DECIDABILITY OF REASONING UNDER THE WELL-FOUNDED SEMANTICS

Master's Thesis

by

Natalia Cherchago

Dresden University of Technology

March 2010

Supervisors: Prof. Dr. Steffen Hölldobler

Prof. Dr. Pascal Hitzler

Table of Contents

Abstract

In this thesis we investigate issues concerning decidability of reasoning under the well-founded semantics. First, we give an overview of meta-level properties of the well-founded semantics. We also define a new property—we call it the *maximal path property*. It tells us that the number of W_P-iterations in the well-founded model computation is limited by the cardinality of the longest dependency path in the atom-call graph of the program.

We propose a new setting for analysis of decidability of logic programs. We focus our attention on properties of query atoms and their relevant subprograms rather than on a logic program as a whole. Our observation is that within a logic program P various atoms in its Herbrand base can have various properties, for example, finiteness of the dependency relation or length of the maximal dependency path. Within a single program P different query atoms have different relevant subprograms and thus yield different decidability results. We divide a Herbrand base of a logic program P into two subsets with respect to the cardinality of an atom dependency relation. Analysis of the meta-properties and their combination leads us to the definition of three classes of atoms with decidable query evaluation: *finitely recursive,* ω*-recursive* and ω*-restricted* atoms. *Finitely recursive atoms* with a finite dependency relation enjoy decidability of query evaluation because their relevant subprograms are finite; ω*-recursive atoms* have an infinite dependency relation, nevertheless they are decidable because their value can be computed in finite time due to the *maximal path property*; ω*-restricted atoms* have relevant subprograms with a certain syntactic restriction that allows for model computation to be a 2-**NEXP**-complete problem.

Acknowledgements

I would like to thank my supervisors. I am grateful to Prof. Dr. Steffen Hölldobler for his constant support and stimulating discussions that contributed to the growth of this work. I thank Prof. Dr. Pascal Hitzler for his idea, patience, many precious suggestions and deep and careful reviews of this work.

Introduction

The well-founded semantics for normal logic programs is one of the most widely studied and commonly accepted paradigms for nonmonotonic reasoning under the closed-world assumption [vGRS91, ADP95, Fit91, CSW95, BD98]. It was implemented in several top-down reasoning systems, most prominent of which is XSB [RSS+97].

The relevance and actuality of studies on the subject of reasoning under the well-founded semantics can be motivated by its closed relation to the well-known and widely used mechanisms in the field of modern information technologies—to Answer Set Programming (ASP) and F-logic, and finally, by means of both, to the Semantic Web. The Semantic Web[1] is an evolving development of the World Wide Web in which the meaning of information and services on the web is defined, making it possible for the web to "understand" and satisfy the requests of people and machines to use the web content.

As a semantics of XSB, the well-founded semantics lies in the core system engine for implementation of an F-logic formalism, which is often viewed as a logic programming language with an object-oriented approach. With the advent of the Semantic Web, especially, with the development of the RDF standard, there has been a new wave of interest in systems for logic-based processing of object-oriented meta-data. Being a rule-language, F-logic is a part of the Rule Interchange Format (RIF) effort, that is, an initiative within the Semantic Web activity to devise standards for interoperability of existing rule-based technologies.

Answer Set Programming is a nonmonotonic problem-solving framework. It has

[1]The definition of the Semantic Web is taken from `http://en.wikipedia.org/wiki/Semantic_Web`

sophisticated, well-optimized implementations and has been shown competitive with other frameworks on several benchmarks. It has been applied, for instance, to plan generation and product configuration problems in artificial intelligence and to graph-theoretic problems arising in VLSI[2] design and in historical linguistics. Recently, it has also been widely used to implement the ideas of the Semantic Web project.

As an approximation of stable model semantics [vGRS91, Fit02], which lies in the basis of ASP, the well-founded semantics plays an important role in research dedicated to ASP computation methods. Computing the well-founded model of propositional programs is polynomial [vG89] while computing stable models is NP-hard [MT91]. Consequently, evaluating the well-founded semantics can be used as an effective pre-processing technique in algorithms to compute stable models [SNV95]. In addition, as demonstrated by `smodels` [NS96] and `dlv` [EFLP00], at present the most advanced and most efficient systems to compute stable models of DATALOG programs, the well-founded semantics can be used as a powerful lookahead mechanism.

It was shown by Schlipf in [Sch95] that the well-founded semantics and the stable model semantics over infinite Herbrand universes have the same expressive power. The problem of deciding whether a tuple of elements of the Herbrand universe is a correct answer for a query is highly non-recursive, i.e. not even semi-decidable. This has led to the fact that state-of-the-art answer set solvers have a major drawback—they work only on finite domains, finite sets of constants.

Our goal is to define decidable subsets of the well-founded semantics, continuing the studies on this problem started in [CHH07]. Based on the decidable classes of logic programs defined in [CHH07], we want to perform a further analysis of the matter, and, if possible, generalize or modify the conditions that define decidable subsets. As a powerful tool of decidability analysis, we use the meta-properties of the semantics in the art of Dix [Dix95b].

The report is organized as follows. Chapter 1 introduces key notions and terminology. Chapter 2 is dedicated to the well-founded semantics. We start with decidability

[2]Very-large-scale integration (VLSI) is the process of creating integrated circuits by combining thousands of transistor-based circuits into a single chip. The definition is taken from `http://en.wikipedia.org/wiki/Very-large-scale_integration`

issues and then give an account of abstract properties of the well-founded semantics, such as *relevance* [Dix95b], the *level-mapping characterization* [HW02, HW05] and *stratified relevance* [CHH07]. Our contribution in this chapter is contained in Section 2.6, where we define the *maximal path property*. In Chapter 3 we present three classes of atoms with decidable query evaluation. The first class, *finitely recursive atoms* in Section 3.1, is based on the definition of finitely recursive programs from [Bon01b] and [CHH07], but we re-collect it here in a new setting. The subsequent class of $\omega-$*recursive atoms* in Section 3.2 is introduced on the basis of the *maximal path property*. The last class of atoms, $\omega-$*restricted atoms* in Section 3.4, is based on the definition of $\omega-$restricted programs by Syrjänen in [Syr01]. We prove decidability results, compare these classes of atoms and provide examples. We conclude with Chapter 4 including discussion of applicability in Section 4.1, related work account in Section 4.2 and summary in Section 4.3.

Chapter 1

Preliminaries

In this chapter we review the basic notions of logic programming and the well-founded semantics. For a more detailed account of the material given here the main references are [Llo87] and [vGRS91], respectively.

1.1 Logic Programs

To make this work self-contained, we give here basic definitions, starting with an alphabet and concluding with a notion of a normal logic program.

Definition 1.1.1. (Alphabet) An *alphabet* Σ consists of the following disjoint sets of symbols:

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

Please note that the sets of variables, function symbols and predicate symbols are countably infinite.

Definition 1.1.2. (Term) A *term* is defined inductively as follows:

1. A variable is a term.
2. A constant is a term.
3. If f is a function symbol with arity n and $t_1, ..., t_n$ are terms, then $f(t_1, ..., t_n)$ is a term.
4. There are no other terms.

Definition 1.1.3. (Atom) Let p be a predicate symbol with arity $n \in \mathbb{N}$. Let $t_1, ..., t_n$ be terms. Then $p(t_1, ..., t_n)$ is an *atomic formula* or *atom*. We also set $pred(p(t_1, ..., t_n) := p)$.

Definition 1.1.4. (Literal) A *literal* is a positive literal or a negative literal. A *positive literal* is an atom A. A *negative literal* is the negation $\neg A$ of an atom A.

Definition 1.1.5. (Complement) Let L be a literal and B its atom. The *complement* $\neg L$ of L is defined as follows:

$$\neg L := \begin{cases} \neg B & \text{if } L = B \\ B & \text{if } L = \neg B \end{cases}$$

For a set S of literals, $\neg S$ denotes the set of the complements of the literals in S: $\neg S := \{\neg L \mid L \in S\}$

Definition 1.1.6. (Positive and Negative Atoms) For a set S of literals we define the set $pos(S)$ of *positively occurring atoms* in S and the set $neg(S)$ of *negatively occurring atoms* in S as follows:

$pos(S) := \{A \mid A \in S \text{ is a positive literal in } S\}$,

$neg(S) := \{A \mid \neg A \in S \text{ is a negative literal in } S\}$.

Definition 1.1.7. (Well-Formed Formula) A *(well-formed) formula* is defined inductively as follows:

1. An atom is a formula.
2. If F and G are formulas, then so are $\neg F, F \wedge G, F \wedge G, F \rightarrow G$, and $F \leftrightarrow G$.

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

4. There are no other formulas.

It will often be convenient to write the formula $F \rightarrow G$ as $G \leftarrow F$.

Definition 1.1.8. (First Order Language) The *first order language* $\mathcal{L}$ for an alphabet Σ consists of the set of all formulas that can be constructed according to Definition 1.1.7 using symbols of Σ.

By $\mathcal{L}_p$ we shall define the language of propositional (quantifier-free) logic.

Definition 1.1.9. (Occurrence of a Variable) The set $\mathcal{V}(t)$ of variables that *occur in* a term is defined as follows:

$$\mathcal{V}(t) := \begin{cases} \emptyset, & \text{if } t \text{ is a constant,} \\ \{t\}, & \text{if } t \text{ is a variable,} \\ \bigcup_{i=1}^{m} \mathcal{V}(t_i), & \text{if } t \text{ is of the form} f(t_1, ..., t_m). \end{cases}$$

$\mathcal{V}$ is extended to literals as follows:
$\mathcal{V}(p(t_1, ..., t_n)) := \bigcup_{i=1}^{n} \mathcal{V}(t_i)$ and $\mathcal{V}(\neg p(t_1, ..., t_n)) := \bigcup_{i=1}^{n} \mathcal{V}(t_i)$, where p is a predicate symbol and t_i are terms.

Definition 1.1.10. (Rule) The *program clause* or *normal rule* R is of the form

$$A \leftarrow L_1 \wedge ... \wedge L_n$$

where A is an atom and L_i are literals. A is called the *head* of the rule and is denoted $head(R)$, the L_i are called *body literals*, and their conjunction $L_1 \wedge ... \wedge L_n$ is called the *body* of the rule and is denoted $body(R)$ or simply $\mathcal{B}$. The sets of positive and negative literals in the body will be denoted $body^+(R)$ and $body^-(R)$, respectively. And $lit(R)$ stands for $head(R) \cup body(R)$.

The set of variables that occur in a rule R is defined in terms of variables that occur in its literals:

$$\mathcal{V}(R) = \mathcal{V}(head(R)) \cup \bigcup_{L \in body(R)} \mathcal{V}(L).$$

Definition 1.1.11. (Groundness) A term, atom, rule or program with no variables occurring in it is called *ground*.

Definition 1.1.12. (Fact) A *fact* is a (normal) rule without any body literals. We will denote a fact just by its head atom A, or in the set notation by $A \leftarrow \emptyset$.

Definition 1.1.13. (Query) A *query* or a *goal* is a rule of the form

$\leftarrow L_1, ..., L_n$

having no atom in the consequent.

A query or a goal is called *atomic* if it consists of a single atom.

Definition 1.1.14. (Normal Program) A *normal program* is a set of normal rules.

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

Definition 1.1.15. (Definite Rule) A *definite rule* or *positive rule* is a normal rule of the form

$A \leftarrow B_1, ..., B_n$

having no negative body literals.

Definition 1.1.16. (Definite Program) A *definite program* or *positive program* is a set of definite rules.

Definition 1.1.17. (Facts, Heads, Predicates) Let P be a program. We define the *set of facts* in P, $facts(P)$, the *set of heads* in P, $heads(P)$ and the *predicates* in P, $preds(P)$, as follows:

$facts(P) := \{A \mid (A \leftarrow \emptyset) \in P\}$,

$heads(P) := \{A \mid$ there is a possibly empty *body* such that $(A \leftarrow body) \in P\}$,

$preds(P) := \{pred(R) \mid R \in P\}$.

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

Definition 1.1.18. (Predicate Definition) The definition of a predicate p with respect to the program P, denoted by $def(p)$, is the set of rules in P with head predicates p, i.e.
$def(p) := \{R \mid R \in P$ and $pred(head(R)) = p\}$.

Definition 1.1.19. (Substitution) Let $\mathcal{L}$ be a first order language. A *substitution* θ in $\mathcal{L}$ is a finite set of the form $\{V_1/t_1, ..., V_n/t_n\}$, where each V_i is a variable in $\mathcal{L}$, each t_i is a term in $\mathcal{L}$ distinct from V_i and the variables $V_1, ..., V_n$ are pairwise distinct. Each element V_i/t_i is called a *binding* for V_i. θ is called a *ground substitution* if the t_i are all ground terms. θ is called a *variable-pure substitution* if the t_i are all variables. We will omit the language $\mathcal{L}$ when it is clear from the context.

Definition 1.1.20. (Renaming Substitution) Let E be an expression and $\mathcal{V}$ be the set of variables occurring in E. A *renaming substitution* for E is a variable-pure substitution $\{X_1/Y_1, ..., X_n/Y_n\}$ such that $\{X_1, ..., X_n\} \subseteq \mathcal{V}$, the Y_i are pairwise distinct and $(\mathcal{V} \backslash \{X_1, ..., X_n\}) \cap \{Y_1, ..., Y_n\} = \emptyset$.

1.2 Interpretations and Models

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

Definition 1.2.1. (Pre-Interpretation) A *pre-interpretation* of a first order language $\mathcal{L}$ consists of the following:

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

Definition 1.2.2. (Interpretation) An *interpretation* I of a first order language $\mathcal{L}$ consists of a pre-interpretation J with domain $\mathcal{D}$ of $\mathcal{L}$ together with the following:

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

We say I is *based on* J.

Definition 1.2.3. (Variable Assignment) Let J be a pre-interpretation of a first order language $\mathcal{L}$. A *variable assignment* w.r.t. J is an assignment of an element in the domain of J to each variable in $\mathcal{L}$.

Definition 1.2.4. (Term Assignment) Let J be a pre-interpretation of a first order language $\mathcal{L}$ with domain $\mathcal{D}$. Let V be a variable assignment. The *term assignment* w.r.t. J and V of the terms in $\mathcal{L}$ is defined as follows:

1. Each variable is given its assignment according to V.

2. Each constant is given its assignment according to J.

3. If $t_1^{'}, ..., t_n^{'}$ are the term assignments of $t_1, ...t_n$ and $f^{'}$ is the assignment of the function symbol f with arity n, then $f^{'}(t_1^{'}, ..., t_n^{'}) \in \mathcal{D}$ is the term assignment of $f(t_1, ..., t_n)$.

Definition 1.2.5. (Truth Values) Let I be an interpretation of a first order language $\mathcal{L}$ with domain $\mathcal{D}$ and let $\mathcal{V}$ be a variable assignment. Then a formula F in $\mathcal{L}$ can be given a *truth value*, *true* or *false* (w.r.t. I and $\mathcal{V}$) as follows:

1. If the formula is an atom $p(t_1, ..., t_n)$, then the truth value is obtained by calculating the value of $p'(t'_1, ..., t'_n)$, where p' is a mapping assigned to p by I and $t'_1, ..., t'_n$ are the term assignments of $t_1, ..., t_n$ w.r.t. I and $\mathcal{V}$.

2. If the formula has the form $\neg F, F \wedge G, F \vee G, F \rightarrow G, F \leftrightarrow G$, then the truth value of the formula is given by the following table:

F	G	$\neg F$	$F \wedge G$	$F \vee G$	$F \rightarrow G$	$F \leftrightarrow G$
true	true	false	true	true	true	true
true	false	false	false	true	false	false
false	true	true	false	true	true	false
false	false	true	false	false	true	true

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

4. If the formula has the form $(\forall X)F$, then the truth value of the formula is *true* if, for all $d \in \mathcal{D}$, we have that F has truth value *true* w.r.t. I and $\mathcal{V}(X/d)$; otherwise, its truth value is *false*.

