\, even(n-1)\ \} \\
\mapsto_P & \{ & \quad even(0), \\
 & & \quad even(2), \\
 & & \quad even(3) \leftarrow \mathbf{not}\, even(2), \\
 & & \quad \ldots, \\
 & & \quad even(n) \leftarrow \mathbf{not}\, even(n-1)\ \} \\
\mapsto_N & & \quad \ldots \\
\mapsto_P & & \quad \ldots \\
\vdots & & \\
\mapsto_R^* & \{ & \quad even(0), even(2), even(4), \ldots, even(n)\ \}.
\end{array}
$$

Obviously, the residual program $res(EvenNum)$ can be derived in $O(n)$ transformation steps. In contrast, the alternating fixpoint procedure (cf. Example 3.2.6) needs a quadratic number of derivation steps. □

Remark 4.2.15 We compare the number of applied transformations with the number of derivation steps needed by the fixpoint iteration of the alternating fixpoint approach. This is fair since in both cases the truth values of body literals of ground rule instances have to be checked to derive the truth value of the head. Both computation models, program transformation and fixpoint iteration, can be implemented based on the algorithm of Dowling and Gallier (cf. Definition 8.2.1) where the examination of the truth value of a body literal is done in constant time. □

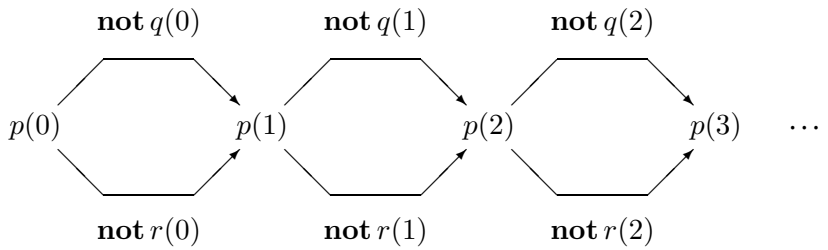
However, whereas the alternating fixpoint procedure is guaranteed to derive a number of facts that is at most quadratic in the size of the Herbrand base of the given program, there are cases where the residual program can grow to exponential size. This problem connected with the delayed evaluation of negative literals was already noted in [CSW95] (cf. Remark 11.3.5).

Example 4.2.16 (Exponential Residual Program) Consider the following program Exp.

$$
\begin{array}{lcl}
p(0). & & \\
p(X) & \leftarrow & p(Y) \wedge succ(Y,X) \wedge \mathbf{not}\, q(Y). \\
p(X) & \leftarrow & p(Y) \wedge succ(Y,X) \wedge \mathbf{not}\, r(Y). \\
q(X) & \leftarrow & succ(X,\_) \wedge \mathbf{not}\, q(X). \\
r(X) & \leftarrow & succ(X,\_) \wedge \mathbf{not}\, r(X).
\end{array}
$$

$$
\begin{array}{l}
succ(0,1). \\
\quad \vdots \\
succ(n-1,n).
\end{array}
$$

The rules for q and r ensure that the corresponding negative body literals must be delayed, i.e., they cannot be reduced but have to be propagated by the *unfolding* transformation. But then the set of delayed literals in the body of a rule for p encodes the derivation path for the p-fact in the head. The following diagram depicts all possible paths on which a p-fact can be derived:



Thus, the residual program contains the following conditional facts for p:

$$
\begin{array}{lcl}
p(0). & & \\
p(1) & \leftarrow & \mathbf{not}\, q(0). \\
p(1) & \leftarrow & \mathbf{not}\, r(0). \\
p(2) & \leftarrow & \mathbf{not}\, q(0) \wedge \mathbf{not}\, q(1). \\
p(2) & \leftarrow & \mathbf{not}\, q(0) \wedge \mathbf{not}\, r(1). \\
p(2) & \leftarrow & \mathbf{not}\, r(0) \wedge \mathbf{not}\, q(1). \\
p(2) & \leftarrow & \mathbf{not}\, r(0) \wedge \mathbf{not}\, r(1). \\
p(3) & \leftarrow & \mathbf{not}\, q(0) \wedge \mathbf{not}\, q(1) \wedge \mathbf{not}\, q(2). \\
p(3) & \leftarrow & \mathbf{not}\, q(0) \wedge \mathbf{not}\, q(1) \wedge \mathbf{not}\, r(2). \\
p(3) & \leftarrow & \mathbf{not}\, q(0) \wedge \mathbf{not}\, r(1) \wedge \mathbf{not}\, q(2). \\
p(3) & \leftarrow & \mathbf{not}\, q(0) \wedge \mathbf{not}\, r(1) \wedge \mathbf{not}\, r(2). \\
p(3) & \leftarrow & \mathbf{not}\, r(0) \wedge \mathbf{not}\, q(1) \wedge \mathbf{not}\, q(2). \\
p(3) & \leftarrow & \mathbf{not}\, r(0) \wedge \mathbf{not}\, q(1) \wedge \mathbf{not}\, r(2). \\
p(3) & \leftarrow & \mathbf{not}\, r(0) \wedge \mathbf{not}\, r(1) \wedge \mathbf{not}\, q(2). \\
p(3) & \leftarrow & \mathbf{not}\, r(0) \wedge \mathbf{not}\, r(1) \wedge \mathbf{not}\, r(2). \\
 & \vdots &
\end{array}
$$

It is obvious that every possible path is encoded in the residual program. Each possible fact $p(n)$ is represented by 2^n conditional facts. □

4.3 The Program Remainder Approach

We now define the main transformations of our ground transformation approach. It is also based on a set of transformations applicable to ground programs, but we guarantee a polynomial complexity, as opposed to the residual program approach described in Section 4.2 that may get exponential.

As we have seen in Example 4.2.16, the residual program can grow to exponential size. The main reason for this is the *unfolding* transformation which may successively replace a positive body literal by an exponential number of combinations of negative literals. In the rewriting system $\mapsto_R$ (Definition 4.2.6) only negative body literals are delayed, positive body literals are eliminated by *unfolding* and *deletion of tautologies*, even if their truth value is not yet known. In contrast to [BD97, BD98b], in this work we propose *not to replace* a positive body literal A by the bodies of rules for A, but to *delay* the processing of positive body literals until their truth value is obvious. This simply means that we do not allow transformations on positive body literals except when they are given as facts (and thus known to be true) or do not occur in any rule head (and thus are known to be false). Thus, we adapt the definition of a conditional fact from [Bry89, Bry90, BD97, BD98b, BD98a] by allowing also positive literals in the condition:

Definition 4.3.1 (Conditional Fact) A *conditional fact* $A \leftarrow \mathcal{C}$ is a ground rule containing possibly both positive and negative literals in the body. □

Remark 4.3.2 In this definition a conditional fact is simply a ground rule instance. This terminology underlines our intention that in our approach sets of conditional facts, i.e., ground rule instances, should replace sets of normal or unconditional facts, i.e., ground atoms (cf. Definition 2.1.23). □

To evaluate delayed negative literals when their truth value becomes known, we have the two transformations *positive reduction* $\mapsto_P$ and *negative reduction* $\mapsto_N$ (cf. Definitions 4.2.4 and 4.2.5). Let us now consider positive body literals. If there is a fact B, we obviously should be able to "simplify away" the condition B in a rule:

Definition 4.3.3 (Success) Let P_1 and P_2 be ground programs. P_2 results from P_1 by *success* ($P_1 \mapsto_S P_2$) iff there is a rule $A \leftarrow \mathcal{B}$ in P_1 and a positive literal $B \in \mathcal{B}$ such that B is true in P_1, and $P_2 = (P_1 \backslash \{A \leftarrow \mathcal{B}\}) \cup \{A \leftarrow (\mathcal{B} \backslash \{B\})\}$. □

Lemma 4.3.4 If a semantics $\mathcal{S}$ allows *unfolding* and the *deletion of non-minimal rules*, then it also allows *success*.

Proof: *Success* corresponds to *unfolding*, when the program contains a fact $B \leftarrow \emptyset$ and B is replaced by $\emptyset$. However, there might be further rules for B besides this fact. But then the resulting rules are certainly non-minimal, so we can remove them with $\mapsto_M$. □

Next, when there is no longer any possibility to prove a literal B, we can remove rule instances depending on B:

Definition 4.3.5 (Failure) Let P_1 and P_2 be ground programs. P_2 results from P_1 by *failure* ($P_1 \mapsto_F P_2$) iff there is a rule $A \leftarrow \mathcal{B}$ in P_1 and a positive literal $B \in \mathcal{B}$ such that B is false in P_1, and $P_2 = P_1 \backslash \{A \leftarrow \mathcal{B}\}$. □

Lemma 4.3.6 If a semantics $\mathcal{S}$ allows *unfolding*, then it also allows *failure*.

Proof: Obviously, the transformation $\mapsto_F$ is a very special case of *unfolding*, i.e., if there are no rules having the unfolded atom as rule head. □

Definition 4.3.7 Let $\mapsto_{PSNF}$ denote the rewriting system which evaluates negative literals by *positive* and *negative reduction* and positive literals by *success* and *failure*:

$$\mapsto_{PSNF} := \mapsto_P \cup \mapsto_S \cup \mapsto_N \cup \mapsto_F.$$

□

It seems to be very natural to apply these transformations. The following lemma confirms its compatibility with the transformations of Section 4.2.

Lemma 4.3.8 Let $\mathcal{S}$ be a semantics which allows *unfolding*, *deletion of non-minimal rules*, and *positive* and *negative reduction*. Then also

$$P_1 \mapsto_{PSNF} P_2 \implies \mathcal{S}(P_1) = \mathcal{S}(P_2)$$

for all ground programs P_1 and P_2.

Proof: The lemma follows immediately from the Lemmata 4.3.4 and 4.3.6. □

Example 4.3.9 (Exponential Residual Program Revisited) Starting from the program of Example 4.2.16, the rewriting system $\mapsto_{PSNF}$ yields the following irreducible rules for p:

$$\begin{array}{lcl} p(0). & & \\ p(1) & \leftarrow & \mathbf{not}\, q(0). \\ p(1) & \leftarrow & \mathbf{not}\, r(0). \\ p(2) & \leftarrow & p(1) \wedge \mathbf{not}\, q(1). \\ p(2) & \leftarrow & p(1) \wedge \mathbf{not}\, r(1). \\ p(3) & \leftarrow & p(2) \wedge \mathbf{not}\, q(2). \\ p(3) & \leftarrow & p(2) \wedge \mathbf{not}\, r(2). \\ & \vdots & \end{array}$$

Each possible fact $p(n)$ (for $n \geq 1$) is represented by exactly 2 rules. If we allow full *unfolding* of the positive body literals, we immediately get the exponential residual program of Example 4.2.16. □

The next lemma and theorem state that the four transformations *success*, *failure*, *positive* and *negative reduction* correspond exactly to the Fitting operator Φ_P (cf. Definition 3.3.1).

Lemma 4.3.10 (Confluence of $\mapsto_{PSNF}$) The rewriting system $\mapsto_{PSNF}$ is confluent and strongly terminating.

Proof: Strong termination is obvious, because all transformations strictly reduce the number of literals occurring in the transformed program.

Confluence is given as a consequence of the following fact. Whenever one of the transformations P, N, S, or F is applicable this is due to the existence of a fact for S and

N or the non-existence of a head atom for F and P. Note that by all of the four transformations no fact can be deleted and no rule with a new head atom can be generated. Thus, the transformation will stay being applicable whatever other transformations are applied first. The only possible conflict is when two transformations concern the same rule instance and at least one of the transformations will delete the instance. But then the result is always the same independent of the order of application: the rule instance will get deleted. □

Definition 4.3.11 (Fitting Normal Form) Let P be a normal program. The uniquely determined normalform of P w.r.t. the rewriting system $\mapsto_{PSNF}$ is called *Fitting normalform* of P, $fitt(P)$. □

Theorem 4.3.12 ($\mapsto_{PSNF}$ corresponds to $\mathrm{lfp}(\Phi_P)$)
Let P be a normal program. The known literals with respect to the Fitting normal form $fitt(P)$ of P constitute exactly the least fixpoint $\mathrm{lfp}(\Phi_P)$ of Fitting's operator Φ_P:

$$\mathrm{lfp}(\Phi_P) = known_{BASE(P)}(fitt(P)).$$

Proof: We define the following sequence of transformations. Let P be a program. Let $I = known_{BASE(P)}(P)$ be the set of known literals in P. Then $f(P)$ is defined to be the result after applying the following transformations to P:

- Remove all body literals L in rule instances in P that are true in I by applying $\mapsto_S$ and $\mapsto_P$.
- In the result delete all rule instances having at least one body literal L that is false in I by applying $\mapsto_F$ and $\mapsto_N$.

Now we show by induction on n that

$$known_{BASE(P)}(f^{\uparrow n}(P)) = \Phi_P^{\uparrow n}(known_{BASE(P)}(P))$$

holds for all $n \geq 0$. For the base case with $n = 0$ we have to show

$$known_{BASE(P)}(P) = known_{BASE(P)}(P)$$

which holds trivially. Now assume that the induction hypothesis holds for n. We have to show that

$$known_{BASE(P)}(f^{\uparrow n+1}(P)) = \Phi_P^{\uparrow n+1}(known_{BASE(P)}(P))$$

holds. Consider any atom $A \in known_{BASE(P)}(f^{\uparrow n+1}(P))$. Then and only then $A \in f^{\uparrow n+1}(P)$ holds. But then and only then there is rule instance $A \leftarrow \mathcal{B}$ in P such that

all body literals $L \in \mathcal{B}$ are true in $f^{\uparrow n}(P)$, i.e., $\mathcal{B} \subseteq known_{BASE(P)}(f^{\uparrow n}(P))$. By the induction hypothesis this is equivalent to $\mathcal{B} \subseteq \Phi_P^{\uparrow n}(known_{BASE(P)}(P))$. But this holds if and only if $A \in \Phi^{\uparrow n+1}(known_{BASE(P)}(P))$ due to the definition of T_P in Definition 3.3.1.

Now consider any negative literal $\mathbf{not}\, A \in known_{BASE(P)}(f^{\uparrow n+1}(P))$. Then and only then $A \notin heads(f^{\uparrow n+1}(P))$ holds. But then and only then for all rule instances of P with A as head there is a body literal L that is false in $f^{\uparrow n}(P)$, i.e., $\sim L \in known_{BASE(P)}(f^{\uparrow n}(P))$. By the induction hypothesis this is equivalent to $\sim L \in \Phi_P^{\uparrow n}(known_{BASE(P)}(P))$ for all such literals L. But this holds if and only if $\mathbf{not}\, A \in \Phi^{\uparrow n+1}(known_{BASE(P)}(P))$ due to the definition of F_P in Definition 3.3.1.

Since P is finite a fixpoint is reached after a finite number of iterations, and let m be this number. Then on the left hand side we have

$$known_{BASE(P)}(f^{\uparrow m}(P)) = known_{BASE(P)}(fitt(P))$$

since the rewriting system $\mapsto_{PSNF}$ is confluent and its normalform $fitt(P)$ is reached by iteratively applying f to P. On the right hand side we have

$$\Phi_P^{\uparrow m}(known_{BASE(P)}(P)) = \Phi_P^{\uparrow m}(\Phi_P(\emptyset)) = \mathrm{lfp}(\Phi_P)$$

Together the theorem is proven. □

In the residual program approach of Section 4.2, where only negative literals are delayed, the two reductions $\mapsto_P$ and $\mapsto_N$ are sufficient to compute the well-founded model. The transformations $\mapsto_S$ and $\mapsto_F$ seem to be the corresponding operations for positive literals. However, they are not strong enough. Whereas the well-founded semantics depends on negative loops to determine undefined truth values, positive loops have to be detected and removed.

Example 4.3.13 (Positive Loop) Consider the following program $Loop$:

$$\begin{array}{lcl} p. & & \\ q & \leftarrow & \mathbf{not}\, p. \\ q & \leftarrow & r. \\ r & \leftarrow & q. \end{array}$$

We can apply *negative reduction* to delete $q \leftarrow \mathbf{not}\, p$, since $\mathbf{not}\, p$ is obviously false. But $\mapsto_{PSNF}$ does not allow to delete $q \leftarrow r$ and $r \leftarrow q$. □

Apparently our reductions are still too weak. As for the residual program, we want to obtain the well-founded model of a program directly from its normal form w.r.t. the rewriting system used. This is not possible in Example 4.3.13. In the rewriting system $\mapsto_R$ we could unfold one of the two last rules and delete the resulting tautology by the application of $\mapsto_T$. Since we do not want to allow full *unfolding*, we have to introduce a new transformation that detects this kind of tautology but does not imply the risk of an exponential blow-up. In other words: weakening unfolding forces us to strengthen deletion of tautologies. This stronger transformation is called *Loop Detection*.

Definition 4.3.14 (Loop Detection) Let P_1 and P_2 be ground programs. P_2 results from P_1 by *loop detection* ($P_1 \mapsto_L P_2$) iff there is a set $\mathcal{A}$ of ground atoms such that

1. for each rule $A \leftarrow \mathcal{B}$ in P_1, if $A \in \mathcal{A}$, then $\mathcal{B} \cap \mathcal{A} \neq \emptyset$,
2. $P_2 := \{A \leftarrow \mathcal{B} \in P_1 \mid A \notin \mathcal{A}\}$,
3. $P_1 \neq P_2$.

□

The *loop detection* can be viewed as an generalized form of the *deletion of tautologies* of Definition 4.2.1. There a rule could be deleted if *one atom* appears simultaneously in the head and the body of the same rule. This condition can be checked locally. With *loop detection* a set of rules can be deleted, if every atom from a *set of atoms* occurs simultaneously in the heads and bodies of the rules concerned. This condition cannot be checked locally but needs a detection algorithm that works more globally. This globality is quite expensive, thus our goal will be to reduce the number of applications of *loop detection* as far as possible.

At a first glance, it might seem that we have to "guess" an appropriate set $\mathcal{A}$. However, such a set $\mathcal{A}$ is nothing else than an unfounded set (cf. Definition 3.1.7). The greatest unfounded set consists of all positive ground atoms which are not possibly true, i.e., cannot be derived by assuming all negative literals to be true. According to Definition 3.2.1, the extended immediate consequence operator $T_{P_1,\emptyset}$ computes the possibly true atoms. Therefore, the greatest unfounded set is given by $BASE(P)\backslash \mathrm{lfp}(T_{P_1,\emptyset})$. This set has the property required for $\mathcal{A}$ in Definition 4.3.14:

Lemma 4.3.15 (Loop Detection) Let P_1 be a ground program and let

$$P_2 = \{(A \leftarrow \mathcal{B}) \in P_1 \mid A \in \mathrm{lfp}(T_{P_1,\emptyset})\}.$$

Then:

1. $P_1 \mapsto_L P_2$ (unless $P_2 = P_1$), and
2. P_2 is irreducible w.r.t. $\mapsto_L$.

Proof:

1. First we have to show that $\mathcal{A} = BASE(P) \backslash \mathrm{lfp}(T_{P_1,\emptyset})$ satisfies the first requirement in Definition 4.3.14, namely that for every rule $A \leftarrow \mathcal{B}$ in P_1, if $A \in \mathcal{A}$, then $\mathcal{B} \cap \mathcal{A} \neq \emptyset$. Suppose that this were not the case. Then P_1 would contain a rule $A \leftarrow \mathcal{B}$ with $pos(\mathcal{B}) \subset \mathrm{lfp}(T_{P_1,\emptyset})$ and $A \notin \mathrm{lfp}(T_{P_1,\emptyset})$. But this is impossible, since if $pos(\mathcal{B})$ is already derived by $T_{P_1,\emptyset}$, then A will be derived in the next step.

   So *loop detection* can indeed be applied with respect this set $\mathcal{A}$, and we get

   $$P_2 = \{(A \leftarrow \mathcal{B}) \in P_1 \mid A \in \mathrm{lfp}(T_{P_1,\emptyset})\}.$$

2. The second part is to show that P_2 is irreducible w.r.t. $\mapsto_L$. Suppose that *loop detection* were applicable to P_2 based on some set $\mathcal{A}'$ of ground atoms. Because of the second and third condition in Definition 4.3.14, an atom from $\mathcal{A}'$ must appear in at least one rule head of P_2. By the construction of P_2, this atom is contained in $\mathrm{lfp}(T_{P_1,\emptyset})$. But this is impossible: We show by induction on i that $\mathcal{A}' \cap (T_{P_1,\emptyset} \uparrow i) = \emptyset$. For $i = 0$ this is trivial. Let now $A \leftarrow \mathcal{B}$ be applied in the i-th iteration. By the induction hypothesis, we know that $pos(\mathcal{B}) \cap \mathcal{A}' = \emptyset$ and thus $\mathcal{B} \cap \mathcal{A}' = \emptyset$. Since $A \in (T_{P_1,\emptyset} \uparrow i)$, the rule is also contained in P_2. But then the first condition in Definition 4.3.14 implies that $A \notin \mathcal{A}'$.

□

As a consequence we see that also the operation of *loop detection* can be performed in polynomial time w.r.t. the size of the EDB: We compute $\mathrm{lfp}(T_{P_1,\emptyset})$ and delete all rules $A \leftarrow \mathcal{B}$ with $A \notin \mathrm{lfp}(T_{P_1,\emptyset})$. So it is neither necessary to construct all possible sets $\mathcal{A}$ to check whether $\mapsto_L$ is applicable, nor to explicitly construct the greatest unfounded set $\mathcal{A}$ as the complement of $\mathrm{lfp}(T_{P_1,\emptyset})$.

The following lemma shows that *loop detection* is a special case of *deletion of tautologies* and *unfolding*.

Lemma 4.3.16 If $P_1 \mapsto_L P_2$, then also $P_1 \mapsto^*_{TU} P_2$.

Proof: Let $\mathcal{A}$ be a set of ground atoms such that for all rules $A \leftarrow \mathcal{B}$ in P_1, if $A \in \mathcal{A}$, then $\mathcal{B} \cap \mathcal{A} \neq \emptyset$. Let $P_2 := \{A \leftarrow \mathcal{B} \in P_1 \mid A \notin \mathcal{A}\}$. We have to show that $P_1 \backslash P_2$, i.e. the rules with an atom from $\mathcal{A}$ in the head, can be deleted by unfolding and elimination of tautologies. We will show $P_1 \mapsto^*_{TU} P_2$ by induction on the number n of atoms from $\mathcal{A}$, which actually appear in rule bodies of $P_1 \backslash P_2$.

If $n = 0$, no atom of $\mathcal{A}$ appears in a body of a rule from $P_1 \backslash P_2$. But since each of the deleted rules must have such an atom in the body, this means $P_1 = P_2$, and thus $P_1 \mapsto^*_{TU} P_2$ trivially holds.

Let now $n > 0$ and $A_0 \in \mathcal{A}$ be an atom which appears in at least one body of a rule from $P_1 \backslash P_2$. Then we first eliminate all tautological rules containing A_0 in head and body. These rules are contained in $P_1 \backslash P_2$, so we are allowed to remove them. After that, we unfold all remaining occurrences of A_0 in bodies of rules which also contain an atom from $\mathcal{A}$ in their head. Let the resulting program be called P_1'. Then we have the following:

- After these operations, A_0 does not appear in any rule body with an atom from $\mathcal{A}$ in the head (Since we have eliminated the tautologies first, the rules about A_0 do not contain A_0 in their bodies.).
- Furthermore, no new atoms appear in the bodies of rules with a head from $\mathcal{A}$ (since $A_0 \in \mathcal{A}$), so the number n has decreased.
- The rules which were deleted by the unfolding steps contain an element of $\mathcal{A}$ in the head, so they are contained in $P_1 \backslash P_2$, and we are really allowed to delete them. The rules which were generated by the unfolding step also contain an atom from $\mathcal{A}$ in the head (since the head is not changed by unfolding). Thus, $P_2 = \{A \leftarrow \mathcal{B} \in P_1' \mid A \notin \mathcal{A}\}$.

The induction hypothesis then gives $P_1' \mapsto^*_{TU} P_2$, and thus together we have $P_1 \mapsto^*_{TU} P_1' \mapsto^*_{TU} P_2$. □

Corollary 4.3.17 Any semantics $\mathcal{S}$, which allows *unfolding* and *deletion of tautologies,* also allows *loop detection.*

Proof: This follows directly from lemma 4.3.16. □

Example 4.3.18 (Loop Detection) Consider again the program $Loop$ of Example 4.3.13:

$$
\begin{array}{lcl}
p. & & \\
q & \leftarrow & \mathbf{not}\, p. \\
q & \leftarrow & r. \\
r & \leftarrow & q.
\end{array}
$$

After deleting the second rule by *negative reduction*, the resulting program $Loop'$ is irreducible w.r.t. $\mapsto_{PSNF}$. Now *loop detection* has to be applied. First, the set $\{p\}$ of possible facts is computed by iterating the $T_{Loop',\emptyset}$ operator. We have $\mathcal{A} = \{p, q, r\} \backslash \{p\}$. According to Definition 4.3.14 the last two rules are deleted

because their heads q and r are contained in $\mathcal{A}$. Note that in operational terms it would have been sufficient to test whether q and r are elements of $\{p\} = \mathrm{lfp}(T_{Loop',\emptyset})$. It is not necessary to explicitly compute the complement. □

Example 4.3.19 (Repeated Loop Detection) Unfortunately, a single application of the (still) expensive *loop detection* is not always sufficient for computing the well-founded model:

$$
\begin{array}{lcl}
p(1). & & \\
q(1) & \leftarrow & \mathbf{not}\, p(1). \\
q(1) & \leftarrow & r(1). \\
r(1) & \leftarrow & q(1). \\
p(2) & \leftarrow & \mathbf{not}\, q(1). \\
q(2) & \leftarrow & \mathbf{not}\, p(2). \\
q(2) & \leftarrow & r(2). \\
r(2) & \leftarrow & q(2).
\end{array}
$$

First, *negative reduction* eliminates the second rule because $\mathbf{not}\, p(1)$ is obviously false. In the next step, we need to apply *loop detection* in order to get rid of the rules $q(1) \leftarrow r(1)$ and $r(1) \leftarrow q(1)$. After that we can evaluate $\mathbf{not}\, q(1)$ to true and apply *positive reduction* and then $\mathbf{not}\, p_2$ to false and apply *negative reduction*. Now *loop detection* is again necessary to delete the rules $q(2) \leftarrow r(2)$ and $r(2) \leftarrow q(2)$. □

Example 4.3.19 shows that we must iterate all five transformations. Obviously, *loop detection* is the most expensive transformation. In Chapter 6 and 7 we will determine classes of programs for which no or only a limited number of applications of *loop detection* is necessary.

Definition 4.3.20 (X-Transformation) Let $\mapsto_X$ denote our final rewriting system:

$$\mapsto_X := \mapsto_P \cup \mapsto_N \cup \mapsto_S \cup \mapsto_F \cup \mapsto_L.$$

□

The well-founded semantics allows the $\mapsto_X$ transformation.

Corollary 4.3.21 (WFS allows $\mapsto_X$) Let P be any normal program. Let P' be any program such that $P \mapsto_X^* $ holds. Then $WFS(P) = WFS(P')$.

Proof: This follows from Proposition 4.2.12, Lemma 4.3.8, Lemma 4.3.16, and Definition 4.3.20. □

The rewriting system $\mapsto_X$ can be seen as the extension of Fittings least fixpoint operator Φ_P by *loop detection* (see Theorem 4.3.12).

A normal form w.r.t. the rewriting system $\mapsto_X$ is called a program remainder:

Definition 4.3.22 (Program Remainder) Let P be a program. Let the program $\widehat{P}$ satisfy

1. $ground(P) \mapsto_X^* \widehat{P}$,
2. $\widehat{P}$ is irreducible w.r.t. $\mapsto_X$, i.e. there is no P' with $\widehat{P} \mapsto_X P'$.

Then $\widehat{P}$ is called a *program remainder* of P. □

Lemma 4.3.23 Every program P has a program remainder, i.e. the rewriting system $\mapsto_X$ is terminating.

Proof: All our transformations strictly reduce the total number of occurring literals. So if $P_0 := ground(P)$ and P_i is any program with $P_{i-1} \mapsto_X P_i$, $i \geq 1$, then we must reach a program P_n in which no further transformation can be applied. This is a program remainder of P. □

A program remainder is equivalent to the original program under the well-founded and stable model semantics [GL88], and, in fact, under a large class of semantics which allow the elementary transformations introduced in Section 4.2:

Theorem 4.3.24 (Equivalence of Program Remainder) Let $\mathcal{S}$ be any semantics which allows *unfolding*, *deletion of tautologies*, *positive* and *negative reduction*, and the *deletion of non-minimal rules*. Then

$$\mathcal{S}(\widehat{P}) = \mathcal{S}(P)$$

for every program P with remainder $\widehat{P}$.

Proof: This follows immediately from the Lemmata 4.3.8 and 4.3.16. □

The program remainder behaves like the residual program of Definition 4.2.9. A literal A is true in the well-founded model iff A is contained as a fact in the program remainder, $A \in facts(\widehat{P})$. Conversely, the well-founded model satisfies $\mathbf{not}\, A$ iff A does not appear in any rule head of the program remainder, i.e., $A \notin heads(\widehat{P})$. In other words, in the well-founded model exactly the literals of $known_{BASE(P)}(\widehat{P})$ (cf. Definition 4.1.4) have a definite truth value. All other literals are undefined in the well-founded model.

Theorem 4.3.25 (Computation of Well-Founded Semantics) For every program P and remainder $\widehat{P}$, the well-founded model W_P^* of P satisfies exactly those positive and negative literals which are immediately obvious in $\widehat{P}$, i.e.

$$W_P^* = known_{BASE(P)}(\widehat{P}).$$

Proof: By Theorem 4.3.24 and since the well-founded semantics allows the transformations of the rewriting system $\mapsto_X$ by Corollary 4.3.21, P and $\widehat{P}$ have the same well-founded model, i.e., $W_P^* = W_{\widehat{P}}^*$. Thus it suffices to show that $W_{\widehat{P}}^* = known_{BASE(P)}(\widehat{P})$. We do this by executing the alternating fixpoint procedure on $\widehat{P}$.

1. $K_0 = facts(\widehat{P})$: It is clear that at least these facts are contained in K_0, i.e. $facts(\widehat{P}) \subseteq K_0$. To show equality assume that $K_0 \backslash facts(\widehat{P}) \neq \emptyset$. Let $A \leftarrow L_1 \wedge \cdots \wedge L_n$ be the first rule with non-empty body applied by the alternating fixpoint procedure. By Definition 3.2.2 the L_i are all positive and contained in $facts(\widehat{P})$. But then the *success* transformation is applicable to $\widehat{P}$. This contradicts the irreducibility of $\widehat{P}$.

2. $U_0 = heads(\widehat{P})$: The inclusion $U_0 \subseteq heads(\widehat{P})$ is trivial. Now assume that $U_0 \subset heads(\widehat{P})$ holds. Let $\mathcal{A} := BASE(P) \backslash U_0$ and let $A \leftarrow \mathcal{B}$ be an arbitrary rule of $\widehat{P}$ with $A \in \mathcal{A}$ (such a rule exists because $U_0 \neq heads(\widehat{P})$). If $\mathcal{B} \cap \mathcal{A} \neq \emptyset$, *loop detection* were applicable, contradicting the irreducibility of $\widehat{P}$. Thus we have $\mathcal{B} \cap \mathcal{A} = \emptyset$, and therefore $pos(\mathcal{B}) \subseteq U_0$. But then, since $A \in \mathcal{A}$, i.e., $A \notin U_0$, there must be a negative literal $\mathbf{not}\, B \in \mathcal{B}$ with $B \in K_0$. In this case *negative reduction* is be applicable to $\widehat{P}$ which is impossible because $\widehat{P}$ is irreducible.

3. $K_1 = facts(\widehat{P})$: The inclusion $facts(\widehat{P}) \subseteq K_1$ is trivial, because the fixpoint $K_1 := \mathrm{lfp}(T_{\widehat{P},U_0})$ (cf. Definition 3.2.2) of course contains all facts of $\widehat{P}$. To show equality assume that $K_1 \backslash facts(\widehat{P}) \neq \emptyset$. Let $A \leftarrow L_1 \wedge \cdots \wedge L_n \in \widehat{P}$ be the first rule with non-empty body applied during the computation of K_1. If it contains a positive body literal L_i, $L_i \in facts(\widehat{P})$ must hold. But then, *success* is applicable, contradicting the irreducibility of $\widehat{P}$. If it contains a negative body literal $L_1 = \mathbf{not}\, B$, $B \notin U_0 = heads(\widehat{P})$ must hold. But then, *positive reduction* is applicable, contradicting again the irreducibility of $\widehat{P}$.

4. Since $K_1 = K_0$, we also have $U_1 = U_0$, so the fixpoint is reached.

□

From this it is also simple to derive an explicit characterization of the program remainder: From the Herbrand instantiation of the program we obtain the program remainder by

1. deleting every rule instance with a body literal which is false in the well-founded model, and

2. removing from the remaining rule instances the body literals which are true in the well-founded model.

Theorem 4.3.26 (Program Remainder) Let $\widehat{P}$ be a remainder of a program P. The following holds:

$$\widehat{P} = \{A \leftarrow (\mathcal{B} \backslash W_P^*) \mid A \leftarrow \mathcal{B} \in ground(P) \text{ and for every } B \in \mathcal{B}\text{: } \sim B \notin W_P^*\}.$$

Thus the remainder of a program is uniquely determined.

Proof: Let us temporarily call the transformation on the right hand side $wred$, i.e. for every program P' let

$$wred(P') := \{A \leftarrow (\mathcal{B} \backslash W_{P'}^*) \mid A \leftarrow \mathcal{B} \in ground(P') \text{ and for every } B \in \mathcal{B}\text{: } \sim B \notin W_{P'}^*\}.$$

1. We first show that

$$P_1 \mapsto_X P_2 \implies wred(P_1) = wred(P_2).$$

   (a) $P_1 \mapsto_P P_2$: *Positive reduction* removes a negative body literal $\mathbf{not}\ B$ from some rule $A \leftarrow \mathcal{B}$, if $B \notin heads(P_1)$. *Positive reduction* does not change the well-founded model. Furthermore $\mathbf{not}\ B$ is certainly true in the well-founded model. Thus $\mathcal{B} \backslash W_{P_1}^* = (\mathcal{B} \backslash \{\mathbf{not}\ B\}) \backslash W_{P_2}^*$.

   (b) $P_1 \mapsto_N P_2$: *Negative reduction* deletes a rule $A \leftarrow \mathcal{B}$ where $\mathcal{B}$ contains a negative literal $\mathbf{not}\ B$ such that $B \in facts(P_1)$. But then B is true in the well-founded model of P_1 and P_2, so the $wred$-transformation will delete this rule anyway.

   (c) $P_1 \mapsto_S P_2$: *Success* removes a positive body literal B from some rule $A \leftarrow \mathcal{B}$, if $B \in facts(P_1)$. *Success* does not change the well-founded model, and furthermore B is certainly true in this well-founded model. Thus we can conclude $\mathcal{B} \backslash W_{P_1}^* = (\mathcal{B} \backslash \{B\}) \backslash W_{P_2}^*$.

   (d) $P_1 \mapsto_F P_2$: *Failure* deletes a rule $A \leftarrow \mathcal{B}$ where $\mathcal{B}$ contains a positive literal B such that $B \notin heads(P_1)$. But then $\mathbf{not}\ B$ is true in the well-founded model of P_1 and P_2, so the $wred$-transformation will delete this rule anyway.

(e) $P_1 \mapsto_L P_2$: *Loop detection* w.r.t. to an unfounded set $\mathcal{A}$ deletes all rules $A \leftarrow \mathcal{B}$ where $A \in \mathcal{A}$. By the first condition of Definition 4.3.14, this also means that $\mathcal{B}$ contains a positive body literal B with $B \in \mathcal{A}$. But since all atoms in the unfounded set $\mathcal{A}$ are false in the well-founded model of P_1 and P_2, the $wred$-transformation will delete these rules anyway.

2. Since $ground(P) \mapsto_X^* \widehat{P}$, we have $wred(ground(P)) = wred(\widehat{P})$. Because the well-founded semantics is instantiation-invariant (cf. Definition 3.1.14) the definition of $wred$ implies $wred(P) = wred(\widehat{P})$. We now only have to show $wred(\widehat{P}) = \widehat{P}$. Assume that $wred(\widehat{P}) \neq \widehat{P}$. Then there is a rule $A \leftarrow \mathcal{B} \in \widehat{P}$ such that there is $B \in \mathcal{B}$ and either $A \leftarrow \mathcal{B}$ has been modified by $wred$ because $B \in W^*_{\widehat{P}}$ or entirely deleted because $\sim B \in W^*_{\widehat{P}}$. But we know by Theorem 4.3.25 that

$$W^*_{\widehat{P}} = known_{BASE(P)}(\widehat{P}).$$

We can distinguish the following four cases:

(a) $B \in W^*_{\widehat{P}}$.

i. B is positive. It follows that $B \in facts(\widehat{P})$. But then *success* is applicable, contradicting the irreducibility of $\widehat{P}$.

ii. $B = \mathbf{not}\, C$ is negative. It follows that $C \notin heads(\widehat{P})$. But then *positive reduction* is applicable, again contradicting the irreducibility of $\widehat{P}$.

(b) $\sim B \in W^*_{\widehat{P}}$.

i. B is positive. It follows that $\mathbf{not}\, B \in W^*_{\widehat{P}}$ and thus $B \notin heads(\widehat{P})$. But then *failure* is applicable, contradicting the irreducibility of $\widehat{P}$.

ii. $B = \mathbf{not}\, C$ is negative. It follows that $C \in W^*_{\widehat{P}}$ and thus $C \in facts(\widehat{P})$. But then *negative reduction* is applicable, again contradicting the irreducibility of $\widehat{P}$.

□

Corollary 4.3.27 (Confluence of $\mapsto_X$) The rewriting system $\mapsto_X$ is confluent, strongly terminating and the program remainder $\widehat{P}$ is the unique normal form of $ground(P)$ w.r.t. $\mapsto_X$ in the worst case. □

Remember that, as a consequence of unlimited unfolding, the residual program can grow to a size that is exponential in the size of the original program. In our approach the program does not grow at all since all transformations strictly reduce the size of the program.

Lemma 4.3.28 (Quadratic Time Complexity) Let P be a ground program. Let n be the size of P, i.e., the number of occurrences of literals in P. Then P can be transformed into its remainder $\widehat{P}$ in time $O(n^2)$.

Proof: In Section 8.2 we describe an algorithm that computes the normal form of P w.r.t. $\mapsto_{PSNF}$ in time $O(n)$. Also one application of *loop detection* can be performed in time $O(n)$. Since by each application of *loop detection* the size of the program is strictly reduced, *loop detection* can be applied at most n times. Altogether, the remainder $\widehat{P}$ of P is reached in time $O(n^2)$. □

In the following example the transformation reaches the upper bound and needs quadratic time.

Example 4.3.29 (Quadratic Time) Consider the following program that extends the program of Example 4.3.19.

$$
\begin{array}{lcl}
p(1). & & \\
q(1) & \leftarrow & \mathbf{not}\, p(1). \\
q(1) & \leftarrow & r(1). \\
r(1) & \leftarrow & q(1). \\
p(2) & \leftarrow & \mathbf{not}\, q(1). \\
q(2) & \leftarrow & \mathbf{not}\, p(2). \\
q(2) & \leftarrow & r(2). \\
r(2) & \leftarrow & q(2). \\
 & \vdots & \\
p(n) & \leftarrow & \mathbf{not}\, q(n-1). \\
q(n) & \leftarrow & \mathbf{not}\, p(n). \\
q(n) & \leftarrow & r(n). \\
r(n) & \leftarrow & q(n).
\end{array}
$$

This program contains n positive loops for which n applications of *loop detections* are necessary. With *negative reduction* we can delete the second rule. Then we apply *loop detection* the first time and delete the third and fourth rule. Then we reduce the fifth rule with *positive reduction* and delete the sixth rule with *negative reduction*. Then the second application of *loop detection* is needed. This sequence of transformations is repeated until *loop detection* has been applied n times.

According to Lemma 4.3.15 we apply *loop detection* by computing the set $poss$ of possible atoms and deleting all rules with a head A not in $poss$. The set $poss$ contains $O(n)$ elements that are in a simple implementation recomputed for each application of *loop detection*. Thus, the overall running time is quadratic in n for this example. □

Note that quadratic time is needed in the worst case if *loop detection* is to be applied n times. Time complexity can be improved up to linear time, if the number of applications of *loop detection* can be reduced, depending on the dependencies in P. For instance, consider Example 4.3.9.

Chapter 5

Grounding

The program transformations defined in Chapter 4 are applicable only to ground programs. However, most programs occurring in practice are non-ground.

5.1 Full Instantiation

According to the "instantiation invariance" required by Definition 3.1.14, we could simply apply the transformations to the Herbrand instantiation $ground(P)$ of a given non-ground program P (cf. Definition 2.4.5). However, the following example shows that the construction of the Herbrand instantiation can be very expensive.

