FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

On Transforming Answer-Set Programs Towards Natural-Language Representations

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Computational Intelligence

eingereicht von

Benjamin Kiesl
Matrikelnummer 1127227

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: a.o. Univ.-Prof. Dr. Hans Tompits
Mitwirkung: Dr. Peter Schüller

Wien, 14.12.2014

(Unterschrift Verfasser) (Unterschrift Betreuung)

Erklärung zur Verfassung der Arbeit

Benjamin Kiesl
Jungstraße 8/23, 1020 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum) (Unterschrift Verfasser)

Any fool can have an opinion;
to know what one needs to know
to have an opinion is wisdom;
which is another way of saying
that wisdom means knowing what
questions to ask about knowledge.

— Neil Postman

Danksagung

Zuallererst möchte ich mich bei meinen beiden Betreuern, Hans Tompits und Peter Schüller, bedanken, welche beide einen beträchtlichen Zeitaufwand in die Entstehung dieser Arbeit investiert haben. Hans Tompits hat mich nicht nur inhaltlich mit vielen hilfreichen Ratschlägen unterstützt, sondern mir auch entscheidend dabei geholfen, meinen Sinn für die sprachlichen und stilistischen Feinheiten zu schärfen. Peter Schüller hatte die ursprüngliche Idee zu dieser Arbeit. Er hat mich aus der Ferne durch sein stets ausführliches Feedback und die prompte Beantwortung all meiner Fragen in einer Art und Weise betreut, die keine Wünsche offen lässt.

Neben der fachlichen Hilfestellung zählen aber vor allem die Menschen, die einem am nächsten stehen. Mein Dank geht deshalb an meine Eltern, Gertrud und Gerhard, welche wohl den größten Anteil an meiner persönlichen Entwicklung haben und auf deren Unterstützung ich mich immer verlassen kann. Außerdem möchte ich mich bei meinen Brüdern Alexander und Jochen für die vielen lustigen gemeinsamen Momente bedanken. Besonderer Dank gilt an dieser Stelle auch meinem Taufpaten Josef Käfer und meiner Taufpatin Britta Käfer, die leider während der Entstehung dieser Arbeit, viel zu jung, nach schwerer Krankheit von uns gegangen ist.

Diese Arbeit wäre wohl nicht zustande gekommen, hätten mich nicht meine Freundinnen und Freunde oftmals erfolgreich davon abgelenkt. Ihnen allen gebührt einerseits größter Dank dafür, dass sie in stressigen Momenten Nachsicht mit mir hatten und andererseits Hochachtung dafür, dass sie mein teils überschwängliches Gemüt ertragen oder gar mitunter als angenehm empfinden können. Besondere Erwähnung soll hier mein Kommilitone Gerald Berger finden. Mit ihm gemeinsam habe ich nicht nur im Laufe meines Studiums zahlreiche Lehrveranstaltungen besucht, sondern auch den Fortschritt dieser Arbeit intensiv diskutiert, wobei er mir oftmals mit nützlichen Tipps weiterhelfen konnte.

Zu guter Letzt geht mein Dank an den wichtigsten Menschen in meinem Leben, Sarah Reiter, dafür, dass sie nun schon seit so vielen Jahren an meiner Seite ist und ich mich während der Entstehung dieser Arbeit stets auf ihren so wichtigen persönlichen Rückhalt verlassen konnte.

Abstract

Answer-set programming is a logic-programming formalism which has proven to be exceptionally well-suited for knowledge representation, reasoning, and solving complex search problems. Its fully declarative nature together with its simple and clear syntax allow for the formulation of compact problem specifications. While many answer-set programs may be relatively easy to understand for a programmer, their meaning may be rather hard to grasp for persons without a background in logic programming. Since many real world problems from various domains can be formulated within answer-set programming, it would be useful to translate answer-set programs into a form which is easier to understand and closer to natural language.

In this thesis, we introduce some results which should help making such a translation possible. First, we analyse the structure of a given program: Most answer-set programs follow the so-called *generate-define-test* paradigm. Within this paradigm, a program is seen to consist of three different parts, called *generate*, *define*, and *test*. We introduce a formal definition of these three parts and develop an algorithm which classifies the rules of a given program accordingly. For the implementation of this algorithm, we make use of techniques from meta-programming, meaning that we write a logic program which operates on other logic programs. Following this, we also discuss how the rules of a program should be ordered before they are translated into some form of natural language.

A straightforward translation of program rules into a form of natural language may yield rather clumsy results in general. One reason for this is that many real-world programs use more than one rule to define something which a human would explain within a single sentence. We thus introduce a generalisation of the well-known *partial evaluation* transformation which helps us to transform a given program into another program that is easier to translate. An important requirement for such a transformation is that it preserves equivalence, i.e., that it does not change the meaning of the original program. To this end, we use a notion of equivalence called *relativised uniform equivalence* which is stronger than ordinary equivalence but weaker than uniform equivalence. As a key result, we prove that our transformation preserves relativised uniform equivalence, which yields the straightforward corollary that it preserves ordinary equivalence over the answer-set semantics too.

Kurzfassung

Die Antwortmengenprogrammierung ist eine Form der logischen Programmierung, welche sich als besonders geeignet für die Repräsentation und Verarbeitung von Wissen sowie für die Lösung komplexer Suchprobleme erwiesen hat. Eine einfache Syntax in Verbindung mit vollständiger Deklarativität ermöglichen die kompakte Formalisierung von Problemspezifikationen. Doch obwohl viele Antwortmengenprogramme für Programmiererinnen und Programmierer recht leicht verständlich sein mögen, kann es für Personen ohne ein entsprechendes Hintergrundwissen durchaus Probleme bereiten, deren Bedeutung zu erfassen. Es erscheint uns daher nützlich, diese Programme in eine einfacher verständliche Form zu übersetzen, die sich näher an der natürlichen Sprache orientiert.

Wir präsentieren deshalb in dieser Arbeit eine Reihe von Ergebnissen, welche die Umsetzung einer derartigen Übersetzung ermöglichen sollen. Zuallererst werden wir ein Verfahren zur Analyse von Programmen entwickeln: Die Struktur der meisten Antwortmengenprogramme entspricht dem sogenannten *Generate-Define-Test*-Modell. Entsprechend dieses Modells besteht ein Programm aus drei verschiedenen Teilen, welche als *Generate*-, *Define*- und *Test*-Teil bezeichnet werden. Wir werden eine formale Definition dieser drei Teile einführen und einen Algorithmus entwickeln, der die Regeln eines gegebenen Programms entsprechend dieser Definition klassifiziert. Für die anschließende Implementierung dieses Algorithmus werden wir von Methoden der Metaprogrammierung Gebrauch machen, da die größten Teile des Algorithmus durchaus elegant als Antwortmengenprogramm formuliert werden können. Danach werden wir noch eine Reihenfolge zur Beschreibung von Programmregeln vorschlagen, welche eine möglichst verständliche Erläuterung zur Folge haben soll.

Die naive Übersetzung einzelner Programmregeln in eine natürlichsprachliche Form kann zu unbefriedigenden Resultaten führen. Ein Grund dafür ist die Tatsache, dass viele praktische Programme mehrere verschiedene Regeln verwenden, um Dinge zu beschreiben, die man in natürlicher Sprache durch einen einzelnen Satz ausdrücken würde. Wir werden deshalb eine Verallgemeinerung der sogenannten *Partial-Evaluation-Transformation* einführen, welche uns dabei helfen kann, Programme in eine verständlicher übersetzbare Form zu transformieren. Es ist für eine derartige Transformation entscheidend, dass sie äquivalenzerhaltend ist, dass sie also die ursprüngliche Bedeutung eines Programms nicht verändert. Wir werden deshalb die sogenannte *relativierte uniforme Äquivalenz*, welche stärker als der konventionelle Äquivalenzbegriff, aber schwächer als die *uniforme Äquivalenz* ist, verwenden und beweisen, dass unsere Transformation diese Form der Äquivalenz präserviert. Dieses Ergebnis liefert dann das Korollar, dass unsere Transformation auch die schwächere Form der konventionellen Äquivalenz präserviert.

Contents

CHAPTER 1

Introduction

One of the major problems within computer science is the search for programming languages which allow for an adequate specification of problems so that computers can be used for determining their solutions. This search lead to the development of *imperative* programming: In this kind of programming, a programmer specifies *how* a certain problem is to be solved by writing a sequence of statements which describes the according computation process. While imperative programs can be efficiently applied to various tasks, there exist many interesting real-world problems for which imperative programs tend to be rather large and hard to understand.

Because of this, several *declarative* programming languages where suggested in the past [56]. In contrast to imperative programs, declarative programs are meant to state *what* is computed rather than *how* it is to be computed, involving that the actual solution computation is delegated to an underlying computational formalism which is transparent to the programmer [43]. One important approach developed within this context is the theory of *logic programming* which aims for using methods from mathematical logic to specify and solve computational problems and which led to the development of *answer-set programming*, a logic-programming paradigm that attracts a lot of attention nowadays.

A wide range of problems from areas such as artificial intelligence or combinatorial optimisation can be elegantly solved within answer-set programming, and its close relation to nonmonotonic logics allows for the compact representation of knowledge and real-world-reasoning tasks. But, although many answer-set programs may be relatively easy to understand for programmers, people without a background in logic programming may find it hard to grasp their meaning.

1.1 Aim of the Work

We consider it thus useful to transform answer-set programs into a form which is easier to understand and closer to natural language. This could benefit many people such as domain experts who are involved in the creation of knowledge-based systems or users of a system who could

obtain a comprehensible explanation of its behaviour. But programmers themselves could also profit from it, as it could help them to quicker gain an understanding of a given program. In this thesis we thus want to develop some results which should help making such a transformation, which is obviously far from trivial, possible.

Most answer-set programs follow the so-called *generate-define-test* paradigm. Within this paradigm, a program – which is, roughly speaking, nothing else than a set of logical rules – is seen to consist of three different parts, called *generate*, *define*, and *test part*, respectively. The generate part non-deterministically generates a set of candidate solutions from which the test part eliminates those candidates which are not the desired solutions of the specified problem. In order to do so, both the generate and the test part may use concepts which are specified in the define part. Classifying the rules of a given program into this trichotomy is thus crucial for obtaining a reasonable explanation of the whole program.

Once the rules of a program are successfully classified, we want to preprocess them in order to simplify a translation to some form of natural language. This is necessary, since a simple rule-by-rule translation of a given program may yield rather clumsy results in general. Consider as an example the following rules of a program which can be found in a textbook [2]:

$$\begin{aligned} a\_row\_is\_not\_filled &\leftarrow row(X), \text{not } row\_is\_filled(X), \\ &\leftarrow a\_row\_is\_not\_filled. \end{aligned}$$

If we tried to directly translate these two rules into some form of natural language, we could end up with the following explanation: "*It may not be the case that a row is not filled. A row is not filled if there exists a row which is not filled.*" One may agree that this translation sounds rather unnatural. The problem here is, that the program uses more than one rule to define something which a human speaker would explain in one sentence. If we could transform the above two rules into one single rule which has the same meaning, we could possibly obtain a better translation. Consider therefore the following rule:

$$\leftarrow row(X), \text{not } row\_is\_filled(X).$$

This rule could be explained by the following, arguably clearer, sentence: "*It may not be the case that there exists a row which is not filled.*" The transformation we applied here is commonly known as *partial evaluation* and it is crucial that this transformation preserves equivalence, i.e., that it does not change the meaning of the original program.

There already exist formulations of the partial-evaluation rule in answer-set programming but they have some shortcomings: First, they only preserve the weak notion of ordinary equivalence which does not suffice for our purposes. Second, they are only defined on restricted classes of programs or are defined in a way which does not really cover the idea behind partial evaluation. We will thus use a notion of equivalence which is stronger than ordinary equivalence and propose a partial-evaluation transformation which preserves this kind of equivalence.

1.2 Results

- Since we do not know of any existing formal definitions which exactly state when a rule is contained in the generate, define, or test part of a program, we will introduce such

definitions and argue for their usefulness by applying them to real-world programs. In the course of this, we will also review different notions of stratification.

- Based on the mentioned definitions, we will implement an algorithm which automatically classifies the rules of a given program into a generate, define, and test part. For this implementation we will make use of techniques from meta programming, meaning that we will implement the main part of the classification algorithm as an answer-set program. Our work is thus another contribution to existing meta-programming approaches in answer-set programming. We will give a detailed explanation of the implementation and illustrate its behaviour by providing example classifications of real-world programs.

- When translating a program into a form which is closer to natural language, the order in which the rules of the program are explained is very important. We will thus introduce an algorithm which computes such an order that should lead to meaningful explanations.

- We will introduce a propositional partial-evaluation transformation and its generalisation to the non-ground case. We will then prove that our transformations preserve *relativised uniform equivalence* [49, 70]. This yields the straightforward corollary that they preserve ordinary equivalence over the answer-set semantics too.

1.3 Structure of the Thesis

Chapter 2 contains preliminaries which are important for the rest of the thesis. It contains a short introduction into the development of logic programming and the answer-set-programming paradigm in particular. Graph representations of logic programs are covered as well as an introduction into the generate-define-test paradigm. Finally, some syntactic extensions of logic programs which are used in practice are presented.

In Chapter 3 we develop a formal definition of the generate, define, and test part of an answer-set program as well as discuss the implementation of the corresponding classification algorithm. Since the search for an accurate definition of the generate part is strongly related to notions of *stratification*, we will introduce some of these notions too. A small part of the chapter is also used for suggesting an order in which the rules of a program can be reasonably explained.

Our results regarding equivalence-preserving transformations are contained in Chapter 4. It starts with a short overview over propositional program transformations and a motivation why the partial-evaluation transformation is important for us. We review well-known notions of equivalence, most importantly (non-ground) relativised uniform equivalence, which we will, for the ease of notation, denote as input equivalence.

After this, we introduce a propositional partial-evaluation transformation for which we show equivalence preservation with respect to input equivalence. We then present a generalisation of this transformation to the non-ground case and mention some of the difficulties that the non-ground case brings with it. A large part of the chapter covers the proof that our transformation preserves input equivalence and following this we show how our results can be used together to help transforming answer-set programs into a form which is closer to natural language. Fi-

nally, Chapter 5 concludes with an overview over related approaches in the field of answer-set programming, an outlook on future work, and a summary of the thesis.

CHAPTER 2

Preliminaries

Answer-set programming (ASP) is a fully declarative logic-programming paradigm which was developed in the 1990s. The basic idea is to encode a problem specification as a logic program which is passed to a so-called *solver*. The solver then evaluates the program and returns the solution of the problem as a result [21]. Historically, a *normal* logic program is a finite set of rules in the form

$$A_0 \leftarrow A_1, \ldots, A_m, \text{not } A_{m+1}, \ldots, \text{not } A_n,$$

where $A_0, \ldots A_n$ are atoms of a given first-order language.

One of the main challenges in the study of semantics of logic programming is to assign to such a program a "reasonable" semantics [38]. Consider as an example the problem of finding the root node of a given tree. The root node of a tree is that node which has no parent node. We can encode the problem as a logic program containing the following rules:

$$\begin{aligned} root(X) &\leftarrow \text{not } has\_parent(X), \\ has\_parent(X) &\leftarrow node(X), node(Y), edge(Y, X). \end{aligned}$$

If we add to this program for every node v of a given tree the rule

$$node(v) \leftarrow,$$

and for every edge (u, v) the rule

$$edge(u, v) \leftarrow,$$

then under many known semantics, the atom $root(w)$ would be considered true if and only if w is the actual root node of the given tree.

One of the most popular logic-programming languages is Prolog, whose semantics is based on Kowalskis highly efficient SLD-resolution (selective linear definite clause resolution) [34]. For a given Prolog program, one can ask so-called *queries* in the form of atoms. For the above

program, one could for example ask the query $root(X)$. If there exists an SLD-refutation of the program conjuncted with the negated query, then a substitution $X = c$ is returned, meaning that c is a constant such that $root(c)$ is considered to be true. But Prolog is not fully declarative: The order of atoms in rule bodies as well as the order of rules within a program can influence termination.
In contrast, various fully declarative semantics have been suggested in the past, with the *perfect-model semantics* [54] being an approach which is not only considered to be intuitive, but also shown to be equivalent to suitable forms of the major formalisations of non-monotonic reasoning. However, the perfect-model semantics has the drawback that it is only defined on a relatively narrow class of programs, the so called *locally stratified* programs (a definition of this class will be given later) [55]. Consider as an example the program Π_1 which consists of the following rules [7]:

$$open \leftarrow \text{not } closed,$$
$$closed \leftarrow \text{not } open.$$

This program is not locally stratified and its *intuitive* meaning is apparently not so clear. For a time it was thought that programs which are not locally stratified do not really make sense. They were considered faulty and thus it was not seen as a flaw of the perfect-model semantics that it is not defined on these programs. But experience cast doubt on this view and thus different generalisations of the perfect-model semantics were proposed [67].

Some of those proposed semantics, most notably the *weakly-perfect-model semantics* [53] and the *well-founded semantics* [67], assign to a program exactly one model. This approach is referred to as the *canonical-model approach* [67]. In contrast to this, the *stable-model semantics* [29] assigns various (possibly no) stable models to a logic program, although it should be noted that Gelfond and Lifschitz considered the stable-model semantics of a program to be defined only if the program has exactly one stable model.

There was a long-lasting dispute over the question which approach should be preferred [21], but despite all the differences, the three mentioned semantics all agree with the perfect-model semantics on locally stratified programs [53] and it was the introduction of the stable-model semantics which laid the foundations for the fruitful development of answer-set programming.

We will thus in the following first introduce a broader definition of logic programs which generalises the already mentioned normal logic programs. After this, we will give a definition of the *answer-set semantics* on *extended disjunctive logic programs* and show how it is related to the stable-model semantics. We conclude this chapter with an overview over the so-called *generate-define-test paradigm* as well as over some syntactic extensions of answer-set programs which are useful in answer-set-programming practice.

2.1 Logic Programs

In general, a *logic program* is a finite set of *rules* in the form

$$L_0 \vee \cdots \vee L_k \leftarrow L_{k+1}, \ldots, L_m, \text{not } L_{m+1}, \ldots, \text{not } L_n$$

where $L_0, \ldots L_n$ are literals, i.e., atoms from some first-order language $\mathcal{L}$, possibly preceded by the strong-negation symbol "$\neg$". If not mentioned otherwise, we will assume that $\mathcal{L}$ does not contain function symbols. We call the literals $L_0, \ldots, L_k$ the *head* and $L_{k+1}, \ldots, L_n$ the *body* of the rule. Furthermore, the literals $L_{k+1}, \ldots, L_m$ are called the *positive body* while $L_{m+1}, \ldots, L_n$ are called the *negative body* of the rule. Literals which are possibly preceded by the not-symbol will be referred to as *naf-literals* (for *negation as failure*).

By $head(r)$ and $body(r)$ we denote the sets of literals occurring in the head and body, respectively. We write $body^+(r)$ for the set of positive body literals and $body^-(r)$ for that of the negative body literals. By $at(r)$ and $lit(r)$ we will respectively denote the set of all atoms and literals of a rule. We furthermore write $at(\Pi)$ and $lit(\Pi)$ for the set of all atoms and literals of a program, respectively.

Let $A = \{A_0, \ldots, A_k\}$, $B = \{B_0, \ldots, B_l\}$, and $C = \{C_0, \ldots, C_m\}$ be sets of literals. Then,

$$A \leftarrow B \cup \mathrm{not}\ C$$

represents the rule

$$A_0 \vee \cdots \vee A_k \leftarrow B_0, \ldots, B_l, \mathrm{not}\ C_0, \ldots, \mathrm{not}\ C_m.$$

When saying that some rule r is equal to or has the form $A \leftarrow B \cup \mathrm{not}\ C$, we mean that r contains exactly the literals of A, B, and C in $head(r)$, $body^+(r)$, and $body^-(r)$, respectively.

If a rule contains exactly one head literal, we say that it is *definite*. If it contains more than one head literal, we say that it is *disjunctive* and if it contains no head atoms at all, we call it a *constraint*. A program which does not contain strong negation and for which all rules are definite is called a *normal logic program*. A program which contains disjunctive rules but no strong negation is called a *disjunctive logic program*. A program without disjunctive rules but with strong negation is called an *extended logic program* and if a program contains both strong negation and disjunctive rules it is called an *extended disjunctive logic program*. A definite rule r with an empty body, such as $L \leftarrow$, is called a *fact* and can be written as L. For a program without default-negated literals we say that it is *positive*.

We will use strings starting with lower-case letters as predicate symbols and constant symbols, respectively. Variable symbols are denoted by strings starting with an upper-case letter. A rule which does not contain any variable symbols is called *ground* whereas rules with variable symbols are called *non-ground*. A program is ground iff all of its rules are ground. Programs which contain only predicates of arity 0 are called *propositional logic programs*. If we replace every atom of a ground program by an according 0-ary predicate, we obtain a corresponding propositional program. Hence, we will sometimes use the terms *ground* and *propositional* interchangeably.

If not mentioned otherwise, we will use the term *predicate* for predicate symbols as well as their strong negation (i.e., the predicate symbol preceded by the strong-negation symbol). If a literal is preceded by the not-symbol, we say that it is *default negated*. Let L be a naf-literal which is not default negated. By $\bar{L}$ we denote the default-negated naf-literal $\mathrm{not}\ L$. For a default-negated naf-literal L of the form $\mathrm{not}\ C$ we denote by $\bar{L}$ the literal C. Let S be a set. Then, we denote the power set of S by 2^S.

The answer-set semantics of non-ground programs is defined in terms of the *ground instantiation* of a logic program which we will define in the following.

2.2 Substitutions

Let Π be a logic program. Then, by $V(\Pi)$ we denote the set of variables occurring in Π. The set of all constants occurring in Π is denoted by $HU(\Pi)$, called the *Herbrand universe* of Π. The following definition of a substitution is a slight modification of the one given by Leitsch [36].

Definition 2.2.1 (Substitution). Let V be a set of variables and C a set of constants. A *substitution* is a mapping $\lambda : V \to V \cup C$ such that $\lambda(X) \neq X$ only for finitely many $X \in V$. $\diamondsuit$

If λ is a substitution, the set $\{X \mid \lambda(X) \neq X\}$ is called the *domain* of λ, written as $dom(\lambda)$. The set $\{\lambda(X) \mid X \in dom(X)\}$ is called the *range* of λ, written as $rg(\lambda)$.

Let Π a logic program. Then, a substitution λ is called *ground* iff $V = V(\Pi)$ and $rg(\lambda) \subseteq HU(\Pi)$.

We use the following post-fix notation:

$$\begin{aligned}
\lambda\mu &= \mu \circ \lambda \text{ for substitutions } \mu, \lambda,\\
X\lambda &= \lambda(X) \text{ for } X \in V,\\
c\lambda &= c \text{ for } c \in HU(\Pi),\\
a(T_1, \ldots, T_n)\lambda &= a(T_1\lambda, \ldots, T_n\lambda) \text{ for } a \in at(\Pi) \text{ and } T_1, \ldots, T_n \in V \cup HU(\Pi),\\
(\neg A)\lambda &= \neg(A\lambda) \text{ for } A \in at(\Pi),\\
(\text{not } L)\lambda &= \text{not } (L\lambda) \text{ for } L \in lit(\Pi).
\end{aligned}$$

Let furthermore r be a rule of form

$$L_1 \vee \cdots \vee L_k \leftarrow L_{k+1}, \ldots, L_i, \ldots, L_m, \text{not } L_{m+1}, \ldots, \text{not } L_n.$$

Then, we write $r\lambda$ for

$$L_1\lambda \vee \cdots \vee L_k\lambda \leftarrow L_{k+1}\lambda, \ldots, L_m\lambda, \text{not } L_{m+1}\lambda, \ldots, \text{not } L_n\lambda.$$

By an *expression* we mean either a rule, a naf-literal, a literal, a variable, or a constant. For an expression E, we denote by $V(E)$ the set of variables occurring in E. For a set of expressions C, we write $C\lambda$ for $\bigcup_{x \in C}\{x\lambda\}$. Finally, for substitutions λ we use the notation

$$\lambda = \{X \mapsto T \mid (X, T) \in \lambda \text{ and } X\lambda \neq X\}.$$

For example, $\{X \mapsto A, Y \mapsto B\}$ denotes the substitution $\{(X, A), (Y, B)\} \cup \{(Z, Z) \mid Z \in V\}$.

Note that the Herbrand universe is always finite and thus, for every logic program Π, there exists only a finite number of possible ground substitutions. For any rule, naf-literal, or literal E, and a ground substitution λ with $V(E) \subseteq dom(\lambda)$, we call $E\lambda$ a *ground instance* of E. For a logic program Π, we call the set

$$\bigcup_{r \in \Pi}\{r\lambda \mid \lambda \text{ is a ground substitution with } V(r) \subseteq dom(\lambda)\}$$

the *ground instantiation* or *grounding* of Π, written as $grd(\Pi)$. The set of all atoms which can be formed from predicate symbols in Π and terms of $HU(\Pi)$ is called the *Herbrand base* of Π, written as $HB(\Pi)$.

2.3 Answer Sets

The notion of an *answer set* was originally defined by Gelfond and Lifschitz [30]. To define the answer sets of a program we need the notion of the so-called *reduct*, also referred to as *Gelfond-Lifschitz reduct*.

Definition 2.3.1 (Gelfond-Lifschitz Reduct [29]). Let Π be a ground extended disjunctive logic program. For any set $\mathcal{M}$ of atoms from Π, the *Gelfond-Lifschitz reduct* $\Pi^{\mathcal{M}}$ is obtained from Π by deleting

(i) each rule that has a default-negated naf-literal $\mathrm{not}\ B$ in its body with $B \in \mathcal{M}$, and

(ii) all default-negated naf-literals in the bodies of the remaining rules.

$\diamondsuit$

An answer set of a program Π which contains default-negated literals is defined in terms of the reduct $\Pi^{\mathcal{M}}$ which does not contain default-negated literals. We will thus first define answer sets of positive programs.

Definition 2.3.2 (Answer Set of a Positive Program [30]). Let Π be an extended disjunctive logic program without default-negated literals and $\mathcal{M}$ a set of literals. We say that $\mathcal{M}$ is an *answer set* of Π if it is a minimal set of literals such that

(i) for each rule $r \in grd(\Pi)$, if $body(r) \subseteq \mathcal{M}$, then $head(r) \cap \mathcal{M} \neq \emptyset$;

(ii) if $\mathcal{M}$ contains a pair of complementary literals, then $\mathcal{M} = lit(grd(\Pi))$.

$\diamondsuit$

Condition (ii) expresses the inconsistency of an answer set which contains both an atom and its strong negation.

Definition 2.3.3 (Answer Set [30]). Let Π be an extended disjunctive logic program and $\mathcal{M}$ a set of literals. We say that $\mathcal{M}$ is an *answer set* of Π if it is an answer set of $grd(\Pi)^{\mathcal{M}}$. $\diamondsuit$

The answer sets of a program without strong negation are exactly its *stable models*. In the absence of strong negation we will thus use the terms *answer set* and *stable model* interchangeably. As already mentioned in the introduction of this chapter, Gelfond and Lifschitz stated that the stable-model semantics is defined for a logic program Π if Π has exactly one stable model. One can consider this restriction as somewhat arbitrary and in modern answer-set-programming practice, the semantics of an answer-set program is defined as the collection of its answer sets [26].

Example 2.3.4. Consider again the rules of program Π_1 from above:

$$open \leftarrow \text{not } closed, \tag{2.1}$$
$$closed \leftarrow \text{not } open. \tag{2.2}$$

Let $\mathcal{M} = \{open\}$. Then, $\Pi_1^{\mathcal{M}}$ is obtained by removing the body atom $closed$ from Rule (2.1), as well as deleting the whole Rule (2.2). Hence, $\Pi_1^{\mathcal{M}} = \{open \leftarrow\}$ and since $\mathcal{M}$ is an answer set of $\Pi_1^{\mathcal{M}}$, it is an answer set of Π_1. Similarly, $\{closed\}$ is also an answer set of Π_1. The set $\{open, closed\}$ is not an answer set of Π_1, since it leads to an empty reduct, of which $\{open, closed\}$ is not an answer set. For the empty set $\emptyset$, we get the reduct

$$\Pi_1^{\emptyset} = \{open \leftarrow, \\ closed \leftarrow\}.$$

Since $\Pi_1^{\emptyset}$ has $\{open, closed\}$ as its (unique) answer set, the empty set is not an answer set of Π_1. We conclude that $\{open\}$ and $\{closed\}$ are the only answer sets of Π_1. $\diamondsuit$

According to Gelfond and Lifschitz, the intuition behind stable models – and thus of answer sets – of a program Π can, like the intuition behind stable expansions in autoepistemic logic [48], be described as *possible sets of beliefs that a rational agent might hold, given* Π *as his premisses* [29]. For the above example, if we assume that $open$ and $closed$ are referring to the state of a door, an agent can believe that the door is closed. He can also believe that the door is open, but he cannot believe that the door is both open and closed, as well as he cannot believe that neither is the case. The program Π_1 can be equivalently expressed by the disjunctive program

$$\Pi_1' = \{open \vee closed \leftarrow\}.$$

Here, again only one of $open$ and $closed$ can be true in a stable model of Π_1'. This is because of the minimality condition of answer sets.

There are two notions of falsity within answer-set programming, namely falsity in the sense that *an atom cannot be derived*, expressed by default negation, and falsity in the sense that *its negation can be derived*, expressed by strong negation [30]. The following example is due to John McCarthy and was used by Gelfond and Lifschitz to illustrate the difference between strong and default negation [30]:

Example 2.3.5. Suppose we want to express the knowledge that a school bus should cross railway tracks under the condition that there is no approaching train. Consider

$$\Pi_d = \{cross \leftarrow \text{not } train\}.$$

Here, if the information about the presence of a train is not available – for example, if the vision of the bus driver is blocked – the school bus will cross railway tracks. Under program

$$\Pi_c = \{cross \leftarrow \neg train\}$$

this is not the case. Here, the school bus only crosses railway tracks if there is definitive knowledge that no train is approaching. $\diamondsuit$

2.4 Graph Representations of Logic Programs

There exist various different graph representations of logic programs. In the following we introduce three of them. We start with the so-called *dependency graph*. To this end, we need the notion of a *labelled graph* as well as the notion of *direct dependency*.

Definition 2.4.1 (Labelled graph). A *labelled graph* is a triple (V, E, ξ), where V is the set of vertices, E is the set of edges (i.e., a binary relation on V), and $\xi : E \to X$ is a labelling function which assigns to each edge a label from some set X. $\diamondsuit$

Definition 2.4.2 (Direct Dependency). Let Π be a logic program. For any two predicates u and v we say that u *depends directly positive* (*negative*) on v iff there exists some rule $r \in \Pi$ such that u occurs in the head of r and v occurs in the positive (negative) body of r.