Definition 1.2.6. (Model) Let I be an interpretation of a first order language $\mathcal{L}$. Then I is a *model* of a closed formula F, if F is true w.r.t. I. Further, I is a model of a set S of closed formulas, if I is a model of each formula of S.

Definition 1.2.7. (Herbrand Universe) Let $\mathcal{L}$ be a first order language with at least one constant. The *Herbrand universe* $U_{\mathcal{L}}$ of $\mathcal{L}$ is the set of all ground terms, which can be formed out of the constants and function symbols of $\mathcal{L}$.

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

Definition 1.2.8. (Herbrand Base) Let $\mathcal{L}$ be a first order language. The *Herbrand base* $B_{\mathcal{L}}$ of $\mathcal{L}$ is the set of all ground atoms which can be formed by using predicate symbols from $\mathcal{L}$ with ground terms from $U_{\mathcal{L}}$ as arguments.

Definition 1.2.9. (Herbrand Instantiation) Let $\mathcal{L}$ be a first order language. The *Herbrand instantiation* $ground_{\mathcal{L}}(P)$ of P consists of all ground instances (w.r.t. the Herbrand universe $U_{\mathcal{L}}$) of all rules in P. We will omit the language $\mathcal{L}$, when it is clear from the context.

We will also assign a Herbrand universe U_P and a Herbrand base B_P to a program P by assuming that the underlying first order language $\mathcal{L}_P$ consists of exactly the constants, function symbols and predicate symbols occurring in the program. The ground instantiation $ground(P)$ of P consists of all ground instances w.r.t the Herbrand base B_P of all the rules in P.

Definition 1.2.10. (Herbrand Pre-Interpretation) Let $\mathcal{L}$ be a first order language. The *Herbrand pre-interpretation* for $\mathcal{L}$ is given by the following:

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

Definition 1.2.11. (Herbrand Interpretation) A *Herbrand interpretation* is an interpretation based on a Herbrand pre-interpretation.

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

Definition 1.2.12. (Herbrand Model) Let $\mathcal{L}$ be a first order language and S a set of closed formulas of $\mathcal{L}$. A *Herbrand model* of S is a Herbrand interpretation of $\mathcal{L}$ which is a model of S.

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

Proposition 1.2.1. (Model Intersection Property) *Let P be a definite program. Let M be a non-empty set of Herbrand models for P. Then the intersection $\bigcap M$ of all models is an Herbrand model for P.*

Since every program P has B_P as a Herbrand model, the set of all Herbrand models for P is always non-empty.

Definition 1.2.13. (Least Herbrand Model) Let P be a definite program. Then the *least Herbrand model* M_P of P is the intersection of all Herbrand models for P.

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

Definition 1.2.14. (Partial and Total Interpretation) Let P be a normal program. A *partial interpretation* I is a set of ground literals such that for no atom A both A and $\neg A$ are contained in I, i.e.

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

and whose atoms are contained in the Herbrand base B_P of P , i.e.

$$pos(I) \cup neg(I) \subseteq B_P.$$

I is a *total interpretation*, if I is a partial interpretation and for every atom $A \in B_P$ it contains either A or $\neg A$, i.e.

$$pos(I) \cup neg(I) = B_P.$$

Definition 1.2.15. (Truth Values) Let I be a partial interpretation. A ground literal L is *true* in I, if L is contained in I, i.e. $L \in I$. L is *false* in I, if its complement $\neg L$ is contained in I, i.e. $\neg L \in I$. L is *undefined* in I, if it is neither *true* nor *false* in I.

We can see here that a partial interpretation is represented by a disjoint pair of sets, $\langle pos(I), neg(I)\rangle$, $pos(I) \cap neg(I) = \emptyset$, with the following meaning:

- members of $pos(I)$ have the truth value *true*,
- members of $neg(I)$ have the truth value *false*,
- and members of neither are mapped to *unknown*.

There are two common partial orderings on truth values, the *truth ordering* and the *knowledge ordering*. By a *partial ordering* we mean a binary relation which is reflexive, antisymmetric and transitive.

Truth Ordering $\leq_t$: $false \leq_t undefined, undefined \leq_t true$

Knowledge Ordering $\leq_k$: $undefined \leq_k false, undefined \leq_k true$

These partial orderings can be generalized (pointwise) to partial Herbrand interpretations as follows: for $x \in \{t, k\}$,

$I_1 \leq_x I_2$ iff $I_1(A) \leq_x I_2(A)$ for all $A \in B_P$.

Truth ordering reflects truth content of an interpretation, while knowledge ordering reflects the amount of knowledge contained in an interpretation. For example, given two interpretations $I_1 = \{p, q, \neg r\}$ and $I_2 = \{p, q\}$, we order them in the following way:

- $I_1 \leq_t I_2$ because $I_1(r) \leq_t I_2(r)$, but
- $I_1 \geq_k I_2$ because $I_1(r) \geq_k I_2(r)$,

since r is evaluated to *false* in I_1 and to *undefined* in I_2. With representation of partial interpretations by two sets, of positive and negative literals, we have: $I_1 \leq_k I_2$ if $pos(I_1) \subseteq pos(I_2)$ and $neg(I_1) \subseteq neg(I_2)$.

In the following, we will often use a shorthand notation for the truth values *true, false* and *undefined*–we denote them *t, f* and *u*, respectively.

Definition 1.2.16. (Partial and Total Model) Let P be a normal program. We say that a ground rule $A \leftarrow \mathcal{B}$ is *satisfied* in a (partial or total) interpretation I, if its head A is true in I or if at least one body literal $L \in \mathcal{B}$ is false in I. I is a *total model* of P, if it is a total interpretation and every ground instance $A \leftarrow \mathcal{B} \in ground(P)$ of a rule of P is satisfied in I. I is a *partial model* of P, if it is a partial interpretation that can be extended (by adding literals to I) to a total model of P.

A partial model is a partial interpretation such that some instantiated rules may not be satisfied, but there is a (possibly empty) set of literals whose addition to the partial interpretation will satisfy all rules. For example, consider the program $P = \{p \leftarrow q, q \leftarrow \neg r\}$. $I = \{p, q\}$ is a partial model of P, it can be extended to the total model by adding the literals r or $\neg r$, but even if we do not add any literals (empty set), still all rules of P are satisfied.

1.3 The Well-Founded Semantics

In this section we briefly outline the notion of the well-founded semantics originally defined in [vGRS91].

Definition 1.3.1. (Unfounded Set) Let P be a normal program. Let I be a partial interpretation. Let $\mathcal{A} \subseteq B_P$ be a set of ground atoms. $\mathcal{A}$ is an unfounded set of P w.r.t. I, if for every atom $A \in \mathcal{A}$ and every ground rule instance $A \leftarrow \mathcal{B} \in ground(P)$ at least one of the following conditions holds:

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

Definition 1.3.2. (Greatest Unfounded Set) Let P be a normal program. Let I be a partial interpretation. The greatest unfounded set of P w.r.t. I is the union of all unfounded sets of P w.r.t. I.

Definition 1.3.3. (Immediate Consequences) Let P be a normal program. Let I be a partial interpretation. The three operators T_P, U_P, and W_P are defined as follows:

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

$U_P(I) :=$ the greatest unfounded set of P w.r.t. I

$W_P(I) := T_P(I) \cup \neg U_P(I)$

Lemma 1.3.1. ([vGRS91]) *T_P, U_P, and W_P are monotonic operators.*

Theorem 1.3.2. ([vGRS91]) *Let P be a normal program. For every countable ordinal α, $W_P \uparrow \alpha$ is a partial model of P.*

Definition 1.3.4. (Partial and Total Well-Founded Model) Let P be a program. The *well-founded (partial) model* of P, W_P^* , is the least fixpoint of W_P. If W_P^* is a total interpretation, it is called the *well-founded total model* of P.

Definition 1.3.5. (Semantics) A *semantics* is a mapping $\mathcal{S}$, which assigns to every logic program P a set $\mathcal{S}(P)$ of (partial) models of P such that $\mathcal{S}$ is "instantiation invariant", i.e. $\mathcal{S}(P) = \mathcal{S}(ground(P))$.

Now we can define the *well-founded semantics*.

Definition 1.3.6. (Well-Founded Semantics) The *Well-Founded Semantics* assigns to every logic program P the well-founded partial model W_P^* of P:

$WFS(P) := W_P^*$

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

We illustrate computation of a well-founded model of a program by the following two examples.

Example 1.3.1. ***(Detection of Odd Cycles)***

$$
\begin{array}{lll}
P_1: & g. & W_P \uparrow 1 = T_P(\emptyset) \cup \neg U_P(\emptyset) \\
 & a \leftarrow b, c. & = \{g\} \cup \neg\{k\} \\
 & b \leftarrow e, \neg k. & W_P \uparrow 2 = \{g, \neg k\} = \\
 & c \leftarrow h. & = W_P \uparrow 1 = lfp(W_P) \\
 & d \leftarrow \neg a. & \\
 & e \leftarrow g, d. & \\
 & h \leftarrow d. &
\end{array}
$$

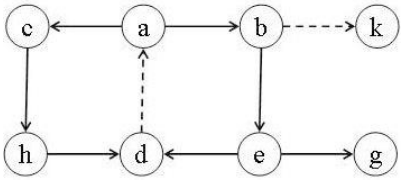

Figure 1.1: Dependency graph of P_1.

The atom g gets into the model of P trivially as a fact, via T_P, while $\neg k$ is added to the model by the first unfoundedness condition—there is no rule that can be used to establish truth of k, in other words, no rule in P defines k. But it is more interesting to observe which atoms are left *unknown* in the model of P. These are $\{a, b, c, d, e, h\}$. It shows us how the well-founded semantics deals with *odd cycles* (cycles in the dependency graph with an odd number of negative edges, see Figure 1.1 and Section 2.2 for the definition). The well-founded semantics leaves odd-cyclic atoms "out" of the model, while the same very situation leads to an inconsistency in another nonmonotonic logic programming semantics, the stable model semantics (this fact is illustrated in [Bon01b]).

Example 1.3.2. ***(Detection of Cycles)***

$$
\begin{array}{lll}
P_2: & g. & W_P \uparrow 1 = T_P(\emptyset) \cup \neg U_P(\emptyset) \\
 & a \leftarrow b, c. & \quad = \{g\} \cup \neg\{k, a, d, h, e, c, b\} \\
 & b \leftarrow e. & W_P \uparrow 2 = W_P \uparrow 1 = lfp(W_P) \\
 & c \leftarrow h. & \\
 & d \leftarrow \neg a, e. & \\
 & d \leftarrow a, \neg k. & \\
 & e \leftarrow g, d. & \\
 & h \leftarrow d. &
\end{array}
$$

The above program slightly differs from that in the previous example. We have eliminated $\neg k$ from the second rule, added e into the body of the fourth rule and added one more rule: $d \leftarrow a, \neg k$. Our manipulations are aimed at elimination of odd-cycles. As a consequence, the atoms $\{a, b, c, d, e, h\}$ are now evaluated to *false* because they form an unfounded set with respect to $\emptyset$ due to the second unfoundedness condition. We can see that among these atoms none can be the first to be proven, none can be the first given the value *true*. Declaring any of a, b, c, d, e, h *false* does not create a proof that any other element of the set is *true*. And if all are declared false at once, we do not have an inconsistency. These atoms are not odd-cyclic anymore, but they

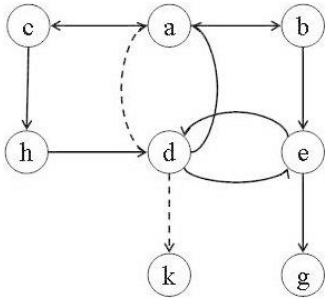

Figure 1.2: Dependency graph of P_2.

still form a cycle (see Figure 1.2). The well-founded semantics detects cycles, and this is one of the reasons of its undecidability. We discuss it further on in Chapter 2.

Chapter 2

Properties of the Well-Founded Semantics

2.1 Decidability Issues

Although logic programming semantics are usually thought of as models of human inference, they can also be thought of as methods for defining relations on databases. In this respect the primary concern is in what relations a semantics can define, how expressive it is. The input databases are arbitrary (not necessarily recursive) relations on the infinite Herbrand universe of the program. For normal logic programming, Schlipf [Sch95] has shown that for the well-founded semantics over infinite Herbrand universes, the problem of deciding whether a tuple of elements of the Herbrand universe is a correct answer for a query is highly non-recursive.

Theorem 2.1.1. ([Sch95]) *Logic programming with negation under the well-founded semantics is* Π^1_1*-complete.*

The class Π^1_1 belongs to the *analytical hierarchy* in a relational form and contains those relations that are definable by a second-order formula $\Phi(X) = \forall\mathcal{P}\phi(\mathcal{P}; X)$, where $\mathcal{P}$ is a finite set of predicate variables and ϕ is a first-order formula with free variables X.

While the polynomial hierarchy is an attempt at characterizing "how polynomially

uncomputable" is a decidable function, the arithmetic and analytical hierarchies—which are precursors of the polynomial hierarchy—are characterizations of "how undecidable" is a function. For more details about this class in the context of logic programming, see, for example, Schlipf [Sch95] and Eiter and Gottlob [EG97]. To illustrate Π^1_1-completeness, we simply mention that such problems are of much higher complexity than the well-known halting problem of Turing machines.

The result given above is often referred to, but it was never closely considered and analyzed. What is the reason of this "high non-recursiveness"? Originally, the well-founded semantics was invented to "improve" the existing semantics, such as the Clark's completion-semantics [Cla87], in the way that it delivers more intuitive answers for programs with negation. Take a look at the following example.

Example 2.1.1. ***(Loop Detection)*[Dix95a]**

$P:\ p \leftarrow p$ $q \leftarrow \neg p$	$P':\ p \leftarrow p$ $q \leftarrow \neg p$ $r \leftarrow \neg r$
$comp(P):\ p \leftrightarrow p$ $q \leftrightarrow \neg p$	$comp(P'):\ p \leftrightarrow p$ $q \leftrightarrow \neg p$ $r \leftrightarrow \neg r$
$?-q:$ *No* (completion). *Yes* (NMR-intuition).	$?-p:$ *Yes* (completion). *No* (NMR-intuition).

For both programs, the answers of the completion-semantics do not match the non-monotonic intuition. In the case of P we expect q to be derivable, since we expect $\neg p$ to be derivable: the only possibility to derive p is the rule $p \leftarrow p$ which will never succeed. But q is not in the deductive closure of $q \leftrightarrow \neg p$.

In the case of P' we expect p not to be derivable, for the same reason: the only possibility to derive p is the rule $p \leftarrow p$. But deductive closure of $r \leftrightarrow \neg r$ is the set of all formulae, the completion of P' is inconsistent, therefore p is among logical consequences of the completion of P'. Note that the answers of the completion semantics agree with the mechanism of SLDNF: $p \leftarrow p$ represents a loop.

This example motivates the need for semantics that detects loops. As we have seen in Section 1.3, the well-founded semantics detects loops. It evaluates all the atoms belonging to an odd negative cycle (a cycle with an odd number of negative edges) in the atom-call dependency graph as *undefined*. All other loop-atoms also constitute

potential candidates for unfounded sets. In general, loop detection is undecidable because the halting problem of Turing machines[1] reduces to it. This is the reason why the well-founded semantics is not even recursively enumerable.