Example 5.1.1 (Non-Ground Even Numbers) Consider the following logic program $EvenNum2$ which defines again the even numbers between 0 and some arbitrary but fixed number n:

$$
\begin{array}{l}
even(0). \\
even(X) \leftarrow succ(Y, X) \wedge \mathbf{not}\, even(Y). \\
succ(0, 1). \\
\quad \vdots \\
succ(n-1, n).
\end{array}
$$

The alternating fixpoint procedure produces the same sequence of sets K_i and U_i as in Example 3.2.6, except that here these sets contain also the given facts about $succ$. Again, a quadratic number of derivations is needed to compute the well-founded model. □

Example 5.1.2 (Full Instantiation) Consider the program $EvenNum2$ from Example 5.1.1. The program contains n constants. The second rule contains two different variables. Thus in the ground instantiation $ground(EvenNum2)$ there are at least n^2 rule instances. □

Example 5.1.2 illustrates that in general it is too expensive to compute the ground instantiation of a non-ground program in order to apply the ground transformations to it. However, the full ground instantiation of the program $EvenNum2$ contains rule instances that are obviously useless. In this example only those ground instances of the second rule are relevant for that there are corresponding $succ$ facts. Based on this observation we propose a method to derive only relevant rule instances in the next section.

5.2 Bottom-Up Grounding

Given a non-ground program P, our aim is now to find a ground program P' that is equivalent to P (and to its Herbrand instantiation $ground(P)$), i.e.,

$$\mathrm{WFS}(P') = \mathrm{WFS}(P) = \mathrm{WFS}(ground(P)).$$

The idea is to use a version of the T_P operator which keeps the bodies of the applied rule instances and assumes that all negative body literals are possibly true. We can start with rule instances having no positive body literals. The heads of these rule instances are possibly true, thus further rule instances having these atoms as positive body literals can be derived successively.

Definition 5.2.1 (Immediate Consequences with Delayed Literals) Let P be a program. Given a set S of conditional facts the operator $\bar{T}_P$ computes the following set of conditional facts:

$$\bar{T}_P(S) \;:=\; \{A \leftarrow \mathcal{B} \;\in\; ground(P) \mid pos(\mathcal{B}) \subseteq heads(S)\}.$$

□

Remark 5.2.2 (Range-Restriction) Although not required by Definition 5.2.1, it is important for an efficient implementation, that the program P is range-restricted. In this case, the condition $pos(\mathcal{B}) \subseteq heads(S)$ binds all the variables and we never have to resort to a full Herbrand instantiation of any variable. For instance, in the above example, we would never consider the rule instance

$$even(2) \leftarrow succ(0,2) \wedge \mathbf{not}\, even(0).$$

We use the existing $succ$-facts to instantiate the rule. Since there is no fact $succ(0,2)$ the rule instance above is not generated. □

Proposition 5.2.3 Let P be a normal program. Then the following properties of the least fixpoint of the $\bar{T}_P$ operator are obvious.

1. $\mathrm{lfp}(\bar{T}_P) \subseteq ground(P)$.
2. $heads(\mathrm{lfp}(\bar{T}_P)) = \mathrm{lfp}(T_{P,\emptyset})$. (The operators are very similar, only $T_{P,\emptyset}$ "forgets" the bodies of the applied rule instances.)
3. If P is range-restricted, then $\mathrm{lfp}(\bar{T}_P)$ contains only constants appearing in P. (This can be easily proven by an induction on the number of derivation steps.)
4. $\mathrm{lfp}(\bar{T}_P)$ can be computed in polynomial time with respect to the size of the EDB.

□

The following lemma embeds this kind of "intelligent grounding" into our transformation-based approach:

Lemma 5.2.4 Let P be a normal program (not necessarily ground), $\bar{T}_P$ be defined as above, and let $\mapsto_{LF} := \mapsto_L \cup \mapsto_F$, i.e. the combination of *loop detection* and *failure*. Then we have

$$ground(P) \;\mapsto^*_{LF}\; \mathrm{lfp}(\bar{T}_P).$$

By confluence of $\mapsto_X$, this immediately implies $\mathrm{lfp}(\bar{T}_P) \mapsto^*_X \widehat{P}$, so we can start the computation of the program remainder at $\mathrm{lfp}(\bar{T}_P)$.

Proof: Following Lemma 4.3.15, $\mathcal{A} := BASE(P)\backslash\mathrm{lfp}(T_{P,\emptyset})$ is a valid reference set for *loop detection*. Let P' be the result of applying *loop detection* w.r.t. this set $\mathcal{A}$ to $ground(P)$. From Definition 4.3.14 it follows that

$$P' = \{A \leftarrow \mathcal{B} \in ground(P) \mid A \in \mathrm{lfp}(T_{P,\emptyset})\}.$$

Thus $heads(P') \subseteq \mathrm{lfp}(T_{P,\emptyset})$, and we can use *failure* to remove also the rules which contain a positive body literal B with $B \notin \mathrm{lfp}(T_{P,\emptyset})$. Then the result is

$$P'' = \{A \leftarrow \mathcal{B} \in ground(P) \mid \{A\} \cup pos(\mathcal{B}) \subseteq \mathrm{lfp}(T_{P,\emptyset})\}.$$

As mentioned above, it is obvious that $\mathrm{lfp}(T_{P,\emptyset}) = heads(\mathrm{lfp}(\bar{T}_P))$. To prove $P'' = \mathrm{lfp}(\bar{T}_P)$, we show that $P'' \subseteq \mathrm{lfp}(\bar{T}_P)$ and $P'' \supseteq \mathrm{lfp}(\bar{T}_P)$:

1. Consider any $A \leftarrow \mathcal{B} \in P''$. From the definition of P'' it follows that

   $$pos(\mathcal{B}) \subseteq \mathrm{lfp}(T_{P,\emptyset}) = heads(\mathrm{lfp}(\bar{T}_P)),$$

   thus $A \leftarrow \mathcal{B}$ must also be contained in $\mathrm{lfp}(\bar{T}_P)$.

2. If $A \leftarrow \mathcal{B} \in ground(P)$ is derived by $\bar{T}_P$, this means that

$$pos(\mathcal{B}) \subseteq heads(\text{lfp}(\bar{T}_P)) = \text{lfp}(T_{P,\emptyset})$$

and $A \in heads(\text{lfp}(\bar{T}_P)) = \text{lfp}(T_{P,\emptyset})$.

□

Example 5.2.5 Consider again the program $EvenNum2$ of Example 5.1.1. Following Lemma 5.2.4, we can start the transformation process at $\text{lfp}(\bar{T}_{EvenNum2})$ instead of the complete ground instantiation:

$$\begin{aligned}\text{lfp}(\bar{T}_{EvenNum2}) = \{\quad & succ(0,1),\\ & \ldots,\\ & succ(n-1,n),\\ & even(0),\\ & even(1) \leftarrow succ(0,1) \wedge \mathbf{not}\, even(0),\\ & \ldots,\\ & even(n) \leftarrow succ(n-1,n) \wedge \mathbf{not}\, even(n-1)\}.\end{aligned}$$

This start set can be constructed using $O(n)$ derivation steps. The following $O(n)$ transformation steps transform the start set into the program remainder of $EvenNum2$. First we apply $\mapsto_S$ (*success*) to evaluate the positive body literals and get:

$$\begin{aligned}EvenNum3 := \{\quad & succ(0,1),\\ & \ldots,\\ & succ(n-1,n),\\ & even(0),\\ & even(1) \leftarrow \mathbf{not}\, even(0),\\ & \ldots,\\ & even(n) \leftarrow \mathbf{not}\, even(n-1)\}.\end{aligned}$$

Except for the additional $succ$-facts, this is exactly our program $EvenNum$ of Example 3.2.6. So, with alternating applications of $\mapsto_N$ and $\mapsto_P$, we get the well-founded model:

$$\{succ(0,1),\ \ldots,\ succ(n-1,n),\ even(0),\ even(2),\ \ldots,\ even(n)\}.$$

In summary, $O(n)$ derivation/transformation steps have been performed. □

5.3 SCC-Oriented Grounding

In any practical implementation, one would not apply the transformations to the entire set of rules but partition the set of program rules according to the *strongly connected components (SCCs)* of its static dependency graph. This induces an order for applying our transformations which allows one to delay fewer body literals, because often their truth value is already known. Also the intelligent grounding explained in Section 5.2 becomes even more selective when the information about lower SCCs is already available. As we will see, this has also advantages for the expensive *loop detection*: Often the *loop detection* which is performed implicitly by the intelligent grounding is already sufficient, when the grounding is done component-wise. For many programs, no explicit *loop detection* is needed.

As shown in [Tar72], the SCCs can be determined in linear time (see also [AHU87]). Thus, the additional effort needed for the partitioning according to the SCC structure cannot degrade the overall complexity. First, we define the dependence relation for ground programs.

Definition 5.3.1 (Dependence) Let P_1 and P_2 be two ground programs. We say that P_1 depends on P_2, $P_1 \leftarrow P_2$, if at least one head atom of P_2 occurs anywhere in P_1, i.e.,

$$At(P_1) \cap heads(P_2) \neq \emptyset.$$

□

The following lemma gives the reason why an SCC-wise application of our transformations is possible:

Lemma 5.3.2 Let P_1, P_2, and P be ground programs such that P_1 and P_2 do not depend on P. Then

$$P_1 \mapsto_X P_2 \iff P_1 \cup P \mapsto_X P_2 \cup P.$$

Proof: Let $A \leftarrow \mathcal{B}$ be the rule affected (i.e. deleted or modified) by one of the transformations *positive* and *negative reduction*, *success* and *failure* ($\mapsto_{PSNF}$). Then the applicability of this transformation depends only on rules having a ground atom from $pos(\mathcal{B}) \cup neg(\mathcal{B})$ in the head. Therefore, the rules in P have no influence on the applicability of the transformation: If $P_1 \mapsto_{PSNF} P_2$, then $P_1 \cup P \mapsto_{PSNF} P_2 \cup P$, and if $P_1 \not\mapsto_{PSNF} P_2$, then $P_1 \cup P \not\mapsto_{PSNF} P_2 \cup P$.

The same is true for *loop detection*: Let $P_1 \mapsto_L P_2$ and let $\mathcal{A}$ be the set of ground atoms used. Then *loop detection* gives the same result if we use the atom set $\mathcal{A}' := \{A \in \mathcal{A} \mid A \text{ occurs in } P_1\}$ instead of $\mathcal{A}$. But with this unfounded set it is obvious that

the addition of the rules P has no influence on the applicability of *loop detection* and on the set of deleted rules.

Conversely, suppose that $P_1 \cup P \mapsto_L P_2 \cup P$ with some unfounded set $\mathcal{A}$ as a reference set. Apparently, no rules of P are deleted. Thus $\mapsto_L$ is applicable to P_1 with the same reference set $\mathcal{A}$, and therefore we have $P_1 \mapsto_L P_2$. □

This means that, when applying our transformations to a program fragment P, we can ignore higher SCCs, i.e., fragments P does not depend on. If a transformation is applicable without considering the rules of higher SCCs, i.e., if $P_1 \mapsto_X P_2$ holds, this is also true after adding these rules, i.e., $P_1 \cup P \mapsto_X P_2 \cup P$ follows.

On the other hand, no new transformation becomes applicable by taking rules of higher SCCs into account. Thus, once P_1 is irreducible (i.e. there is no P_2 with $P_1 \mapsto_X P_2$) we do not need to care about P_1 anymore, in particular, we do not need to try to apply one of our transformations to a rule of P_1 when proceeding to $P_1 \cup P$. Lemma 5.3.2 ensures only that in this case any transformation must necessarily change one of the new rules in P. But in fact, the irreducible P_1 remains unchanged:

Lemma 5.3.3 Let P_1 and P be ground programs, such that P_1 is irreducible w.r.t. $\mapsto_X$ and P_1 does not depend on P. If $P_1 \cup P \mapsto_X P'$ for some ground program P', then $P_1 \subseteq P'$.

Proof: For *positive* and *negative reduction*, *success* and *failure* this is a corollary of Lemma 5.3.2, because these transformations change only a single rule and Lemma 5.3.2 implies that P must be changed. Now let $P_1 \cup P \mapsto_L P'$, let $\mathcal{A}$ be the set of ground atoms as in Definition 4.3.14, and let $\mathcal{A}' := \mathcal{A} \cap heads(P_1)$. If $\mathcal{A}' = \emptyset$ holds, $P_1 \subseteq P'$ follows immediately. Otherwise, $\mathcal{A}'$ demonstrates that P_1 is not irreducible: *Loop detection* can be applied to P_1 based on $\mathcal{A}'$.

□

With these lemmata, it is clear that our transformations can be applied locally to the single program components.

Definition 5.3.4 (Evaluation Sequence) Let P be a program (not necessarily ground). A sequence $P^{(1)}, \ldots, P^{(m)}$ of subsets of P is called an evaluation sequence for P iff

1. $P = P^{(1)} \cup \cdots \cup P^{(m)}$,
2. $P^{(i)} \cap P^{(j)} = \emptyset$ for $i \neq j$,
3. all predicates occurring in $P^{(i)}$, $1 \leq i \leq m$, do not occur in the heads of $P^{(j)}$ for $j > i$.

□

Of course, the single components $P^{(i)}$ are usually chosen as small as possible, i.e. either all rules about a single non-recursive predicate or all rules about a set of mutually recursive predicates. This is, however, not necessary for our theorems.

Now we can compute the program remainder component-wise in the order given by any evaluation sequence:

Corollary 5.3.5 Let $P^{(1)}, \ldots, P^{(m)}$ be an evaluation sequence for P, and let the ground programs $\mathcal{R}_0, \ldots, \mathcal{R}_m$ be defined as follows:

- $\mathcal{R}_0 := \emptyset$,
- $\mathcal{R}_i$ is the program remainder of $\mathcal{R}_{i-1} \cup ground(P^{(i)})$, for $0 < i \leq m$.

Then $\mathcal{R}_m$ is the remainder of P. Furthermore, $\mathcal{R}_{i-1} \subseteq \mathcal{R}_i$ for $0 < i \leq m$.

Proof: The set inclusion follows directly from Lemma 5.3.2 and Lemma 5.3.3. R_m is the remainder of P due to the confluence of $\mapsto_X$. □

We can now combine the component-wise approach described above with the intelligent grounding introduced in Section 5.2. Since the remainder $\mathcal{R}_{i-1}$ of the lower program components is already known when the program component $P^{(i)}$ is processed, the set of ground instances to be computed is smaller in general . In particular, we do not need to generate a rule instance if a negative body literal is known to be false because the corresponding positive atom has already been derived as a fact. Already with this simple optimization, the implicit *loop detection* performed by the grounding is sufficient in many cases.

Definition 5.3.6 Let P be a program and $\mathcal{R}$ be a ground program. Given a set S of conditional facts, the operator $\hat{T}_{P,R}$ computes the following set of conditional facts:

$$\hat{T}_{P,R}(S) := \{A \leftarrow \mathcal{B} \in ground(P) \mid \begin{array}{l} pos(\mathcal{B}) \subseteq heads(R) \cup heads(S) \\ \text{and } neg(\mathcal{B}) \cap facts(R) = \emptyset\}. \end{array}$$

If R is empty, we obtain the operator $\bar{T}_P$ of Definition 5.2.1 by $\bar{T}_P(S) = \hat{T}_{P,\emptyset}(S)$. □

Of course, in a practical implementation one would not only use the negative knowledge to create as few ground instances as possible, but also positive knowledge to remove body literals which are already known to be true. For instance, we could apply *success* and *positive reduction* already during the grounding phase. However, in this work we choose to separate issues for the sake of clarity and readability.

The following lemma shows that computing the least fixpoint of this operator is only a more efficient implementation of certain transformations (this generalizes Lemma 5.2.4):

Lemma 5.3.7 Let P be a program, and let $\mathcal{R}$ be a ground program. Then

$$\mathcal{R} \cup ground(P) \mapsto_X^* \mathcal{R} \cup \mathrm{lfp}(\hat{T}_{P,R}).$$

Furthermore, if $\mathcal{R}$ is irreducible w.r.t. $\mapsto_L$, then $\mathcal{R} \cup \mathrm{lfp}(\hat{T}_{P,R})$ is irreducible w.r.t. $\mapsto_L$.

Proof: First we apply *negative reduction* to $\mathcal{R} \cup ground(P)$ and obtain the rule set

$$\mathcal{R} \cup \{A \leftarrow \mathcal{B} \in ground(P) \mid neg(\mathcal{B}) \cap facts(\mathcal{R}) = \emptyset\}.$$

Now we apply *loop detection* w.r.t. the set

$$\mathcal{A} := BASE(P) \backslash heads(\mathcal{R} \cup \mathrm{lfp}(\hat{T}_{P,R})).$$

Of course, if it should happen that no elements of this set appear as rule heads, *loop detection* is not needed. Otherwise we have to show that the condition of Definition 4.3.14 is satisfied. Let us assume that there is a rule $A \leftarrow \mathcal{B}$ such that the head A is contained in $\mathcal{A}$, but $\mathcal{B} \cap \mathcal{A} = \emptyset$. Since $heads(\mathcal{R}) \cap \mathcal{A} = \emptyset$ by definition of $\mathcal{A}$, it is clear that such a rule must be an element of $ground(P)$. Furthermore, the construction of $\mathcal{A}$ entails $A \notin heads(\mathrm{lfp}(\hat{T}_{P,R}))$ and $pos(\mathcal{B}) \subseteq heads(\mathcal{R} \cup \mathrm{lfp}(\hat{T}_{P,R}))$. But this is a contradiction since in this case the operator $\hat{T}_{P,R}$ would have derived the rule $A \leftarrow \mathcal{B}$:

$$\begin{array}{rcll} A \leftarrow \mathcal{B} & \in & \{A \leftarrow \mathcal{B} \in ground(P) \mid & pos(\mathcal{B}) \subseteq heads(R) \cup heads(\mathrm{lfp}(\hat{T}_{P,R})) \\ & & & \text{and } neg(\mathcal{B}) \cap facts(R) = \emptyset\} \\ & = & \hat{T}_{P,R}(\mathrm{lfp}(\hat{T}_{P,R})) & \\ & = & \mathrm{lfp}(\hat{T}_{P,R}) & \end{array}$$

Consequently, we would have a rule $A \leftarrow \mathcal{B}$ which is not contained in $\mathrm{lfp}(\hat{T}_{P,R})$ (since $A \notin heads(\mathrm{lfp}(\hat{T}_{P,R}))$), although $pos(\mathcal{B}) \subseteq heads(\mathcal{R} \cup \mathrm{lfp}(\hat{T}_{P,R}))$ and $neg(\mathcal{B}) \cap facts(\mathcal{R}) = \emptyset$ (since otherwise the rule would have been deleted by *negative reduction*). Thus, by contradiction, $\mathcal{A}$ qualifies as a reference set for *loop detection*. Now we can apply *loop detection* w.r.t. $\mathcal{A}$ and get

$$\mathcal{R} \cup \{A \leftarrow \mathcal{B} \in ground(P) \mid A \in heads(R) \cup heads(\mathrm{lfp}(\hat{T}_{P,R})) \text{ and } neg(\mathcal{B}) \cap facts(R) = \emptyset\}.$$

Note, that $\mathcal{R}$ is unchanged because $\mathcal{A} \cap heads(\mathcal{R}) = \emptyset$ by construction. Applying *failure* we obtain

$$\mathcal{R} \cup \{A \leftarrow \mathcal{B} \in ground(P) \mid A \in heads(R) \cup heads(\mathrm{lfp}(\hat{T}_{P,R})) \text{ and } pos(\mathcal{B}) \subseteq heads(R) \cup heads(\mathrm{lfp}(\hat{T}_{P,R})) \text{ and } neg(\mathcal{B}) \cap facts(R) = \emptyset\}.$$

But as we have just seen, if both $pos(\mathcal{B}) \subseteq heads(R) \cup heads(\text{lfp}(\hat{T}_{P,R}))$ and $neg(\mathcal{B}) \cap facts(R) = \emptyset$ for a rule $A \leftarrow \mathcal{B}$ in $ground(P)$, then $A \in heads(\text{lfp}(\hat{T}_{P,R}))$. Thus, the formula can be simplified and we end up with the set

$$\begin{aligned} & \mathcal{R} \cup \{A \leftarrow \mathcal{B} \in ground(P) \mid \quad pos(\mathcal{B}) \subseteq heads(R) \cup heads(\text{lfp}(\hat{T}_{P,R})) \\ & \qquad\qquad\qquad\qquad\qquad\qquad \text{and } neg(\mathcal{B}) \cap facts(R) = \emptyset\} \\ = \quad & \mathcal{R} \cup \hat{T}_{P,R}(\text{lfp}(\hat{T}_{P,R})) \\ = \quad & \mathcal{R} \cup \text{lfp}(\hat{T}_{P,R}). \end{aligned}$$

This completes the proof of the first part of the lemma, i.e., we have shown

$$\mathcal{R} \cup ground(P) \mapsto_X^* \mathcal{R} \cup \text{lfp}(\hat{T}_{P,R}).$$

Now let us prove the second part of the lemma stating that the resulting set $\mathcal{R} \cup \text{lfp}(\hat{T}_{P,R})$ is irreducible w.r.t. *loop detection*. Let us assume the contrary. Then there is a set $\mathcal{A}$ which satisfies the conditions of Definition 4.3.14. In particular, this means that there is a rule $A \leftarrow \mathcal{B}$ with $A \in \mathcal{A}$. Because $\mathcal{R}$ was assumed to be irreducible, this rule cannot be contained in $\mathcal{R}$ (otherwise *loop detection* could be applied to $\mathcal{R}$ w.r.t. the same reference set $\mathcal{A}$). However, we now show by an induction on i that $\hat{T}_{P,R} \uparrow i$ also cannot contain any rules which *loop detection* would delete. For $i = 0$ this is trivial. Suppose that $A \leftarrow \mathcal{B}$ is generated by $\hat{T}_{P,R}$ in the $i+1$-st iteration. This implies that $pos(\mathcal{B}) \subseteq heads(\mathcal{R}) \cup heads(\hat{T}_{P,R} \uparrow i)$. By the induction hypothesis, $heads(\hat{T}_{P,R} \uparrow i) \cap \mathcal{A} = \emptyset$, and by the irreducibility of $\mathcal{R}$, $heads(\mathcal{R}) \cap \mathcal{A} = \emptyset$. But then $\mathcal{B} \cap \mathcal{A} = \emptyset$, and thus the first condition in Definition 4.3.14 entails $A \notin \mathcal{A}$. □

Due to Definition 5.3.6 and Lemma 5.3.7 we can refine our algorithm as follows:

Theorem 5.3.8 Let $P^{(1)}, \ldots, P^{(m)}$ be an evaluation sequence for P, and let the ground programs $\mathcal{R}_0, \ldots, \mathcal{R}_m$ be defined as follows:

- $\mathcal{R}_0 := \emptyset$,
- $\mathcal{R}_i$ is the program remainder of $\mathcal{R}_{i-1} \cup \text{lfp}(\hat{T}_{P^{(i)},\mathcal{R}_{i-1}})$, for $0 < i \leq m$.

Then $\mathcal{R}_m$ is the remainder of P. Furthermore, $\mathcal{R}_{i-1} \subseteq \mathcal{R}_i$ for $0 < i \leq m$.

Proof: The set inclusion follows from Lemma 5.3.3 and Lemma 5.3.7. R_m is the remainder of P due to Lemma 5.3.7 and the confluence of $\mapsto_X$. □

Our next goal is to avoid the costly *loop detection* whenever possible. First, *loop detection* is obviously only needed if a program component contains positive recursion. For instance, in the above examples, $even$ calls itself only through negation. In this case, *loop detection* is not needed.

Lemma 5.3.9 Let P and $\mathcal{R}$ be ground programs such that $\mathcal{R}$ does not depend on P. Let further $\mathcal{R}$ be irreducible w.r.t. $\mapsto_X$, and let P contain no predicate which depends positively on itself. If $(\mathcal{R} \cup P) \mapsto_L P'$, then $(\mathcal{R} \cup P) \mapsto_F^* P'$.

Proof: Let $\mathcal{A}$ be the unfounded set on which this application of *loop detection* is based. Since $\mathcal{R}$ is irreducible, $heads(R) \cap \mathcal{A} = \emptyset$ holds and thus $\mathcal{R}$ is not affected by $\mapsto_L$. Now let $\mathcal{A}$ be partitioned into subsets $\mathcal{A}_1, \ldots, \mathcal{A}_n$ such that P contains no rule $A \leftarrow \mathcal{B}$ with $A \in \mathcal{A}_i$, $pos(\mathcal{B}) \cap \mathcal{A}_j \neq \emptyset$ and $i \leq j$. Since P contains no positive recursion, such a partition exists.

Loop detection deletes all rules with an element of $\mathcal{A}$ in the head. We now show by induction on i, $i \geq 1$, that we can eliminate all rules with an element of $\mathcal{A}_i$ in the head by $\mapsto_F$.

$i = 1$: The elements of $\mathcal{A}_1$ cannot appear in rule heads, since such a rule must contain an element of $\mathcal{A}$ in the body (violating the acyclicity condition).

$i - 1 \rightarrow i$: Let $A \leftarrow \mathcal{B} \in P$ such that $A \in \mathcal{A}_i$. Then there is a $j < i$ and a $B \in \mathcal{B}$ such that $B \in \mathcal{A}_j$, i.e., $B \in \mathcal{A}_1 \cup \ldots \cup \mathcal{A}_{i-1}$. But then any rule having B as its head atom has been deleted by *failure*. Thus, $A \leftarrow \mathcal{B}$ can be deleted by *failure*, too. □

In the scenario of Lemma 5.3.9 all applications of *loop detection* can be replaced by a number of applications of *failure* – there are no "proper" applications of *loop detection*. This means that in the algorithm of Corollary 5.3.8, we need only $\mapsto_{PSNF}$ to compute the program remainder $\mathcal{R}_i$ of $\mathcal{R}_{i-1} \cup \mathrm{lfp}(\hat{T}_{P^{(i)},\mathcal{R}_{i-1}})$.

However, this is not the only case where this much more efficient computation scheme is applicable. One single application of *loop detection* is sufficient even in the presence of positive recursion as long as no negative recursion occurs in the same program component. This is due to the fact that once a program is irreducible w.r.t. *loop detection*, only *negative reduction* that deletes a possible support of a ground atom can enable another application of *loop detection* to this program.

Lemma 5.3.10 Let P_1 and P_2 be ground programs with $P_1 \mapsto_{PSF} P_2$ (where $\mapsto_{PSF}$ is the union of *positive reduction*, *success* and *failure*). If P_1 is irreducible w.r.t. $\mapsto_L$, then P_2 is also irreducible w.r.t. $\mapsto_L$.

Proof: Let P_1 be irreducible w.r.t. $\mapsto_L$. Let us assume that $\mapsto_L$ is applicable to P_2 w.r.t. an unfounded set $\mathcal{A}$ as reference set. If $P_1 \mapsto_{PS} P_2$ holds, $\mathcal{A}$ is also a valid reference set to apply $\mapsto_L$ to P_1, contradicting the irreducibility of P_1 w.r.t. $\mapsto_L$.

Let $P_1 \mapsto_F P_2$ hold. Then there is a rule $A \leftarrow \mathcal{B} \in P_1 \backslash P_2$ and an atom $B \in \mathcal{B}$ with $B \notin heads(P_1)$. But then $\mathcal{A} \cup \{B\}$ is a valid reference set to apply $\mapsto_L$ to P_1, again contradicting the irreducibility of P_1 w.r.t. $\mapsto_L$. □

If all negative literals in a program component have an atom from a lower component, we can first do all applications of *negative reduction* and then apply *loop detection* w.r.t. the greatest unfounded set. After this, *negative reduction* and the *loop detection* do not become applicable again. So a single application of the *loop detection* suffices, and, even better, this *loop detection* is already done implicitly by the intelligent grounding.

Theorem 5.3.11 Let P be a program. Let $\mathcal{R}$ be a ground program which contains no predicate being the head predicate of a rule of P. Further, let $\mathcal{R}$ be irreducible w.r.t. $\mapsto_X$, and let P contain no predicate which depends on itself both, positively and negatively (i.e. through negation). Then the program remainder of $\mathcal{R} \cup P$ can be computed without an explicit application of the *loop detection*:

$$\mathcal{R} \cup \mathrm{lfp}(\hat{T}_{P,\mathcal{R}}) \quad \mapsto^*_{PSNF} \quad \widehat{\mathcal{R} \cup P}.$$

□

Chapter 6

Flexible Scheduling Strategies

6.1 Relation to Alternating Fixpoint

The computation of the set of all possibly true facts that is needed for the *loop detection* $\mapsto_L$ transformation (cf. Definition 4.3.14) is very similar to the computation of possible facts performed in each iteration step of the alternating fixpoint procedure (c.f. Definition 3.2.2). There is an even closer correspondence as we will show in this section.

Our main result is that a particular ordering of our transformations corresponds exactly to the alternating fixpoint procedure. To characterize the computation performed by the alternating fixpoint procedure we define a sequence $(\bar{K}_i, \bar{U}_i)_{i\geq 0}$ where $\bar{K}_i$ and $\bar{U}_i$ are constructed by the program transformations defined in this paper and that is closely related to the sequence $(K_i, U_i)_{i\geq 0}$ computed by the alternating fixpoint procedure (cf. Definition 3.2.2).

Definition 6.1.1 (Alternating Fixpoint by Program Transformations)
Let P be a normal logic program. We define the sequence $(\bar{K}_i, \bar{U}_i)_{i\geq 0}$ of sets $\bar{K}_i$ and $\bar{U}_i$ of conditional facts as follows:

$$\begin{array}{rll} & \bar{K}_0 & \text{is the normal form of } ground(P) \text{ w.r.t. } \mapsto_{LFS}, \\ & & \text{i.e., } ground(P) \mapsto^*_{LF} \text{lfp}(\bar{T}_P) \mapsto^*_S \bar{K}_0, \\ & \bar{U}_0 & \text{is the normal form of } \bar{K}_0 \text{ w.r.t. } \mapsto_{NLF}, \\ i>0: & \bar{K}_i & \text{is the normal form of } \bar{U}_{i-1} \text{ w.r.t. } \mapsto_{PS}, \\ i>0: & \bar{U}_i & \text{is the normal form of } \bar{K}_i \text{ w.r.t. } \mapsto_{NLF}. \end{array}$$

The computation terminates when the sequence becomes stationary, i.e., when a fixpoint is reached in the sense that

$$(\bar{K}_j, \bar{U}_j) = (\bar{K}_{j+1}, \bar{U}_{j+1}).$$

□

Lemma 6.1.2 (Confluence of $\mapsto_{PS}$ and $\mapsto_{NLF}$) In Definition 6.1.1, the normal form of $\bar{U}_i$, $i \geq 0$, w.r.t. $\mapsto_{PS}$ and the normal form of $\bar{K}_i$, $i \geq 0$, w.r.t. $\mapsto_{NLF}$ are uniquely determined.

Proof: To begin with, we recall Lemma 5.2.4 and observe that $\bar{K}_0$ is well-defined. The proof by induction is based on the following reasoning:

1. As one possible application of $\mapsto_{PS}$ never interferes with another possible application of $\mapsto_{PS}$, the transformations $\mapsto_P$ and $\mapsto_S$ can be applied in any order and always produce the same result. In particular, it is legal to apply all possible $\mapsto_P$ transformations first followed by all possible $\mapsto_S$ transformations, because $\mapsto_S$ does not derive any new negative information, i.e., it does not delete any rule instance.

2. Analogously, to construct the normal form w.r.t. $\mapsto_{NLF}$ it is legal to apply all possible $\mapsto_N$ transformations first before applying $\mapsto_{LF}$, since *loop detection* and *failure* do not derive any new facts, i.e., they only delete rule instances.

   All applicable $\mapsto_N$ transformations can be applied in any order because they do not interfere with each other and always delete the same set of rule instances.

   Since the greatest unfounded set of a program is the union of all unfounded sets, the normal form w.r.t. $\mapsto_L$ can be constructed by one application of $\mapsto_L$ w.r.t. this greatest unfounded set (cf. Lemma 4.3.15).

   Finally, rule instances with positive body literals known to be false are deleted by applying $\mapsto_F$. If for an atom A all rule instances having A as their head atom can be deleted by $\mapsto_F$ then $\{A\}$ is an unfounded set and the corresponding rules have already been deleted by $\mapsto_L$. Thus all possible applications of $\mapsto_F$ can be performed in any order.

□

Remark 6.1.3 Due to Lemma 4.3.23, Corollary 4.3.27, and Lemma 5.2.4 the sequence $(\bar{K}_i, \bar{U}_i)_{i\geq 0}$ of Definition 6.1.1 reaches a fixpoint $(\bar{K}_j, \bar{U}_j)$ and computes the program remainder $\widehat{P} = \bar{K}_j = \bar{U}_j$. □

In the first step of the alternating fixpoint procedure, definitely true facts are derived by $K_0 = \mathrm{lfp}(T_{P^+})$. The same facts can be derived by applying the *success* transformation.

For the computation of U_i, $i \geq 0$, those possible facts that are still derivable are recomputed. Rule instances with negative subgoals whose complements are known to be true, i.e. are contained in K_i, are not used. As a byproduct of this recomputation, the generation of any positive cycles is avoided. Disregarding rule instances with false negative subgoals corresponds to the *negative reduction* $\mapsto_N$. However, by this reduction, positive cycles may be produced. Thus, *loop detection* $\mapsto_L$ has to be applied.

Finally, $\mapsto_F$ is applied to remove rule instances with positive body literals known to be false, but without deriving new information.

For the computation of K_i, $i > 0$, the alternating fixpoint procedure derives more definitely true facts by regarding those negative subgoals as true whose complements are not contained in U_{i-1}. Newly derived true facts are used to derive more true facts by rule applications. At the program transformation side, this corresponds to the *positive reduction* $\mapsto_P$ that deletes negative conditions whose complements are known to be false, i.e. not contained in $heads(\bar{U}_{i-1})$. Facts obtained this way can be used to derive further facts by the *success* $\mapsto_S$ transformation that deletes positive conditions that are now known to be true.

Example 6.1.4 Consider the program $EvenNum2$ of Examples 5.1.1 and 5.2.5. The sequence $(\bar{K}_i, \bar{U}_i)_{i\geq 0}$ as defined in Definition 6.1.1 is computed as follows:

$$
\begin{array}{rcl}
\bar{K}_0 & = \{ & even(0), succ(0,1), \ldots, succ(n-1,n), \\
 & & even(1) \leftarrow \mathbf{not}\, even(0), \ldots, even(n) \leftarrow \mathbf{not}\, even(n-1) \,\} \\
\bar{U}_0 & = \{ & even(0), succ(0,1), \ldots, succ(n-1,n), \\
 & & even(2) \leftarrow \mathbf{not}\, even(1), \ldots, even(n) \leftarrow \mathbf{not}\, even(n-1) \,\} \\
\bar{K}_1 & = \{ & even(0), succ(0,1), \ldots, succ(n-1,n), \\
 & & even(2), \\
 & & even(3) \leftarrow \mathbf{not}\, even(2), \ldots, even(n) \leftarrow \mathbf{not}\, even(n-1) \,\} \\
\bar{U}_1 & = \{ & even(0), succ(0,1), \ldots, succ(n-1,n), \\
 & & even(2), \\
 & & even(4) \leftarrow \mathbf{not}\, even(3), \ldots, even(n) \leftarrow \mathbf{not}\, even(n-1) \,\} \\
 & \vdots & \\
\bar{K}_{n/2} & = & \bar{U}_{n/2} = \{even(0), succ(0,1), \ldots, succ(n-1,n), \\
 & & \qquad even(2), even(4), even(6), \ldots, even(n) \,\}
\end{array}
$$

The set $\bar{K}_0$ is computed by fixpoint computation performing $O(n)$ derivation steps. Each set $\bar{K}_i$, for $i > 0$, is generated from $\bar{U}_{i-1}$ by one single application of *positive reduction*:

$$\bar{U}_{i-1} \;\mapsto_P\; \bar{K}_i.$$

Each set $\bar{U}_i$, for $i \geq 0$, can be constructed from $\bar{K}_i$ by one application of *negative reduction* as shown in Example 5.2.5:

$$\bar{K}_i \;\mapsto_N\; \bar{U}_i.$$

However, since $\bar{U}_i$ is defined as the normal form of $\bar{K}_i$ w.r.t. $\mapsto_{NLF}$, *loop detection* has to be applied, if possible. Following Lemma 4.3.15, the set

$$\begin{array}{lcl} poss & := & \mathrm{lfp}(T_{\bar{U}_i,\emptyset}) \\ & = & \bar{K}_0 \cup \{even(2), even(4), \ldots, even(2i), \\ & & \qquad even(2i+1), \ldots, even(n)\} \end{array}$$

serves as a reference set for *loop detection*. Since $\bar{U}_i$ does not contain any rules with positive body literals, the application of *loop detection* has no effect. However, for each set $\bar{U}_i$, $i > 0$, the set $poss$ has to be actually computed, where each computation needs $O(n)$ derivation steps. In summary, using this order of transformations which simulates the computation of the alternating fixpoint procedure, the computation of the well-founded model of $EvenNum2$ needs $O(n^2)$ derivation/transformation steps. This corresponds to the costs of the original alternating fixpoint procedure as shown in Example 5.1.1. In contrast, by choosing another strategy, the time complexity can be reduced to $O(n)$ as shown in Example 5.2.5. □

The relation between the alternating fixpoint computation and the sequence of program transformations defined above is described by the following theorem.

Theorem 6.1.5 Let P be a normal logic program. Let $(K_i, U_i)_{i\geq 0}$ and $(\bar{K}_i, \bar{U}_i)_{i\geq 0}$ be the sequences given by Definitions 3.2.2 and 6.1.1, respectively. Then the following holds: $K_i = facts(\bar{K}_i)$ and $U_i = heads(\bar{U}_i)$. □

To prove the theorem we need some auxiliary lemmata.

Lemma 6.1.6 Let P_1 be any ground program and let P_2 be the normal form of P_1 w.r.t. $\mapsto_S$ (*success*). Then:

$$P_2 = \{A \leftarrow (\mathcal{B} \backslash facts(P_2)) \wedge \mathbf{not}\, \mathcal{C} \mid A \leftarrow \mathcal{B} \wedge \mathbf{not}\, \mathcal{C} \ \in \ P_1\}.$$

Proof: Every rule in P_2 results from a rule of P_1 by removing positive body literals. Only positive body literals can be removed which are known as facts, and the set of facts can only increase during this process. Since P_2 is irreducible w.r.t. $\mapsto_S$, atoms in $facts(P_2)$ cannot occur any longer as positive body literals. □

Lemma 6.1.7 Let P_1 be any ground program and let P_2 be the normal form of P_1 w.r.t. $\mapsto_N$ (*negative reduction*). Then:

$$P_2 = \{A \leftarrow \mathcal{B} \wedge \mathbf{not}\, \mathcal{C} \ \in \ P_1 \mid \mathcal{C} \cap facts(P_1) = \emptyset\}.$$

Proof: *Negative reduction* deletes the rules which have a negative body literal conflicting with a fact. The set of facts is not changed during this process, i.e. $facts(P_1) = facts(P_2)$. □

Lemma 6.1.8 Let P_1 be any ground program and let P_2 be the normal form of P_1 w.r.t. $\mapsto_P$ (*positive reduction*). Then:

$$P_2 = \{A \leftarrow \mathcal{B} \wedge \mathbf{not}(\mathcal{C} \cap heads(P_1)) \mid A \leftarrow \mathcal{B} \wedge \mathbf{not}\,\mathcal{C} \in P_1\}.$$

Proof: *Positive reduction* removes the negative body literals for which the corresponding positive atom does not appear as a head literal. The set of head literals is not changed during this process, i.e. $heads(P_1) = heads(P_2)$. □

Lemma 6.1.9 Let P_1 be any ground program and let P_2 be the normal form of P_1 w.r.t. $\mapsto_{LF}$ (*loop detection* and *failure*). Then: $P_2 = \mathrm{lfp}(\bar{T}_{P_1})$.

Proof:

1. $P_1 \mapsto^*_{LF} \mathrm{lfp}(\bar{T}_{P_1})$ has already been proven in Lemma 5.2.4.