For any two ground literals u and v we say that u *depends directly positive* (*negative*) on v iff there exists some rule $r \in grd(\Pi)$ such that u occurs in the head of r and v occurs in the positive (negative) body of r. $\diamondsuit$

Definition 2.4.3 (Dependency Graph). Let Π be a logic program. The *dependency graph*, $DG(\Pi)$, of Π is the labelled graph (V, E, ξ), where

1. V is the set of all predicates occurring in Π,
2. $(u, v) \in E$ iff there exists some rule $r \in \Pi$ such that u occurs in $body(r)$ and v occurs in $head(r)$,
3. $\xi : E \to 2^{\{+,-\}}$,
4. $\xi(u, v) = \{+, -\}$ iff v depends directly positive and directly negative on u,
5. $\xi(u, v) = \{+\}$ iff v depends directly positive but not directly negative on u, and
6. $\xi(u, v) = \{-\}$ iff v depends directly negative but not directly positive on u.

$\diamondsuit$

For an edge e we say that e is *positive* iff "$+$" $\in \xi(e)$ and *negative* iff "$-$" $\in \xi(e)$. When visualising a dependency graph, we will draw positive edges as solid lines and negative edges as dashed lines. Furthermore, if an edge is positive and negative we will draw both a solid and a dashed line.

Example 2.4.4. Consider the program Π_2, consisting of the following rules:

$$\begin{aligned} symmetric\_difference(X) &\leftarrow union(X), \text{not } intersection(X), \\ union(X) &\leftarrow a(X), \\ union(X) &\leftarrow b(X), \\ intersection(X) &\leftarrow a(X), b(X). \end{aligned}$$

The dependency graph of Π_2 is given in Figure 2.1.

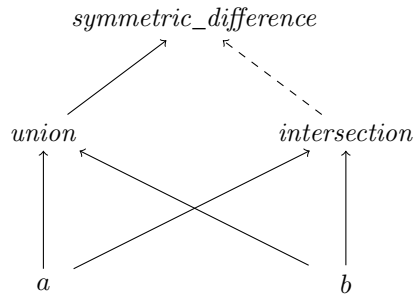

Figure 2.1: Dependency graph of Π_2.

The program Π_3, consisting of the following rules, is different from Π_2 but has the same dependency graph:

$$\begin{aligned} \mathit{symmetric\_difference}(X) &\leftarrow \mathit{union}(X), \text{not } \mathit{intersection}(X), \\ \mathit{union}(X) &\leftarrow a(x), b(X), \\ \mathit{intersection}(X) &\leftarrow a(X), \\ \mathit{intersection}(X) &\leftarrow b(X). \end{aligned}$$

This demonstrates that the dependency graph does not uniquely determine a corresponding logic program. ♢

Definition 2.4.5 (Ground Dependency Graph). Let Π be a logic program. Then, the *ground dependency graph*, $GDG(\Pi)$, of Π is the dependency graph obtained from $grd(\Pi)$, but with the vertex set $V = lit(grd(\Pi))$. ♢

Definition 2.4.6 (Dependency of Predicates). Let Π be a logic program with dependency graph $DG(\Pi)$ and A, B predicates occurring in Π. Then, A *depends positively* on B iff $DG(\Pi)$ contains a directed path from B to A consisting only of positive edges. A *depends negatively* on B, denoted $B < A$, iff $DG(\Pi)$ contains a directed path from B to A passing trough a negative edge. Furthermore, A *depends* on B, denoted $B \leq A$, iff it depends negatively or positively on B. ♢

The dependency notions for ground literals (instead of predicates) are defined analogously by considering the ground dependency graph instead of the dependency graph in Definition 2.4.6. With the notion of dependency, we can define the so-called *head-cycle free* rules and programs which will be important later on.

Definition 2.4.7 (Head-Cycle Free [3, 20]). Let Π be a propositional logic program and $r \in \Pi$ a rule. Then, r is *head-cycle free* in Π iff the dependency graph of Π does not contain a positive directed cycle which goes through two distinct head atoms of r. Π is head-cycle free iff all of its rules are head-cycle free. ♢

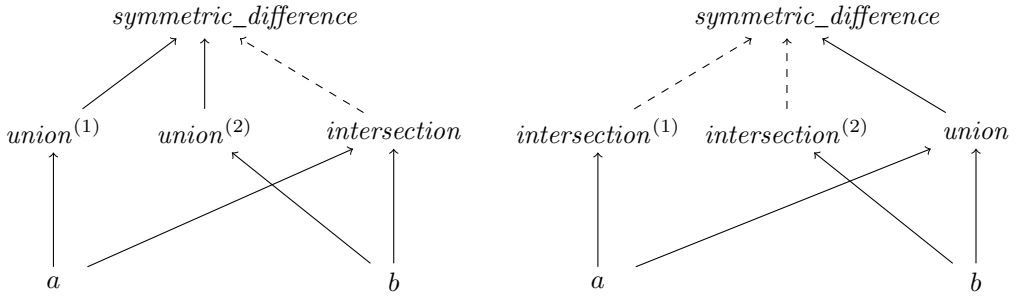

Figure 2.2: Extended dependency graphs of Π_2 (left) and Π_3 (right).

Brignoli, Costantini, D' Antona, and Provetti [8] introduced the so called *extended dependency graph* (*EDG*). In the extended dependency graph the following holds: If a literal L occurs in the head of n different rules, then the EDG contains n different vertices $L^{(1)}, \ldots, L^{(n)}$.

Definition 2.4.8 (Extended Dependency Graph [8]). Let Π be a ground normal logic program. The *extended dependency graph*, $EDG(\Pi)$, is the labelled graph (V, E, ξ), where

1. for each rule r in Π, V contains a vertex $a_i^{(k)}$, where a_i is the name of the head of r and k is the index of r in the definition of a_i (i.e., r is the k-th rule having a_i as head, for some (arbitrary) order on the rules defining a_i),

2. for each atom u never appearing in a head, V contains a vertex labelled u,

3. $\xi : E \to 2^{\{+,-\}}$,

4. for each $c_j^{(l)} \in V$, there is a positive edge $(c_j^{(l)}, a_i^{(k)})$ iff c_j occurs in the positive body of the k-th rule defining a_i, and

5. for each $c_j^{(l)} \in V$, there is a negative edge $(c_j^{(l)}, a_i^{(k)})$ iff c_j occurs in the negative body of the k-th rule defining a_i.

$\diamondsuit$

For programs where every literal is defined by at most one rule, the EDG coincides with the conventional dependency graph. But in contrast to the conventional dependency graph, the EDG uniquely determines its corresponding program [10].

Example 2.4.9. Consider again the programs Π_2 and Π_3 from Example 2.4.4. Although they have the same dependency graphs, their – distinct – extended dependency graphs are given in Figure 2.2. $\diamondsuit$

The last graph representation is called the *rule graph* of a logic program and was introduced by Dimopoulus and Torres [14] for programs without positive body literals. Our definition is a slight modification which takes positive body literals and unifiability into account.

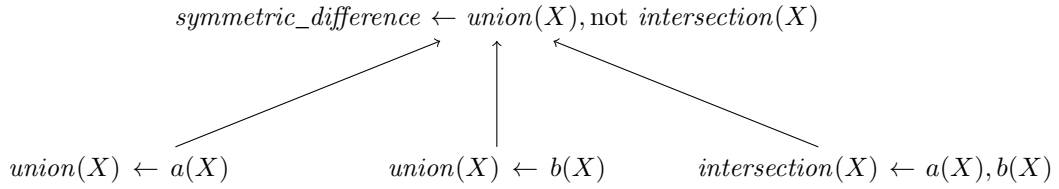

Figure 2.3: Rule graph of program Π_2.

Definition 2.4.10 (Rule Graph). Let Π be a logic program. The *rule graph*, $RG(\Pi)$, of Π is the graph (V, E), where

1. V is the set of all rules occurring in Π,

2. $(u, v) \in E$ iff u contains a head literal A and v contains a body literal B such that A and B are unifiable.

$\diamondsuit$

Example 2.4.11. Consider again program Π_2 from above. Its rule graph is given in Figure 2.3. $\diamondsuit$

2.5 The Generate-Define-Test Paradigm

In answer-set programming, problems can be encoded following the so called *guess and check* paradigm. An introduction into the usage of this paradigm was given by Eiter, Faber, Leone, and Pfeifer [17]. It was further refined by Lifschitz [37] who divided example programs into a *generate*, *define*, and *test* part, leading to the so called *generate-define-test* paradigm.

Within this paradigm, the generate part normally creates multiple candidate answer sets from which the test part eliminates those answer sets which are not the desired solutions of a problem. Both the generate and the test part may use predicates which are defined in the define part.

Example 2.5.1. Consider the problem of finding a Hamiltonian path from some start node v within a directed graph. A Hamiltonian path is a path which visits all nodes exactly once. The following encoding Π_4 is a slight modification of an encoding by Eiter et al. [21]. which has as answer sets the valid Hamiltonian paths starting from v. It is assumed that an input instance contains the atom $start\_node(v)$:

$$in\_path(X, Y) \vee not\_in\_path(X, Y) \leftarrow edge(X, Y),$$

$$reached(X) \leftarrow start\_node(X),$$
$$reached(X) \leftarrow reached(Y), in\_path(Y, X),$$
$$a\_node\_is\_not\_reached \leftarrow node(X), \text{not } reached(X),$$

$$\leftarrow a\_node\_is\_not\_reached,$$
$$\leftarrow in\_path(X,Y), in\_path(X,Z), Y \neq Z,$$
$$\leftarrow in\_path(Y,X), in\_path(Z,X), Y \neq Z.$$

The first rule plays the role of the generate part: It generates answer sets in which arbitrary edges are contained in the path. The next three rules can be considered the define part: They define the auxiliary predicates $reached$ and $a\_node\_is\_not\_reached$ which are then used in the test part (last three rules) to eliminate answer sets which do not correspond to valid Hamiltonian paths. ♢

2.6 Syntactic Extensions of Programs

To simplify programming, several syntactic shortcuts were incorporated into the *ASP-Core-2* language standard [26]. We will introduce some of them in the following.

Anonymous Variables

An anonymous variable in a rule is denoted by "_" (character underscore). Each occurrence of "_" stands for a fresh variable in the respective context (i.e., different occurrences of anonymous variables represent distinct variables). Consider as an example the following rule:

$$has\_parents(X) \leftarrow father(Y,X), mother(Z,X).$$

Since the actual values of Y and Z can be considered irrelevant in such a rule, anonymous variables allow us to write it as follows:

$$has\_parents(X) \leftarrow father(\_,X), mother(\_,X).$$

Built-In Atoms

A *built-in atom* has the form $t \prec u$ for terms t and u with $\prec \in \{<, \leq, =, \neq, >, \geq\}$. They are evaluated according to a well-defined order which is defined in the ASP-Core-2 standard. For example, in the following rule the built-in atom ">" is used:

$$older\_sister\_of(X,Y) \leftarrow sister\_of(X,Y), has\_age(X,U), has\_age(Y,V), U > V.$$

Arithmetic Terms

An *arithmetic term* has the form $-(t)$ or $t \circ u$ for terms t and u with $\circ \in \{+, -, *, /\}$. Parentheses can be omitted in cases where the standard operator precedences apply. Arithmetic terms require that the corresponding arithmetic-operations are well-defined for the terms to which they apply. They are evaluated after grounding [26]. The usage of arithmetic terms is illustrated by the following rules:

$$state(X, T+1) \leftarrow state(X,T), \text{not } \neg state(X, T+1),$$
$$state(a,1) \leftarrow .$$

Choice Rules

A *choice rule* has the form

$$\{e_1; \ldots; e_m\} \prec u \leftarrow B_1, \ldots, B_n,$$

where $B_1, \ldots, B_n$ are naf-literals for $n \geq 0$, u is a term, $\prec \in \{<, \leq, =, \neq, >, \geq\}$ and $e_1, \ldots, e_m$ are so-called *choice elements* for $m \geq 0$. A choice element is of the form

$$A : L_1, \ldots, L_k,$$

where A is a literal and $L_1, \ldots, L_k$ are naf-literals for $k \geq 0$.

The idea of a choice rule is that if the body $B_1, \ldots, B_n$ is true in an answer set, then some subset S of $\{e_1, \ldots, e_m\}$ must be chosen as true, meaning that the A literal of the corresponding choice element e_i is set to true if all the $L_1, \ldots, L_k$ of e_i are true. The size of S is restricted by "$\prec u$". For the complete formal definition we refer to the ASP-Core 2 standard. For example, the following rule expresses that if a is a vegan pizza, then at least two of the toppings cheese, pepperoni, and olives must be chosen. Furthermore, cheese can only be chosen if it is explicitly vegan:

$$\{top(a, cheese) : vegan(cheese); top(a, pepperoni); top(a, olives)\} \geq 2 \leftarrow veg\_pizza(a).$$

CHAPTER 3

Analysing the Structure of Programs

The goal of this section is to introduce methods that help to analyse the overall structure of programs. We will develop an algorithm which takes an answer set program and automatically classifies its rules into the three parts *generate*, *define*, and *test*. To this end, we will first develop formal definitions of these three parts. The classification algorithm will then be based on these definitions and for its implementation we will make use of meta-programming techniques. We conclude the chapter by suggesting a reasonable order in which the rules of a program can be explained.

3.1 Definition of the Generate Part

Intuitively, we would want the generate part to contain those rules which cause non-determinism. Our goal is thus to identify those rules of a program which can lead to the generation of more than one answer set, and classify them as *generating rules*. Disjunctive heads and aggregates are obvious sources of non-determinism. Apart from them, default negation can cause the creation of more than one answer set, but there, things are a little more involved.

Stratification

To identify a possibly very large class of non-deterministic programs, we will use some concepts of *stratification*. Several notions of stratification were introduced in the past, with the most well-known being (*ordinary*) *stratification* [1] and the stronger concept of *local stratification* [54]. Gelfond and Lifschitz [29] showed that every locally-stratified normal logic program has a unique answer set. Furthermore, most semantics for normal logic programs agree on locally-stratified programs and we already mentioned earlier that for some time it was thought that programs which are not locally stratified do not really make sense [67]. However, we will show that there are programs with unique answer sets which are not locally stratified and it can also be argued that they are quite intuitive.

Definition 3.1.1 (Stratification [40]). An extended logic program Π is *stratified* if there exists a mapping f from predicate symbols to ordinals such that, for every rule $r \in \Pi$ and any predicates P_1, P_2,

- if P_1 and P_2 occur in $head(r)$ then $f(P_1) = f(P_2)$,
- if P_1 occurs in $head(r)$ and P_2 occurs in $body^+(r)$ then $f(P_1) \geq f(P_2)$,
- if P_1 occurs in $head(r)$ and P_2 occurs in $body^-(r)$ then $f(P_1) > f(P_2)$.

$\diamondsuit$

The function f is also referred to as a *level mapping*.

Definition 3.1.2 (Local Stratification). Let Π be a logic program and Π' the program obtained by replacing every literal in $grd(\Pi)$ by a corresponding propositional predicate symbol. Then, Π is *locally stratified* if Π' is stratified. $\diamondsuit$

Note that every stratified program is locally stratified and that the empty program $\emptyset$ is trivially stratified by a level mapping with an empty domain.

Example 3.1.3. Consider the program Π_5 which consists of the following rules:

$$c(X) \leftarrow \text{not } b(X), \tag{3.1}$$
$$b(X) \leftarrow a(X), \tag{3.2}$$
$$a(2) \leftarrow \text{not } a(1). \tag{3.3}$$

Π_5 is locally stratified by the following level mapping f:

$$c(2) \mapsto 3,$$
$$a(2), b(2), c(1) \mapsto 2,$$
$$a(1), b(1) \mapsto 1.$$

However, it is not stratified, since the predicate a is contained both in the head and the negative body of Rule (3.3) which would require $f(a) > f(a)$. $\diamondsuit$

Weak Stratification

We already mentioned that there are classes of programs which have a unique answer set despite not being locally stratified. One such example is the class of *weakly stratified* programs [53]. We will in the following give some preliminary definitions and examples from Przymusinki and Przymusinska [53] which lead to the notion of *weak stratification*. After this, we develop our definition of the generate part based on the ideas behind weak stratification.

We start with the notion of a *partial interpretation*. Let Π be a normal logic program. By a partial interpretation of Π we mean a signed subset of the Herbrand base $HB(\Pi)$, i.e., a subset of $HB(\Pi)$, some of whose elements may be default negated and thus considered false. An *interpretation* $\mathcal{M}$ of Π is a partial interpretation such that for all $A \in HB(\Pi)$ we have either

$A \in \mathcal{M}$ or $\bar{A} \in \mathcal{M}$. An interpretation $\mathcal{M}$ is a *Herbrand model* of a program Π if for each rule $r \in grd(\Pi)$ it holds that if $body^+(r) \subseteq \mathcal{M}$ and $body^-(r) \cap \mathcal{M} = \emptyset$, then $head(r) \cap \mathcal{M} \neq \emptyset$. $\mathcal{M}$ is the (unique) *least Herbrand model* of Π if there is no other model of Π containing less positive atoms.

Definition 3.1.4. Let $\sim$ be the equivalence relation between ground atoms of Π defined as follows:

$$A \sim B \text{ iff } A = B, \text{or } A < B \text{ and } B < A.$$

We call its equivalence classes *components* of the ground dependency graph $GDG(\Pi)$. A component is *trivial* if it consists of a single element A such that $A \not< A$. $\diamondsuit$

The following order relation over components is transitive and asymmetric.

Definition 3.1.5. Let C_1 and C_2 be two components of $GDG(\Pi)$. We define:

$$C_1 \prec C_2 \text{ iff } C_1 \neq C_2 \text{ and there exist } L_1 \in C_1 \text{ and } L_2 \in C_2 \text{ such that } L_1 < L_2.$$

We call a component C_1 *minimal* if there is no component C_2 such that $C_2 \prec C_1$. $\diamondsuit$

Definition 3.1.6. By the *bottom stratum*, $S(\Pi)$, of Π we mean the union of all minimal components of Π, i.e.,

$$S(\Pi) = \{C \mid C \text{ is a minimal component of } GDG(\Pi)\}.$$

By the *bottom layer* $L(\Pi)$ of Π we mean the corresponding subprogram of Π, i.e.,

$$L(\Pi) = \bigcup \{r \in \Pi_g \mid head(r) \subseteq S(\Pi)\}.$$

Herbrand models $\mathcal{M}$ of the subprogram $L(\Pi)$ will be identified with signed subsets of the bottom stratum $S(\Pi)$. $\diamondsuit$

Example 3.1.7. Consider the following program Π_6 which consists of the following rules:

$$\begin{aligned} q(X) &\leftarrow p(X,Y), \text{not } q(Y), \\ p(1,2) &\leftarrow . \end{aligned}$$

The grounding $grd(\Pi)$ consists of the following rules:

$$\begin{aligned} q(1) &\leftarrow p(1,1), \text{not } q(1), \\ q(1) &\leftarrow p(1,2), \text{not } q(2), \\ q(2) &\leftarrow p(2,1), \text{not } q(1), \\ q(2) &\leftarrow p(2,2), \text{not } q(2), \\ p(1,2) &\leftarrow . \end{aligned}$$

Because of the negative dependency relations $q(1) < q(2)$ and $q(2) < q(1)$ we have the following components: $\{q(1), q(2)\}$, $\{p(1,1)\}$, $\{p(1,2)\}$, $\{p(2,1)\}$, and $\{p(2,2)\}$. The component $\{q(1), q(2)\}$ is the only non-minimal component, hence we get the bottom stratum $S(\Pi) = \{p(1,1), p(1,2), p(2,1), p(2,2)\}$ and the corresponding bottom layer $L(\Pi) = \{p(1,2)\}$. $\diamondsuit$

In the following, for a ground naf-literal L and a partial interpretation $\mathcal{M}$, we will say that $\mathcal{M} \models L$ if L is in $\mathcal{M}$ and $\mathcal{M} \models \bar{L}$ if $\bar{L}$ is in $\mathcal{M}$

Definition 3.1.8. Let Π be a logic program and $\mathcal{M}$ a set of naf-literals. By a *reduction of* Π *modulo* $\mathcal{M}$ we mean a new program $\frac{\Pi}{\mathcal{M}}$ obtained from Π by performing the following two reductions:

- removing from Π all rules which contain a body naf-literal L such that $\mathcal{M} \models \bar{L}$ or whose head belongs to $\mathcal{M}$ (in other words, removing all rules true in $\mathcal{M}$);
- removing from all the remaining rules those body naf-literals L which are satisfied in $\mathcal{M}$, i.e., such that $\mathcal{M} \models L$.

Finally, we also remove from the resulting program all non-facts whose heads appear as facts in the program. This step ensures that the set of literals appearing in facts, also called *extensional* literals, is disjoint from the set of literals appearing in heads of non-facts, also called *intensional* literals. ♢

Definition 3.1.9. Suppose that Π is a normal logic program and let $\Pi_0 = \Pi$ and $\mathcal{M}_0 = \emptyset$. Suppose that $\alpha > 0$ is a countable ordinal such that programs Π_δ and partial interpretations $\mathcal{M}_\delta$ have been already defined for all $\delta < \alpha$. Let

$$\mathcal{N}_\alpha = \bigcup_{0<\delta<\alpha} \mathcal{M}_\delta,$$

$$\Pi_\alpha = \frac{\Pi}{\mathcal{N}_\alpha}, S_\alpha = S(\Pi_\alpha), \text{ and } L_\alpha = L(\Pi_\alpha).$$

- If the program Π_α is empty, then the construction stops and $\mathcal{M}_\Pi = \mathcal{N}_\alpha$ is the *weakly perfect model* of Π.
- Otherwise, if the bottom stratum $S_\alpha = S(\Pi_\alpha)$ of Π_α is empty or if the least Herbrand model of the bottom layer $L_\alpha = L(\Pi_\alpha)$ of Π_α does not exist, then the construction also stops and $\mathcal{M}_\Pi = \mathcal{N}_\alpha$ is the *partial weakly perfect model* of Π.
- Otherwise, the partial interpretation $\mathcal{M}_\alpha$ is defined as the least Herbrand model of the bottom layer $L_\alpha = L(\Pi_\alpha)$ of Π_α and the construction continues.

In the first two cases, α is called the *breadth of* Π and is denoted by $\delta(\Pi)$. For $0 < \alpha < \delta(\Pi)$, the set S_α is called the α*-th stratum* of Π and the program L_α is called the α*-th layer of* Π.

In the process of constructing the strata S_α, some ground atoms may be eliminated by the reduction and not fall into any stratum. Such atoms should be added to an arbitrary stratum, e.g. the first, and assumed false in $\mathcal{M}_\Pi$. ♢

Note that the construction always stops after countably many steps and therefore the (partial) weakly perfect model $\mathcal{M}_\Pi$ of a program is always defined and unique.

Now, the class of weakly stratified programs is defined as those programs with a weakly perfect model, for which all the strata S_α consist only of trivial components.

Definition 3.1.10 (Weak Stratification). We say that a normal logic program Π is *weakly stratified* if it has a weakly perfect model and if all of its strata S_α consist only of trivial components or – equivalently – when all of its layers L_α are positive logic programs. In this case, we call the set of program's strata $\{S_\alpha \mid 0 < \alpha < \delta(\Pi)\}$ the *weak stratification* of Π. $\diamondsuit$

Example 3.1.11. Consider again program Π from Example 3.1.7. It is weakly stratified and has thus a unique answer set but it is not locally stratified. We already identified the bottom stratum $S(\Pi)$ and the bottom layer $L(\Pi)$, so we obtain:

$$\begin{aligned} \Pi_1 &= \Pi, \\ S_1 &= S(\Pi) = \{p(1,1), p(1,2), p(2,1), p(2,2)\}, \\ L_1 &= L(\Pi) = \{p(1,2)\}, \end{aligned}$$

and therefore

$$\mathcal{M}_1 = \{p(1,2), \overline{p(1,1)}, \overline{p(2,1)}, \overline{p(2,2)}\}.$$

Consequently, $\Pi_2 = \frac{\Pi_1}{\mathcal{M}_1} = \{q(1) \leftarrow \text{not } q(2)\}$ is the union of the minimal components of Π_2 and $L_2 = L(\Pi_2) = \emptyset$ is the set of rules from Π_2 whose heads belong to S_2. Therefore, $\mathcal{M}_2 = \{\overline{q(2)}\}$. As a result,

$$\begin{aligned} \Pi_3 &= \frac{\Pi_2}{\mathcal{M}_1 \cup \mathcal{M}_2} = \{q(1)\}, \\ S_3 &= \{q(1)\}, \\ L_3 &= \Pi_3, \text{ and} \\ \mathcal{M}_3 &= S_3 = \{q(1)\}. \end{aligned}$$

Since $\Pi_4 = \frac{\Pi_3}{\mathcal{M}_1 \cup \mathcal{M}_2 \cup \mathcal{M}_3} = \emptyset$, the construction is completed, Π is weakly stratified, $\{S_1, S_2, S_3\}$ is its weak stratification, and

$$\mathcal{M}_\Pi = \mathcal{M}_1 \cup \mathcal{M}_2 \cup \mathcal{M}_3 = \{p(1,2), q(1), \overline{p(1,1)}, \overline{p(2,1)}, \overline{p(2,2)}, \overline{q(2)}\}$$

is its unique weakly perfect model. $\diamondsuit$

We have seen that there exist weakly-stratified programs which are not locally stratified. With the following theorem we get that the weakly-stratified programs are a strict superclass of the locally-stratified programs.

Theorem 3.1.12 ([53]). *Every* (*locally*) *stratified program is weakly stratified.*

Analysis of Weak Stratification

In the above example the things are pretty clear: We can identify the whole program as deterministic and thus the generate part would be empty, but even after a slight modification, the situation

becomes much more complicated. Consider the program Π', a modification of the program Π from Example 3.1.11 which consists of the following rules:

$$\begin{aligned} q(X) &\leftarrow r(1), p(X,Y), \text{not } q(Y), \\ p(1,2) &\leftarrow, \\ r(1) &\leftarrow \text{not } r(2), \\ r(2) &\leftarrow \text{not } r(1). \end{aligned}$$

Its grounding $grd(\Pi')$ consists of the following rules:

$$q(1) \leftarrow p(1,1), r(1), \text{not } q(1), \tag{3.4}$$
$$q(1) \leftarrow p(1,2), r(1), \text{not } q(2), \tag{3.5}$$
$$q(2) \leftarrow p(2,1), r(1), \text{not } q(1), \tag{3.6}$$
$$q(2) \leftarrow p(2,2), r(1), \text{not } q(2), \tag{3.7}$$
$$p(1,2) \leftarrow, \tag{3.8}$$
$$r(1) \leftarrow \text{not } r(2), \tag{3.9}$$
$$r(2) \leftarrow \text{not } r(1). \tag{3.10}$$

Here, as in Example 3.1.11 above, the Rules (3.4)-(3.8) do still not lead to the creation of more than one answer set. Because of (3.9) and (3.10), we can choose whether $r(1)$ or $r(2)$ should be contained in an answer set, but independently from our choice, the Rules (3.4)-(3.8) do not create any more answer sets. It seems thus reasonable to not include the Rules (3.4)-(3.8) in the generate part.
We will now try to use the notion of weak stratification to identify the Rules (3.4)-(3.8) as a deterministic part of the program. If we ask the question, whether the program consisting of exactly those rules is weakly stratified, the answer is affirmative. This comes from the fact that in the process of constructing a weakly perfect model, $\{r(1)\}$ would be a minimal component and since it does not occur in a head, it would be considered false in $\mathcal{M}_1$. Hence, the Rules (3.4)-(3.7) are removed in the next reduction step, resulting in an empty program.

But, in some cases, the assumption that literals which do not occur in the heads of the examined rule set are false can lead to wrong conclusions. Consider for example the following rules:

$$a(1) \leftarrow r(1), \text{not } b(1), \tag{3.11}$$
$$b(1) \leftarrow r(1), \text{not } a(1), \tag{3.12}$$
$$r(1) \leftarrow \text{not } r(2), \tag{3.13}$$
$$r(2) \leftarrow \text{not } r(1). \tag{3.14}$$

If we ask whether the program consisting exactly of the Rules (3.11) and (3.12) is weakly stratified, the answer would again be affirmative, because $r(1)$ is assumed to be false. But if we consider all four rules, there can be answer sets in which $r(1)$ is true and in this case the Rules (3.11) and (3.12) obviously lead to the creation of two different answer sets.

So, if we examine some set of rules Π_x and ask the question whether this set is weakly stratified, we should also add all rules r for which it holds, that a head literal contained in Π_x depends on some literal in $head(r)$. For the program Π'_g given above, this would mean to ask whether the whole program Π'_g is weakly stratified.

It is easily seen that Π'_g is not weakly stratified since the minimal component $\{r(1), r(2)\}$ is not trivial. We can thus observe the following problem: If, in the process of constructing the strata S_α, some non-trivial component is minimal (and hence contained in the bottom stratum), the property of being weakly stratified is violated. Nevertheless, it may be the case that the actual rules under consideration form a program which does never lead to the creation of more than one answer set, like (3.4)-(3.8) in program Π'_g above.

The Bottom Reduct

Since the straightforward use of weak stratification leads to the mentioned problem, we will in the following adapt the original definitions by Przymusinski and Przymusinska to develop an approach which takes a program and calculates its so-called *bottom reduct*. We can then use the bottom reduct to check whether some set of rules is contained in the generate part of the program. This approach covers examples like the one from Π'_g. Furthermore, it is also applicable to disjunctive programs and, with a slight adaption which we will introduce later, even programs with aggregates in the head.

The idea of the layer-by-layer computation of a model will in the following be used. We will, nevertheless, not compute a model of the whole program but instead modify the existing program. The main idea is, to identify that part of the program which, intuitively spoken, contains the program's deterministic portion. We will then compute the unique answer set of this deterministic portion and use this answer set to modify the original program. The resulting program can then later be checked for the occurrence of unstratified negation to see whether it is deterministic.

Because of the problems with non-trivial components, we will drop the notion of *components*. Instead, we will just talk about literals and the order defined by the negative dependency relation. This gives rise to the notion of *weak minimality* of literals given in Definition 3.1.13.

Definition 3.1.13 (Weak Minimality). We say that a literal $L \in grd(\Pi)$ is *weakly minimal* if the following holds:

- there is no literal $L' \in lit(grd(\Pi))$ such that $L' < L$, and
- there is no literal $L' \in lit(grd(\Pi))$ which occurs in the head of a disjunctive rule such that $L' \leq L$.

$\diamondsuit$

Definition 3.1.6 is in the following adapted according to the notion of weak minimality. The new bottom stratum, which we call *deterministic-bottom stratum*, is then not the union of the minimal components but the union of the weakly-minimal literals. The non-trivial minimal components are thus ignored since their literals are non-minimal by definition.