Despite the importance of the well-founded semantics, the question of its not-semi-decidability has not attracted significant attention. The problems of semantics decidability lie on the meta-level, more in the scope of philosophical logic, nevertheless we believe they should not be put aside by those interested in optimizing the behavior of logic programs. Sometimes even rather superficial abstract analysis might lead to the definition of classes of programs that demonstrate decidability of query evaluation, e.g. as shown in [Bon01b]. Keeping the above observation in mind, in this chapter we analyze the well-founded semantics on the meta-level. We perform such an analysis by means of abstract properties of *relevance*, *level-mapping characterization*, *stratified relevance* and, finally, a new property called *maximal path property*, which is defined in Section 2.6.

2.2 Relevance

An abstract property of *relevance* is introduced in this section. It is one of the so-called *weak properties* formulated by Dix in [Dix95b]. These properties were inspired by irregular behavior of some of the logic programming semantics, such as, for example, the stable model semantics [GL88]. Every weak principle characterizes one of the aspects of semantics behavior. We give a brief description of *relevance* together with its formal definition.

We start with some basic definitions.

Definition 2.2.1. (Predicate-Call Dependency Graph) Let P be a normal logic program and p, q its predicates. The *predicate-call dependency graph* $\mathcal{G}_P^{pred}$ is a directed graph whose nodes are the predicates from P.

(i) We say that p *refers positively to* q (resp. *negatively*) iff there is a rule in P that uses p in its head and q in a positive (resp. negative) body literal. We denote *positively (negatively) refers to* as $>_+$ ($>_-$).

[1] See Section 3.1 for the definition of the halting problem.

(ii) There is a *positive edge* (resp. *negative*) from p to q iff $p >_+ q$ (resp. $p >_- q$).

(iii) We say that p *depends on* q iff there is a path in $\mathcal{G}_P^{pred}$ from p to q. We denote *depends on* as $\triangleright$.

We say that q is *lower* than p, or, equivalently, that q is a *dependency successor* of p, if there is a path from p to q, but no path from q to p. We say that p and q are in the same *strongly connected component (SCC)* if there is a path from p to q and from q to p.

Definition 2.2.2. (Atom-Call Dependency Graph) Let $ground(P)$ be a ground instantiation of normal logic program P and A, B its atoms. The *atom-call dependency graph* $\mathcal{G}_P^{atom}$ is a directed graph whose nodes are the elements of the Herbrand base B_P.

(i) We say that A *refers positively to* B (resp. *negatively*) iff there is a rule in $ground(P)$ that uses A in its head and B in a positive (resp. negative) body literal. We denote *positively (negatively) refers to* as $>_+$ ($>_-$).

(ii) There is a *positive edge* (resp. *negative*) from A to B iff $A >_+ B$ (resp. $A >_- B$).

(iii) We say that A *depends on* B iff there is a path in $\mathcal{G}_P^{atom}$ from A to B. We denote *depends on* as $\triangleright$.

We also say:

- A *depends positively on* B (resp. *negatively*) if there is a path in $\mathcal{G}_P^{atom}$ from A to B containing only positive edges (resp. at least one negative edge).
- A *depends evenly on* B (resp. *oddly*) if there is a path in $\mathcal{G}_P^{atom}$ from A to B containing an even number of negative edges (resp. an odd number).

One atom can be *lower* than another (a *dependency successor*) or in the same *SCC* of an atom-call graph in an analogous way to predicates in a predicate-call graph.

By definition, *depends on* is a transitive closure of the *refers to* relation. We shall often omit positive or negative character of the dependency if it is irrelevant. In the following, it will sometimes be necessary to use the notion of a *reverse dependency relation* on predicates (resp. atoms): if $p \triangleright q$ (resp. $A \triangleright B$), then $q \triangleright^{-1} p$ (resp. $B \triangleright^{-1} A$), where $\triangleright^{-1}$ denotes a reverse dependency relation.

Definition 2.2.3. (Dependency Path) The *predicate dependency path* is a path in $\mathcal{G}_P^{pred}$ which is denoted by a list of its nodes as follows: $\pi = \{p_1, p_2, ..., p_n\}$.

The *atom dependency path* is a path in $\mathcal{G}_P^{atom}$ and is denoted by a list of nodes in an analogous way: $\pi = \{A_1, A_2, ..., A_n\}$. We shall denote a dependency path starting with the node representing an atom A by π_A.

Cardinality of a dependency path π is denoted by $|\pi|$ and is defined as the number of distinct members of the list representing the path.

Please note that a *cyclic dependency path* representing a cycle in the dependency graph is denoted by a list with the same node as the first and the last member. It is important to count only distinct members for cardinality so that in cyclic paths the starting node is not counted twice. For a given predicate dependency path $\pi = \{p_1, p_2, ..., p_n\}$ its reverse version is $\pi^{-1} = \{p_n, p_{n-1}, ..., p_1\}$. The same holds for an atom dependency path.

Definition 2.2.4. (Dependency Relation) The *predicate dependency relation* $D^{pred}(P) \subseteq preds(P) \times preds(P)$ of a logic program P is defined as follows:

$$D^{pred}(P) := \{\langle p, q\rangle \,|p \triangleright q\}$$

All predicates on which p depends are said to belong to the dependency relation of p denoted by $D(p)$, i.e. $D(p) := \{q|p \triangleright q\}$.

The *atom dependency relation* $D^{atom}(P) \subseteq B_P \times B_P$ of a ground logic program $ground(P)$ is defined in an analogous way.

All atoms on which some ground atom $A \in B_P$ depends are analogously said to form the dependency relation of A, $D(A) := \{B|A \triangleright B\}$.

We recall that by a semantics we mean a mapping $\mathcal{S}$ from the class of all programs into the powerset of the set of all 3-valued Herbrand interpretations. $\mathcal{S}$ assigns to every program P a set of 3-valued Herbrand models of P.

The property of *relevance* states that a goal in P can be answered using only the atoms from $ground(P)$ that are in a syntactic dependency with it. Let us call this set of atoms a *relevant universe* of the goal. In fact, it is a union of dependency relations of goal atoms. We use this notion to define a *relevant subprogram*—a program that contains only the rules defining the atoms from the relevant universe. This is a so-called *module*, or *stratum*—a part of the program solely responsible for the evaluation of the query defined by it.

Definition 2.2.5. (Relevant Universe [Bon01b]) The *relevant universe* for a program P and a ground query Q, denoted by $U_{rel}(P, Q)$, is the set of all ground atoms B such that some atom from Q depends on B. In symbols:

$$U_{rel}(P, Q) := \{B| \text{ some } A \in Q, A \triangleright B\}.$$

Definition 2.2.6. (Relevant Subprogram [Bon01b]) The *relevant subprogram* for a ground query Q (w.r.t. a normal program P), denoted by P_Q, is the set of all rules in $ground(P)$ whose head belongs to $U_{rel}(P, Q)$:

$$P_Q := \{R|R \in ground(P) \text{ and } head(R) \in U_{rel}(P, Q)\}.$$

In the following, we will denote a relevant subprogram also by P_{rel} if the name of the query is not given or not important.

Definition 2.2.7. (Relevance [Dix95b]) *Relevance* states that for all literals L, P entails L under semantics $\mathcal{S}$ iff P_L entails L under $\mathcal{S}$: $\mathcal{S}(P)(L) = \mathcal{S}(P_L)(L)$, where P_L is a relevant subprogram of P with respect to L.

In other words, *relevance* states that for all ground formulae Q and all normal logic programs P, P entails Q under semantics $\mathcal{S}$ iff P_Q entails Q under $\mathcal{S}$. Such a property is quite natural and one can expect any logic programming semantics to satisfy it. However, not all semantics do. For example, the stable model semantics does not satisfy *relevance*. Negative odd-cycles (cycles in the dependency graph with an odd number of negative edges) affect the meaning of the query even if they do not have any syntactic dependency with it. A rule involved in an odd-cycle can never be ignored because it may cause the program to be inconsistent (or without stable model).

As noted by Dix [Dix95b], the well-founded semantics is among very few logic programming semantics that satisfy *relevance*.

We illustrate *relevance* by the following example.

Example 2.2.1. ***(Relevance)***

$$\begin{array}{lll} P: & d. & Q: \ ?-c. \\ & a \leftarrow b, c. & \\ & b \leftarrow e, \neg k. & P_Q: \ d. \\ & c \leftarrow h. & \quad c \leftarrow h. \\ & e \leftarrow g, d. & \quad h \leftarrow d. \\ & h \leftarrow d. & \end{array}$$

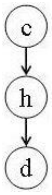

Figure 2.1: Dependency graph of P_Q.

We can see here that the relevant subprogram of the program P with respect to the query Q contains all rules that could ever contribute to Q's derivation or its non-derivability. We can also observe (see Figure 2.1) that all the atoms in P_Q are in the dependency relation of the atom c which constitutes the query. To sum up, *relevance* states that only atoms belonging to the dependency relation of query atoms affect their meaning in P.

The principle of *relevance* can be very useful. First, it allows to minimize instantiation costs because relevant subprogram suffices for query answering, the whole $ground(P)$ is not needed. Second, when program instantiation $ground(P)$ is infinite (in the presence of function symbols), one can still have a finite relevant subprogram for the given query so that its value can always be computed. With these considerations, we have defined classes of normal logic programs with semi-decidable query evaluation under the well-founded semantics (see [CHH07]). Moreover, we use *relevance* later on in Chapter 3 in the definition of atoms with decidable query evaluation.

2.3 The Level Mapping Characterization

In this section we introduce the level mapping characterization of the well-founded semantics. Hitzler and Wendt [HW02, HW05] have proposed a methodology that allowed to obtain uniform characterizations of different logic programming semantics. This uniform approach is inspired by the fact that most semantics introduced so far are defined and characterized using a wide range of various techniques, including semantic operators, program transformations, restriction to suitable subprograms

etc. In this rich but heterogeneous context establishing relationships between the semantics and comparing the respective models appear to be rather intricate tasks. Characterizations by Hitzler and Wendt allow immediate comparison between the semantics. Moreover, they enable to make some new and interesting observations, such as the fact that the well-founded semantics can formally be understood as the Fitting semantics augmented with a form of stratification.

The main tool employed for characterization is the notion of *level mapping*. Level mappings are mappings from Herbrand bases to ordinals, i.e. they induce orderings on the set of all ground atoms while disallowing infinite descending chains. Level mappings are used here as a technical tool for capturing dependencies between atoms in a program.

Definition 2.3.1. (I-partial Level Mapping [HW02, HW05]) For an interpretation I and a program P, an *I-partial level mapping* for P is a partial mapping $l : B_P \to \alpha$ with domain $dom(l) = \{A | A \in I \text{ or } \neg A \in I\}$, where α is some (countable) ordinal. A *(total) level mapping* is a total mapping $l : B_P \to \alpha$ for some (countable) ordinal α.

We extend every level mapping to literals by setting $l(A) = l(\neg A)$ for all $A \in dom(l)$.

Definition 2.3.2. (WF-properties [HW02, HW05]) Let P be a normal logic program, I be a model for P, and let l be an I-partial level mapping for P. We say that P *satisfies* (WF) *with respect to I and l* if each $A \in dom(l)$ satisfies one of the following conditions.

(WFi) $A \in I$ and there is a clause $A \leftarrow L_1, .., L_n$ in $ground(P)$ such that $L_i \in I$ and $l(A) > l(L_i)$ for all i.

(WFii) $\neg A \in I$ and for each clause $A \leftarrow A_1, .., A_n, \neg B_1, .., \neg B_m$ in $ground(P)$ one (at least) of the following conditions holds:

- **(WFiia)** There exists i with $\neg A_i \in I$ and $l(A) \geq l(A_i)$.
- **(WFiib)** There exists j with $B_j \in I$ and $l(A) > l(B_j)$.

If $A \in dom(l)$ satisfies (WFi), then we say that A *satisfies* (WFi) *with respect to I and l*, and similarly if $A \in dom(l)$ satisfies (WFii). Atoms (literals) satisfying one of the WF-conditions with respect to I and l are called *kernel atoms (literals)*.

We shall denote sets of kernel atoms (literals) as $\mathcal{K}$. In the following, the notation $R = A \leftarrow \mathcal{K}, \mathcal{B}$ means that $body(R) = \mathcal{K} \cup \mathcal{B}$, where $\mathcal{K}$ is the set of kernel atoms (literals) belonging to the rule R, and $\mathcal{B}$ is the set of body literals of R that are not kernel.

Theorem 2.3.1. ([HW02, HW05]) *Let P be a normal logic program with well-founded model M. Then, in the knowledge ordering, M is the greatest model amongst all models I for which there exists an I-partial level mapping l for P such that P satisfies WF-properties with respect to I and l.*

In the following, we shall often refer to the property of the well-founded semantics stated in Theorem 2.3.1 as the *level mapping characterization* of the well-founded semantics. We illustrate it by the following example.

Example 2.3.1. ***(Level Mapping Characterization)***

$$
\begin{array}{lllll}
P: & r. & W_P \uparrow 1 = & \{r\} \cup \neg\{k, s\} & \Lambda_0 = \{r, \neg k, \neg s\} \\
 & p \leftarrow q, r. & W_P \uparrow 2 = & \{r, q\} \cup \neg\{k, s\} & \Lambda_1 = \{q\} \\
 & q \leftarrow r. & W_P \uparrow 3 = & \{r, q, p\} \cup \neg\{k, s\} & \Lambda_2 = \{p\} \\
 & s \leftarrow q, k. & W_P \uparrow 4 = & W_P \uparrow 3 = lfp(W_P) &
\end{array}
$$

As the above example shows, atoms that get the value $undefined$ in the well-founded model, are not ordered into levels by the level mapping l_P based on the W_P-operator computations. The domain of the model-driven l_P includes only atoms whose literals are computed by the W_P-operator, in other words, only atoms that are evaluated to *true* or *false*.

2.4 Stratified Relevance

Given any semantics $\mathcal{S}$, it is reasonable to expect that a truth value of a ground query Q only depends on the relevant subprogram P_Q for Q with respect to P. As we have seen in Section 2.2, this idea was formalized by Dix [Dix95b] in the property of *relevance*. The property called *stratified relevance* [CHH07] is based on level mappings and Theorem 2.3.1 from Hitzler, Wendt ([HW02, HW05]). It absorbs the advantages of *stratification* and *relevance*, and appears to be a good instrument in achieving our goal—to define a class of logic programs with decidable query evaluation.

Definition 2.4.1. (Model-Driven Level Mapping) For a normal program P and its well-founded model M, define an M-partial level mapping l_P as follows: $l_P(A) = \alpha$, where α is the least ordinal such that A is not undefined in $W_P \uparrow (\alpha + 1)$. We call this level mapping a *model-driven level mapping*.

Please note that each program has a unique model-driven level mapping because each program has a unique well-founded model. Let $l^{-1}(\alpha) = \Lambda_\alpha \subseteq M$ be a set of ground literals of level α, $W_P \uparrow (\alpha + 1) \backslash W_P \uparrow (\alpha) = \Lambda_\alpha$. In the following, when it is clear from the context, we call a set of literals of some level simply *a level*. If we evaluate a query Q, we start from some set Λ_α such that $Q \in \Lambda_\alpha$ (assume this is an atomic ground query for the moment). According to the WF-properties that a model M enjoys by Theorem 2.3.1, every evaluation step either "goes down" to the previous level, or "stays" at the same level. The sequence of sets Λ_α is well-founded—the last set is $\Lambda_0 = W_P \uparrow 1 \backslash W_P \uparrow 0 = W_P \uparrow 1 \backslash \emptyset = W_P \uparrow 1$. This observation lets us limit the new relevant universe $U^*_{rel}(P, Q)$ (and thus the relevant subprogram P^*_Q) to those levels that are less than or equal to the level of query atoms w.r.t the level mapping l_P.