2. We show that $\mathrm{lfp}(\bar{T}_{P_1})$ is irreducible w.r.t. $\mapsto_{LF}$. Let us assume that $\mapsto_L$ is applicable. Let $\mathcal{A}$ be as in the definition of $\mapsto_L$ (Definition 4.3.14). Consider the first iteration of the fixpoint iteration in which a rule instance $A \leftarrow \mathcal{B} \wedge \mathbf{not}\,\mathcal{C}$ is derived with $\mathcal{B} \cap \mathcal{A} \neq \emptyset$. There must be such a rule instance since *loop detection* would not be applicable otherwise. Let $B \in \mathcal{B} \cap \mathcal{A}$. Then a rule instance $B \leftarrow \mathcal{B}' \wedge \mathbf{not}\,\mathcal{C}$ has been derived in a previous iteration. This means that $\mathcal{B}' \cap \mathcal{A} = \emptyset$. But this contradicts the first condition in Definition 4.3.14 which implies that $\mapsto_L$ is not applicable to $\mathrm{lfp}(\bar{T}_{P_1})$. Now let us assume that $\mapsto_F$ is applicable. Then there is a rule instance $A \leftarrow \mathcal{B} \wedge \mathbf{not}\,\mathcal{C}$ and an atom $B \in \mathcal{B}$ that is not contained in $heads(\mathrm{lfp}(\bar{T}_{P_1}))$. But this contradicts the iterative construction of $\mathrm{lfp}(\bar{T}_{P_1})$.

□

Lemma 6.1.10 Let P_1 be any ground program and let P_2 be the normal form of P_1 w.r.t. $\mapsto_{LF}$ (*loop detection* and *failure*). Then:

$$P_2 = \{A \leftarrow \mathcal{B} \wedge \mathbf{not}\,\mathcal{C} \in P_1 \mid \mathcal{B} \subseteq heads(P_2)\}.$$

Proof: This follows immediately from Lemma 6.1.9: Since P_2 is a fixpoint of $\bar{T}_{P_1}$, every rule $A \leftarrow \mathcal{B} \wedge \mathbf{not}\,\mathcal{C} \in P_1$ with $\mathcal{B} \subseteq heads(P_2)$ must be contained in P_2. On the other hand, it follows from the iterative construction of $\mathrm{lfp}(\bar{T}_{P_1})$ that any rule $A \leftarrow \mathcal{B} \wedge \mathbf{not}\,\mathcal{C}$ contained in $\mathrm{lfp}(\bar{T}_{P_1})$ must satisfy $\mathcal{B} \subseteq heads(\mathrm{lfp}(\bar{T}_{P_1}))$. □

Proof: *(of Theorem 6.1.5)* We do not only show $facts(\bar{K}_i) = K_i$ and $heads(\bar{U}_i) = U_i$ but also the following stronger characterizations of $\bar{K}_i$ and $\bar{U}_i$:

$$\begin{aligned}
\bar{K}_0 &= \{A \leftarrow (\mathcal{B}\backslash K_0) \wedge \mathbf{not}\,\mathcal{C} \mid A \leftarrow \mathcal{B} \wedge \mathbf{not}\,\mathcal{C} \in ground(P), \\
&\qquad\qquad \mathcal{B} \subseteq \mathrm{lfp}(T_{P,\emptyset})\} \\
\bar{U}_0 &= \{A \leftarrow (\mathcal{B}\backslash K_0) \wedge \mathbf{not}\,\mathcal{C} \mid A \leftarrow \mathcal{B} \wedge \mathbf{not}\,\mathcal{C} \in ground(P), \\
&\qquad\qquad \mathcal{B} \subseteq U_0,\ \mathcal{C} \cap K_0 = \emptyset\}
\end{aligned}$$

$$\begin{aligned}
\bar{K}_i &= \{A \leftarrow (\mathcal{B}\backslash K_i) \wedge \mathbf{not}(\mathcal{C} \cap U_{i-1}) \mid A \leftarrow \mathcal{B} \wedge \mathbf{not}\,\mathcal{C} \in ground(P), \\
&\qquad\qquad \mathcal{B} \subseteq U_{i-1},\ \mathcal{C} \cap K_{i-1} = \emptyset\}, i > 0 \\
\bar{U}_i &= \{A \leftarrow (\mathcal{B}\backslash K_i) \wedge \mathbf{not}(\mathcal{C} \cap U_{i-1}) \mid A \leftarrow \mathcal{B} \wedge \mathbf{not}\,\mathcal{C} \in ground(P), \\
&\qquad\qquad \mathcal{B} \subseteq U_i,\ \mathcal{C} \cap K_i = \emptyset\}, i > 0.
\end{aligned}$$

The proof is by induction on i.

1. **Proof for $\bar{K}_0$:** Due to Lemma 5.2.4 we already know that $ground(P) \mapsto_{LF} \mathrm{lfp}(\bar{T}_P)$. It is easy to show by induction on the number of derivation steps that

   $$\mathrm{lfp}(\bar{T}_P) = \{A \leftarrow \mathcal{B} \wedge \mathbf{not}\,\mathcal{C} \mid A \leftarrow \mathcal{B} \wedge \mathbf{not}\,\mathcal{C} \in ground(P),\ \mathcal{B} \subseteq T_{P,\emptyset}\}$$

   (because $heads(\bar{T}_P \uparrow j) = T_{P,\emptyset} \uparrow j$). By Definition 6.1.1 $\bar{K}_0$ results from $\mathrm{lfp}(\bar{T}_P)$ by applying $\mapsto_S$ until irreducibility w.r.t. $\mapsto_S$. We show that $facts(\bar{K}_0) = K_0$:

   (a) $K_0 \subseteq facts(\bar{K}_0)$: We show $T_{P^+} \uparrow j \subseteq facts(\bar{K}_0)$ by induction on j: The case $j = 0$ is trivial since $T_{P^+} \uparrow 0 = \emptyset$. Now suppose that A has been derived in step j by applying $A \leftarrow \mathcal{B} \in ground(P^+)$. Then $\mathcal{B} \subseteq T_{P^+} \uparrow (j-1)$ and therefore $\mathcal{B} \subseteq facts(\bar{K}_0)$ by the induction hypothesis. Furthermore we know that $T_{P^+} \uparrow (j-1) \subseteq \mathrm{lfp}(T_{P^+}) \subseteq \mathrm{lfp}(T_{P,\emptyset})$ and therefore $A \leftarrow \mathcal{B} \in \mathrm{lfp}(\bar{T}_P)$. But since $\mathcal{B} \subseteq facts(\bar{K}_0)$ and $\bar{K}_0$ is the normal form of $\mathrm{lfp}(\bar{T}_P)$ w.r.t. $\mapsto_S$, it follows that $A \in facts(\bar{K}_0)$.

   (b) $facts(\bar{K}_0) \subseteq K_0$: The proof is by induction on the number of applications of $\mapsto_S$. All facts contained in $\mathrm{lfp}(\bar{T}_P)$ are also contained in $ground(P^+)$ and thus in K_0. Suppose that A is derived from the rule $A \leftarrow \mathcal{B} \in \mathrm{lfp}(\bar{T}_P)$ by j applications of $\mapsto_S$. Then the atoms in $\mathcal{B}$ are derived in fewer steps, and thus $\mathcal{B} \subseteq K_0$ by the induction hypothesis. Since K_0 is a fixpoint of T_{P^+} and $A \leftarrow \mathcal{B} \in ground(P^+)$ it follows that $A \in K_0$.

   The characterization of $\bar{K}_0$ now follows from $facts(\bar{K}_0) = K_0$ by Lemma 6.1.6 and the introductory remark on $\mathrm{lfp}(\bar{T}_P)$.

2. **Proof for $\bar{U}_0$:** To construct $\bar{U}_0$ from $\bar{K}_0$ we may compute the normal form of $\bar{K}_0$ w.r.t. $\mapsto_N$ first (see proof of Lemma 6.1.2) which we denote by $\bar{N}_0$ in the following. By Lemma 6.1.7 and the characterization of $\bar{K}_0$ already shown above

$$\bar{N}_0 = \{A \leftarrow (\mathcal{B}\backslash K_0) \wedge \mathbf{not}\,\mathcal{C} \mid A \leftarrow \mathcal{B} \wedge \mathbf{not}\,\mathcal{C} \in \mathit{ground}(P), \\ \mathcal{B} \subseteq \mathrm{lfp}(T_{P,\emptyset}),\ \mathcal{C} \cap K_0 = \emptyset\}$$

holds. Let $\bar{U}_0$ be the normal form of $\bar{N}_0$ w.r.t. $\mapsto_{LF}$. We show that $\mathit{heads}(\bar{U}_0) = U_0$:

   (a) $U_0 \subseteq \mathit{heads}(\bar{U}_0)$: Because we have $U_0 = \mathrm{lfp}(T_{P,K_0})$, we prove $T_{P,K_0} \uparrow j \subseteq \mathit{heads}(\bar{U}_0)$ by induction on j. The case $j = 0$ is trivial. Let now $A \in T_{P,K_0} \uparrow j$ have been derived by using the rule instance $A \leftarrow \mathcal{B} \wedge \mathbf{not}\,\mathcal{C}$. Then $\mathcal{B} \subseteq T_{P,K_0} \uparrow (j-1)$ and $\mathcal{C} \cap K_0 = \emptyset$. Also $T_{P,K_0} \uparrow (j-1) \subseteq \mathrm{lfp}(T_{P,K_0}) \subseteq \mathrm{lfp}(T_{P,\emptyset})$, thus $A \leftarrow (\mathcal{B}\backslash K_0) \wedge \mathbf{not}\,\mathcal{C}$ is contained in $\bar{N}_0$. By the induction hypothesis, we have $\mathcal{B} \subseteq \mathit{heads}(\bar{U}_0)$. By Lemma 6.1.10 this implies that $A \leftarrow (\mathcal{B}\backslash K_0) \wedge \mathbf{not}\,\mathcal{C}$ is not deleted by *loop detection* or *failure*. Thus $A \in \mathit{heads}(\bar{U}_0)$.

   (b) $\mathit{heads}(\bar{U}_0) \subseteq U_0$: By Lemma 6.1.9, $\bar{U}_0 = \mathrm{lfp}(\bar{T}_{\bar{N}_0})$. Therefore, we show $\mathit{heads}(\bar{T}_{\bar{N}_0} \uparrow j) \subseteq U_0$ by induction on j. The case $j = 0$ is trivial again. Now suppose that $A \leftarrow (\mathcal{B}\backslash K_0) \wedge \mathbf{not}\,\mathcal{C}$ has been derived in step j. Then $(\mathcal{B}\backslash K_0) \subseteq \mathit{heads}(\bar{T}_{\bar{N}_0} \uparrow (j-1))$ and therefore $(\mathcal{B}\backslash K_0) \subseteq U_0$ by the induction hypothesis. But $K_0 \subseteq U_0$ also holds by Lemma 3.2.5, thus $\mathcal{B} \subseteq U_0$. We also have $\mathcal{C} \cap K_0 = \emptyset$, otherwise the rule instance would not be contained in $\bar{N}_0$. But since $U_0 = \mathrm{lfp}(T_{P,K_0})$, it follows that $A \in U_0$.

   Our characterization of $\bar{U}_0$ follows from $\mathit{heads}(\bar{U}_0) = U_0$ by Lemma 6.1.10.

3. **Proof for $\bar{K}_i$, $i \geq 1$:** To construct $\bar{K}_i$, we first apply *positive reduction* to $\bar{U}_{i-1}$ until irreducibility w.r.t. $\mapsto_P$ and denote the resulting set by $\bar{N}_i$. By the induction hypothesis we have $\mathit{heads}(\bar{U}_{i-1}) = U_{i-1}$ and

$$\bar{U}_{i-1} = \{A \leftarrow (\mathcal{B}\backslash K_{i-1}) \wedge \mathbf{not}(\mathcal{C} \cap U_{i-2}) \mid A \leftarrow \mathcal{B} \wedge \mathbf{not}\,\mathcal{C} \in \mathit{ground}(P), \\ \mathcal{B} \subseteq U_{i-1},\ \mathcal{C} \cap K_{i-1} = \emptyset\}.$$

   By Lemma 6.1.8 and since $U_{i-1} \subseteq U_{i-2}$ (Lemma 3.2.5) it follows that

$$\bar{N}_i = \{A \leftarrow (\mathcal{B}\backslash K_{i-1}) \wedge \mathbf{not}(\mathcal{C} \cap U_{i-1}) \mid A \leftarrow \mathcal{B} \wedge \mathbf{not}\,\mathcal{C} \in \mathit{ground}(P), \\ \mathcal{B} \subseteq U_{i-1},\ \mathcal{C} \cap K_{i-1} = \emptyset\}$$

   holds. Now we apply $\mapsto_S$ until irreducibility w.r.t. $\mapsto_S$, the result is $\bar{K}_i$. We show that $\mathit{facts}(\bar{K}_i) = K_i$:

(a) $facts(\bar{K}_i) \subseteq K_i$: The proof is by induction on the number of applications of $\mapsto_S$. Let $A \in facts(\bar{K}_i)$. Either $A \in facts(\bar{K}_{i-1})$ and the main induction hypothesis gives $A \in K_{i-1} \subseteq K_i$ or A results from a rule

$$A \leftarrow (\mathcal{B} \backslash K_{i-1}) \wedge \mathbf{not}(\mathcal{C} \cap U_{i-1})$$

in $\bar{N}_i$ where $\mathcal{C} \cap U_{i-1} = \emptyset$ and $\mathcal{B} \backslash K_{i-1}$ can be derived as facts with fewer applications of $\mapsto_S$. Then the induction hypothesis gives $(\mathcal{B} \backslash K_{i-1}) \subseteq K_i$. Furthermore $K_{i-1} \subseteq K_i$ by Lemma 3.2.5. Thus $\mathcal{B} \subseteq K_i$. But then the rule instance $A \leftarrow \mathcal{B} \wedge \mathbf{not}\, \mathcal{C}$ is applicable in the computation of $\mathrm{lfp}(T_{P,U_{i-1}})$ and we get $A \in K_i$.

(b) $K_i \subseteq facts(\bar{K}_i)$: Because of the equality $K_i = \mathrm{lfp}(T_{P,U_{i-1}})$, we show $(T_{P,U_{i-1}} \uparrow j) \subseteq facts(\bar{K}_i)$ by induction on j: The case $j = 0$ is trivial. Now suppose that A has been derived in step j by applying $A \leftarrow \mathcal{B} \wedge \mathbf{not}\, \mathcal{C}$. Then $\mathcal{B} \subseteq (T_{P,U_{i-1}} \uparrow (j-1))$ and $\mathcal{C} \cap U_{i-1} = \emptyset$. The first part gives us $\mathcal{B} \subseteq K_i$ which together with Lemma 3.2.5 implies $\mathcal{B} \subseteq K_i \subseteq U_i \subseteq U_{i-1}$. $\mathcal{C} \cap U_{i-1} = \emptyset$ implies, again using Lemma 3.2.5, that $\mathcal{C} \cap K_{i-1} = \emptyset$. Together we can conclude that

$$A \leftarrow (\mathcal{B} \backslash K_{i-1}) \wedge \mathbf{not}(\mathcal{C} \cap U_{i-1})$$

is contained in $\bar{N}_i$. Moreover, $\mathcal{C} \cap U_{i-1} = \emptyset$ shows that there are actually no negative body literals, so the rule applied is $A \leftarrow (\mathcal{B} \backslash K_{i-1})$. $\mathcal{B} \subseteq (T_{P,U_{i-1}} \uparrow (j-1))$ together with the induction hypothesis gives us $\mathcal{B} \subseteq facts(\bar{K}_i)$. But since $\bar{K}_i$ is irreducible w.r.t. $\mapsto_S$, and $A \leftarrow (\mathcal{B} \backslash K_{i-1})$ is contained in $\bar{N}_i$, we can finally conclude $A \in \bar{K}_i$.

Our characterization now follows from $facts(\bar{K}_i) = K_i$ by Lemma 6.1.6 and Lemma 3.2.5 ($K_{i-1} \subseteq K_i$).

4. **Proof for $\bar{U}_i$, $i \geq 1$:** To construct $\bar{U}_i$ from $\bar{K}_i$, we first compute the normal form of $\bar{K}_i$ w.r.t. $\mapsto_N$ which we denote by $\bar{N}_i$. By Lemma 6.1.7

$$\bar{N}_i = \{A \leftarrow \mathcal{B} \wedge \mathbf{not}\, \mathcal{C} \in \bar{K}_i \mid \mathcal{C} \cap facts(\bar{K}_i) = \emptyset\}.$$

As we have just seen, the induction hypothesis implies the characterization of $\bar{K}_i$ and $facts(\bar{K}_i) = K_i$. Therefore and since $K_{i-1} \subseteq K_i$ (see Lemma 3.2.5) we can conclude that

$$\bar{N}_i = \{A \leftarrow (\mathcal{B} \backslash K_i) \wedge \mathbf{not}(\mathcal{C} \cap U_{i-1}) \mid A \leftarrow \mathcal{B} \wedge \mathbf{not}\, \mathcal{C} \in ground(P), \\ \mathcal{B} \subseteq U_{i-1},\ \mathcal{C} \cap K_i = \emptyset\}$$

holds. Let $\bar{U}_i$ be the normal form of $\bar{N}_i$ w.r.t. $\mapsto_{LF}$. We show $heads(\bar{U}_i) = U_i$:

(a) $heads(\bar{U}_i) \subseteq U_i$: By Lemma 6.1.9, $\bar{U}_i = \mathrm{lfp}(\bar{T}_{\bar{N}_i})$. Therefore, we show $heads(\bar{T}_{\bar{N}_i} \uparrow j) \subseteq U_i$ by induction on j. The case $j = 0$ is trivial. Now suppose that

$$A \leftarrow (\mathcal{B} \backslash K_i) \wedge \mathbf{not}(\mathcal{C} \cap U_{i-1})$$

has been derived in step j. Then $(\mathcal{B} \backslash K_i) \subseteq heads(\bar{T}_{\bar{N}_i} \uparrow (j-1))$ and therefore $(\mathcal{B} \backslash K_i) \subseteq U_i$ by the induction hypothesis. But $K_i \subseteq U_i$ also holds by Lemma 3.2.5, and thus $\mathcal{B} \subseteq U_i$. We also have $\mathcal{C} \cap K_i = \emptyset$, otherwise the derived ground rule would not be contained in $\bar{N}_i$. But $U_i = \mathrm{lfp}(T_{P,K_i})$, and we can apply $A \leftarrow \mathcal{B} \wedge \mathbf{not}\, \mathcal{C}$ in the computation of $\mathrm{lfp}(T_{P,K_i})$, so $A \in U_i$.

(b) $U_i \subseteq heads(\bar{U}_i)$: Since $U_i = \mathrm{lfp}(T_{P,K_i})$, we prove

$$(T_{P,K_i} \uparrow j) \subseteq heads(\bar{U}_i)$$

by induction on j. The case $j = 0$ is trivial. Let now $A \in (T_{P,K_i} \uparrow j)$ be derived by applying the rule instance $A \leftarrow \mathcal{B} \wedge \mathbf{not}\, \mathcal{C}$. Then $\mathcal{B} \subseteq (T_{P,K_i} \uparrow (j-1))$ and $\mathcal{C} \cap K_i = \emptyset$. Since $T_{P,K_i} \uparrow (j-1) \subseteq \mathrm{lfp}(T_{P,K_i}) = U_i$ and $U_i \subseteq U_{i-1}$ by Lemma 3.2.5, we get $\mathcal{B} \subseteq U_{i-1}$. Therefore, the ground rule

$$A \leftarrow (\mathcal{B} \backslash K_i) \wedge \mathbf{not}(\mathcal{C} \cap U_{i-1})$$

is contained in $\bar{N}_i$. From $\mathcal{B} \subseteq (T_{P,K_i} \uparrow (j-1))$ the induction hypothesis gives us $\mathcal{B} \subseteq heads(\bar{U}_i)$. But then $A \in heads(\bar{U}_i)$ follows from Lemma 6.1.10.

Our characterization of $\bar{U}_i$ follows from $heads(\bar{U}_i) = U_i$ by Lemma 6.1.10 and Lemma 3.2.5 ($U_i \subseteq U_{i-1}$).

□

The proof of Theorem 6.1.5 shows that the program transformation approach will never do more work than the alternating fixpoint procedure. Although the set $\bar{U}_0$ contains more elements than U_0, the same number of derivation steps is needed to compute U_0. Consider the case that one possible fact $A \in U_0$ is represented by more than one conditional fact $A \leftarrow \mathcal{C}$. These conditional facts are the rule instances used to derive the possible fact A. Then during the computation of U_0, the fact A is derived by the same number of rule instances. Only due to duplicate elimination, the cardinality of U_0 is less than the number of derivation steps needed to compute U_0.

For the computation of the set $\bar{K}_i$, $i > 0$, the same argument holds. Whenever a fact can be derived by the deletion of the body literals of several conditional facts, the fact is repeatedly derived in K_i by several rule instances, and the duplicates are eliminated afterwards. Of course, for the derivation of one fact in $\bar{K}_i$, several transformation steps are needed. But analogously, during the iteration of the $T_{P,U_{i-1}}$ operator, also several body literals have to be checked before the head of one rule instance can be derived.

As proposed in [KSS95], the sets K_i do not have to be recomputed in each iteration step, but can be maintained in terms of increments. This optimization is automatically performed by the transformation approach, since conditional facts that were transformed into facts by the deletion of their body literals are never changed afterwards.

The more interesting part is the recomputation of the possible facts. Since the alternating fixpoint procedure does not memorize by which rule instances the possible facts in U_{i-1} have been derived, the set U_i has to be recomputed in each iteration step. In the transformation approach, possible facts do not always have to be recomputed. Possible facts that become false can rather be deleted directly if one of their body literals has become false. However, by this reduction, positive cycles can be generated and have to be eliminated by the more costly transformation *loop detection*. Although, for many programs the number of applications of *loop detection* can be reduced, we have chosen the worst case for this comparison in the sense that in each iteration step the *loop detection* is applied.

For *loop detection*, those facts have to be computed that are still possibly true. *Negative reduction* prevents us from using rule instances with negative body literals already known to be false. *Loop detection* deletes those conditional facts having a head atom that is not derivable anymore. Exactly the same is done by the alternating fixpoint procedure during the computation of $U_i := \mathrm{lfp}(T_{P,K_i})$. Again, rule instances with negative body literals whose complements are in K_i are excluded.

This comparison illustrates that the computation performed by the alternating fixpoint procedure corresponds to the alternating application of the transformations $\mapsto_{PS}$ and $\mapsto_{NLF}$. Thus, the alternating fixpoint procedure can be seen as one particular implementation of our approach (given by choosing a fixed ordering of applying the transformations).

However, *loop detection* is a costly operation since it needs a recomputation of all possible facts as it is done by the alternating fixpoint procedure. Instead, the cheaper transformation *failure* should be preferred as long as no *loop detection* is needed. One possibility is to apply *loop detection* only if no application of $\mapsto_{PSNF}$ is possible. Due to Theorem 4.3.12 this corresponds to computing the least fixpoint of Fitting's Φ_P operator. This way, many programs can be evaluated more efficiently than by the alternating fixpoint procedure, as illustrated in Examples 3.2.6 and 5.2.5.

In the transformation approach all rule instances that are still relevant are explicitly stored. The alternating fixpoint procedure needs to consider the same rule instances, but generates them on demand without storing them. Thus, while having a (sometimes) better time complexity, the transformation approach is expected to have a space complexity worse than the alternating fixpoint approach.

6.2 Regular Strategy Expressions

In Section 6.1 we have seen that we can choose the evaluation strategy just by rearranging the order of transformation application. To specify a transformation order we introduce the following regular expression syntax.

Inductive Definition 6.2.1 (Regular Strategy Expression) A *regular strategy expression* is inductively defined as follows:

- Let $\mapsto_e$ be a ground transformation. Then e is an *atomic* regular strategy expression.
- If e_1 and e_2 are regular strategy expressions, then $(e_1|e_2)$ is an *alternative* regular strategy expression.
- If e_1 and e_2 are regular strategy expressions, then $(e_1 e_2)$ is a *sequential* regular strategy expression.
- If e is a regular strategy expression, then e^* is a *closure* regular strategy expression.

□

Inductive Definition 6.2.2 (Application of a Regular Strategy Expression) Let P be a ground program. Let e be a regular strategy expression. P' is a possible result of the *application of* e *to* P, $P \overset{e}{\mapsto} P'$, if the following conditions hold:

- If e is an *atomic* regular strategy expression, i.e., there is a ground transformation $\mapsto_e$, then $P \mapsto_e P'$ holds.
- If $e \equiv (e_1|e_2)$ is an alternative regular strategy expression, then $P \overset{e_1}{\mapsto} P'$ or $P \overset{e_2}{\mapsto} P'$ holds.
- If $e \equiv (e_1 e_2)$ is a sequential regular strategy expression, then there exists a ground program P'' such that $P \overset{e_1}{\mapsto} P''$ and $P'' \overset{e_2}{\mapsto} P'$ hold.
- If $e \equiv e_1^*$ is a closure regular strategy expression, then P' results from P by n applications of e_1, with $n \geq 0$.

□

Remark 6.2.3 (Parentheses) We will omit parentheses in regular strategy expressions whenever possible. This will hold for parenthesis at top level or for alternative or sequential compositions of more than two expressions. □

Remark 6.2.4 (Applicability) The simple expression S applies the *success* transformation exactly once. Note, that this expression is applicable only to programs to which $\mapsto_S$ is applicable. In this work we usually will use only such expressions that are applicable to any program. This is possible by using closure expressions like S^* that is applicable to any program P. □

Remark 6.2.5 (Fixpoint Expressions) Consider any closure expression e^*, that allows any number of applications of the expression e. However, if we use the regular strategy expression to specify an executable strategy, we usually mean that the expression e is applied until no further application is possible or until any additional application will have no effect on the state. In this context, we call the expression e^* a *fixpoint expression.* □

Example 6.2.6 (Fitting Expression) Following Theorem 4.3.12, the least fixpoint of the Fitting operator corresponds to the normal form $fitt(P)$ of the rewriting system $\mapsto_{PSNF}$. Thus the computation of $fitt(P)$ for a given program P can be expressed by the regular strategy expression

$$(P|S|N|F)^*.$$

□

Lemma 6.2.7 (Complexity of Fitting Expression) Let P be a ground program. Let n be the size of P, i.e., the number of occurrences of literals in P. Then the strategy $(P|S|N|F)^*$ can be applied to P in linear time w.r.t. n.

Proof: To compute the normal form of P w.r.t. $\mapsto^*_{PSNF}$ each body literal of P has to be processed at most one time. An algorithm for a linear time implementation of the fitting reduction is given in Section 8.2. □

Remark 6.2.8 (Loop Detection) The expression L specifies to apply *loop detection* w.r.t. any unfounded set. We will use the expression L^* to express the application of *loop detection* until no further application is possible. Following Lemma 4.3.15 this condition is satisfied by one application of *loop detection* w.r.t. the greatest unfounded set, i.e., by using the set $\mathrm{lfp}(T_{P,\emptyset})$ as set of possibly true atoms. □

Lemma 6.2.9 (Complexity of Loop Detection Expression) Let P be a ground program. Let n be the size of P, i.e., the number of occurrences of literals in P. Then the application of the strategy L^* to P needs linear time w.r.t. n in the worst case.

Proof: This result follows from Lemma 4.3.15 and Remark 6.2.8. □

Example 6.2.10 (AFP Expression) Following Definition 6.1.1 and Theorem 6.1.5 one iteration of the alternating fixpoint procedure can be described by the expressions $(P|S)^*$ for the computation of $\bar{K}_i$ and $(N|L|F)^*$ for the computation of $\bar{U}_i$. Thus, the model computation of the alternating fixpoint procedure can be described by the regular strategy expression

$$((P|S)^*(N|L|F)^*)^*.$$

□

Lemma 6.2.11 (Complexity of AFP Expression) Let P be a ground program. Let n be the size of P, i.e., the number of occurrences of literals in P. Then the application of the strategy $((P|S)^*(N|L|F)^*)^*$ to P needs quadratic time w.r.t. n in the worst case.

Proof: The expression $(P|S)^*(N|L|F)^*$ can be applied at most n times since by every application the size of the program is strictly reduced. The overall time for the application of the expressions P, S, N, and F is $O(n)$ as stated by Lemma 6.2.7. Following Lemma 6.2.9 each application of the *loop detection* needs $O(n)$ time, thus the overall time for applying $((P|S)^*(N|L|F)^*)^*$ is $O(n^2)$ in the worst case. □

In the expression $((P|S)^*(N|L|F)^*)^*$ of Example 6.2.10 the expensive *loop detection* is applied in every iteration. In the following optimized strategy *loop detection* is removed from the inner loop and is only applied if no other transformation is applicable.

Example 6.2.12 (Optimized Remainder Expression) A more efficient strategy to compute the remainder of a given program is expressed by the regular strategy expression

$$((P|S|N|F)^*L^*)^*.$$

□

Lemma 6.2.13 (Complexity of Optimized Remainder Expression) Let P be a ground program. Let n be the size of P, i.e., the number of occurrences of literals in P. In general the application of the strategy $((P|S|N|F)^*L^*)^*$ needs quadratic time w.r.t. n. If P does not contain any positive cycles, then the strategy $((P|S|N|F)^*L^*)^*$ can be applied in linear time w.r.t. n.

Proof: If P does contain positive cycles, the number of needed *loop detections* is in general linear in n. Then quadratic time is needed (cf. Example 4.3.29). If P does not contain positive cycles, the remainder of P is computed by the strategy $(P|S|N|F)^*$ and the outer loop containing the *loop detection* needs only one iteration. In this case only linear time w.r.t. n is needed. □

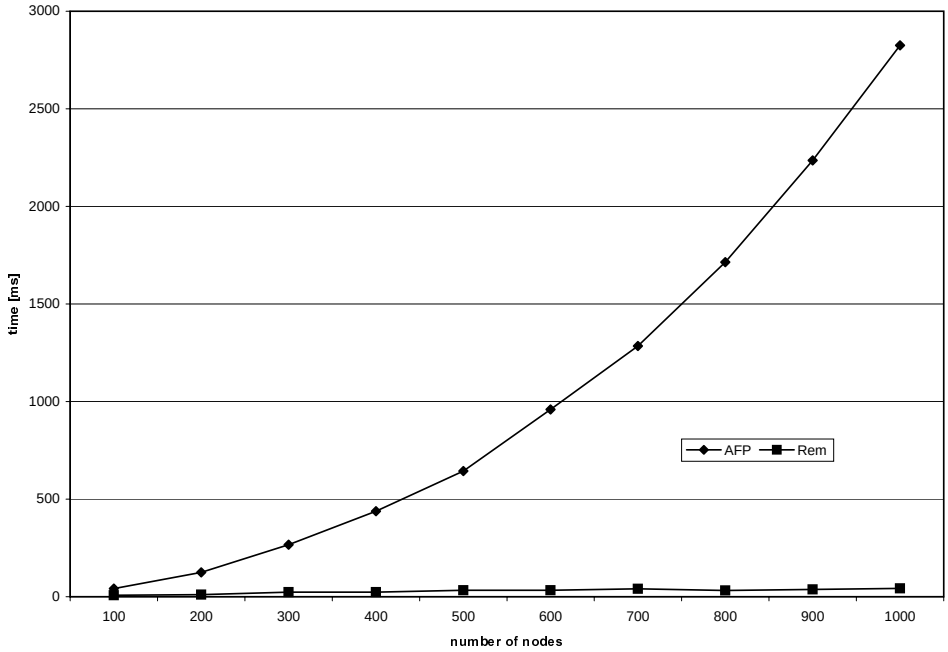


Figure 6.1: Experimental Results: Model Computation

Example 6.2.14 (Experimental Results) Consider the following logic program taken from [KSS95, Mor96]

$$\begin{array}{lcl} p(X) & \leftarrow & t(X,Y,Z), \mathbf{not}\, p(Y), \mathbf{not}\, p(Z). \\ p(X) & \leftarrow & p_0(X). \end{array}$$

and the following base facts:

$$p_0(c_2), t(a,a,b_1), t(b_1,c_1,b_2), t(b_2,c_2,b_3), \ldots, t(b_n,c_n,b_{n+1}).$$

and its ground instantiation. Figure 6.1 illustrates execution times of the AFP strategy from Example 6.2.10 and the optimized remainder strategy from Example 6.2.12 for different program sizes n ranging from 100 to 1000. The quadratic and linear time complexity of the two strategies, respectively, is obvious.

The tests have been run using our prototype implementation in Java on a PC with an Intel Pentium II processor with 450 MHz and 256 MB of main memory, MS Windows NT 4.0 operating system and Sun JDK 1.3 java virtual machine. The implementation is described in Chapter 8. □

Chapter 7

Magic Transformation

7.1 Well-Founded Semantics and Magic Set Transformation

It is well-known that the magic set transformation [BR91, Ram91] causes problems in the context of the well-founded semantics. For a given query the well-founded model of the magic set transformed program and the well-founded model of the original program may not agree on the query, if magic facts get undefined truth values in the model of the transformed program [Ros94, KSS95, SS97].

We will define some notation for using the magic set transformation. However, we assume some familiarity with this topic and omit the details which can be found in [BR91, Ram91, KSS95].

Remark 7.1.1 (Sideways Information Passing Strategy [BR91]) A *sideways information passing strategy* for a rule determines how certain results of the evaluation of one body literal should be used to restrict the evaluation of other body literals. We do not discuss the details of the different sideways information passing strategies, since they are not relevant for our results, and assume that always standard left to right binding propagation is applied. Thus, the result of evaluating one body literal L in the body of a rule r is used to restrict the evaluation of all body literals standing on the right side of L in r. □

Definition 7.1.2 (Magic Set Transformation [BR91, KSS95]) Let P be a normal program. Let S be any sideways information passing strategy. Let Q be a query. Then $MP := Magic(P, S, Q)$ denotes the magic set transformed program of P w.r.t. S and Q. By $PP \subseteq MP$ we denote the set of modified rules of P but without the rules defining the magic predicates. □

Example 7.1.3 (WFS and MST) Consider the following program P.

$$
\begin{array}{lcl}
p(a) & \leftarrow & \mathbf{not}\, p(a), p(b). \\
p(b) & \leftarrow & \mathbf{not}\, p(c), p(d). \\
p(c). & & \\
p(d). & &
\end{array}
$$

The remainder of P is derived by applying *negative reduction* and *failure*:

$$
\begin{array}{l}
p(c). \\
p(d).
\end{array}
$$

Thus the well-founded model $W_P^* = \{\mathbf{not}\, p(a), \mathbf{not}\, p(b), p(c), p(d)\}$ is found easily.

Now consider the magic set transformed version MP of P for the query $p(a)$:

$$
\begin{array}{lcl}
p(a) & \leftarrow & m\_p(a), \mathbf{not}\, p(a), p(b). \\
p(b) & \leftarrow & m\_p(b), \mathbf{not}\, p(c), p(d). \\
p(c) & \leftarrow & m\_p(c). \\
p(d) & \leftarrow & m\_p(d). \\
\\
m\_p(a). & & \\
m\_p(a) & \leftarrow & m\_p(a). \\
m\_p(b) & \leftarrow & m\_p(a), \mathbf{not}\, p(a). \\
m\_p(c) & \leftarrow & m\_p(b). \\
m\_p(d) & \leftarrow & m\_p(b), \mathbf{not}\, p(c).
\end{array}
$$

Only *success* is applicable to MP, yielding the following remainder:

$$
\begin{array}{lcl}
p(a) & \leftarrow & \mathbf{not}\, p(a), p(b). \\
p(b) & \leftarrow & m\_p(b), \mathbf{not}\, p(c), p(d). \\
p(c) & \leftarrow & m\_p(c). \\
p(d) & \leftarrow & m\_p(d). \\
\\
m\_p(a). & & \\
m\_p(b) & \leftarrow & \mathbf{not}\, p(a). \\
m\_p(c) & \leftarrow & m\_p(b). \\
m\_p(d) & \leftarrow & m\_p(b), \mathbf{not}\, p(c).
\end{array}
$$

The well-founded model $W_{MP}^* = \{m\_p(a)\}$ contains only one fact, the remaining formerly contained atoms are undefined. □

7.2 Well-Founded Magic Sets

There are two prominent extensions of the alternating fixpoint procedure that solve the problem illustrated in Section 7.1 by considering undefined magic facts to be true at different steps of the computation. Note, that magic literals only serve as filters to block unused derivations and this change of their truth values cannot compromise correctness. The first of the two approaches mentioned above is the *well-founded magic set* method.

Definition 7.2.1 (Well-Founded Magic Set [KSS95]) Let P be a normal program. Let S be any sideways information passing strategy. Let Q be a query. The *well-founded magic set* method proceeds by the following steps:

1. $MP :\equiv Magic(P, S, Q)$ is the magic set transformed program of P w.r.t. S and Q. Let $PP \subseteq MP$ be defined as in Definition 7.1.2.
2. The well-founded model W^*_{MP} of MP is computed using the alternating fixpoint procedure.
3. Let M denote the set of magic facts that are true or undefined in W^*_{MP}.
4. $PM :\equiv WFMagic(P, S, Q) := PP \cup M$ is the *well-founded magic set transformed* program of P w.r.t. S and Q.
5. The well-founded model W^*_{PM} of PM is computed using the alternating fixpoint procedure.

□

The correctness of the method of Definition 7.2.1 is expressed by the following theorem.

Theorem 7.2.2 (Well-Founded Magic Set [KSS95]) Let P be a normal logic program. Let S be any sip strategy (cf. Definition 7.2.1). Let Q be a query. Let $PM := WFMagic(P, S, Q)$. Then the well-founded models of PM and P agree on Q. □

Example 7.2.3 (Well-Founded Magic Set) Consider the program P and its magic set transformed version MP from Example 7.1.3. In the well-founded model of MP the

magic facts $M = \{m\_p(a), m\_p(b), m\_p(c), m\_p(d)\}$ are true or undefined. The well-founded magic set transformed program PM is given by:

$$
\begin{array}{lcl}
p(a) & \leftarrow & m\_p(a), \mathbf{not}\, p(a), p(b). \\
p(b) & \leftarrow & m\_p(b), \mathbf{not}\, p(c), p(d). \\
p(c) & \leftarrow & m\_p(c). \\
p(d) & \leftarrow & m\_p(d).
\end{array}
$$

$$
\begin{array}{l}
m\_p(a). \\
m\_p(b). \\
m\_p(c). \\
m\_p(d).
\end{array}
$$

All magic facts are true in PM, thus they have no effect on the original rules in this example and the complete well-founded model is computed:

$$W^*_{MP} = \{m\_p(a), m\_p(b), m\_p(c), m\_p(d), \mathbf{not}\, p(a), \mathbf{not}\, p(b), p(c), p(d)\}.$$

□

One disadvantage of the well-founded magic set method is that all undefined magic facts are turned to true facts simultaneously. This can cause unnecessary computations since as a consequence of making some magic facts true it can happen that other magic facts become false, i.e., the computation of the corresponding subquery is not needed to answer the query.

This is the motivation for another approach. The *magic alternating fixpoint* method uses a modified version of the alternating fixpoint procedure where in every computation of true facts all magic facts that were undefined in the preceding computation of true or possible facts are considered true.

Definition 7.2.4 (Magic Alternating Fixpoint [Mor96]) Let P be a normal logic program. Let S be any sip strategy (cf. Definition 7.2.1). Let Q be a query. Let $MP = Magic(P, S, Q)$ be the magic transformed program of P w.r.t. S and Q. Let MP^+ denote the subprogram consisting of the definite rules of MP. The *magic alternating fixpoint procedure* computes the sequence $(K_i, U_i)_{i \geq 0}$ defined as follows:

$$
\begin{array}{lcll}
 & K_0 & := & \mathrm{lfp}(T_{MP^+}) \\
 & U_0 & := & \mathrm{lfp}(T_{MP, K_0}) \\
i > 0: & K_i & := & \mathrm{lfp}(T_{MP \cup magic\_heads(U_{i-1}), U_{i-1}}) \\
i > 0: & U_i & := & \mathrm{lfp}(T_{MP, K_i}).
\end{array}
$$

where $magic\_heads(U_{i-1})$ denotes the set of magic atoms in $heads(U_{i-1})$. □

The correctness of this method is stated by the following theorem.

Theorem 7.2.5 (Magic Alternating Fixpoint [Mor96]) Let P be a normal logic program. Let S be any sip strategy (cf. Definition 7.2.1). Let Q be a query. Let the sequence $(K_i, U_i)_{i\geq 0}$ be defined as above. Then the computation gets stationary, i.e., there is a j such that $K_j = K_{j+1}$ and $U_j = U_{j+1}$. Furthermore, the corresponding interpretation $I = K_j \cup \sim(BASE(P)\backslash U_j)$ and the well-founded model W_P^* of P agree on Q. □

This approach dynamically changes the set of true magic atoms. The advantage of this method is described by the following theorem.

