

KATHOLIEKE UNIVERSITEIT LEUVEN
FACULTEIT TOEGEPASTE WETENSCHAPPEN
DEPARTEMENT COMPUTERWETENSCHAPPEN
Celestijnenlaan 200A – B-3001 Heverlee

SEMANTICS OF LOGIC PROGRAMS WITH AGGREGATES

Promotoren:
Prof. dr. ir. Maurice Bruynooghe
Prof. dr. Marc Denecker

Proefschrift voorgedragen tot
het behalen van het doctoraat
in de toegepaste wetenschappen

door

Nikolay PELOV

April, 2004



KATHOLIEKE UNIVERSITEIT LEUVEN
FACULTEIT TOEGEPASTE WETENSCHAPPEN
DEPARTEMENT COMPUTERWETENSCHAPPEN
Celestijnenlaan 200A – B-3001 Heverlee

SEMANTICS OF LOGIC PROGRAMS WITH AGGREGATES

Jury:
Prof. E. Aernoudt, voorzitter
Prof. dr. ir. Maurice Bruynooghe, promotor
Prof. dr. Marc Denecker, promotor
Prof. dr. Danny De Schreye
Prof. dr. ir. Ronald Cools
Prof. Mirosław Truszczyński (Univ. of Kentucky, USA)
Prof. Michael Gelfond (Texas Tech University, USA)

Proefschrift voorgedragen tot
het behalen van het doctoraat
in de toegepaste wetenschappen

door

Nikolay PELOV

U.D.C. 681.3*I23

April, 2004

© Katholieke Universiteit Leuven - Faculteit Toegepaste Wetenschappen
Kasteelpark Arenberg 1, B-3001 Heverlee, Belgium

Alle rechten voorbehouden. Niets uit deze uitgave mag worden vermenigvuldigd en/of openbaar gemaakt worden door middel van druk, fotocopie, microfilm, elektronisch of op welke andere wijze ook zonder voorafgaande schriftelijke toestemming van de uitgever.

All rights reserved. No part of the publication may be reproduced in any form by print, photoprint, microfilm or any other means without written permission from the publisher.

D/2004/7515/22

ISBN 90-5682-483-X

Semantics of Logic Programs with Aggregates

Nikolay Pelov

Department of Computer Science, K.U.Leuven
Celestijnenlaan 200A, B-3001 Heverlee, Belgium

Abstract

Aggregates are functions that take sets as arguments. Examples are the function that maps a set to the number of its elements or the function which maps a set to its minimal element. Aggregates are frequently used in relational databases and have many applications in combinatorial search problems and knowledge representation. Aggregates are of particular importance for several extensions of logic programming which are used for declarative programming like Answer Set Programming, Abductive Logic Programming, and the logic of inductive definitions (ID-Logic). Aggregate atoms not only allow a broader class of problems to be represented in a natural way but also allow a more compact representation of problems which often leads to faster solving times.

Extensions of specific semantics of logic programs with, in many cases, specific aggregate relations have been proposed before. The main contributions of this thesis are: (i) we extend all major semantics of logic programs: the least model semantics of definite logic programs, the standard model semantics of stratified programs, the Clark completion semantics, the well-founded semantics, the stable models semantics, and the three-valued stable semantics; (ii) our framework admits arbitrary aggregate relations in the bodies of rules.

We follow a denotational approach in which a semantics is defined as a (set of) fixpoint(s) of an operator associated with a program. The main tool of this work is Approximation Theory. This is an algebraic theory which defines different types of fixpoints of an approximating operator associated with a logic program. All major semantics of a logic program correspond to specific types of fixpoints of an approximating operator introduced by Fitting. We study different approximating operators for aggregate programs and investigate the precision and complexity of the semantics generated by them. We study in detail one specific operator which extends the Fitting operator and whose semantics extends the three-valued stable semantics of logic programs without aggregates. We look at algorithms, complexity, transformations of aggregate atoms and programs, and an implementation in XSB Prolog.

Acknowledgements

The material in this thesis is a result of the research which I started exactly three years ago. There are two people without whom this thesis would not have been possible: my two supervisors Maurice Bruynooghe and Marc Denecker. Their continuous support, feedback, flow of new ideas, and trust in me is simply amazing and their help stretches far beyond research. They were always with me in the, otherwise lonely, land of aggregate programs. Thank you!