Definition 3.1.14. By the *deterministic-bottom stratum*, $S(\Pi)$, of Π we mean the set of weakly-minimal literals which do not occur in the heads of disjunctive rules, i.e.,

$$S(\Pi) = \{L \in \mathit{lit}(\mathit{grd}(\Pi)) \mid L \text{ is weakly minimal and does not occur in the head of a disjunctive rule in } \Pi\}.$$

By the *deterministic-bottom layer* $L(\Pi)$ of Π we mean the corresponding subprogram of Π, i.e.

$$L(\Pi) = \bigcup\{r \in \mathit{grd}(\Pi) \mid \mathit{head}(r) \subseteq S(\Pi)\}.$$

$\diamondsuit$

Clearly, the deterministic-bottom layer $L(\Pi)$ is always a positive logic program since none of the head literals of $L(\Pi)$ depend negatively on other literals. It thus has a unique answer set.

Furthermore, if some literal L is not contained in the answer set of $L(\Pi)$, we can safely assume it to be false in any answer set of Π. This is due to the following (inductive) argument: By Definition 3.1.14, every literal in $S(\Pi)$ depends only on weakly-minimal literals which are thus also in $S(\Pi)$. But, since $L(\Pi)$ contains all rules whose heads are in $S(\Pi)$, every literal from $S(\Pi)$ is fully defined by $L(\Pi)$.

The following operator Φ is based on the reduction from Definition 3.1.8 (the only difference is that we do not require Π to be a *normal* logic program anymore, but the transformation itself stays exactly the same). It computes the unique answer set $\mathcal{M}$ of the bottom layer $L(\Pi)$ and then modifies $\mathit{grd}(\Pi)$ accordingly to obtain a new program $\Phi(\Pi)$. The operator Φ will be applied until a fixed point is reached.

Definition 3.1.15 (Operator Φ). Let Π be a ground logic program with bottom stratum $S(\Pi)$ and bottom layer $L(\Pi)$. Let furthermore $\mathcal{M}$ be the unique answer set of $L(\Pi)$ and $\mathcal{M}' = \mathcal{M} \cup \{\bar{L} \mid L \in S(\Pi) \text{ and } L \notin \mathcal{M}\}$. Then, $\Phi(\Pi) = \frac{\Pi}{\mathcal{M}'}$. We furthermore define $\Phi^{(0)}(\Pi) = \Pi$ and $\Phi^{(i)}(\Pi) = \Phi(\Phi^{(i-1)}(\Pi))$. $\diamondsuit$

Definition 3.1.16 (Bottom Reduct). Let Π be a logic program and n the smallest natural number such that $\Phi^{(n)}(\mathit{grd}(\Pi)) = \Phi^{(n+1)}(\mathit{grd}(\Pi))$. Then, $\Phi^{(n)}(\mathit{grd}(\Pi))$ is called the *bottom reduct* of Π. $\diamondsuit$

Example 3.1.17. Consider the program Π_7 which consists of the following rules:

$$\begin{aligned} \mathit{in}(X) &\leftarrow \mathit{succ}(Y, X), \mathit{dis}(Y), \text{not } \mathit{moving}(Y), \\ \mathit{moving}(X) &\leftarrow \mathit{in}(X), \\ \mathit{dis}(1) \vee \neg\mathit{dis}(1) &\leftarrow, \\ \mathit{succ}(1, 2) &\leftarrow . \end{aligned}$$

Its grounding consists of the following rules:

$$in(1) \leftarrow succ(1,1), dis(1), \text{not } moving(1), \tag{3.15}$$
$$in(1) \leftarrow succ(2,1), dis(2), \text{not } moving(2), \tag{3.16}$$
$$in(2) \leftarrow succ(1,2), dis(1), \text{not } moving(1), \tag{3.17}$$
$$in(2) \leftarrow succ(2,2), dis(2), \text{not } moving(2), \tag{3.18}$$
$$moving(1) \leftarrow in(1), \tag{3.19}$$
$$moving(2) \leftarrow in(2), \tag{3.20}$$
$$dis(1) \vee \neg dis(1) \leftarrow, \tag{3.21}$$
$$succ(1,2) \leftarrow . \tag{3.22}$$

For Π_7 we obtain as deterministic-bottom stratum the set $S(\Pi_7) = \{succ(1,1),\ succ(1,2),\ succ(2,1),\ succ(2,2)\}$. Note that $dis(1)$ and $\neg dis(1)$ are weakly minimal but since they appear in the head of a disjunctive rule, they are not in the deterministic-bottom stratum. The deterministic-bottom layer of the program is thus $L(\Pi_7) = \{succ(1,2)\}$, having the unique answer $\mathcal{M} = \{succ(1,2)\}$ and so $S(\Pi_7) \setminus \mathcal{M} = \{succ(1,1), succ(2,1), succ(2,2)\}$.

For computing $\Phi^{(1)}(\Pi_7)$, we will first remove all rules which contain a naf-literal N such that $\mathcal{M} \models \bar{N}$. In the above case these are (3.15), (3.16), and (3.18). Furthermore, (3.22) is removed and $succ(1,2)$ is removed from (3.17), since $\mathcal{M} \models succ(1,2)$. There is no non-fact whose head appears as a fact. We obtain $\Phi^{(1)}(\Pi_7)$ which consists of the following rules:

$$in(2) \leftarrow dis(1), \text{not } moving(1), \tag{3.23}$$
$$moving(1) \leftarrow in(1), \tag{3.24}$$
$$moving(2) \leftarrow in(2), \tag{3.25}$$
$$dis(1) \vee \neg dis(1) \leftarrow . \tag{3.26}$$

Now we compute again the deterministic-bottom stratum. The literals $moving(1)$ and $in(1)$ are the weakly-minimal literals which do not occur in the head of a disjunctive rule, hence $S(\Phi(\Pi_7)) = \{moving(1), in(1)\}$ and $L(\Phi(\Pi_7)) = \{moving(1) \leftarrow in(1)\}$ with the unique answer set $\mathcal{M}' = \emptyset$. Since $\mathcal{M}' \models \text{not } moving(1)$, we can remove $\text{not } moving(1)$ from (3.23). Furthermore, (3.24) is removed since $\mathcal{M}' \models \text{not } in(1)$. We get $\Phi^{(2)}(\Pi_7)$ as follows:

$$in(2) \leftarrow dis(1), \tag{3.27}$$
$$moving(2) \leftarrow in(2), \tag{3.28}$$
$$dis(1) \vee \neg dis(1) \leftarrow . \tag{3.29}$$

For $\Phi^{(2)}(\Pi_7)$ the deterministic-bottom stratum is empty, hence $\Phi^{(3)}(\Pi_7) = \Phi^{(2)}(\Pi_7)$ and thus $\Phi^{(2)}(\Pi_7)$ is the bottom reduct of Π_7. ♢

Theorem 3.1.18. *The bottom reduct of a weakly stratified program is empty.*

Proof. Let Π be a weakly stratified logic program. By the definition of weak stratification, Π has a weakly perfect model and thus there exists an n such that Π_n, which was constructed according to Definition 3.1.9, is empty. We will thus show that for any n, $\Pi_n = \Phi^{(n)}(\Pi)$.

First, since Π is weakly stratified, all of its strata S_α consist only of trivial components. Since the bottom stratum is defined as the union of the minimal components and the deterministic-bottom stratum is defined as the union of all weakly-minimal literals, the deterministic-bottom stratum is identical to the the bottom stratum. The same holds for the respective bottom layers.

Furthermore, for positive normal logic programs, the notions of the least Herbrand model and the unique signed set $\mathcal{M}'$ used in the definition of the operator Φ coincide. For a program Π we will thus in the following use $L(\Pi)$ to denote both its bottom stratum and its deterministic-bottom stratum. We will also use $LHM(\Pi)$ to denote both the unique signed set $\mathcal{M}'$ and the least Herbrand model of Π.

We proceed by induction on n.

INDUCTION BASE ($n = 0$): By definition, $\Phi^{(0)}(\Pi) = \Pi_0 = \Pi$.

INDUCTION STEP ($n > 0$): We assume that the statement holds for all $0 < i \leq n$. By definition we have that

$$\Phi^{(n+1)}(\Pi) = \Phi(\Phi^{(n)}(\Pi)) = \frac{\Phi^n(\Pi)}{LHM(L(\Phi^{(n)}(\Pi)))},$$

and

$$\Pi_{n+1} = \frac{\Pi}{\bigcup\limits_{0<i<n+1} LHM(L(\Pi_i))}.$$

But this is the same as

$$\frac{\frac{\Pi}{\bigcup\limits_{0<i<n} LHM(L(\Pi_i))}}{LHM(L(\Pi_n))}.$$

Now, by the definition of Π_n and the induction hypothesis it follows that

$$\frac{\Pi}{\bigcup\limits_{0<i<n} LHM(L(\Pi_i))} = \Pi_n = \Phi^{(n)}(\Pi).$$

But then,

$$\Pi_{n+1} = \frac{\Pi_n}{LHM(L(\Pi_n))} = \frac{\Phi^n(\Pi)}{LHM(L(\Phi^{(n)}(\Pi)))} = \Phi^{(n+1)}(\Pi).$$

□

Since every (locally) stratified program is weakly stratified, we get the following corollary:

Corollary 3.1.19. *The bottom reduct of a (locally) stratified program is empty.*

The following proposition tells us that there is no constant upper bound for the number of iterations of Φ.

Proposition 3.1.20. *For every $n \in \mathbb{N}^+$ there exists a program Π^n such that the computation of its bottom reduct requires n applications of the Φ-operator.*

Proof. Let Π^n be the program consisting of the following rules:

$$
\begin{aligned}
&step_1(a) \leftarrow, \\
&step_2(a) \leftarrow step_1(a), \text{not } step_1(b), \\
&step_2(b) \leftarrow \text{not } step_1(a), \\
&step_3(a) \leftarrow step_2(a), \text{not } step_2(b), \\
&step_3(b) \leftarrow \text{not } step_2(a), \\
&\qquad\vdots \\
&step_n(a) \leftarrow step_{n-1}(a), \text{not } step_{n-1}(b), \\
&step_n(b) \leftarrow \text{not } step_{n-1}(a).
\end{aligned}
$$

Π^n has the deterministic-bottom stratum $S(\Pi^n) = \{step_1(a), step_1(b)\}$ and thus the deterministic-bottom layer $L(\Pi^n) = \{step_1(a)\}$. The unique answer set of the deterministic-bottom layer is $\{step_1(a)\}$, hence $\Phi^{(1)}(\Pi^n)$ is as follows:

$$
\begin{aligned}
&step_2(a) \leftarrow, \\
&step_3(a) \leftarrow step_2(a), \text{not } step_2(b), \\
&step_3(b) \leftarrow \text{not } step_2(a), \\
&\qquad\vdots \\
&step_n(a) \leftarrow step_{n-1}(a), \text{not } step_{n-1}(b), \\
&step_n(b) \leftarrow \text{not } step_{n-1}(a).
\end{aligned}
$$

Now we get the deterministic-bottom stratum $S(\Phi^{(1)}(\Pi^n)) = \{step_2(a), step_2(b)\}$ and the bottom layer $L(\Phi^{(1)}(\Pi^n)) = \{step_2(a)\}$. We can observe that the computation of $\Phi^{(2)}(\Pi^n)$ is similar to that of $\Phi^{(1)}(\Pi^n)$ and that this is also the case for all $\Phi^{(i)}(\Pi^n)$ with $i < n$. We thus end up with $\Phi^{(n-1)}(\Pi^n)$, consisting of the single fact:

$$step_n(a) \leftarrow .$$

Now, after applying the Φ-operator once more, we obtain $\Phi^{(n)}(\Pi^n) = \emptyset$, hence $\Phi^{(n)}(\Pi^n)$ is the bottom reduct of Π^n. Its computation required n applications of the Φ-operator. □

We can now give a definition for the generate part of a logic program Π.

Definition 3.1.21 (Generate Part). A rule r belongs to the *generate part* if and only if r is disjunctive or a ground instance of r is involved in a negative cycle within the extended dependency graph of the bottom reduct of Π. Such a rule is referred to as a *generating rule*. ♢

By Theorem 3.1.18 we immediately get the following result:

Corollary 3.1.22. *The generate part of every weakly stratified and thus of every* (*locally*) *stratified program is empty.*

For programs which require an explicit input in the form of facts, the bottom reduct of a program should be formed by including some input instance to make sure that naf-literals which are true in the actual computation of the program are not wrongly considered false. Consider for example the following encoding Π_8 of the graph-3-colouring problem which consists of the following rules:

$$red(X) \leftarrow vertex(X), \text{not } green(X), \text{not } blue(X), \tag{3.30}$$
$$green(X) \leftarrow vertex(X), \text{not } red(X), \text{not } blue(X), \tag{3.31}$$
$$blue(X) \leftarrow vertex(X), \text{not } red(X), \text{not } green(X), \tag{3.32}$$
$$\leftarrow edge(X,Y), red(X), red(Y), \tag{3.33}$$
$$\leftarrow edge(X,Y), green(X), green(Y), \tag{3.34}$$
$$\leftarrow edge(X,Y), blue(X), blue(Y). \tag{3.35}$$

Here, for a graph (V, E), an encoded problem instance should contain the facts $\{vertex(u) \mid u \in V\}$ and $\{edge(u,v) \mid (u,v) \in E\}$. If we do not add these facts, then all ground instances of $vertex(X)$ and $edge(X,Y)$ in the bodies of the instances of (3.30)-(3.35) are contained in the deterministic-bottom stratum but are not true in the unique answer set of the bottom layer. Hence, they are considered false and thus the bottom reduct does not contain any rules. But this would mean that the first three rules are, by Definition 3.1.21, not rules of the generate part which would not be intuitive.

3.2 Definitions of Test and Define

The definitions for the define and test part are easier and are given below.

Definition 3.2.1 (Test Part). A rule belongs to the *test part* if and only if it is a constraint. ♢

Definition 3.2.2 (Define Part). A rule belongs to the *define part* if and only if it does neither belong to the generate nor to the test part. Such a rule is referred to as a *defining rule*. ♢

Example 3.2.3. Consider the program Π_9, consisting of the following rules:

$$q(3) \leftarrow \text{not } q(1), \text{not } q(4), \tag{3.36}$$
$$q(4) \leftarrow \text{not } q(1), \text{not } q(3), \tag{3.37}$$
$$q(1) \leftarrow p(1,2), \text{not } q(2), \tag{3.38}$$
$$q(2) \leftarrow p(2,1), \text{not } q(1), \tag{3.39}$$
$$p(1,2) \leftarrow . \tag{3.40}$$

Here we have the two negative cycles, $\{q(1), q(2)\}$ and $\{q(3), q(4)\}$. Nevertheless, since $p(2,1)$ is not contained in a rule head, it cannot be in an answer set. Hence, the Rule (3.39) is removed

during the computation of the bottom reduct. It follows then that $q(1)$ must be contained in an answer set, since $p(1,2)$ in the body of Rule (3.38) is true by (3.40) and $q(2)$ cannot be true. But then, the Rules (3.36) and (3.37) are removed during the computation of the bottom reduct.

It follows that the bottom reduct of Π_9 does not contain a negative cycle and hence all the above rules are classified as defining rules. This should match our intuition since the program has the unique answer set $\{p(1,2), q(1)\}$. In contrast, if we replace the fact $p(1,2)$ in Π_9 by the fact $p(2,1)$, we obtain the program Π_9' which consists of the following rules:

$$q(3) \leftarrow \text{not } q(1), \text{not } q(4), \tag{3.41}$$
$$q(4) \leftarrow \text{not } q(1), \text{not } q(3), \tag{3.42}$$
$$q(1) \leftarrow p(1,2), \text{not } q(2), \tag{3.43}$$
$$q(2) \leftarrow p(2,1), \text{not } q(1), \tag{3.44}$$
$$p(2,1) \leftarrow . \tag{3.45}$$

Here, $q(1)$ cannot be in an answer set and thus the Rules (3.41) and (3.42) are not removed during the computation of the bottom reduct. But, because of the negative cycle $\{q(3), q(4)\}$, they are generating rules. The other rules are still considered defining rules. This should again match our intuition, since Π_9' has the two answer sets $\{p(2,1), q(2), q(3)\}$ and $\{p(2,1), q(2), q(4)\}$. $\diamondsuit$

3.3 Implementation of the Classification Algorithm

In this section we introduce an implementation of an algorithm which takes as input a logic program and outputs a classification of the program rules into the *generate*, *define*, and *test parts*. Some parts of the algorithm (overall control of the data flow, parsing of an input program, etc.) are implemented in a procedural programming-language (Java), but the most complex part, namely the computation of the bottom reduct and the identification of rules which are involved in negative cycles within the bottom reduct, is done by a meta logic program on which we will lay the main focus here.

To be able to apply a meta logic program to some input program, it is necessary to transform the input program into a set of atoms. For this transformation, to which we will refer in the following as *reification*, we make use of the functionality `reify` which is provided by the well-known grounder `gringo` (version 3.0.5) and is explained in detail in a paper by Gebser, Kaminski and Schaub [28]. This implies that our implementation can cope with exactly those programs which are in the `gringo` input format. The solution step (i.e., the application of the meta program to the reified input program) is handled by the solver `clasp` (version 3.1.0). Before we go into further detail, we first give a sketch of the implementation of the whole classification algorithm as follows:

1. Read the input program Π, ground it, and transform it into a set S_Π of atoms (reification).
2. Compute the bottom reduct of Π by applying a meta logic program to S_Π.
3. Identify the set C of rules whose ground instances are involved in negative cycles within the extended dependency graph of the bottom reduct.

4. Classify rules from C and rules with disjunction or aggregates in the head as generating rules. Classify rules with empty heads as constraints (test part) and all the others as defining rules.

Steps 2 and 3 are fully handled by the meta logic program which we will later explain in detail. Step 1 is partly handled by it and the rest is handled by `gringo`, `clasp`, and our imperative program. Note that in Step 4 we also take aggregates into account which is not the case in the definitions of the preceding chapter.

We incorporated aggregates mainly because of practical considerations. They are a part of the `gringo` syntax and many programs from practice use aggregates as syntactical extensions which allow for the introduction of non-determinism. From a formal standpoint, this leads to a slight modification of Definitions 3.1.13, 3.1.14, and 3.1.21, instead of which we use the following Definitions 3.3.1, 3.3.2, and 3.3.3, respectively.

Definition 3.3.1 (Weak Minimality, `gringo` Syntax). We say that a literal $L \in lit(\Pi_g)$ is *weakly minimal* if the following holds.

- There is no literal $L' \in lit(\Pi_g)$ such that $L' < L$, and
- there is no literal $L' \in lit(\Pi_g)$ which occurs in the head of a disjunctive rule or in an aggregate within a rule head such that $L' \leq L$.

♢

The dependency notions and thus the relations $<$ and $\leq$ generalise naturally to literals in aggregates.

Definition 3.3.2 (Deterministic-Bottom Stratum/Layer, `gringo` Syntax). By the *deterministic-bottom stratum* $S(\Pi)$ of Π we mean the set of weakly-minimal literals which do not occur in the heads of disjunctive rules or within aggregates in rule heads.

By the *deterministic-bottom layer* we mean the set of rules from Π whose head atoms are contained in the deterministic-bottom stratum. ♢

Definition 3.3.3 (Generate Part, `gringo` Syntax). A rule r belongs to the generate part if and only if it fulfils at least one of the following criteria:

1. r is disjunctive.
2. r contains an aggregate in the head.
3. A ground instance of r is involved in a negative cycle within the extended dependency graph of the *bottom reduct* of Π.

♢

Note that we do not adapt the operator Φ. Next, we will first describe the reification which takes place before the meta program is called. After this we will give a detailed explanation of the meta program.

Reifiying the Input for the Meta Program

The whole reification is done in three main steps of which the first two are handled outside the meta program. In the first step, we add to every (non-fact) rule of an input program Π a positive body atom $rule\_number(n)$, where n is a natural number which uniquely identifies the corresponding rule. For the predicate $rule\_number$ we then add the statement "`#external rule_number/1`" to the program. This statement tells `gringo` that the respective atoms are not part of the input and can thus not be removed during the grounding process [27]. With this little "trick" we can later, for any ground rule r_g, obtain the corresponding non-ground rule r of which r_g is an instance. To illustrate this, consider the following example:

Example 3.3.4. Let Π_{10} be the program which consists of the following rules:

$$d \leftarrow, \tag{3.46}$$
$$\leftarrow e, \tag{3.47}$$
$$a \leftarrow \text{not } c, \tag{3.48}$$
$$f \vee g \leftarrow \neg h, \tag{3.49}$$
$$\{b, c\} \leftarrow e. \tag{3.50}$$

By adding to every non-fact rule a corresponding body atom with the predicate $rule\_number$, we get the program Π'_{10}, consisting of the following rules:

$$d \leftarrow, \tag{3.51}$$
$$\leftarrow e, rule\_number(1), \tag{3.52}$$
$$a \leftarrow rule\_number(2), \text{not } c, \tag{3.53}$$
$$f \vee g \leftarrow \neg h, rule\_number(3), \tag{3.54}$$
$$\{b, c\} \leftarrow e, rule\_number(4). \tag{3.55}$$

$\diamondsuit$

In the second step, the resulting program is passed to `gringo` which is called with the command "`gringo --reify input_filename`". This command tells `gringo`, that it should encode the grounding of the program as a set of facts. The most important predicate symbols for this encoding are $rule/2$, $set/2$, and $wlist/4$. Before we go into further detail, we give the following example:

Example 3.3.5. Consider program Π'_{10} from Example 3.3.4. It could be represented by `gringo` in the following way:

```
1 rule(pos(atom(d)),pos(conjunction(0))).
2 rule(pos(false),pos(conjunction(1))).
3 set(1,pos(atom(e))).
4 set(1,pos(rule_number(1))).
5 rule(pos(atom(a)),pos(conjunction(2))).
6 set(2,pos(rule_number(2))).
```

```
 7  set(2,neg(atom(c))).
 8  rule(pos(disjunction(3)),pos(conjunction(4)).
 9  set(3,pos(atom(f))).
10  set(3,pos(atom(g))).
11  set(4,pos(atom(-h))).
12  set(4,pos(rule_number(3))).
13  rule(pos(sum(0,5,2)),pos(conjunction(6))).
14  wlist(5,0,pos(atom(b)),1).
15  wlist(5,1,pos(atom(c)),1).
16  set(6,pos(atom(e))).
17  set(6,pos(atom(rule_number(4)))).
```

♢

The predicate $rule$ represents a rule as a pair, where the first and the second element represent the head and the body, respectively. Sets of literals, which can be interpreted as conjunctions or disjunctions, are represented by the predicate set, whose first element specifies an index of the set and the second element is a literal which is contained in the set. The elements within an aggregate are represented by the predicate $wlist$. The first element denotes the index of the aggregate, the second one denotes the index of a literal within the aggregate, the third element is the literal itself and the fourth one denotes the *weight* associated to the literal.

To encode literals, conjunctions, disjunctions and aggregate functions, function symbols are used. The function symbols neg and pos denote default negation or its absence. Literals are specified by the function symbol $atom$. Disjunction and conjunction are specified by the function symbols $disjunction$ and $conjunction$, respectively, and an index of a set which contains the corresponding literals. Aggregate functions can be encoded by the function symbols sum, min, max, $even$, and odd. Furthermore, note that empty rule bodies are encoded by $conjunction(n)$ where n is an index for which no set was specified.

The Meta Program for the Classification

The whole meta program, which we will denote as $\Pi_{classify}$, consists of nearly 100 rules, hence we will in the following not explain all of them but give an overview instead. The whole program can be found in Appendix A.

The program $\Pi_{classify}$ consists of the following modules:

(i) Π_{reify} which reifies the input,

(ii) $\Pi_{dependency}$ which defines the dependency notions,

(iii) $\Pi_{bottomlayer}$ which identifies the deterministic-bottom stratum and the corresponding deterministic-bottom layer,

(iv) $\Pi_{bottomreduct}$ which encodes the operator Φ and applies it successively to compute the bottom reduct,

(v) $\Pi_{generate}$ which finally identifies the rules which are involved in negative cycles within the extended dependency graph.

In the following, we outline these modules. For this, we use the `gringo` syntax in which "`:-`" is used for the symbol "$\leftarrow$".

Module Π_{reify} – Reification of the Input

Module Π_{reify} rewrites the $rule$- and set-facts obtained from `gringo` into atoms with the predicate symbols $irule/4$ and $iset/3$, respectively. The $irule$ and $iset$ predicates contain as a first parameter, additionally to the parameters of $rule$ and set, the index of the iteration in the computation of the bottom reduct, i.e., the index i for operator $\Phi^{(i)}$. This way, we can in every iteration refer to the program to which we apply the operator Φ. Furthermore, the atoms with the predicate symbol $rule\_number$ are thrown out of the corresponding sets and their parameters are used to encode the rule number as the second parameter of the $irule$ predicate. The following example demonstrates the last reification step.

Example 3.3.6. Consider the `gringo` output from Example 3.3.5. It is rewritten to the following program.

```
1  irule(0,-1,pos(atom(d)),pos(conjunction(0))).
2  irule(0,1,pos(false),pos(conjunction(1))).
3  iset(0,1,pos(atom(e))).
4  irule(0,2,pos(atom(a)),pos(conjunction(2))).
5  iset(0,2,neg(atom(c))).
6  irule(0,3,pos(disjunction(3)),pos(conjunction(4)).
7  iset(0,3,pos(atom(f))).
8  iset(0,3,pos(atom(g))).
9  iset(0,4,pos(atom(-h))).
10 irule(0,6,pos(sum(0,5,2)),pos(conjunction(6))).
11 wlist(5,0,pos(atom(b)),1).
12 wlist(5,1,pos(atom(c)),1).
13 iset(0,6,pos(atom(e))).
```

$\diamondsuit$

Note that the $wlist$ atoms are not indexed since they are not affected by the computation of the bottom reduct and thus stay the same over all iterations. The rewriting for the rules is straightforward. One of the rules which define the $irule$ predicate is given below:

```
1  irule(0,R,pos(atom(X)),pos(conjunction(Y))) :-
2          rule(pos(atom(X)),pos(conjunction(Y))),
3          set(Y,pos(atom(rule_number(R)))).
```

For the definition of the $iset$ predicate we must, apart from the straightforward rewriting, throw away the $rule\_number$ atoms from the bodies which is handled by the following two rules:

```
1  is_rule_numbering(S,pos(atom(rule_number(R)))) :-
2      set(S,pos(atom(rule_number(R)))).
3
```

```
4 iset(0,S,pos(atom(X))) :-
5     set(S,pos(atom(X))),
6     not is_rule_numbering(S,pos(atom(X))).
```

Finally, Π_{reify} also defines the predicates *disjunctive_head_literal*, *aggr_head_literal* and *literal* to identify the literals within disjunctions, heads of aggregates and all literals within the program, respectively. Based on the predicates defined in Π_{reify}, we can later identify the bottom layer and the bottom stratum as well as those rules which are involved within negative cycles in the bottom reduct.

Module $\Pi_{dependency}$ – Definition of the Dependency Notions

The definitions of the dependency notions are straightforward and thus we do not list the respective rules here. It is only important to note that we use the following predicate symbols:

- *depends_directly_positive* and *depends_positively* for direct positive and positive dependency, respectively,
- *depends_directly_negative* and *depends_negatively* for direct negative and and negative dependency, respectively, and
- *depends* for dependency in the common sense, no matter if negative or positive.

The predicates *depends_directly_positive* and *depends_directly_negative* contain four parameters: The first parameter specifies, similarly to *iset* and *irule*, the index of the iteration in the computation of the bottom reduct, the third (head) and the fourth parameter (body) specify the involved literals and the second parameter provides the information, via which rule the direct dependency of the two literals is established. Hence, $depends\_directly\_positive(I, R, H, B)$ encodes that in the I-th iteration the literal H depends directly positive on the literal B via the rule R. Similarly for $depends\_directly\_negative(I, R, H, B)$.

The predicates *depends_positively* and *depends_negatively* do naturally not have a parameter for the rule index, thus $depends\_positively(I, A, B)$ encodes that in iteration I, the literal A depends positively on B. Similarly for *depends_negatively* and *depends*. Finally, note that *depends_positively* and *depends_negatively* represent $\leq$ and $<$, respectively.

Module $\Pi_{bottomlayer}$ – Identification of the Deterministic Bottom Layer and Stratum

In module $\Pi_{bottomlayer}$, we define the weakly minimal literals, the deterministic-bottom stratum, and the deterministic-bottom layer. For this, we use the predicates *weakly_minimal_literal*, *bottom_stratum*, and *bottom_layer_rule*, respectively. The first four rules identify the weakly minimal literals:

```
1 -weakly_minimal_literal(I,L) :- literal(I,L),
2     depends_negatively(I,L,_).
3 -weakly_minimal_literal(I,L) :- literal(I,L),
4     disjunctive_head_literal(I,_,Y), depends(I,L,Y).
```

```
5  -weakly_minimal_literal(I,L) :- literal(I,L),
6      aggr_head_literal(I,_,Y), depends(I,L,Y).
7
8  weakly_minimal_literal(I,L) :- literal(I,L),
9      not -weakly_minimal_literal(I,L).
```

Note that the *weakly_minimal_literal* predicate has also the index of the iteration as first parameter. The deterministic-bottom stratum and the deterministic-bottom layer can then, according to Definition 3.3.2, be specified by the following four rules:

```
1  -bottom_stratum(I,L) :- disjunctive_head_literal(I,_,L).
2  -bottom_stratum(I,L) :- aggr_head_literal(I,_,L).
3
4  bottom_stratum(I,L) :- weakly_minimal_literal(I,L),
5      not -bottom_stratum(I,L).
6  bottom_layer_rule(I,pos(atom(H)),B) :-
7      irule(I,_,pos(atom(H)),B),
8      bottom_stratum(I,H).
```

Both predicates contain again as a first parameter the index of the iteration in the computation.