Theorem 7.2.6 (Comparison [KSS95, Mor96]) Let P be a normal logic program. Let Q be a query. The magic alternating fixpoint procedure is guaranteed to terminate after no more iterations than the well-founded magic sets approach, and the set of magic facts in the final set U_j of the magic alternating fixpoint procedure is a subset of the set M of true or undefined magic sets in the well-founded magic sets approach. □

Example 7.2.7 (Magic Alternating Fixpoint) Consider again the magic program MP from Example 7.1.3. The magic alternating fixpoint procedure computes the following sequence of sets of facts:

$$\begin{array}{lcl} K_0 & = & \{m\_p(a)\} \\ U_0 & = & \{m\_p(a), m\_p(b), m\_p(c), m\_p(d), p(a), p(b), p(c), p(d)\} \\ K_1 & = & \{m\_p(a), m\_p(b), m\_p(c), m\_p(d), p(c), p(d)\} \\ U_1 & = & \{m\_p(a), m\_p(b), m\_p(c), p(c)\} \\ K_2 & = & \{m\_p(a), m\_p(b), m\_p(c), p(c)\} \\ U_2 & = & \{m\_p(a), m\_p(b), m\_p(c), p(c)\} \end{array}$$

The atoms $m\_p(d)$ and $p(d)$ are not contained in the computed model, since they are not needed to derive the answer $\mathbf{not}\, p(a)$. □

Remark 7.2.8 (Monotonicity) In Example 7.2.7 it can be observed, that $K_1 \subseteq K_2$ does not hold. Thus the monotonicity property of the alternating fixpoint procedure (cf. Lemma 3.2.5 does not hold for the magic alternating fixpoint procedure. □

7.3 Magic Reduction

Both methods defined in Section 7.2 are based on the alternating fixpoint procedure. Thus they both suffer from the same disadvantages, i.e., they perform unnecessary recomputations of possible facts. In this section we extend our transformation system in a way that the results of Chapter 6 apply to magic programs.

To describe and analyze the methods defined in Section 7.2 we need another transformation that allows us to use undefined magic facts as if they were true.

Definition 7.3.1 (Magic Reduction) Let P_1 and P_2 be ground programs. P_2 results from P_1 by *magic reduction* ($P_1 \mapsto_M P_2$) iff there is a rule $A \leftarrow \mathcal{B}$ in P_1 with head predicate p and a positive body literal $B \equiv magic\_p(\vec{t})$ such that $B \in heads(P_1)$ and $P_2 = (P_1 \backslash \{A \leftarrow \mathcal{B}\}) \cup \{A \leftarrow (\mathcal{B} \backslash \{B\})\}$. □

Our extended transformation system is defined as follows.

Definition 7.3.2 (MX-Transformation) Let $\mapsto_{MX}$ denote the extended rewriting system:

$$\mapsto_{MX} := \mapsto_X \cup \mapsto_M .$$

□

Example 7.3.3 (MX-Transformation) Consider the remainder of the magic transformed program MP from Example 7.1.3. By one application of *magic reduction* we derive the fact $p(c)$ and get the following program:

$$\begin{array}{lcl} p(a) & \leftarrow & \mathbf{not}\, p(a), p(b). \\ p(b) & \leftarrow & m\_p(b), \mathbf{not}\, p(c), p(d). \\ p(c). & & \\ p(d) & \leftarrow & m\_p(d). \\ & & \\ m\_p(a). & & \\ m\_p(b) & \leftarrow & \mathbf{not}\, p(a). \\ m\_p(c) & \leftarrow & m\_p(b). \\ m\_p(d) & \leftarrow & m\_p(b), \mathbf{not}\, p(c). \end{array}$$

By some applications of the rewriting system $\mapsto_X$ we reach the following program

$$\begin{array}{l} p(c). \\ m\_p(a). \\ m\_p(b). \\ m\_p(c). \end{array}$$

that is irreducible w.r.t. $\mapsto_{MX}$. □

Note that this new transformation system is not confluent any more. Depending on the application of the *magic reduction* transformation different strategies for handling undefined magic facts can be realized.

Example 7.3.4 (Non-Confluence of $\mapsto_{MX}$) Consider again the remainder of the magic transformed program MP from Example 7.1.3. If we apply *magic reduction* two times to derive $p(c)$ and $p(d)$ we get the following program

$$
\begin{array}{l}
p(c). \\
p(d). \\
m\_p(a). \\
m\_p(b). \\
m\_p(c).
\end{array}
$$

that is also irreducible w.r.t. $\mapsto_{MX}$ but contains the additional fact $p(d)$. □

However, the following proposition and theorem state that any strategy based on the MX-transformation leads to a correct answer for the given query.

Proposition 7.3.5 Let P be a normal logic program. Let S be any sip strategy (cf. Remark 7.1.1). Let Q be a query. Let $MP = Magic(P, S, Q)$ and $PM = WFMagic(P, S, Q)$ be as defined above. Let R be the remainder of MP. Let R' be the normalform of R w.r.t. *magic reduction*. Let P' be the remainder of R'. I.e., we have

$$MP \mapsto_X^* R \mapsto_M^* R' \mapsto_X^* P'$$

Then the well-founded models of PM and P' agree on Q.

Proof: From Definition 7.2.1 it follows that $PM = PP \cup M$. Consequently we have

$$W_{PM}^* = W_{PP \cup M}^*.$$

MP consists of PP plus the definitions of the magic predicates. Since M contains all magic atoms, that are true or undefined in the model of MP we have

$$W_{PP \cup M}^* = W_{MP \cup M}^*$$

From

$$MP \mapsto_X^* R$$

and $M \subseteq heads(R)$ it follows that

$$MP \cup M \mapsto_X^* R \cup M$$

holds. Thus we have

$$W^*_{MP \cup M} = W^*_{R \cup M}$$

By the transformation $R \mapsto^*_M R'$ all body literals contained in M are deleted from the rule bodies in R. This is equivalent to adding all facts in M to R and apply *success* to remove these facts from the rule bodies. As a consequence, we have

$$R \cup M \mapsto^*_S R' \cup M$$

and thus

$$W^*_{R \cup M} = W^*_{R' \cup M}$$

Since all atoms in M have been removed from rule bodies in R', it follows that

$$W^*_{R' \cup M} = W^*_{R'} \cup M$$

holds. P' is the remainder of R', thus

$$W^*_{R'} \cup M = W^*_{P'} \cup M$$

holds. Together we have

$$W^*_{PM} = W^*_{P'} \cup M$$

From this it follows that the well-founded models of PM and P' differ only in the truth values of magic atoms, thus they agree on Q. □

Theorem 7.3.6 (Correctness of the MX-Transformation) Let P be a normal logic program. Let S be any sip strategy (cf. Remark 7.1.1). Let Q be a query. Let $MP = Magic(P, S, Q)$ be as defined above. Let P' be any normal form of MP w.r.t. $\mapsto_{MX}$. Then the well-founded models of P' and P agree on Q.

Proof: Let us first assume that P' has been derived by the sequence of transformations defined in Proposition 7.3.5:

$$MP \mapsto^*_X R \mapsto^*_M R' \mapsto^*_X P'$$

Then from Proposition 7.3.5 it follows that the well-founded models of P' and $PM = WFMagic(P, S, Q)$ coincide on Q. Together with Theorem 7.2.2 it follows that the well-founded models of P' and P agree on Q.

Now consider any other transformation order

$$MP \mapsto^*_{MX} P'.$$

If exactly the magic facts from the set M as defined above are used for *magic reduction* then the result is the same as above.

If more magic facts than contained in M are used for *magic reduction* then possibly more rules have been used to derive facts than in the original sequence. Thus, possibly a larger result has been computed. However, since all rules are modified rules from the original program P, only correct answers w.r.t. the well-founded model of P can be derived.

But it is possible that not all magic facts from M have been used for *magic reduction*. Assume that a conditional magic fact with head $A \equiv m\_p(\vec{t})$ gets deleted before all occurrences of A in the bodies of other rules have been removed by applications of *magic reduction*. In this case the magic fact has become false because at least one of the body literals in each of its conditional facts has become false. But then also the rule instance the magic fact had been generated for can be deleted, because its body contains all literals that the magic fact depends on, due to the definition of the magic set transformation. Thus, the deleted magic fact is not needed to answer the given query. □

We now extend our regular strategy expression syntax by the new transformation.

Remark 7.3.7 (Regular Strategy Expression (cont.)) Since $\mapsto_M$ is a ground transformation, the single letter M is also an atomic regular strategy expression in the sense of Definition 6.2.1. □

According to Definition 6.2.2 and Remark 6.2.5, the expression M^* denotes the computation of the normal form w.r.t. $\mapsto_M$, i.e. the removal of all true or undefined magic filters from rules with heads with non-magic predicates. This allows us to embed the well-founded magic set method as a special case of our transformation method.

Definition 7.3.8 (Well-Founded Magic Sets Strategy) The *well-founded magic sets regular strategy expression* is defined by

$$((P|S)^*(N|L|F)^*)^* M^* ((P|S)^*(N|L|F)^*)^*.$$

□

Lemma 7.3.9 The computation of the well-founded magic sets method of Definition 7.2.1 can be described by the well-founded magic sets regular strategy expression of Definition 7.3.8.

Proof: This result follows from Theorem 6.1.5, Example 6.2.10, and Definition 7.3.1 □

It is obvious that the well-founded magic set strategy can be optimized by delaying the *loop detection* transformation.

Definition 7.3.10 (Well-Founded Remainder) The *well-founded remainder* regular strategy expression is defined by

$$((P|S|N|F)^*L^*)^*M^*((P|S|N|F)^*L^*)^*.$$

□

Lemma 7.3.11 Let MP be a magic set transformed ground program. The execution of the well-founded remainder regular strategy expression of Definition 7.3.10 applied to MP is guaranteed to need no more work than the computation corresponding to the well-founded magic sets expression of Definition 7.3.8.

Proof: This statement follows directly from Theorem 6.1.5. □

We will show in Example 7.3.23 that the well-founded remainder strategy can be strictly more efficient than the well-founded magic set strategy for some programs.

One disadvantage of the two strategies in Lemma 7.3.11 is that by the application of the *magic reduction* transformation all undefined magic facts are considered to be true and removed from all bodies of non-magic rules. This can cause unnecessary computations since as a consequence of making some magic facts true it can happen that other magic facts become false, i.e., the computation of the corresponding subquery is not needed to answer the query.

In the magic alternating fixpoint procedure [Mor96] undefined magic facts are considered to be true only if this enables the derivation of new facts. This can be described by a special case of our *magic reduction* transformation. The *restricted magic reduction* transformation allows to remove a magic filter with undefined truth value from a non-magic rule only if it is the only body literal, i.e., the application transforms the rule into a fact.

Definition 7.3.12 (Restricted Magic Reduction) Let P_1 and P_2 be ground programs. P_2 results from P_1 by *restricted magic reduction* ($P_1 \mapsto_{M_R} P_2$) iff there is a rule $A \leftarrow \{B\}$ in P_1 with head predicate p and a positive body literal $B \equiv magic\_p(\vec{t})$ such that $B \in heads(P_1)$ and $P_2 = (P_1 \backslash \{A \leftarrow \{B\}\}) \cup \{A\}$. □

Definition 7.3.13 (RX-Transformation) Let $\mapsto_{RX}$ denote the extended rewriting system:

$$\mapsto_{RX} := \mapsto_X \cup \mapsto_{M_R}.$$

□

It is easy to see that this *restricted magic reduction* transformation is sufficient to evaluate magic set transformed programs.

Lemma 7.3.14 (Correctness of RX-Transformation) Let MP be a magic set transformed ground program. Let P' be a normal form of MP w.r.t. $\mapsto_{RX}$. Let P'' be a normal form of P' w.r.t. $\mapsto_{MX}$, i.e.,

$$MP \mapsto_{RX}^* P' \mapsto_{MX}^* P''.$$

Then $known_{BASE(MP)}(P') = known_{BASE(MP)}(P'')$ holds.

Proof: Let P' be a normal form of MP w.r.t. $\mapsto_{RX}$. Then P' is irreducible w.r.t. $\mapsto_{RX}$. If P' is irreducible w.r.t. $\mapsto_{MX}$, then $P' = P''$ follows and the result is proven. Now assume that P' is not irreducible w.r.t. $\mapsto_{MX}$. Let P'_1 be a result of applying $\mapsto_{MX}$ to P', i.e.,

$$P' \mapsto_{MX} P'_1.$$

Since P' is irreducible w.r.t. $\mapsto_{RX}$, the only transformation applicable to P' is $\mapsto_M$, thus we have

$$P' \mapsto_M P'_1.$$

According to Definition 7.3.1, the following holds:

$$P'_1 = P' \backslash \{A \leftarrow m\_p(\vec{t}) \wedge \mathcal{B}\} \cup \{A \leftarrow \mathcal{B}\}.$$

Now assume that $\mathcal{B} = \emptyset$. Then

$$P' \mapsto_{M_R} P'_1$$

follows. But this contradicts the irreducibility of P' w.r.t. $\mapsto_{RX}$. Thus $\mathcal{B} \neq \emptyset$ holds. From this it follows that $heads(P') = heads(P'_1)$ and $facts(P') = facts(P'_1)$ and thus

$$known_{BASE(MP)}(P') = known_{BASE(MP)}(P'_1)$$

hold. Further, P'_1 is also irreducible w.r.t. $\mapsto_{RX}$. By induction, we conclude that

$$P' \mapsto_{M_R}^* P''$$

and

$$known_{BASE(MP)}(P') = known_{BASE(MP)}(P'')$$

hold. □

Remark 7.3.15 (Restricted Magic Reduction is Sufficient) From Lemma 7.3.14 it follows that for every transformation sequence of $\mapsto_{RX}$ applied to MP there is a transformation sequence of $\mapsto_{MX}$ that computes the same result. Together with Theorem 7.3.6 which states the correctness of the MX-transformation, we know that every transformation sequence of the RX-transformation leads to a correct result. □

The following definition allows us to use the *restricted magic reduction* transformation in regular strategy expressions.

Remark 7.3.16 (Regular Strategy Expression (cont.)) Since $\mapsto_{M_R}$ is a ground transformation, the single letter M_R is also an atomic regular strategy expression in the sense of Definition 6.2.1. □

Now we can express the magic alternating fixpoint procedure as a special case of our rewriting system $\mapsto_{MX}$.

Definition 7.3.17 (Magic Alternating Fixpoint Strategy) The Magic Alternating Fixpoint regular strategy expression is defined by

$$((P|S|M_R)^*(N|L|F)^*)^*.$$

□

The following theorem describes the relation between the magic alternating fixpoint procedure and the RX-Transformation.

Theorem 7.3.18 Let MP be a magic set transformed ground program. Let the magic alternating fixpoint sequence $(K_i, U_i)_{i\geq 0}$ be defined as in Definition 7.2.4. Let the sequence $(\bar{K}_i, \bar{U}_i)_{i\geq 0}$ be defined as in Definition 6.1.1 except that $\bar{K}_i, i > 0$ is the normal form of $\bar{U}_{i-1}$ w.r.t. $\mapsto_{PSM_R}$. Then $U_i = heads(\bar{U}_i)$ holds. Further, $K_i = facts(\bar{K}_i)$ holds for non-magic facts, i.e., facts having no magic predicate.

Proof: By Theorem 6.1.5 we have the following result for the original alternating fixpoint procedure:

$$\begin{array}{rlcll} & K_0 & := & \text{lfp}(T_{MP^+}) & = & facts(\bar{K}_0) \\ & U_0 & := & \text{lfp}(T_{MP,K_0}) & = & heads(\bar{U}_0) \\ i>0: & K_i & := & \text{lfp}(T_{MP,U_{i-1}}) & = & facts(\bar{K}_i) \\ i>0: & U_i & := & \text{lfp}(T_{MP,K_i}) & = & heads(\bar{U}_i) \end{array}$$

and

$$ground(MP) \;\mapsto^*_{LFS}\; \bar{K}_0 \;\mapsto^*_{NLF}\; \bar{U}_0 \ldots \;\mapsto^*_{PS}\; \bar{K}_i \;\mapsto^*_{NLF}\; \bar{U}_i \ldots$$

For the magic alternating fixpoint procedure we have the following modification according to Definition 7.2.4

$$i > 0: \quad K_i \quad := \quad \mathrm{lfp}(T_{MP \cup magic\_heads(U_{i-1}), U_{i-1}})$$

and $\bar{K}_i, i > 0$ is the normal form of $\bar{U}_{i-1}$ w.r.t. $\mapsto_{PSM_R}$ instead of $\mapsto_{PS}$.

We have to show that $K_i = facts(\bar{K}_i)$ holds for non-magic facts. Let

$$T^j \; := \; T_{MP \cup magic\_heads(U_{i-1}), U_{i-1}}{}^{\uparrow j}(\emptyset)$$

be the result of the j-th application of the operator in the least fixpoint computation of K_i. We show by induction over j that $T^j \subseteq facts(\bar{K}_i)$ holds for non-magic facts. The base case for $j = 0$ is trivial. Let $A \in T^{j+1}$ be any non-magic fact, i.e., any atom having a non-magic predicate. Then from Definition 3.2.1 it follows, that there is a rule instance

$$r \; :\equiv \; (A \leftarrow \{A'\} \cup \mathcal{B} \cup \mathbf{not}\, \mathcal{C}) \in ground(MP)$$

such that $A' \in U_{i-1}$ is the magic filter of the rule, $\mathcal{B} \subseteq T^j$ and $\mathcal{C} \cap U_{i-1} = \emptyset$. From the induction hypothesis it follows that $\mathcal{B} \subseteq facts(\bar{K}_i)$ holds. Further we know that $\mathcal{C} \cap heads(\bar{U}_{i-1})$ and $A' \in heads(\bar{U}_{i-1})$ hold as a consequence of Theorem 6.1.5. Then in the transformation sequence for the computation of $\bar{K}_i$ the fact A can be produced by removing all positive body literals in $\mathcal{B}$ from the rule r by applying *success*, removing all negative body literals in $\mathbf{not}\, \mathcal{C}$ by applying *positive reduction*, and finally by removing the magic filter A' by applying the *restricted magic reduction* transformation. Since no body literal of r is false in the current transformation state, the rule r has not been deleted from $ground(MP)$ in the transformation sequence before. It follows that $A \in facts(\bar{K}_i)$ holds.

Now it remains to be shown that $facts(\bar{K}_i) \subseteq K_i$ holds for non-magic facts. From Theorem 6.1.5 we know that this result holds for the original alternating fixpoint procedure. The only modification here is that for the computation of $\bar{K}_i$ the *restricted magic reduction* may be applied as well. Let $A \in facts(\bar{K}_i)$ be an additional fact generated by removing the magic filter A' from a rule of the form

$$A \leftarrow A'.$$

Then $A' \in heads(\bar{U}_{i-1})$ and also $A' \in U_{i-1}$ hold. But then A' is contained in $magic\_heads(U_{i-1})$ and thus also in K_i. As a consequence the fact A will be derived by the fixpoint computation for K_i as well. □

Remark 7.3.19 (Monotonicity) Note that the sequence of sets K_i of true facts is not monotonically increasing in the magic alternating fixpoint procedure, because some magic facts and facts depending on them that have once been considered true may possibly not be recomputed in a later iteration step. However, in the magic alternating fixpoint transformation, magic conditional facts may be deleted, but non-magic facts once derived as unconditional facts are never deleted afterwards. Thus, our transformational counterpart of the magic alternating fixpoint procedure has the monotonicity property that is missing in the original approach of Morishita [Mor96]. □

Examining the magic alternating fixpoint strategy expression $((P|S|M_R)^*(N|L|F)^*)^*$ reveals two possible optimizations. The obvious optimization is to delay the application of *loop detection*. The second optimization concerns the time when undefined magic facts are to be considered true. In [KSS95] it is argued that using magic facts too early may cause unnecessary computations since the undefined magic facts may become false later. Therefore we delay also the application of the *restricted magic reduction* in the following strategy.

Definition 7.3.20 (Magic Remainder) The Magic Remainder regular strategy expression is defined by

$$(((P|S|N|F)^* {M_R}^*)^* L^*)^*.$$

□

Lemma 7.3.21 Let MP be a magic set transformed ground program. The execution of the magic remainder regular strategy expression of Definition 7.3.20 applied to MP is guaranteed to need not more work than the computation corresponding to the magic alternating fixpoint expression of Definition 7.3.17.

Proof: The expression $((P|S|N|F)^* {M_R}^*)^*$ can be executed in linear time w.r.t. the given ground program. For the expression $(P|S|N|F)^*$ this has been discussed before. For the *restricted magic reduction* each rule has to be checked at most once, i.e., if all body literals except the magic filter have been removed. This operation adds extra costs that are also linear in the size of MP. The only expensive operation is the *loop detection* transformation which is removed from the inner loop and is applied only if no other transformation is applicable. □

We will show in Example 7.3.23 that the magic remainder strategy is strictly more efficient than the magic alternating fixpoint strategy for some programs.

Remark 7.3.22 (Comparing Strategies) We have optimized our transformation strategies along two dimensions. The first goal was to shift the *loop detection* from

inner loops (in the strategy expression) to outer loops. This way we derived the well-founded remainder strategy

$$((P|S|N|F)^*L^*)^*M^*((P|S|N|F)^*L^*)^*$$

from the well-founded magic set strategy

$$((P|S)^*(N|L|F)^*)^*M^*((P|S)^*(N|L|F)^*)^*.$$

Further, we derived the magic remainder strategy

$$(((P|S|N|F)^*M_R{}^*)^*L^*)^*$$

from the magic alternating fixpoint strategy

$$((P|S|M_R)^*(N|L|F)^*)^*.$$

We expect a significant speedup for programs without positive loops, since the desired normalform can the be computed by the inner expressions and the *loop detection* in the outer loops is not applied or at most one time. Then the execution time can be reduced from quadratic to linear w.r.t. the size of the program. However, if the program contains many positive loops, e.g. such that the number of positive loops is linear in the size of the program, *loop detection* has to be applied many times, then the effect of the optimization will reduced.

The other goal of optimization was to reduce the number of applied *magic reductions*. Based on Theorem 7.2.6 we tried to avoid unnecessary computations by applying *magic reduction* only if necessary. In a first step we introduced the *restricted magic reduction* in Definition 7.3.12 that applies *magic reduction* only where the application leads directly to a new fact, instead of applying it whenever possible. As a consequence we expect the magic alternating fixpoint strategy to perform better than the well-founded magic set strategy and the magic remainder better than the well-founded remainder. □

Example 7.3.23 Consider the following program P taken from [KSS95, Mor96]

$$\begin{array}{lcl} p(X) & \leftarrow & t(X,Y,Z), \mathbf{not}\, p(Y), \mathbf{not}\, p(Z). \\ p(X) & \leftarrow & p_0(X). \end{array}$$

and the following base facts:

$$p_0(c_{\frac{n}{4}}), t(a,a,b_1), t(b_1,c_1,b_2), t(b_2,c_2,b_3), \ldots, t(b_n,c_n,b_{n+1}).$$

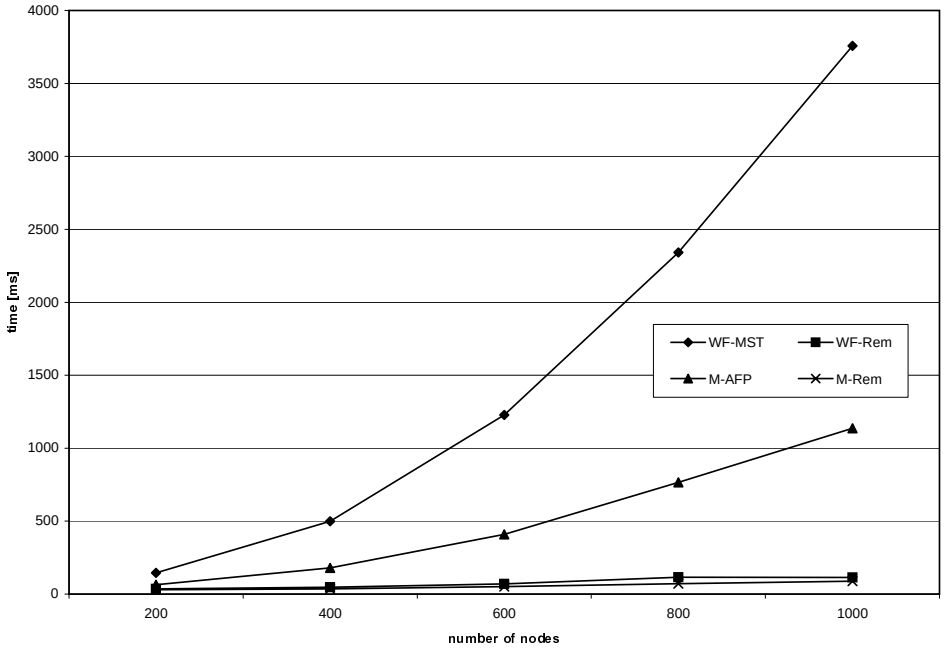


Figure 7.1: Experimental Results: Magic Reduction

Consider the following program MP

$$
\begin{array}{lcl}
p(X) & \leftarrow & m\_p(X), t(X,Y,Z), \mathbf{not}\, p(Y), \mathbf{not}\, p(Z). \\
p(X) & \leftarrow & m\_p(X), p_0(X). \\
m\_p(a). & & \\
m\_p(Y) & \leftarrow & m\_p(X), t(X,Y,Z). \\
m\_p(Z) & \leftarrow & m\_p(X), t(X,Y,Z), p(Y).
\end{array}
$$

that results from the program P by the magic set transformation for the query $p(a)$. Figure 7.1 illustrates the execution times of the four strategies of Definitions 7.3.8 (WF-MST), 7.3.10 (WF-Rem), 7.3.17 (M-AFP) and 7.3.20 (M-Rem) for different program sizes n. It can be seen that the WF-MST strategy considers all magic facts to be true and needs quadratic time to compute the complete well-founded model of the original program. The M-AFP strategy uses only the relevant magic facts and computes only the facts from $p(b_1)$ to $p(b_{\frac{n}{4}})$, but also needs quadratic time. The strategies WF-Rem and M-Rem compute the result in linear time, since the program does not contain positive loops and thus the *loop detection* is not needed.

Consider now the same program but with an additional rule $p(X) \leftarrow p(X)$. Now every p-fact depends positively on itself and a linear number of applications of *loop detection* is needed to compute the result. Figure 7.2 illustrates the execution times of the four strategies for the modified program. The execution times of the strategies

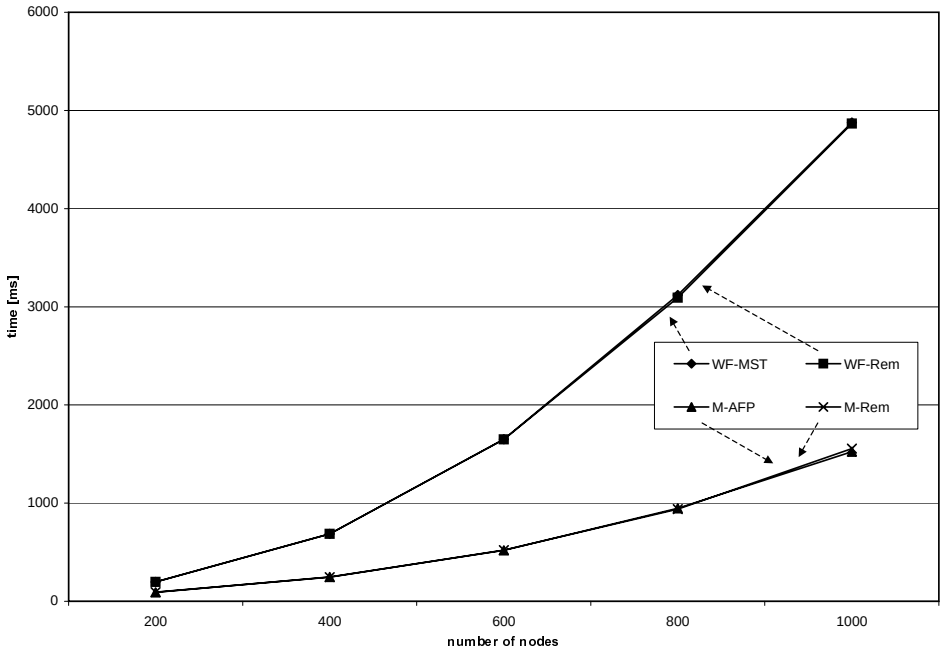


Figure 7.2: Experimental Results: Magic Reduction with Positive Loops

WF-MST and M-AFP coincide with the times for the strategies WF-Rem and M-Rem, respectively, and the delay of the *loop detection* has no effect for this program. □

The experimental results have confirmed the analytical comparison of Remark 7.3.22. For programs without positive loops the remainder strategies both need execution time $O(n)$ where n is the program size. For programs with positive loops, it is important to avoid unnecessary applications of *magic reduction*. In this case the magic alternating fixpoint and the magic remainder strategy perform better than the other two strategies.

In summary, the magic remainder strategy combines the advantages of all other strategies. It reduces the number of applications of *loop detection* and *magic reduction*. Thus, this strategy is a good candidate for a default strategy. It yields good results for all programs we have tested.

As a main result of this chapter we emphasize that the development of the magic remainder evaluation strategy was possible by just analyzing known algorithms by means of finding adequate regular strategy expressions, and by modifying theses expressions in a straightforward way at a high level just by shifting expensive operations from inner to outer loops. Finally, our prototypical implementation confirmed the expected results and proves that the performance of the known computation methods could be optimized significantly, at least for some classes of programs.

Chapter 8

Ground Implementation

In this chapter we give an overview of our prototype implementation of the grounding algorithm of Chapter 5 and the ground transformation defined in the Chapters 4 and 7. We will omit technical details, class charts, architecture diagrams, since they are not relevant for this work. However, we will present the main algorithms that we used and point out where we modified known methods to meet the requirements of this work.

The prototype system is implemented in Java. We used the *Java Development Kit* JDK in the version 1.3 that is freely available from the web site *java.sun.com* of Sun Microsystems. We adopted many features already contained in the standard libraries, such as the collections framework for lists and sets of rules, goals, etc. For the relational algebra we developed an extension of the iterator concept. For parsing queries and programs we built a grammar driven scanner and parser using the java tools *JLex* and *JavaCup* that are also freely available. Using standard Java libraries it was easy to develop a graphical user interface and a servlet based front end, that demonstrates some features of our implementation at our web site *daisy.fmi.uni-passau.de*.

8.1 Relational Grounding

To apply the ground transformations of Chapter 4 to a non-ground program P we need a grounding algorithm to generate a set of ground instances relevant for the well-founded model of P.

One such grounding algorithm is proposed in [NS96].

Definition 8.1.1 (Simple Grounding [NS96]) Let P be a range-restricted normal program without function symbols. Then the grounding algorithm proceeds by the following steps.

1. Start with the given program: $R := P$.
2. Whenever the head B' of a ground rule $B' \leftarrow \mathcal{C}'$ in R is an instance $B' = B\sigma$ of a positive body literal $B \in \mathcal{B}$ of a rule $A \leftarrow \mathcal{B}$ in R, then the instance $(A \leftarrow \mathcal{B})\sigma$ is added to R, if it is new.
3. Repeat step 2 until no new instance can be generated.
4. Return all ground rules contained in R:

$$gr(P) := \{(A \leftarrow \mathcal{B}) \in R \mid A \leftarrow \mathcal{B} \text{ is ground.}\}$$

□

Lemma 8.1.2 ([NS96]) Let P be range-restricted normal program without function symbols. Let $gr(P)$ be the result of the grounding algorithm of Definition 8.1.1 applied to P. Then $W_P^* = W_{gr(P)}^*$ holds. □

However, this algorithm has some disadvantages. The set of R may get large and contain many non-ground instances. Further, it is difficult to efficiently find pairs of unifiable head and body atoms.

In this work we extend the relational implementation of the T_P operator for definite programs such that it can be applied to normal programs. Our work is based on the description in [Fre00]. More details can be found in [Ull88, Ull89, EN94, BR86b, BPRM91, KKTG95, GKB87, SKGB87].

The next definition sketches the approach for definite programs.

Definition 8.1.3 (Definite Relational Expression [Fre00]) Let P be a range-restricted definite program. Then the relational expression E_P for P is constructed by the following steps.

1. Let I be a set of ground atoms given as input to the expression.
2. The relational expression E_B for a body literal B occurring in P selects all atoms from I that are unifiable with B.
3. The relational expression $E_{\mathcal{B}}$ for the body of a rule $A \leftarrow \mathcal{B}$ is the natural join of the expressions E_B of all body literals $B \in \mathcal{B}$, i.e., the join condition is based on the common variables in the body literals.
4. The relational expression $E_{A \leftarrow \mathcal{B}}$ for the rule $A \leftarrow \mathcal{B}$ of P is the projection of $E_{\mathcal{B}}$ to the schema of the head atom A of the rule, i.e., instances of A are produced.
5. The relational expression E_P for the program P is the union of the expressions $E_{A \leftarrow \mathcal{B}}$ for all rules $A \leftarrow \mathcal{B}$ in P.

□

Note, that we defined the expression E_P such that it operates on sets of atoms rather than on sets of tuples. This allows us to formulate the following theorem.

Theorem 8.1.4 (Correctness of Definite Relational Expression [Fre00]) Let P be a range-restricted definite program. Let $I \subseteq BASE(P)$ be a set of atoms from the Herbrand base of P. Then

$$T_P(I) = E_P(I)$$

holds. □

This is the basis for the implementation of the T_P operator of a range-restricted definite program P as a relational expression whose iterative evaluation computes the least fixpoint $lfp(T_P)$ of the T_P operator which gives the well-founded model W_P^* of P by Theorem 2.5.8.

However, we want to implement the $\bar{T}_P$ operator of Definition 5.2.1. To this end we modify the relational expression from Definition 5.2.1 in the following way.

Definition 8.1.5 (Normal Relational Expression) Let P be a range-restricted normal program. Then the relational expression $\bar{E}_P$ for P is constructed by the following steps.

1. Let R be a set of ground rules given as input to the expression.
2. The relational expression $\bar{E}_B$ for a body literal B occurring in P selects all rules from I having a head unifiable with B.
3. The relational expression $\bar{E}_\mathcal{B}$ for the body of a rule $A \leftarrow \mathcal{B}$ is the natural join of the expressions R_B of all *positive* body literals $B \in \mathcal{B}$.
4. The relational expression $\bar{E}_{A \leftarrow \mathcal{B}}$ for the rule $A \leftarrow \mathcal{B}$ of P is the projection of $\bar{E}_\mathcal{B}$ to the schema of the rule $A \leftarrow \mathcal{B}$, i.e., instances of the rule itself are produced.
5. The relational expression $\bar{E}_P$ for the program P is the union of the expressions $\bar{E}_{A \leftarrow \mathcal{B}}$ for all rules $A \leftarrow \mathcal{B}$ in P.

□

The relational expression $\bar{E}_P$ operates on sets of ground rules. We have the following result.

Theorem 8.1.6 (Correctness of Normal Relational Expression) Let P be a range-restricted normal program. Let R be a set of ground rules in the first order language $\mathcal{L}$ of P. Then

$$\bar{T}_P(R) = \bar{E}_P(R)$$

holds.

Proof: The proof is based on Theorem 8.1.4. One application of E_P to a set I of ground atoms gives the set $T_P(I)$. The expression $\bar{E}$ computes the the same set of atoms. The only difference is that not only the atoms but also the corresponding ground instances of rules of P are derived. This corresponds exactly to Definition 5.2.1. □

In our prototype we have implemented a relational algebra operating on sets of ground rules as a set of Java classes. A compiler constructs the expression $\bar{E}_P$ for a given program P according to Definition 8.1.5. Known optimizations such as index joins, differential fixpoint iteration [BR86b, GKB87], SCC-wise fixpoint iteration etc. are applied. A visualization of a sample relational expression is shown in the snapshot of the graphical user interface in Figure 8.1. In the center there is the original program, together with a query, and a desired transformation strategy. At the bottom right side the magic transformed program is displayed, and on the left side a tree view of the relational expression to compute the grounded version of the magic set transformed program is shown. Finally, under the menu bar, there is a sequence of buttons allowing to apply transformations manually, using the notation of the regular strategy expressions.

Another optimization is to make the projection in the expression $\bar{E}_{A \leftarrow \mathcal{B}}$ more intelligent.

Definition 8.1.7 (Improved Projection) Let $\bar{E}_{A \leftarrow \mathcal{B}}$ the relational expression for the rule $A \leftarrow \mathcal{B}$ as given in Definition 8.1.5. We modify the projection as follows:

- For a positive body literal $B \in \mathcal{B}$, omit the body literal B in the projection result, if a *fact* unifiable with B is found in the input set R. By this optimization we apply *success* already in the grounding phase.
- For a negative body literal $B = \mathbf{not}\, A \in \mathcal{B}$, if the input set R is already fixed,
  - produce no result, if a fact A is contained in R,
  - omit the body literal B in the projected result, if there is no rule with head A in R,
  - otherwise, leave the negative body literal in the rule body of the result of the projection.

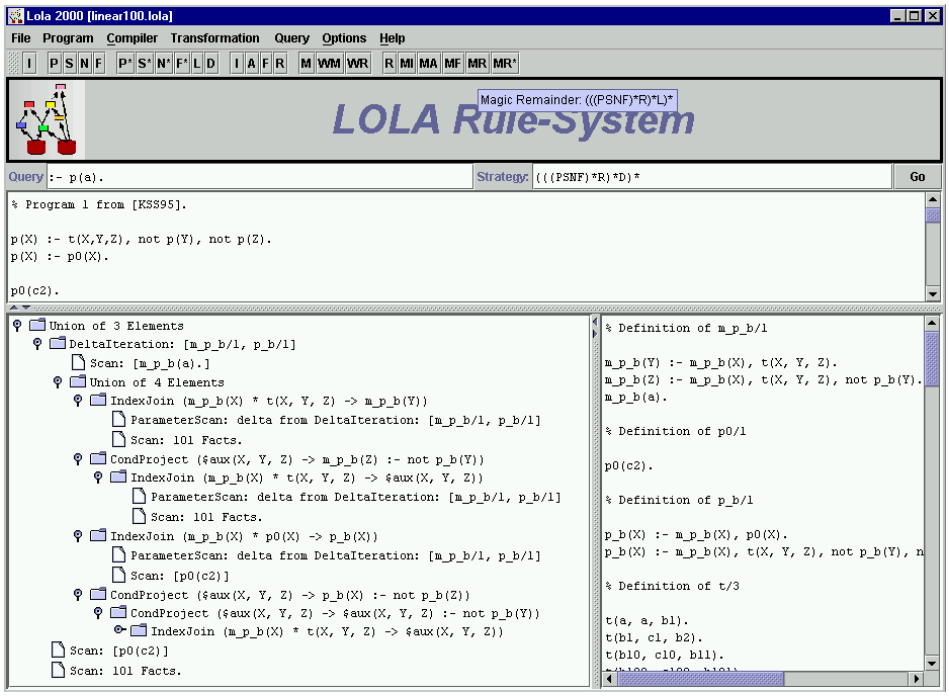


Figure 8.1: Visualization of Relational Expression

By this optimization we apply *negative reduction* or *positive reduction* where possible.

- If the set R is not yet complete, always leave the negative body literal in the result.

□

Note, that for the treatment of a negative body literal a semi-antijoin can be used, that produces a result only if there is no join partner found.

Remark 8.1.8 (Model for Definite or Stratified Programs) Consider the relational expression $\bar{E}_P$ of 8.1.5 modified by Definition 8.1.7 for a range-restricted definite program P. Then only facts are derived, since all body literals are omitted from the projection result. In this case, the set of facts coincides with the set of atoms computed by the expression E_P of Definition 8.1.3. As a consequence, the well-founded model of P is exactly the least fixpoint of $\bar{E}_P$.

If for a stratified program, i.e., a program with no predicate depending negatively on itself, each SCC is evaluated by a separate relational expression, such that the result for the lower SCCs is already complete, the truth value of each negative body literal

is known and again only facts are derived. This computation coincides with the computation of the perfect model for stratified programs [Prz88], which in turn coincides with the well-founded model of P. □

With this optimization we can compute the well-founded model for definite or stratified programs efficiently using relational operations and optimizations known from relational databases. Conditional facts are derived only for predicates depending negatively on themselves, and for predicates depending on them. If conditional facts are produced, a transformation phase is to be appended that computes the remainder of P.

8.2 Queue-Based Transformation

The four basic transformations *success*, *failure*, *positive reduction* and *negative reduction* are implemented based on an algorithm that is described in [DG84] as a linear time algorithm for the computation of the deductive closure of a definite ground program.

Definition 8.2.1 (Closure Algorithm for Definite Programs [DG84]) Let P be a definite program. For each rule $A \leftarrow \mathcal{B}$ in P there is counter holding the number of active body literals, initialized to the number of body literals in $\mathcal{B}$. For each atom there is a list, containing all rules having the atom as a body literal.

The algorithm uses a queue $posQ$ for true atoms, that are still to be processed. The queue $posQ$ is initialized with the atoms occurring as facts in P.

To process a true atom $B \in posQ$ means to visit each rule $A \leftarrow \mathcal{B}$ having $B \in \mathcal{B}$ as a body literal, and decrease its counter of active body literals. If the counter reaches zero, the head A is derived to be true. If this fact is new, it is added to $posQ$. □

All the counters and lists are stored in arrays, with the rule number or atom number being the index. Thus, after initializing the array, each operation described above can be executed in constant time. Since each body literal is processed at most one time, the execution time of this algorithm is in $O(size(P))$.