Definition 2.4.2. (Stratified Relevant Universe) The *stratified relevant universe* for P and query Q, denoted by $U^*_{rel}(P, Q)$, is the set of all ground atoms B such that some atom $A \in Q$ depends on B and $l_P(A) \geq l_P(B)$, where l_P is an I-partial level mapping for P as defined above. In symbols:

$$U^*_{rel}(P, Q) = \{B | \text{ some } A \in Q, A \triangleright B \text{ and } l_P(A) \geq l_P(B)\}.$$

Thus, the definition of $U^*_{rel}(P, Q)$ differs from that of $U_{rel}(P, Q)$. Due to stratification of the well-founded semantics, it can be shown that only the levels less than or equal to those of query atoms are relevant for the query evaluation.

In order to define a stratified relevant subprogram, we first need the following definition:

Definition 2.4.3. Let P be a logic program and Q a ground query. P'_Q is the set of all rules $R\sigma$ such that there exists a rule R in P and a substitution σ meeting the following conditions:

- The head of $R\sigma$ is in $U^*_{rel}(P, Q)$.

- At least one atom occurring in the body of $R\sigma$ is contained in $U^*_{rel}(P, Q)$.
- Let $A_1, ..., A_n$ be all atoms occurring in R, such that $A_i\sigma \in U^*_{rel}(P, Q)$ for all $1 \leq i \leq n$. Then the following must hold:
  - σ is the most general unifier for the unification problem $\{A_i = A_i\sigma | i = 1, ..., n\}$.
  - There does not exist an atom B occurring in R, which is distinct from all $A_i (i = 1, ..., n)$ such that there is a substitution θ with $B\sigma\theta \in U^*_{rel}(P, Q)$.

Definition 2.4.4. (Stratified Relevant Subprogram) Let P be a logic program and Q a ground query. The *stratified relevant subprogram* for P and Q w.r.t. an I-partial level mapping l_P, denoted by P^*_Q, is the set of all rules R' defined as follows:

- for any rule R in P'_Q, let R' be the rule which is obtained by removing all non-ground literals from R.

Note that by definition the head and at least one of the body literals of R' are never removed.

The definition of stratified relevant subprogram P^*_Q is more complicated than that of P_Q. The underlying intuition is that we use rules from $ground(P)$ where the head and all body atoms are contained in the stratified relevant universe, as they are those which contribute to the well-founded semantics in the sense of condition (WFi) in Definition 2.3.2. In order to accommodate condition (WFii) from Definition 2.3.2 it suffices if one *witness of unusability* [2] remains in the body of the rule, which is the rationale behind the remaining definition of stratified relevant subprogram.

Definition 2.4.5. (**Stratified Relevance**) *Stratified relevance* states that for all literals L, P entails L under semantics $\mathcal{S}$ iff P^*_L entails L under $\mathcal{S}$: $\mathcal{S}(P)(L) = \mathcal{S}(P^*_L)(L)$, where P^*_L is a stratified relevant subprogram of P with respect to L.

In other words, *stratified relevance* states that for all ground formulae Q and all normal logic programs P, P entails Q under semantics $\mathcal{S}$ iff P^*_Q entails Q under $\mathcal{S}$.

Proposition 2.4.1. [CHH07] *The well-founded semantics satisfies stratified relevance.*

[2] these are literals satisfying one of the unfoundedness conditions[vGRS91].

Thus, if a literal L is in the well-founded model, it is brought into this model by literals it refers to (there is always a syntactic dependency) and these literals come from the same or the previous level of the level mapping l_P. The fact that there is always a syntactic dependency is not new—it is expressed in *relevance*. An important point for us is the level of literals responsible for L's derivation. This proposition shows us that a stratified relevant subprogram P_L^* by its definition contains all the necessary information for L's derivation or non-derivability.

We illustrate the property of *stratified relevance* by the following example.

Example 2.4.1. ***(Stratified Relevance)***

$$
\begin{aligned}
P: \quad & p(f(X)) \leftarrow p(X), q(X).\\
& q(a) \leftarrow s(f(X)), r(X).\\
& r(a) \leftarrow r(a).\\
& u(X) \leftarrow p(X).
\end{aligned}
$$

$$
Q: \quad ? - p(f(a)).
$$

$$
\begin{aligned}
U_{rel}(P,Q): \quad & \{p(f(a)), p(a),\\
& q(a), s(f^m(a)), r(f^n(a)) | m \geq 1, n \geq 0\}
\end{aligned}
$$

$$
\begin{aligned}
P_Q: \quad & \{p(f(a)) \leftarrow p(a), q(a);\\
& q(a) \leftarrow s(f^m(a)), r(f^n(a)) | m \geq 1, n \geq 0\}
\end{aligned}
$$

Both $U_{rel}(P,Q)$ and P_Q are infinite. Q is not finitely recursive because of the second rule, where $q(a)$ depends on infinite sequences of atoms of the form $s(f^m(X)), m > 0$ and $r(f^n(X)), n \geq 0$. However, given a level mapping characterization l_P of the well-founded model of P, that maps $l_P(p(f(a))) = 3$ and $l_P(s(f(a))) = 0, l_P(r(a)) = 1$, and at the same time the set of atoms of the form $\{s(f^m(a)), r(f^n(a)) | m > 1, n > 0\}$ is not in the domain $dom(l_P)$, we observe that it leaves all the rules $q(a) \leftarrow s(f^m(a)), r(f^n(a))$ with $m > 1, n > 0$ out of our new stratified relevant universe and relevant subprogram, P_Q^*:

$$
\begin{aligned}
P_Q^*: \quad & p(f(a)) \leftarrow p(a), q(a).\\
& q(a) \leftarrow s(f(a)), r(a).
\end{aligned}
$$

Thus *stratified relevance* helps to keep the fragment of the program that is relevant for query evaluation finite.

2.5 Approximation of Level Mappings

We have seen in the previous section that the well-founded semantics enjoys the property of *stratified relevance.* This property tells us that we do not need to include all the relevant rules into P_{rel} but only those that define atoms coming from the levels that are lower than the query atom. But the definition of P^*_{rel} is not purely syntactic anymore. It becomes proof-theoretic in a sense that a level mapping characterization of the well-founded model comes into play. The level of an atom in this stratification is based on the number of iterations it takes the W_P-operator to determine the truth value of this atom. As such, this stratification exactly follows the computation of the well-founded model and is, therefore, the most precise possible stratification of a program under the well-founded semantics. But we should admit that there exist no syntactic criteria which can be used to determine whether a certain stratification is the level mapping stratification of the program. The only way of deciding this is by actually constructing the well-founded model of the program. Therefore this concept cannot be used to perform the kind of static, upfront analysis of decidable fragments which is our goal.

If we take a closer look at the level mapping stratification, we see that it is tightly bounded to the dependency relation of atoms in the program. If we reformulate the property of *relevance*, we can state that the truth value of an atom depends solely on the atoms that are *lower* in the dependency relation. This has led us to the idea to approximate the level mapping characterization of the well-founded model. The approximate stratification is based on the dependency relation and is therefore syntactically defined. This approximation is of the following character: one approximation stratum includes all the possible candidates that might get into the corresponding level of the M-partial level mapping, where M is the well-founded model of the program.

Definition 2.5.1. (**Stratification**) Let P be a logic program with Herbrand base B_P. A *stratification* of P is a partition $(S_i)_{i \in Ind}$ of B_P, such that the well-founded order $\geq$ on the index set Ind agrees with the dependency relation $\triangleright$ of P, i.e. if $A \triangleright B, A \in S_i$ and $B \in S_j$, then $i \geq j$. For each $i \in Ind$, the program P_i consists of

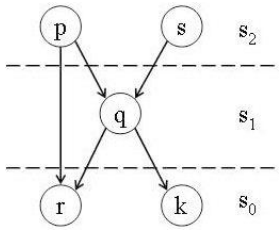

Figure 2.2: Dependency graph of P and stratification S_i.

all clauses which have an atom from S_i in their head.

We illustrate the above definition by the following example.

Example 2.5.1. ***(Dependency-Driven Stratification)***

$P:$	$r.$	$W_P \uparrow 1 =$	$T_P(\emptyset) \cup \neg U_P(\emptyset)$
	$p \leftarrow q, r.$	$=$	$\{r\} \cup \neg\{k, s\}$
	$q \leftarrow r.$	$W_P \uparrow 2 =$	$\{r, q\} \cup \neg\{k, s\}$
	$s \leftarrow q, k.$	$W_P \uparrow 3 =$	$\{r, q, p\} \cup \neg\{k, s\}$
		$W_P \uparrow 4 =$	$W_P \uparrow 3 = lfp(W_P)$
	$S_0 = \{r, k\}$	$\Lambda_0 =$	$\{r, \neg k, \neg s\}$
	$S_1 = \{q\}$	$\Lambda_1 =$	$\{q\}$
	$S_2 = \{p, s\}$	$\Lambda_2 =$	$\{p\}$

The dependency graph of P together with stratification S_i are given in Figure 2.2. Model-driven level mapping l_P orders literals into levels Λ_i as given above. Dependency-driven stratification $(S_i)_{i \in Ind}$ approximates l_P in the following manner: atoms that get into stratum S_i cannot get higher in l_P than level Λ_i, but they can appear lower, in levels Λ_j for $0 \leq j \leq i$. And we observe it in the above example: r and k cannot get higher than Λ_0, q cannot get higher than Λ_1, and, finally, p and s do not get over Λ_2.

The above stratification orders atoms of B_P into strata that approximate the levels of a model-driven level mapping. As already noted, to obtain a model-driven level mapping, one needs to compute a well-founded model of the program, which is, in general, an undecidable task. On the contrary, a stratification can always be computed. We believe that such a dependency-driven approximation of a model-driven level mapping can be shown useful in a number of situations that arise in the

analysis of program and query decidability.

2.6 Maximal Path Property

In this section we formulate a new property for the well-founded semantics—we call it the *maximal path property*. The following observation is central in this section: the number of W_P-iterations in computation of a well-founded model of P is limited by the cardinality of the longest dependency path in $\mathcal{G}_P^{atom}$, as long as this path is of finite length. When $\mathcal{G}_P^{atom}$ has at least one infinite path, one can only conclude that there is no guarantee that the least fixed point of W_P is reached before ω. We use the convention that $\omega > n$ for all $n \in \mathbb{N}$.

We remind that, if we evaluate a query Q, we start from some set Λ_α such that $Q \in \Lambda_\alpha$ (assume this is an atomic ground query for the moment). According to the WF-properties that a model M enjoys by Theorem 2.3.1, every evaluation step either "goes down" to the previous level, or "stays" at the same level. The sequence of sets Λ_α is well-founded—the last set is $\Lambda_0 = W_P \uparrow 1 \backslash W_P \uparrow 0 = W_P \uparrow 1 \backslash \emptyset = W_P \uparrow 1$. We regard the well-founded model M of a program P as stratified and ordered into a (possibly infinite) number of levels Λ_α by a model-driven level mapping l_P, as defined in Section 2.4.

From *relevance* it follows that the truth value of an atom A in the well-founded model of P and in the well-founded model of P_A is the same. P_A contains by definition only atoms that are below A in the dependency relation. Therefore the truth value of A is defined solely by truth values of atoms below in the dependency relation and, consequently, below in all possible dependency paths starting from A (there might be an infinite number of them). Each atom in Λ_α has a truth value depending solely on truth values of atoms below in all dependency paths starting with this atom. In order to assign a truth value to an atom A in any level Λ_α of M, the model computation starts at some atom A_0 in Λ_0 and then "goes up" the levels along the reverse dependency path of A (in other words, along the dependency path of A in an opposite direction, from its "end", the node A_0, to the "beginning" with the starting

node being atom A). Let us call the dependency path that starts with an atom A and ends with some atom in the level Λ_0 the *evaluation path* for an atom A. We will show that each $A \in M$ has an evaluation path. If an atom A is a starting node of the longest dependency path in $\mathcal{G}_P^{atom}$, denoted by π^{max}, then its evaluation path can be at most π^{max}. In the extreme case, only one atom from π^{max} is evaluated in each level Λ_α and the atom A is evaluated in Λ_β, where β is the cardinality of π^{max}. This maximal situation gives us β as the maximal possible number of levels in M and, consequently, the upper limit for the number of iterations of W_P-operator needed to reach the least fixed point.

Definition 2.6.1. Given a logic program P, its atom-call dependency graph $\mathcal{G}_P^{atom}$, the *maximal path*, denoted by π^{max}, is defined as the dependency path in $\mathcal{G}_P^{atom}$ with the largest cardinality, that is for all dependency paths π in the graph $\mathcal{G}_P^{atom}$ it holds that $|\pi| < |\pi^{max}|$.

By π_A we denote a dependency path that starts with the node A.

We remind that *kernel atoms (literals)* are atoms (literals) satisfying one of the WF-conditions (see Section 2.3). Intuitively, kernel atoms are the atoms in the body of a rule that "cause" its head to get into the model, or, in other words, assign the truth value to the head atom. If the head is assigned the value *true*, then, by definition, it is caused by the truth of the whole body of the rule—this means that all body atoms are kernel. If the head is assigned the value *false*, it can be caused by falsity of a single body atom—this only atom is considered kernel while the rest of the body is not.

Lemma 2.6.1. *Given a logic program P and its well-founded model M, for each $A \in M$ with $A \in \Lambda_\alpha, \alpha > 0$, there exists at least one kernel atom B such that $B \in \Lambda_\alpha$ or $B \in \Lambda_{\alpha-1}$ and $B \in \pi_A$.*

Proof. By Theorem 2.3.1 [HW02, HW05], M is the greatest model in the knowledge ordering, for which there exists an M-partial level mapping l for P such that P satisfies (WF) with respect to M and l. One such level mapping is the model-driven level mapping l_P as defined in Section 2.4. The proof is based on the evaluation of L under the well-founded semantics. We have to consider two cases: $L = A$ is a positive literal and $L = \neg A$ is a negative literal. Let $A \in dom(l_P)$, suppose $l_P(A) = \alpha$.

Case i. $L = A$. By (WFi) there is at least one rule $R = A \leftarrow L_1, .., L_n$ in $ground(P)$ such that $L_i \in M$ and $l_P(A) > l_P(L_i)$ for all i. By definition of W_P-operator, $A \in T_P(W_P \uparrow \alpha)$ and all $L_i \in body(R)$ are true in $W_P \uparrow \alpha$. At least one of L_i must have been added to M in $W_P \uparrow \alpha$ and not earlier, otherwise A would have been added to M also earlier. This means that at least one literal from $body(R)$ must belong to $\Lambda_{\alpha-1}$. We denote this literal by $L_{\alpha-1}$ and its atom by $A_{\alpha-1}$. We see that A refers to $A_{\alpha-1}$ or, in other words, $A \triangleright A_{\alpha-1}$. Thus there is at least one atom in $\Lambda_{\alpha-1}$ such that $A \triangleright A_{\alpha-1}$ and, consequently, $A_{\alpha-1} \in \pi_A$.

Case ii. $\neg A \in M$. Then $A \in U_P(W_P \uparrow \alpha)$ and by (WFii) for all rules $A \leftarrow A_1, .., A_n, \neg B_1, .., \neg B_m$ one of the following holds:

(a) there is i with $\neg A_i \in M$ and $l_P(A) \geq l_P(A_i)$. This means $A \triangleright A_i$, consequently $A_i \in \pi_A$. A_i is a kernel atom for A. By definition of W_P, $A \in U_P(W_P \uparrow \alpha)$ and, by definition of U_P, A_i is either also in $U_P(W_P \uparrow \alpha)$ or in $U_P \uparrow \alpha$. A_i must have been added to M not earlier than in $W_P \uparrow \alpha$, otherwise A would have been added to M also earlier. This means that $A_i \in \Lambda_\alpha$ or $A_i \in \Lambda_{\alpha-1}$.