An important source of inspiration for my work has been the research of Ilkka Niemelä and his students on the SMODELS system. He was also one of the first people to introduce me to the area of non-monotonic reasoning and Answer Set Programming. Part of my research on the stable semantics of disjunctive logic programs was done during a visit to the University of Kentucky where Mirek Truszczyński gave me a very warm welcome and we spend many hours in discussions. I appreciate very much all his attention.

I am very grateful to Maurice Bruynooghe, Marc Denecker, Danny De Schreye, Ronald Cools, Mirosław Truszczyński, and Michael Gelfond who kindly accepted to be members of the jury. I know that reading the following pages may not always have been easy and I really appreciate their efforts. Maurice Bruynooghe deserves a particular acknowledgement for carefully reading every single formula and proof.

Many thanks go to past and present office-mates (Bert Van Nuffelen, Emmanuel De Mot, David Gilis and Maarten Mariën) and to all other members of the DTAI group for providing a creative and pleasant working environment. The support of the secretary of the DTAI group Karin Michiels and the system group are particularly appreciated as they made my stay at the department hassle free.

On a personal level, I am indebted to my wife Nevena for her support during the first years of my PhD, for going with me trough some very difficult times, and for constantly encouraging me to continue my work. Several other people gave me a lot of emotional support and deserve my deepest gratitude: Henk Vandecasteele, Nancy Mazur, Alexander Serebrenik, Ivan & Nadia Markovsky, Jesus Portilla, Caroline Macé, Nele Phillips and especially Hugo Litaer. I am also very grateful to several family members and old friends who visited me in the past years, cheered me up, and spend a quality time together: my parents, Cade, Vania & Vlado.

The research in this thesis has been supported by the following projects and grants: the GOA project LP+, Research Council grant DB/02/44, the FWO project "Theory of nonmonotone Inductive Definitions and its use for building Knowledge-based systems", the "Inductive Knowledge Bases" project and the European Working group on Answer Set Programming (WASP).

March, 2004
Heverlee, Belgium

Contents

1 Introduction **1**
- 1.1 Logic Programming and Declarative Problem Solving 1
- 1.2 Aggregates 3
  - 1.2.1 Programs with Recursive Aggregates 4
  - 1.2.2 Overview of Existing Work 7
  - 1.2.3 Overview of Results 8
  - 1.2.4 Methodology 11
- 1.3 Overview of the Thesis 12

2 Technical Background **13**
- 2.1 Lattice Theory 13
  - 2.1.1 Fixpoint Theory of Monotone Functions 14
- 2.2 Computational Complexity 16
- 2.3 First-order Logic 18
- 2.4 Constraint Domains 20
- 2.5 Logic Programming 21
- 2.6 Approximation Theory 23
  - 2.6.1 Bilattices 23
  - 2.6.2 Approximating Operators 25
  - 2.6.3 Consistent Approximations 27
  - 2.6.4 Precision of Approximations 28
  - 2.6.5 Stratification of Operators 29
  - 2.6.6 Approximations in Logic Programming 31

3 Aggregate Programs and Two-valued Semantics **35**
- 3.1 Aggregates 36
  - 3.1.1 Standard Aggregates 37
  - 3.1.2 Derived Aggregate Relations 38
  - 3.1.3 Monotone Aggregates 38
- 3.2 Aggregate Atoms and Programs 41
  - 3.2.1 Semantics 42

3.2.2 Aggregate Programs 44
3.3 Two-valued Semantics 44
3.3.1 Definite Aggregate Programs 44
3.3.2 Stratified Aggregate Programs 47
3.3.3 Supported Models 49
3.4 Related Work 52
3.5 Conclusions 53