In [NS96] this algorithm is extended to compute the fitting reduction (cf. Section 3.3) of a normal ground program.

Definition 8.2.2 (Fitting Reduction [NS96]) Let P be a normal program. For each rule in P there is the counter for active body literals, and additionally a counter for inactive body literals, that is initialized to zero. For each atom there is a counter for the number of rules having this atom as its head.

The algorithm uses two queues, one queue $posQ$ for true atoms, and one queue $negQ$ for false atoms still to be processed. The queue $posQ$ is initialized with the facts in P. The queue $neqQ$ initially holds all atoms occurring in a rule body but not in a head in P.

To process a true atom $B \in posQ$ means:

- Visit each rule $A \leftarrow \mathcal{B}$ having $B \in \mathcal{B}$ as a *positive* body literal, and decrease its counter of active body literals. If the rule is still active and the counter reaches zero, the head A is derived to be true. If this fact is new, it is added to $posQ$.
- Visit each rule $A \leftarrow \mathcal{B}$ having $\mathbf{not}\ B \in \mathcal{B}$ as a *negative* body literal, and increase its counter of inactivate body literals. If the counter steps from zero to one, the rule is inactivated. Then the rule counter for its head A is decremented. If this counter reaches zero, the atom A is added to $negQ$.

To process a false atom $B \in negQ$ means:

- Visit each rule $A \leftarrow \mathcal{B}$ having $\mathbf{not}\ B \in \mathcal{B}$ as a *negative* body literal, and decrease its counter of active body literals. If the rule is still active and the counter reaches zero, the head A is derived to be true. If this fact is new, it is added to $posQ$.
- Visit each rule $A \leftarrow \mathcal{B}$ having $B \in \mathcal{B}$ as a *positive* body literal, and increase its counter of inactivate body literals. If the counter steps from zero to one, the rule inactivated. Then the rule counter for its head A is decremented. If this counter reaches zero, the atom A is added to $negQ$.

□

Theorem 8.2.3 (Correctness of Fitting Reduction [NS96]) Let P be a normal ground program. The algorithm in Definition 8.2.2 computes the set $\mathrm{lfp}(\Phi_P)$ in time $O(size(P))$. All atoms that have been in $posQ$ are true in $\mathrm{lfp}(\Phi_P)$, and all atoms that (do not occur in P or) have been in $newQ$ during the computation are false in $\mathrm{lfp}(\Phi_P)$.
□

In our implementation we have extended the algorithm of Definition 8.2.2 slightly. We use four queues, one for each basic transformation. Thus, we can apply the transformations separately and are more flexible to implement the computation according to strategy expressions.

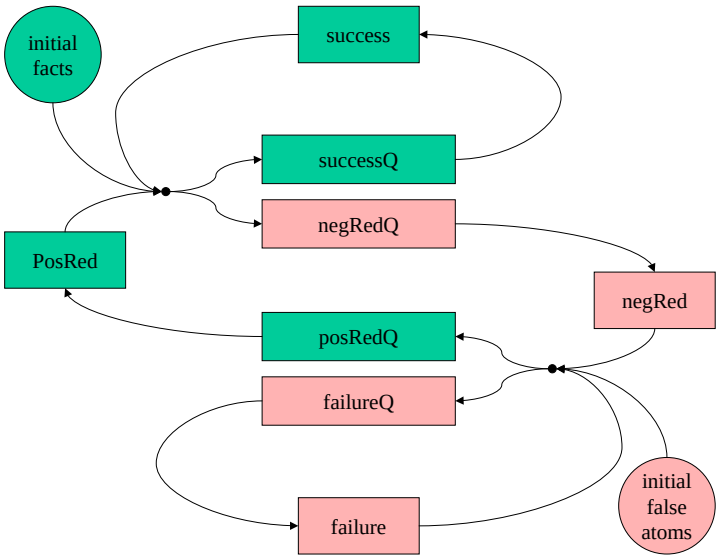


Figure 8.2: Usage of Queues

Definition 8.2.4 (Usage of Queues) The usage of queues in our implementation can be seen in Figure 8.2. The initial true atoms are given by the facts in P. They are copied to both $successQ$ and $negRedQ$. Atoms occurring in body literals of P but in no heads, are initially false and copied to both $posRedQ$ and $failureQ$.

If atoms from $successQ$ are used for applying *success*, it is possible that new true atoms are derived, that are again copied to $successQ$ and $negRedQ$. If atoms from $negRedQ$ are used to apply *negative reduction*, rules are deleted and possibly new false atoms are derived. New false atoms are copied to $posRedQ$ and $failureQ$. False atoms from $failureQ$ or $posRedQ$ are processed accordingly.

The atoms in the queues can be processed in any order. This allows the implementation of different transformation strategies. □

The data structure to represent the transformed program can be seen in Figure 8.3.

Definition 8.2.5 (Data Structure) Each different atom $A \in At(P)$ occurring in P is represented by a data structure (an instance of a Java class) visualized by the vertical bar on the left side of Figure 8.3. For each atom the data structure stores

- the list $posLitList$ of references to all rules having the atom A as a positive body literal,
- the list $negLitList$ of references to all rules having the literal $\mathbf{not}\ A$ as a negative body literal,

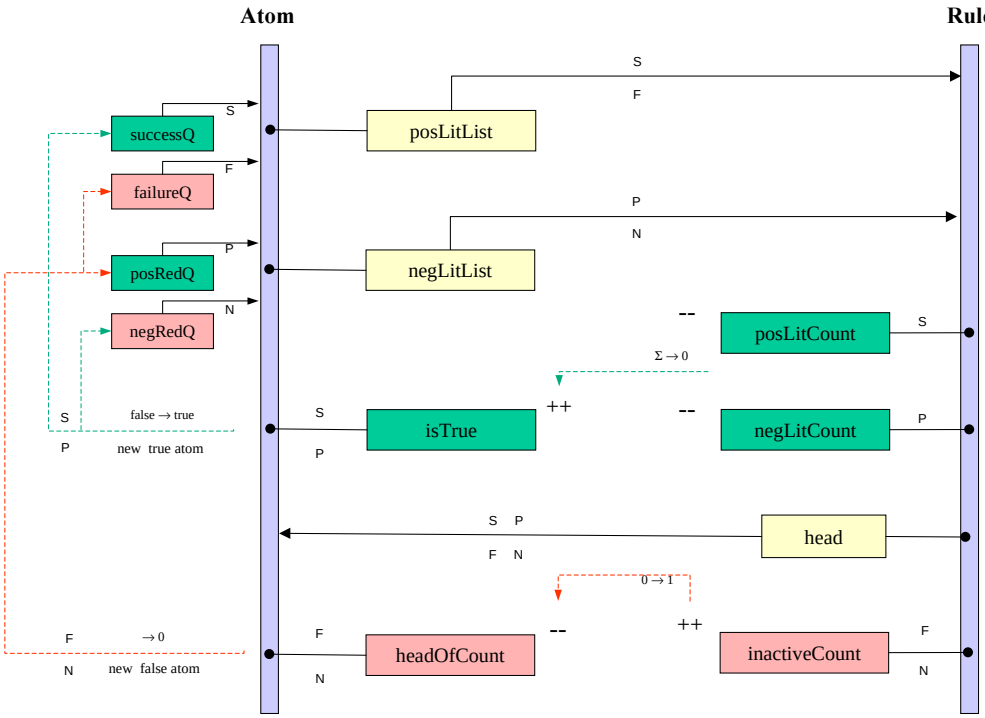


Figure 8.3: Queue-Based Ground Transformation

- the boolean flag $isTrue$ that is set for true atoms,
- and the counter $headOfCount$ holding the number of rules with head atom A.

Each rule of $A \leftarrow \mathcal{B}$ if P is represented by a data structure that is depicted by the vertical bar on the right side of Figure 8.3. For each rule the data structure stores

- a counter $posLitCount$, initialized with the number of positive body literals in $\mathcal{B}$,
- a counter $negLitCount$, initialized with the number of negative body literals in $\mathcal{B}$,
- a reference $head$ to the data structure for the head atom A,
- a counter $inactiveCount$, initialized to zero.

The queues for the four basic transformations are implemented as lists of references to atoms. □

The transformation proceeds as follows.

Definition 8.2.6 (Transformation Process) Let P be a normal ground program. First, the data structure of Definition 8.2.5 is initialized such that for each different atom occurring in P and for each rule of P a corresponding object is created. The references and counters are initialized as defined in Definition 8.2.5. References to atoms with the $isTrue$-flag set are copied to $successQ$ and $negRedQ$. References to atoms with $headOfCount = 0$ are copied to $failureQ$ and $posRedQ$.

Now a strategy component can select an atom B from one of the four queues. If an atom B from $successQ$ or $failureQ$ is selected, the following steps are performed:

1. The atom B is removed from the queue.
2. For each rule $r \equiv A \leftarrow \mathcal{B}$ in $B.posLitList$ do:
   (a) For *success*: decrement the counter $r.posLitCount$.
   (b) For *failure*: increment the counter $r.inactiveCount$.

If an atom B from $posRedQ$ or $negRedQ$ is selected, the following steps are performed:

1. The atom B is removed from the queue.
2. For each rule $r \equiv A \leftarrow \mathcal{B}$ in $B.negLitList$ do:
   (a) For *positive reduction*: decrement the counter $r.negLitCount$.
   (b) For *negative reduction*: increment the counter $r.inactiveCount$.

If for a rule $r \equiv A \leftarrow \mathcal{B}$ the conditions $r.posLitCount + r.negLitCount = 0$ and $r.inactiveCount = 0$ become true during the application of *success* or *positive reduction*, then:

1. Get the head atom $A \equiv r.head$ of the rule.
2. If $A.isTrue$ is not set, then:
   (a) Set $A.isTrue$.
   (b) Copy a reference to A to $successQ$ and $negRedQ$.

If for one rule $r \equiv A \leftarrow \mathcal{B}$ the counter $r.inactiveCount$ changes from 0 to 1 during the application of *failure* or *negative reduction*, then:

1. Get the head atom $A \equiv r.head$ of the rule.

2. Decrease the counter $A.headOfCount$.
3. If $A.headOfCount$ reaches the value 0, then:
   (a) Copy a reference to A to $failureQ$ and $PosRedQ$.

This process is continued until all four queues are empty. □

Lemma 8.2.7 (Termination) Let P be a ground program. Then the queue-based transformation as described in Definition 8.2.6 terminates.

Proof: It is guaranteed that each atom is processed at most once by one of the four transformations, since true atoms are only added to the queues if the flag $isTrue$ is set from false to true, and false atoms are only added to the queues, when the counter $headOfCount$ steps from 1 to 0. Each application of a transformation terminates since it essentially performs a loop over a list of body literals. □

We can even show that the execution terminates in linear time.

Lemma 8.2.8 (Time Complexity of Base Transformations) The overall execution time for the application of the four base transformations is in $O(size(P))$.

Proof: Each body literal is processed at most one time during the transformation process. There are at most $size(P)$ transformations applicable. Each transformation is applicable in constant time, since only atomic operations to counters or references are to be done. □

Remark 8.2.9 (Costs of Initialization) As in [DG84] we assume that the initialization of the data structure can be done in time $O(size(P))$. The most expensive part is the requirement that an atom that occurs more than once in the program is to be represented by the same data object. In [DG84] it is assumed that the parser replaces atoms by numbers identifying several occurrences of the same atom, e.g., be using symbol tables. However, we use hash tables to check whether an atom has occurred before. Thus we can do the check in constant time as long as the size of the program is in a range where the hash function does work properly. □

Lemma 8.2.10 (Correctness of Queue Based Transformation) Let P be a normal program. Then the queue based transformation described in Definition 8.2.6 computes the fitting reduction $\mathrm{lfp}(\Phi_P)$ of P.

Proof: It is easy to see, that the transformations applied during the queue based transformation correspond exactly to the transformations as they are defined in Chapter 4. It is still to show, that each applicable transformation is actually applied. This is the case, since it is guaranteed that all true facts are copied to the queues $successQ$ and $negRedQ$ and that all false atoms are copied to $failureQ$ and $posRedQ$, already in the initialization phase or later during the transformation. Together with Theorem 4.3.12 the result follows. □

Remark 8.2.11 (Transformed Program) Note, that with the algorithm of Definitions 8.2.5 and 8.2.6 the current state of the transformed program is not stored directly. To this end, we added an extra field to the rule data structure keeping a transformed version of the corresponding rule. Thus it is easy to print the current state of the program at any time during the transformation. Especially the remainder can be read off directly from the final state. Otherwise, it has to be reconstructed using the truth values of the atoms. □

8.3 Loop Detection

For the *loop detection* the set of atoms that are still possible is to be computed according to Lemma 4.3.15. To this end we use all rules that have not been deleted until now, i.e., all rules having the counter $inactiveCount = 0$.

Definition 8.3.1 (Implementation of Loop Detection) We extend the data structure for atoms by a boolean flag $isStillPossible$. The data structure for rules gets an additional counter $posLitCount2$. Further we create another queue $possibleQ$ that is initially empty.

The loop detection proceeds as follows. For all atoms A do:

1. Set the flag $A.isStillPossible$ to the same value as $A.isTrue$.
2. If $A.isTrue$ is set, add the atom A to the queue $possibleQ$.

For all rules $r \equiv A \leftarrow \mathcal{B}$ do:

1. Set $r.posLitCount2$ to the number of positive body literals in $\mathcal{B}$.

As long as $possibleQ$ is not empty, do:

1. Remove an atom A from $possibleQ$.
2. For all rules $r \equiv A \leftarrow \mathcal{B}$ in $A.posLitList$ do:
   (a) Decrement the counter $r.posLitCount2$.
   (b) If $r.poslitCount2$ has reached the value 0, and $r.inactiveCount = 0$ holds do:
      i. Let $A \equiv r.head$ be the head of the rule.
      ii. If $A.isStillPossible$ is not set, then set the flag and add the atom A to $possibleQ$.

Now all atoms having the flag $isStillPossible$ not set are unfounded and can be deleted by *loop detection*. For all atoms A do: If $A.isStillPossible$ is not set, then add A to $failureQ$ and $posRedQ$. Finally, process all atoms in $failureQ$ as described in Definition 8.2.6 to inactivate all rules that are to be deleted by *loop detection*. □

Lemma 8.3.2 (Time Complexity of Loop Detection) The execution time for each application of *loop detection* is in $O(size(P))$.

Proof: The most expensive part is the closure computation for possible facts. This is based on the closure algorithm of Definition 8.2.1 which has an execution time in $O(size(P))$. □

Remark 8.3.3 (Optimizations) Several optimizations for the implementation of the *loop detection* have been proposed, e.g. in [NS96]. These optimizations take into account, for instance, the set of rules that have been deleted since the previous application of the *loop detection*, or the SCC structure of the original ground program, etc. We have implemented several of these optimizations in our prototype yielding a linear time complexity for many programs. However, none of these optimizations reduces the worst case time complexity. Thus we omit their details here. □

8.4 Magic Reduction

The magic reduction can be implemented efficiently as part of the queue-based transformation.

Definition 8.4.1 (Implementation of Restricted Magic Reduction) We create a new queue $magicQ$, that keeps a list of references to all rules that are reducible by *restricted magic reduction*.

Initially, $magicQ$ contains all rules $A \leftarrow \{B\}$ such that A has no magic predicate, but B has.

Whenever *success* or *positive reduction* is applied to a rule, check wether all body literals except the magic filter, if it exists, are reduced. If this is the case, add the rule to $magicQ$.

The $magicQ$ contains all rules, to that *restricted magic reduction* may be applied. To apply the reduction, do the following:

1. Remove one rule $r \equiv A \leftarrow \mathcal{B}$ from $magicQ$.
2. If the condition for *restricted magic reduction* does still hold for r, then

(a) Decrement the counter $r.posLitCount$. By definition, it reaches the value 0.

(b) Let $A = r.head$ be the head of r.

(c) If $A.isTrue$ is not set, then:

i. Set $A.isTrue$.

ii. Add A to $successQ$ and $negRedQ$.

□

Lemma 8.4.2 (Time Complexity of Restricted Magic Reduction) The overall time for the execution of the *restricted magic reduction* is in $O(size(P))$.

Proof: Every rule is checked for the condition for *restricted magic reduction* at most one time for each body literal. Further, each rule is added to $magicQ$ and removed from $magicQ$ and checked again at most one time. Thus, at most $size(P)$ checks are performed. □

Remark 8.4.3 (Full Magic Reduction) The full *magic reduction* transformation should be applied only once to each program. Thus it can be implemented trivially by a loop over all rules. □

Chapter 9

Non-Ground Transformation

Since the ground transformations defined in Chapter 4 are applicable only to ground programs, we always had to prepend a grounding phase as proposed in Chapter 5. This is adequate for the computation of the complete well-founded model of a program, since in this case our grounding algorithm considers the same set of relevant ground instances as any bottom-up fixpoint iteration.

To answer a query w.r.t. a given normal program, we introduced the *magic reduction* transformation to evaluate magic set transformed programs correctly. However, as we have seen in Example 7.3.23 a large set of rule instances may be generated in the grounding phase (or in the first computation of possible facts in any alternating fixpoint based method) that is possibly discarded immediately in the next step.

Top-down approaches as the SLD resolution [Llo87] for definite programs or the SLG resolution [CW93, CSW95, SSW00] for normal programs apply activation and instantiation on demand. Instead of computing an initial set of ground instances, rules are instantiated not before they are needed.

In this chapter we extend our transformation approach such that it is applicable to non-ground rules. Instead of prepending a grounding phase, we define new transformations for the activation of goals and instantiation of rules. By this extension a goal-directed evaluation is possible without generating unnecessary ground instances. Further, magic set like techniques are not needed and thus their problems in the context of the well-founded semantics are avoided.

9.1 Transformation States

The transformation system presented in this chapter operates on transformation states rather than plain ground programs as the ground transformation of Chapter 4.

Definition 9.1.1 (Non-Ground Transformation State) Let G be a set of atoms. Let R and U be sets of rule instances. Then

$$S \ := \ (G, R, U)$$

is called a *(non-ground) transformation state*. G is called the set of *activated goals*. R is called the set of *active rule instances*. U is called the set of *used rule instances*. □

The set G is the set of queries to be answered. To keep the presentation simple, we will assume that initially one single query atom Q is given. More queries are added later during the transformation. However, it is also possible to start with a larger set G of queries. For example, to compute the complete model of P all rules of P can be activated simultaneously by adding a query atom with pairwise different variables for each predicate defined in P. If an (not necessarily ground) atom B is contained in G, all instances of B are considered as activated as well.

Definition 9.1.2 (Active Goals) Let $S = (G, R, U)$ be a transformation state. An atom A is *activated in S*, iff there is an atom $B \in G$ such that A is an instance of B. □

The set R is what is actually transformed: the set of rule instances relevant to compute the answers to the queries in G. R is initialized with those instances of the rules of P the head of which is unifiable with Q.

Definition 9.1.3 (Relevant Rule Instances) Let P be a normal program. Let Q be an atom. The set of *relevant rule instances of P w.r.t. Q* is defined by

$$instances(P, Q) \ := \ \{(A \leftarrow \mathcal{B})\sigma \mid \begin{array}{l} (A \leftarrow \mathcal{B}) \in P \text{ and} \\ A \text{ and } Q \text{ are unifiable and} \\ \sigma = mgu(A, Q)\}. \end{array}$$

□

Remark 9.1.4 (Renaming Variables) We assume that for testing unifiability always variants of rules are used with variables renamed. □

Example 9.1.5 (Relevant Rule Instances) Consider the following program P that is also used as an example in [CSW95].

$$\begin{array}{lcl} q(X) & \leftarrow & p(X). \\ q(a). & & \\ p(X) & \leftarrow & q(X). \\ p(X) & \leftarrow & \mathbf{not}\, r. \\ r & \leftarrow & \mathbf{not}\, s. \\ s & \leftarrow & \mathbf{not}\, r. \end{array}$$

Let the initial query be $Q = q(X)$. Then the relevant rule instances are

$$instances(P, Q) = \{q(X) \leftarrow p(X).\ q(a).\}.$$

For this query the rules from P defining q are used. For the query $q(a)$ we would get the following instances.

$$instances(P, q(a)) = \{q(a) \leftarrow p(a).\ q(a).\}.$$

For the query $q(b)$ we get only one rule instance

$$instances(P, q(b)) = \{q(b) \leftarrow p(b).\}.$$

□

The set R changes during transformation. To keep track of the transformation process and enable us to determine its progress the set U stores the rule instances that have added new information to R.

Definition 9.1.6 (Used and New Rule Instances) Let $S = (G, R, U)$ be a transformation state. A rule instance $r \equiv A \leftarrow \mathcal{B}$ is *used in* S if U contains a variant of r. Otherwise r is *new in* S. □

For a normal program P and a query atom Q, the initial transformation state contains Q as the only activated atom and the corresponding set of relevant rule instances as the initial rule set.

Definition 9.1.7 (Initial Transformation State) Let P be a normal program. Let Q be an atom. Then

$$S_0 \ := \ (\{Q\}, instances(P, Q), instances(P, Q))$$

is called the *initial transformation state* of P w.r.t. Q. □

Example 9.1.8 (Initial Transformation State) Consider again the program P from Example 9.1.5 and the query $Q = q(X)$. In the initial transformation state

$$S_0 = (G_0, R_0, U_0)$$

of P w.r.t. Q we have $G_0 = \{q(X)\}$ and

$$R_0 = U_0 = \{q(X) \leftarrow p(X).\ q(a).\}.$$

□

With each transformation state S we associate a partial interpretation $I(S)$. As in the definition of known literals (cf. Definition 4.1.4) a (possibly non-ground) fact A in the rule set R represents positive information, i.e., all instances of A are considered to be true in S. In Example 9.1.8 the atom $q(a)$ is expected to be true in the well-founded model of P.

For negative information the analogous definition would consider an atom A to be false if it is not unifiable with any head in R. In Example 9.1.8 we could conclude that $p(a)$ is false in W_P^* because there is no head in R unifiable with $p(a)$. But this is not true, because in the initial transformation state S_0 the rules defining p have not been considered. Thus no statement about truth values of p-atoms should be made. In general, we can get information from a transformation state S about the truth value of an atom A only if A is activated in S. An atom A is considered to be false in S if A is activated in S and there is still no rule head in R that is unifiable with A.

Definition 9.1.9 (Interpretation of a State) Let $S = (G, R, U)$ be a transformation state. An atom A is called *true in* S, if A is activated in S and there is a fact B in R such that A is an instance of B. An atom is called *false in* S if A is activated in S and there is no rule in R with a head that is unifiable with A. A negative literal $L = \mathbf{not}\, A$ is called true (false) in S, if A is false (true) in S.

The set $true(S)$ of ground atoms true in S and the set $false(S)$ of ground atoms false in S are defined as follows:

$$\begin{array}{ll} true(S) & := \{A \in BASE(P) \mid A \text{ is true in } S\}, \\ false(S) & := \{A \in BASE(P) \mid A \text{ is false in } S\}. \end{array}$$

The *interpretation of* S, that contains all ground literals that are true in S is defined as follows:

$$I(S) := true(S) \cup \sim false(S).$$

□

Remark 9.1.10 (Non-Ground Facts) Let A be a non-ground atom that is activated in a transformation state S for a program P and that is contained as a fact in the rule set R of S. According to Definition 9.1.9 A is true in S. This means that all ground instances of A (w.r.t. the Herbrand base of P) are expected to be true in W_P^*. Accordingly, from Definition 9.1.9 it follows that $ground(A) \subseteq true(S)$ holds. Thus the non-ground true fact A can be considered as an abbreviation meaning that all atoms in $ground(A)$ are true in S. The analogous statement holds for false facts. If a non-ground atom A is false in S, then $ground(A) \subseteq false(S)$ holds. □

Example 9.1.11 (Initial Interpretation) Consider the initial transformation state S_0 from Example 9.1.8. Its interpretation is given by

$$I(S_0) = \{q(a)\}.$$

This means that $q(a)$ is true in S. The truth value of all other instances of $q(X)$ and all p-atoms and r and s are undefined in this interpretation. □

9.2 Well-Founded Transformation

In this section we will define a set of transformations that are to be applied to the initial transformation state. As in the ground case in Chapter 4, transformations should be applied until a normalform is reached, i.e., no transformation is applicable any more. But the interpretation $I(S)$ of the final transformation state S is not required to be equal to the well-founded model W_P^* of the given program P, but it is sufficient that $I(S)$ and W_P^* agree on the given query Q.

In Example 9.1.5 and 9.1.8 we have seen, that the initial set of rules does contain all relevant rules defining the query predicate q. However, the rules for p, r, and s that are also relevant for the computation of the result have not been considered until now. The first non-ground transformation activates new atoms and adds new rule instances to R.

Definition 9.2.1 (Activation) Let P be a normal program. Let $S_1 = (G_1, R_1, U_1)$ and $S_2 = (G_2, R_2, U_2)$ be transformation states of P. S_2 *results from* S_1 *by activation*, $S_1 \rightsquigarrow_A S_2$, iff there is a rule instance $A \leftarrow \mathcal{B}$ in $\mathcal{R}_1$ and a positive or negative literal $(\mathbf{not})B \in \mathcal{B}$ such that B is not activated in S_1 and

$$\begin{array}{lcll} G_2 & = & G_1 & \cup \quad \{B\} \\ R_2 & = & R_1 & \cup \quad instances(P, B) \\ U_2 & = & U_1 & \cup \quad instances(P, B) \end{array}$$

holds. □

Remark 9.2.2 (New Activation) Note, that by the transformation $\rightsquigarrow_A$ we can activate only those atoms that are not activated yet. This condition is important for termination. Without this condition it would be possible to activate the same atom again and again without generating new information. This point will be discussed in Remark 11.3.14. □

Example 9.2.3 (Activation) The only body literal that can be activated in the initial transformation state S_0 of Example 9.1.8 is $p(X)$. Thus in a new transformation state we add $p(X)$ to G and the two rule instances $p(X) \leftarrow \mathbf{not}\, r$ and $p(X) \leftarrow q(X)$ to both R and U. Then we can activate r and add the rule $r \leftarrow \mathbf{not}\, s$ and finally activate s and add the rule $s \leftarrow \mathbf{not}\, r$. The rule set R of the new transformation state S contains all rules from P. But until now there are no new answers in $I(S)$. □

Remark 9.2.4 (Activation and Resolution) The *activation* transformation of Definition 9.2.1 corresponds to selecting a literal $(\mathbf{not})B$ from a rule body and finding instances $instances(P, B)$ of all rules in P whose head is unifiable with B. The actual resolution, i.e., the unfolding of the rule instances for B and the instantiation w.r.t. the most general unifiers is not done in this step. As in the ground case in Chapter 4 we avoid full unfolding and allow only special cases like *success* and *failure*. For the instantiation that is part of the resolution we will introduce a new transformation. □

Remark 9.2.5 (Parallel Activation) Note, that Definition 9.2.1 allows to activate more than one body literal of a rule instance at a time. This is an extension of the prominent resolution approaches like SLD [Llo87] or SLG [CW93, CSW95, SSW00] that allow the selection of only one literal per rule instance. We will show in Example 11.3.19 that this can lead to more efficient search strategies. □

The following four transformations are generalizations of the corresponding ground transformations. They remove obviously true body literals from rule instances or delete rule instances containing body literals that are obviously false in the current transformation state.

Definition 9.2.6 (Success) Let P be a normal program. Let $S_1 = (G_1, R_1, U_1)$ and $S_2 = (G_2, R_2, U_2)$ be transformation states of P. *S_2 results from S_1 by success,* $S_1 \rightsquigarrow_S S_2$, iff $G_1 = G_2$ and $U_1 = U_2$ and there is a rule $A \leftarrow \mathcal{B}$ in R_1 and a positive literal $B \in \mathcal{B}$ such that B is true in S_1 and

$$R_2 = R_1 \setminus \{A \leftarrow \mathcal{B}\} \cup \{A \leftarrow (\mathcal{B} \setminus \{B\})\}.$$

□

Definition 9.2.7 (Failure) Let P be a normal program. Let $S_1 = (G_1, R_1, U_1)$ and $S_2 = (G_2, R_2, U_2)$ be transformation states of P. *S_2 results from S_1 by failure,* $S_1 \rightsquigarrow_F S_2$, iff $G_1 = G_2$ and $U_1 = U_2$ and there is a rule $A \leftarrow \mathcal{B}$ in R_1 and a positive literal $B \in \mathcal{B}$ such that B is false in S_1 and

$$R_2 = R_1 \setminus \{A \leftarrow \mathcal{B}\}.$$

□

Definition 9.2.8 (Positive Reduction) Let P be a normal program. Let $S_1 = (G_1, R_1, U_1)$ and $S_2 = (G_2, R_2, U_2)$ be transformation states of P. S_2 *results from* S_1 *by positive reduction,* $S_1 \rightsquigarrow_P S_2$, iff $G_1 = G_2$ and $U_1 = U_2$ and there is a rule $A \leftarrow \mathcal{B}$ in R_1 and a negative literal $\mathbf{not}\, B \in \mathcal{B}$ such that B is false in S_1 and

$$R_2 = R_1 \setminus \{A \leftarrow \mathcal{B}\} \cup \{A \leftarrow (\mathcal{B} \setminus \{\mathbf{not}\, B\})\}.$$

□

Definition 9.2.9 (Negative Reduction) Let P be a normal program. Let $S_1 = (G_1, R_1, U_1)$ and $S_2 = (G_2, R_2, U_2)$ be transformation states of P. S_2 *results from* S_1 *by negative reduction,* $S_1 \rightsquigarrow_N S_2$, iff $G_1 = G_2$ and $U_1 = U_2$ and there is a rule $A \leftarrow \mathcal{B}$ in R_1 and a negative literal $\mathbf{not}\, B \in \mathcal{B}$ such that B is true in S_1 and

$$R_2 = R_1 \setminus \{A \leftarrow \mathcal{B}\}.$$

□

In Definitions 9.2.6 through 9.2.9 of the basic transformations, an activated body literal of a rule instance is considered to be true or false if *all* its ground instances are true or false in S (cf. Remark 9.1.10).

Example 9.2.10 Consider the current transformation state S of Example 9.2.3. The only atom with a defined truth value in $I(S)$ is the fact $q(a)$. Unfortunately, this atom does not occur in any rule body in R. Thus, none of the four basic transformations is applicable. However, we would like to apply the rule $p(X) \leftarrow q(X)$ or one of its instances to derive the fact $p(a)$. □

The *instantiation* transformation to be defined next allows to focus on single instances of positive body literals rather than having to take into account the entire set of ground instances.

Definition 9.2.11 (Instantiation) Let P be a normal program. Let $S_1 = (G_1, R_1, U_1)$ and $S_2 = (G_2, R_2, U_2)$ be transformation states of P. S_2 *results from* S_1 *by instantiation,* $S_1 \rightsquigarrow_I S_2$, iff $G_1 = G_2$ and there is a rule instance $A \leftarrow \mathcal{B}$ in R_1 and an atom $B \in \mathcal{B}$ such that B is activated in S_1 and there is a rule instance $B' \leftarrow \mathcal{C}'$ in R_1 such that B and B' are unifiable and $\sigma = mgu(B, B')$ and $(A \leftarrow \mathcal{B})\sigma$ is new in S_1 and

$$\begin{array}{rcll} R_2 & = & R_1 & \cup \quad \{(A \leftarrow \mathcal{B})\sigma\} \\ U_2 & = & U_1 & \cup \quad \{(A \leftarrow \mathcal{B})\sigma\} \end{array}$$

holds. □

Example 9.2.12 (Instantiation and Success) In the transformation states of Example 9.2.3, we can apply *instantiation* and add the instance $p(a) \leftarrow q(a)$ of $p(X) \leftarrow q(X)$ and then the instance $q(a) \leftarrow p(a)$ of $q(X) \leftarrow p(X)$ to R and U. A subsequent application of *success* removes the body atoms $q(a)$ and then $p(a)$ from the new instances in R. In the resulting transformation state S we have

$$
\begin{array}{lll}
G = \{ & q(X),\ p(X),\ r,\ s & \} \\
R = \{ & q(a).\ p(a).\ q(X) \leftarrow p(X).\ p(X) \leftarrow q(X). & \\
 & p(X) \leftarrow \mathbf{not}\, r.\ r \leftarrow \mathbf{not}\, s.\ s \leftarrow \mathbf{not}\, r. & \} \\
U = \{ & q(a) \leftarrow p(a).\ p(a) \leftarrow q(a).\ q(X) \leftarrow p(X).\ p(X) \leftarrow q(X). & \\
 & q(a).\ p(X) \leftarrow \mathbf{not}\, r.\ r \leftarrow \mathbf{not}\, s.\ s \leftarrow \mathbf{not}\, r. & \}
\end{array}
$$

The corresponding interpretation $I(S)$ contains the atoms $q(a)$ and $p(a)$. □

If all relevant instances of a rule have been generated, the rule itself should be deleted.

Example 9.2.13 Consider the following program.

$$
\begin{array}{lll}
p(X) & \leftarrow & q(X). \\
q(a). & & \\
r(b). & &
\end{array}
$$

By applying *instantiation* we get the following program.

$$
\begin{array}{lll}
p(X) & \leftarrow & q(X). \\
p(a) & \leftarrow & q(a). \\
q(a). & & \\
r(b). & &
\end{array}
$$

The second rule is also added to the set U of used rule instances. By applying *success* we can derive $p(a)$ and get the following program.

$$
\begin{array}{lll}
p(X) & \leftarrow & q(X). \\
p(a) & & \\
q(a). & & \\
r(b). & &
\end{array}
$$

The interpretation of this final transformation state is

$$I(S) = \{p(a), q(a), \mathbf{not}\, q(b), \mathbf{not}\, r(a), r(b)\}.$$

The truth values of the q-atoms and r-atoms are determined correctly. However, the atom $p(b)$ is undefined in the interpretation although it is false in the well-founded model. This is due to the first rule $p(X) \leftarrow q(X)$. We know that $q(X)$ is false except for the $X = a$. Thus we should conclude that $p(X)$ is false except for $X = a$. □

In the next definition we identify those rules that have been instantiated completely, i.e., all relevant instances have been taken into account.

Definition 9.2.14 (Complete Instantiation) Let $S = (G, R, U)$ be a transformation state. Let $A \leftarrow \mathcal{B}$ be a rule instance in R, and let $B \in \mathcal{B}$ be an atom that is activated in S. $A \leftarrow \mathcal{B}$ is *completely instantiated through B w.r.t. R in S*, iff for every rule $B' \leftarrow \mathcal{C}'$ in R it holds that: if B and B' are unifiable with $\sigma = mgu(B, B')$ then $(A \leftarrow \mathcal{B})\sigma$ is used in S and $(A \leftarrow \mathcal{B})\sigma$ is not a variant of $A \leftarrow \mathcal{B}$. □

Remark 9.2.15 (Complete Instantiation) A rule instance $A \leftarrow \mathcal{B}$ in R is said to be completely instantiated w.r.t. R in S, if there is at least *one* positive body literal $B \in \mathcal{B}$ such that the condition of Definition 9.2.14 is satisfied. It is not necessary that the rule is completely evaluated through *all* its body atoms. Further, only positive body literals are considered for completion. This corresponds to Definition 9.2.11 in which *instantiation* is defined to be applicable w.r.t. positive body literals only. □

Example 9.2.16 Consider the rule set R of the final transformation state S from Example 9.2.13. Its first rule satisfies the condition of Definition 9.2.14. The rule $p(X) \leftarrow q(X)$ is completely instantiated through the body literal $q(X)$ w.r.t. R in S. $q(X)$ is unifiable with the rule heads $q(a)$ and $q(b)$ and both instances $p(a) \leftarrow q(a)$ and $p(b) \leftarrow q(b)$ are used in S. Finally neither rule instance is a variant of the rule itself. □

If a rule is completely instantiated, all relevant rule instances have been used, thus the rule can be deleted.

Definition 9.2.17 (Completion) Let P be a normal program. Let $S_1 = (G_1, R_1, U_1)$ and $S_2 = (G_2, R_2, U_2)$ be transformation states of P. *S_2 results from S_1 by completion*, $S_1 \rightsquigarrow_C S_2$, iff $G_1 = G_2$ and $U_1 = U_2$ and there is a rule instance $A \leftarrow \mathcal{B}$ in R_1 and an atom $B \in \mathcal{B}$ such that $A \leftarrow \mathcal{B}$ is completely instantiated through B w.r.t. R_1 in S_1 and

$$R_2 = R_1 \backslash \{A \leftarrow \mathcal{B}\}.$$

□

Example 9.2.18 Consider again the final transformation state from Example 9.2.13 and 9.2.16. Since the first rule is completely instantiated we can delete it by one application of the *completion* transformation and get the following final set of rules.

$$p(a).$$
$$q(a).$$
$$r(b).$$

Obviously, now the interpretation of this state contains the literal $\mathbf{not}\, p(b)$ that was missing in Example 9.2.13. □

Remark 9.2.19 (Variant Condition) Note, that the rule $p(X) \leftarrow q(X)$ in Example 9.2.12 is not completely instantiated. The body literal $q(X)$ is unifiable with the head of the rule $q(X) \leftarrow q(X)$. The most general unifier is the empty substitution, thus the generated rule instance is $p(X) \leftarrow q(X)$ which is clearly a variant of itself.

The variant condition of Definition 9.2.14 ensures that a rule is completely instantiated only if all instances that can be generated are proper instances, i.e., they are no variants of the rule. Otherwise the rule itself is one of the relevant instances, thus it is still relevant and should not be deleted. □

Remark 9.2.20 (Completion subsumes Failure) If a rule $A \leftarrow \mathcal{B}$ can be deleted by *failure*, i.e., an atom $B \in \mathcal{B}$ is false in S, then $A \leftarrow \mathcal{B}$ is completely instantiated through B and can be deleted by *completion*. However, the applicability of *failure* is easier to check than the applicability of *completion*. Thus, the transformation *failure* is still relevant. □

The *completion* transformation is not applicable to rules whose activated body literal depends (directly or indirectly) on the rule head.

Example 9.2.21 (One Recursive Rule) Consider the following program P.

$$\begin{array}{lcl} p(X) & \leftarrow & p(X). \\ p(a). & & \end{array}$$

We can apply *instantiation* to generate the rule instance $p(a) \leftarrow p(a)$ and then apply *success* to derive (again) the fact $p(a)$. Then we would like to delete the first rule. However, the *completion* transformation is not applicable, because the rule $p(X) \leftarrow p(X)$ is not completely instantiated w.r.t. the body literal $p(X)$ and the rules in P. We have instantiated the rule through the fact $p(a)$, but the body literal $p(X)$ is still unifiable with the rule head $p(X)$. Thus the condition from Definition 9.2.17 is not satisfied. However, the rule is obviously a tautology and cannot produce any new answers. Thus it should be deleted by an appropriate transformation. □

Remark 9.2.22 (Extended Completion) We could define the following extension to the *completion* transformation. A rule $(A \leftarrow \mathcal{B}) \in R$ can be deleted by *extended completion* if the rule is completely instantiated through a positive body literal $B \in \mathcal{B}$ w.r.t. the set $R \backslash \{A \leftarrow \mathcal{B}\}$ containing all rules from R except the rule that should be deleted. However, this is only a special case of the next transformation. □

Example 9.2.23 (Mutually Recursive Rules) Consider the following program P.

$$\begin{array}{lcl} p(X) & \leftarrow & q(X). \\ q(X) & \leftarrow & p(X). \end{array}$$

The two rules depend on each other in a positive cyclic dependency. If the ground *loop detection* of Definition 4.3.14 was applicable to non-ground programs, we could consider the set $\{p(X), q(X)\}$ as an unfounded set and delete the two rules simultaneously. In this example the result would be correct. Now consider the following program P' which contains the additional fact $p(a)$.

$$\begin{array}{lcl} p(X) & \leftarrow & q(X). \\ q(X) & \leftarrow & p(X). \\ p(a). & & \end{array}$$

Obviously, we are not allowed two delete the two first rules in this example, since we need them to derive the fact $q(a)$ which is true in the well-founded model of this program. Now let us apply *instantiation* to generate the two rule instances $q(a) \leftarrow p(a)$ and $p(a) \leftarrow q(a)$. Then we can apply *success* to derive the facts $q(a)$ and $p(a)$. The resulting program P'' is

$$\begin{array}{lcl} p(X) & \leftarrow & q(X). \\ q(X) & \leftarrow & p(X). \\ p(a). & & \\ q(a). & & \end{array}$$

Now the two first rules are not needed any more, since all relevant instances have been used, and they should be deleted. □

The next transformation simultaneously deletes a set R' of rules if all relevant instances w.r.t. the remaining rules have been used.