(b) there is j with $B_j \in M$ and $l_P(A) > l_P(B_j)$. Again $A \triangleright B_j$ and $\neg A$ is brought into M by B_j via U_P. B_j is a kernel atom for A and $B_j \in \pi_A$. By definition of W_P, $A \in U_P(W_P \uparrow \alpha)$ and $B_j \in body(R)$ is in $U_P \uparrow \alpha$. B_j must have been added to M in $W_P \uparrow \alpha$ and not earlier, otherwise A would have been added to M also earlier. This means that $B_j \in \Lambda_{\alpha-1}$.

To summarize all the cases, we can say that the following property holds: given a program P, a well-founded model M and some literal $L \in M$ with $atom(L) = A$ and $l_P(A) = \alpha$, there exists an atom B such that $B \in \Lambda_\alpha$ or $B \in \Lambda_{\alpha-1}$ and $B \in \pi_A$. □

Thus, all atoms that are not in the bottom level Λ_0 must have at least one dependency successor either in the same level or in the previous one. The only level allowed to contain atoms without dependency successors is Λ_0 which corresponds to the definition of W_P-operator.

In the following, when it is not important whether a positive or negative literal of an atom A occurs in M or in some level Λ_α, we say simply $A \in M$ or $A \in \Lambda_\alpha$ instead of $L \in M$ and $atom(L) = A$.

The following lemma ensures the existence of an evaluation path for every $A \in M$.

Lemma 2.6.2. *Given a logic program P and its well-founded model M, for each $A \in M$ with $A \in \Lambda_\alpha, \alpha > 0$ there exists at least one dependency path π_A such that $A_0 \in \pi_A$ and $A_0 \in \Lambda_0$.*

Proof. Assume that all dependency paths starting at A have no atoms in the bottom level Λ_0, i.e. there is no π_A such that some $A_0 \in \pi_A$ and $A_0 \in \Lambda_0$. Then all π_A have their last member, an atom without any dependency successors, in one of the levels Λ_α, $\alpha > 0$. But this contradicts the definition of W_P-operator, according to which only $W_P \uparrow 1 = W_P(\emptyset)$ can contain atoms without dependency successors, in other words, atoms that are *true* in the empty set $\emptyset$—facts, or atoms that are *false* in the empty set $\emptyset$—atoms not occurring in the head of any rule. □

The following lemma states that each evaluation path starting in atom $A \in \Lambda_\alpha$ contains at least one atom per level and "goes down" from the level Λ_α to the level Λ_0.

Lemma 2.6.3. *Given a logic program P and its well-founded model M, for each $A \in M$ there exists at least one dependency path π_A such that $A_\beta \in \pi_A$ and $A_\beta \in \Lambda_\beta$ for all ordinals $\beta \leq l_P(A)$.*

Proof. Lemma 2.6.2 states that every $A \in M$ has an evaluation path π_A that goes down to the bottom level Λ_0. From this fact and from Lemma 2.6.1 it follows that π_A has at least one atom A_β in each Λ_β for all ordinals $\beta \leq \alpha$. □

The main result of this section states the *maximal path property*.

Proposition 2.6.4. *Given a normal logic program P, its atom dependency graph $\mathcal{G}_P^{atom}$ with $|\pi^{max}| = \beta$ and the well-founded model M of P with $M = lfp(W_P) = W_P \uparrow \alpha$, the following holds:*

$$\alpha \leq \beta,$$

where α and β are finite ordinals.

Proof. Assume there is $A \in M$ such that $A \in \Lambda_\beta$ and $M = lfp(W_P) = W_P \uparrow (\beta + 1)$. The last member of A's evaluation path π_A must belong to Λ_0 by Lemma 2.6.2. Lemma 2.6.3 states that at least one atom from π_A must belong to each level $\Lambda_\gamma, \gamma \leq l_P(A)$. But $l_P(A) = \beta$. This leaves us with at least $\beta + 1$ atoms in the evaluation path π_A. And we know that the maximal dependency path in $\mathcal{G}_P^{atom}$ contains only β atoms, a contradiction. □

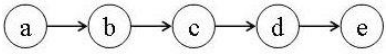

Figure 2.3: Dependency graph of P.

Thus we have shown that the number of W_P-iterations in computation of a well-founded model of P is limited by the cardinality of the longest dependency path in $\mathcal{G}_P^{atom}$, as long as this path is of finite length.

We illustrate the *maximal path property* by the following example.

Example 2.6.1. ***(Maximal Path Property)***

$$
\begin{array}{lll}
P: & e. & W_P \uparrow 1 = \{e\} \cup \{\neg g\} \\
 & a \leftarrow b. & W_P \uparrow 2 = \{e, d\} \cup \{\neg g\} \\
 & b \leftarrow c, \neg g. & W_P \uparrow 3 = \{e, d, c\} \cup \{\neg g\} \\
 & c \leftarrow d. & W_P \uparrow 4 = \{e, d, c, b\} \cup \{\neg g\} \\
 & d \leftarrow e. & W_P \uparrow 5 = \{e, d, c, b, a\} \cup \{\neg g\} \\
 & & W_P \uparrow 6 = W_P \uparrow 5 = lfp(W_P)
\end{array}
$$

As we can see on the dependency graph of P (see Figure 2.3), $\pi_{max} = \{a, b, c, d, e\}$, thus $|\pi_{max}| = 5$ and $\beta = 5$. And we also have $W_P \uparrow 5 = lfp(W_P)$, therefore $\alpha = 5$. We can see here that $\alpha \leq \beta$ holds.

The *maximal path property* gives us an upper limit for the number of W_P-iterations. This is a theoretical observation, but it will be of practical use in Chapter 3, where a decidable subset of the well-founded semantics is defined on the basis of this property.

Chapter 3

Atoms with Decidable Query Evaluation

In this chapter we propose a new setting for analysis of decidability of logic programs. In fact, instead of analyzing decidability of programs, we propose to analyze decidability of queries. We focus our attention on properties of query atoms and their relevant subprograms rather than on a logic program as a whole. Our observation is that within a logic program P various atoms in its Herbrand base can have various properties, e.g., finiteness of the dependency relation or length of the maximal dependency path. This leads to the fact that within a single program P different query atoms have different relevant subprograms and thus yield different decidability results. We divide a Herbrand base of a normal logic program P into three subsets with respect to the cardinality and properties of an atom dependency relation. *Finitely recursive atoms* with a finite dependency relation enjoy decidability of query evaluation because their relevant subprograms are finite. Decidability of atoms with an infinite dependency relation is obtained with the help of certain restrictions on the form of their relevant subprograms. For example, *ω-recursive atoms* are decidable because their value can be computed in finite time due to the *maximal path property* defined in Chapter 2. And *ω-restricted atoms* have relevant subprograms whose model computation is a 2-**NEXP**-complete problem.

3.1 Finitely Recursive Atoms

We focus the analysis of a program on its Herbrand base. Let us take a look at the Herbrand base B_P of a normal logic program P. We divide it into subsets with respect to finiteness of dependency relation of atoms.

Definition 3.1.1. (Finitely Recursive and Infinitely Recursive Atoms) Let P be a normal logic program and B_P its Herbrand base with $A \in B_P$. B_P is divided into subsets by the following principles:

(i) We say that A is *a finitely recursive atom* iff its dependency relation D_A is finite. We denote *a finitely recursive subset* of B_P containing only finitely recursive atoms as $\mathcal{FIN}$.

(ii) We say that A is *an infinitely recursive atom* iff its dependency relation D_A is infinite. We denote *an infinitely recursive subset* of B_P containing only infinitely recursive atoms as $\mathcal{INF}$.

From the fact that the well-founded semantics enjoys *relevance* it follows that queries can be answered using a fragment of the program instantiation. Such a fragment includes definitions of atoms occurring in the query plus definitions of all atoms on which the query atoms depend. In other words, this fragment includes all rules that define the dependency relation of query atoms.

We know that logic programming with negation under the well-founded semantics is Π^1_1-complete (see [Sch95]). This allows us to conclude that, in general, an infinitely recursive atom can have a dependency relation of cardinality larger than ω.

Bonatti [Bon01b] has defined finitely recursive programs as a technical concept used to introduce finitary programs under the stable model semantics. We have shown in [CHH07] that finitely recursive programs constitute an independent class of programs under the well-founded semantics which enjoys semi-decidability of query evaluation. Now we have another definition of finitely recursive programs as programs with B_P consisting solely of finitely recursive atoms. As we already know (see [CHH07]), query evaluation for such programs is decidable for ground queries and semi-decidable for existentially quantified queries. We can thus conclude that for all atoms from $\mathcal{FIN} \subseteq B_P$ of some normal logic P query evaluation is (semi-)decidable.

However, in contrast to the results obtained in [CHH07], we observe that a program P does not have to be finitely recursive for decidability of ground queries. Finiteness of dependency relation of a query atom alone guarantees finiteness of its relevant subprogram thus ensuring decidability.

Lemma 3.1.1. *Each finitely recursive ground atom A has a finite relevant subprogram P_A.*

Proof. If A is finitely recursive, $U_{rel}(P, A)$ must be finite by definition. So it follows that P_A is finite, as well. □

Proposition 3.1.2. *Given a normal logic program P and a query Q consisting of finitely recursive ground atoms, the truth value of Q under the well-founded model of P is decidable.*

Proof. By *relevance*, Q is entailed by P iff Q is entailed by P_Q. By Lemma 3.1.1, P_Q is finite, so its well-founded model can be computed in finite time. It follows that the problem of entailment for P and Q is decidable. □

It easily follows that existentially quantified finitely recursive goals are semi-decidable. From now on, the existential closure of Q is denoted by $\exists Q$.

Corollary 3.1.3. *Given a normal logic program P and a query Q consisting of finitely recursive ground atoms, the truth value of $\exists Q$ under the well-founded model of P is semi-decidable.*

Proof. The formula $\exists Q$ is entailed by P iff there exists a grounding substitution θ such that $Q\theta$ is entailed by P. By Proposition 3.1.2, the latter problem is decidable, and all grounding substitutions θ for Q can be recursively enumerated. Thus, existential entailment can be reduced to a potentially infinite recursive sequence of decidable tests, that terminates if and only if some $Q\theta$ is entailed. It follows that the problem of entailment for P and $\exists Q$ is semi-decidable. □

The following example shows us a program with infinite instantiation, but with finitely recursive atoms.

Example 3.1.1. ***(Finitely Recursive Atoms)***

$$
\begin{array}{rl}
P: & p(a). \\
R_1 = & p(f(X)) \leftarrow p(X), \neg q(X). \\
R_2 = & q(X) \leftarrow r(X). \\
R_3 = & n(X) \leftarrow \neg p(Y). \\
\hline
Q_1: & ? - p(f^2(a)). \\
Q_2: & ? - n(f(a)). \\
P_{Q_1}: & p(a). \\
 & p(f^2(a)) \leftarrow p(f(a)), \neg q(f(a)). \\
 & p(f(a)) \leftarrow p(a), \neg q(a). \\
 & q(f(a)) \leftarrow r(f(a)). \\
 & q(a) \leftarrow r(a). \\
\hline
P_{Q_2}: & p(a). \\
R_1: & p(f(a)) \leftarrow p(a), \neg q(a). \\
 & \vdots \\
 & p(f^i(a)) \leftarrow p(f^{i-1}(a)), \neg q(f^{i-1}(a)). \\
 & \vdots \\
R_2: & q(a) \leftarrow r(a). \\
 & \vdots \\
 & q(f^i(a)) \leftarrow r(f^i(a)). \\
 & \vdots \\
R_3: & n(f(a)) \leftarrow \neg p(a). \\
 & n(f(a)) \leftarrow \neg p(f(a)). \\
 & \vdots \\
 & n(f(a)) \leftarrow p(f^i(a)). \\
 & \vdots \\
 & i \in \mathbb{N}.
\end{array}
$$

The program instantiation $ground(P)$ is infinite. This is due to the the function symbol in the head of the first rule and a local variable in the last rule. A *local variable* is a variable occurring only in the body of a rule.

The first rule contains a function symbol that makes $ground(P)$ infinite. But we also see that both arguments in the body predicates of this rule are strict subterms of the head's argument. The arity of the function symbol decreases along the dependency path of the head atom thus ensuring its finiteness. Therefore $p(f^2(a))$ is a finitely recursive atom. The relevant subprogram P_{Q_1} is finite, even though the atom contains

a function symbol in the argument.

The second condition that makes $ground(P)$ infinite is a local variable in the last rule. In contrast to Q_1, Q_2 is not finitely recursive. We can see that the relevant subprogram of Q_2 is infinite. We shall take a look at this case in Section 3.2.

Recognizing the class of finitely recursive atoms is not decidable in general. Checking whether an atom A is finitely recursive can be reduced to the halting problem of a Turing machine with semi-infinite tape. The *halting problem* [1] is a decision problem about properties of computer programs on a fixed Turing-complete model of computation. The problem is to determine, given a program and an input to the program, whether the program will eventually halt when run with that input. In this abstract framework, there are no resource limitations of memory or time on the program's execution; it can take arbitrarily long, and use arbitrarily much storage space, before halting. The question is simply whether the given program will ever halt on a particular input.

To show undecidability of finitely recursive atoms recognition, we use a Turing machine encoding by the program $P_{\mathcal{M}}$ from [Bon01b]. This program is appealing because each rule has at most one subgoal, and therefore SLD-derivations are in one-to-one correspondence with paths in the dependency graph. We are going to prove that the class of finitely recursive atoms is undecidable by showing that an atom $A \in B_{P_{\mathcal{M}}}$ is finitely recursive iff $\mathcal{M}$ terminates from the configuration encoded by A. The Turing machine configuration encoded by an atom A will be called *A-configuration*.

Let $\mathcal{M}$ be an arbitrary deterministic Turing machine with semi-infinite tape; let S and V be $\mathcal{M}$'s set of states and tape alphabet, respectively. Recall that the instructions of $\mathcal{M}$ are 5-tuples $\{s, v, v', s', m\} \in S \times V \times V \times S \times \{left, right\}$, where s and v are the current state and symbol, respectively, v' is the symbol to be overwritten on v, s' is the next state, and m specifies $\mathcal{M}$'s head movement.

We say that a computation uses at most k cells if each configuration in the computation satisfies the following property: if the machine's head is on the i-th cell of

[1] The definition of the halting problem is taken from `http://en.wikipedia.org/wiki/Halting_problem`

`t(s,L,v,[V|R],C)`$\leftarrow$ `t(s',[v'|L],V,R,C+1).`	for all inst. $\langle s, v, v', s', right\rangle$
`t(s,L,v,[],C)`$\leftarrow$`t(s',[v'|L],b,[],C+1).`	for all inst. $\langle s, v, v', s', right\rangle$
`t(s,[V|L],v,R)`$\leftarrow$`t(s',L,V,[v'|R],C+1).`	for all inst. $\langle s, v, v', s', left\rangle$
`t(s,[],v,R,C)`$\leftarrow$`t(s',[],b,[v'|R],C+1).`	for all inst. $\langle s, v, v', s', left\rangle$
`t(s,L,v,R).`	for all final states s.

Figure 3.1: *A Turing machine encoding by program $P_{\mathcal{M}}$.*

the tape, then $i \leq k$.

Each configuration of $\mathcal{M}$ can be finitely encoded by an atom $t(s, L, v, R)$ where s is the current state, v is the current symbol, L is the list of symbols on the left of the machine's head in reverse order, and R is the list of symbols on the right of the machine's head, truncated after the last non-blank symbol (but not necessarily immediately after it).