Module $\Pi_{bottomreduct}$ – Computation of the Bottom Reduct

Once we have the bottom layer, we have to compute its unique answer set to reduce the given program of index i according to the definition of the operator Φ in order to obtain the program of the next iteration (index $i + 1$). This reduction step is repeatedly applied until a fixed point is reached. For the details of the answer-set computation and the corresponding reduction, we refer to the full program in the appendix. Roughly speaking, we define which rules and sets should explicitly not be contained in the program for the next iteration by using the classical negation of $irule$ and $iset$. Based on this definition, we then define the program for the following iteration as follows:

```
1  irule(I+1,R,H,B) :- check(I),
2                      irule(I,R,H,B), not -irule(I+1,R,H,B).
3  iset(I+1,S,L) :- check(I),
4                   iset(I,S,L), not -iset(I+1,S,L).
```

The predicate $check$ is true for an integer i if the program should terminate at iteration i, based on the fixed-point condition (predicate $layer\_changed$) of the operator Φ and a constant ($max\_iterations$) which defines the maximum number of iterations. The definitions of $check$ and $layer\_changed$ are as follows:

```
1  layer_changed(I+1) :- irule(I,R,B,H), -irule(I+1,R,B,H).
2  layer_changed(I+1) :- iset(I,S,L), -iset(I+1,S,L).
3
4  check(I) :- layer_changed(I), I < max_iterations.
5  check(0).
```

When the fixed-point of the operator Φ is reached, i.e., when the program was not changed during the last iteration, we can define the bottom reduct (identified by the iteration index $bottom\_reduct$):

```
1 irule(bottom_reduct,R,X,Y) :- irule(I,R,X,Y),
2     layer_changed(I), not layer_changed(I+1).
3 iset(bottom_reduct,X,Y) :- iset(I,X,Y),
4     layer_changed(I), not layer_changed(I+1).
```

Module $\Pi_{generate}$ – Identification of Generating Rules

In the last module $\Pi_{generate}$, we first identify those rules which are involved in a negative cycle within the extended dependency graph of the bottom reduct as generating rules:

```
1 in_negative_cycle(R) :-
2     depends_directly_positive(bottom_reduct,R,X,Y),
3     depends_negatively(bottom_reduct,Y,X).
4 in_negative_cycle(R) :-
5     depends_directly_negative(bottom_reduct,R,X,Y),
6     depends(bottom_reduct,Y,X).
```

Finally, all rules which involved in a negative cycle are identified as generating rules by the predicate $generating\_rule$ which encodes the actual output of the program:

```
1 generating_rule(R) :- in_negative_cycle(R).
```

3.4 Example Programs

In the following, we give example classifications of two programs from logic programming practice to illustrate the usefulness of our approach.

Stable Marriage

The *stable marriage problem* is defined as follows [32, 47]: Given two distinct sets M and W (commonly referred to as *men* and *women*) where each element of M has assigned a score to every element of W and vice versa, find a *perfect stable matching* between the elements of M and W. A perfect stable matching is a bijection f from M to W such that there exist no elements $m \in M$ and $w \in W$ which are not related to each other by f but which assign a higher score to each other than to the elements to which they are related by f.
Consider the following classification of an encoding for the stable marriage problem.[1] The input is encoded by atoms with the predicate symbols $manAssignsScore$ and $womanAssignsScore$.

[1] Encoding by Francesco Ricca, Mario Alviano and Marco Manna. Retrieved from `http://asparagus.cs.uni-potsdam.de/encoding/show/id/14319`.

Generate

$$\begin{aligned} match(M,W) \leftarrow\ & manAssignsScore(M,Fv1,Fv2),\\ & womanAssignsScore(W,Fv3,Fv4),\\ & \text{not } nonMatch(M,W),\\ nonMatch(M,W) \leftarrow\ & manAssignsScore(M,Fv1,Fv2),\\ & womanAssignsScore(W,Fv3,Fv4),\\ & \text{not } match(M,W), \end{aligned}$$

Define

$$jailed(M) \leftarrow match(M,Fv1).$$

Test

$$\begin{aligned} &\leftarrow manAssignsScore(M,Fv1,Fv2), \text{not } jailed(M),\\ &\leftarrow match(M,W), match(M,W1), W \neq W1,\\ &\leftarrow match(M1,W), match(M,W), M \neq M1,\\ &\leftarrow match(M,W1), manAssignsScore(M,W,Smw), W1 \neq W,\\ &\quad manAssignsScore(M,W1,Smw1), Smw > Smw1,\\ &\quad match(M1,W), womanAssignsScore(W,M,Swm),\\ &\quad womanAssignsScore(W,M1,Swm1), Swm \geq Swm1, \end{aligned}$$

The classification was obtained by computing the bottom reduct of the program combined with an input instance. Since the first two rules are involved in a negative cycle, they are classified as generating rules.

15-Puzzle

The following variant of the *15-puzzle problem* is considered: Let a 4×4 grid be given where each cell is identified by its coordinates (x, y) with $1 \leq x, y \leq 4$. Out of the 16 cells, 15 contain a tile and each tile is numbered with a distinct number $1 \leq i \leq 15$. The task is to find a sequence of tile moves such that all tiles are in their goal position (see Figure 3.1). A tile at (x, y) can only be moved to (x', y') if (x', y') is empty and $|x - x'| + |y - y'| = 1$.

In the following encoding Π_{11} of the 15-puzzle,[2] we have also choice atoms. The initial positions are encoded by atoms with the predicate symbol $in0$. Facts are omitted for the sake of compactness.

[2] Slight modification of an encoding by Martin Gebser and Roland Kaminski which was retrieved from `http://asparagus.cs.uni-potsdam.de/encoding/show/id/5307`

10	9	14	6
15	5	3	8
7	1		13
12	11	2	4

	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Figure 3.1: A possible input configuration of the 15-puzzle (left) and the goal configuration (right).

Generate

$$1\{move(T,X,Y) : pos(X) : pos(Y)\}1 \leftarrow time(T), maxtime(M), T < M, \text{not } goal(T). \tag{3.56}$$

Define

$$goal(S) \leftarrow goal(T), time(T), maxtime(M), succ(S,T), T < M, \tag{3.57}$$

$$goal(T) \leftarrow in(T,X,Y,A) : in\_t(X,Y,A), time(T), \tag{3.58}$$

$$in(0,X,Y,A) \leftarrow in0(X,Y,A), \tag{3.59}$$

$$in(S,X,Y,0) \leftarrow move(T,X,Y), succ(S,T), \tag{3.60}$$

$$in(S,X_1,Y_1,A) \leftarrow in(T,X_1,Y_1,0), in(T,X_2,Y_2,A), move(T,X_2,Y_2), neighbour(X_1,Y_1,X_2,Y_2), entry(A), succ(S,T), A > 0, \tag{3.61}$$

$$in(S,X,Y,A) \leftarrow in(T,X,Y,A), pos(X), pos(Y), entry(A), A > 0, time(T), maxtime(M), T < M, succ(S,T), \text{not } move(T,X,Y), \text{not } goal(T), \tag{3.62}$$

$$neighbour(X,Y,X+1,Y) \leftarrow pos(X), pos(Y), pos(X+1), \tag{3.63}$$

$$neighbour(X,Y,X,Y+1) \leftarrow pos(X), pos(Y), pos(Y+1), \tag{3.64}$$

$$neighbour(X,Y,X,Y-1) \leftarrow pos(X), pos(Y), pos(Y-1), \tag{3.65}$$

$$neighbour(X,Y,X-1,Y) \leftarrow pos(X), pos(Y), pos(X-1). \tag{3.66}$$

Test

$$\leftarrow \text{not } goal(M), maxtime(M), \tag{3.67}$$

$$\leftarrow move(T,X_1,Y_1), \{in(T,X_2,Y_2,0) : neighbour(X_2,Y_2,X_1,Y_1)\}0. \tag{3.68}$$

Note that the Rules (3.56), (3.58), and (3.68) contain aggregate constructs with so-called *conditional atoms*. These are syntactical constructs allowed by `gringo` whose formal semantics are

not really important for this example. The only crucial thing is, that the aggregate in the head of Rule (3.56) causes non-determinism in the sense, that exactly one instance of $move(T, X, Y)$, for whose instantiation of X and Y the atoms $pos(X)$ and $pos(Y)$ are true, has to be contained in an answer set of Π_{11}. For further details, we refer to the user's guide of `gringo`, `clasp`, `clingo`, and `iclingo` [27].

Program Π_{11} should illustrate why the computation of the bottom reduct can make a difference. Consider for example the Rules (3.56), (3.58), and (3.60):

$$\begin{aligned} 1\{move(T, X, Y) : pos(X) : pos(Y)\}1 &\leftarrow time(T), maxtime(M), T < M, \text{not } goal(T), \\ goal(T) &\leftarrow in(T, X, Y, A) : in\_t(X, Y, A), time(T), \\ in(S, X, Y, 0) &\leftarrow move(T, X, Y), succ(S, T). \end{aligned}$$

Because of the cycle including the predicates $goal$, $move$, and in, there are ground literals which depend negatively on themselves within the ground instantiation of Π_{11}. But this is not the case in the bottom reduct of Π_{11}.

In the last rule, the atom $succ(S, T)$, whose ground instances are all contained in the bottom stratum, is only true if S is instantiated to the successor of T. Hence, only rules in which S is instantiated to the successor element of T, are contained in the bottom reduct of Π_{11}. But a cycle can only be closed if in all rules the variables S and T are mapped to the same constant. The same holds for the other rules which are involved in negative cycles and thus none of these rules are generating rules. The only generating rule is (3.56) because its head contains an aggregate.

3.5 An Explanation Order for Rules

If we want to obtain an understandable explanation of a program, the order in which the rules are explained is very important. One way to obtain a relatively useful explanation is to start with generating rules and their predicates (which may be defined in the define part) and then continue by explaining how the predicates of the generate part are constrained by rules from the test and possibly the define part. We will in the following develop a formalisation of this main idea which also takes some subtleties with respect to the rule order into account. Consider as an example the disjunctive program Π_{12} for the graph 3-colouring problem which consists of the following rules:

$$\begin{aligned} red(X) \vee green(X) \vee blue(X) &\leftarrow vertex(X), && (3.69) \\ &\leftarrow edge(X, Y), red(X), red(Y), && (3.70) \\ &\leftarrow edge(X, Y), green(X), green(Y), && (3.71) \\ &\leftarrow edge(X, Y), blue(X), blue(Y). && (3.72) \end{aligned}$$

Here, Rule (3.69) is the only generating rule and we would, according to the things stated above, start with this rule and then continue with the other three rules from the test part. A natural-language explanation with that order could be of the following form:

For any vertex, choose whether it is red, green, or blue, according to the following constraints:

- *There must not be an edge (x, y) such that both x and y are red.*
- *There must not be an edge (x, y) such that both x and y are green.*
- *There must not be an edge (x, y) such that both x and y are blue.*

This explanation is arguably clear, but things get a little more complicated if several generating rules, which may even be constrained via distinct constraints, are involved. Assume we want to colour the vertices of a graph in the colours red and blue but with the additional constraint that some of the vertices cannot be red or blue, expressed by the predicates $cannot\_be\_red$ and $cannot\_be\_blue$. A possible encoding is the program Π_{13} which consists of the following rules:

$$red(X) \vee \neg red(X) \leftarrow vertex(X), \tag{3.73}$$
$$blue(X) \vee \neg blue(X) \leftarrow vertex(X), \tag{3.74}$$
$$\leftarrow vertex(X), red(X), cannot\_be\_red(X), \tag{3.75}$$
$$\leftarrow vertex(X), blue(X), cannot\_be\_blue(X), \tag{3.76}$$
$$\leftarrow vertex(X), red(X), blue(X). \tag{3.77}$$

Here, we have the two generating rules, (3.73) and (3.74), which assign red or blue to a vertex. We then have three constraints of which (3.75) is only related to the atoms generated by (3.73) and (3.76) is only related to those generated by (3.74) while the last Rule (3.77) is related to atoms which are generated by both generating rules. We think that it leads to a useful explanation if we start by explaining the generating rule (3.73) together with the constraint (3.75). After this we would then explain (3.74) together with Rule (3.76) and at the end we would mention Rule (3.77) which is related to both generating rules. A natural language explanation with this order could be as follows:

For any vertex, choose whether it is red or not such that no vertex which cannot be red is red. For any vertex, choose whether it is blue or not such that no vertex which cannot be blue is blue. Additionally, it must not be the case that there is a vertex which is both blue and red.

In order to formalise the intuition behind all this, we make use of the rule graph of a program. We also need the notion of a *strongly-connected component*, which is defined as follows:

Definition 3.5.1 (Strongly-connected Component). Let G be a graph with vertex set U and edge set E. A strongly-connected component is a maximal subset of V such that for every two vertices $u, v \in V$, G contains a directed path from u to v and vice versa. $\diamondsuit$

Definition 3.5.2 (Reduction Graph). Let G be a graph, then the reduction graph G_R of G is defined as the pair (V', E'), where V' is the set of all strongly-connected components of G, and $(S, T) \in E'$ iff G contains an edge (s, t) with $s \in S$ and $t \in T$. $\diamondsuit$

For a strongly-connected component S of a graph G, we say that S is *maximal* iff it does not have an outgoing edge within the reduction graph of G. For a vertex v which is contained in a maximal component, we say that v is maximal. Let Π be a logic program with rule graph

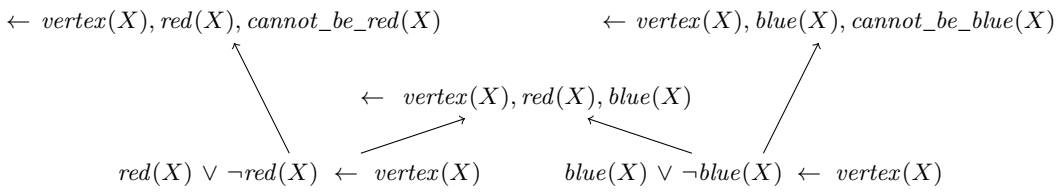

Figure 3.2: The rule graph of Π_{13}.

$RG(\Pi)$ and r_G a generating rule which is contained in the strongly-connected component S of $RG(\Pi)$. We say that r_G is a *maximal generating rule* iff there is no generating rule r'_G in a strongly-connected component $T \neq S$ such that $RG(G)$ contains a directed path from r_G to r'_G and r_G contains the most outgoing edges of all generating rules in S. Program Π_{13} showed that there can be constraints which are associated to certain generating rules. This is formalised by the following definition:

Definition 3.5.3. Let Π be a logic program with rule graph $RG(\Pi)$. Let furthermore r_C be a constraint in Π. For a rule r, we say that r is constrained by r_C iff $RG(\Pi)$ contains a directed path from r to r_C. ♢

Note that constraints are trivially constrained by themselves.

Example 3.5.4. Consider again program Π_{13}. The rule graph of Π_{13} is given in Figure 3.2. Here, for the constraint (3.75), we have that (3.73) is the only generating rule which it constrains. For (3.76), Rule (3.74) is the only generating rule it constrains. Furthermore, (3.77) constrains both (3.73) and (3.74). ♢

Algorithm 1 on page 42 assigns to every rule a natural number which defines the order in which the rules are explained. The intuition behind the algorithm is that we explain rules in the order in which they are visited during a depth-first traversal of the rule graph. During the depth-first traversal, rules which are in the same strongly-connected component are preferred and, additionally, as soon as all the rules from which a generating rule r_G is reachable have been visited, the constraints which constrain r_G are explained. If there are unconstrained rules, we start with a maximal one (this is for example the case in programs without constraints, but also when concepts are defined in terms of other concepts from the generate part). If there is no such rule, we start with a maximal generating rule. Otherwise, we choose an arbitrary rule which has not yet been visited.

The following example illustrates the behaviour of Algorithm 1.

Example 3.5.5. Consider again program Π_{13}. Since Π_{13} does not contain an unconstrained rule, we start with one of the two generating rules, say (3.73). It has no ingoing edges but after it is labelled, we get that all rules which are constrained by (3.75) are labelled, hence (3.75) is visited next. After this, we continue with the next maximal unlabelled generating rule (3.74). Again, it has no ingoing edges but after it has been labelled, (3.76) is visited since all rules which are constrained by (3.76) are labelled. After this, the Rule (3.77) is visited, since now all two

```
input  : rule graph G
output: order in which the rules are explained, defined by order
let sameSCC and otherSCC be stacks
let i=1
while G contains unlabelled rules do
    if there is an unlabelled unconstrained rule then
        let r be a maximal unlabelled unconstrained rule
    else if there is an unlabelled generating rule then
        let r be a maximal unlabelled generating rule
    else
        let r be an arbitrary maximal unlabelled rule
    push(sameSCC, r)
    while sameSCC and otherSCC are not both empty do
        if sameSCC is not empty then
            set r = pop(sameSCC)
        else
            set r = pop(otherSCC)
        if r is not labelled then
            label r as visited and set order(r) = i
            set i = i + 1
            // Iterate over the r' in the order in which
            // they appear in the body of r.
            for all edges (r', r) in G do
                if r' is in the same SCC as r then
                    push(sameSCC, r')
                else
                    push(otherSCC, r')
            Let l be the list of all unlabelled constraints, ordered by the number of rules which they
            constrain, in descending order
            // Start with the rule c which constrains the most rules.
            for all rules c in l do
                if all rules which are constrained by c are labelled then
                    push(otherSCC, c)
```

Algorithm 1: Computes the order in which rules are explained.

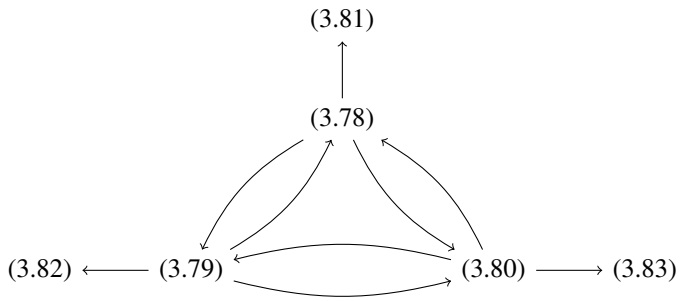

Figure 3.3: The rule graph of Π_8.

rules which it constrains have been labelled. Note that (3.77) is visited after (3.76) because of the fact that (3.77) constrains more rules than (3.76). We get the following order:

$$\begin{aligned}
red(X) \vee \neg red(X) &\leftarrow vertex(X), && 1\\
blue(X) \vee \neg blue(X) &\leftarrow vertex(X), && 2\\
&\leftarrow vertex(X), red(X), cannot_be_red(X), && 3\\
&\leftarrow vertex(X), blue(X), cannot_be_blue(X), && 4\\
&\leftarrow vertex(X), red(X), blue(X). && 5
\end{aligned}$$

This is exactly the order we wanted to obtain at the beginning of this section. ♢

Example 3.5.6. Finally, consider again the graph-3-colouring program Π_8 with the following rules:

$$\begin{aligned}
red(X) &\leftarrow vertex(X), \text{not } green(X), \text{not } blue(X), && (3.78)\\
green(X) &\leftarrow vertex(X), \text{not } red(X), \text{not } blue(X), && (3.79)\\
blue(X) &\leftarrow vertex(X), \text{not } red(X), \text{not } green(X), && (3.80)\\
&\leftarrow edge(X,Y), red(X), red(Y), && (3.81)\\
&\leftarrow edge(X,Y), green(X), green(Y), && (3.82)\\
&\leftarrow edge(X,Y), blue(X), blue(Y). && (3.83)
\end{aligned}$$

The rule graph of Π_8 is given in Figure 3.3. Here, first the three generating rules (3.78)-(3.80), which are in the same strongly-connected component, are visited. After this, the three remaining constraints are visited, again leading to the desired order. ♢

Example 3.5.7. Consider the stable-marriage encoding from page 36. There again, the generating rules are explained before the corresponding constraints. The generate part consists of the following two rules:

$$\begin{aligned} match(M,W) \leftarrow\ & manAssignsScore(M,Fv1,Fv2) && (3.84)\\ & womanAssignsScore(W,Fv3,Fv4),\\ & \text{not } nonMatch(M,W),\\ nonMatch(M,W) \leftarrow\ & manAssignsScore(M,Fv1,Fv2), && (3.85)\\ & womanAssignsScore(W,Fv3,Fv4),\\ & \text{not } match(M,W). \end{aligned}$$

All the constraints refer to the predicate $match$ while the predicate $nonMatch$ is not constrained. Because of this, Rule (3.84) has more outgoing edges than (3.85) and thus (3.84) is by definition maximal while (3.85) is not. Hence, (3.84) is explained before (3.85) which matches again our intuition. ♢

The examples should show that our approach leads to intuitive explanation orders. Additionally, it is flexible, since one could easily switch from a depth-first traversal to a breadth-first traversal of the rule graph. One would just have to use queues instead of stacks in Algorithm 1 and introduce a queue for constraints and their ingoing nodes, which has to be separately managed in order to make sure that constraints are explained immediately after all the generating rules they constrain.

CHAPTER 4

Equivalence-Preserving Program Transformations

In this chapter, we deal with equivalence-preserving transformations both on propositional and non-ground programs without strong negation. Various transformations have been developed in the past [5, 19, 20, 50] and they can serve different purposes such as optimisation, simplification, etc. Our goal is to use program transformations to (i) simplify programs and (ii) preprocess programs for a translation into (controlled) natural language.

We will start by introducing the most common notions of equivalence in logic programming and listing a table of propositional transformations which was originally presented by Eiter, Fink, Tompits and Woltran [20]. We will shortly discuss those transformations before we take a closer look on *partial evaluation* (*GPPE*), the most interesting rule for our purposes, also known as *partial deduction* or *unfolding*.

Partial evaluation was introduced into logic programming by Komorowski in 1981 [33, 44]. In its most simple form of application, it allows to replace a positive body atom of a rule r with its defining body. Consider for example the following program:

$$a \leftarrow b$$
$$b \leftarrow c, \text{not } d$$

GPPE transforms this program into the following program:

$$a \leftarrow c, \text{not } d$$
$$b \leftarrow c, \text{not } d$$

As we will see later, for body atoms which are defined by various rules, GPPE adds more than one new rule. In the presence of disjunction, things get a little more involved too.

Unfortunately, GPPE does only preserve ordinary equivalence, which restricts its range of applicability. Because of this, we use a notion of equivalence – originally called (*non-ground*)

relativised uniform equivalence [49, 70] but in the following, for the ease of notation, denoted as *input equivalence* – which is stronger than ordinary equivalence but weaker than uniform equivalence. We then introduce a generalisation of GPPE, called GPPE$_I$, that – as we show – preserves input equivalence, which allows us to apply GPPE$_I$ to a wide range of programs from logic programming practice.

In many cases we have to deal with non-ground programs since most of the practical programs are non-ground. To this end, we define an intuitive generalisation of GPPE$_I$ to the non-ground case and, as the main result of the chapter, prove that it preserves input equivalence. An easy corollary of this is that it preserves ordinary equivalence too.

Consider as a motivating example again the following two rules which we already presented in the introduction of this thesis:

$$\begin{aligned} \mathit{a\_row\_is\_not\_filled} &\leftarrow \mathit{row}(X), \text{not } \mathit{row\_is\_filled}(X), \\ &\leftarrow \mathit{a\_row\_is\_not\_filled}. \end{aligned}$$

We already argued in the introductory chapter that the transformation of these two rules into a single rule could lead to a much clearer translation into natural language. GPPE$_I$ allows us to do exactly this, as long as some restrictions, which we will discuss later, are met. Hence, we can transform the above two rules into the following rule:

$$\leftarrow \mathit{row}(X), \text{not } \mathit{row\_is\_filled}(X).$$

But not only a natural-language translation of the resulting rule would be clearer, also the program itself may be easier to understand after the application of this transformation. Although there already exist generalisations of GPPE to the non-ground case, they all have some drawbacks, most notably none of them preserves a stronger notion than ordinary equivalence. It will also be seen in the this chapter, that in the non-ground case the disjunction in rule heads leads to several issues which do not occur in the propositional setting.

4.1 Propositional Program Transformations

Various notions of equivalence between logic programs have been studied in the past: Apart from (*ordinary*) *equivalence*, which requires that two programs have the answer sets, *strong equivalence* [12, 39, 42, 52, 65, 66] and *uniform equivalence* [18, 46] are probably the most important ones.

Definition 4.1.1 ([20]). Let Π_a and Π_b be two logic programs. Then,

(i) Π_a and Π_b are (*ordinarily*) *equivalent*, denoted $\Pi_a \equiv_o \Pi_b$, iff they have the same answer sets.

(ii) Π_a and Π_b are *uniformly equivalent*, denoted $\Pi_a \equiv_u \Pi_b$, iff, for any set of facts Π_f, the programs $\Pi_a \cup \Pi_f$ and $\Pi_b \cup \Pi_f$ are equivalent.

(iii) Π_a and Π_b are *strongly equivalent*, denoted $\Pi_a \equiv_s \Pi_b$, iff, for any program Π, the programs $\Pi_a \cup \Pi$ and $\Pi_b \cup \Pi$ are equivalent. ♢

Name	Condition	Transformation
TAUT	$head(r) \cap body^+(r) \neq \emptyset$	$\Pi' = \Pi \setminus \{r\}$
RED$^+$	$A \in body^-(r_1)$,$\nexists r_2 \in \Pi$: $A \in head(r_2)$	$\Pi' = \Pi \setminus \{r_1\} \cup \{r'\}^\dagger$
RED$^-$	$head(r_2) \subseteq body^-(r_1)$, $body(r_2) = \emptyset$	$\Pi' = \Pi \setminus \{r_1\}$
CONTRA	$body^+(r) \cap body^-(r) \neq \emptyset$	$\Pi' = \Pi \setminus \{r\}$
NONMIN	$head(r_2) \subseteq head(r_1)$, $body^+(r_2) \subseteq body^+(r_1)$, $body^-(r_2) \subseteq body^-(r_1)$	$\Pi' = \Pi \setminus \{r_1\}$
S-IMP	$r, r' \in \Pi, r \lhd r'$ (see Definition 4.1.3)	$\Pi' = \Pi \setminus \{r'\}$
LSH	$r \in \Pi$ is head-cycle free in Π, $head(r)$ contains at least two distinct atoms	$\Pi' = \Pi \setminus \{r\} \cup r^{\rightarrow\dagger\dagger}$
GPPE	$A \in body^+(r_1), G_A \neq \emptyset$, for $G_A = \{r_2 \in \Pi \mid A \in head(r_2)\}$	$\Pi' = \Pi \setminus \{r_1\} \cup G'_A{}^\ddagger$
WGPPE	same condition as for GPPE	$\Pi' = \Pi \cup G'_A$

$^\dagger r' : head(r_1) \leftarrow body^+(r_1) \cup \text{not } (body^-(r_1) \setminus \{A\})$.
$^\ddagger G'_A = \{head(r_1) \cup (head(r_2) \setminus \{A\}) \leftarrow (body^+(r_1) \setminus \{A\}) \cup \text{not } body^-(r_1) \cup body(r_2) \mid r_2 \in G_A\}$.
$^{\dagger\dagger} r^{\rightarrow} = \{A \leftarrow body(r), \text{not } (head(r) \setminus \{A\}) \mid A \in head(r)\}$

Table 4.1: Local syntactic transformation rules

We say that a binary relation ρ is *stronger* than a binary relation ρ', iff $\rho \subseteq \rho'$, likewise we say that ρ is *strictly stronger* than ρ', iff $\rho \subset \rho'$. For the above mentioned equivalence relations we have that $\equiv_s$ is strictly stronger than $\equiv_u$, and $\equiv_u$ is strictly stronger than $\equiv_o$ which is easily demonstrated: If we restrict the set of context programs Π in the definition of strong equivalence from arbitrary programs to sets of facts, we obtain uniform equivalence. If we further restrict the context programs by only allowing the empty program as a context program, we arrive at ordinary equivalence.

The programs $\Pi_a = \{a \leftarrow \text{not } b, a \leftarrow b\}$ and $\Pi_b = \{a \leftarrow \text{not } c, a \leftarrow c\}$, taken from Eiter and Fink [18], demonstrate that strong equivalence is strictly stronger than uniform equivalence. Let $\Pi = \{b \leftarrow a\}$, then $\Pi_a \cup \Pi$ has no answer set while $\Pi_b \cup \Pi$ has the answer set $\{a, b\}$. However, one can easily check that for any set of facts Π_f, $\Pi_a \cup \Pi_f$ and $\Pi_b \cup \Pi_f$ are equivalent.

To show that uniform equivalence is strictly stronger than ordinary equivalence, consider the programs $\Pi_a = \{a \leftarrow \text{not } b\}$ and $\Pi_b = \{a\}$. They both have (only) the answer set $\{a\}$, but for $\Pi_f = \{b\}$ we get that $\Pi_a \cup \Pi_f$ has only the answer set $\{b\}$ while $\Pi_b \cup \Pi_f$ has only the answer set $\{a, b\}$.

Table 4.1, originally presented by Eiter et al. [20], gives an overview over some local transformation rules considered in the following. Next to the name of a transformation we state the kind of equivalence preserved by the transformation. If not mentioned otherwise, the equivalence results are taken from Eiter et al. [20]

TAUT (elimination of tautologies, preserves strong equivalence). This rule allows to remove tautological rules, i.e., rules which contain one and the same atom both in the head and the

positive body. Consider as an example the following rules which can be removed by TAUT:

$$a \leftarrow a, \tag{4.1}$$
$$a \vee b \leftarrow b, c, \tag{4.2}$$
$$a \vee b \vee c \leftarrow b, c, d, e. \tag{4.3}$$

RED$^+$ (positive reduction, preserves only ordinary equivalence [6]). If the body of a rule r contains a default-negated atom A which does not occur in the head of any other rule, then A is removed from the negative body of r. Consider for example the program Π_{14} which consists of the following rules:

$$a \vee b \leftarrow c, d, \text{not } e, \text{not } f, \tag{4.4}$$
$$c \leftarrow, \tag{4.5}$$
$$f \leftarrow . \tag{4.6}$$

Applying RED$^+$ to Rule (4.4) gives program Π'_{14}, consisting of the following rules:

$$a \vee b \leftarrow c, d, \text{not } f, \tag{4.7}$$
$$c \leftarrow, \tag{4.8}$$
$$f \leftarrow . \tag{4.9}$$

In Π'_{14}, the Rule (4.7) was obtained from (4.4) by removing the negative body atom e, which was not contained in a head of a rule.

RED$^-$ (negative reduction, preserves strong equivalence). If all the head atoms of a fact r_2 are contained in the negative body of a rule r_1, then r_1 is removed. Consider program Π_{15} which consists of the following rules:

$$a \leftarrow \text{not } b, \tag{4.10}$$
$$b \leftarrow, \tag{4.11}$$
$$c \leftarrow \text{not } d, \text{not } e, \text{not } f, \tag{4.12}$$
$$d \vee e \vee f \leftarrow . \tag{4.13}$$

Here, the application of RED$^-$ allows to remove (4.10) and (4.12).

CONTRA (elimination of contradictions, preserves strong equivalence). If the body of a rule r contains both an atom and its default negation, then the rule is removed. Examples of such rules are (4.14) and (4.15) below:

$$a \leftarrow b, \text{not } b, \tag{4.14}$$
$$a \vee b \leftarrow c, d, e, \text{not } d. \tag{4.15}$$

NONMIN (elimination of non-minimal rules, preserves strong equivalence). A rule r which is *implied* by some other rule r' (and thus *non-minimal*) is removed.