Definition 9.2.24 (Loop Detection) Let P be a normal program. Let $S_1 = (G_1, R_1, U_1)$ and $S_2 = (G_2, R_2, U_2)$ be transformation states of P. S_2 *results from* S_1 *by loop detection,* $S_1 \rightsquigarrow_L S_2$, iff $G_1 = G_2$ and $U_1 = U_2$ and there is a non-empty set $R' \subseteq R_1$ such that for each rule instance $A \leftarrow \mathcal{B}$ in R' there is an activated atom $B \in \mathcal{B}$ such that $A \leftarrow \mathcal{B}$ is completely instantiated through B w.r.t. $R_1 \backslash R'$ in S_1, and $R_2 = R_1 \backslash R'$. □

Example 9.2.25 (Loop Detection) Consider the program P'' from Example 9.2.23. The rule set

$$R' = \{p(X) \leftarrow q(X).\ q(X) \leftarrow p(X).\}$$

satisfies the condition of Definition 9.2.24. Both rules are completely instantiated through their body literal w.r.t. the remaining rules since the instances $p(a) \leftarrow q(a)$ and $q(a) \leftarrow p(a)$ have been used. Thus the rules in R' may be deleted by one application of *loop detection* from the program P''. □

Remark 9.2.26 (Loop Detection subsumes Completion) If a rule $A \leftarrow \mathcal{B}$ can be deleted by *completion*, i.e., it is completely instantiated w.r.t. a positive body literal $B \in \mathcal{B}$ and the set R of rules, then it can be deleted by *loop detection*, since it is also completely instantiated w.r.t. B and the set $R \backslash \{A \leftarrow \mathcal{B}\}$. Further, if the set R' of rules to be deleted contains only one element, we have the special case of the *extended completion* proposed in Remark 9.2.22. □

Definition 9.2.27 (Well-Founded Transformation) Let $\rightsquigarrow_X$ denote our final rewriting system:

$$\rightsquigarrow_X \; := \; \rightsquigarrow_P \cup \rightsquigarrow_S \cup \rightsquigarrow_N \cup \rightsquigarrow_F \cup \rightsquigarrow_L \cup \rightsquigarrow_A \cup \rightsquigarrow_I .$$

□

The well-founded transformation consists of the non-ground versions of the five transformations used already in the ground transformation of Chapter 4 plus the new transformations *activation* and *instantiation*. The transformations *completion* and *extended completion* are also included implicitly, since they are special cases of *loop detection*.

Remark 9.2.28 (Non-Ground Magic Reduction) Note, that in the non-ground case there is no *magic reduction* transformation. Since the non-ground transformation is goal-directed by itself, the evaluation of magic set transformed programs is not necessary. □

For function-free programs every transformation sequence terminates.

Theorem 9.2.29 (Termination) Let P be a *function-free* normal program. Let Q be an atom. Let S_0 be the initial transformation state of P w.r.t. Q. Then every sequence of transformation states generated by applying $\rightsquigarrow_X$ starting from S_0 reaches a transformation state that is irreducible w.r.t. $\rightsquigarrow_X$ after a finite number of transformation steps.

Proof: The proof of this theorem is based on the following observations that are consequences of the definitions of the transformations:

1. Each application of the transformations *success*, *failure*, *positive reduction*, *negative reduction*, and *loop detection* strictly reduces the size of the rule set R of the current transformation state.

2. Each application of the *activation* transformation adds a new element to the set G of activated goals.

3. Each application of the *instantiation* transformation adds at least one new element to the set U of used rule instances.

From the first point it follows, that after a finite number of applications of the five base transformations always *activation* or *instantiation* has to be applied. But due to the finite Herbrand base of the function-free program P we cannot generate an infinite number of different atoms or rule instances. Together it follows, that there can be no infinite sequence of transformations. □

9.3 Floundering

Note, that we have allowed to activate non-ground negative literals. It may happen that the activated literal is deleted later during the transformation.

Example 9.3.1 (Non-Ground Negative Activation) Consider the following program $different$:

$$\begin{array}{l} different(X,Y) \quad \leftarrow \quad \mathbf{not}\, equal(X,Y). \\ equal(X,X). \end{array}$$

The query $different(X,X)$ leads to an initial transformation state with the single rule

$$different(X,X) \leftarrow \mathbf{not}\, equal(X,X).$$

The only transformation applicable in this state is *activation*. By activating $\mathbf{not}\, equal(X,X)$ we derive the fact $equal(X,X)$. Then we can delete the first rule by *negative reduction* and conclude that $different(X,X)$ is false in the final transformation state. □

In Example 9.3.1 we have activated a non-ground negative literal. Since the rule was deleted later by *negative reduction* we finally derived the correct result. However, if a non-ground negative literal still occurs in the final transformation state, the transformation is called floundered.

Definition 9.3.2 (Floundering) Let $S = (G, R, U)$ be a transformation state. Let S be irreducible w.r.t. $\rightsquigarrow_X$ and let R contain a rule $A \leftarrow \mathcal{B}$ that contains a non-ground negative body literal. Then S is called *floundered*. □

Example 9.3.3 (Floundering) Consider again the program $different$ from Example 9.3.1. The query $different(a,X)$ to the program $different$ leads to an irreducible transformation state with the following rule set R:

$$\begin{array}{l} different(a,X) \quad \leftarrow \quad \mathbf{not}\, equal(a,X). \\ equal(a,a). \end{array}$$

The program is floundered since it contains the non-ground negative body literal $\mathbf{not}\, equal(a, X)$. The application of *instantiation* w.r.t. the negative body literal is not allowed since *instantiation* is defined only for positive body literals. Assume we would apply *instantiation* to generate the rule instance $different(a, a) \leftarrow \mathbf{not}\, equal(a, a)$. Then we could delete the new instance immediately by applying *negative reduction*.

In the well-founded model of this program all ground instances of the atom $different(a, X)$ are true except for the binding $X = a$. There is no way to express this answer except by enumerating all true atoms w.r.t. the given Herbrand base. Thus the only way of computing the answer is replacing the first rule by its Herbrand instantiation, i.e., all ground instances. Assume the Herbrand universe contains the constants a, b, and c. Then the Herbrand instantiation $ground(different)$ of the program is as follows:

$$
\begin{array}{lcl}
different(a, a) & \leftarrow & \mathbf{not}\, equal(a, a). \\
different(a, b) & \leftarrow & \mathbf{not}\, equal(a, b). \\
different(a, c) & \leftarrow & \mathbf{not}\, equal(a, c). \\
equal(a, a). &&
\end{array}
$$

We can apply *positive* and *negative reduction* to obtain the remainder

$$
\begin{array}{l}
different(a, b). \\
different(a, c). \\
equal(a, a).
\end{array}
$$

whose interpretation agrees with the well-founded model of the program $different$ on the given query. □

If an irreducible transformation state S for a program P and an initial query Q is reached that is not floundered, then the well-founded model W_P^* of P and the interpretation $I(S)$ of S agree on the query Q. This result is stated by Theorem 9.5.13 and will be proved in Section 9.5.

Example 9.3.4 (Well-Founded Model) The final transformation state S of Example 9.2.12 is irreducible w.r.t. $\rightsquigarrow_X$ and is not floundered. The final rule set R is

$$
\begin{array}{lcl}
q(a). &&\\
p(a). &&\\
q(X) & \leftarrow & p(X). \\
p(X) & \leftarrow & q(X). \\
p(X) & \leftarrow & \mathbf{not}\, r. \\
r & \leftarrow & \mathbf{not}\, s. \\
s & \leftarrow & \mathbf{not}\, r
\end{array}
$$

The answer to the initial query $q(X)$ is that $q(X)$ is true for $X = a$ in the well-founded model of P and undefined for all other possible bindings of X. In *this* particular example, we have actually computed the complete well-founded model of P since during the computation all rule heads in P have been activated. Thus we have

$$W_P^* = I(S) = \{q(a), p(a)\}.$$

□

Of course, in general the transformation method does not compute the entire well-founded model to answer a query. Consider the program

$$P' \ := \ P \ \cup \ \{ \ m(X) \leftarrow p(X). \ \}$$

extending the program P of Example 9.1.8. In the process of answering the query $Q = q(x)$ the new rule is not activated by the transformation method.

Remark 9.3.5 (Non-Ground Answers) Note, that the well-founded transformation approach is not limited to range-restricted programs. Thus, as illustrated in Example 9.3.4, the rules of the final transformation state do not have to be ground and may contain non-ground facts. □

Example 9.3.6 (Non-Ground Answers) Consider again the program $different$ from Example 9.3.1. For the query $equal(f(X), Y)$ we get an irreducible transformation state with the following rule set R:

$$equal(f(X), f(X)).$$

Here the non-ground answer $equal(f(X), f(X))$ states that all instances of this atom are true and that all remaining instances of the query are false in the well-founded model of the program $different$. This corresponds to Remark 9.1.10 in the sense that a non-ground fact A represents all ground instances $ground(A)$ without enumerating the actual ground instances. □

For range-restricted programs (cf. Definition 2.1.29) we have the following result.

Lemma 9.3.7 Let P be a normal program that is range-restricted. Let Q be a query. Let S be any transformation state that is reachable from the initial transformation state for P and Q. Then the rule set R of S is range-restricted.

Proof: We prove the lemma by induction over the number n of applied transformations. For $n = 0$ we have

$$R = instances(P, Q).$$

Since $instances(P, Q)$ contains only instances of rules of P it is range-restricted.

Now assume the induction hypothesis for an arbitrary n. If the next transformation is *negative reduction*, *failure*, or *loop detection*, rules are deleted and thus range-restriction is preserved. By applying *activation* or *instantiation* instances of P or R are added. Since P and R are range-restricted, all new instances are range-restricted, too. If the next transformation is *positive reduction*, a negative literal is removed from a rule body. This preserves range-restriction, too.

The only transformation that is critical for range-restriction is the *success* transformation, since it removes a positive body literal. However, from the induction hypothesis we know that R is range-restricted, and thus all facts in R are ground. Therefore also the deleted positive body literal must be ground. □

As a consequence of Lemma 9.3.7 it follows that all facts derived from a range-restricted program are ground. But we have an even stronger result.

Theorem 9.3.8 (Ground Remainder for Range-Restricted Programs) Let P be a normal program that is range-restricted. Let Q be an arbitrary query. Let S be any irreducible transformation state that is reachable from the initial transformation state for P and Q. Then the rule set R of S is ground.

Proof: Assume that the rule set R of the final transformation state S is not ground. Let $R' \subseteq R$ consist of exactly the non-ground rules of R. Let $r \equiv (A \leftarrow \mathcal{B}) \in R'$ be any rule of R'. From Lemma 9.3.7 it follows that r is range-restricted. Thus any variable in the rule occurs also in a positive body literal of r. Let $B \in \mathcal{B}$ be such a body literal. B is activated in S since S is irreducible. Further, the rule r is completely instantiated w.r.t. the body literal B and the rule set $R \backslash R'$, because for each head B' in $R \backslash R'$, if B' and B are unifiable with $\sigma = mgu(B, B')$ then the instance $r\sigma$ is used since S is irreducible, and $r\sigma$ is no variant of r because B does contain a variable and B' does not, and thus σ maps the variable in B onto a ground term. Since this condition holds for any rule $r \in R'$ we can apply *loop detection* to delete all rules in R'. But this contradicts our precondition that S is irreducible. Thus the set R must be ground. □

Corollary 9.3.9 (Floundering and Range-Restriction) Let P be a normal program that is range-restricted. Let Q be a query. Let S be any irreducible transformation state that is reachable from the initial transformation state for P and Q. Then S is not floundered.

Proof: Theorem 9.3.8 states that the rule set R in S is ground, i.e., it contains no non-ground body literals at all. Thus S is not floundered. □

9.4 Feasible Transformation States

To prove the correctness of our approach we need some auxiliary definitions. The first definition describes the relation of transformed rules to the original rules of a program P.

Definition 9.4.1 (Reduced Rule Instance) Let L be a first order language. Let $A \leftarrow \mathcal{B}$ and $A' \leftarrow \mathcal{B}'$ be two rules. Let I be an interpretation. $A' \leftarrow \mathcal{B}'$ is called a *reduced rule instance* of $A \leftarrow \mathcal{B}$ w.r.t. I if there is a substitution σ in L (cf. Definition 2.2.1) such that the following conditions hold:

- $A' = A\sigma$,
- $\mathcal{B}' \subseteq \mathcal{B}\sigma$,
- All literals in $\mathcal{B}\sigma \backslash \mathcal{B}'$ are true in I (cf. Definition 3.1.16).

□

A transformation state of P is correct w.r.t. a given interpretation I, if it contains only rule instances, that can be produced by building instances of rules of P and removing body literals that are true in I.

Definition 9.4.2 (Correct Transformation State) Let P be a normal program. Let $S = (G, R, U)$ be a transformation state. Let I be an interpretation. We call S *correct* w.r.t. P and I, if for every rule instance $A' \leftarrow \mathcal{B}' \in R$ there is a rule $A \leftarrow \mathcal{B} \in P$ such that $A' \leftarrow \mathcal{B}'$ is a reduced rule instance of $A \leftarrow \mathcal{B}$ w.r.t. I. □

The next definition describes the relation of transformed rules to (ground) rule instances of P.

Definition 9.4.3 (Extended Rule Instance) Let L be a first order language. Let $A \leftarrow \mathcal{B}$ and $A' \leftarrow \mathcal{B}'$ be two rules. Let I be an interpretation. $A' \leftarrow \mathcal{B}'$ is called an *extended rule instance* of $A \leftarrow \mathcal{B}$ w.r.t. I if there is a substitution σ in L such that the following conditions hold:

- $A' = A\sigma$,
- $\mathcal{B}' \supseteq \mathcal{B}\sigma$,
- All literals in $\mathcal{B}' \backslash \mathcal{B}\sigma$ are true in I.

□

A transformation state $S = (G, R, U)$ of P is complete w.r.t. a given interpretation I, if it contains a more general representation of every ground instance $A' \leftarrow \mathcal{B}'$ of P that has an activated head A' and no body literal false in I.

Definition 9.4.4 (Complete Transformation State) Let P be a normal program. Let $S = (G, R, U)$ be a transformation state. Let I be an interpretation. S is called *complete* w.r.t. P and I, if for each ground instance $(A' \leftarrow \mathcal{B}') \in ground(P)$ holds: if A' is activated in S and $\mathcal{B}'$ contains no body literal $B \in \mathcal{B}'$ false in I, then there is a rule $(A \leftarrow \mathcal{B}) \in R$, such that $A' \leftarrow \mathcal{B}'$ is an extended rule instance of $A \leftarrow \mathcal{B}$ w.r.t. I. □

Definition 9.4.5 (Feasible Transformation State) Let P be a normal program. Let $S = (G, R, U)$ be a transformation state. Let I be an interpretation. We call S *feasible* w.r.t. P and I, if it is correct and complete w.r.t. P and I. □

The next three lemmata state that a transformation state that is correct and complete w.r.t. a given interpretation I_1 stays correct and complete for any interpretation I_2 with $I_1 \subseteq I_2$.

Lemma 9.4.6 (Positive Monotonicity) Let P be a normal program. Let $S = (G, R, U)$ be a transformation state. Let I_1 and I_2 be interpretations such that $I_1 \subseteq I_2$. Let S be correct w.r.t. P and I_1. Then S is correct w.r.t. P and I_2.

Proof: Follows from Definition 9.4.2 since reduced rule instances w.r.t. I_1 are also reduced rule instances w.r.t. I_2. □

Lemma 9.4.7 (Negative Monotonicity) Let P be a normal program. Let $S = (G, R, U)$ be a transformation state. Let I_1 and I_2 be interpretations such that $I_1 \subseteq I_2$. Let S be complete w.r.t. P and I_1. Then S is complete w.r.t. P and I_2.

Proof: Follows from Definition 9.4.4 since extended rule instances w.r.t. I_1 are also extended rule instances w.r.t. I_2. □

Lemma 9.4.8 (Positive and Negative Monotonicity) Let P be a normal program. Let $S = (G, R, U)$ be a transformation state. Let I_1 and I_2 be interpretations such that $I_1 \subseteq I_2$. Let S be feasible w.r.t. P and I_1. Then S is feasible w.r.t. P and I_2.

Proof: Follows directly from Definition 9.4.5 and Lemmata 9.4.6 and 9.4.7. □

The following three lemmata show that the interpretation $I(S)$ of a transformation state S that is correct and complete w.r.t. an interpretation I contains no more positive and negative consequences than the T_P and U_P operators of Definition 3.1.9, respectively.

Lemma 9.4.9 (Positive Consequences) Let P be a normal program. Let $S = (G, R, U)$ be a transformation state. Let I be an interpretation. Let S be correct w.r.t. P and I. Then

$$true(S) \subseteq T_P(I)$$

holds.

Proof: Let $A'' \in true(S)$. Then there is a fact $A' \in R$ such that $A'' = A'\theta$ is an instance of A'. Since S is correct w.r.t. I, is follows that there is a rule $A \leftarrow \mathcal{B}$ in P such that A' is a reduced rule instance of $A \leftarrow \mathcal{B}$. By Definition 9.4.1 there is a substitution σ such that $A' = A\sigma$ and all literals in $\mathcal{B}\sigma$ are true in I. Now consider the instance $(A \leftarrow \mathcal{B})\sigma\theta$. We have $A'' = A'\theta = A\sigma\theta$ and all literals in $\mathcal{B}\sigma\theta$ are true in I. From Definition 3.1.9 it follows that $A'' \in T_P(I)$ holds. □

Lemma 9.4.10 (Negative Consequences) Let P be a normal program. Let $S = (G, R, U)$ be a transformation state. Let I be an interpretation. Let S be complete w.r.t. P and I. Then

$$false(S) \subseteq U_P(I)$$

holds.

Proof: Let $A'' \in false(S)$. Then A'' is activated in S and there is no rule instance in R the head of which is unifiable with A''. Assume that there is a ground rule instance $A'' \leftarrow \mathcal{B}''$ of P such that no body literal $L \in \mathcal{B}''$ is false in I. Since S is complete w.r.t. P and I, R must contain a rule instance of which $A'' \leftarrow \mathcal{B}''$ is an extended rule instance. But then it follows that $A'' \notin false(S)$. By contradiction, every ground rule instance $A'' \leftarrow \mathcal{B}''$ has at least one body literal $L \in \mathcal{B}$ that is false in I. Thus $A'' \in U_P(I)$ follows. □

Lemma 9.4.11 (Positive and Negative Consequences) Let P be a normal program. Let $S = (G, R, U)$ be a transformation state. Let I be an interpretation. Let S be feasible w.r.t. P and I. Then

$$I(S) \subseteq W_P(I)$$

holds.

Proof: Follows directly from Lemmata 9.4.9 and 9.4.10. □

9.5 Correctness of Transformation

Now we will show that each transformation state reachable from the initial state of a program P and a query Q is feasible w.r.t. an appropriate interpretation I. We start with the initial transformation state.

Lemma 9.5.1 (Feasibility of Initial State) Let P be a normal program. Let Q be an atom. Let $S_0 = (G_0, R_0, U_0)$ be the initial transformation state of P w.r.t. Q (cf. Definition 9.1.7). Then S_0 is feasible w.r.t. the empty interpretation $I = \emptyset$.

Proof: All rules in $R_0 = instances(P, Q)$ are (reduced) instances of rules of P, thus S_0 is correct w.r.t. I. Further, for each ground instance $(A' \leftarrow \mathcal{B}') \in ground(P)$ with A' activated in S_0, R_0 contains a rule $A \leftarrow \mathcal{B}$ of which $A' \leftarrow \mathcal{B}'$ is an (extended) instance. Thus S_0 is complete w.r.t. I. With Definition 9.4.5 it follows that S_0 is feasible w.r.t. P and I. □

If a transformation state is correct and complete w.r.t. a given interpretation, then it is still correct and complete after one application of *activation*.

Lemma 9.5.2 (Soundness of Activation) Let P be a normal program. Let $S_1 = (G, R, U)$ be a transformation state. Let I be an interpretation. Let S_1 be feasible w.r.t. P and I. Let S_2 be another transformation state such that $S_1 \rightsquigarrow_A S_2$ holds. Then S_2 is feasible w.r.t. P and I.

Proof: S_2 stays correct, because all rule instances added to R are (reduced) instances of rules of P. Further, for each ground instance $(A' \leftarrow \mathcal{B}') \in ground(P)$ with A' activated in S_2 but not in S_1, a rule $A \leftarrow \mathcal{B}$ of which $A' \leftarrow \mathcal{B}'$ is an (extended) instance is added to R. Thus S_2 is complete w.r.t. I. With Definition 9.4.5 it follows that S_2 is feasible w.r.t. P and I. □

By applying *instantiation* rule instances are added that preserve the correctness of the transformation state.

Lemma 9.5.3 (Soundness of Instantiation) Let P be a normal program. Let $S_1 = (G, R, U)$ be a transformation state. Let I be an interpretation. Let S_1 be feasible w.r.t. P and I. Let S_2 be another transformation state such that $S_1 \rightsquigarrow_I S_2$ holds. Then S_2 is feasible w.r.t. P and I.

Proof: S_2 is complete w.r.t. P and I because no rule instance is removed from R by the transformation step.

Since the rule to be instantiated is a reduced rule instance of a rule of P w.r.t. I, all its instances that have been added to R are also reduced rule instances of the same rule of P w.r.t. I. Thus S_2 is correct w.r.t. P and I. □

With *success* and *positive reduction* we can remove body literals that are true in the current transformation state.

Lemma 9.5.4 (Soundness of Success and Positive Reduction) Let P be a normal program. Let $S_1 = (G, R, U)$ be a transformation state. Let I be an interpretation. Let S_1 be feasible w.r.t. P and I. Let S_2 be another transformation state such that $S_1 \rightsquigarrow_S S_2$ or $S_1 \rightsquigarrow_P S_2$ holds. Then S_2 is feasible w.r.t. P and $I \cup I(S_1)$.

Proof: Let $A \leftarrow \mathcal{B}$ be the rule instance removed from R by success or positive reduction. Since S_1 is correct w.r.t. P and I, $A \leftarrow \mathcal{B}$ is a reduced instance of a rule of P w.r.t. I. Let $A \leftarrow (\mathcal{B} \backslash \{L\})$ be the rule instance added to R by the transformation step. Then L is true in $I(S_1)$ and $A \leftarrow (\mathcal{B} \backslash \{L\})$ is also a reduced rule instance of a rule of P w.r.t. $I \cup I(S_1)$. Thus S_2 is correct w.r.t. P and $I \cup I(S_1)$.

Further, if a ground rule instance of P is an extended rule instance of $A \leftarrow \mathcal{B}$ w.r.t. I then it is also an extended rule instance of $A \leftarrow (\mathcal{B} \backslash \{L\})$ w.r.t. $I \cup I(S_1)$. Thus S_2 is complete w.r.t. P and $I \cup I(S_1)$. □

Lemma 9.5.5 (Soundness of Failure and Negative Reduction) Let P be a normal program. Let $S_1 = (G, R, U)$ be a transformation state. Let I be an interpretation. Let S_1 be feasible w.r.t. P and I. Let S_2 be another transformation state such that $S_1 \rightsquigarrow_F S_2$ or $S_1 \rightsquigarrow_N S_2$ holds. Then S_2 is feasible w.r.t. P and $I \cup I(S_1)$.

Proof: S_2 is correct w.r.t. P and I because no rule instance is added to R by the transformation step. Thus it is also correct w.r.t. P and $I \cup I(S_1)$ with Lemma 9.4.6.

Let $A \leftarrow \mathcal{B}$ be the rule instance removed from R by *failure* or *negative reduction*. Then there is a body literal $L \in \mathcal{B}$ that is false in $I(S_1)$. Therefore all extended ground rule instances of $A \leftarrow \mathcal{B}$ have a body literal that is false in $I(S_1)$. It follows that S_2 is complete w.r.t. P and $I \cup I(S_1)$. □

Remark 9.5.6 Note, that the transformation state S_2 in Lemma 9.5.4 is not correct w.r.t. I, if the removed body literal is true in $I(S_1)$ but not in I. The corresponding statement is true also for Lemma 9.5.5. □

Now we can state the soundness of all transformations except *loop detection*.

Lemma 9.5.7 (Soundness of Base Transformations) Let P be a normal program. Let $S_1 = (G, R, U)$ be a transformation state. Let I be an interpretation. Let S_1 be feasible w.r.t. P and I. Let $I \cup I(S_1) \subseteq W_P^*$. Let S_2 be another transformation state such that $S_1 \rightsquigarrow_T S_2$ holds with $T \in \{A, I, P, S, N, F\}$. Then S_2 is feasible w.r.t. P and $I \cup I(S_1) \cup I(S_2)$ and $I(S_2) \subseteq W_P^*$ holds.

Proof: From Lemmata 9.5.2, 9.5.3, 9.5.4, and 9.5.5 it follows that S_2 is feasible w.r.t. P and $I \cup I(S_1)$. With Lemma 9.4.11 it follows $I(S_2) \subseteq W_P(I \cup I(S_1))$. Since W_P

is monotonic, W_P^* is the least fixpoint of W_P and together with $I \cup I(S_1) \subseteq W_P^*$ it follows that $I(S_2) \subseteq W_P^*$. With Lemma 9.4.8 it follows that S_2 is feasible w.r.t. P and $I \cup I(S_1) \cup I(S_2)$. □

For *loop detection* we do not have a result as in Lemma 9.5.4 or 9.5.5. By applying *loop detection* the completeness of a transformation state may get lost.

Example 9.5.8 (Loop Detection) Consider the program $p{-}loop$ with one rule $p \leftarrow p$. For the query p the initial transformation state is

$$S_0 = \{\{p\}, \{p \leftarrow p\}, \{p \leftarrow p\}\}.$$

By Lemma 9.5.5, S_0 is feasible w.r.t. the empty interpretation $\emptyset$. Now we can delete the rule $p \leftarrow p$ by applying *loop detection*. The resulting transformation state S_1 is not complete w.r.t. the empty interpretation $\emptyset$, because it does not contain a representation of the ground rule $p \leftarrow p$. However, S_1 is feasible w.r.t. the new interpretation $I(S_1) = \{\mathbf{not}\, p\}$. Further, $I(S_1) = W_P(\emptyset) = W_P^*$ holds in this example. □

The following lemma shows that the set of new false atoms resulting from *loop detection* is always an unfounded set w.r.t. the interpretation of the current transformation state.

Lemma 9.5.9 (Loop Detection and Unfounded Set) Let P be a normal program. Let $S_1 = (G, R, U')$ be a transformation state. Let I be an interpretation. Let S_1 be feasible w.r.t. P and I. Let S_2 be another transformation state such that $S_1 \rightsquigarrow_L S_2$ holds. Let R' be the set of rules deleted by the *loop detection* from R. Let

$$U \;:=\; \sim(I(S_2) \backslash I(S_1))$$

be the set of ground atoms that are false in S_2 but not in S_1. Then U is an unfounded set of P w.r.t. the interpretation $I \cup I(S_1)$.

Proof: From Definition 9.1.9 of the interpretation and since the rules in R' are deleted from the rule set R, it follows that

$$U = ground(heads(R')) \backslash ground(heads(R \backslash R'))$$

holds.

According to Definition 3.1.7, we have to show that for all ground instances $(A'' \leftarrow \mathcal{B}'') \in ground(P)$, if $A'' \in U$ then there is a positive body literal $B'' \in \mathcal{B}''$ that is false in $I \cup I(S_1)$ or contained in U.

Let $A'' \leftarrow \mathcal{B}''$ be any ground instance of a rule of P such that $A'' \in U$ holds. If there is a positive body literal $B'' \in \mathcal{B}$ such that B'' is false in I, we are done.

Assume that there is no body literal $B'' \in \mathcal{B}''$ false in I. Since S_1 is complete w.r.t. P and I, there is a rule $A \leftarrow \mathcal{B}$ in R such that $A'' \leftarrow \mathcal{B}''$ is an extended rule instance of $A \leftarrow \mathcal{B}$. According to Definition 9.4.3 there is an substitution σ such that $A'' = A\sigma$ and $\mathcal{B}'' \supseteq \mathcal{B}\sigma$ holds.

Assume that $A \leftarrow \mathcal{B}$ is contained in $R \backslash R'$. Then A'' cannot be in U. Thus, $A \leftarrow \mathcal{B}$ must be contained in R'.

From Definition 9.2.24 is follows that the rule $A \leftarrow \mathcal{B}$ is completely instantiated through a positive body literal $B \in \mathcal{B}$ w.r.t. in $R \backslash R'$. Since $\mathcal{B}'' \supseteq \mathcal{B}\sigma$ holds, $B'' := B\sigma$ is a positive ground literal in $\mathcal{B}''$. Let us consider the following three cases.

1. If B'' is not unifiable with any head in R, B'' is false in I. But this contradicts our assumption that no body literal $B'' \in \mathcal{B}''$ is false in I.
2. If B'' is unifiable with a head in R' but not with a head in $R \backslash R'$ then B'' is contained in U. In this case the assertion follows.
3. Otherwise B'' is unifiable with a head in $R \backslash R'$.

In the last case, we know that $A \leftarrow \mathcal{B}$ is completely instantiated through B w.r.t. $R \backslash R'$. Further, $B'' = B\sigma$ is unifiable with a head in $R \backslash R'$. Thus, also B is unifiable with that head. Let θ be their most general unifier. From Definition 9.2.14 it follows that the instance $A' \leftarrow \mathcal{B}' := (A \leftarrow \mathcal{B})\theta$ is used in S_1. Note, that $A'' \leftarrow \mathcal{B}''$ is an extended rule instance of $A' \leftarrow \mathcal{B}'$. There are again three possible cases.

1. If $A' \leftarrow \mathcal{B}'$ is not contained in R any more, then it has been deleted due to a body literal false in $I(S_1)$. But, being an instance of $\mathcal{B}'$, also $\mathcal{B}''$ contains a literal false in $I(S_1)$. In this case the assertion follows.
2. If $A' \leftarrow \mathcal{B}'$ is still contained in $R \backslash R'$ then A'' as an instance of A' cannot be contained in U. This is a contradiction, since A' is contained in U by definition.
3. Otherwise $A' \leftarrow \mathcal{B}'$ is contained in R'.

In the last case we have found another rule $(A' \leftarrow \mathcal{B}') \in R'$ of which $A'' \leftarrow \mathcal{B}''$ is an extended rule instance. Now consider the same cases that we have considered for the rule $A \leftarrow \mathcal{B}$. It may be, that we have to examine these cases for each positive body literal of $A'' \leftarrow \mathcal{B}''$. However, by instantiating $A \leftarrow \mathcal{B}$ again and again, each positive body literal can be considered at most one time. Finally, we must reach a case where we find a $B'' \in \mathcal{B}''$ such that B'' is false in I or in $I(S_1)$ or that B'' is contained in U.

□

As a consequence of Lemma 9.5.9 we get the following result, which extends the statement of Lemma 9.5.7 from the base transformations to the *loop detection*.

Lemma 9.5.10 (Soundness of Loop Detection) Let P be a normal program. Let $S_1 = (G, R, U)$ be a transformation state. Let I be an interpretation. Let S_1 be feasible w.r.t. P and I. Let $I \cup I(S_1) \subseteq W_P^*$. Let S_2 be another transformation state such that $S_1 \rightsquigarrow_L S_2$ holds. Then S_2 is feasible w.r.t. P and $I \cup I(S_1) \cup I(S_2)$ and $I(S_2) \subseteq W_P^*$ holds.

Proof: Let L be in $I(S_2)$. If $L \in I(S_1)$ then $L \in W_P^*$ follows since $I(S_1) \subseteq W_P^*$ holds. Otherwise $\sim L$ is in contained in

$$U := \sim(I(S_2) \backslash I(S_1)).$$

From Lemma 9.5.7 we know that U is an unfounded set w.r.t. $I \cup I(S_1)$. Thus,

$$U \subseteq U_P(I \cup I(S_1))$$

holds, and $L \subseteq W_P^*$ follows. Together, we get $I(S_2) \subseteq W_P^*$.

Finally, we have to show that S_2 is feasible w.r.t. P and $I \cup I(S_1) \cup I(S_2)$. S_2 is correct, since S_1 is correct w.r.t. P and I, and we have only deleted rules from R. Now consider any ground instance $(A'' \leftarrow \mathcal{B}'') \in ground(P)$ such that A'' is activated in S_2 and no body literal in $\mathcal{B}''$ is false in $I \cup I(S_1) \cup I(S_2)$. Since S_1 is complete w.r.t. I and thus also w.r.t. $I \cup I(S_1) \cup I(S_2)$, we conclude that R contains a rule $A \leftarrow \mathcal{B}$ of which $A'' \leftarrow \mathcal{B}''$ is an extended rule instance. Since no body literal in $\mathcal{B}$ is false in $I(S_2)$, it cannot be deleted in the last application of *loop detection*. Therefore, S_2 is also complete and thus feasible w.r.t. $I \cup I(S_1) \cup I(S_2)$. □

Theorem 9.5.11 (Soundness of Transformation) Let S_n be a transformation state of P that is the result of n transformation applications to an initial transformation state S_0. Then S_n is feasible w.r.t. P and $\bigcup_{i=0}^{n} I(S_i)$ and

$$I(S_n) \subseteq W_P^*$$

holds.

Proof: The proof is by induction on n. With Lemma 9.5.1 and Lemma 9.4.11 the base case follows immediately.

Let us assume that S_n is feasible w.r.t. P and $I_n := \bigcup_{i=0}^{n} I(S_i)$ and that $I_n \subseteq W_P^*$ holds. Let S_{n+1} be another transformation state such that $S_n \rightsquigarrow_X^* S_{n+1}$ holds. With Lemmata 9.5.7 and 9.5.10 it follows that S_{n+1} is feasible w.r.t. P and I_{n+1} and that $I(S_{n+1}) \subseteq I_n \subseteq W_P^*$ holds. □

Lemma 9.5.12 (Completeness of Transformation) Let P be a normal program. Let $S = (G, R, U)$ be a transformation state. Let S be feasible w.r.t. P and $I(S)$. Let S be irreducible w.r.t. $\rightsquigarrow_X$ and not floundered. Let Q be an atom that is activated in S. Then

$$T_P(I(S)) \cap ground(Q) \subseteq true(S)$$

and

$$U_P(I(S)) \cap ground(Q) \subseteq false(S)$$

hold.

Proof: Let A be any atom in $T_P(I(S))$. Then there is a ground rule $A \leftarrow \mathcal{B}$ in $ground(P)$ such that $\mathcal{B} \subseteq I(S)$ holds. Thus all body literals in $\mathcal{B}$ are true in S. Due to the completeness of S w.r.t. P and $I(S)$, there must be a rule $A' \leftarrow \mathcal{B}'$ in R such that $A \leftarrow \mathcal{B}$ is an extended instance of $A' \leftarrow \mathcal{B}'$. Then with *instantiation* and *success* all positive body literals in $A' \leftarrow \mathcal{B}$ can be removed. Since S is irreducible, this has been done already. Since S is not floundered, the remaining negative body literals must be ground. Thus they must be contained in $\mathcal{B}$ and be true. Since S is irreducible they must have been reduced. Now A must be an instance of the remaining head and thus A is true in S. It follows that $T_P(I) \subseteq true(S)$ holds.

Now let $\mathcal{A} = W_P(I(S))$ be the greatest unfounded set of P w.r.t. $I(S)$. Let R' be the subset of rules in R whose head in unifiable with an atom from R'. If $R' = \emptyset$ holds, we have $\mathcal{A} \subseteq false(S)$. Now assume that R' is not empty. Let $A' \leftarrow \mathcal{B}'$ be any rule in R'. Since $\mathcal{A}$ is an unfounded set and due to the correctness of S, for each ground instance of $A \leftarrow \mathcal{B}$ of $A' \leftarrow \mathcal{B}'$, $\mathcal{B}$ must contain a literal that is false in $I(S)$ or an atom contained in $\mathcal{A}$. In the first case the rule would have been removed by *instantiation* and *completion*. But this is not possible since S is irreducible. In the second case an atom in $\mathcal{B}'$ does contain an atom that is unifiable with an atom from $\mathcal{A}$, and thus unifiable with a head in R'. But then *loop detection* can be applied to delete all rules in R' simultaneously. But this is again not possible since S is irreducible. By contradiction, R' must be empty. □

Theorem 9.5.13 (Partial Correctness and Completeness) Let P be a normal program. Let S_n be a transformation state of P that is the result of n transformations starting from the initial transformation state S_0 of P w.r.t. any query. Let Q be an atom that is activated in S_n. If S_n is irreducible w.r.t. $\rightsquigarrow_X$ and is not floundered, then $I(S_n)$ and W_P^* agree on Q.

Proof: From Theorem 9.5.11 it follows that

$$I(S_n) \subseteq W_P^*$$

and thus

$$I(S_n) \subseteq W_P(I(S_n))$$

holds. From Lemma 9.5.12 it follows that

$$W_P(I(S)) \cap (ground(Q) \cup ground(\mathbf{not}\, Q)) \subseteq I(S)$$

holds. Together it follows that $I(S_n)$ and W_P^* agree on Q. □

Remark 9.5.14 (Non-Confluence of $\leadsto_X$) The non-ground transformation is not confluent. Depending on which atoms are activated during a transformation sequence different subsets of the well-founded model will be computed. One special case is to compute the complete well-founded model W_P^* by activating all atoms occurring in the program P. However, Theorem 9.5.13 guarantees that the interpretation of any final (non-floundered) transformation state agrees with the well-founded model W_P^* of P on the given query Q. This means that in any case the same result for Q is computed. □

From Theorem 9.5.13 follows that by the non-ground transformation we always get a correct subset of the well-founded model W_P^* of a given normal program. Thus, regarding the final interpretation, we always consider a set of ground literals. For range-restricted programs this is what we expect, since Definition 2.1.29 of range-restriction ensures that all variables are bound to terms of U_P. However, for programs that are not range-restricted non-ground facts may be computed. The following corollary states that if a non-ground literal is true or undefined w.r.t. the well-founded semantics of a program P, then in the final state of a non-ground transformation a corresponding non-ground fact or rule head is present. This means, that most-general results are derived instead of enumerating all ground terms from the underlying first order language.

Corollary 9.5.15 (Most General Answers) Let P be a normal program. Let Q be an atom. Let $S = (G, R, U)$ be any non-ground transformation state reached from the initial state for P and Q that is irreducible and not floundered. Let Q'' be any instance of Q. If Q'' is true or undefined w.r.t. the well-founded semantics of P (cf. Definition 3.1.16), then the set R contains a fact Q' or rule head Q', respectively, such that Q'' is an instance of Q'.

Proof: This result is a consequence of Theorem 9.5.13 and Definition 9.1.9. Let $X_1, \ldots, X_n$ be the different variables occurring in Q''. Let $c_1, \ldots, c_n$ be n new constants not occurring in P. Let

$$Q_c \;:=\; Q''\{X_1/c_1, \ldots, X_n/c_n\}$$

be the ground instance of Q'' resulting from replacing the variables X_i by c_i, $1 \leq i \leq n$. Since Q'' is true or undefined w.r.t. W^*_P, we have $Q_c \in I(S)$ as a consequence of Theorem 9.5.13. But since the transformations use only constants from the program and do not invent new constants, Q_c must be an instance of an atom Q' that is a fact or head in R and does not contain any of the constants $c_1, \ldots, c_n$. Thus, Q'' is an instance of Q'. □

Chapter 10

Non-Ground Implementation

In this chapter we will give an overview of our prototypical implementation of the non-ground transformation. We will point out how we extended the ground implementation described in Chapter 8. However, we will omit many details necessary for an actual implementation since they are rather technical and not essential for this work.

10.1 Queue-Based Non-Ground Transformation

We implemented the non-ground transformation as an extension of the queue-based ground transformation described in Section 8.2. The major differences between the two implementations are as follows:

- In the non-ground transformation we have to process a body atom B not if there is (or is not) a head atom A that is equal to B, but if there is (or is not) a head atom A that is *unifiable* with B.
- Whereas for the ground transformation the set of transformed rules is only reduced, in the non-ground transformation new instances are generated during the transformation, thus, we have to add new elements to the data structure not only in the initialization phase.

From the first point it follows that we have to change our data structure. Instead of only considering atoms, that may occur in rule heads or rule bodies, we distinguish between head atoms A, that may become true or get deleted, and body atoms B, that have to be processed, if they are instances of A or at least unifiable with A.