Now consider the definite program $P_{\mathcal{M}}$ illustrated in Figure 3.1. The last argument of predicate t is a step counter, and the second and fourth rules are introduced to extend the tape encoding and add blanks (represented by b) whenever the head moves beyond tape boundaries.

Theorem 3.1.4. *Given a logic program P and an atom $A \in B_P$, checking whether A is finitely recursive is not decidable.*

Proof. **"If"-direction:** we assume that A is finitely recursive and prove that the corresponding A-configuration of $\mathcal{M}$ terminates.

By the program's construction, (i) the computations of $\mathcal{M}$ on tape x, (ii) the SLD-derivations starting from $t(s, L, V, R, c)$ where s is a state, terms L, V, R encode correctly x, and c is any fixed term, and (iii) the corresponding paths in the dependency graph, are in one to one correspondence. More precisely, the nth configuration in $\mathcal{M}$'s computation is encoded by the unique atom belonging to the nth goal in the SLD-derivation, and such atom equals the nth element in the path. The step counter (last argument of t) ensures that the dependency graph is acyclic. Then, if $\mathcal{M}$ falls into a cycle from A-configuration, the dependency graph of $P_{\mathcal{M}}$ contains an infinite acyclic path starting with A and hence A is not finitely recursive. By contraposition, it proves that if A is finitely recursive, then the computation of A-configuration terminates.

"Only if"-direction: suppose that A is not finitely recursive and prove that the corresponding A-configuration of $\mathcal{M}$ is non-terminating.

A is not finitely recursive. Then, there must be an infinite acyclic path in the dependency graph starting from A. Now there are two possibilities.

In the first case, the initial element of the path is an atom $A = t(s, L, V, R, t)$ where s is a state and L, V, R is a correct tape encoding. In this case, $\mathcal{M}$ has a corresponding non-terminating computation starting with A-configuration by the program's construction.

The second possibility is that either s is not a state or L, V, R is not a correct tape encoding. If s were not a state or the head moved over a symbol that does not belong to $\mathcal{M}$'s alphabet, then the SLD-derivation corresponding to the path would finitely fail, by the program's definition, and hence the path would be finite—a contradiction. Therefore, s must be a state and the head must never move over a cell with illegal content. In this case, it can be easily verified that by uniformly substituting illegal tape values with any legal values in all of the path's elements, we obtain another path of the dependency graph, where the initial element A correctly encodes an A-configuration of $\mathcal{M}$. Therefore, $\mathcal{M}$ has a corresponding non-terminating computation. This proves that if A is not finitely recursive then A-configuration of $\mathcal{M}$ induces an infinite computation.

We conclude that A is finitely recursive iff the computation starting from A-configuration of $\mathcal{M}$ terminates. □

We have seen than checking whether a given atom is finitely recursive is an undecidable problem. However, a subclass of finitely recursive atoms can be effectively checked. When does a ground atom A depend on infinitely many atoms? It happens in the following cases:

- a dependency path starting in the node A is of infinite length,
- a dependency path starting in the node A contains infinite branching.

A dependency path of infinite length appears in $ground(P)$ if a program P before instantiation contains the following schema:

$$\begin{array}{rl} P: & p(X) \leftarrow p(f(X)). \\ \hline ground(P): & p(a) \leftarrow p(f(a)). \\ & p(f(a)) \leftarrow p(f^2(a)). \\ & p(f^2(a)) \leftarrow p(f^3(a)). \\ & \vdots \end{array}$$

We see here that a recursive predicate definition with the body predicate containing a super-term of the head's argument (a term "grows" along the dependency path) always gets instantiated into an infinite dependency path in the atom-call dependency graph of P. As we have seen in the example above, if the term in the body (a subterm of head's argument) is smaller or of the same size as in the head, we do not have an infinite recursion:

$P_1:$	$p(X) \leftarrow p(X).$	$P_2:$	$p(f(X)) \leftarrow p(X).$
$ground(P_1):$	$p(a) \leftarrow p(a).$	$ground(P_2):$	$p(f(a)) \leftarrow p(a).$
	$p(f(a)) \leftarrow p(f(a)).$		$p(f^2(a)) \leftarrow p(f(a)).$
	$p(f^2(a)) \leftarrow p(f^2(a)).$		$p(f^3(a)) \leftarrow p(f^2(a)).$
	$\vdots$		$\vdots$

So, in order to recognize finitely recursive atoms, the recursion patterns of the input predicates have to be analyzed, looking for arguments whose terms do not increase indefinitely.

The second case with an infinite branching from a node in an atom-call dependency graph of P arises in the presence of local variables. This situation can be represented by the following schema:

$P:$	$p(X) \leftarrow q(Y).$
$ground(P):$	$p(a) \leftarrow q(a).$
	$p(a) \leftarrow q(f(a)).$
	$p(a) \leftarrow p(f^2(a)).$
	$\vdots$

To exclude infinite branching, one can check if the head arguments in a rule bound the local variables in the body.

If predicate's definition as well as all predicates below it in the dependency relation do not contain any of the situations that lead to infiniteness in a dependency graph of P, that is, terms that "grow" along the dependency path or local variables, then this predicate will be instantiated to finitely recursive atoms.

As an example of predicates that usually get instantiated into finitely recursive atoms, one can regard most classical predicates on recursive data structures such as lists and trees (e.g. predicates `member, append, reverse` in Figure 3.2). We

```
member(X,[X|Y]).
member(X,[Y|Z])← member(X,Z).

append([],L,L).
append([X|Xs],L,[X|Ys])← append(Xs,L,Ys).

reverse(L,R)← reverse(L,[],R).
reverse([],R,R).
reverse([X|Xs],A,R)← reverse(Xs,[X|A],R).
```

Figure 3.2: *List processing, finitely recursive predicates.*

can see that these predicates generate finitely recursive atoms because the terms occurring in the body of a rule occur also in the head, often as strict subterms of the head's arguments. Such predicates are also called *covered* or predicates *without local variables*.

Bonatti in [Bon01a] has developed a prototype recognizer, a part of which is aimed at finitely recursive atoms. It operates on a finite non-ground program. Therefore, in order to identify a finitely recursive atom with the help of this recognizer, its relevant subprogram has to be introduced in a non-ground version. Intuitively, such a non-ground subprogram contains definitions of all predicates that occur in the dependency relation of a given atom. Here is a formal definition.

Definition 3.1.2. (Non-ground Relevant Subprogram) The *non-ground relevant subprogram* for a ground query Q (w.r.t. a normal program P), denoted by P_Q^{NG}, is the set of all rules in P whose grounded versions belong to P_Q:

$$P_Q^{NG} = \{R | R \in P \text{ and } ground(R) \in P_Q\}.$$

The part of the prototype recognizer that discerns finitely recursive atoms consists of three stages.

Interargument analysis checks relationships between the size of terms occurring as predicate arguments in a rule of the input program. It takes a special care of the size of head arguments with respect to the size of body arguments. This is done in order to find out if the head arguments in a rule bound the local

variables in the body; or if the head recursive predicate can occur in the body on arguments with a bigger size (note that this could generate infinite sequences of recursive calls without repeats).

Recursive analysis uses the results of the interargument analysis and finds, for each predicate symbol, the group of its arguments whose size decreases (or at least does not increase "too much") at each recursive call. It also examines cyclic atom dependencies classifying them in *good recursion cycles* and *false cycles* by identifying suitable recursion patterns. Intuitively, an atom whose relevant subprogram contains only good recursion cycles or false cycles, is finitely recursive.

Recursive domain predicate identification considers negative cycles spotted as false cycles during the previous phase for identifying domain predicates that can be used to compute an optimized partial evaluation of the input program. The subprogram of all domain predicates has the well-founded model that is contained in the well-founded model of the entire program. It is the case because domain predicates constitute the bottom program of a splitting set of the given program. This model can be used to simplify the ground program instantiation by considering only rule instances whose domain subgoals are true. Depending on its recursion properties, such as the existence or lack of positive cycles, a domain predicate may be evaluated in a naive top-down fashion, in a bottom-up fashion, or in a tabled fashion.

We sum up by saying that even though finitely recursive atoms are an undecidable class, in practice this negative result can be evaded in many cases.

3.2 Omega-Recursive Atoms

In this section we define *ω-recursive atoms* and show that, even if a logic program P contains function symbols and all of its Herbrand base atoms are infinitely recursive, we can still obtain decidability of query evaluation for a certain subset of its Herbrand base. This subset contains ground atoms whose relevant subprograms are *ω-recursive*.

As we have seen in the previous section, ground finitely recursive atoms enjoy decidability when they occur in queries. If an atom has an infinite dependency relation, its cardinality, in general, can be larger than ω, because the logic programming with negation under the well-founded semantics is expressive enough to encode Π^1_1-relations (see [Sch95]). In this section we describe a situation when a ground atom A has an infinite dependency relation but, if its relevant subprogram meets a certain syntactic condition, a model of this subprogram can be computed in finite time.

We want a relevant subprogram to be *ω-recursive*—the cardinality of the longest path in its dependency graph, including cyclic paths, must be finite, strictly smaller than ω. In other words, to obtain decidability, the dependency graph of the relevant subprogram is allowed to have only finite dependency paths. The restriction on the length of dependency paths is less strong than that on the size of dependency relation for finitely recursive atoms. This new condition admits infinite dependency relations—the number of finite dependency paths can still be infinite.

Definition 3.2.1. (ω-Recursive Atoms) Let P be a normal logic program, $\mathcal{G}_P^{atom}$ its atom-call dependency graph and B_P its Herbrand base with $A \in B_P$. We say that A is *an ω-recursive atom* iff all dependency paths π_A starting at A are of finite cardinality: for all $\pi_A \in \mathcal{G}_P^{atom}$ it holds that $|\pi_A| < \omega$.

Proposition 3.2.1. *Given a normal logic program P and a query Q consisting of ω-recursive ground atoms, the truth value of Q under the well-founded model of P is decidable.*

Proof. By *relevance*, Q is entailed by P iff Q is entailed by P_Q. Let $Q = A_1, ..., A_n$. Since all $A_i, i = 1..n$ are ω-recursive atoms, all π_{A_i} are of finite cardinality by definition. It follows that each relevant subprogram P_{A_i} has a dependency graph with $|\pi^{max}| = \beta$ such that $\beta < \omega$. By Proposition 2.6.4 the model M_{A_i} of each P_{A_i} can be computed in at most β iterations, $M_{A_i} = lfp(W_P) = W_P \uparrow \alpha$ with $\alpha \leq \beta$ and $\beta < \omega$. It follows that $\alpha < \omega$, so the well-founded model of each π_{A_i} can be computed in finite time. It follows that the problem of entailment for P and Q is decidable. □

It easily follows that existentially quantified finitely recursive queries are semi-decidable.

Corollary 3.2.2. *Given a normal logic program P and a query Q consisting of ω-recursive ground atoms, the truth value of $\exists Q$ under the well-founded model of P is semi-decidable.*

Proof. The formula $\exists Q$ is entailed by P iff there exists a grounding substitution θ such that $Q\theta$ is entailed by P. By Proposition 3.2.1, the latter problem is decidable, and all grounding substitutions θ for Q can be recursively enumerated. Thus, existential entailment can be reduced to a potentially infinite recursive sequence of decidable tests, that terminates if and only if some $Q\theta$ is entailed. It follows that the problem of entailment for P and $\exists Q$ is semi-decidable. □

We take an example from the previous section, slightly modify it and show that, with the help of the *maximal path property*, we have enlarged the subset of decidable queries.

Example 3.2.1. ***(ω-Recursive Atoms)***

$$
\begin{array}{rl}
P: & p(a). \\
R_1 = & p(f(X)) \leftarrow p(a), \neg q(X). \\
R_2 = & q(X) \leftarrow r(X). \\
R_3 = & n(X) \leftarrow \neg p(Y). \\
\hline
Q: & ? - n(f(a)). \\
\hline
P_Q: & p(a). \\
R_1: & p(f(a)) \leftarrow p(a), \neg q(a). \\
 & \vdots \\
 & p(f^i(a)) \leftarrow p(a), \neg q(f^{i-1}(a)). \\
 & \vdots \\
R_2: & q(a) \leftarrow r(a). \\
 & \vdots \\
 & q(f^i(a)) \leftarrow r(f^i(a)). \\
 & \vdots \\
R_3: & n(f(a)) \leftarrow \neg p(a). \\
 & \vdots \\
 & n(f(a)) \leftarrow p(f^i(a)). \\
 & \vdots \\
 & i \in \mathbb{N}.
\end{array}
$$

The program instantiation $ground(P)$ is infinite. This is due to the function symbol in the head of the first rule and a local variable in the third rule. The query

Q is not finitely recursive. We can see that the relevant universe as well as the relevant subprogram of Q are infinite. It is caused by the local variable in the rule that defines this atom's predicate.

However, Q is ω-recursive. There is an infinite number of dependency paths from the atom $n(f(a))$ making its dependency relation infinite. But all these dependency paths are finite. $\pi^{max} = \{n(f(a)), p(f^i(a)), q(f^{i-1}(a)), r(f^{i-1}(a))\}$. The cardinality of the maximal dependency path is finite, $|\pi^{max}| = 4 = \beta$, and it is easy to verify that the model of P_Q is computed (or the least fixed point of W_{P_Q} is reached) after 2 iterations:

$W_{P_Q} \uparrow 1 = \{p(a)\} \cup \{\neg r(f^i(a)), \neg q(f^i(a))\}, i \geq 0$
$W_{P_Q} \uparrow 2 = \{p(f^i(a))\}, i \geq 0 \cup \{\neg r(f^i(a)), \neg q(f^i(a)), \neg n(f(a))\}$
$W_{P_Q} \uparrow 3 = W_{P_Q} \uparrow 2 = lfp(W_{P_Q})$

Thus we have $W_{P_Q} \uparrow 2 = lfp(W_{P_Q}), \alpha = 2$ and $\alpha \leq \beta$ holds.

The *maximal path property* allows us to exclude the occurrence of local variables from the undecidability conditions. Even if there is an infinite number of branches in the atom-call dependency graph of a program that start in the node A, we are still guaranteed a possibility to compute the value of A as long as all branches are finite.

It is easy to see that ω-recursive atoms form a superset of finitely recursive atoms. Since the class of finitely recursive atoms is not decidable, it follows that the problem of ω-recursive atoms recognition is also undecidable.

Corollary 3.2.3. *Given a logic program P and an atom $A \in B_P$, checking whether A is ω-recursive is not decidable.*

3.3 Omega-Restricted Programs

In this section we introduce a concept of *ω-restricted programs*. The idea of ω-restricted programs is first given by Syrjänen [Syr01]. He has defined them as a class of programs that admit function symbols and nevertheless enjoy decidability under the stable model semantics.

We need ω-restricted programs for the definition of the next class of decidable atoms—we call them also *ω-restricted*. We want a relevant subprogram for these atoms to be ω-restricted—it should be possible to construct a hierarchy of predicates

such that all variables occurring in a rule of level $n+1$ have to also occur in a positive literal of level n or lower in the rule body. The definition of such a hierarchy extends the usual principle of stratification by adding a special ω-stratum to hold the unstratifiable part of the program. The restriction on variables guarantees that each ground instantiation for the stratified part is a ground instantiation also for the ω-stratum.

Note that an ω-restricted program is also *range-restricted*.

Definition 3.3.1. (Range-Restricted Program) A program is *range-restricted* if in every rule each variable that occurs in the head or in a negative body literal also occurs in a positive body literal.

We start introduction of ω-restricted programs with the definition of *ω-stratification*.