Definition 4.1.2 (Implication [5]). A rule r *implies* a rule r' iff $head(r) \subseteq head(r')$, $body^+(r) \subseteq body^+(r')$ and $body^-(r) \subseteq body^-(r')$. ♢

Consider as an example the program Π_{16}, consisting of the following rules:

$$a \vee b \vee c \leftarrow d, e, \text{not } f, \tag{4.16}$$
$$a \vee b \leftarrow d, e, \text{not } f. \tag{4.17}$$

Rule (4.16) is implied by (4.17), hence we can apply NONMIN and remove (4.16). Another example is the program Π_{17} which consists of the rules:

$$a \vee b \vee c \leftarrow d, e, \text{not } f, \tag{4.18}$$
$$b \vee c \leftarrow e, \text{not } f. \tag{4.19}$$

Here we can apply NONMIN to remove (4.18). Note that in the propositional case, implication is the same as *subsumption*, a concept well-known in the field of automated theorem proving [45].

S-IMP (s-implication, preserves strong equivalence). S-IMP was originally introduced by Wang and Zhou [68] with the goal of strengthening the notion of implication, which may explain the name. We will first give the definition of s-implication.

Definition 4.1.3 (s-implication). A rule r' is an *s-implication* of a rule $r \neq r'$, symbolically $r \lhd r'$, iff there exists a set $A \subseteq body^-(r')$ such that

(i) $head(r) \subseteq head(r') \cup A$,

(ii) $body^-(r) \subseteq body^-(r') \setminus A$, and

(iii) $body^+(r) \subseteq body^+(r')$.

♢

In other words, a rule r s-implies a rule r' if, by shifting some set of negative body atoms of r' to its head, we can obtain a rule which is implied by r. The following example is by Wang and Zhou [68]. Consider the program Π_{18} which consists of the following rules:

$$a \vee b \leftarrow \text{not } c, \tag{4.20}$$
$$b \leftarrow \text{not } c. \tag{4.21}$$

Here we can conclude that a is not in an answer set. In the presence of the Rule (4.21), the Rule (4.20) does not contain any further information and since it is implied by (4.21), we can remove it by an application of NONMIN. Consider now in contrast the program Π'_{18}, consisting of the following rules:

$$a \vee b \leftarrow \text{not } c, \tag{4.22}$$
$$b \vee c \leftarrow . \tag{4.23}$$

Again, we can conclude that a is not in an answer set and that in the presence of the Rule (4.23), the Rule (4.22) does not contain any further information. However, Rule (4.23) does not imply (4.22), but it s-implies (4.20), hence (4.20) can be removed by one application of S-IMP.

LSH (local shifting, preserves uniform but not strong equivalence). Local shifting allows to replace a head-cycle free disjunctive rule by a set of definite rules. As an example consider the program Π_{19} whose rules are given below:

$$a \vee b \leftarrow e, \tag{4.24}$$
$$c \vee d \leftarrow f, \tag{4.25}$$
$$c \leftarrow d, \tag{4.26}$$
$$d \leftarrow c. \tag{4.27}$$

Let r be the Rule (4.24), then $r^{\rightarrow} = \{a \leftarrow e, \text{not } b, \quad b \leftarrow e, \text{not } a\}$. We thus get the modified program Π'_{19}, consisting of the following rules:

$$a \leftarrow e, \text{not } b, \tag{4.28}$$
$$b \leftarrow e, \text{not } a, \tag{4.29}$$
$$c \vee d \leftarrow f, \tag{4.30}$$
$$c \leftarrow d, \tag{4.31}$$
$$d \leftarrow c. \tag{4.32}$$

Note that the Rule (4.25) is not head-cycle free in Π_{19} since c and d mutually depend on each other, hence LSH cannot be applied to this rule.

GPPE (partial evaluation, preserves only ordinary equivalence [5]). Partial evaluation replaces an atom A in the body of a rule by its defining bodies, thereby creating a new rule for each rule which defines A. If A is contained in the head of a disjunctive rule, the head atoms which are distinct from A are added to the resulting rule. Consider for example program Π_{20} which consists of the following rules:

$$a \leftarrow b, \text{not } c, \tag{4.33}$$
$$b \leftarrow d, \text{not } e. \tag{4.34}$$

Let $r_1 =$ (4.33) and $r_2 =$ (4.34), then $G_b = \{r_2\}$. We get $G'_b = \{a \leftarrow d, \text{not } e, \text{not } c\}$ and thus:

$$a \leftarrow d, \text{not } e, \text{not } c, \tag{4.35}$$
$$b \leftarrow d, \text{not } e. \tag{4.36}$$

Suppose we add the fact b to both Π_{20} and Π'_{20}. Then, $\Pi_{20} \cup \{b\}$ has the answer set $\{a, b\}$ while $\Pi'_{20} \cup \{b\}$ has the answer set $\{b\}$, hence GPPE does not preserve uniform equivalence. Another more complex example is program Π_{21}, whose rules are as follows:

$$a \vee f \leftarrow b, \text{not } c, \tag{4.37}$$
$$b \vee g \leftarrow d, \text{not } e, \tag{4.38}$$
$$b \vee h \leftarrow i, \text{not } j. \tag{4.39}$$

Let r_1 be the Rule (4.44), r_2 the Rule (4.38), and r_3 the Rule (4.39). Then, $G_b = \{r_2, r_3\}$. We get

$$G'_b = \{a \vee f \leftarrow d, \text{not } e, \text{not } c, \\ a \vee g \leftarrow h, \text{not } i, \text{not } c\}.$$

Hence, the reduced program Π'_{21} consists of the following rules:

$$a \vee f \vee g \leftarrow d, \text{not } e, \text{not } c, \tag{4.40}$$
$$a \vee f \vee h \leftarrow i, \text{not } j, \text{not } c, \tag{4.41}$$
$$b \vee g \leftarrow d, \text{not } e, \tag{4.42}$$
$$b \vee h \leftarrow i, \text{not } j. \tag{4.43}$$

Again, one can easily see that the addition of the fact b leads to different answer sets for $\Pi_{21} \cup \{b\}$ and $\Pi'_{21} \cup \{b\}$. While $\Pi_{21} \cup \{b\}$ has the answer sets $\{b, a\}$ and $\{b, f\}$, $\Pi'_{21} \cup \{b\}$ has only the answer set $\{b\}$.

WGPPE (weak partial evaluation, preserves strong equivalence). This transformation is similar to GPPE, with the only difference that r_1 is not removed. For program Π_{21}, this yields the program Π''_{21}, consisting of the following rules:

$$a \vee f \leftarrow b, \text{not } c, \tag{4.44}$$
$$a \vee f \vee g \leftarrow d, \text{not } e, \text{not } c, \tag{4.45}$$
$$a \vee f \vee h \leftarrow i, \text{not } j, \text{not } c, \tag{4.46}$$
$$b \vee g \leftarrow d, \text{not } e, \tag{4.47}$$
$$b \vee h \leftarrow i, \text{not } j. \tag{4.48}$$

Here, $\Pi_{21} \cup \{b\}$ and $\Pi''_{21} \cup \{b\}$ have the same answer sets. This is because the fact b makes the body of r_1 true and thus one of a and f has to be contained in an answer set of $\Pi''_{21} \cup \{b\}$. In the program $\Pi'_{21} \cup \{b\}$ from the GPPE example above this is not the case, since r_1 was removed by the transformation.

4.2 Input Equivalence

We have already seen that there exist transformations, most prominently GPPE, that preserve ordinary equivalence but not uniform equivalence, which restricts their applicability. In logic programming practice many programs require a set of facts as so-called *input instances* – e.g., a graph for the graph-colouring problem. In such cases, a program which is passed to a solver is of the form $\Pi \cup \Pi_f$, where Π is the actual encoding of a specific problem and Π_f is an input instance. Hence, transformations which do not preserve uniform equivalence cannot be carelessly applied to Π.

In order to be able to safely apply transformations like GPPE to such programs, we will use a kind of equivalence that we denote as *input equivalence*. Our goal is then, to introduce a

generalised version of GPPE and show that it preserves input equivalence. In the counterexample which showed that GPPE does not preserve uniform equivalence (program Π_{20}), a fact that is derived in the head of a rule which is involved in the transformation is added. For a similar example consider the program Π_{22} which consists of the following rules:

$$a \leftarrow b, \tag{4.49}$$

$$b \leftarrow c. \tag{4.50}$$

Here, the application of GPPE yields the program Π'_{22}, consisting of the following rules:

$$a \leftarrow c, \tag{4.51}$$

$$b \leftarrow c. \tag{4.52}$$

The two programs are ordinarily equivalent but not uniformly equivalent since the program $\Pi_{22} \cup \{b\}$ has the answer set $\{a, b\}$ while the program $\Pi'_{22} \cup \{b\}$ has the answer set $\{b\}$. But if we restrict the context of possible facts which may be added to Π and Π', to all sets of facts I with $b \notin I$, then it is easily seen that the two programs are uniformly equivalent in this restricted sense.

In many cases, one knows the set of possible input atoms of a program. The idea of input equivalence is thus to consider equivalence with respect to the addition of facts from some fixed set of (input) atoms.

Definition 4.2.1 (Input Equivalence). Let Π_a, Π_b be logic programs and I a set of (input) predicate symbols. Then, Π_a and Π_b are *input equivalent with respect to* I, denoted $\Pi_a \equiv_I \Pi_b$, iff for any set of facts Π_f constructed over atoms with a predicate symbol from I, the programs $\Pi_a \cup \Pi_f$ and $\Pi_b \cup \Pi_f$ are (ordinarily) equivalent. $\diamondsuit$

As already noted, input equivalence is originally called (*non-ground*) *relativised uniform equivalence* [49, 70]. For $I = \emptyset$, input equivalence $\equiv_I$ is equal to ordinary equivalence. If we define I as the set of all predicate symbols of our language, then $\equiv_I$ is equal to uniform equivalence. This shows that any form of input equivalence is at least as strong as ordinary equivalence and at most as strong as uniform equivalence.

The following programs $\Pi_a = \{a \leftarrow b\}$ and $\Pi_b = \{a \leftarrow c\}$ demonstrate that there exist sets of input predicates I such that $\equiv_I$ is strictly stronger than $\equiv_o$, and $\equiv_u$ is strictly stronger than $\equiv_I$. It easily seen that Π_a and Π_b are ordinarily equivalent – they both have as only answer set the empty set.

Now, let $I = \{a\}$ and $\Pi_f = \{a \leftarrow\}$. Then, $\Pi_a \cup \Pi_f$ and $\Pi_b \cup \Pi_f$ both have the answer set $\{a\}$, hence $\Pi_a \equiv_I \Pi_b$. But, for the set of facts $\{c \leftarrow\}$ we get that $\Pi_a \cup \{c \leftarrow\}$ has as its only answer set the empty set, while $\Pi_a \cup \{c \leftarrow\}$ has the answer set $\{a, c\}$ and thus Π_a and Π_b are not uniformly equivalent.

4.3 Extended Propositional Transformations

We already mentioned that Brass and Dix [5] proved the following lemma for programs without strong negation:

Lemma 4.3.1 ([5]). *GPPE preserves ordinary equivalence under the answer-set semantics.*

It remains to define a version of GPPE that preserves input equivalence. We therefore introduce GPPE$_I$, which is basically GPPE with the additional requirement that $A \notin I$ for some set of (input) atoms I.

Definition 4.3.2 (Propositional GPPE$_I$). Let Π be a propositional logic program, I a set of atoms and r a rule of Π having the form

$$head(r) \leftarrow body^+(r) \cup \text{not } body^-(r),$$

with $A \in body^+(r)$ and $A \notin I$. Let furthermore G_A the set of all rules containing A in the head. The transformation GPPE$_I$ is then defined as $\Pi' = \Pi \setminus \{r\} \cup G'_A$ where

$$\begin{aligned} G'_A = \{&head(r) \cup (head(r') \setminus \{A\}) \leftarrow \\ &(body^+(r) \setminus \{A\}) \cup \text{not } (body^-(r) \cup body(r')) \mid r' \in G_A\}. \end{aligned}$$

$\diamondsuit$

Note that GPPE$_I$ is a generalisation of GPPE, since GPPE$_\emptyset$ is GPPE. The following lemma is also a generalisation of Lemma 4.3.1, since input equivalence with respect to the empty set is ordinary equivalence.

Lemma 4.3.3. *Let Π be a (propositional) logic program and Π' obtained from Π by one application of GPPE$_I$. Then, Π and Π' are input equivalent with respect to I.*

Proof. Let Π be a propositional logic program, Π_f an arbitrary set of facts over I, and Π' obtained from Π by one application of GPPE$_I$ to some rule $r \in \Pi$ and its body atom $A \notin I$. Let furthermore Π'' be obtained from $\Pi \cup \Pi_f$ by applying GPPE$_I$ to r and A. From $A \notin I$ it follows that for both Π and $\Pi \cup \Pi_f$, the sets G'_A are the same and thus we have

$$\begin{aligned} \Pi' \cup \Pi_f &= \Pi \setminus \{r\} \cup G'_A \cup \Pi_f \\ \Pi'' &= (\Pi \cup \Pi_f) \setminus \{r\} \cup G'_A. \end{aligned}$$

Furthermore, by Lemma 4.3.1, Π'' is equivalent to $\Pi \cup \Pi_f$. It remains to show that $\Pi'' = \Pi' \cup \Pi_f$.

Since set difference is right distributive over union we get $(\Pi \cup \Pi_f) \setminus \{r\} = (\Pi \setminus \{r\}) \cup (\Pi_f \setminus \{r\})$ and thus

$$\Pi'' = (\Pi \setminus \{r\}) \cup (\Pi_f \setminus \{r\}) \cup G'_A.$$

From $A \in body^+(r)$ it follows that r is not a fact, hence $r \notin \Pi_f$ and thus $\Pi_f \setminus \{r\} = \Pi_f$, but then

$$\Pi'' = \Pi \setminus \{r\} \cup \Pi_f \cup G'_A = \Pi' \cup \Pi_f$$

and since Π'' is equivalent to $\Pi \cup \Pi_f$, we get that $\Pi \cup \Pi_f$ and $\Pi' \cup \Pi_f$ are equivalent. $\square$

4.4 Extended Non-ground Transformations

In the following, our goal is to define a non-ground version of GPPE$_I$ which preserves input equivalence. Consider therefore as an example the program Π_{23} which consists of the following rules:

$$a(X) \leftarrow b(X), \tag{4.53}$$
$$b(Y) \leftarrow c(Y). \tag{4.54}$$

From this we want to obtain the program Π'_{23}, consisting of the following rules:

$$a(X) \leftarrow c(X), \tag{4.55}$$
$$b(Y) \leftarrow c(Y). \tag{4.56}$$

Π_{23} is equivalent to Π'_{23} and for every set of facts Π_f which does not contain the predicate b we even have that $\Pi_{23} \cup \Pi_f$ and $\Pi'_{23} \cup \Pi_f$ are equivalent. Before we can define the non-ground version of GPPE$_I$ we will need some preliminary definitions.

When transforming Π_{23} to Π'_{23} we substituted the variable Y in Rule (4.54) to X in order to add $c(X)$ to the body of (4.55). Such substitutions will be necessary in the non-ground case and we will thus first give formal definitions of (most general) unification and permutations which are (sometimes slight modifications of the definitions) from Leitsch [36].

Definition 4.4.1 (Permutation). A substitution λ is called a *permutation* if λ is one-one and $rg(\lambda) \subseteq V$. ♢

Definition 4.4.2 (Variant). A rule r is called a *variant* of a rule r' if there exists a permutation η such that $r\eta = r'$. ♢

Definition 4.4.3 (Unifier). Let C be a non-empty set of expressions. A substitution σ is called *unifier* of C if $|C\sigma| = 1$. Furthermore, C is called *unifiable* iff there exists a unifier of C. ♢

Definition 4.4.4 (Most General Unifier). Let C be a non-empty set of expressions. Let μ, σ be unifiers of C, then μ is *more general* than σ (denoted by $\mu \leq_s \sigma$) if there exists a substitution λ such that $\mu\lambda = \sigma$. A unifier μ of C is called *most general unifier* (*mgu*) of C if for any unifier λ of C it holds that $\mu \leq_s \lambda$. ♢

It can be shown that any two most general unifiers σ_1 and σ_2 of a set C are equivalent modulo permutation, i.e., there exists a permutation λ such that $\sigma_1\lambda = \sigma_2$ and $\sigma_2\lambda^{-1} = \sigma_1$ [41].

In the following, when we say that some expressions $E_1, \ldots, E_n$ are unifiable, we mean that the set $\{E_1, \ldots, E_n\}$ is unifiable. It is also important to note that if a set of expressions is unifiable by some unifier σ, then it is also unifiable by a most general unifier μ [36]. The following example illustrates why we need most general unification.

Example 4.4.5. Consider the program Π_{24}, consisting of the following rules:

$$a \leftarrow b(X), \tag{4.57}$$
$$b(Y) \leftarrow c(Y), \tag{4.58}$$
$$c(1) \leftarrow, \tag{4.59}$$
$$c(2) \leftarrow . \tag{4.60}$$

The body atom $b(x)$ of (4.57) and the head atom $b(Y)$ of (4.58) are easily seen to be unifiable. If we think of applying GPPE$_I$ to Π_{24} we would expect to obtain a program which consists of the following rules:

$$a \leftarrow c(X), \tag{4.61}$$
$$b(Y) \leftarrow c(Y), \tag{4.62}$$
$$c(1) \leftarrow, \tag{4.63}$$
$$c(2) \leftarrow . \tag{4.64}$$

This is obtained by using the unifier $\mu = \{Y \mapsto X\}$ on the Rules (4.57) and (4.58) to get that the head $b(Y)\mu$ of (4.58) equals the body $b(X)$ of (4.57). We can then replace $b(X)$ in the body of (4.57) by $C(X)$ $(= C(Y)\mu)$ to obtain Rule (4.65). We could have also used the unifier $\rho = \{Y \mapsto 1, X \mapsto 1\}$ but then the resulting program, consisting of the following rules:

$$a \leftarrow c(1), \tag{4.65}$$
$$b(Y) \leftarrow c(Y), \tag{4.66}$$
$$c(1) \leftarrow, \tag{4.67}$$
$$c(2) \leftarrow, \tag{4.68}$$

would not meet our expectations since we would expect the rule $a \leftarrow c(2)$ to be contained in the grounding of the transformed program which is not the case here. $\diamondsuit$

Note that in the above example, $\mu\{X \mapsto 1\} = \rho$ and thus μ is more general than ρ according to Definition 4.4.4. Since we want to avoid unifiers like ρ when unifying a head atom of one rule with a body atom of another rule, we will need the most general unifiers.

We will now give a first naive definition of the non-ground version of GPPE$_I$. This version is only sound for a restricted class of programs. It helps to demonstrate the issues that need to be considered when trying to develop a sound version of non-ground GPPE$_I$ whose application is not restricted to a strict subclass of disjunctive logic programs. We assume without loss of generality, that for any two rules $r, r' \in G$, $V(r) \cap V(r') = \emptyset$.

Definition 4.4.6 (GPPE$_I$ (Naive Non-ground Version)). Let Π be a logic program, I a set of predicate symbols and r a rule of Π having the form

$$head(r) \leftarrow body^+(r) \cup \text{not } body^-(r),$$

where $A \in body^+(r)$ and A does not have a predicate symbol from I. Let furthermore G_A be the set of all rules containing a head atom B such that A and B are unifiable.

The transformation GPPE$_I$ is then defined as $\Pi' = \Pi \setminus \{r\} \cup G'_A$ where G'_A is obtained from G_A as follows:

For any rule $r' \in G_A$ and any head atom B of r' which is unifiable with A: Let σ be an mgu of A and B and add to G'_A the rule $r''\sigma$ where

$$r'' = \mathit{head}(r) \cup (\mathit{head}(r') \setminus \{B\}) \leftarrow (\mathit{body}^+(r) \setminus \{A\}) \cup \mathit{body}^+(r') \cup \\ \text{not } (\mathit{body}^-(r) \cup \mathit{body}^-(r')).$$

$\diamondsuit$

The assumption that no two distinct rules share variables is necessary to avoid cases like the following. Consider program Π_{25} which consists of the following rules:

$$a(X) \leftarrow b(X), c(Y), \tag{4.69}$$
$$b(X) \leftarrow d(X, Y). \tag{4.70}$$

Assume that we would apply GPPE$_I$ to atom $b(X)$ in the body of Rule (4.69) with mgu $\mu = \emptyset$. We would get the program Π'_{25}, consisting of the following rules:

$$a(X) \leftarrow d(X, Y), c(Y), \tag{4.71}$$
$$b(X) \leftarrow d(X, Y). \tag{4.72}$$

But this is not what we want since a "sound" transformation should lead to the program Π''_{25} whose rules are as follows:

$$a(X) \leftarrow d(X, Y), c(Z), \tag{4.73}$$
$$b(X) \leftarrow d(X, Y). \tag{4.74}$$

The problem occurs because the Rules (4.69) and (4.70) both contain the variable Y. The assumption that no two distinct rules share variables can easily be justified, since we could – without changing the meaning of the program – rename Y in (4.70) to some new variable not occurring in (4.69) before applying the transformation.

Unifiable Head Atoms

The main reason why GPPE$_I$ cannot be naively applied to non-ground programs is, that some of the rules may contain unifiable head atoms, as illustrated by the following example.

Example 4.4.7. Consider program Π_{26}, consisting of the following rules:

$$a \leftarrow b(V), \tag{4.75}$$
$$b(X) \vee b(Y) \leftarrow c(X, Y), \tag{4.76}$$
$$c(1, 2) \leftarrow . \tag{4.77}$$

If we apply the naive version of GPPE$_I$ here, we obtain program Π'_{26} which consists of the following rules:

$$a \vee b(Y) \leftarrow c(V, Y), \tag{4.78}$$
$$a \vee b(X) \leftarrow c(X, V), \tag{4.79}$$
$$b(X) \vee b(Y) \leftarrow c(X, Y), \tag{4.80}$$
$$c(1, 2) \leftarrow . \tag{4.81}$$

Every answer set of Π_{26} must contain a, since the fact $c(1, 2)$ implies that the head of (4.76) must be satisfied, hence $b(1)$ or $b(2)$ must be contained in an answer set. But then the Rule (4.75) requires that a is contained in an answer set. However, in Π'_{26} it is not required that a is contained in an answer set: One can easily check that $\{c(1, 2), b(1), b(2)\}$ is an answer set of Π'_{26}. ♢

In the above example, we would want the grounding of Π'_{26} to contain a rule like $a \leftarrow c(1, 2)$, since by Rule (4.76), any ground instance of $c(X, Y)$ leads to the derivation of an atom which makes the body of (a ground instance of) Rule (4.75) true.

This also shows that it is not sufficient to additionally use an mgu which unifies both $b(X)$ and $b(Y)$ in the head of (4.76) with $b(V)$ in the body of (4.75). In this case we would generate the additional rule $a(V) \leftarrow c(V, V)$, but this would still not imply that an answer set must contain a. As we have seen, the naive version of GPPE$_I$ does not preserve equivalence. However, we can show equivalence under some additional constraints. After this, we will later then provide an improved version of GPPE$_I$ which preserves equivalence for all disjunctive programs.

Theorem 4.4.8. *Let Π be a logic program and I a set of predicates. Let furthermore Π' be obtained from Π by one application of naive GPPE$_I$ on the body atom A and assume the following:*

- *Π does not contain rules for which two ore more distinct head atoms B, C are both unifiable with A, and*
- *the head of r does not contain an atom which is unifiable with A.*

Then, Π and Π' are input equivalent with respect to I, i.e., for any set of facts Π_f, containing only facts with predicate symbols from I, $\Pi \cup \Pi_f$ and $\Pi' \cup \Pi_f$ are (ordinarily) equivalent.

Proof. Let Π be a logic program. We assume without loss of generality that for any two distinct rules $r_i, r_j \in \Pi$ the variables appearing in r_i and r_j are different, i.e. $V(r_i) \cap V(r_j) = \emptyset$.

Now, let Π' obtained from Π by one application of naive GPPE$_I$, i.e., by replacing some rule $r \in \Pi$ of the form

$$head(r) \leftarrow body^+(r) \cup \text{not } body^-(r) \tag{4.82}$$

with $A \in body^+(r)$ (and A not a predicate over a symbol from I) by the set G'_A of Definition 4.4.6.

Our aim is to show that the grounding of Π' can be obtained from the grounding of Π by repeated application of propositional (*ground*) GPPE$_I$. We thus make the following observation: Since every rule $r' \in \Pi$ contains at most one head atom which is unifiable with A and since r does not contain head atoms which are unifiable with A, we have that no ground instance of a rule from G'_A contains a head atom which is unifiable with A.

Thus, one application of propositional GPPE$_I$ on a ground instance $r\lambda$ with $A\lambda \in \mathit{body}^+(r\lambda)$ does not create any new rules which have to be taken into account when applying GPPE$_I$ to another ground instance $r\mu$ with $r\mu \neq r\lambda$ and $A\mu \in \mathit{body}^+(r\mu)$. The repeated application of GPPE$_I$ to all ground instances of r is thus equivalent to the parallel application of GPPE$_I$ in the following sense:

Let $r\lambda$ be a ground instance of r and Π'_g be obtained from Π_g by one application of propositional GPPE$_I$ to some ground instance $r\rho$ of r. Then, the set G'_A (of propositional GPPE$_I$) is the same both when applying GPPE$_I$ to $r\lambda$ in Π_g and in Π'_g. Since r and G'_A are the only rules in which Π and Π' differ, it suffices thus to show the following:

(i) For any ground instance $r\lambda$ of r, the ground instantiation Π'_g of Π' contains the rules $G_{A\lambda}$ corresponding to an application of propositional GPPE$_{I_G}$ on $r\lambda$ with I_G being the set of all possible ground instantiations of predicates from I.

(ii) Any rule $r'\lambda \in \Pi'_g$ which is not contained in the ground instantiation Π_g of Π is obtained from Π_g by an application of propositional GPPE$_{I_G}$ on some ground rule $r\rho \in \Pi_g$.

PROOF OF (i): Let λ an arbitrary ground substitution and $r\lambda$ a ground instance of r. Let furthermore r'_g an arbitrary rule of Π_g which contains $A\lambda$ in its head. Since r'_g is ground, there exists a rule $r' \in \Pi$ and some (possibly empty) substitution μ such that $r'\mu = r'_g$. Furthermore, r' contains a head atom B such that $B\mu = A\lambda$. Now, since r and r' have no variables in common, it follows that A and B are unifiable by $\lambda\mu$.

Since A and B are unifiable it follows by the definition of GPPE$_I$ that Π' contains for some mgu σ of A and B the rule $r''\sigma$, where

$$r'' = \mathit{head}(r) \cup (\mathit{head}(r') \setminus \{B\}) \leftarrow (\mathit{body}^+(r) \setminus \{A\}) \cup \mathit{body}^+(r') \cup \\ \text{not } (\mathit{body}^-(r) \cup \mathit{body}^-(r')).$$

Now, since σ is an mgu it follows that there exists some (ground) substitution ρ such that $\sigma\rho = \lambda\mu$, hence $r''\sigma\rho$ $(= r''\lambda\mu)$ which is obtained by applying propositional GPPE$_{I_G}$ to $r\lambda$ and $r'\mu$ is contained in Π'_g.

PROOF OF (ii): Let r'_g a rule from Π'_g which is not contained in Π_g. Since r'_g is not contained in Π_g there must be some rule $r''\sigma \in \Pi'$ such that $r'_g = r''\sigma\lambda$ for some ground substitution λ. Furthermore, rule $r''\sigma$ is obtained from rules $r, r' \in \Pi$ with $A \in \mathit{body}^+(r)$, $B \in \mathit{head}(r')$ and $A\sigma = B\sigma$ by one application of GPPE$_I$ with σ being an mgu of A and B.

Since λ maps all variables in $r''\sigma$ to ground terms, λ maps all variables in $r\sigma$ and $r'\sigma$, except for those in $A\sigma = B\sigma$ but not in $r''\sigma$, to ground terms. Now let S be the set of variables in $A\sigma = B\sigma$ but not in $r''\sigma$ and define $\mu = \{X \mapsto c \mid X \in S\}$ for some arbitrary constant $c \in HU(\Pi)$. Then, $\lambda\mu$ maps all variables in $r\sigma$ and $r'\sigma$ to ground terms, but then $\sigma\lambda\mu$ maps all variables in r and r' to ground terms with $A\sigma\lambda\mu = B\sigma\lambda\mu$, hence $r'_g = r''\sigma\lambda$ can be obtained from $r\sigma\lambda\mu$ and $r'\sigma\lambda\mu$ by one application of propositional GPPE$_{I_G}$. □

A Sound Definition of Non-ground GPPE$_I$

The previous considerations lead us the following definition of non-ground GPPE$_I$. Again, we assume without loss of generality, that for any two rules $r, r' \in G$, $V(r) \cap V(r') = \emptyset$.

Definition 4.4.9 (Operator $\Gamma_{A,r}$). Let r a rule with $A \in body^+(r)$ and G a set of rules. Then, $\Gamma_{A,r}(G)$ is obtained as follows:

1. Let $\Gamma_{A,r}(G)$ the set of all rules $r''\sigma$ with

$$r'' = head(r) \cup (head(r') \setminus \{B\}) \leftarrow (body^+(r) \setminus \{A\}) \cup body^+(r') \cup \\ \text{not } (body^-(r) \cup body^-(r')),$$

   where r' is a rule from G which contains a set of atoms $B \subseteq head(r')$ which is unifiable with A and σ is an mgu of $\{A\} \cup B$.

2. If $\Gamma_{A,r}(G)$ contains for some rule $r \in G$ the variants $r_1, \ldots, r_n$ of r, with $r_i \neq r$ ($1 \leq i \leq n$), remove from $\Gamma_{A,r}(G)$ all $r_1, \ldots, r_n$.

Finally, we define $\Gamma^{(1)}_{A,r}(G) = \Gamma_{A,r}(G)$ and $\Gamma^{(n+1)}_{A,r}(G) = \Gamma_{A,r}(\Gamma^{(n)}_{A,r}(G))$. ♢

Definition 4.4.10 (GPPE$_I$, Non-ground Version). Let Π be a logic program, I a set of predicate symbols and r a rule of Π having the form

$$head(r) \leftarrow body^+(r) \cup \text{not } body^-(r),$$

where $A \in body^+(r)$ and A does not have a predicate symbol from I. Let furthermore G_A the set of all rules containing a head atom B such that A and B are unifiable. We then define

$$G'_A = \bigcup_{i=1}^{\infty} \Gamma^{(i)}_{a,r}(G_A).$$

Now, the transformed program is defined as $\Pi' = \Pi \setminus \{r\} \cup G'_A$. ♢

Without further restrictions it is not guaranteed that the series G'_A converges, i.e., that G'_A is finite. But, assuming a finite Herbrand universe, even if the series G'_A does not converge, there exists some n such that the grounding of $\bigcup_{i=1}^{n} \Gamma^{(i)}_{a,r}(G_A)$ equals the grounding of $\bigcup_{i=1}^{\infty} \Gamma^{(i)}_{a,r}(G_A)$ in the sense, that for every rule r in the grounding of $\bigcup_{i=1}^{\infty} \Gamma^{(i)}_{a,r}(G_A)$ there is a rule s in the grounding of $\bigcup_{i=1}^{n} \Gamma^{(i)}_{a,r}(G_A)$ such that $head(s) = head(r)$, $body^+(s) = body^+(r)$, and $body^-(s) =$

$body^-(r)$. This is due to the observation, that over a finite Herbrand universe, the Herbrand base is also finite and thus, when we ignore multiple occurrences of atoms, there can only be a finite number of different rules.