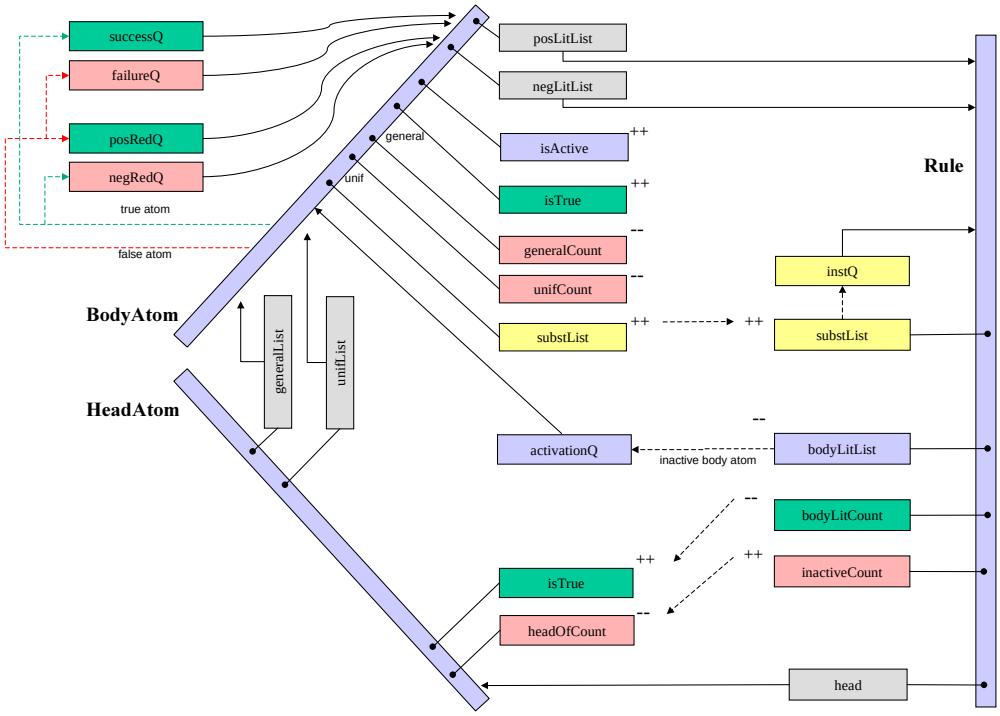


Figure 10.1: Queue-Based Non-Ground Transformation

Definition 10.1.1 (Non-Ground Data Structure) In Figure 10.1 the extended data structure for the non-ground transformation is displayed. For each *different* head atom A there is an object, depicted on the left side at the bottom of Figure 10.1, that has the following fields:

- The flag $isTrue$ is set for facts. This is exactly the same as in the ground case.
- The counter $headOfCount$ holds the number of rules having this atom as their head. If this counter is decremented and reaches zero, this atom may get false, if there is no other head atom unifiable with A.
- The list $generalList$ holds references to those body atoms B that are instances of A, i.e., that are true if A is a fact.
- The list $unifList$ holds references to those body atoms B that are unifiable with A, but that are no instances. Note, that the instances of A are already contained in $generalList$. The body atoms in $unifList$ and $generalList$ may get false, if the last rule having A in its head is deleted.

For each *different* body atom B in the current transformation state we have an object, depicted on the upper left side of Figure 10.1, which has the following fields:

- The list $posLitList$ holds references to all rule objects, that have B as a positive body literal.
- The list $negLitList$ holds references to all rule objects, that have $\mathbf{not}\, B$ as a negative body literal. These two lists are exactly the same as in the ground case.
- The flag $isActive$ is set, if B (or another atom being more general) is activated.
- The flag $isTrue$ is set, as soon as we find a true head atom A that is more general than B (cf. $generalList$).
- The counter $generalCount$ holds the number of head atoms A being more general than B.
- The counter $unifCount$ holds the number of head atoms A being unifiable with but not more general than B. Note, that the number of more general atoms is already stored in $generalCount$.
- For each head atom A that is unifiable with B, the list $substList$ contains their most general unifier $mgu(A, B)$.

For each rule $r \equiv A \leftarrow \mathcal{B}$ in the set R of transformed rules in the current transformation state we have an object depicted at the right side of Figure 10.1, that has the following fields:

- The reference $head$ points to the object associated with the head atom A of the rule.
- The counter $inactiveCount$ holds the number of false body literals and is initialized to 0. This is exactly the same as in the ground case.
- The counter $bodyLitCount$ holds the number of body literals, that have not been removed from the rule. It is initialized to the number of body literals in B. This corresponds to the sum of the counters $posLitCount$ and $negLitCount$ in the ground case.
- The list $bodyLitList$ holds references to the objects representing the body atoms $B \in \mathcal{B}$. This list is needed to select literals for *activation*.
- For each head atom A that is unifiable with an active positive body literal $B \in \mathcal{B}$, the list $substList$ contains their most general unifier $mgu(A, B)$, if it has not yet been applied to r by the *instantiation* transformation.

The following queues are used:

- The queues $successQ$ and $negRedQ$ contain references to true body atoms, and the queues $failureQ$ and $posRedQ$ contain references to false body atoms, that are still to be processed by the corresponding transformation. This is exactly the same as in the ground case.
- The queue $activationQ$ contains references to those body atoms B that occur in a rule $A \leftarrow \mathcal{B}$ having no activated body literal.
- the queue $instQ$ contains references to those rule objects that have a non-empty $substList$.

For an initial transformation state $S = (G, R, U)$ the data structure is initialized such that for each rule $A \leftarrow \mathcal{B}$ in R, objects associated with the rule, the head atom, and the body atoms are created, and the counters and reference lists are initialized such that they satisfy the conditions stated above in this definition. Note, that if an atom occurs in more than one head or body literal, all occurrences are represented by the same object.

As in the ground case, references to true body atoms are copied to $successQ$ and $negRedQ$. References to false body atoms, i.e., that are activated and have $generalCount = 0$ and $unifCount = 0$, are copied to $failureQ$ and $posRedQ$. □

The transformation proceeds as follows.

Definition 10.1.2 (Non-Ground Transformation Process) Let P be a normal program. Let Q be a query. Let the data structure of Definition 10.1.1 be initialized for P and Q as described in Definition 10.1.1.

Now a strategy component can select an atom B or a rule r from one of the six queues. If an atom B from $successQ$ or $failureQ$ is selected, the following steps are performed:

1. The atom B is removed from the queue.
2. For each rule $r \equiv A \leftarrow \mathcal{B}$ in $B.posLitList$ do:
   (a) For *success*: decrement the counter $r.bodyLitCount$, and remove the literal B from $r.bodyLitList$.
   (b) For *failure*: increment the counter $r.inactiveCount$.

If an atom B from $posRedQ$ or $negRedQ$ is selected, the following steps are performed:

1. The atom B is removed from the queue.
2. For each rule $r \equiv A \leftarrow \mathcal{B}$ in $B.negLitList$ do:
   (a) For *positive reduction*: decrement the counter $r.bodyLitCount$, and remove the literal **not** B from $r.bodyLitList$.
   (b) For *negative reduction*: increment the counter $r.inactiveCount$.

If for a rule $r \equiv A \leftarrow \mathcal{B}$ the conditions $r.bodyLitCount = 0$ and $r.inactiveCount = 0$ become true during the application of *success* or *positive reduction*, then:

1. Get the head atom $A \equiv r.head$ of the rule.
2. If $A.isTrue$ is not set, then:
   (a) Set $A.isTrue$.
   (b) For each body atom B in $A.generalList$, having $B.isTrue$ not set, do:
      i. Set $B.isTrue$.
      ii. Copy a reference to B to $successQ$ and $negRedQ$.

If for a rule $r \equiv A \leftarrow \mathcal{B}$ the counter $r.inactiveCount$ changes from 0 to 1 during the application of *failure* or *negative reduction*, then:

1. Get the head atom $A \equiv r.head$ of the rule,
2. Decrease the counter $A.headOfCount$.
3. If $A.headOfCount$ reaches the value 0, then:
   (a) For each body atom B in $A.generalList$ or in $A.unifList$, do:
      i. Decrement the counter $B.generalCount$ or $B.unifCount$, depending on the list in which the body atom B was found.
      ii. If the conditions $B.generalCount = 0$ and $B.unifCount = 0$ are satisfied and $B.isActive$ is set, then copy a reference to B to $failureQ$ and $posRedQ$.

If for a rule $r \equiv A \leftarrow \mathcal{B}$ a true literal L is removed from the list $r.bodyLitList$ during the application of *success* or *positive reduction*, and then $r.bodyLitList$ is not empty but does not contain any activated body literal any more, then select a literal from $r.bodyLitList$ for *activation* and add its atom to $activationQ$.

If a body atom B from $activationQ$ is selected, do the following:

1. Remove the atom B from $activationQ$.

2. If $B.isActive$ is not set, then:

   (a) Add all rules in $instances(P, B)$ to the data structure according to Definition 10.1.1.

      i. Adjust all counters, reference lists, and queues accordingly.

      ii. Update the lists $generalList$ and $unifList$ for all head and body atoms added.

      iii. If a body atom B is added to the list $unifList$ of a head atom A, then:

         A. The unifier $\theta = mgu(A, B)$ is added to $B.substList$.

         B. If $B.isActive$ is set, then for each rule $r \equiv A \leftarrow \mathcal{B}$ in $B.posLitList$, add θ to $r.substList$. If $r.substList$ was empty, then r is added to $instQ$.

   (b) For all head atoms B' that are instances of B, including B itself, having $B'.isActive$ not set, do:

      i. Set $B'.isActive$.

      ii. If the conditions $B'.generalCount = 0$ and $B'.unifCount = 0$ hold, add B' to $failureQ$ and $posRedQ$.

If a rule $r \equiv A \leftarrow \mathcal{B}$ is selected from $instQ$, then do the following steps:

1. Select a substitution θ from $r.substList$ and remove from $r.substList$.

2. If $r.substList$ is empty now, remove r from $instQ$.

3. Add the instance $r\theta$ to the data structure, maintaining all necessary counters, reference lists, and queues.

This process is continued until all six queues are empty. □

Remark 10.1.3 (Selective Activation) Note, that the described non-ground transformation applies only the activation of at most one body literal per rule. For other types of activation, the algorithm can be modified easily. We will discuss different activation strategies in Section 11.1. □

Remark 10.1.4 (Selective Instantiation) The described algorithm applies *instantiation* w.r.t. all head atoms. However, the algorithm can be modified easily such that *instantiation* w.r.t. to facts is applied first, since this implies, that *success* can be applied immediately to the new instance. □

Remark 10.1.5 (Loop Detection) The application of *loop detection* is not supported directly by the data structure of Definition 10.1.1. The *loop detection* has to be implemented separately. The (extended) *completion* transformation of Definition 9.2.17 and Remark 9.2.22 can be implemented using the $substList$ of a rule object. If a rule has an activated positive body literal, and all applicable substitutions have been applied, the rule is a good candidate for *completion*. □

We will not give a correctness proof for this algorithm since the necessary technical details would extend the scope of this work. However, we have the following result concerning the relation of this algorithm to the ground transformation.

Lemma 10.1.6 (Non-Ground Implementation for Ground Programs) Let P be a ground program. Then the application of the non-ground transformation as described in Definitions 10.1.1 and 10.1.2 to P to compute the complete model W_P^* coincides with the queue based transformation of Section 8.2.

Proof: The transformations *activation* and *instantiation* are not needed for this application. Thus we only consider the four basic transformations. Further, we do not need unification to compare head and body atoms, since ground atoms are unifiable if and only if they are equal. Thus, the list $generalList$ is not needed at all, and the $unifList$ always holds a link from a head atom A to a body atom B, if $A \equiv B$ holds. Then the remaining parts of the queue-based non-ground transformation coincide exactly with the ground transformation. □

Remark 10.1.7 (Non-Ground Indexing) For the storage and retrieval of non-ground atoms, we have implemented the *grid* data structure described in [MW88]. This structure is used in the implementation whenever for an atom A another atom B unifiable with A has to be found. For instance, if a new rule $A \leftarrow \mathcal{B}$ is inserted into the data structure of Definition 10.1.1 all body atoms B of existing rules that are unifiable with the rule head A have to be found. □

Experiments using our prototype implementation have shown, that the queue based transformation proposed in this section is competitive for goal-directed query answering as compared to the ground implementation. However, our implementation cannot compete with Prolog implementations such as the XSB system [SSW94, RSS$^+$97], that are highly optimized for their evaluation strategy. For a really efficient implementation of the non-ground transformation, many more optimizations are necessary.

However, we have shown that the non-ground transformation is not just a theoretical concept but can actually be implemented. The implementation is a valuable tool to verify and compare different strategies by applying them to real life applications. Some experimental results of the non-ground implementation will be shown in Example 11.5.2.

Chapter 11

Non-Ground Scheduling Strategies

11.1 Non-Ground Regular Strategy Expressions

We will now extend the definition of regular strategy expressions of Section 6.2 to the non-ground transformation.

Inductive Definition 11.1.1 (Non-Ground Regular Strategy Expression) A *(non-ground) regular strategy expression* is inductively defined as follows:

- Let $\rightsquigarrow_e$ be a non-ground transformation. Then e is an *atomic* (non-ground) regular strategy expressions.
- If e_1 and e_2 are (non-ground) regular strategy expressions, then $(e_1|e_2)$ is an *alternative* (non-ground) regular strategy expression.
- If e_1 and e_2 are (non-ground) regular strategy expressions, then $(e_1 e_2)$ is a *sequential* (non-ground) regular strategy expression.
- If e is a (non-ground) regular strategy expression, then e^* is a *closure* (non-ground) regular strategy expression.

□

Inductive Definition 11.1.2 (Application of a Regular Strategy Expression) Let S be a non-ground transformation state. Let e be a regular strategy expression. Another non-ground transformation state S' is a possible result of the *application of* e *to* S, $S \overset{e}{\mapsto} S'$, if the following conditions hold:

- If e is an *atomic* (non-ground) regular strategy expression, i.e., there is a non-ground transformation $\rightsquigarrow_e$, then $S \rightsquigarrow_e S'$ holds.

- If $e \equiv (e_1|e_2)$ is an alternative (non-ground) regular strategy expression, then $S \overset{e_1}{\mapsto} S'$ or $S \overset{e_2}{\mapsto} S'$ holds.
- If $e \equiv (e_1 e_2)$ is a sequential (non-ground) regular strategy expression, then there exists a non-ground transformation state S'' such that $S \overset{e_1}{\mapsto} S''$ and $S'' \overset{e_2}{\mapsto} S'$ hold.
- If $e \equiv e_1^*$ is a closure (non-ground) regular strategy expression, then S' results from S by n applications of e_1, with $n \geq 0$.

□

Remark 11.1.3 (Parentheses) As in the ground case, we will omit parentheses in regular strategy expressions whenever possible. □

According to Definition 11.1.2 non-ground strategy expression can be applied only to non-ground transformation states. The following definition allows to apply the expression also to programs.

Definition 11.1.4 (Application to Programs) Let P_1 and P_2 be normal programs. Let e be a (non-ground) regular strategy expression. Then $P_1 \overset{e}{\mapsto} P_2$ is a notation for the following statements:

- S_1 is the initial non-ground transformation state for P_1 and any query atom or set of query atoms.
- $S_2 = (G, R, U)$ is a non-ground transformation state with $R = P_2$.
- $S_1 \overset{e}{\mapsto} S_2$ holds.

□

Remark 11.1.5 (Level of Abstraction) In the ground case we had some nice confluence properties allowing us to specify concrete strategies on the abstract levels of regular expressions. In the non-ground case the situation is more complicated. Many decisions, for instance which literal should be activated or delayed and in which situation a parallel activation should be performed, cannot be made on this high level of abstraction but need a careful examination of the current transformation state. □

To improve the expressional power of the strategy expression, we define *selective regular strategy expressions*, that can be combined with additional conditions for the application of the transformation. The non-ground transformation having the greatest effect on the evaluation strategy is the *activation*.

Definition 11.1.6 (Selective Activation) For the non-ground transformation *activation* we define the following three *selective* regular strategy expressions:

- The selective strategy expression A_{single} restricts the application of the non-ground activation such that at most one *single* body literal of each rule gets activated.
- The selective strategy expression A_{delay} is allowed to activate more than one body literal of a rule. Actually, this is the normal *activation*. However, we will use the expression A_{delay} when it is important to apply the parallel activation, e.g. to describe the delay operation of the SLG resolution (cf. Section 11.3).
- The selective strategy expression A_{all} is allowed to activate atoms currently not appearing in the rule set R. This transformation is intended to be applied in the initial transformation state to activate all predicates defined in the original program at once in order to compute the complete well-founded model.

□

11.2 Relation to Ground Transformation

Before we will discuss the features of the transformation that are specific for the non-ground case, we will show that the non-ground transformation is a generalization of the ground transformation.

Theorem 11.2.1 (Non-Ground Subsumes Ground Transformation) Let P be any ground program. Let

$$P = P_0 \mapsto_X P_1 \mapsto_X \ldots \mapsto_X P_n$$

be any sequence of ground transformations (without *magic reduction*) applied to P. Then there is a sequence

$$S_0 \rightsquigarrow_X S_1 \rightsquigarrow_X \ldots \rightsquigarrow_X S_n$$

of non-ground transformations, such that for each $S_i = (G_i, R_i, U_i)$, for $0 \leq i \leq n$, $P_i = R_i$ holds.

Proof: The proof is by induction on n, Let $n = 0$. Then let S_0 be the initial non-ground transformation state for P, such that all predicates are activated, i.e., for each predicate symbol there is an atom A with pairwise different variables in U_0. Then $R_0 = P$ holds, since all rules from P are included without applying any unifier.

Now assume the result for $n - 1$. Then $R_{n-1} = P_{n-1}$ holds by the induction hypothesis. Assume that one of the four basic transformations is applied to P_{n-1}, i.e., $P_{n-1} \mapsto_{PSNF} P_n$ holds. Then the non-ground version of the same transformation is

applicable to S_{i-1} since all atoms true or false in P_{i-1} are also true or false in S_{i-1}, according to the Definitions 4.1.3 and 9.1.9.

If *loop detection* is applied w.r.t. an unfounded set U, let

$$R' := \{(A \leftarrow \mathcal{B}) \in R_{n-1} \mid A \in U\}$$

be the set of rules in R_{n-1} having a head atom contained in U. According to Definition 4.3.14, for each $A \leftarrow \mathcal{B}$ in R' there is a body literal $B \in \mathcal{B}$ such that $B \in U$ holds. There is no rule in $R_{n-1} \backslash R'$ having B as head, due to the definition of R'. Thus $A \leftarrow \mathcal{B}$ is completely instantiated w.r.t. $R_{n-1} \backslash R'$, using Definition 9.2.14. Then the set R' can be deleted by one application of *loop detection* and the result follows. □

Corollary 11.2.2 (Strategy Expressions) Let P_1 and P_2 be ground programs. Let e be a *ground* strategy expression (without *magic reduction*), such that

$$P_1 \overset{e}{\mapsto} P_2$$

holds. Then

$$P_1 \overset{(A_{all} e)}{\mapsto} P_2$$

holds for the non-ground transformation.

Proof: The application of the non-ground expression A_{all} activates all predicates and thus all rules. In the resulting state $S = (G, R, U)$ we have $R = P_1$. Then the result follows from Definition 11.1.4 and Theorem 11.2.1. □

Now we can apply the results of Chapter 6 if we apply the non-ground transformation to ground programs. We will do this in Section 11.4.

11.3 Relation to SLG Resolution

Our non-ground transformation is strongly related to the SLG resolution [CW93, CSW95, SSW00]. For a precise comparison we first recall the definition of the SLG resolution.

Definition 11.3.1 (X-Clause [CSW95]) An *X-clause* is a clause of the form

$$A \leftarrow D | \mathcal{B}$$

where A is an atom, D is a sequence of delayed ground negative literals and possibly non-ground atoms, and $\mathcal{B}$ is a sequence of literals. Literals in D are called *delay literals*. If $\mathcal{B}$ is empty, an X-clause is called an *answer clause*. □

A clause in a program is viewed as an X-clause in which D is empty. We usually omit the | when D is empty. As far as the declarative semantics is concerned, each X-clause is viewed as an ordinary clause whose body is the conjunction of all literals in D and $\mathcal{B}$. That is, the | is purely a control annotation.

Definition 11.3.2 (Computation Rule for X-clauses [CSW95]) Let $A \leftarrow D|\mathcal{B}$ an X-clause. A *computation rule* selects exactly one literal from $\mathcal{B}$. This literal is called the *selected literal*. □

The resolution of X-clauses is defined as follows.

Definition 11.3.3 (SLG Resolution [CSW95]) Let G be an X-clause

$$A \leftarrow D|L_1, ..., L_n$$

where $n > 0$. Let L_i be the selected atom. Let C be an X-clause with no delayed literals, and C' of the form $A' \leftarrow L'_1, ..., L'_m$ be a variant of C with variables renamed so that G and C' have no variables in common. G is *SLG resolvable* with C if L_i and A' are unifiable. The clause

$$(A \leftarrow D|L_1, ..., L_{i-1}, L'_1, ..., L'_m, L_{i+1}, ..., L_n)\theta$$

is the *SLG resolvent* of G with C, where θ is a most general unifier of L_i and A'. □

SLG resolution is used for resolution with a clause in a program or with an answer clause that has an empty sequence of delayed literals. For an answer clause that has a non-empty sequence of delayed literals, relevant variable bindings in the head of the answer clause are propagated by *SLG factoring*, but the sequence of delayed literals in the body is not propagated.

Definition 11.3.4 (SLG Factoring [CSW95]) Let G be an X-clause

$$A \leftarrow D|L_1, ..., L_n$$

where $n > 0$. Let L_i be the selected atom. Let C be an answer clause, and C' of the form $A' \leftarrow D'|$ be a variant of C with variables renamed so that G and C' have no variables in common. If D' is not empty and L_i and A' are unifiable with a most general unifier θ, the *SLG factor* of G with C is

$$(A \leftarrow D, L_i|L_1, ..., L_{i-1}, L_{i+1}, ..., L_n)\theta.$$

□

Remark 11.3.5 The motivation of not propagating delayed literals in an answer clause is to guarantee polynomial complexity for computation of queries on function-free programs. If there are multiple answer clauses with the same atom in the head, only one of them will be propagated by using either SLG resolution or SLG factoring (cf. Examples 4.2.16 and 4.3.9). □

Now we define the central data structure for the SLG resolution.

Definition 11.3.6 (Search Forest) A *search forest* $\mathcal{F}$ for a normal program P and a set $\mathcal{A}$ of activated atoms contains a *search tree* $\mathcal{T}_A$ for each atom $A \in \mathcal{A}$. Each search tree $\mathcal{T}_A$ in $\mathcal{F}$ can be marked as *completed*.

Each node in a search tree $\mathcal{T}_A$ is labeled by an X-clause. The root node R of the tree $\mathcal{T}_A$ is labeled by the clause $A \leftarrow A$. For each rule $r \in instances(P, A)$ the root node R of $\mathcal{T}_A$ has a child node N labeled by the clause r.

Each non-root node N labeled by the rule $A' \leftarrow \mathcal{B}$ has a selected body literal $B \in \mathcal{B}$, if $\mathcal{B}$ is not empty. Further, each non-root node N has a status value that implies a condition. The possible status values and their conditions are as follows:

***new*:** The node is newly created and has no children.

***answer*:** The clause that is the label of N is an answer clause.

***floundered*:** The selected literal of the node is a non-ground negative literal.

***active*:** The selected literal B is not floundered and we are waiting for answers from the tree $\mathcal{T}_B$ that also exists in the forest.

***disposed*:** All possible child nodes of the node have been created and so the node is no longer useful.

For each non-root node N that is *active* and whose selected literal is positive there is a set $Answers_N$ of atoms. Each atom $B' \in Answers_N$ is an answer in the tree $\mathcal{T}_B$ for the selected body literal of N.

The *initial search forest* $\mathcal{F}$ for a program P and an atom A contains only the search tree $\mathcal{T}_A$ for A with the root node R and a *new* child node N for each rule in $instances(P, A)$. □

A set $\mathcal{A}$ of atoms is completely evaluated, if the trees $\mathcal{T}_A$ for all $A \in \mathcal{A}$ in $\mathcal{F}$ cannot produce any new answers.

Definition 11.3.7 (Complete Evaluation) Let $\mathcal{F}$ be a search forest for a program P. Let $\mathcal{A}$ be a set of atoms such that for each atom $A \in \mathcal{A}$ there is a search tree $\mathcal{T}_A \in \mathcal{F}$. We say that $\mathcal{A}$ is *completely evaluated* in $\mathcal{F}$, if for each atom $A \in \mathcal{A}$ the following conditions hold:

- Each non-root node N in $\mathcal{T}_A$ has the status *answer*, *active*, or *disposed*.
- For each *active* node N in $\mathcal{T}_A$, the selected literal B of N is a positive literal and:
  - The tree $\mathcal{T}_B$ is already marked as completed or $B \in \mathcal{A}$ holds.
  - Each answer B' in $\mathcal{T}_B$ is contained in the answer set $Answer_N$.
- There are no *new* or *floundered* nodes in $\mathcal{T}_A$. Further, there is no *active* node having a negative literal as the selected literal.

□

The computation of the SLG resolution proceeds by applying the following transformations to a search forest.

Definition 11.3.8 (SLG Transformations [CSW95]) Let G be an X-clause of a *new* non-root node N.

(i) NEW ANSWER. If G is an answer clause, the status of N is changed to *answer*.

If G is not an answer clause, let L be the selected literal of G.

(ii) FLOUNDERING. If L is a non-ground negative literal, the status of N is changed to *floundered*.

(iii) NEW ACTIVE. If L is an atom B or a ground negative literal $\neg B$, the status of N is changed to *active* and the set $Answers_N$ is set to $\emptyset$. Furthermore, if there is no tree $\mathcal{T}_B$ for B in the current search forest, it is created, with $B \leftarrow B$ as label for the root node and $instances(P, B)$ being the labels of the child nodes.

Let G be the clause of an active node N, and L be the selected literal of G.

(iv) ANSWER RETURN. If L is an atom B and for some answer clause C for B, of the form $H \leftarrow D|$, H is not in the set $Answers_N$, then H is added to $Answers_N$, and N has a *new* child node labeled by the SLG resolvent of G with C on L if D is empty or by the SLG factor of G with C on L if D is not empty.

Let the selected literal L of the X-clause G of an active node N by a ground negative literal $\neg B$. Then there are three cases. Transformations (v) and (vi) solve the ground negative literal L by negation-as-failure if the corresponding positive subgoal B is either successful or failed. Otherwise, transformation (vii) delays the selected ground negative literal.

(v) NEGATION FAILURE-R. If B has an answer with no delayed literals, the status of N i changed to *disposed*.

(vi) NEGATION SUCCESS-R. If B is completely evaluated without any answers, then N has a *new* child node labeled by G with L deleted, and the status of N is changed to *disposed*.

(vii) DELAYING. Otherwise, N has a new child node labeled by a clause obtained from G by moving L into the sequence of delayed literals, and the status of N is changed to *disposed*.

Subgoals that are completely evaluated are deleted.

(viii) COMPLETION. Let $\mathcal{A}$ be a nonempty set of subgoals that is completely evaluated. Then for each $A \in \mathcal{A}$, every *active* node in the tree $\mathcal{T}_A$ for A is *disposed* and $\mathcal{T}_A$ is marked as *completed*.

Given an arbitrary but fixed computation rule, there are programs in which ground negative literals must be delayed before their truth value is known. Additional transformations are needed for simplifying delayed literals when their truth value is determined.

Let G be an X-clause of a node N that is not *disposed*. Let L be delayed literal in G. Suppose that L is a ground negative literal $\neg B$.

(ix) NEGATION FAILURE-L If B has an answer with no delayed literals, the status of N is changed to *disposed*.

(x) NEGATION SUCCESS-L If B is completely evaluated without any answers, then N has a new child node with the same status as N labeled by G with L deleted, and the status of N is changed to *disposed*.

Suppose that L is an atom.

(xi) ATOM SUCCESS-L If L has an answer with no delayed literals, then N has a new child node with the same status as N labeled by G with L deleted, and the status of N is changed to *disposed*.

(xii) ATOM FAILURE-L If L is completely evaluated without any answers, the status of N is changed to *disposed*.

□

The term *SLG resolution* is used to refer to the process of applying transformations starting with the initial forest of a query atom with respect to a program. Since the Herbrand universe is countable, there is a stage, which may be larger than ω when no transformation can be applied to the search forest of a query. It was shown that when no transformation can be applied to a search forest, either some node in the forest is floundered or every subgoal in the forest is marked as completed. In the latter case, the only nodes that are not disposed in the tree of each subgoal are the root node and the answer nodes.

Theorem 11.3.9 (Correctness of SLG resolution [CSW95]) Let P be a program. Let R be an arbitrary but fixed computation rule. Let A be a query atom. Let P_A be the set of all answer clauses in the final search forest derived from A that has no floundered nodes. Let $BASE(P_A)$ be the Herbrand base of P_A, i.e., set of all ground instances of all atoms in P_A. Then the well-founded model of P restricted to $BASE(P_A)$ coincides with the well-founded model of P_A. □

Example 11.3.10 (SLG Resolution) Consider the following program P from Example 9.1.5 that is also used in [CSW95].

$$
\begin{array}{lll}
q(X) & \leftarrow & p(X). \\
q(a). & & \\
p(X) & \leftarrow & q(X). \\
p(X) & \leftarrow & \mathbf{not}\, r. \\
r & \leftarrow & \mathbf{not}\, s. \\
s & \leftarrow & \mathbf{not}\, r.
\end{array}
$$

Let $A = q(X)$ be the query atom. The initial search forest $\mathcal{F}$ for A contains only the tree $\mathcal{T}_{q(X)}$:

$$
\begin{array}{llllll}
\mathcal{T}_{q(X)} & \textit{new}: & q(X) & \leftarrow & p(X). & \\
 & \textit{new}: & q(a). & & &
\end{array}
$$

Then we can apply NEW ANSWER to the second rule and NEW ACTIVE to activate the body atom $p(X)$ of the first rule. We write the set of used answers to the right of the *active* rule.

$$
\begin{array}{llllll}
\mathcal{T}_{q(X)} & \textit{active}: & q(X) & \leftarrow & p(X). & \{\} \\
 & \textit{answer}: & q(a). & & & \\
\mathcal{T}_{p(X)} & \textit{new}: & p(X) & \leftarrow & q(X). & \\
 & \textit{new}: & p(X). & \leftarrow & \mathbf{not}\, r. &
\end{array}
$$

Then we can activate $q(X)$ and r and then s.

$$
\begin{array}{lllllll}
\mathcal{T}_{q(X)} & \textit{active}: & q(X) & \leftarrow & p(X). & \{\} \\
 & \textit{answer}: & q(a). & & & \\
\mathcal{T}_{p(X)} & \textit{active}: & p(X) & \leftarrow & q(X). & \{\} \\
 & \textit{active}: & p(X). & \leftarrow & \mathbf{not}\, r. & \\
\mathcal{T}_{r} & \textit{active}: & r & \leftarrow & s. & \\
\mathcal{T}_{s} & \textit{active}: & s & \leftarrow & r. &
\end{array}
$$

Now we apply ANSWER RETURN and get the following forest.

$$
\begin{array}{lllllll}
\mathcal{T}_{q(X)} & \textit{active}: & q(X) & \leftarrow & p(X). & \{p(a)\} \\
 & \textit{answer}: & q(a). & & & \\
\mathcal{T}_{p(X)} & \textit{active}: & p(X) & \leftarrow & q(X). & \{q(a)\} \\
 & \textit{answer}: & p(a). & & & \\
 & \textit{active}: & p(X). & \leftarrow & \mathbf{not}\, r. & \\
\mathcal{T}_{r} & \textit{active}: & r & \leftarrow & s. & \\
\mathcal{T}_{s} & \textit{active}: & s & \leftarrow & r. &
\end{array}
$$

Since no other transformation is applicable to this forest, we apply DELAYING to the atoms r and s and produce two new answers.

$$
\begin{array}{llllll}
\mathcal{T}_{r} & \textit{disposed}: & r & \leftarrow & s. & \\
 & \textit{answer}: & r & \leftarrow & s|. & \\
\mathcal{T}_{s} & \textit{disposed}: & s & \leftarrow & r. & \\
 & \textit{answer}: & s & \leftarrow & r|. &
\end{array}
$$

Now the atoms r and s are completely evaluated and can be used to resolve the body literal $\mathbf{not}\, r$. However, the truth values of r and s are undefined. Thus we have to apply DELAYING in the tree $\mathcal{T}_{p(X)}$ and get the following:

$$
\begin{array}{lllllll}
\mathcal{T}_{p(X)} & \textit{active}: & p(X) & \leftarrow & q(X). & \{q(a)\} \\
 & \textit{answer}: & p(a). & & & \\
 & \textit{disposed}: & p(X). & \leftarrow & \mathbf{not}\, r. & \\
 & \textit{answer}: & p(X). & \leftarrow & \mathbf{not}\, r|. &
\end{array}
$$

With this new *answer* for $p(X)$ we can apply ANSWER RETURN for $p(X)$ and then for $q(X)$.

$$
\begin{array}{lllll}
\mathcal{T}_{q(X)} & \textit{active}: & q(X) \leftarrow p(X). & \{p(a), p(X)\} \\
 & \textit{answer}: & q(a). & \\
 & \textit{answer}: & q(X) \leftarrow p(X)|. & \\
\mathcal{T}_{p(X)} & \textit{active}: & p(X) \leftarrow q(X). & \{q(a), q(X)\} \\
 & \textit{answer}: & p(a). & \\
 & \textit{answer}: & p(X) \leftarrow q(X)|. & \\
 & \textit{disposed}: & p(X). \leftarrow \mathbf{not}\, r. & \\
 & \textit{answer}: & p(X). \leftarrow \mathbf{not}\, r|. &
\end{array}
$$

Now also the atoms $q(X)$ and $p(X)$ are completely evaluated, since all *answers* have been used, and we can apply COMPLETION to get the final search forest.

$$
\begin{array}{lll}
\mathcal{T}_{q(X)} & \textit{disposed}: & q(X) \leftarrow p(X). \\
 & \textit{answer}: & q(a). \\
 & \textit{answer}: & q(X) \leftarrow p(X)|. \\
\mathcal{T}_{p(X)} & \textit{disposed}: & p(X) \leftarrow q(X). \\
 & \textit{answer}: & p(a). \\
 & \textit{answer}: & p(X) \leftarrow q(X)|. \\
 & \textit{disposed}: & p(X). \leftarrow \mathbf{not}\, r. \\
 & \textit{answer}: & p(X). \leftarrow \mathbf{not}\, r|. \\
\mathcal{T}_{r} & \textit{disposed}: & r \leftarrow s. \\
 & \textit{answer}: & r \leftarrow s|. \\
\mathcal{T}_{s} & \textit{disposed}: & s \leftarrow r. \\
 & \textit{answer}: & s \leftarrow r|.
\end{array}
$$

The resulting set of answer clauses $P_{q(X)}$ is the following:

$$
\begin{array}{ll}
q(X) \leftarrow & p(X)|. \\
q(a). & \\
p(X) \leftarrow & q(X)|. \\
p(a). & \\
p(X) \leftarrow & \mathbf{not}\, r|. \\
r \leftarrow & \mathbf{not}\, s|. \\
s \leftarrow & \mathbf{not}\, r|.
\end{array}
$$

□

Remark 11.3.11 (Remainder) Note, that the set $P_{q(X)}$ of answer clauses in Example 11.3.10 equals the set R of rules in the final transformation state S for the same program in Example 9.3.4. The only difference is that in $P_{q(X)}$ X-clauses are used instead of normal rules. □

The Example 11.3.10 illustrates that our non-ground transformation and the SLG resolution are strongly related. The set of atoms A for that a tree $\mathcal{T}_A$ exists in a forest $\mathcal{F}$ corresponds to the set G of activated atoms of a non-ground transformation state $S = (G, R, U)$. The set of clauses in a forest that are not *disposed* corresponds to the set R of rules in S.

The relation between our non-ground transformation and the SLG resolution is characterized by the following result.

Theorem 11.3.12 (Non-Ground Transformation and SLG) Let P be a normal program. Let Q be a query atom. Let $\mathcal{F}_0, \mathcal{F}_1, \ldots, \mathcal{F}_n$ a sequence of search forests for an SLG computation for P and Q. If in the final search forest $\mathcal{F}_n$ there are no two different trees $\mathcal{T}_{A_1}$ and $\mathcal{T}_{A_2}$ such that A_1 and A_2 are unifiable, then there is a sequence $S_0, S_1 \ldots, S_n$ of non-ground transformation states such that for each i with $1 \leq i \leq n$ the following conditions hold:

- Each atom A, for which there is a tree $\mathcal{T}_A$ in $\mathcal{F}_i$, is also contained in the set G_i of active goals in S_i.
- The set of X-clauses, that are not *disposed* in all trees in $\mathcal{F}_i$ is equal to the set R_i of rules in the non-ground transformation state S_i, if we consider X-clauses as normal rules.

Proof: We prove the result by induction on n. Assume that $n = 0$. The initial search forest $\mathcal{F}_0$ for P and Q contains one tree $\mathcal{T}_Q$ having the rules of $instances(P, Q)$ as labels of its nodes. This is exactly the set R_0 of rules of the initial transformation state S_0 for P and Q.

Now let the result be true for $\mathcal{F}_n$ and S_n. We show that for any SLG transformation applicable to $\mathcal{F}_n$ and the resulting forest $\mathcal{F}_{n+1}$ there is a sequence of non-ground transformations such that $S_n \rightsquigarrow_X^* S_{n+1}$ holds.

(i) NEW ANSWER. This transformation only changes the state of a node from *new* to *answer*. Our rules are state-less, and the result holds trivially.

(ii) FLOUNDERING. The same holds for this transformation. Whether the rule is floundered or not is checked if an irreducible transformation state is reached.

(iii) NEW ACTIVE. A new tree $\mathcal{T}_B$ is created if B is not yet activated in $\mathcal{F}_n$. The rules in $instances(P, B)$ are added as labels to the nodes of the new tree. In this case B is not activated in S_n, and no atom B' unifiable with B is activated. Thus, we can apply *activation* to S_n to activate B and add the rules $instances(P, B)$.

(iv) ANSWER RETURN. An instance of an active X-clause $A \leftarrow \mathcal{B}$ is added as a *new* node if an atom $B \in \mathcal{B}$ has an answer clause $B' \leftarrow D|$ in $\mathcal{T}_B$. If D is empty, the body literal B is removed from the new instance, otherwise it is shifted to the list of delayed literals. Thus we can apply *instantiation* to the rule $A \leftarrow \mathcal{B}$ w.r.t. the head of the rule $B' \leftarrow D$ in S_n. If D is empty, we can apply *success* to remove the literal from the instance. Otherwise we do not change the instance, since we do not distinguish delayed and non-delayed body literals.

(v) NEGATION FAILURE-R. This transformation changes the status value of a node to *disposed* if the atom occurring in the selected negative body literal of an active node is known to be true. Exactly the same does the *negative reduction* transformation. But instead of setting the status value to *disposed* it removes the rule instance from the set R.

(vi) NEGATION SUCCESS-R. This transformation changes the status value of a node to *disposed* if the atom occurring in the selected negative body literal of an active node is known to be false, and creates a new child labeled with the same X-clause as the parent with the selected literal removed. Exactly the same does the *positive reduction* transformation. But instead of creating a new rule instance, the body literal is removed in place.

(vii) DELAYING. This transformation changes the status value of a node to *disposed* if the atom occurring in the selected body literal is not known to be true or to be false, and creates a new child labeled with the same X-clause as the parent with the selected literal moved into the sequence of delayed literals. This step is not needed for the non-ground transformation, since our rules are state-less and we do no distinguish delayed and non-delayed body literals.

(viii) COMPLETION. This transformation changes the status value of all active nodes for a set $\mathcal{A}$ of atoms to *disposed*, if the set $\mathcal{A}$ is *completely evaluated* (cf. Definition 11.3.7). The same rules can be deleted in S_n by one application of *loop detection*.

Let $A \leftarrow \mathcal{B}$ be any rule whose status value is changed from *active* to *disposed*. According to Definition 11.3.7, all active rules $A \leftarrow \mathcal{B}$ have a positive body literal $B \in \mathcal{B}$ such that $\mathcal{T}_B$ is completed or $B \in \mathcal{A}$ holds, and all answers are used for ANSWER RETURN.

But then $A \leftarrow \mathcal{B}$ is completely instantiated according to Definition 9.2.14 w.r.t. the remaining rules. If B is already completed, then the tree $\mathcal{T}_B$ does not contain *new* or *active* nodes, but only *disposed* and *answer* clauses. All answers have been used for ANSWER RETURN in $\mathcal{F}_n$, i.e., they have been used for *instantiation* in S_n. If $B \in \mathcal{A}$ holds, then all *active* rules for B are *disposed*, and all *answers* have been used for ANSWER RETURN. Thus the condition of *complete instantiation* is satisfied and the rules can be deleted by one application of *loop detection*.

(ix) NEGATION FAILURE-L. This transformation corresponds to *negative reduction* like NEGATION FAILURE-R, since we do not distinguish between delayed literals and non-delayed literals.

(x) NEGATION SUCCESS-L. This transformation corresponds to *positive reduction* like NEGATION SUCCESS-R.

(xi) ATOM SUCCESS-L. This transformation changes the status value of a node to *disposed* if an atom in the sequence of delayed literals is known to be true, and creates a child node labeled with the same X-clause as the parent with the atom removed. This corresponds exactly to the non-ground transformation *success*. But instead of creating a new rule instance, the atom is removed in place.