Definition 3.3.2. (ω-Stratification [Syr01]) An *ω-stratification* of a program P is a function $s : preds(P) \rightarrow \mathbb{N} \cup \{\omega\}$ such that:

1. $\forall p_1, p_2 (p_1 >_+ p_2 \Rightarrow s(p_1) \geq s(p_2))$ and
2. $\forall p_1, p_2 (p_1 >_- p_2 \Rightarrow s(p_1) > s(p_2) \vee s(p_1) = \omega)$.

The next concept of *ω-valuation* extends ω-stratification to cover also rules and variables.

Definition 3.3.3. (ω-Valuation [Syr01]) An *ω-valuation* of a rule R under an ω-stratification s is a function

$$\Omega(R, s) = s(pred(head(R))).$$

An *ω-valuation* of a variable v in a rule R under an ω-stratification s is a function

$$\Omega(v, R, s) = min(\{s(pred(A)) | A \in body^+(R) \wedge v \in \mathcal{V}(A)\} \cup \{\omega\}).$$

Definition 3.3.4. (ω-Restricted Rule [Syr01]) Let R be a rule in a logic program P. Then R is *ω-restricted* if and only if there exists an ω-stratification s such that

$$\forall v \in \mathcal{V}(R) : \Omega(v, R, s) < \Omega(R, s).$$

Definition 3.3.5. (ω-Restricted Program [Syr01]) A logic program P is *ω-restricted* if and only if all rules $R \in P$ are ω-restricted.

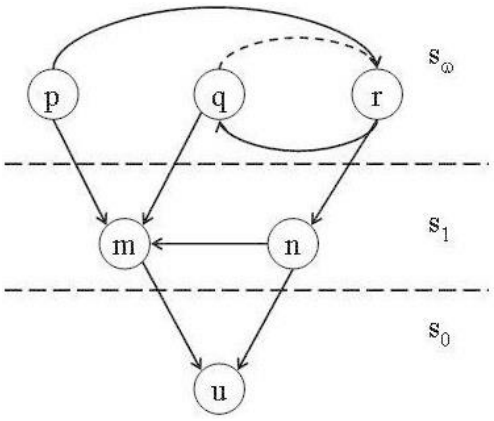

Figure 3.3: Predicate-call dependency graph of P and ω-stratification s.

For example, consider the following program P and its ω-stratification s.

Example 3.3.1. ***(ω-Restricted Programs)***

$$
\begin{array}{rl}
P: & \\
R_1 = & p(X) \leftarrow r(X), m(X). \\
R_2 = & q(X) \leftarrow \neg r(X), m(X). \\
R_3 = & r(X) \leftarrow q(X), n(X). \\
R_4 = & m(X) \leftarrow u(X). \\
R_5 = & n(X) \leftarrow m(X), u(X). \\
\hline
s_\omega = & \{p, q, r\} \\
s_1 = & \{m, n\} \\
s_0 = & \{u\} \\
\hline
\Omega(R_1) = & \omega > \Omega(X, R_1, s) = 1 \\
\Omega(R_2) = & \omega > \Omega(X, R_2, s) = 1 \\
\Omega(R_3) = & \omega > \Omega(X, R_3, s) = 1 \\
\Omega(R_4) = & 1 > \Omega(X, R_4, s) = 0 \\
\Omega(R_5) = & 1 > \Omega(X, R_5, s) = 0
\end{array}
$$

The predicate-call dependency graph of P along with ω-stratification s are given in Figure 3.3. P is ω-restricted because each rule $R \in P$ meets the condition defining ω-restricted programs. We can see that, for the given ω-stratification s, each variable occurs in a positive body literal of a stratum lower than the head in every rule.

Finally, the predicate symbols of P are divided into two classes, *domain predicates*

that are on finite strata and non-domain predicates that are on the ω-stratum.

Definition 3.3.6. (Domain Predicates [Syr01]) Let P be an ω-restricted program. Then a predicate $p \in preds(P)$ is a *domain predicate* if and only if there exists an ω-stratification s such that $s(p) < \omega$. The set of rules defining domain predicates of P is denoted by $\mathcal{D}(P)$.

This means that each ground instantiation for the stratified part is a ground instantiation also for the ω-stratum.

The subprogram $\mathcal{D}(P)$ that defines the domain predicates is stratified, therefore it has the least model $M_{\mathcal{D}(P)}$. $M_{\mathcal{D}(P)}$ forms *a splitting set* of P. The *splitting set* for a program P is any set U such that, for any rule $R \in P$, if $head(R) \cap U \neq \emptyset$ then $lit(R) \subset U$. The idea of splitting a logic program was introduced by Lifschitz and Turner in [LT94]. It implies that a logic program can be split into two parts, so that one of them, the "bottom" part, does not refer to the predicates defined in the "top" part. The bottom rules can be used then for evaluation of the predicates that they define, and the computed rules can be used to simplify the "top" definitions.

$M_{\mathcal{D}(P)}$ is a splitting set of P because it is closed under the condition: if $head(R) \in M_{\mathcal{D}(P)}$ then $body(R) \in M_{\mathcal{D}(P)}$. The domain subprogram $\mathcal{D}(P)$ constitutes a "bottom" part. We can compute the well-founded model of P by first computing $M_{\mathcal{D}(P)}$ and then extending it to cover the atoms on ω-stratum. The condition of ω-restriction put on the rules of P guarantees that each variable occurring in the rule occurs also in a positive body literal. This allows us to create all relevant ground instances of a rule R by computing the natural join of extensions of the domain literals in $body(R)$.

The following algorithm is used in [Syr01] to compute $M_{\mathcal{D}(P)}$ and the instantiation P_{NG} of $P_N = P \backslash \mathcal{D}(P)$:

1. Find all strongly connected components of the predicate-call dependency graph of P. Each component becomes a new stratum with the exception that all components that have a path to a negative dependency cycle are put on the ω-stratum. Order the different strata by doing a depth-first search over the strongly connected components.

2. Instantiate the predicates on finite strata starting from the lowest one. After instantiation, compute the deductive closure of the new ground rules and store the resulting atoms as facts in a database. These facts are then used to give domains for variables when we instantiate the rules on higher strata.

3. Instantiate all rules on the ω-stratum and output them along with the domain facts.

We want to establish decidability of the following two problems for the class of ω-restricted programs.

INSTANTIATION given an ω-restricted program P and a ground atom A, we want to find out whether one of the following holds:

1. $A \in M_{\mathcal{D}(P)}$ or
2. there is a rule $A \leftarrow L_1, ..., L_n$ in P_{NG}.

MODEL given an ω-restricted program P we want to compute a model of P.

Syrjänen [Syr01] has proven the complexity results for the whole class of ω-restricted programs:

- the INSTANTIATION of an ω-restricted program is 2-**EXP**-complete and
- the MODEL of an ω-restricted program is 2-**NEXP**-complete.

Since the MODEL problem is 2-**NEXP**-complete, it follows that:

Theorem 3.3.1. [Syr01] *Both* INSTANTIATION *and* MODEL *are decidable for ω-restricted programs under the stable model semantics.*

We know from [Sch95] that over the class of infinite Herbrand universes the stable model semantics and the well-founded semantics have the same expressive power[2]. This allows us to conclude that:

[2]see Section 4.2 Theorem 4.4 in [Sch95].

Corollary 3.3.2. *Both* INSTANTIATION *and* MODEL *are decidable for* ω*-restricted programs under the well-founded semantics.*

With this result, we can use the ω-restricted programs in the definition of another class of decidable atoms—*ω-restricted atoms*. If such an atom has P_{rel} which is ω-restricted, due to the above statement it will be easy to prove decidability of evaluation of such an atom under the well-founded semantics.

3.4 Omega-Restricted Atoms

In this section we define *ω-restricted atoms* and show that, even if a logic program P contains function symbols and all of its Herbrand base atoms are infinitely recursive, we can still obtain decidability of query evaluation for a certain subset of its Herbrand base. This subset contains ground atoms whose relevant subprograms are ω-restricted.

We need a notion of a non-ground relevant subprogram (see Section 3.1) for the purpose of checking the ω-restricted condition. A non-ground relevant subprogram contains variables and therefore it is an abbreviation for the ground P_{rel}. However, if we take a grounded version of P_{rel}^{NG}, we will get a superset of P_{rel}.

Definition 3.4.1. (ω-Restricted Atoms) Let P be a normal logic program and B_P its Herbrand base with an infinitely recursive atom $A \in B_P$. We say that A is an *ω-restricted atom* iff its non-ground relevant subprogram P_A^{NG} is ω-restricted.

The basic idea here is, as in Section 3.1, to analyze relevant subprograms of queries instead of the whole logic program. A logic program P as a whole might not meet any syntactic conditions that guarantee decidability of query evaluation (e.g., it is not finitely recursive nor ω-restricted), but some of the queries posed to this P can be still decidable—due to decidability of model computation for the query's P_{rel}.

Let us consider a situation when we are given a program P with some ground query atom A. First, we check the dependency relation of A and find out that it is infinite—so, A is infinitely recursive. Second, we obtain P_A and see that it is infinite and not ω-recursive due to the presence of infinite dependency paths. But, if we take the non-ground version of P_A, which is finite, we can check each of its rules for the

ω-restricted condition. We remind that each rule in an ω-restricted program must satisfy the following: every variable should occur at least once in a positive body literal of a stratum lower than the head with respect to some ω-stratification. If the non-ground version of P_A happens to be ω-restricted, we can compute the value of A by solving the MODEL problem from Section 3.3. As we have seen in [Syr01], the MODEL problem for ω-restricted programs is 2-**NEXP**-complete, so it follows that the value of A is decidable in our situation. Formally, it can be stated as follows:

Proposition 3.4.1. *Given a normal logic program P and a query Q consisting of ω-restricted ground atoms, the truth value of Q under the well-founded model of P is decidable.*

Proof. By *relevance*, Q is entailed by P iff Q is entailed by P_Q. By definition of ω-restricted atoms, Q has an ω-restricted relevant subprogram P_Q^{NG}. By Corollary 3.3.2, the well-founded model of P_Q^{NG} can be computed in finite time. It follows that the problem of entailment for P and Q is decidable. □

It follows that existentially quantified ω-restricted goals are semi-decidable.

Corollary 3.4.2. *Given a normal logic program P and a query Q consisting of ω-restricted ground atoms, the truth value of $\exists Q$ under the well-founded model of P is semi-decidable.*

Proof. The formula $\exists Q$ is entailed by P iff there exists a grounding substitution θ such that $Q\theta$ is entailed by P. By Proposition 3.4.1, the latter problem is decidable, and all grounding substitutions θ for Q can be recursively enumerated. Thus, existential entailment can be reduced to a potentially infinite recursive sequence of decidable tests, that terminates if and only if some $Q\theta$ is entailed. It follows that the problem of entailment for P and $\exists Q$ is semi-decidable. □

The following example illustrates ω-restricted atoms.

Example 3.4.1. ***(ω-Restricted Atoms)***

$$
\begin{array}{rll}
P: & p(a). & \\
R_1 = & p(f(X)) \leftarrow p(X), q(X,Y), n(X). & \\
R_2 = & q(X,Y) \leftarrow r(X,Y). & \\
R_3 = & n(X) \leftarrow \neg p(Y), q(X,Y). & \\
\hline
Q: & ?- n(f(a)). & P_Q^{NG} = P
\end{array}
$$

$$
\begin{array}{ll}
P_Q: & p(a). \\
R_1: & p(f(a)) \leftarrow p(a), q(a,a), n(a). \\
 & p(f(a)) \leftarrow p(a), q(a,f(a)), n(a). \\
 & \vdots \\
 & p(f(a)) \leftarrow p(a), q(a,f^i(a)), n(a). \\
 & \vdots \\
 & p(f^i(a)) \leftarrow p(f^{i-1}(a)), q(f^{i-1}(a),a), n(f^{i-1}(a)). \\
 & \vdots \\
 & p(f^i(a)) \leftarrow p(f^{i-1}(a)), q(f^{i-1}(a),f^i(a)), n(f^{i-1}(a)). \\
 & \vdots \\
R_2: & q(a,a) \leftarrow r(a,a). \\
 & q(f(a),a) \leftarrow r(f(a),a). \\
 & \vdots \\
 & q(f^i(a),a) \leftarrow r(f^i(a),a). \\
 & \vdots \\
 & q(a,f(a)) \leftarrow r(a,f(a)). \\
 & \vdots \\
R_3: & n(f(a)) \leftarrow \neg p(a), q(f(a),a). \\
 & n(f(a)) \leftarrow \neg p(f(a)), q(f(a),f(a)). \\
 & \vdots \\
 & n(f(a)) \leftarrow \neg p(f^i(a)), q(f(a),f^i(a)). \\
 & \vdots \\
 & i \in \mathbb{N}.
\end{array}
$$

We have only sketched $ground(P)$ here, because a list of possible instances of rules will take too much space and is not needed. We can still see that the program instantiation $ground(P)$ is infinite, because the function symbol makes Herbrand universe infinite, and local variables occurring in all three rules cause infinite branching in the atom-call dependency graph of P. No query posed to P has a chance to be finitely recursive because local variables occur in all the rules. Q is not ω-recursive, either. It is easy to see that any dependency path starting in $n(f(a))$ has more than one point of infinite branching for the same reason—local variables in the rules defining the predicate n.

However, the local variables that make $ground(P)$ infinite also play another important role—their presence makes P_Q^{NG} (which in this case is equal to P) ω-restricted.

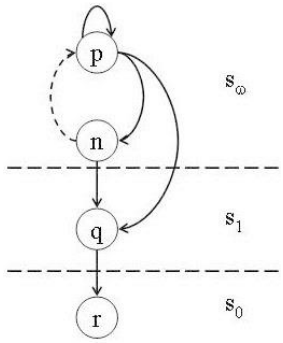

Figure 3.4: Predicate-call dependency graph of P_Q^{NG} and ω-stratification s.

The predicate-call dependency graph of P_Q^{NG} is depicted in Figure 3.4. For the ω-stratification s, as given below, each rule in P_Q^{NG} is ω-restricted.

$$
\begin{array}{llll}
s_\omega = & \{p, n\} & & \\
s_1 = & \{q\} & & \\
s_0 = & \{r\} & & \\
\hline
\Omega(R_1) = & \omega > & \Omega(X, R_1, s) = 1 & \Omega(Y, R_1, s) = 1 \\
\Omega(R_2) = & 1 > & \Omega(X, R_2, s) = 0 & \Omega(Y, R_2, s) = 0 \\
\Omega(R_3) = & \omega > & \Omega(X, R_3, s) = 1 & \Omega(Y, R_3, s) = 1
\end{array}
$$

Since P_Q^{NG} meets the conditions that define ω-restricted programs, we are able to compute its model and, consequently, the value of Q in finite time.

Recognition of an ω-restricted atom is performed via its non-ground relevant subprogram. In other words, the corresponding P_{rel}^{NG} should be ω-restricted. Since ω-restricted programs are defined by a syntactic condition in a non-ground setting, for a finite set of non-ground rules, it is easy to see that this condition can be effectively checked. Therefore the class of ω-restricted atoms is decidable.

Theorem 3.4.3. *Given a logic program P and an atom $A \in B_P$, checking whether A is ω-restricted is decidable.*

Chapter 4

Summary

4.1 Applicability of the Obtained Results

To demonstrate ways of application of the obtained results, we have developed the following algorithm. It takes a normal logic program P and a ground query Q and as an outcome we have two answers:

"decidable" means that the computation of the value of Q is a decidable problem;

"possibly undecidable" means that there is no guarantee that the value of Q can be computed in finite time.

Algorithm.

Input a normal logic program P, a ground query $Q = A_1, ..., A_n$. Initialize $i := 1$, until $i = n$ do the following:

Step 1 if $1 \leq i \leq n$, take A_i, set $A := A_i$ and goto **Step 2**. If $i > n$, **succeed** and output *"decidable"*.