But also with an infinite Herbrand universe, we can observe that one criterion for convergence is that r does not contain a head atom which is unifiable with A. This is easily seen since every rule $r''\sigma$, which is computed by $\Gamma_{a,r}(G_A)$, contains less head atoms that are unifiable with A than the rules r and r' from which it was obtained, hence we get Proposition 4.4.11. In the following, for a rule r, we denote by $u_A(r)$ the number of head atoms in r which are unifiable with A.

Proposition 4.4.11. *Let A be an atom and r a rule for which $u_A(r) = 0$. Let furthermore G_A a set of rules r' for which $u_A(r') \neq 0$. Then, for $n = max\{u_A(r') \mid r' \in G_A\}$ it holds that $\bigcup_{i=1}^{n} \Gamma_{a,r}^{(i)}(G_A) = \bigcup_{i=1}^{\infty} \Gamma_{a,r}^{(i)}(G_A)$.*

The following examples should demonstrate the application of non-ground GPPE$_I$.

Example 4.4.12. The following program Π_{27} is actually part of a program from practice which finds a Hamiltonian cycle within a graph. It consists of the following rules:

$$reached(Y) \leftarrow in\_hm(X,Y), \tag{4.83}$$
$$in\_hm(X,Y) \vee out\_hm(X,Y) \leftarrow bound(X), arc(X,Y), \tag{4.84}$$
$$in\_hm(X,Y) \vee out\_hm(X,Y) \leftarrow reached(X), arc(X,Y). \tag{4.85}$$

Here we can apply non-ground GPPE$_I$ to the atom $in\_hm(X,Y)$ in the body of Rule (4.83). This yields the program with the following rules:

$$reached(Y) \vee out\_hm(X,Y) \leftarrow bound(X), arc(X,Y), \tag{4.86}$$
$$reached(Y) \vee out\_hm(X,Y) \leftarrow reached(X), arc(X,Y), \tag{4.87}$$
$$in\_hm(X,Y) \vee out\_hm(X,Y) \leftarrow bound(X), arc(X,Y), \tag{4.88}$$
$$in\_hm(X,Y) \vee out\_hm(X,Y) \leftarrow reached(X), arc(X,Y). \tag{4.89}$$

Here, $reached$ is directly derived from $bound$, in and out without using in_hm. ♢

In the next example we deal with unifiable head atoms:

Example 4.4.13. Consider the program Π_{28}, consisting of the following rules:

$$coloured(X) \leftarrow chosen(X), \tag{4.90}$$
$$chosen(X) \vee chosen(Y) \leftarrow vertex(X), \text{not } edge(X,Y). \tag{4.91}$$

Applying non-ground GPPE$_I$ to $chosen(X)$ in (4.90) yields the program Π'_{28} as follows:

$$coloured(X) \vee chosen(Y) \leftarrow vertex(X), \text{not } edge(X,Y), \tag{4.92}$$
$$coloured(X) \vee chosen(Y) \leftarrow vertex(Y), \text{not } edge(Y,X), \tag{4.93}$$
$$chosen(X) \leftarrow vertex(X), \text{not } edge(X,X), \tag{4.94}$$
$$coloured(X) \vee coloured(Y) \leftarrow vertex(Y), \text{not } edge(Y,X), \tag{4.95}$$
$$coloured(Y) \vee coloured(X) \leftarrow vertex(X), \text{not } edge(X,Y), \tag{4.96}$$
$$chosen(X) \vee chosen(Y) \leftarrow vertex(X), \text{not } edge(X,Y). \tag{4.97}$$

Note that for $A = chosen(X)$, the set G_A (the set of rules containing a head atom which is unifiable with A) contains Rule (4.91), hence $\Gamma^{(1)}_{A,r}(G_A)$ contains the Rules (4.92), (4.93), and (4.94). The Rules (4.95) and (4.96) are then contained in $\Gamma^{(2)}_{A,r}(G_A)$. The transformation would not be sound if the rules from $\Gamma^{(2)}_{A,r}(G_A)$ were not contained in Π'_{28}. ♢

The following example should illustrate that the application of GPPE$_I$ to a rule r and its body atom A can lead to interesting results if $u_A(r) \neq 0$, i.e., if r contains one or more head atoms which are unifiable with A.

Example 4.4.14. Consider program Π_{29} which consists of the following rules:

$$b(Y,X) \leftarrow b(X,Y), \tag{4.98}$$
$$b(X,Y) \vee b(Y,X) \leftarrow c(X,Y), \tag{4.99}$$
$$c(1,2) \leftarrow . \tag{4.100}$$

When we apply non-ground GPPE$_I$ to Rule 4.98 and its body atom $b(X,Y,Z)$, we obtain the program Π'_{29}, consisting of the following rules:

$$b(Y,X) \leftarrow b(Y,X), \tag{4.101}$$
$$b(Y,X) \leftarrow c(X,Y), \tag{4.102}$$
$$b(Y,X) \leftarrow b(X,Y), \tag{4.103}$$
$$b(Y,X) \leftarrow c(Y,X), \tag{4.104}$$
$$b(X,Y) \vee b(Y,X) \leftarrow c(X,Y), \tag{4.105}$$
$$c(1,2) \leftarrow . \tag{4.106}$$

Here, for $A = b(X,Y)$, the set G_A contains the Rules (4.98) and (4.99). The Rules (4.101) and (4.102) are contained in $\Gamma^{(1)}_{A,r}(G_A)$, and (4.103) as well as (4.104) are contained in $\Gamma^{(2)}_{A,r}(G_A)$. ♢

Note that in the above example, the program obtained by the transformation is a superset of the original program. In the case of recursive predicates one should thus be careful when using non-ground GPPE$_I$.

The next proposition shows that on propositional programs, our non-ground definition of GPPE$_I$ agrees with propositional GPPE$_I$. This should support the claim that we introduced an intuitive generalisation to the non-ground case.

Proposition 4.4.15. *Let Π_p and Π_{ng} be obtained from a propositional program Π by one application of propositional GPPE$_I$ and non-ground GPPE$_I$, respectively. Then, $\Pi_p = \Pi_{ng}$.*

Proof. Let Π be a propositional program. The transformations of both propositional and non-ground GPPE$_I$ are defined as $\Pi' = \Pi \setminus \{r\} \cup G'_A$ for $r \in \Pi$, $A \in body^+(r)$ and $A \notin I$ (note that on the propositional case, the set of all atoms with predicates from I is exactly I). In both cases, G'_A is defined relative to the set G_A of all rules containing A in the head. The only difference is, how G'_A is defined in terms of G_A. To make a distinction, we denote the propositional G'_A as G'_P while we denote the non-ground G'_A as G'_N. We will thus show, that $G'_P = G'_N$.

Since $G'_N = \bigcup_{i=1}^{\infty} \Gamma^{(i)}_{a,r}(G_A)$, we will show that $\Gamma^{(1)}_{a,r}(G_A) = G'_P$ and that for all $i \geq 1$ $\Gamma^{(i+1)}_{a,r}(G_A) \subseteq \Gamma^{(i)}_{a,r}(G_A)$. To show that $\Gamma^{(1)}_{a,r}(G_A) = G'_P$, consider the definitions of r'' in $\Gamma^{(i)}_{a,r}(G_A)$ and G'_N and observe first that a set B of propositional atoms is unifiable with an atom A (with the empty unifier) iff $B = \{A\}$.

Hence, removing a set B of atoms which are unifiable with A from $head(r')$ (as it's the case in the definition of r'' for $\Gamma_{a,r}(G_A)$) is the same as removing just $\{A\}$ (as in the propositional version). It follows that the definitions of r'' are the same in both $\Gamma^{(1)}_{a,r}(G_A)$ and G'_P. Furthermore, variants of rules don't play a role in the propositional case and thus $\Gamma^{(1)}_{a,r}(G_A) = G'_N$.

It remains to show that $\Gamma^{(i+1)}_{a,r}(G_A) \subseteq \Gamma^{(i)}_{a,r}(G_A)$ for all $i \geq 1$: All rules in $\Gamma^{(i)}_{a,r}(G_A)$ are of the form

$$r'' = head(r) \cup (head(r') \setminus B) \leftarrow (body^+(r) \setminus \{A\}) \cup body^+(r') \cup \\ \text{not } (body^-(r) \cup body^-(r')).$$

Now, when computing $\Gamma^{(i+1)}_{a,r}(G_A)$, all these rules are resolved with r. But, the head of a newly created rule r''' is then defined as

$$head(r) \cup (head(r) \cup (head(r') \setminus B)) = head(r'').$$

The same argument holds for the body of r'''. Hence, $\Gamma^{(i+1)}_{a,r}(G_A)$ contains only rules which were already contained in $\Gamma^{(i)}_{a,r}(G_A)$. □

Note that the proof shows, that in the propositional case, one application of $\Gamma_{a,r}(G_A)$ is sufficient. Our goal is now to show that non-ground GPPE$_I$ preserves input equivalence with respect to I, as formulated by the result in the following section.

4.5 Input-Equivalence Preservation

Theorem 4.5.1. *Let Π be a logic program and I a set of predicates. Let furthermore Π' be obtained from Π by one application of GPPE$_I$. Then, Π and Π' are input equivalent with respect to I, i.e., for any set of facts Π_f, containing only facts with predicate symbols from I, $\Pi \cup \Pi_f$ and $\Pi' \cup \Pi_f$ are (ordinarily) equivalent.*

The proof is non-trivial and requires some preparatory work. Our strategy is as follows: We take some program Π, ground it and apply a series of equivalence-preserving propositional transformations to obtain a program Π''. We then show, that Π'' equals the grounding of the program Π', which was obtained from Π by one application of non-ground GPPE$_I$.

An important issue, which has to be dealt with when lifting the propositional version of GPPE$_I$ to the non-ground case, is illustrated by the following example.

Example 4.5.2. Consider the program Π_{30} which consists of the following rules:

$$a(X) \leftarrow b(X), b(Y), \tag{4.107}$$
$$b(1) \leftarrow d(2). \tag{4.108}$$

When applying GPPE$_I$ to the body atom $b(X)$ in Rule (4.107), we obtain the program Π'_{30}, consisting of the following rules:

$$a(1) \leftarrow d(2), b(Y), \tag{4.109}$$
$$b(1) \leftarrow d(2). \tag{4.110}$$

When grounding Π'_{30}, we obtain the following rules:

$$a(1) \leftarrow d(2), b(1), \tag{4.111}$$
$$a(1) \leftarrow d(2), b(2), \tag{4.112}$$
$$b(1) \leftarrow d(2). \tag{4.113}$$

Consider in contrast the grounding of Π_{30}:

$$a(1) \leftarrow b(1), b(1), \tag{4.114}$$
$$a(1) \leftarrow b(1), b(2), \tag{4.115}$$
$$a(2) \leftarrow b(2), b(1), \tag{4.116}$$
$$a(2) \leftarrow b(2), b(2), \tag{4.117}$$
$$b(1) \leftarrow d(2). \tag{4.118}$$

When we apply propositional GPPE$_I$ to the ground instances of $b(X)$ from Π_{30} in Rule (4.107) (i.e., $b(1)$ in (4.114) and (4.115), and $b(2)$ in (4.116) and (4.117)) we end up with the program Π''_{30}, consisting of the following rules:

$$a(1) \leftarrow d(2), \tag{4.119}$$
$$a(1) \leftarrow d(2), b(2), \tag{4.120}$$
$$b(1) \leftarrow d(2). \tag{4.121}$$

Π''_{30} differs from the grounding of Π'_{30} in the Rule (4.119). This is because in Rule (4.107) in Π_{30}, the atoms $b(X)$ and $b(Y)$ are unified by the ground substitution $\{X \mapsto 1, Y \mapsto 1\}$, hence Rule (4.114) contains two occurrences of $b(1)$. When applying (ground or non-ground) GPPE$_I$, we add the Rule (4.119) which is defined as

$$\begin{aligned} head(r) \cup (head(r') \setminus \{A\}) \leftarrow (body^+(r) \setminus \{A\}) \cup \\ \text{not } (body^-(r) \cup body(r'), \end{aligned}$$

where r is Rule (4.114) and r' is Rule (4.118). $\diamondsuit$

The crucial thing in Example 4.5.2 is that we have $body^+(r) \setminus \{A\}$ in the body of r''. Hence, in the propositional (ground) case, *every* occurrence of $A = b(1)$ is removed from $body^+(r)$ while in the non-ground case, only the atom $A = b(X)$ is removed from the positive body of r, but not the atom $b(Y)$.

The problem occurs always then, when in the course of grounding some non-ground rule r, two distinct atoms in r unify to the same ground atom. Consider again the rule $r = a(1) \leftarrow b(X), b(Y)$ from the above example. Clearly, $a(1) \leftarrow b(1), b(1)$ is a ground instance of r. Under the answer-set semantics, r is equivalent to $r' = a(1) \leftarrow b(1)$, and both r and r' have the same set of body atoms. But when applying syntactic transformations, differentiating between r and r' can be important, as we will see later. To avoid this, we will now change our notion from ordinary sets to multisets and keep with this notion until the end of the proof of Theorem 4.5.1. Based on multisets, we will then define according propositional (ground) transformations which will be used in the proof of Theorem 4.5.1.

We thus shortly recall the notion of multisets which, in contrast to ordinary sets, allow multiple occurrences of one and the same element. Formally, a multiset over a set S is a tuple (S, f) where f is a function $f : S \to \mathbb{N}$. Let $M = (S, f)$ be a multiset over $S = \{a, b, c\}$, with $f(a) = 2, f(b) = 1, f(c) = 0$. For such a set, we use the notion $M = \{a, a, b\}$. We say that $a \in S$ iff $f(a) > 0$. If $f(a) = 0$ for all $a \in S$, then we write $M = \emptyset$. An ordinary set A is actually a multiset (A, χ_A), where χ_A is its characteristic function [63].

The support $supp(M)$ of a multiset $M = (A, f)$ is defined as the set of all elements $a \in A$ for which $f(a) > 0$. Assume that $M = (A, f)$ and $N = (A, g)$ are two multisets. We say that M is a sub-multiset of N, denoted $M \sqsubseteq N$ if for all $a \in A$ we have $f(a) \leq g(a)$. We use the $\sqsubseteq$-symbol instead of the more common $\subseteq$-symbol here. This is because we will mainly use the following subset relation.

Definition 4.5.3. Assume that $M = (A, f)$ and $N = (A, g)$ are two multisets. We say that M is a subset of N, denoted $M \subseteq N$ iff $supp(M) \subseteq supp(N)$. $\diamondsuit$

Suppose that $M = (A, f)$ and $N = (A, g)$ are two multisets. Their union, denoted $M \cup N$, is the multiset $Q = (A, h)$ where $h(a) = max(f(a), g(a))$ for all $a \in A$. Likewise, for the intersection $M \cap N$, $h(a) = min(f(a), g(a))$ for all $a \in A$.

Example 4.5.4. Let $M = \{a, a, b, c\}$ and $N = \{a, b, b\}$ two multisets. Then, $N \subseteq M$, but not $N \sqsubseteq M$ or $M \subseteq N$. Furthermore, $M \cup N = \{a, a, b, b, c\}$ and $M \cap N = \{a, b\}$. $\diamondsuit$

The following two notions of set difference will be particularly important to us.

Definition 4.5.5. Assume that $M = (A, f)$ and $N = (A, g)$ are two multisets. Then, $M \setminus N = (A, h)$ with

$$h(a) = \begin{cases} 0, & \text{if } g(a) > 0; \\ f(a), & \text{otherwise.} \end{cases}$$

$\diamondsuit$

Definition 4.5.6. Assume that $M = (A, f)$ and $N = (A, g)$ are two multisets. Then, $M - N = (A, h)$ with

$$h(a) = \begin{cases} 0, & \text{if } g(a) > f(a); \\ f(a) - g(a), & \text{otherwise.} \end{cases}$$

$\diamondsuit$

Example 4.5.7. Let $M = \{a, a, b\}$ and $N = \{a\}$, then $M \setminus N = \{b\}$ and $M - N = \{a, b\}$. $\diamondsuit$

Now, let r be a rule, then r is identified by the multisets $head(r)$, $body^+(r)$, and $body^-(r)$. Consider for example the rule

$$a \vee a \vee b \leftarrow c, d, d, \text{not } e, \text{not } c, \text{not } e.$$

Then, $head(r) = \{a, a, b\}$, $body^+(r) = \{c, d, d\}$, and $body^-(r) = \{e, e, c\}$. One can easily check that the equivalence results of Table 4.1 also hold with the multiset notion instead of ordinary sets. This is due to the fact that we use the $\subseteq$-symbol for the subset-relation of Definition 4.5.3 and not, as it is common, for the multisubset-relation.

We can now introduce the propositional transformations necessary for the proof of Theorem 4.5.1. The transformations differ from GPPE$_I$ and WGPPE by using $body^+(r) - \{A\}$ instead of $body^+(r) \setminus \{A\}$ in the body of the newly added rules. This way, only one occurrence of A in $body^+(r)$ is removed from $body^+(r)$ and not all of them.

Definition 4.5.8 (Extended WGPPE (EWGPPE)/Extended GPPE$_I$ (EGPPE$_I$)). Let Π be a propositional logic program, I a set of atoms, and r a rule of Π having the form

$$head(r) \leftarrow body^+(r) \cup \text{not } body^-(r),$$

with $A \in body^+(r)$. Let furthermore G_A the set of all rules containing A in the head and

$$G'_A = \{head(r) \cup (head(r') \setminus \{A\}) \leftarrow (body^+(r) - \{A\}) \cup \text{not } (body^-(r) \cup body(r')) \mid r' \in G_A\}.$$

Then, the transformation EWGPPE is defined as $\Pi' = \Pi \cup G'_A$. The transformation EGPPE$_I$ is defined as $\Pi' = \Pi \setminus \{r\} \cup G'_A$ with the precondition $A \notin I$. $\diamondsuit$

Example 4.5.9. Consider program Π_{31} which consists of the following rules:

$$a \leftarrow b, b, \tag{4.122}$$
$$b \leftarrow c. \tag{4.123}$$

Applying EGPPE$_I$ to Rule (4.122) and one of its body atoms b yields Π'_{31}, consisting of the following rules:

$$a \leftarrow b, c, \tag{4.124}$$
$$b \leftarrow c. \tag{4.125}$$

This is different from GPPE$_I$ where we would get Π''_{31} whose rules are as follows:

$$a \leftarrow c, \tag{4.126}$$
$$b \leftarrow c. \tag{4.127}$$

Here, b does not occur in the body of Rule (4.126) anymore. ♢

We will in the following develop equivalence-preservation results for EGPPE$_I$ and EWGPPE. Therefore we define the term *G-implication* (from GPPE-implication).

Definition 4.5.10 (G-implication). Let Π be a propositional program, r some arbitrary rule (not necessarily contained in Π) and $r', r'' \in \Pi$, where $A \in head(r')$ and r'' is of the form

$$head(r) \cup (head(r') \setminus \{A\}) \leftarrow body^+(r) \cup body^+(r') \cup \text{not } (body^-(r) \cup body^-(r')).$$

Then, we say that the rule r_G of the form

$$head(r) \cup (head(r') \setminus \{A\}) \leftarrow (body^+(r) \setminus \{A\}) \cup body^+(r') \cup \text{not } (body^-(r) \cup body^-(r'))$$

is *G-implied* by r'' in Π. ♢

Note that, if A is not contained in $body^+(r)$, $r'' = r_G$. Due to this, we do not require A to be contained in $body^+(r)$.

Lemma 4.5.11. *Let Π be a propositional logic program containing a rule r_G which is G-implied by some other rule $r'' \in \Pi$ and let $\Pi' = \Pi \setminus \{r_G\}$. Then, Π and Π' are strongly equivalent.*

Before we prove this lemma, we give an example to illustrate G-implication.

Example 4.5.12. Consider program Π_{32}, given as follows:

$$a \leftarrow b, b, \tag{4.128}$$
$$b \leftarrow c. \tag{4.129}$$

By applying EWGPPE to (4.128) in Π_{32} we obtain Π'_{32}:

$$a \leftarrow b, b, \tag{4.130}$$
$$a \leftarrow c, b, \tag{4.131}$$
$$b \leftarrow c. \tag{4.132}$$

Now we apply GPPE$_I$ to (4.130) in Π'_{32} and get Π''_{32} as follows:

$$a \leftarrow c, \tag{4.133}$$
$$a \leftarrow c, b, \tag{4.134}$$
$$b \leftarrow c. \tag{4.135}$$

Let r be the Rule (4.128), r' the Rule (4.135) and r'' the Rule (4.134). Then, the Rule (4.133) which is of the form

$$head(r) \cup (head(r') \setminus \{b\}) \leftarrow (body^+(r) \setminus \{b\}) \cup body^+(r') \cup \text{not } (body^-(r) \cup body^-(r')),$$

is G-implied by (4.134) in Π'_{32}. ♢

The example should demonstrate that rules which are obtained by using GPPE$_I$ are G-implied by rules which are obtained using EGPPE$_I$ (the same holds for WGPPE and EWGPPE). By Lemma 4.5.11, we can remove them, even though they imply the rules obtained via EGPPE$_I$.

In order to prove Lemma 4.5.11, we use the same strategy that Brass and Dix [5] used to prove Lemma 4.3.1. Note first that a *minimal model* of a program Π is a minimal set of ground atoms $\mathcal{M}$ such that for every rule r, whenever the positive body atoms of r are contained in $\mathcal{M}$ and the negative body atoms of r are not contained in $\mathcal{M}$, then a head atom of r is contained in $\mathcal{M}$. It follows that for positive programs without strong negation, the notion of a minimal model coincides with the notion of an answer set. We will first prove the following lemma:

Lemma 4.5.13. *Let Π be a propositional logic program containing a rule r_G which is G-implied by some other rule $r'' \in \Pi$ and let $\Pi' = \Pi \setminus \{r_G\}$. Then, for any program Π_x, $\Pi \cup \Pi_x$ and $\Pi' \cup \Pi_x$ have the same minimal models.*

Proof. We first show that if $\mathcal{M}$ is a minimal model of $\Pi \cup \Pi_x$ ($\Pi' \cup \Pi_x$) then it is a model of $\Pi' \cup \Pi_x$ ($\Pi \cup \Pi_x$, respectively).

Suppose that $\mathcal{M}$ is a minimal model of $\Pi \cup \Pi_x$ but not a model of $\Pi' \cup \Pi_x$. It follows that there exists a rule $s \in \Pi' \cup \Pi_x$ which is not satisfied by $\mathcal{M}$, but since $\Pi' \cup \Pi_x \subseteq \Pi \cup \Pi_x$ we get that $s \in \Pi \cup \Pi_x$ and thus $\mathcal{M}$ is not a model of $\Pi \cup \Pi_x$, a contradiction.

Now, assume that $\mathcal{M}$ is a minimal model of $\Pi' \cup \Pi_x$ but not a model of $\Pi \cup \Pi_x$. Since r_G is the only rule which is contained in Π but not in Π' it follows that r_G is not satisfied by $\mathcal{M}$ and thus $body(r_G)$ is satisfied by $\mathcal{M}$ but $head(r_G)$ (= $head(r'')$) is not. Since $\mathcal{M}$ satisfies the body of r_G, it satisfies the body of r' and since it also satisfies r' but not $head(r') \setminus \{A\}$ it follows that $\mathcal{M}$ satisfies A. But then, $\mathcal{M}$ satisfies the body of r'' but not its head and thus $\mathcal{M}$ is not a model of $\Pi' \cup \Pi_x$, a contradiction.

It remains to show that a minimal model $\mathcal{M}$ of $\Pi \cup \Pi_x$ ($\Pi' \cup \Pi_x$, respectively) is also minimal of $\Pi' \cup \Pi_x$ ($\Pi \cup \Pi_x$, respectively). Let $\mathcal{M}$ be a minimal model of $\Pi \cup \Pi_x$, then $\mathcal{M}$ is a model of $\Pi' \cup \Pi_x$. Now assume that $\mathcal{M}$ is not a minimal model of $\Pi' \cup \Pi_x$, i.e., there exists some model $\mathcal{M}' \subset \mathcal{M}$ which is a minimal model of $\Pi' \cup \Pi_x$. It follows that $\mathcal{M}'$ is a model of $\Pi \cup \Pi_x$ and thus $\mathcal{M}$ is not a minimal model of $\Pi \cup \Pi_x$, a contradiction. The argument for the other direction is analogous. □

We continue with the proof of Lemma 4.5.11.

Proof. Let Π_x be an arbitrary logic program. We have to show that $\mathcal{M}$ is an answer set of $\Pi \cup \Pi_x$ if and only if $\mathcal{M}$ is an answer set of $\Pi' \cup \Pi_x$. We proceed by a case distinction.

1. $(body^-(r) \cup body^-(r')) \cap \mathcal{M} \neq \emptyset$: In this case (the reduced) r_G is not contained in the reduct of $\Pi \cup \Pi_x$ and thus the reducts of $\Pi \cup \Pi_x$ and $\Pi' \cup \Pi_x$ are the same. It follows that $\mathcal{M}$ is a minimal model of the reduct (and thus an answer set) of $\Pi \cup \Pi_x$ if and only if it is a minimal model of the reduct of $\Pi' \cup \Pi_x$.
2. $(body^-(r) \cup body^-(r')) \cap \mathcal{M} = \emptyset$: Let $r_R = head(r) \leftarrow body^+(r)$. The reducts of

$\Pi \cup \Pi_x$ and $\Pi' \cup \Pi_x$ both contain the reduced rules r'_R and r''_R of r' and r'', respectively, with

$$r'_R = head(r') \leftarrow body^+(r'), \text{ and}$$
$$r''_R = head(r) \cup (head(r') \setminus \{A\}) \leftarrow body^+(r) \cup body^+(r').$$

The only rule in which the two reducts can differ is the rule

$$\begin{aligned} r_{G_R} &= head(r) \cup (head(r') \setminus \{A\}) \leftarrow (body^+(r) \setminus \{A\}) \cup body^+(r') \\ &= head(r_R) \cup (head(r'_R) \setminus \{A\}) \leftarrow (body^+(r_R) \setminus \{A\}) \cup body^+(r'_R) \end{aligned}$$

for which it may be the case that it is only contained in the reduct of $\Pi \cup \Pi_x$ but not in the reduct of $\Pi' \cup \Pi_x$. But then, we have an arbitrary rule r_R, two rules r'_R, r''_R which are contained in the reduct of $\Pi \cup \Pi_x$ with $A \in head(r'_R)$ and r''_R is of the form

$$head(r_R) \cup (head(r'_R) \setminus \{A\}) \leftarrow body^+(r_R) \cup body^+(r'_R).$$

Hence, r_{G_R} is G-implied by r''_R in the reduct of $\Pi \cup \Pi_x$. By Lemma 4.5.13 it follows that $\mathcal{M}$ is a minimal model of the reduct (and thus an answer set) of $\Pi \cup \Pi_x$ if and only if it is a minimal model of the reduct of $\Pi' \cup \Pi_x$.

□

Proposition 4.5.14. *Let Π be a propositional logic program and Π' obtained from Π by one application of EGPPE$_I$. Then, Π and Π' are input equivalent with respect to I.*

Proof. Let Π be a propositional logic program and Π_G obtained from Π by one application of GPPE$_I$ on a rule r with body atom A, i.e. $\Pi_G = \Pi \setminus \{r\} \cup G'_A$. By Lemma 4.3.3 it follows that for any set of facts $\Pi_f \in \mathcal{C}_{\mathcal{U}\setminus I}$, $\Pi \cup \Pi_f$ and $\Pi_G \cup \Pi_f$ are equivalent.

If $body(r)$ contains only one occurrence of A, then $\Pi_G = \Pi'$ and thus the statement holds. Suppose now that A occurs more than once in $body(r)$. The set G'_A contains for every rule r' with $A \in head(r')$ a rule r'' of the form

$$head(r) \cup (head(r') \setminus \{A\}) \leftarrow (body^+(r) \setminus \{A\}) \cup body^+(r') \cup \text{not } (body^-(r) \cup body(r')).$$

Now let Π_I obtained from Π_G by adding for every rule $r'' \in G'_A$ a rule s'' of the form

$$head(r) \cup (head(r') \setminus \{A\}) \leftarrow (body^+(r) - \{A\}) \cup body^+(r') \cup \text{not } (body^-(r) \cup body(r')).$$

Since for every rule s'' there is a rule $r'' \in G'_A$ such that s'' and r'' only differ in $body^+(r) - \{A\}$ and $body^+(r) \setminus \{A\}$ and $body^+(r) \setminus \{A\} \subseteq body^+(r) - \{A\}$, we get that every s'' is implied by some r'', hence Π_G and Π_I are strongly equivalent.

Finally, let Π_R obtained from Π_I by removing every rule $r'' \in G'_A$. Let $s = head(r) \leftarrow (body^+(r) - \{A\}) \cup \text{not } body^-(r)$. We then have an arbitrary rule s, and for every rule $r' \in \Pi$ with $A \in head(r')$ a rule r'' of the form

$$head(r) \cup (head(r') \setminus \{A\}) \leftarrow (body^+(r) \setminus \{A\}) \cup body^+(r') \cup \text{not } (body^-(r) \cup body(r')),$$

which is identical to

$$head(s) \cup (head(r') \setminus \{A\}) \leftarrow (body^+(s) \setminus \{A\}) \cup body^+(r') \cup \text{not } (body^-(s) \cup body(r')).$$

But then, every r'' is G-implied by the corresponding s'' of the form

$$head(s) \cup (head(r') \setminus \{A\}) \leftarrow body^+(s) \cup body^+(r') \cup \text{not } (body^-(s) \cup body(r')).$$

Hence, removing every r'' preserves strong equivalence. But now $\Pi_R = \Pi'$ and thus, for every set of facts $\Pi_f \in \mathcal{C}_{\mathcal{U}\setminus I}$, $\Pi \cup \Pi_f$ and $\Pi' \cup \Pi_f$ are equivalent. □

Proposition 4.5.15. *Let Π be a propositional logic program and Π' obtained from Π by one application of EWGPPE. Then, Π and Π' are strongly equivalent.*

Proof. The proof is analogous to the one of Proposition 4.5.14. The only difference is that we obtain Π_G from Π by an application of WGPPE and not GPPE$_I$. Since WGPPE preserves strong equivalence, Π and Π' are strongly equivalent. □

We will now introduce the notion of *G-depth* which we will use for our induction argument in the proof of Theorem 4.5.1. Recall that the transformations of propositional and non-ground GPPE$_I$ are defined as $\Pi' = \Pi \setminus \{r\} \cup G'_A$ for some G'_A. Furthermore, for WGPPE the transformation is defined as $\Pi' = \Pi \cup G'_A$.