(xii) ATOM FAILURE-L. This transformation changes the status value of a node to *disposed* if an atom in the sequence of delayed literals is known to be false. This corresponds to the non-ground transformation *failure* that removes a rule instance from the set R if it contains a positive body literal that is false in S_n.

□

The following definition describes the standard strategy of the SLG resolution.

Definition 11.3.13 (SLG Strategy) The *SLG regular strategy expression* is defined by

$$((A^*_{single}\ (I|P|S|N|F|L)^*)^*\ A_{delay})^*.$$

□

In the inner loop of the SLG expression, the activation of at most one literal per rule and the reduction of the generated rules is performed alternatingly. The delay operation is only triggered if no other transformation is applicable. This corresponds to the standard SLG strategy that tries to delay as few literals as possible.

Remark 11.3.14 (Activation of Instances) The most significant difference between SLG and the transformation is that in a search forest we have a separate tree $\mathcal{T}_A$ for each activated atom A with its own set of clauses, whereas in a non-ground transformation state S there is only one global set of rules. In a search tree there can be several trees $\mathcal{T}_{A'}$ for several instances A' of an atom A, each doing a separate computation. In the non-ground transformation it is not allowed to activate an instance A' of another atom A that is already activated. □

Example 11.3.15 (Activation of Instances) Consider the following program P

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

and the query atom $Q = p(X)$. In the initial non-ground transformation state for P and Q we are not allowed to activate the atom $p(f(X))$ since $p(X)$ is already activated. But we can apply *loop detection* to delete the rule from the rule set R and terminate immediately.

In the SLG resolution we have to activate $p(f(X))$, $p(f(f(X)))$, and so on and produce an infinite number of trees. The COMPLETION is not applicable, according to Definition 11.3.7, because it is not possible to activate all subgoals in a finite number of steps. □

Remark 11.3.16 (Activation of Non-Ground Negative Literals) Another difference lies in the fact that in the non-ground transformation it is allowed to activate non-ground negative literals. It may be the case that the literal can be reduced later by *positive reduction* or *negative reduction*. □

Example 11.3.17 (Activation of Non-Ground Negative Literals) Consider the following program P_1:

$$p(X) \leftarrow \mathbf{not}\, q(X).$$

We can activate $p(X)$ and then $q(X)$. Since $q(X)$ is false in that state, we can apply *positive reduction* to derive that $p(X)$ is true w.r.t. the well-founded semantics of P. In SLG, for each ground instance $p(t)$ of $p(X)$ the query $p(t)$ would succeed. Let $Q = p(a)$. Then we get the following final search forest:

$\mathcal{T}_{p(a)}$ *disposed* : $p(a) \leftarrow \mathbf{not}\, q(a).$
answer : $p(a).$

$\mathcal{T}_{q(a)}$

However, for the query $p(X)$ we get a floundered forest:

$\mathcal{T}_{p(X)}$ *floundered* : $p(X) \leftarrow \mathbf{not}\, q(X).$

Now consider another program P_2:

$$p(X) \leftarrow \mathbf{not}\, q(X).$$
$$q(X) \leftarrow \mathbf{not}\, r(X).$$
$$r(a).$$

In the well-founded model $W^*_{P_2}$ of P_2 we have that $p(X)$ is true for $X = a$ and false otherwise. For the query $p(a)$ we get the expected answer. However, for the query $p(X)$ the final non-ground transformation state is floundered and contains all rules from P_2. The reason is, that the non-ground literal $\mathbf{not}\, r(X)$ cannot be reduced. In the remainder there is no way to represent the answer that $q(X)$ is false for $X = a$ and true otherwise. □

Remark 11.3.18 (Parallel Activation) SLG is restricted to select only one body literal of a rule instance at a time. Only negative body literals or literals depending on the latter can be delayed to enable the selection of another body literal. Our transformation approach does not use explicit delaying but treats all remaining body literals equally. While the delaying of negative body literals is necessary to compute the well-founded semantics in SLG, the parallel activation of more than one positive body literal can lead to more efficient search strategies. □

Example 11.3.19 (Parallel Activation) Consider the following program fragment

$$p(X) \leftarrow r(X), s(X).$$
$$r(a) \leftarrow ...$$
$$s(b) \leftarrow ...$$

and the query $Q = p(X)$ where we assume that the goals $r(a)$ and $s(b)$ need a considerable amount of evaluation effort. By the SLG resolution the body literal first selected in the first clause has to be evaluated completely before an answer is returned. Then the other body literal can be selected and will fail.

By our transformation the first clause can be instantiated w.r.t. any body literal using $\rightsquigarrow_I$. Then the original rule instance can be deleted by $\rightsquigarrow_C$ and the instantiated rule can be deleted by $\rightsquigarrow_F$ after having activated the other body literal. Thus the query fails directly without having to evaluate $r(a)$ or $s(b)$. □

Definition 11.3.20 (Parallel Activation Strategy) The *parallel activation regular strategy expression* is defined by

$$(A^*\ (I|P|S|N|F|L)^*)^*.$$

□

This strategy expression is similar to the SLG expression of Definition 11.3.13. The only difference is that we now allow the application of the full *activation* transformation, that is allowed to activate more than one body literal of one rule.

Remark 11.3.21 (Search Space) In Example 11.3.19 we have seen that with our transformation system we have a shorter proof than can be found in SLG resolution. However, adding an inference rule to a system can shorten proofs, but it can also slow down the performance of the system because then it has a larger search space to traverse. Thus, strategies that open up the search space have to be applied carefully. □

11.4 Relation to Bottom-Up Methods

Section 11.3 illustrated, that it is natural to realize top-down strategies using the non-ground transformation framework. In this section we show that the classical bottom-up strategies can be modeled as well.

Lemma 11.4.1 (Transformation Subsumes Definite Bottom-Up Iteration) Let P be a definite range-restricted program. Then there is a sequence

$$S_1 \rightsquigarrow_X^* S_2 \rightsquigarrow_X^* \ldots \rightsquigarrow_X^* S_n$$

of non-ground transformations, such that for each $S_i = (G_1, R_i, U_i)$ for $1 \leq i \leq n$

$$facts(R_i) = T_P \uparrow i$$

holds.

Proof: The proof is by induction on n. Let $n = 1$. Then we have to show

$$facts(R_1) = T_P \uparrow 1 = T_P(\emptyset).$$

Let $S_1 = (G_1, R_1, U_1)$ be the initial non-ground transformation state for P with all predicates activated, such that $R_1 = P$ holds. Then $facts(R_1) = facts(P)$ holds. Since P is range-restricted, we know that $T_P(\emptyset) = facts(P)$ holds. Together the result follows.

Now assume that the result holds for $n-1$. Now consider every ground instance $A' \leftarrow \mathcal{B}'$ of a rule $A \leftarrow \mathcal{B}$ of P such that $\mathcal{B}' \subseteq facts(R_{n-1})$ holds. If $A' \in facts(R_{n-1})$ holds, do nothing. Otherwise apply *instantiation* to the rule $A \leftarrow \mathcal{B} \in R_{n-1}$ for each ground literal $B' \in \mathcal{B}'$, successively until in the result $A'' \leftarrow \mathcal{B}'$ all body literals are ground. $A'' \equiv A'$ is ground, too, since P is range-restricted. Apply *success* to each body literal in $A' \leftarrow \mathcal{B}'$ such that the fact A' is derived. Then $T_P(facts(R_{n-1})) = facts(R_n)$ follows. Together with the induction hypothesis we get $T_P \uparrow n = facts(R_n)$. □

Definition 11.4.2 (Bottom-Up Strategy Expression) The *bottom-up regular strategy expression* is defined by

$$A^*_{all}\ (I|S)^*\ L^*.$$

□

Lemma 11.4.3 (Least Herbrand Model) Let P be a definite range-restricted program. Let e be the bottom-up strategy expression of Definition 11.4.2. If $\mathrm{lfp}(T_P)$ is finite, then

$$P \overset{e}{\mapsto} \mathrm{lfp}(T_P)$$

holds.

Proof: First we compute all atoms in $\mathrm{lfp}(T_P)$ as facts, according to Lemma 11.4.1. Then we apply *loop detection* to remove all rules with a non-empty body. All non-ground rules can be deleted, because they are instantiated completely w.r.t. the set of facts, that are all ground. If a ground rule has only true body literals, then it could be reduced by *success* in the previous step. If a ground rule has a false body literal that is no fact, then the rule is not completely instantiated w.r.t. the set of facts, too. Thus, all rules, that are no facts, can be deleted. □

For definite programs, the result of Lemma 11.4.3 is not surprising. We even know, that every irreducible result for the strategy expression $A^*_{all}(I|S|L)^*$ is equal to $\mathrm{lfp}(T_P)$, since this is the well-founded model of P. More interesting is the fixpoint iteration for normal programs.

Lemma 11.4.4 (Transformation Subsumes Relational Grounding) Let P be a range-restricted normal program. There is a sequence

$$S_1 \rightsquigarrow^*_X S_2 \rightsquigarrow^*_X \ldots \rightsquigarrow^*_X S_n$$

of non-ground transformations, such that for each $S_i = (G_1, R_i, U_i)$ for $1 \leq i \leq n$

$$\bar{T}_P \uparrow i \subseteq R_i$$

holds. If $\mathrm{lfp}(\bar{T}_P)$ is finite, then

$$\mathrm{lfp}(\bar{T}_P) = R_n$$

holds.

Proof: The proof is by induction on n. Let $n = 1$. We have to show

$$\bar{T}_P(\emptyset) \subseteq R_1.$$

Let $S_1 = (G_1, R_1, U_1)$ be the initial non-ground transformation state for P with all predicates activated, such that $R_1 = P$ holds. $\bar{T}_P(\emptyset)$ contains all ground instances

of rules r in P that have no positive body literals (cf. Definition 5.2.1). Since P is range-restricted, these rules r are ground in P, thus they are contained in $R_1 = P$.

Now assume that the result holds for $n-1$. Now consider every ground instance $A' \leftarrow \mathcal{B}'$ of a rule $A \leftarrow \mathcal{B}$ of P such that $pos(\mathcal{B}') \subseteq heads(R_{n-1})$ holds. If $(A' \leftarrow \mathcal{B}') \in R_{n-1}$ holds, do nothing. Otherwise apply *instantiation* to the rule $A \leftarrow \mathcal{B} \in R_{n-1}$ for each ground atom $B' \in pos(\mathcal{B}')$, successively until in the result $A'' \leftarrow \mathcal{B}'$ all positive body literals are ground. The resulting rule is ground and $A'' = A'$ holds, since P is range-restricted and thus each variable occurring in the head A' or in a negative body literals also occurs in a positive body literal. Then $\bar{T}_P(R_{n-1}) \subseteq R_n$ follows. Together with the induction hypothesis we get $\bar{T}_P \uparrow n \subseteq R_n$.

If $\mathrm{lfp}(\bar{T}_P)$ is finite, let the fixpoint be reached at the stage $n-1$, i.e., $\mathrm{lfp}(\bar{T}_P) = \bar{T}_P \uparrow n-1$. Let

$$R' \ := \ R_{n-1} \backslash \mathrm{lfp}(\bar{T}_P).$$

All non-ground rules on R' are instantiated completely w.r.t. the remaining rules: if all positive body literals are unifiable with heads from the remaining rules, the instance has been generated, otherwise it is instantiated completed by definition. If a ground rule in R' has only positive body literals with heads in the remaining rules, then the rule is in $\mathrm{lfp}(\bar{T}_P)$ and thus not in R'. If it has a positive body literal not in the heads of the remaining rules, it is instantiated completely. □

Definition 11.4.5 (Grounding Strategy Expression) The *grounding regular strategy expression* is defined by

$$A^*_{all} \ I^* \ L^*.$$

□

Lemma 11.4.6 (Relevant Ground Instances) Let P be a range-restricted normal program. Let e be the grounding strategy expression of Definition 11.4.5. If $\mathrm{lfp}(\bar{T}_P)$ is finite, then

$$P \stackrel{e}{\mapsto} \mathrm{lfp}(\bar{T}_P)$$

holds.

Proof: The result follows from Lemma 11.4.4. □

The characterization of the bottom-up fixpoint iteration for range-restricted programs has two advantages. First, since only ground facts or ground rule instances are derived, and thus only matching is required instead of unification, efficient evaluation techniques can be applied. Whenever a combined transformation strategy chooses to completely evaluate a range-restricted part of a program, one of these optimized implementations can be used. Second, since the result of such a fixpoint iteration is always ground, efficient evaluation techniques optimized for ground programs can be used to further process the result.

Let us now lift the results from Chapter 6 to the non-ground transformation.

Definition 11.4.7 (Non-Ground AFP Expression) The (non-ground) *AFP regular strategy expression* is defined by

$$A^*_{all} I^* L^* \ ((P|S)^* \ (N|L|F)^*)^*.$$

□

Lemma 11.4.8 (Non-Ground AFP Transformation) Let P be a range-restricted normal program. Let e be the AFP expression of Definition 11.4.7. If the set $\mathrm{lfp}(\bar{T}_P)$ is finite, then the transformation

$$P \overset{e}{\mapsto} \widehat{P}$$

describes the computation the model W^*_P by the alternating fixpoint procedure.

Proof: The result follows from Lemma 11.4.6 and Theorem 6.1.5. □

This strategy expression is a combination of the *bottom-up* expression from Definition 11.4.2, that generates a ground program according to Lemma 11.4.6, and the *AFP expression* that we know already from Example 6.2.10 for ground programs.

Recall from Section 6.1 that the repeated application of the *loop detection* is the reason for the quadratic worst case time complexity of the alternating fixpoint procedure. Thus, we apply the obvious optimization to remove the *loop detection* from the inner loop and get the following strategy expression.

Definition 11.4.9 (Non-Ground Optimized Remainder Expression) The (non-ground) *optimized remainder regular strategy expression* is defined by

$$A^*_{all} I^* L^* \ ((P|S|N|F)^* L^*)^*.$$

□

This expression is composed of the *bottom-up* expression, the *Fitting* expression known from Example 6.2.6 for ground programs, and the *loop detection* in the outer loop. As we have discussed in Section 6.2, all applications of the fitting expression together can be executed in linear time, and the expensive loop detection is applied only if necessary, i.e., if the program does actually contain positive loops.

We have lifted the results from Section 6.2 to the non-ground case to show that the non-ground transformation is able to apply not only the top-down strategies in the style of SLG, that are important for efficient query answering, but also the classical bottom-up strategies, for that optimized implementations are known in order to compute the complete model of a given program. For instance, if the program is range-restricted, the grounding phase can be implemented efficiently using relational techniques like those described in Section 8. The reduction of the ground intermediate result can be implemented efficiently since no unification or matching is needed any more.

We do not lift the results for magic set transformed programs from Section 7.3 to the non-ground case, since the non-ground transformation is goal-directed by itself, and the emulation of a goal-directed evaluation in a goal-directed approach would not be appropriate.

11.5 Mixed Strategies

We have shown that the non-ground transformation approach generalizes many methods known for the computation of the well-founded semantics. We have given regular strategy expressions characterizing the evaluation strategies of those methods. However, the major strength of our approach lies in the fact that we now can combine fragments of several strategies in a flexible way and thus compose new strategies by combining the advantages of several methods.

Example 11.5.1 (Mixed Strategy) Consider the following regular strategy expression:

$$((A_{single}\ I^*L^*\ ((P|S|N|F)^*L^*)^*)^*\ A_{delay})^*$$

This strategy starts with the goal-directed activation of the SLG resolution. In the second step a fragment of the set-oriented bottom-up computation strategy of Definition 11.4.2 is used. The next step is based on the efficient reduction strategy from the smodels approach, given in Definition 11.4.9. The final transformation is the delay operation, that is to be applied only if no other transformation is applicable. This idea of minimizing the number of delays is adopted from the SLG resolution in order to keep the search space as small as possible. □

There are many other situations where the flexible combination of several strategies has been demanded by several researchers. We will give here just a few examples:

- A program consists of several parts such that for different parts different strategies are appropriate. Then in our approach we can attach different strategy expressions to the parts of the program, such that the evaluation engine can apply the optimal strategy for each part.
- There are different types of queries posed to the same program. For selective queries the top-down strategy in the style of SLG will be the best choice. However, if a query requires to compute the complete relation for a predicate, it can be more efficient to restrict the program to all predicates the query depends on and then compute the complete model of this sub-program by applying an efficient bottom-up strategy.
- Whenever during a goal-directed evaluation a sub-query requires the computation of the complete relation of a predicate, the evaluation engine can automatically switch to a bottom-up style strategy and apply the corresponding efficient implementation for this sub-query.
- A program may be based on base relations that are stored in an external system and may change from time to time. Depending on the current extension of the base relations different strategies may be appropriate.
- In a distributed environment several parts of a program may be evaluated at different locations by different processors. Each processor may select the evaluation strategy best suited for its conditions. For instance, processors relying on base relations in external systems should use a set-oriented strategy that is compatible with the page-oriented access methods for the external system. Processors specialized on finding single answers quickly, e.g. for existential queries, should apply a depth-first search strategy processing tuple-at-a-time.

Example 11.5.2 (Experimental Results) Consider again the program of Example 6.2.14. In Figure 11.1 the results of four evaluation strategies can be seen.

- For the *ground model* computation the program has been grounded using the least fixpoint computation according to Lemma 5.2.4, implemented by the relational algebra described in Section 8.1. To the ground program we applied the *optimized remainder* strategy of Example 6.2.12.
- For the *magic* computation we applied the magic set transformation w.r.t. the query $p(a)$, then applied the relational grounding. Finally, we applied the *well-founded remainder* strategy of Definition 7.3.10.

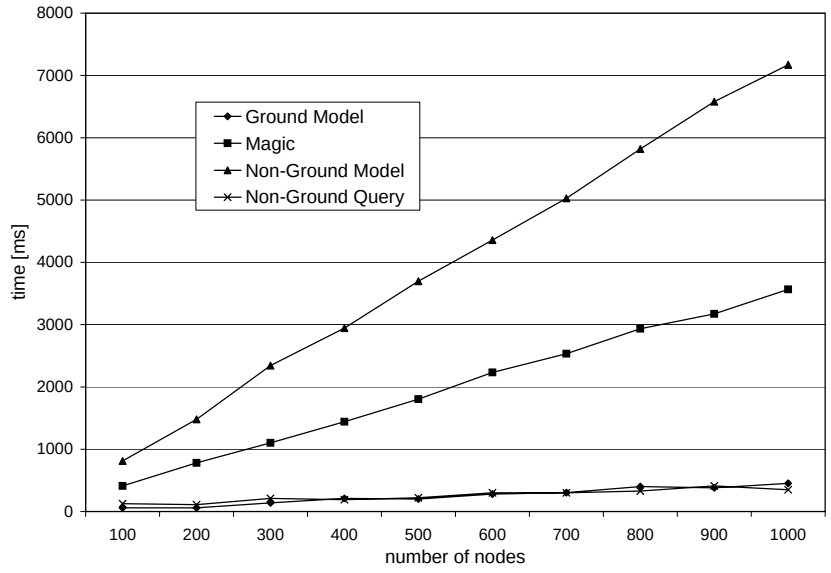


Figure 11.1: Experimental Results: Ground and Non-Ground Strategies

- For the *non-ground model* computation we applied the standard *SLG strategy* of the non-ground transformation for the query $p(X)$, using the queue based implementation described in Section 10.1.
- Finally, for the *non-ground query* computation we applied the same *SLG strategy* for the query $p(a)$.

For this example we get the result, that the model computation by relational grounding and ground transformation is much more efficient than by non-ground transformation for the query $p(X)$. One possible reason for this result is, that by the relational grounding only matching operations have to be performed and all the joins are applied to sets of ground rules. For the transformation the efficient implementation described in Section 8.2 was used. For the query $p(X)$ in the non-ground transformation many applications of the *instantiation* transformation have to be applied. Many unification operations are necessary for this strategy.

However, for the goal-directed query answering, in this example for the query $p(a)$, the goal-directed non-ground transformation is much more efficient then the magic set based ground transformation. In this example the overhead for computation of the additional magic predicates is higher than the optimization by the magic filters. □

The experiment of Example 11.5.2 illustrates, that depending on the application, different strategies can achieve significantly different results. As we expected, the relational grounding connected with the ground transformation is very efficient for the model computation. However, for applications that need only a fragment of the well-founded model, the goal-directed non-ground transformation is more appropriate.

Chapter 12

Conclusion

In this work we have presented a framework that generalizes all major computation approaches for the well-founded semantics by using a common data structure and provides a common language to describe their evaluation strategy. Our framework is based on a set of program transformations. In the first part of this work we presented a rewriting system for ground programs. In the second part we have extended the ground transformation to a goal-directed computation scheme for non-ground programs.

Among others, we have shown the following results:

- The ground transformation system is strongly terminating and confluent. I.e., by applying the transformations to a given program in any order, the uniquely determined normalform is always reached.
- As opposed to the residual program approach, that may get exponential, we guarantee that our transformation system can be applied in polynomial time w.r.t. the program size.
- We have shown that the computation performed by our program transformation does not need more time than the execution of the alternating fixpoint procedure for this program. For many programs for which the alternating fixpoint procedure needs quadratic time, the transformation method is linear.
- Our approach is also applicable to magic set transformed programs and subsumes and even optimizes known magic set extensions of the alternating fixpoint procedure like the well-founded magic sets method and the magic alternating fixpoint procedure.
- Although the non-ground transformation is not confluent, any irreducible (and non-floundered) transformation state for a program and a query contains a fragment of the well-founded model that is sufficient to answer the query correctly.

- We have shown that for a large class of computations of the SLG resolution the non-ground transformation is able to perform exactly the same computation. Further, we have shown that by our non-ground transformation queries can be answered, for that the SLG resolution gets floundered or does not terminate.
- We have defined a regular-expression-like syntax to declaratively describe evaluation strategies just by specifying orders of program transformations. We have shown that many of the known algorithms can be specified as a special case of our transformation system.
- We have provided algorithms that allow to efficiently implement the transformation framework.
- We have illustrated by experimental results, that the theoretical comparisons carry over to the prototype implementation.

Although we have implemented the transformation using an efficient queue based algorithm, we do not intend to completely replace existing implementations. Our implementation is a good tool to apply different strategies to the same application and find an optimal strategy expression in this way. However, if fragments of the final strategy expression coincide with a strategy for which specialized implementations exist, then these implementations can be applied to execute the corresponding part of the strategy. We have illustrated that for instance the computation of the complete model for a program can be more efficient, if a relational implementation of the grounding algorithm in combination with the efficient queue based ground transformation is used. If a tuple-oriented top-down evaluation is required as part of another strategy, the SLG resolution is a good candidate. Another algorithm for the separate evaluation of subgoals using relations and magic set techniques is described in [ZF96]. The strength of our framework lies in the possibility to specify a mixed strategy within one application and let the optimizer identify predefined strategy expressions and call specialized implementations for these fragments.

The framework allows also to prove the correctness of combined implementations: whenever a part of the algorithm can be described as a sequence of transformations, possibly using our regular expression syntax, the correctness of the complete system follows easily. It remains only to show that an irreducible state in the sense of the transformation system will be reached, i.e., that all transformations that are applicable are applied eventually.

The results of this work are already used as a basis for the ULTRA logic language [WF97, WFF98, FWF00, Wic00], where updates and transactions are defined based on the well-founded semantics.

One topic of ongoing work will be the extension of the transformation framework to programs with aggregation [VG92, Lef94, KR98]. Aggregation is a concept that is

useful for many practical applications, and has many properties in common with the concept of negation. There are also extensions of the well-founded semantics [DO97, OJ97, OJ99] suited for programs with aggregation that should be considered in this context.

This also concerns the new SQL 1999 standard [ISO99, EM99]. It describes a relational query language containing recursion, negation, and aggregation. For the evaluation of such queries containing recursion and negation, all the results of this thesis may be applicable. Thus, if these concepts can be brought together, the result will be of a major practical relevance.

Bibliography

[AB94] Krzysztof R. Apt and Roland N. Bol. Logic programming and negation: A survey. *The Journal of Logic Programming*, 19/20:9–71, 1994.

[AD95] Chandrabose Aravindan and Phan Minh Dung. On the correctness of unfold/fold transformation of normal and extended logic programs. *The Journal of Logic Programming*, 24(3):201–217, 1995.

[AHU87] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1987.

[BD94] Stefan Brass and Jürgen Dix. A disjunctive semantics based on unfolding and bottom-up evaluation. In Bernd Wolfinger, editor, *Innovationen bei Rechen- und Kommunikationssystemen*, (IFIP '94-Congress, Workshop FG2: Disjunctive Logic Programming and Disjunctive Databases), pages 83–91, Berlin, 1994. Springer.

[BD95] Stefan Brass and Jürgen Dix. Characterizations of the stable semantics by partial evaluation. In Anil Nerode, editor, *Logic Programming and Nonmonotonic Reasoning, Proc. of the Third Int. Conf. (LPNMR'95)*, number 928 in LNCS, pages 85–98. Springer, 1995.

[BD97] Stefan Brass and Jürgen Dix. Characterizations of the Disjunctive Stable Semantics by Partial Evaluation. *Journal of Logic Programming*, 32(3):207–228, 1997. (Extended abstract appeared in: Characterizations of the Stable Semantics by Partial Evaluation *LPNMR, Proceedings of the Third International Conference, Kentucky*, pages 85–98, 1995. LNCS 928, Springer.).

[BD98a] Stefan Brass and Jürgen Dix. Characterizations of the Disjunctive Well-founded Semantics: Confluent Calculi and Iterated GCWA. *Journal of Automated Reasoning*, 20(1):143–165, 1998. (Extended abstract appeared in: Characterizing D-WFS: Confluence and Iterated GCWA. *Logics in Artificial Intelligence, JELIA '96*, pages 268–283, 1996. Springer, LNCS 1126.).

[BD98b] Stefan Brass and Jürgen Dix. Semantics of (Disjunctive) Logic Programs Based on Partial Evaluation. *Journal of Logic Programming*, accepted for publication, 1998. (Extended abstract appeared in: Disjunctive Semantics Based upon Partial and Bottom-Up Evaluation, *Proceedings of the 12-th International Logic Programming Conference, Tokyo*, pages 199–213, 1995. MIT Press.).

[BDFZ01] Stefan Brass, Jürgen Dix, Burkhard Freitag, and Ulrich Zukowski. Transformation-based bottom-up computation of the well-founded model. *Theory and Practice of Logic Programming*, to appear, 2001.

[BFZ97] Stefan Brass, Burkhard Freitag, and Ulrich Zukowski. Transformation Based Bottom-Up Computation of the Well-Founded Model. In Jürgen Dix, Luis Moniz Pereira, and Teodor C. Przymusinski, editors, *Nonmonotonic Extensions of Logic Programming*, LNAI 1216, pages 171–201. Springer, Berlin, 1997.

[BPRM91] Isaac Balbin, Graeme S. Port, Kotagiri Ramamohanarao, and Krishnamurthy Meenakshi. Efficient bottom-up computation of queries on startified databases. *The Journal of Logic Programming*, 11:295–344, 1991.

[BR86a] Isaac Balbin and Kotagiri Ramamohanarao. A differential approach to query optimisation in recursive databases. Technical Report 86/7, University of Melbourne, Dep. of Computer Science, Parkville, Australia, 1986.

[BR86b] Francois Bancilhon and Raghu Ramakrishnan. An amateur's introduction to recursive query processing. In Carlo Zaniolo, editor, *Proc. of SIGMOD'86*, pages 16–52, 1986.

[BR87] Isaac Balbin and Kotagiri Ramamohanarao. A generalization of the differential approach to recursive query evaluation. *Journal of Logic Programming*, 4(3), 1987.

[BR91] Catriel Beeri and Raghu Ramakrishnan. On the power of magic. *The Journal of Logic Programming*, 10:255–299, 1991.

[Bry89] François Bry. Logic programming as constructivism: A formalization and its application to databases. In *Proc. of the 8$^{\text{th}}$ ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'89)*, pages 34–50, 1989.

[Bry90] François Bry. Negation in logic programming: A formalization in constructive logic. In Dimitris Karagiannis, editor, *Information Systems and Artificial Intelligence: Integration Aspects*, number 474 in LNCS, pages 30–46. Springer, 1990.

[BSF95] Kenneth A. Berman, John S. Schlipf, and John V. Franco. Computing the Well-Founded Semantics Faster. In A. Nerode, W. Marek, and M. Truszczyński, editors, *Logic Programming and Non-Monotonic Reasoning, Proceedings of the Third International Conference*, LNCS 928, pages 113–126, Berlin, June 1995. Springer.

[CSW95] Weidong Chen, Terrence Swift, and David S. Warren. Efficient top-down computation of queries under the well-founded semantics. *Journal of Logic Programming*, 24(3):161–199, 1995.

[CW93] Weidong Chen and David S. Warren. Query evaluation under the well founded semantics. In *Proc. 12th ACM SIGACT–SIGMOD–SIGART Conference on the Principles of Database Systems (PODS'93)*, 1993.

[CW96] Weidong Chen and David S. Warren. Tabled evaluation with delaying for general logic programs. *Journal of the ACM*, 43(1):20–74, 1996.

[DG84] William F. Dowling and Jean H. Gallier. Linear-time algorithms for testing the satisfiability of propositional horn formulae. *Journal of Logic Programming*, 1(3):267–284, 1984.

[DK89a] Phan Minh Dung and Kanchana Kanchansut. A fixpoint approach to declarative semantics of logic programs. In *Proc. North American Conference on Logic Programming (NACLP'89)*, pages 604–625, 1989.

[DK89b] Phan Minh Dung and Kanchana Kanchansut. A natural semantics of logic programs with negation. In *Proc. of the Ninth Conf. on Foundations of Software Technology and Theoretical Computer Science*, pages 70–80, 1989.

[DM94] Jürgen Dix and Martin Müller. Partial evaluation and relevance for approximations of the stable semantics. In Z.W. Ras and M. Zemankova, editors, *Proc. of the 8th Int. Symp. on Methodologies for Intelligent Systems*, LNAI. Springer, 1994. To appear.

[DO97] Jürgen Dix and Mauricio Osorio. On Well-Behaved Semantics Suitable for Aggregation. In Jan Maluszynski, editor, *Logic Programming: Proceedings of the 1997 International Symposium*, page 505. MIT Press, 1997.

[EM99] Andrew Eisenberg and Jim Melton. SQL: 1999, formerly known as SQL 3. *SIGMOD Record*, 28(1):131–138, 1999.

[EN94] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems*. Benjamin/Cummings, Redwood City, California, 1994.

[Fit85] Melvin C. Fitting. A Kripke-Kleene Semantics of logic Programs. *Journal of Logic Programming*, 4:295–312, 1985.

[Fre00] Burkhard Freitag. Deduktive Datenbanken. Script accompanying the lecture in the winter term 1999/2000, University of Passau, 2000.

[FSW98] Juliana Freire, Terrance Swift, and David S. Warren. Beyond depth-first strategies: Improving tabled logic programs through alternative scheduling. *Journal of Functional and Logic Programming*, 1998(3), 1998.

[FW97] Juliana Freire and David S. Warren. Controlling the search in tabled evaluations. In *Proceedings 14th International Logic Programming Symposium, Port Jefferson, NY*, page 409, 1997.

[FWF00] Alfred Fent, Carl-Alexander Wichert, and Burkhard Freitag. Logical update queries as open nested transactions. In G. Saake, K. Schwarz, and C. Türker, editors, *Transactions and Database Dynamics*, volume 1773 of *LNCS*, pages 46–67. Springer-Verlag, Berlin, Germany, 2000.

[GKB87] Ulrich Güntzer, Werner Kießling, and Rudolf Bayer. On the evaluation of recursion in (deductive) database systems by efficient differential fixpoint iteration. In *Proceedings of the 3rd International Conference on Data Engineering, Los Angeles, Ca.*, pages 120–129, 1987.

[GL88] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In Robert A. Kowalski and Kenneth A. Bowen, editors, *Logic Programming, Proc. of the 5th Int. Conf. and Symp.*, pages 1070–1080, Cambridge, Mass., 1988. MIT Press.

[ISO99] ISO/IEC. *Information technology – Database languages – SQL. ISO Standard 9075, Parts 1–5*. ISO, 1999.

[KKTG95] Gerhard Küstler, Werner Kießling, Helmut Thöne, and Ulrich Güntzer. Fixpoint iteration with subsumption in deductive databases. *Journal of Intelligent Information Systems*, 4(2):123–148, 1995.

[KR98] David B. Kemp and Kotagiri Ramamohanarao. Efficient recursive aggregation and negation in deductive databases. *IEEE Transactions on Knowledge and Data Engineering*, 10(5):727–745, 1998.

[KSS95] David B. Kemp, Divesh Srivastava, and Peter J. Stuckey. Bottom-up evaluation and query optimization of well-founded models. *Theoretical Computer Science*, 146:145–184, 1995.

[Lef94] Alexandre Lefebvre. Towards an efficient evaluation of recursive aggregates in deductive databases. *New Generation Computing*, 12(2):131–160, 1994.

[Llo87] John Wylie Lloyd. *Foundations of Logic Programming*. Springer-Verlag, Berlin, 2nd edition, 1987.

[LT00] Zbigniew Lonc and Miroslaw Truszczynski. On the problem of computing the well-founded semantics. In *Proc. of the First International Conference on Computational Logic (CL2000). 24th to 28th July, 2000, Imperial College, London, UK*, 2000.

[Mor96] Shinichi Morishita. An extension of Van Gelder's alternating fixpoint to magic programs. *Journal of Computer and System Sciences*, 52:506–521, 1996.

[MT91] Wiktor Marek and Miroslaw Truszczynski. Autoepistemic logic. *Journal of the ACM*, 38(3):588–619, 1991.

[MW88] David Maier and David S. Warren. *Computing with Logic — Logic Programming with Prolog*. Benjamin Cummings, Menlo Park, CA, 1988.

[NS96] Ilka Niemelä and Patrik Simons. Efficient implementation of the well-founded and stable model semantics. Technical Report 7/96, Universität Koblenz-Landau, 1996.

[OJ97] Mauricio Osorio and Bharat Jayaraman. Aggregation and WFS$^+$. In J. Dix, L. Pereira, and T. Przymusinski, editors, *Nonmonotonic Extensions of Logic Programming*, LNAI 1216, pages 71–90. Springer, Berlin, 1997.

[OJ99] Mauricio Osorio and Bharat Jayaraman. Aggregation and negation-as-failure. *New Generation Computing*, 17(3):255–284, 1999.

[PP88] Halina Przymusinska and Teodor C. Przymusinski. Weakly perfect model semantics for logic programs. In Robert A. Kowalski and Kenneth A. Bowen, editors, *Logic Programming, Proc. of the 5th Int. Conf. and Symp.*, pages 1106–1122, Cambridge, Mass., 1988. MIT Press.

[Prz88] Teodor C. Przymusinski. Perfect model semantics. In Robert A. Kowalski and Kenneth A. Bowen, editors, *Logic Programming, Proc. of the 5th Int. Conf. and Symp.*, pages 1081–1096, Cambridge, Mass., 1988. MIT Press.

[Prz89] Teodor Przymusinski. On the declarative and procedural Semantics of logic Programs. *Journal of Automated Reasoning*, 5:167–205, 1989.

[Prz90] Teodor Przymusinski. Well-founded Semantics Coincides With Three-Valued Stable Semantics. *Fundamenta Informaticae*, XIII:445–463, 1990.

[Prz91] Teodor Przymusinski. Three-valued non-monotonic formalisms and semantics of logic programs. *Artificial Intelligence*, 49:309–343, 1991. (Extended abstract appeared in: Three-valued non-monotonic formalisms and logic programming. *Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning (KR'89), Toronto, Canada*, pages 341–348, Morgan Kaufmann, 1989.).

[Ram91] Raghu Ramakrishnan. Magic templates: A spellbinding approach to logic programs. *Journal of Logic Programming*, 11(3/4):189–216, October/November 1991.

[Ros94] Kenneth A. Ross. Modular stratification and magic sets for datalog programs with negation. *Journal of the ACM*, 41(6):1216–1266, 1994.

[RRS+99] I. V. Ramakrishnan, Prasad Rao, Konstantinos Sagonas, Terrance Swift, and David S. Warren. Efficient access mechanisms for tabled logic programs. *Journal of Logic Programming*, 38(1):31–54, 1999.

[RSS+97] Prasad Rao, Konstantinos Sagonas, Terrance Swift, David S. Warren, and Juliana Freire. XSB: A System for Efficiently Computing Well-Founded Semantics. In J. Dix, U. Furbach, and A. Nerode, editors, *Logic Programming and Non-Monotonic Reasoning, Proceedings of the Fourth International Conference*, LNAI 1265, pages 430–440, Berlin, June 1997. Springer.

[Sek93] Hirohisa Seki. Unfold/fold transformation of general logic programs for the well-founded semantics. *The Journal of Logic Programming*, 16(1):5–23, 1993.

[SKGB87] Helmut Schmidt, Werner Kießling, Ulrich Güntzer, and Rudolf Bayer. Compiling exploratory and goal-directed deduction into sloppy delta-iteration. In *Proceedings of the 4th Symposium on Logic Programming, San Francisco, Ca.*, pages 234–243, 1987.

[SS94] Chiaki Sakama and Hirohisa Seki. Partial deduction of disjunctive logic programs: A declarative approach. In *Fourth Int. Workshop on Logic Program Synthesis and Transformation (LOPSTR'94)*, LNCS. Springer, 1994.

[SS97] Peter J. Stuckey and S. Sudarshan. Well-founded ordered search: Goal-directed bottom-up evaluation of well-founded models. *The Journal of Logic Programming*, 32(3):171–205, 1997.

[SSW94] Konstantinos Sagonas, Terrance Swift, and David S. Warren. XSB as an efficient deductive database engine. In Richard T. Snodgrass and Marianne Winslett, editors, *Proc. of the 1994 ACM SIGMOD Int. Conf. on Management of Data (SIGMOD'94)*, pages 442–453, 1994.

[SSW00] Konstantinos Sagonas, Terrance Swift, and David S. Warren. An abstract machine for efficiently computing queries to well-founded models. *Journal of Logic Programming*, 45(1–3):1–41, 2000.

[Tar72] Robert E. Tarjan. Depth first search and linear graph algorithms. *SIAM Journal on Computing*, 1:146–160, 1972.

[Ull88] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems, Vol. 1*. Computer Science Press, 1988.

[Ull89] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems, Vol. 2*. Computer Science Press, 1989.

[VG89] Allen Van Gelder. The alternating fixpoint of logic programs with negation. In *Proc. of the Eighth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'89)*, pages 1–10, 1989.

[VG92] Allen Van Gelder. The well-founded semantics of aggregation. In *Proc. 11th ACM SIGACT–SIGMOD–SIGART Conference on the Principles of Database Systems (PODS'92)*, 1992.

[VG93] Allen Van Gelder. The alternating fixpoint of logic programs with negation. *Journal of Computer and System Sciences*, 47(1):185–221, 1993.

[VGRS91] Allen Van Gelder, Kenneth A. Ross, and John S. Schlipf. The well-founded semantics for general logic programs. *Journal of the Association for Computing Machinary (JACM)*, 38:620–650, 1991.

[WF97] Carl-Alexander Wichert and Burkhard Freitag. Capturing database dynamics by deferred updates. In *Proc. of the 1997 International Conference on Logic Programming (ICLP'97). July 8 – 12, 1997, Leuven, Belgium*, pages 226–240. MIT Press, 1997.

[WFF98] Carl-Alexander Wichert, Burkhard Freitag, and Alfred Fent. Logical transactions and serializability. In B. Freitag, H. Decker, M. Kifer, and A. Voronkov, editors, *Transactions and Change in Logic Databases*, volume 1472 of *LNCS*. Springer-Verlag, Berlin, Germany, 1998. to appear.

[Wic00] Carl-Alexander Wichert. *ULTRA – A Logic Transaction Programming Language*. Dissertation, University of Passau, June 2000.

[ZBF97] Ulrich Zukowski, Stefan Brass, and Burkhard Freitag. Improving the alternating fixpoint: The transformation approach. In Jürgen Dix, Ulrich Furbach, and Anil Nerode, editors, *Logic Programming and Nonmonotonic Reasoning*, LNAI 1265, pages 40–59, Berlin, 1997. Springer.

[ZF96] Ulrich Zukowski and Burkhard Freitag. Adding flexibility to query evaluation for modularly stratified databases. In *Proc. of the 1996 Joint International Conference and Symposium on Logic Programming (JICSLP'96). September 2 – 6, 1996, Bonn, Germany*, pages 304–319. MIT Press, 1996.

[ZF99] Ulrich Zukowski and Burkhard Freitag. Well-founded semantics by transformation: The non-ground case. In *Proc. of the International Conference on Logic Programming (ICLP '99), November 29 - December 4, 1999, Las Cruces, New Mexico*, 1999.

[ZFB96] Ulrich Zukowski, Burkhard Freitag, and Stefan Brass. Transformation-based bottom-up computation of the well-founded model. Technical Report MIP-9620, Universität Passau, 1996.

Index