4 Three-Valued Stable Semantics of Aggregate Programs 55
4.1 Ultimate Three-Valued Stable Semantics 56
4.2 Three-Valued Stable Model Semantics 58
4.2.1 Aggregate Programs with Finite Multisets 61
4.2.2 Splitting Theorem 63
4.2.3 Related Work 64
4.3 The Three-valued Stable Semantics based on the Ultimate Approximating Aggregate 65
4.3.1 Examples 66
4.3.2 Definite and Weakly Stratified Aggregate Programs 67
4.3.3 Related Work 69
4.4 Stable Semantics of Disjunctive Aggregate Programs 71
4.4.1 Preliminaries 71
4.4.2 Non-deterministic Operators 72
4.4.3 Immediate Consequence Operators of Disjunctive Programs 72
4.4.4 Definite Disjunctive Programs 75
4.4.5 Approximating Operators and Stable Models 77
4.4.6 Comparison with Other Semantics 81
4.5 Conclusions 82

5 The Three-valued Stable Semantics Based on the Ultimate Approximating Aggregate 85
5.1 Ultimate Approximating Aggregates 86
5.1.1 Algebraic Properties 86
5.1.2 Algorithms 89
5.1.3 Complexity 96
5.2 Program Transformation 97
5.3 Transformations of Aggregate Atoms 98
5.3.1 Complement and Negation 99
5.3.2 Aggregates as Quantifiers 99
5.3.3 Derived Aggregate Relations 100
5.3.4 Extrema Predicates 102
5.3.5 Summation 103
5.3.6 General Transformation 105
5.4 Implementing the Well-founded Semantics in XSB Prolog 109

5.4.1 Incremental Aggregate Functions 113
5.5 Example 114
5.6 Conclusions 117

6 Propositional Aggregate Programs **119**
6.1 Preliminaries 119
6.1.1 Three-Valued Stable Semantics 121
6.2 Complexity of the Semantics 121
6.2.1 Proofs of Membership 125
6.2.2 Proofs of Hardness 127
6.3 Related Work 129
6.3.1 Aggregates and Constraint Programming 129
6.3.2 Weight Constraint Rules 130
6.3.3 Other 135
6.3.4 Complexity 136
6.4 Conclusions 136

7 Conclusions **137**

Chapter 1

Introduction

1.1 Logic Programming and Declarative Problem Solving

Logic programming (Lloyd, 1987) is a research area that studies declarative programming languages that are based on a formal logic. The landmark of this area is the development of the Prolog language. Its execution scheme is based on the resolution procedure of first-order logic. Originally, the language was restricted to Horn clauses which can be written as implications with a single positive literal in the consequent and no negative literals in the antecedent. The SLD resolution can be used to derive positive consequences of a logic program.

For many practical applications it is useful not only to be able to derive negative consequences from a program but also to allow a negation to be used in the bodies of rules. This operator had a procedural semantics of negation as finite failure (Clark, 1978).

The negation operator in logic programming initiated a large amount of research on semantics foundations and proof procedures. We briefly recall the more important semantics. For an extensive overview, we refer the reader to (Apt and Bol, 1994). The first semantics of negation was given by classical models of the program completion (Clark, 1978). This is a first-order logic theory in which the combination of all rules of some predicate are considered as the *if-and-only-if* definition of this predicate. The models of this theory were later characterized as supported models and fixpoints of the immediate consequence operator T_P (Apt et al., 1988). An important milestone was the use of three-valued logic to define semantics of logic programs. Using Kripke's strong three-valued logic, Fitting (Fitting, 1985) defined an immediate consequence operator that is monotone and its least fixpoint is proposed as the intended meaning of the program. This semantics was further refined by Kunen (Kunen, 1987) who considered three-valued

models of the program completion. The two most influential semantics of logic programming are the stable models semantics (Gelfond and Lifschitz, 1988) and the well-founded semantics (Van Gelder et al., 1991). The second semantics is similar to the Kripke-Kleene semantics of Fitting (Fitting, 1985) in that every logic program has a unique three-valued well-founded model. This model is in general more precise than the Kripke-Kleene model. Its main advantage is that it can model inductive definitions. The stable semantics is similar to Clark's completion semantics (Clark, 1978) in the respect that a logic program may have zero, one, or any number of stable models (which are two-valued models). In fact stable models are a subset of the models of the program completion. The stable semantics was later extended to the answer set semantics for programs with two types of negation and disjunction in the head (Gelfond and Lifschitz, 1991). The two semantics have been shown to be well-suited for knowledge representation and non-monotonic reasoning (Baral, 2003; Baral and Gelfond, 1994; Gelfond, 2002; Lifschitz, 2002).