Definition 4.5.16 (G-depth). Let Π' obtained from Π by a series of applications of (either propositional or non-ground) GPPE$_I$ or WGPPE. Then, if a rule r' is contained in Π and it was not contained in any of the sets G'_A which are defined by the respective transformations, it has G-depth 0. If a rule r'' is contained in Π' and it was obtained by resolving a rule $r \in \Pi$ (with a body atom A) with a rule r', having G-depth n, r'' has G-depth $n + 1$. ♢

Note that this assigns a G-depth to every rule $r \in \Pi'$. This is, because every rule $r \in \Pi'$ was either already contained in Π, and if not, it was obtained by an application of GPPE$_I$, WGPPE, or non-ground GPPE$_I$.

Example 4.5.17. Consider the program Π_{33} which consists of the following rules:

$$a(X) \leftarrow b(X), \tag{4.136}$$
$$b(X) \vee b(Y) \leftarrow c(X, Y). \tag{4.137}$$

If we apply GPPE$_I$ to Rule (4.136), we obtain the program Π'_{33}, consisting of the following rules:

$$a(X) \vee b(Y) \leftarrow c(X, Y), \tag{4.138}$$
$$a(X) \vee b(X) \leftarrow c(X, X), \tag{4.139}$$
$$a(X) \leftarrow c(X, X), \tag{4.140}$$
$$a(X) \vee a(Y) \leftarrow c(Y, X), \tag{4.141}$$
$$b(X) \vee b(Y) \leftarrow c(X, Y). \tag{4.142}$$

The first three rules are obtained by resolving Rule (4.137) with (4.137), hence they have G-depth 1. Rule (4.141) was obtained by resolving the new Rule (4.138) with (4.136), hence it has G-depth 2. Finally, the last rule was already obtained in Π and thus it has G-depth 0. $\diamondsuit$

Note that the G-depth of a rule r'' may not be unique, since it may be the case that r'' was obtained in various different ways, e.g., by resolving a rule of G-depth i with a rule $r \in \Pi$ but also by resolving a rule of G-depth $j \neq i$ with a rule $r \in \Pi$. For the proof of Theorem 4.5.1 we need to assign to each rule a unique G-depth, therefore we will use the minimal G-depth of a rule. With the minimal G-depth it still holds that a rule r'' of minimal G-depth n was obtained by resolving a rule r with some rule r' of minimal G-depth $< n$.
We are now finally in the position to give a proof of Theorem 4.5.1, which states that non-ground GPPE$_I$ preserves input equivalence with respect to I. In the proof, when saying that Π and Π' are equivalent, we actually mean that Π and Π' are input equivalent with respect to I.

Proof of Theorem 4.5.1

Proof. Let Π be a logic program and Π' obtained from Π by an application of GPPE$_I$ to a rule r and its body atom A. For the rest of the proof, we assume that A does not occur more than once in the positive body of r. We will show that Π and Π' are equivalent by proving the following statement:

There exists a program Π'' which can be obtained from the grounding $grd(\Pi)$ of Π by a series of applications of propositional EWGPPE and EGPPE$_I$, such that the following holds

(i) Every rule s which is contained in Π'' is also contained in $grd(\Pi')$.

(ii) For every rule $s \in grd(\Pi')$, Π'' contains a rule which implies s.

Since Π'' was obtained from $grd(\Pi)$ by successively applying EWGPPE and EGPPE$_I$ it is equivalent to $grd(\Pi)$. Furthermore, we can add to Π'' every rule $s \in grd(\Pi')$ which is not contained in Π'', since Π'' contains at least a rule s' which implies s and adding a rule which is implied by some other rule preserves (even strong) equivalence. Hence, we can obtain $grd(\Pi')$ from $grd(\Pi)$ by a series of equivalence preserving transformations which proves that $grd(\Pi)$ and $grd(\Pi')$ are equivalent.

Intuitively, we want Π'' to be obtained from $grd(\Pi)$ by successively applying propositional EWGPPE and EGPPE$_I$ to all ground instances of r until no new rules are created. So let Π'' be computed by Algorithm 2.

If the Herbrand universe is finite, Algorithm 2 will always terminate. An argument for this goes as follows: Since we only have a finite number of predicate symbols of finite arity and the Herbrand universe is finite, the Herbrand base is also finite. Furthermore, the rules in G'_A are defined by existing rules via union and difference of multisets defining existing rules ($head(r)$,$body^+(r)$, etc.).

1. Let $\Pi''_0 = grd(\Pi)$.

2. Order all ground instances of r and denote them as $r_1, \ldots, r_n$.

3. Let $j = 1$.

4. Let $\Pi''_j = \Pi''_{j-1}$ and $i = 1$.

5. Apply EWGPPE to $r_i \in \Pi''_j$ and its according ground instance of A in $body^+(r_i)$ and denote the resulting program as Π''_j.

6. If $i < n$ set $i = i + 1$ and go to Step 5, else go to the next step.

7. If $\Pi''_j \neq \Pi''_{j-1}$, set $j = j + 1$ and go to Step 4, else go the next step.

8. Apply EGPPE$_I$ to every rule r_i which is not contained in $grd(\Pi')$, and denote the resulting program as Π''. (This does actually nothing else than removing ground instances of r, since no new rules are added. We do this since it helps us to prove the statement.)

Algorithm 2: An algorithm which computes Π''.

Let $\circ \in \{\cup, \setminus, -\}$, $M = (A, f)$ and $N = (A, g)$ multisets and $Q = (A, h) = M \circ N$. Then, for any $a \in A$, we have that $h(a) \leq max(f(a), g(a))$, and thus the number of occurrences of an atom in $head(r)$, $body^+(r)$, or $body^-(r)$ of a newly added rule is not increased by the transformation. Therefore, the number of occurrences of a single atom is also bounded and hence there is only a finite number of different rules which can be created.

Note also that we used EWGPPE first and EGPPE$_I$ only at the end. This is because EWGPPE adds exactly the same rules as EGPPE$_I$ but it does not remove any of the r_i. Furthermore, EWGPPE preserves also equivalence (in fact even strong equivalence). We proceed by proving statement (i).

PROOF OF (i): By induction on the minimal G-depth n of s.

INDUCTION BASE ($n = 0$): Assume that $s \in \Pi''$ but $s \notin grd(\Pi')$. Since s has minimal G-depth 0, it follows that s is contained in $grd(\Pi)$ but was not contained in any of the sets G'_A. Apart from ground instances of r, all rules of $grd(\Pi)$ are contained in $grd(\Pi')$, hence s is a ground instance of r. But then, since in Step 8 in the creation of Π'', EGPPE$_I$ is applied to all ground instances of r which are not contained in $grd(\Pi')$, and EGPPE$_I$ takes a program Π and transforms it to $(\Pi \setminus \{r\}) \cup G'_A$, it follows that s was removed in Step 8 and is thus not in Π'', a contradiction.

INDUCTION STEP ($n < 0$): Assume that the statement holds for all $k \leq n$ with $k > 0$, we will show that it then also holds for $n + 1$. Suppose therefore that s has minimal G-depth $n + 1$. This means that s was obtained by resolving a ground instance $r\lambda$ of r (with $A\lambda \in body^+(r\lambda)$) and some rule s' (with $A\lambda \in head(s')$) having minimal G-depth $k \leq n$.

By the induction hypothesis, s' is contained in $grd(\Pi')$, hence there exists some rule $r' \in \Pi'$ and

a ground substitution μ such that $s' = r'\mu$. Furthermore, since $A\lambda \in head(r'\mu)$, there exists a set of atoms $B \in head(r')$ such that $B\mu = \{A\lambda\}$, and thus s is of the form

$$head(r\lambda) \cup (head(r'\mu) \setminus B\mu) \leftarrow (body^+(r\lambda) - A\lambda) \cup body^+(r'\mu) \cup \text{not } (body^-(r\lambda) \cup body^-(r'\mu)).$$

Now, since $B\mu = \{A\lambda\}$ we have that $\{A\}$ and B are unified by $\lambda\mu$ (note that r and r' contain no common variables). But then, Π' contains the rule $r''\sigma$ where r'' is of the form

$$head(r) \cup (head(r') \setminus B) \leftarrow (body^+(r) \setminus \{A\}) \cup body^+(r') \cup \text{not } (body^-(r) \cup body^-(r'))$$

and σ is an mgu of $\{A\} \cup B$. Hence, σ is more general than $\lambda\mu$ and thus there exists a ground substitution ρ such that $\sigma\rho = \lambda\mu$ and so $r''\sigma\rho \in grd(\Pi')$. But then we can observe the following:

- $head(r)\sigma\rho = head(r\lambda\mu) = head(r\lambda)$;
- since B is the set of atoms which unify with $\{A\lambda\}$ under $\lambda\mu$ (and thus under $\sigma\rho$), we get $(head(r') \setminus B)\sigma\rho = head(r'\lambda\mu) \setminus B\lambda\mu = head(r'\mu) \setminus \{A\lambda\}$;
- since $A\sigma\rho = A\lambda\mu = A\lambda$ and A occurs only once in $body^+(r)$, we get $(body^+(r) \setminus \{A\})\sigma\rho = body^+(r\lambda\mu) - \{A\lambda\mu\} = body^+(r\lambda) - \{A\lambda\}$;
- $body^+(r')\sigma\rho = body^+(r'\lambda\mu) = body^+(r'\mu)$;
- $body^-(r)\sigma\rho = body^-(r\lambda\mu) = body^-(r\lambda)$;
- $body^-(r')\sigma\rho = body^-(r'\lambda\mu) = body^-(r'\mu)$.

It follows that $s = r''\sigma\rho$ and thus s is contained in $grd(\Pi')$.

PROOF OF (ii): Every rule s in $grd(\Pi')$ is a ground instance $r_s\mu$ of some rule $r_s \in \Pi'$. We thus proceed by induction on the minimal G-depth n of the rule r_s from which s was obtained by ground instantiation.

INDUCTION BASE ($n = 0$): Assume that $s \in grd(\Pi')$ but $s \notin \Pi''$. Since r_s has minimal G-depth 0, it is contained in Π but not in any of the G'_A. From $\Pi' = \Pi \setminus \{r\} \cup G'_A$ it follows, that $r_s \in \Pi$. Now, the only rules in $grd(\Pi)$ for which it may be the case that they are not contained in Π'' are (rules that are equal to) ground instances of r which are removed in Step 8 of the creation of Π''. Hence, s was removed in Step 8, but then $s \notin grd(\Pi)$, for otherwise it would not have been removed, a contradiction.

INDUCTION STEP ($n < 0$): Assume that the statement holds for all rules in $grd(\Pi')$ which are ground instances of rules with minimal G-depth $k \leq n$ with $k > 0$. We will show that it then also holds for all ground instances of rules with minimal G-depth $n + 1$. So suppose that r_s has G-depth $n + 1$. It follows that r_s was obtained from an application of GPPE$_I$ to r and a rule r' having minimal G-depth $k < n$ and thus r_s is of the form $r''\sigma$ with

$$r'' = head(r) \cup (head(r') \setminus B) \leftarrow (body^+(r) \setminus \{A\}) \cup body^+(r') \cup \text{not } (body^-(r) \cup body^-(r')),$$

where σ is an mgu of $\{A\} \cup B$. We then get that

$$s = r_s\mu = r''\sigma\mu.$$

Furthermore, $\sigma\rho$ maps all variables contained in r'' to ground terms. It thus maps all variables in r and r' to ground terms, except those which are contained in B or A but not in r''. So let ρ be a substitution which assigns to each variable X that is contained in B or A but not in r'' some constant $c \in HU(\Pi)$. Then, $\sigma\mu\rho$ is a unifier of $\{A\} \cup B$ and the rules $r\sigma\mu\rho$ and $r'\sigma\mu\rho$ are ground rules contained in $grd(\Pi')$.

Since r' has minimal G-depth $\leq n$ we get by the induction hypothesis, that Π'' contains a rule s' which implies $r'\sigma\mu\rho$, i.e., $head(s') \subseteq r'\sigma\mu\rho$, $body^+(s') \subseteq body^+(r'\sigma\mu\rho)$ and $body^-(s') \subseteq body^+(r'\sigma\mu\rho)$. We proceed by a case distinction.

1. $A\sigma\mu\rho \notin head(s')$: We can observe the following:

   - since $B\sigma\mu\rho = \{A\sigma\mu\rho\}$ and $head(s') \subseteq head(r'\sigma\mu\rho)$, it follows that $head(s') \subseteq (head(r') \setminus B)\sigma\mu\rho = (head(r') \setminus B)\sigma\mu$;
   - $body^+(s') \subseteq body^+(r'\sigma\mu\rho) = body^+(r')\sigma\mu$;
   - $body^-(s') \subseteq body^-(r'\sigma\mu\rho) = body^-(r')\sigma\mu$.

   But then s' implies $r''\sigma\mu = s$.

2. $A\sigma\mu\rho \in head(s')$: Since we have $A\sigma\mu\rho \in body^+(r\sigma\mu\rho)$, in the course of constructing Π'', the rules s' and $r\sigma\mu\rho$ are resolved with each other, leading to the rule s'' which is of the form

   $$head(r\sigma\mu\rho) \cup (head(s') \setminus \{A\sigma\mu\rho\}) \leftarrow (body^+(r\sigma\mu\rho) - \{A\sigma\mu\rho\}) \cup body^+(s') \cup \text{not } (body^-(r\sigma\mu\rho) \cup body^-(s')).$$

   But then we can observe the following:

   - $head(r\sigma\mu\rho) = head(r)\sigma\mu$;
   - since $s' \subseteq r'\sigma\mu\rho$ and under $\sigma\mu\rho$ we have that $\{A\sigma\mu\rho\} = B\sigma\mu\rho$, $head(s') \setminus \{A\sigma\mu\rho\} \subseteq head(r'\sigma\mu\rho) \setminus \{A\sigma\mu\rho\} \subseteq (head(r') \setminus B)\sigma\mu\rho = (head(r') \setminus B)\sigma\mu$;
   - since A is only contained once in $body^+(r)$, $body^+(r\sigma\mu\rho) - \{A\sigma\mu\rho\} = (body^+(r) \setminus \{A\})\sigma\mu$;
   - $body^+(s') \subseteq body^+(r'\sigma\mu\rho) = body^+(r')\sigma\mu$;
   - $body^-(r\sigma\mu\rho) = body^-(r)\sigma\mu$;
   - $body^-(s') \subseteq body^-(r'\sigma\mu\rho) = body^-(r')\sigma\mu$.

   It follows that s'' implies $r''\sigma\rho = s$.

Hence, in both cases Π'' contains a rule which implies s. □

Corollary 4.5.18. *Non-ground GPPE preserves ordinary equivalence.*

4.6 A Final Example

To conclude this chapter, we give an example which demonstrates how our methods can be used together in order to preprocess a program for a translation which is closer to natural language. Consider therefore again the program Π_4 from Chapter 2 which encodes the Hamiltonian path problem. It is given by the following rules:

$$in\_path(X,Y) \vee not\_in\_path(X,Y) \leftarrow edge(X,Y), \tag{4.143}$$
$$reached(X) \leftarrow start\_node(X), \tag{4.144}$$
$$reached(X) \leftarrow reached(Y), in\_path(Y,X), \tag{4.145}$$
$$a\_node\_is\_not\_reached \leftarrow node(X), \text{not } reached(X), \tag{4.146}$$
$$\leftarrow a\_node\_is\_not\_reached, \tag{4.147}$$
$$\leftarrow in\_path(X,Y), in\_path(X,Z), Y \neq Z, \tag{4.148}$$
$$\leftarrow in\_path(Y,X), in\_path(Z,X), Y \neq Z. \tag{4.149}$$

Here, we can first observe that a translation of the Rules (4.146) and (4.147) could lead to a clumsy translation like the following:

It must not be the case that a node is not reached. A node is not reached if there exists a node x such that x is not reached.

Since the predicate $a\_node\_is\_not\_reached$ is not part of an input instance, we can safely apply GPPE$_I$ to replace Rule (4.147) by the rule:

$$\leftarrow node(X), \text{not } reached(X). \tag{4.150}$$

This can lead to the arguably clearer translation:

It must not be the case that there exists a node x such that x is not reached.

For explaining the program, we will in the following ignore the original Rule (4.148) since it only defined an auxiliary predicate for the test part which is of no use anymore. Now we can classify the rules of the transformed program Π_4' into the generate, define, and test parts:

Generate

$$in\_path(X,Y) \vee not\_in\_path(X,Y) \leftarrow edge(X,Y), \tag{4.151}$$

Define

$$reached(X) \leftarrow start\_node(X), \tag{4.152}$$
$$reached(X) \leftarrow reached(Y), in\_path(Y,X), \tag{4.153}$$

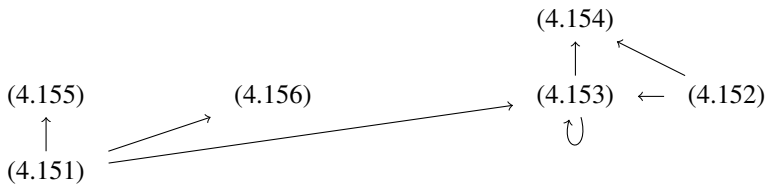

Figure 4.1: The rule graph of Π'_4.

Test

$$\leftarrow node(X), \text{not } reached(X), \tag{4.154}$$
$$\leftarrow in\_path(X,Y), in\_path(X,Z), Y \neq Z, \tag{4.155}$$
$$\leftarrow in\_path(Y,X), in\_path(Z,X), Y \neq Z. \tag{4.156}$$

Note that if we replace for example Rule (4.151) by the following two non-disjunctive rules

$$in\_path(X,Y) \leftarrow edge(X,Y), \text{not } not\_in\_path(X,Y), \tag{4.157}$$
$$not\_in\_path(X,Y) \leftarrow edge(X,Y), \text{not } in\_path(X,Y), \tag{4.158}$$

we get the the result that both rules are in the generate part since they are involved in a negative cycle which is also the case in a bottom reduct. Having identified the generate, define and test parts of the program, it remains to compute an explanation order for the rules. Consider therefore the rule graph of Π'_4 which is given in Figure 4.1.

Since Rule (4.151) is the only generating rule, it is, according to Algorithm 1, explained first. After this, all rules which are constrained by the constraints (4.155) and (4.156) have been explained, so they are the next ones. We then continue with Rule (4.154) since it is the only remaining maximal unlabelled rule. Finally, in a depth-first manner, Rules (4.152) and (4.153) are explained. This leads to the following order:

$$in\_path(X,Y) \vee not\_in\_path(X,Y) \leftarrow edge(X,Y), \quad 1$$
$$\leftarrow in\_path(X,Y), in\_path(X,Z), Y \neq Z, \quad 2$$
$$\leftarrow in\_path(Y,X), in\_path(Z,X), Y \neq Z, \quad 3$$
$$\leftarrow node(X), \text{not } reached(X), \quad 4$$
$$reached(X) \leftarrow start\_node(X), \quad 5$$
$$reached(X) \leftarrow reached(Y), in\_path(Y,X). \quad 6$$

We finish the chapter by giving a natural-language explanation of these rules which is based on the obtained explanation order:

For any edge (x,y) choose whether (x,y) is in the path or not according to the following constraints: It must not be the case that the path contains (x,y) and (x,z) for $y \neq z$. It must not be the case that the path contains (y,x) and (z,x) for $y \neq z$. It must not be the case that there exists a node x such that x is not reached. x is reached if it is a start node or if there exists an y which is reached and (y,x) is in the path.

CHAPTER 5

Conclusion

To conclude this thesis, we first review related approaches and then give an outlook on some possible future work. Finally, we summarise our main results.

5.1 Related Work

Answer-Set Programming and (Controlled) Natural Language. Although we do not know of any existing translations of answer-set programs into a form of natural language, there exist approaches which translate into the opposite direction by using controlled-natural-language (CNL) specifications. Furthermore, CNL is also used for the interpretation of answer sets.

Schwitter [59] defined a translation from different classes of sentences formulated in the language PENG Light [69] to rules of an answer-set program. He demonstrated his results by first translating an arguably clear CNL-specification of the marathon puzzle – a well-known logic puzzle – into an answer-set program and then solving it. He also provided a solution of the so-called jobs puzzle via a CNL specification [60] and showed how defaults can be handled in CNL [61], both via translations into answer-set programs.

Erdem and Yeniterzi [24] defined a CNL called BIOQUERYCNL which can be used for formulating queries over biomedical ontologies. They defined a translation of such queries into answer-set-programs and illustrated their results on some example queries which are very easily readable. Later, even a mechanism which provides natural-language-explanations of the query-answers was developed [23].

Fang [25] developed methods for interpreting the answer sets of a given program in a CNL which she specified. The approach is based on meta information which can be added to answer-set programs via the annotation language LANA [13]. A programmer can thus use LANA to specify the intended meaning of predicates. One could for example add to an atom $hasSkill(E, S)$ the annotation "Employee E has skill S", thereby telling the translation mechanism how certain atoms can be meaningfully explained. The whole functionality was also implemented as a plug-in for the answer-set programming IDE `SeaLion` [9].

Generate-Define-Test and Meta Programming. A compact introduction into the *guess and check paradigm*, which is strongly connected to the generate-define-test paradigm, was given by Eiter, Faber, Leone and Pfeifer [17]. They stated an informal definition of the guess and check parts of a program and provided many examples which demonstrate how complex problems can be elegantly specified by means of guess and check.

Lifschitz [37] was the first who divided example programs into a *generate*, *define*, and *test part*. He listed various answer-set programs together with the comments `GENERATE`, `DEFINE`, and `TEST` to get a clear separation of the three program parts. He also used the comment `DISPLAY`, to identify the part of his program that tells the solver which elements of an answer set should be included in the output.

Eiter and Polleres [22] developed a sophisticated approach for automatically integrating separate programs for guessing and checking into one single program. They started from the observation that most Σ_2^P-complete[1] problems can be solved with answer-set-programming methods by using the following two-step approach:

1. Use a program Π_{guess} to generate a candidate solution S.

2. Check the solution by running a program Π_{check} on S which has *no* answer set if and only if S is a valid solution.

Now, since all problems in Σ_2^P can be solved with extended disjunctive-logic-programs [11], it must be possible to write a single program which has the same input-output behaviour as the described two-step approach.

But such a single program may be hard to find. In the naive union of Π_{guess} and Π_{check} the check part would eliminate valid solutions of the guess part which does not lead to a desired solution. To overcome these problems, they developed a meta logic-program which takes separate Π_{guess} and Π_{check} programs as input and combines them to a single program $\Pi_{solve} = \Pi_{guess} \cup \Pi'_{check}$, where Π'_{check} is obtained from Π_{check} by means of rewriting.

Gebser, Kaminski, and Schaub [28] implemented a meta interpreter for logic programs that computes answer sets which are optimal with respect to complex optimisation statements based on criteria like inclusion-, cardinality- or preference minimality. For the implementation they made use of the reification capabilities provided by the grounder gringo, whose output format they described in a detailed fashion.

Partial-Evaluation Transformations. As already mentioned in Chapter 4, GPPE was introduced into logic programming by Komorowski in 1981 [33, 44]. For propositional disjunctive programs, partial evaluation transformations and the according equivalence results over the stable model semantics were independently introduced by Brass and Dix [4, 5] and Sakama and Seki [57].

Lloyd and Shepherdson [44] introduced partial evaluation for non-ground normal programs and gave conditions under which it preserves equivalence with respect to Clark's completion semantics [58]. Tamaki and Sato [64] introduced an unfold/fold framework on definite programs

[1] Σ_2^P is the class of all problems which can be solved in polynomial time by a non-deterministic Turing-machine which has access to an NP-oracle [51].

and showed that the unfolding preserves equivalence with respect to the minimal model semantics. The framework and the equivalence results were extended to the stable model semantics by Seki [62].

Gergatsoulis [31] took an approach similar to ours: He used the idea of successively applying most general unification to define an unfold transformation on non-ground disjunctive programs containing only positive atoms. He could then show that the transformed program implies (in the classical sense) the same set of positive clauses as the original program.

For folding on a body atom A of a clause r, Gergatsoulis collects all clauses which contain head atoms that are unifiable with A (compare to G_A) and marks all these atoms. Similarly to the successive application of our $\Gamma_{A,r}$-operator, in a loop he then resolves step by step those clauses with r by unifying single marked atoms with A. In this setting, he could then argue that his procedure always terminates after n iterations, where n is the maximum number of marked atoms in the head of a rule.

Another unfolding transformation for non-ground disjunctive programs was introduced by Sakama and Seki [58]. They went around the problem of unifiable head atoms by keeping the transformed rule r in the program if there is a rule r' which contains more than one head atoms which are unifiable with the atom which was unfolded in the body of r. In such cases their transformation can thus be seen as a generalisation of WGPPE to the non-ground case. However, preservation of stable models as well as minimal models was shown for it.

Dix and Stolzenburg also mentioned that in combination with unifiable head atoms, GPPE can lead to problems. They handled those problems by defining a non-ground version of GPPE using methods from constraint logic programming to transform program rules into rules with equational constraints, thereby extending the formalism of logic programming [15, 16]. Without going into too much detail, the following example should illustrate the intuition behind their approach [15]:

Example 5.1.1. Consider program Π_{26}, consisting of the following rules:

$$p(X) \leftarrow q(X), \tag{5.1}$$
$$q(X) \vee q(a) \leftarrow . \tag{5.2}$$

Here the rule (5.2) is split into two new rules using equational constraints (which are written at the right of a rule with a "/"-symbol):

$$p(X) \leftarrow q(X), \tag{5.3}$$
$$q(X) \vee q(a) \leftarrow /X \neq a, \tag{5.4}$$
$$q(a) \leftarrow . \tag{5.5}$$

After this, GPPE can be applied to $q(X)$ in the body of (5.3), and the equational constraints are

incorporated into the newly added rules. This yields the following transformed program:

$$p(X) \vee q(a) \leftarrow /X \neq a, \tag{5.6}$$
$$p(a) \vee q(X) \leftarrow /X \neq a, \tag{5.7}$$
$$p(a) \leftarrow, \tag{5.8}$$
$$q(X) \vee q(a) \leftarrow /X \neq a, \tag{5.9}$$
$$q(a) \leftarrow . \tag{5.10}$$

♢

This way, the problem with unifiable head atoms can be solved, since the equational constraints restrict the domain of certain rules. Note that an equational constraint can be more complex than just a single inequality. More complex constraints can be formed from equalities and inequalities via conjunction, disjunction and even quantifiers.

5.2 Future Work

Building on the results of this thesis, we can develop further results that bring us closer towards our ultimate goal: A procedure which translates an answer-set program into an easy-to-understand explanation which is closer to natural language. But our work raised also questions which are interesting for other purposes.

- Our classification algorithm from Chapter 3 helps us to isolate the generate part of a program from the rest. Now, although answer-set programming is fully declarative, the choice of a suitable encoding can strongly influence the efficiency of the solution process. It is thus often helpful to prune the search space of a given program by incorporating concepts of the test or define part into the generate part [2]. A question that arises in this context is whether such optimisations can be automatised, which would improve the runtime of many existing encodings.

- Although we argued why the application of partial-evaluation transformations is useful in order to obtain clearer natural language explanations, it would still be interesting to further examine when such transformations should be applied and when not. This should optimally lead to a formal criterion which is based on the properties of a given program and whose usefulness is evaluated against a wide range of real-world programs.

- When it comes to translating the rules of a program, the first important task is to find an adequate form of (controlled) natural language which serves as the target language. There already exist a lot of controlled natural languages [35] and so we believe that it is not necessary to develop a completely new one. Once a language is found, defining and implementing an actual translation mechanism is a challenging task for the future.

- Regarding our non-ground partial-evaluation transformation, the following is an interesting open question: Let Π be a logic program and r a rule with a body atom A to which we

apply GPPE$_I$. Let furthermore n be the maximal number of head atoms within a single rule which can be unified with A. Can it be shown (possibly by modifying the transformation) that the number of steps of the $\Gamma_{A,r}$-operator can be limited to n while still preserving input equivalence of the original program and the transformed program?

5.3 Summary

In this thesis, we developed various results which should help to transform answer-set programs into a form which is closer to natural language and easier to understand. We introduced a method which helps to automatically analyse the structure of a given program by classifying its rules into the three parts *generate*, *define*, and *test*. In order to do so, we first introduced formal definitions of these three parts, where the definition of the generate part emerged as the most involved one. The generate part can intuitively be seen as the non-deterministic portion of a program but isolating it from the rest is nontrivial. Syntactical constructs like disjunction or aggregates are obvious sources of non-determinism but a certain use of default-negation can also lead to the creation of multiple answer-sets.

We thus introduced the so called *bottom reduct*, a transformation on programs which eliminates certain rules. Based on the bottom reduct one can then check for negative cycles to obtain a strong indication of non-deterministic behaviour. To implement our classification method we implemented a meta answer-set-program for which the reification method of the grounder `gringo` proved to be very useful. Our implementation was also tested on real-world problems where it led to intuitive classifications. Since the clarity of a program translation strongly depends on the order in which the rules are explained, we proposed such an order which is based on the generate-define-test classification as well as the dependency relation of the predicates.

A straightforward translation of program rules into a form of natural language can lead to rather unnatural results in general. We thus identified partial-evaluation transformations as a helpful tool for preprocessing a program. The problem with these transformation is, that they do only preserve the weak notion of ordinary equivalence and so they cannot be thoughtlessly applied to programs which operate on separate program instances in the form of facts. Because of this, we used the notion of *non-ground relativised uniform equivalence* [49, 70], which we denoted as *input equivalence*. We then developed partial-evaluation transformations, called GPPE$_I$, which preserve input equivalence.

In the propositional case, the definition of an adequate transformation was rather unproblematic and equivalence-preservation could be easily shown, but in the non-ground case things turn out to be rather complicated. The main reason for this is the fact that rules may contain multiple unifiable head atoms. To cope with this problem, we invented the operator $\Gamma_{A,r}$ which is successively applied to the body atom of a rule until a fixed-point criterion is met. The proof that our non-ground transformation preserves equivalence is then based on a sophisticated lifting of the propositional version. Overall, our thesis can be seen as a solid theoretical basis for further work on making a translation of answer-set programs into an easily understandable form of natural language possible.