Step 2 if $|D_A| < \omega$, then A is *finitely recursive*, set $i := i + 1$ and goto **Step 1**. Otherwise goto **Step 3**.

Step 3 if $|\pi_A^{max}| < \omega$, then A is ω-*recursive*, set $i := i+1$ and goto **Step 1**. Otherwise goto **Step 4**.

Step 4 if P_A^{NG} is *ω-restricted*, then A is *ω-restricted*, set $i := i + 1$ and goto **Step 1**. Otherwise **fail** and output *"possibly undecidable"*.

The given algorithm checks ground atoms of Q for decidability, one by one. If an atom is decidable, then the next member of Q is checked. With the first failure to establish decidability of a query atom, the algorithm outputs *"possibly undecidable"* and stops, because there is no need to check further members of Q. If all query atoms have undergone the decidability check, the algorithm succeeds and outputs *"decidable"*.

In **Step 2** a query atom is evaluated for finite recursiveness. For this condition to hold, we need the dependency relation of this query atom to be finite. As we have seen in Section 3.1, the condition defining finitely recursive atoms is, in general, undecidable. However, we have shown that this undecidability can be circumvented in practice. There also exists a prototype recognizer (see [Bon01a]) that can be used to identify finitely recursive atoms.

In **Step 3** a query atom is checked for ω-recursiveness. This condition allows for an infinite dependency relation, but the maximal dependency path of the query atom must be finite. We remind that, unfortunately, the condition defining ω-recursive atoms is undecidable. However, we believe that further studies of the problem will allow to effectively recognize a subclass of ω-recursive atoms.

In **Step 4** a query atom is tested for the ω-restricted condition. This condition allows for both an infinite dependency relation and for an infinite dependency path (see Section 3.4). For this condition to hold, we need a non-ground relevant subprogram of this query atom to be ω-restricted. For a detailed account on the condition that defines ω-restricted programs see Section 3.3 and [Syr01]. In contrast to the two conditions considered above, the ω-restricted programs condition can be effectively checked.

If at least one query atom does not satisfy any of the above conditions, then the algorithm outputs *"possibly undecidable"*. This means, on the basis of our results, the query Q does not meet any of the conditions that guarantee its decidability. We state "possibly" because there is still a possibility that a value of Q can be computed

in finite time. But this can be checked only by actual computation, to the best of our knowledge no preliminary, upfront check can be performed.

4.2 Related Work

The part of this work dedicated to the abstract properties of the well-founded semantics is based on the general approach introduced by Kraus, Lehmann and Magidor [KLM90] and then further developed by Makinson [Mak89, Mak93], Freund and Lehmann [FLM90, Fre93, FL94]. This abstract approach to the nonmonotonic inference was adopted for the logic programming setting and extended to a general theory of logic programming semantics by Dix [Dix95a, Dix95b].

The work of Hitzler and Wendt [HW02, HW05] was the starting point for our investigations. The maximal path property introduced in our work was stimulated by the level mapping characterization of the well-founded semantics presented in [HW02, HW05].

The work on finitely recursive atoms was inspired by the paper of Bonatti [Bon01b] on finitary programs. The idea of ω-restricted programs was adopted from Syrjänen [Syr01]. Work in a similar direction comprises papers on acyclic programs [AB91], acceptable programs [AP93], Φ-accessible programs [HS99], (locally) stratified programs [ABW88, Prz88], affordable classes of programs [SS97]. An infinitary proof theory can be found, for example, in [Mil99]. In [Ros99], issues related to decidable nonmonotonic reasoning in MKNF are investigated. Efficient methods for top-down computation of queries under the well-founded semantics are presented in [CSW95, RSS+97].

4.3 Conclusion

We have defined a new meta-level property of the well-founded semantics—*the maximal path property*. It tells us that the maximal possible number of l_P-levels in a model of normal logic program P is limited by the cardinality of the longest dependency path in the atom-call dependency graph of P. With this property it is possible to "predict" the number of W_P-operator iterations in computation of the well-founded

model of P. If the longest dependency path is finite, i.e. with cardinality strictly less than ω, we can conclude that the number of W_P-operator iterations will be less or equal to the number of atoms in the longest dependency path.

We have proposed a new setting for study of decidability of logic programs. We focus our attention on properties of query atoms and their relevant subprograms rather than on a logic program as a whole. Given a logic program P, atoms in its Herbrand base can have different properties, for example, finiteness of the dependency relation or length of the maximal dependency path. Within a single program P various query atoms have different relevant subprograms and thus yield different decidability results. The new setting allows us to operate on classes of decidable atoms that constitute subsets of a Herbrand base, rather than on classes of decidable programs.

A *finitely recursive atom* has a finite dependency relation and therefore a finite relevant subprogram. This definition is an adaptation of the definition of finitely recursive programs taken from [Bon01b]. Our results are new in the following aspect: we show that the whole logic program does not need to be finitely recursive to obtain decidability of query evaluation.

The new maximal path property has allowed us to define a class of *ω-recursive atoms*. Given a ground query consisting of ω-recursive atoms, we know that the relevant subprogram of this query can be infinite. But all dependency paths in the atom-call graph of the relevant subprogram are finite. And the maximal path property allows us to conclude that it is always possible to compute a model of such a relevant subprogram in finite time.

The class of *ω-restricted atoms* is based on the definition of ω-restricted programs from [Syr01]. A relevant non-ground subprogram of such an atom has to be ω-restricted, which guarantees that its model computation is a 2-**NEXP**-complete problem, as reported in [Syr01].

Both classes of finitely recursive and ω-recursive atoms are in general not decidable. However, there exists a recognition procedure for the subclass of finitely recursive atoms [Bon01a], and we believe that future studies will allow to develop

an effective procedure that discerns a subclass of ω-recursive atoms. The class of ω-restricted atoms is decidable due to its non-ground syntactic definition.

We have defined three classes of decidable atoms, i.e. atoms with relevant subprograms, well-founded models of which can be computed in finite time. All the three classes of atoms constitute a very expressive fragment of Default Logic, Autoepistemic Logic, and normal logic programs under the well-founded semantics. Many classical AI applications and problems such as diagnosis, reasoning about actions and change, propositional satisfiability [Bon01b], epistemic reasoning, and more, can be naturally modeled with programs containing the decidable classes of atoms in the Herbrand base. We claim that function symbols and recursion need not be excluded from ASP.

Our results show how queries can be answered by using only a strict subset of the ground instantiation of the program, that is, the relevant subprogram. This technique is interesting, and may have practical advantages also in the absence of function symbols.[1] It is well-known that the instantiation process is one of the most expensive computation phases of the existing inference engines. A smaller ground program instance may significantly reduce memory requirements and cut down the search space.

Decidable classes of atoms are also interesting from a theoretical perspective. Their properties are unusual among nonmonotonic formalisms. For example, first-order nonmonotonic reasoning over infinite domains is typically not semi-decidable.

[1] This observation is due to Bonatti [Bon01b].

Bibliography

[AB91] K. R. Apt and M. Bezem, *Acyclic programs*, New Generation Computing **9** (1991), no. 3-4, 335–365.

[ABW88] K. R. Apt, H. A. Blair, and A. Walker, *Towards a theory of declarative knowledge*, Foundations of Deductive Databases and Logic Programming (Jack Minker, ed.), Morgan Kaufmann, Los Altos, CA, USA, 1988, pp. 89–148.

[ADP95] J.J. Alferes, C.V. Damasio, and L.M. Pereira, *A logic programming system for nonmonotonic reasoning*, Journal of Automated Reasoning **14** (1995), no. 1, 93–147.

[AP93] K.R. Apt and D. Pedreschi, *Reasoning about termination of pure prolog programs*, Information and Computation **106** (1993), 109–157.

[BD98] S. Brass and J. Dix, *Characterizations of the disjunctive well-founded semantics: Confluent calculi and iterated GCWA*, Journal of Automated Reasoning **20** (1998), no. 1, 143–165.

[Bon01a] P. A. Bonatti, *Prototypes for reasoning with infinite stable models and function symbols*, Proceedings of the Sixth International Conference on Logic Programming and Nonmonotonic Reasoning, LPNMR 2001 (T. Eiter, W. Faber, and M. Truszczynski, eds.), Lecture Notes in Computer Science, vol. 2173, Springer Verlag, 2001, pp. 416–419.

[Bon01b] ———, *Reasoning with infinite stable models*, Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence, IJCAI 2001 (B. Nebel, ed.), Morgan Kaufmann, 2001, pp. 603–610.

[CHH07] N. Cherchago, P. Hitzler, and S. Hölldobler, *Decidability under the well-founded semantics*, Proceedings of the First International Conference on Web Reasoning and Rule Systems, RR 2007 (M. Marchiori, J. Z. Pan, and C. de Sainte Marie, eds.), Lecture Notes in Computer Science, vol. 4524, Springer Verlag, 2007, pp. 269–278.

[Cla87] K. L. Clark, *Negation as failure*, Readings in nonmonotonic reasoning, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1987, pp. 311–325.

[CSW95] W. Chen, T. Swift, and D.S. Warren, *Efficient top-down computation of queries under the well-founded semantics*, Journal of Logic Programming **24** (1995), no. 3, 161–199.

[Dix95a] J. Dix, *A classification theory of semantics of normal logic programs: I. Strong Properties*, Fundamenta Informaticae **22** (1995), no. 3, 227–255.

[Dix95b] J. Dix, *A classification theory of semantics of normal logic programs: II. Weak Properties*, Fundamenta Informaticae **22** (1995), no. 3, 257–288.

[EFLP00] T. Eiter, W. Faber, N. Leone, and G. Pfeifer, *Declarative problem solving using the DLV system*, Logic-based artificial intelligence (Norwell, MA, USA), Kluwer Academic Publishers, 2000, pp. 79–103.

[EG97] T. Eiter and G. Gottlob, *Expressiveness of stable model semantics for disjunctive logic programs with functions*, Journal of Logic Programming **33** (1997), no. 2, 167–178.

[Fit91] M. C. Fitting, *Well-founded semantics, generalized*, Logic Programming, Proceedings of the 1991 International Symposium (V. Saraswat and K. Ueda, eds.), The MIT Press, 1991, pp. 71–84.

[Fit02] ———, *Fixpoint semantics for logic programming: a survey*, Theoretical Computer Science **278** (2002), no. 1-2, 25–51.

[FL94] M. Freund and D. Lehmann, *Nonmonotonic Reasoning: From Finitary Relations to the Infinitary Case*, Studia Logica **53** (1994), no. 2, 161–201.

[FLM90] M. Freund, D. Lehmann, and D. Makinson, *Canonical extensions to the infinite case of finitary nonmonotonic inference relations*, Proceedings of the Workshop on Nonmonotonic Reasoning (G. Brewka and H. Freitag, eds.), Arbeitspapiere der GMD, vol. 443, 1990, pp. 133–138.

[Fre93] M. Freund, *Supracompact inference operations*, Studia Logica **52** (1993), 457–481.

[GL88] M. Gelfond and V. Lifschitz, *The stable model semantics for logic programming*, Proceedings of the Fifth International Conference on Logic Programming (R. A. Kowalski and K. Bowen, eds.), The MIT Press, 1988, pp. 1070–1080.

[HS99] P. Hitzler and A. K. Seda, *Characterizations of classes of programs by three-valued operators*, Proceedings of the 5th International Conference on Logic Programming and Non-Monotonic Reasoning, LPNMR'99, El Paso, Texas, USA (M. Gelfond, Nicola Leone, and Gerald Pfeifer, eds.), Lecture Notes in Artificial Intelligence, vol. 1730, Springer, 1999, pp. 357–371.

[HW02] P. Hitzler and M. Wendt, *The well-founded semantics is a stratified Fitting semantics*, Proceedings of the Twentyfifth Annual German Conference on Artificial Intelligence, KI2002 (M. Jarke, J. Koehler, and G. Lakemeyer,

eds.), Lecture Notes in Artificial Intelligence, vol. 2479, Springer Verlag, 2002, pp. 205–221.

[HW05] ———, *A uniform approach to logic programming semantics*, Theory and Practice of Logic Programming **5** (2005), no. 1–2, 123–159.

[KLM90] S. Kraus, D. Lehmann, and M. Magidor, *Nonmonotonic reasoning, preferential models and cumulative logics*, Artificial Intelligence **44** (1990), 167–207.

[Llo87] J. W. Lloyd, *Foundations of logic programming*, Springer Verlag, Berlin, 1987.

[LT94] V. Lifschitz and H. Turner, *Splitting a logic program*, Proceedings of the Eleventh International Conference on Logic Programming (Cambridge, MA, USA), MIT Press, 1994, pp. 23–37.

[Mak89] D. Makinson, *General theory of cumulative inference*, Non-Monotonic Reasoning (M. Reinfrank et al., ed.), Lecture Notes on Artificial Intelligence, vol. 346, 1989, pp. 1–18.

[Mak93] ———, *General Patterns in Nonmonotonic Reasoning*, Handbook of Logic in Artificial Intelligence and Logic Programming (D. M. Gabbay, C.J. Hogger, and J.A. Robinson, eds.), vol. 3, Oxford University Press, 1993, pp. 35–110.

[Mil99] R.S. Milnikel, *Nonmonotonic logic: a monotonic approach*, Ph.D. thesis, Cornell University, 1999.

[MT91] W. Marek and M. Truszczynski, *Autoepistemic logic*, Journal of the ACM **38** (1991), no. 3, 588–619.

[NS96] I. Niemela and P. Simons, *Efficient implementation of the well-founded and stable model semantics*, Joint International Conference and Symposium on Logic Programming, 1996, pp. 289–303.

[Prz88] T. Przymusinski, *On the declarative semantics of deductive databases and logic programs*, Foundations of Deductive Databases and Logic Programming (Jack Minker, ed.), Morgan Kaufmann, Los Altos, CA, USA, 1988, pp. 193–216.

[Ros99] R. Rosati, *Towards first-order nonmonotonic reasoning*, Proceedings of the 5th International Conference on Logic Programming and Non-Monotonic Reasoning, LPNMR'99, El Paso, Texas, USA (M. Gelfond, N. Leone, and G. Pfeifer, eds.), Lecture Notes in Artificial Intelligence, vol. 1730, Springer, 1999, pp. 332–346.

[RSS+97] P. Rao, K.F. Sagonas, T. Swift, D.S. Warren, and J. Freire, *XSB: A system for effciently computing WFS*, Logic Programming and Non-monotonic Reasoning, 1997, pp. 431–441.

[Sch95] J. S. Schlipf, *The expressive powers of the logic programming semantics*, Selected papers of the 9th annual ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems (Orlando, FL, USA), Academic Press, Inc., 1995, pp. 64–86.

[SNV95] V. S. Subrahmanian, D. Nau, and V. Vago, *Wfs + branch and bound = stable models*, IEEE Transactions on Knowledge and Data Engineering **7** (1995), no. 3, 362–377.

[SS97] J. Seitzer and J. S. Schlipf, *Affordable classes of normal logic programs*, Logic Programming and Non-monotonic Reasoning, 1997, pp. 92–111.

[Syr01] T. Syrjänen, *Omega-restricted logic programs*, Proceedings of the Sixth International Conference on Logic Programming and Nonmonotonic Reasoning, LPNMR 01, 2001.

[vG89] A. van Gelder, *The alternating fixpoint of logic programs with negation*, Proceedings of the 8th ACM Symposium on Principles of Database Systems, ACM Press, 1989, pp. 1–10.

[vGRS91] A. van Gelder, K. A. Ross, and J. S. Schlipf, *The well-founded semantics for general logic programs*, Journal of the ACM **38** (1991), no. 3, 620–650.