APPENDIX A

Answer-Set Program for Generate-Define-Test Classification

```
1  #const max_iterations = 10.
2  #const bottom_reduct = -1.
3  #const fact_nr = -1.
4  #hide.
5  #show generate_rule/1.
6  #show layer_changed/1.
7
8  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
9  % Reification and definitions of literals (disjunctive, aggregate, ect.)%
10 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
11
12 % irule(I,R,H,B) - denotes a ground rule
13 % I - Index of iteration in the computation of the bottom-reduct.
14 % R - Index of non-ground rule from which the ground rule was obtained.
15 % H - Head of the rule
16 % B - Body of the rule
17
18 % iset(I,S,L) - denotes that a certain literal is contained within a set
19 % I - Index of iteration in the computation of the bottom-reduct.
20 % S - Index of the set of literals
21 % L - Literal
22
23 irule(0,R,pos(atom(X)),pos(conjunction(Y))) :-
24         rule(pos(atom(X)),pos(conjunction(Y))),
25         set(Y,pos(atom(rule_number(R)))).
26 irule(0,R,pos(disjunction(X)),pos(conjunction(Y))) :-
27         rule(pos(disjunction(X)),pos(conjunction(Y))),
28         set(Y,pos(atom(rule_number(R)))).
29 irule(0,R,pos(aggr(sum,L,X,O)),pos(conjunction(Y))) :-
30         rule(pos(sum(L,X,O)),pos(conjunction(Y))),
31         set(Y,pos(atom(rule_number(R)))).
```

```
32 irule(0,R,pos(aggr(max,L,X,O)),pos(conjunction(Y))) :-
33         rule(pos(max(L,X,O)),pos(conjunction(Y))),
34         set(Y,pos(atom(rule_number(R)))).
35 irule(0,R,pos(aggr(min,L,X,O)),pos(conjunction(Y))) :-
36         rule(pos(min(L,X,O)),pos(conjunction(Y))),
37         set(Y,pos(atom(rule_number(R)))).
38 irule(0,R,pos(aggr(even,1,X,1)),pos(conjunction(Y))) :-
39         rule(pos(even(X)),pos(conjunction(Y))),
40         set(Y,pos(atom(rule_number(R)))).
41 irule(0,R,pos(aggr(odd,1,X,1)),pos(conjunction(Y))) :-
42         rule(pos(odd(X)),pos(conjunction(Y))),
43         set(Y,pos(atom(rule_number(R)))).
44
45 irule(0,fact_nr,X,pos(conjunction(0))) :- rule(X,pos(conjunction(0))).
46 irule(0,fact_nr,pos(atom(X)),pos(conjunction(0))) :-
47         rule(pos(atom(X)),pos(conjunction(0))).
48 irule(0,fact_nr,pos(disjunction(X)),pos(conjunction(0))) :-
49         rule(pos(disjunction(X)),pos(conjunction(0))).
50 irule(0,fact_nr,pos(aggr(sum,L,X,O)),pos(conjunction(0))) :-
51         rule(pos(sum(L,X,O)),pos(conjunction(0))).
52 irule(0,fact_nr,pos(aggr(max,L,X,O)),pos(conjunction(0))) :-
53         rule(pos(max(L,X,O)),pos(conjunction(0))).
54 irule(0,fact_nr,pos(aggr(min,L,X,O)),pos(conjunction(0))) :-
55         rule(pos(min(L,X,O)),pos(conjunction(0))).
56 irule(0,fact_nr,pos(aggr(even,1,X,1)),pos(conjunction(0))) :-
57         rule(pos(even(X)),pos(conjunction(0))).
58 irule(0,fact_nr,pos(aggr(odd,1,X,1)),pos(conjunction(0))) :-
59         rule(pos(odd(X)),pos(conjunction(0))).
60
61 % Rule numberings are body atoms of the predicate 'rule_number' which specify
62 % the non-ground rule from which a ground rule was obtained. They are not
63 % part of the original program and are thus removed at the beginning.
64
65 is_rule_numbering(S,pos(atom(rule_number(R)))) :-
66     set(S,pos(atom(rule_number(R)))).
67
68 iset(0,S,pos(atom(X))) :- set(S,pos(atom(X))),
69                           not is_rule_numbering(S,pos(atom(X))).
70 iset(0,S,neg(atom(X))) :- set(S,neg(atom(X))).
71 iset(0,S,pos(aggr(sum,L,X,O))) :- set(S,pos(sum(L,X,O))).
72 iset(0,S,neg(aggr(sum,L,X,O))) :- set(S,neg(sum(L,X,O))).
73 iset(0,S,pos(aggr(max,L,X,O))) :- set(S,pos(max(L,X,O))).
74 iset(0,S,neg(aggr(max,L,X,O))) :- set(S,neg(max(L,X,O))).
75 iset(0,S,pos(aggr(min,L,X,O))) :- set(S,pos(min(L,X,O))).
76 iset(0,S,neg(aggr(min,L,X,O))) :- set(S,neg(min(L,X,O))).
77 iset(0,S,pos(aggr(even,1,X,1))) :- set(S,pos(even(X))).
78 iset(0,S,neg(aggr(even,1,X,1))) :- set(S,neg(even(X))).
79 iset(0,S,pos(aggr(odd,1,X,1))) :- set(S,pos(odd(X))).
80 iset(0,S,neg(aggr(odd,1,X,1))) :- set(S,neg(odd(X))).
81
82 disjunctive_head_literal(I,R,X) :- irule(I,R,pos(disjunction(Y)),_),
83                                    iset(I,Y,pos(atom(X))).
84
```

```
aggr_head_literal(I,R,L) :- irule(I,R,pos(aggr(_,_,S,_)),_),
                            wlist(S,_,pos(atom(L)),_).
aggr_head_literal(I,R,L) :- irule(I,R,pos(aggr(_,_,S,_)),_),
                            wlist(S,_,neg(atom(L)),_).

aggr_body_literal(I,R,L) :- irule(I,R,H,pos(conjunction(S))),
                            iset(I,S,pos(aggr(_,_,W,_))),
                            wlist(W,_,pos(atom(L)),_).
aggr_body_literal(I,R,L) :- irule(I,R,H,pos(conjunction(S))),
                            iset(I,S,neg(aggr(_,_,W,_))),
                            wlist(W,_,pos(atom(L)),_).
aggr_body_literal(I,R,L) :- irule(I,R,H,pos(conjunction(S))),
                            iset(I,S,pos(aggr(_,_,W,_))),
                            wlist(W,_,neg(atom(L)),_).
aggr_body_literal(I,R,L) :- irule(I,R,H,pos(conjunction(S))),
                            iset(I,S,neg(aggr(_,_,W,_))),
                            wlist(W,_,neg(atom(L)),_).

literal(I,L) :- irule(I,_,pos(atom(L)),_).
literal(I,L) :- irule(I,_,_,pos(conjunction(S))), iset(I,S,pos(atom(L))).
literal(I,L) :- irule(I,_,_,pos(conjunction(S))), iset(I,S,neg(atom(L))).
literal(I,L) :- disjunctive_head_literal(I,_,L).
literal(I,L) :- aggr_head_literal(I,_,L).
literal(I,L) :- aggr_body_literal(I,_,L).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Definitions of dependency notions %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

depends_directly_positive(I,R,X,Y) :-
        irule(I,R,pos(atom(X)),pos(conjunction(B))), iset(I,B,pos(atom(Y))).
depends_directly_positive(I,R,X,Y) :-
        irule(I,R,pos(disjunction(H)),pos(conjunction(B))),
        iset(I,H,pos(atom(X))), iset(I,B,pos(atom(Y))).
depends_directly_positive(I,R,X,Y) :-
        irule(I,R,pos(aggr(_,_,H,_)),pos(conjunction(B))),
        wlist(H,_,pos(atom(X)),_), iset(I,B,pos(atom(Y))).
depends_directly_positive(I,R,X,Y) :-
        irule(I,R,pos(aggr(_,_,H,_)),pos(conjunction(B))),
        wlist(H,_,neg(atom(X)),_), iset(I,B,pos(atom(Y))).
depends_directly_positive(I,R,X,Y) :-
        irule(I,R,pos(aggr(_,_,H,_)),pos(conjunction(B))),
        wlist(H,_,pos(atom(X)),_),
        iset(I,B,pos(aggr(_,_,W,_))), wlist(W,_,pos(atom(Y)),_).
depends_directly_positive(I,R,X,Y) :-
        irule(I,R,pos(aggr(_,_,H,_)),pos(conjunction(B))),
        wlist(H,_,neg(atom(X)),_),
        iset(I,B,pos(aggr(_,_,W,_))), wlist(W,_,neg(atom(Y)),_).

depends_directly_negative(I,R,X,Y) :-
    irule(I,R,pos(atom(X)),pos(conjunction(B))), iset(I,B,neg(atom(Y))).
depends_directly_negative(I,R,X,Y) :-
    irule(I,R,pos(disjunction(H)),pos(conjunction(B))),
```

```
138     iset(I,H,pos(atom(X))), iset(I,B,neg(atom(Y))).
139 depends_directly_negative(I,R,X,Y) :-
140     irule(I,R,pos(aggr(_,_,H,_)),pos(conjunction(B))),
141     wlist(H,_,pos(atom(X)),_), iset(I,B,neg(atom(Y))).
142 depends_directly_negative(I,R,X,Y) :-
143     irule(I,R,pos(aggr(_,_,H,_)),pos(conjunction(B))),
144     wlist(H,_,neg(atom(X)),_), iset(I,B,neg(atom(Y))).
145 depends_directly_negative(I,R,X,Y) :-
146     irule(I,R,pos(aggr(_,_,H,_)),pos(conjunction(B))),
147     wlist(H,_,pos(atom(X)),_),
148     iset(I,B,pos(aggr(_,_,W,_))), wlist(W,_,neg(atom(Y)),_).
149 depends_directly_negative(I,R,X,Y) :-
150     irule(I,R,pos(aggr(_,_,H,_)),pos(conjunction(B))),
151     wlist(H,_,neg(atom(X)),_),
152     iset(I,B,pos(aggr(_,_,W,_))), wlist(W,_,pos(atom(Y)),_).
153
154 depends_positively(I,X,Y) :- depends_directly_positive(I,_,X,Y).
155 depends_positively(I,X,Z) :- depends_positively(I,X,Y),
156                              depends_directly_positive(I,_,Y,Z).
157
158 depends_negatively(I,X,Y) :- depends_directly_negative(I,_,X,Y).
159 depends_negatively(I,X,Z) :- depends_negatively(I,X,Y),
160                              depends_directly_negative(I,_,Y,Z).
161 depends_negatively(I,X,Z) :- depends_positively(I,X,Y),
162                              depends_negatively(I,Y,Z).
163 depends_negatively(I,X,Z) :- depends_negatively(I,X,Y),
164                              depends_positively(I,Y,Z).
165
166 depends(I,X,Y) :- depends_positively(I,X,Y).
167 depends(I,X,Y) :- depends_negatively(I,X,Y).
168
169 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
170 % Definitions of weakly minimal literals and bottom stratum/layer %
171 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
172
173 -weakly_minimal_literal(I,L) :- literal(I,L), depends_negatively(I,L,_).
174 -weakly_minimal_literal(I,L) :- literal(I,L),
175     disjunctive_head_literal(I,_,Y), depends(I,L,Y).
176 -weakly_minimal_literal(I,L) :- literal(I,L),
177     aggr_head_literal(I,_,Y), depends(I,L,Y).
178 weakly_minimal_literal(I,L) :- literal(I,L),
179                                not -weakly_minimal_literal(I,L).
180
181 -bottom_stratum(I,L) :- disjunctive_head_literal(I,_,L).
182 -bottom_stratum(I,L) :- aggr_head_literal(I,_,L).
183 bottom_stratum(I,L) :- weakly_minimal_literal(I,L), not -bottom_stratum(I,L).
184 bottom_layer_rule(I,pos(atom(H)),B) :- irule(I,_,pos(atom(H)),B),
185                                        bottom_stratum(I,H).
186
187 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
188 % Definition of operator phi and bottom reduct %
189 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
190
```

```
% Empty sets of literals are vacuously true
set_contains_an_element(I,S) :- iset(I,S,_).
set_holds_in_bottom_layer(I,S) :- irule(I,_,_,pos(conjunction(S))),
                                  not set_contains_an_element(I,S).

% A set is true if it does not contain a literal which is not true

-set_holds_in_bottom_layer(I,S) :- iset(I,S,pos(atom(L))),
                                   not lit_holds_in_bl(I,L).
set_holds_in_bottom_layer(I,S) :- iset(I,S,_),
                                  not -set_holds_in_bottom_layer(I,S).

% Literals of the bottom stratum are true if they are head atoms of
% a rule in the bottom layer whose body is true. Literals of the
% bottom stratum which are not true in the bottom layer are false.

lit_holds_in_bl(I,L) :-
        bottom_layer_rule(I,pos(atom(L)),pos(conjunction(S))),
        set_holds_in_bottom_layer(I,S).
-lit_holds_in_bl(I,L) :- bottom_stratum(I,L), not lit_holds_in_bl(I,L).

% A rule is not contained in the next iteration of the computation if
% one of its naf-literals in the body is false or if it is a fact.
% Otherwise it is contained in the next iteration

-irule(I+1,R,H,pos(conjunction(B))) :- check(I),
                                       irule(I,R,H,pos(conjunction(B))),
                                       iset(I,B,pos(atom(L))),
                                       -lit_holds_in_bl(I,L).
-irule(I+1,R,H,pos(conjunction(B))) :- check(I),
                                       irule(I,R,H,pos(conjunction(B))),
                                       iset(I,B,neg(atom(L))),
                                       lit_holds_in_bl(I,L).
-irule(I+1,R,pos(atom(L)),B) :- check(I),
                                irule(I,R,pos(atom(L)),B),
                                lit_holds_in_bl(I,L).
irule(I+1,R,H,B) :- check(I), irule(I,R,H,B), not -irule(I+1,R,H,B).

% A set of literals is false if one of its naf-literals is false.

set_is_false(I+1,S) :- check(I), iset(I,S,pos(atom(L))),
                       -lit_holds_in_bl(I,L).
set_is_false(I+1,S) :- check(I), iset(I,S,neg(atom(L))),
                       lit_holds_in_bl(I,L).

% If a set is false then it is not contained in the next iteration

-iset(I+1,S,N) :- check(I), iset(I,S,N), set_is_false(I+1,S).

% If a naf-literal of a set is true in the bottom layer, we remove
% it from the set.

-iset(I+1,S,pos(atom(L))) :- check(I),
```

```
                              iset(I,S,pos(atom(L))), lit_holds_in_bl(I,L).
-iset(I+1,S,neg(atom(L))) :- check(I),
                              iset(I,S,neg(atom(L))), -lit_holds_in_bl(I,L).

iset(I+1,S,L) :- check(I), iset(I,S,L), not -iset(I+1,S,L).

% Check if an iteration leads to a change (if not, the fixed point
% of the operator phi is reached and the computation stops).

layer_changed(I+1) :- irule(I,R,B,H), -irule(I+1,R,B,H).
layer_changed(I+1) :- iset(I,S,L), -iset(I+1,S,L).

check(I) :- layer_changed(I), I < max_iterations.
check(0).

irule(bottom_reduct,R,X,Y) :- irule(I,R,X,Y),
                               layer_changed(I), not layer_changed(I+1).
iset(bottom_reduct,X,Y) :- iset(I,X,Y),
                           layer_changed(I), not layer_changed(I+1).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Definition of generating rules %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Define those rules which are involved in a negative cycle within
% the extended dependency graph of the bottom reduct as 'generating rules'.

in_negative_cycle(R) :- depends_directly_positive(bottom_reduct,R,X,Y),
                        depends_negatively(bottom_reduct,Y,X).
in_negative_cycle(R) :- depends_directly_negative(bottom_reduct,R,X,Y),
                        depends(bottom_reduct,Y,X).

generating_rule(R) :- in_negative_cycle(R).
```

Bibliography

[1] Krzysztof R. Apt, Howard A. Blair, and Adrian Walker. *Towards a theory of declarative knowledge*. IBM Thomas J. Watson Research Center, 1986.

[2] Chitta Baral. *Knowledge Representation, Reasoning, and Declarative Problem Solving*. Cambridge University Press, New York, NY, USA, 2003.

[3] Rachel Ben-Eliyahu and Rina Dechter. Propositional semantics for disjunctive logic programs. *Annals of Mathematics and Artificial Intelligence*, 12(1-2):53–87, 1994.

[4] Stefan Brass and Jürgen Dix. A disjunctive semantics based on unfolding and bottom-up evaluation. In Bernd Wolfinger, editor, *Innovationen bei Rechen- und Kommunikationssystemen*, Informatik aktuell, pages 83–91. Springer, 1994.

[5] Stefan Brass and Jürgen Dix. Characterizations of the disjunctive stable semantics by partial evaluation. *The Journal of Logic Programming*, 32(3):207 – 228, 1997.

[6] Stefan Brass and Jürgen Dix. Disjunctive semantics based upon partial and bottom-up evaluation. In Leon Sterling, editor, *Proceedings of the 12th International Conference on Logic Programming* (*ICLP'95*), pages 199–213. MIT Press, 1995.

[7] Gerhard Brewka, Thomas Eiter, and Mirosław Truszczyński. Answer set programming at a glance. *Communications of the ACM*, 54(12):92–103, 2011.

[8] Gianpaolo Brignoli, Stefania Costantini, Ottavio D'Antona, and Alessandro Provetti. Characterizing and computing stable models of logic programs: The non-stratified case. In *Proceedings of the Conference on Information Technology* (*CIT'99*), pages 197–201, 1999.

[9] Paula-Andra Busoniu, Johannes Oetsch, Joerg Puehrer, Peter Skočovský, and Hans Tompits. SeaLion: An eclipse-based IDE for answer-set programming with advanced debugging support. *Theory and Practice of Logic Programming*, 13(4-5):657–673, 2013.

[10] Stefania Costantini. Comparing different graph representations of logic programs under the answer set semantics. In Alessandro Provetti and Tran Cao Son, editors, *Proceedings of the AAAI Spring Symposium on: Answer Set Programming, Towards Efficient and Scalable Knowledge Representation and Reasoning* (*ASP 2001*), 2001.

[11] Evgeny Dantsin, Thomas Eiter, Georg Gottlob, and Andrei Voronkov. Complexity and expressive power of logic programming. *ACM Computing Surveys*, 33(3):374–425, 2001.

[12] Dick H. J. De Jongh and Lex Hendriks. Characterization of strongly equivalent logic programs in intermediate logics. *Theory and Practice of Logic Programming*, 3:259–270, 2003.

[13] Marina De Vos, Doga Gizem Kısa, Johannes Oetsch, Jörg Pührer, and Hans Tompits. Annotating answer-set programs in LANA. *Theory and Practice of Logic Programming*, 12(4-5):619–637, 2012.

[14] Yannis Dimopoulos and Alberto Torres. Graph theoretical structures in logic programs and default theories. *Theoretical Computer Science*, 170(1):209–244, 1996.

[15] Jürgen Dix and Frieder Stolzenburg. Computation of non-ground disjunctive well-founded semantics with constraint logic programming. In Jürgen Dix, Luís Moniz Pereira, and Teodor C. Przymusinski, editors, *Proceedings of the 2nd International Workshop on Non-Monotonic Extensions of Logic Programming* (*NMELP'96*), volume 1216 of *Lecture Notes in Computer Science*, pages 202–224. Springer, 1997.

[16] Jürgen Dix and Frieder Stolzenburg. A framework to incorporate non-monotonic reasoning into constraint logic programming. *The Journal of Logic Programming*, 37(1–3):47 – 76, 1998.

[17] Thomas Eiter, Wolfgang Faber, Nicola Leone, and Gerald Pfeifer. Declarative problem-solving using the dlv system. In Jack Minker, editor, *Logic-Based Artificial Intelligence*, volume 597 of *The Springer International Series in Engineering and Computer Science*, pages 79–103. Springer, 2000.

[18] Thomas Eiter and Michael Fink. Uniform equivalence of logic programs under the stable model semantics. In Catuscia Palamidessi, editor, *Proceedings of the 19th International Conference on Logic Programming* (*ICLP 2003*), volume 2916 of *Lecture Notes in Computer Science*, pages 224–238. Springer, 2003.

[19] Thomas Eiter, Michael Fink, Hans Tompits, Patrick Traxler, and Stefan Woltran. Replacements in non-ground answer-set programming. In *Proceedings of the 10th International Conference on Principles of Knowledge Representation and Reasoning* (*KR 2006*), pages 340–351, 2006.

[20] Thomas Eiter, Michael Fink, Hans Tompits, and Stefan Woltran. Simplifying logic programs under uniform and strong equivalence. In Vladimir Lifschitz and Ilkka Niemelä, editors, *Proceedings of the 7th International Conference on Logic Programming and Non-monotonic Reasoning* (*LPNMR 2004*), volume 2923 of *Lecture Notes in Computer Science*, pages 87–99. Springer, 2004.

[21] Thomas Eiter, Giovambattista Ianni, and Thomas Krennwallner. Answer set programming: A primer. In Sergio Tessaris, Enrico Franconi, Thomas Eiter, Claudio Gutierrez, Siegfried

Handschuh, Marie-Christine Rousset, and Renate A. Schmidt, editors, *Reasoning Web. Semantic Technologies for Information Systems*, volume 5689 of *Lecture Notes in Computer Science*, pages 40–110. Springer, 2009.

[22] Thomas Eiter and Axel Polleres. Towards automated integration of guess and check programs in answer set programming. In Vladimir Lifschitz and Ilkka Niemelä, editors, *Proceedings of the 7th International Conference on Logic Programming and Nonmonotonic Reasoning* (*LPNMR 2004*), volume 2923 of *Lecture Notes in Computer Science*, pages 114–126. Springer, 2004.

[23] Esra Erdem and Umut Oztok. Generating explanations for biomedical queries. *Theory and Practice of Logic Programming*, pages 1–44, 2013.

[24] Esra Erdem and Reyyan Yeniterzi. Transforming controlled natural language biomedical queries into answer set programs. In *Proceedings of the Workshop on Current Trends in Biomedical Natural Language Processing* (*BioNLP 2009*), pages 117–124. Association for Computational Linguistics, 2009.

[25] Min Fang. A controlled natural language approach for interpreting answer sets. Bachelor's thesis, Vienna University of Technology, 2013.

[26] Calimeria Francesco, Faber Wolfgang, Gebser Martin, Ianni Giovambattista, Kaminski Roland, Krennwallner Thomas, Leone Nicola, Ricca Francesco, and Schaub Torsten. Aspcore-2, input language format, 2012.

[27] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Max Ostrowski, Torsten Schaub, and Sven Thiele. A user's guide to gringo, clasp, clingo, and iclingo. Unpublished draft, 2008. Retrieved from `http://downloads.sourceforge.net/potassco/guide.pdf`.

[28] Martin Gebser, Roland Kaminski, and Torsten Schaub. Complex optimization in answer set programming. *Theory and Practice of Logic Programming*, 11(4-5):821–839, 2011.

[29] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In *Proceedings of the 5th International Conference and Symposium on Logic Programming* (*ICLP/SLP'88*), pages 1070–1080. MIT Press, 1988.

[30] Michael Gelfond and Vladimir Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9:365–385, 1991.

[31] Manolis Gergatsoulis. Unfold/fold transformations for disjunctive logic programs. *Information Processing Letters*, 62(1):23–29, 1997.

[32] Kazuo Iwama and Shuichi Miyazaki. A survey of the stable marriage problem and its variants. In *Proceedings of the International Conference on Informatics Education and Research for Knowledge-Circulating Society* (*ICKS 2008*), pages 131–136, 2008.

[33] Henryk Jan Komorowski. *A specification of an abstract Prolog machine and its application to partial evaluation*. PhD thesis, Linköping University, The Institute of Technology, 1981.

[34] Robert Kowalski. Predicate logic as programming language. In Jack L. Rosenfeld, editor, *Proceedings of the 6th Congress of the International Federation for Information Processing* (*IFIP'74*), volume 74, pages 569–544, 1974.

[35] Tobias Kuhn. A survey and classification of controlled natural languages. *Computational Linguistics*, 40(1):121–170, 2014.

[36] Alexander Leitsch. *The resolution calculus*. Texts in theoretical computer science. Springer, 1997.

[37] Vladimir Lifschitz. Answer set programming and plan generation. *Artificial Intelligence*, 138(1–2):39 – 54, 2002.

[38] Vladimir Lifschitz. Thirteen definitions of a stable model. In *Fields of logic and computation*, pages 488–503. Springer, 2010.

[39] Vladimir Lifschitz, David Pearce, and Agustín Valverde. Strongly equivalent logic programs. *ACM Transactions on Computational Logic*, 2(4):526–541, 2001.

[40] Vladimir Lifschitz and Hudson Turner. Splitting a logic program. In Pascal Van Hentenryck, editor, *Proceedings of the 11th International Conference on Logic Programming* (*ICLP'94*), pages 23–37, 1994.

[41] Antoni Ligeza. *Logical Foundations for Rule-Based Systems*, volume 11 of *Studies in Computational Intelligence*. Springer, 2006.

[42] Fangzhen Lin. Reducing strong equivalence of logic programs to entailment in classical propositional logic. In Dieter Fensel, Fausto Giunchiglia, Deborah L. McGuinness, and Mary-Anne Williams, editors, *Proceedings of the 8th International Conference on Principles of Knowledge Representation and Reasoning* (*KR 2002*), pages 170–176. Morgan Kaufmann, 2002.

[43] John W. Lloyd. Practical advantages of declarative programming. In *Proceedings of the Joint Conference on Declarative Programming* (*GULP-PRODE'94*), volume 94, page 94, 1994.

[44] John W. Lloyd and John C. Shepherdson. Partial evaluation in logic programming. *The Journal of Logic Programming*, 11(3):217–242, 1991.

[45] Donald W. Loveland. *Automated Theorem Proving: A Logical Basis*. Fundamental Studies in Computer Science. North-Holland Pub. Co. New York, Amsterdam, New York, Oxford, 1978.

[46] M.J. Maher. Equivalences of logic programs. In Ehud Shapiro, editor, *Proceedings of the 3rd International Conference on Logic Programming* (*ICLP'86*), volume 225 of *Lecture Notes in Computer Science*, pages 410–424. Springer, 1986.

[47] D. G. McVitie and L. B. Wilson. The stable marriage problem. *Communications of the ACM*, 14(7):486–490, 1971.

[48] Robert C. Moore. Semantical considerations on nonmonotonic logic. *Artificial Intelligence*, 25(1):75 – 94, 1985.

[49] Johannes Oetsch and Hans Tompits. Program correspondence under the answer-set semantics: The non-ground case. In *Proceedings of the 24th International Conference on Logic Programming* (*ICLP 2008*), volume 5366 of *Lecture Notes in Computer Science*.

[50] Mauricio Osorio, Juan Antonio Navarro Pérez, and José Arrazola. Equivalence in answer set programming. In Alberto Pettorossi, editor, *Proceedings of the 11th International Workshop on Logic Based Program Synthesis and Transformation* (*LOPSTR 2001*), volume 2372 of *Lecture Notes in Computer Science*, pages 57–75. Springer, 2001.

[51] Christos H Papadimitriou. *Computational complexity*. John Wiley and Sons Ltd., 2003.

[52] David Pearce, Hans Tompits, and Stefan Woltran. Encodings for equilibrium logic and logic programs with nested expressions. In Pavel Brazdil and Alípio Jorge, editors, *Progress in Artificial Intelligence*, volume 2258 of *Lecture Notes in Computer Science*, pages 306–320. Springer, 2001.

[53] H. Przymusinska and T. C. Przymusinski. Weakly stratified logic programs. *Fundamenta Informaticae*, 13(1):51–65, 1990.

[54] T. C. Przymusinski. On the declarative semantics of deductive databases and logic programs. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 193–216. Morgan Kaufmann, San Francisco, CA, USA, 1988.

[55] Teodor C. Przymusinski. Three-valued nonmonotonic formalisms and semantics of logic programs. *Artificial intelligence*, 49(1):309–343, 1991.

[56] Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming*. MIT Press, Cambridge, MA, USA, 2004.

[57] Chiaki Sakama and Hirohisa Seki. Partial deduction of disjunctive logic programs: A declarative approach. In *Logic Program Synthesis and Transformation—Meta-Programming in Logic*, pages 170–182. Springer, 1994.

[58] Chiaki Sakama and Hirohisa Seki. Partial deduction in disjunctive logic programming. *The Journal of Logic Programming*, 32(3):229 – 245, 1997.

[59] Rolf Schwitter. Answer set programming via controlled natural language processing. In *Controlled Natural Language*, pages 26–43. Springer, 2012.

[60] Rolf Schwitter. The jobs puzzle: Taking on the challenge via controlled natural language processing. *Theory and Practice of Logic Programming*, 13(4-5):487–501, 2013.

[61] Rolf Schwitter. Working with defaults in a controlled natural language. In *Proceedings of the Australasian Language Technology Association Workshop* (*ALTA 2013*), page 106, 2013.

[62] Hirohisa Seki. A Comparative Study of the Well-Founded and the Stable Model Semantics: Transformation's Viewpoint. In *Proceedings of the 6th International Conference on Logic Programming and Nonmonotonic Reasoning* (*LPNMR'90*), volume 2173 of *Lecture Notes in Computer Science*, pages 115–123. Springer, 1990.

[63] Apostolos Syropoulos. Mathematics of multisets. In Cristian Calude, Gheorghe Paun, Grzegorz Rozenberg, and Arto Salomaa, editors, *Proceedings of the Workshop on Multiset Processing: Mathematical, Computer Science, and Molecular Computing Points of View* (*WMP 2000*), volume 2235 of *Lecture Notes in Computer Science*, pages 347–358. Springer, 2000.

[64] Hisao Tamaki and Taisuke Sato. Unfold/fold transformation of logic programs. In Sten-Åke Tärnlund, editor, *Proceedings of the 2nd International Conference on Logic Programming* (*ICLP'84*), pages 127–138. Uppsala University, 1984.

[65] Hudson Turner. Strong equivalence for logic programs and default theories (made easy). In Thomas Eiter, Wolfgang Faber, and Miroslaw Truszczyński, editors, *Proceedings of the 6th International Conference on Logic Programming and Nonmotonic Reasoning* (*LPNMR 2001*), volume 2173 of *Lecture Notes in Computer Science*, pages 81–92. Springer, 2001.

[66] Hudson Turner. Strong equivalence made easy: Nested expressions and weight constraints. *Theory and Practice of Logic Programming*, 3:609–622, 7 2003.

[67] Allen Van Gelder, Kenneth A. Ross, and John S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):619–649, 1991.

[68] Kewen Wang and Lizhu Zhou. Comparisons and computation of well-founded semantics for disjunctive logic programs. *ACM Transactions on Computational Logic*, 6(2):295–327, April 2005.

[69] Colin White and Rolf Schwitter. An update on PENG Light. In *Proceedings of the Australasian Language Technology Association Workshop* (*ALTA 2009*), volume 7, pages 80–88, 2009.

[70] Stefan Woltran. Characterizations for relativized notions of equivalence in answer set programming. In Jóse Júlio Alferes and João Leite, editors, *Proceedings of the 9th European Conference on Logics in Artificial Intelligence* (*JELIA 2004*), volume 3229 of *Lecture Notes in Computer Science*, pages 161–173. Springer, 2004.