Initially, only programs with a single answer set were considered well-behaved (Gelfond and Lifschitz, 1988; Gelfond and Lifschitz, 1991). The advantages of having multiple stable models were recognized much later (Marek and Truszczyński, 1999; Niemelä, 1999). They allow the possibility to encode constraint satisfaction problems (CSP) and combinatorial optimization problems that have multiple solutions (Niemelä, 1999). This resulted in the area of Answer Set Programming (ASP) (Marek and Truszczyński, 1999; Niemelä, 1999). This is a novel paradigm for declarative programming in which the answers of a logic program are encoded as a set of models in contrast to traditional logic programming where answers are given as a set of answer subsitutions. The success of ASP is also to a large extend due to a number of efficient systems for computing answer sets like SMODELS (Simons et al., 2002), DLV (Eiter et al., 2000), and CMODELS-2 (Lierler and Maratea, 2004).

Abductive logic programming (Kakas et al., 1998) is another framework for declarative programming which is closely related to answer set programming. It distinguishes a set of open predicate A called *abducibles*, a logic program P defining a set of rules, and a set of *integrity constraints* $\mathcal{IC}$. A solution to an abductive problem is a subset Δ of the abducible atoms such that $P \cup \Delta \models_{SEM} \mathcal{IC}$ where the entailment relation $\models_{SEM}$ depends on the choice of a semantics SEM. Besides the conceptual difference between logic programming under the answer set semantics and abduction, there is also a difference in how answers are computed. While ASP systems compute models of propositional logic programs obtained after a grounding process, abductive systems follow the more traditional approach in logic programming by developing top-down resolution based proof procedures (Denecker and De Schreye, 1998; Fung and Kowalski, 1997; Kakas et al., 2000; Kakas et al., 2001). The strength of the recent abductive solvers (Kakas et al., 2000; Kakas et al., 2001; Van Nuffelen, 2004) is the integration of constraint solvers in a similar way as in Constraint Logic Programming. For constraint satisfaction problems

these solvers can reduce, often in a backtrack free manner, a declarative problem specification to a constraint store produced by a CLP program modeling the same problem. Another advantage of abductive solvers over ASP systems is that they work with a non-ground first-order specification and do not suffer from the blow up of the size of ground logic programs. So, in general they scale much better. They are better suited to solve planning and scheduling problems which involve large resource amounts. The main disadvantage of abductive systems is that they are incomplete and do not handle well programs with recursive definitions.

Another formalism which fits in the Answer Set Programming paradigm is the recent extension of classical logic with inductive definitions (ID-Logic) (Denecker, 2000). It is similar to abductive logic programming because it also considers open predicates, definitions, and integrity constraints. One of the goals of this logic is to address a critique on ambiguity of the interpretation of the answer set semantics (Denecker, 2004). Unlike the interpretation of the negation operator under the stable semantics as default negation, in ID-Logic it is interpreted as classical negation. The underlying principle of this logic is to interpret programs as inductive definitions and not as non-monotonic theories as is the case with the stable semantics (Gelfond and Lifschitz, 1988). The semantics of ID-Logic is based on the well-founded semantics of logic programs (Van Gelder et al., 1991) which Denecker argues, formalizes the principle of generalized inductive definitions (Denecker, 1998; Denecker et al., 2001a).

1.2 Aggregates

Aggregates are functions which take sets or multisets as arguments. Examples of aggregate functions are CARD which return the number of elements in the input multiset and SUM which returns the sum of the elements. In our work we study not only aggregate functions but also *aggregate relations*. These are binary relations where the first argument ranges over multisets and the second argument over simple values. Aggregate relations can be used to model partial aggregate functions (not defined for all multisets). For example the aggregate function AVG which returns the average of a multiset is not defined for the empty multiset. Aggregate relations can also model aggregate functions which return a set of elements. For example, a subset of a partially ordered set may have several minimal elements.

Aggregates are frequently used in relational databases and the SQL query language. They are also common in many combinatorial search problems. There are several reasons why extending logic programming with aggregates is useful. Using aggregates often allows a more succinct representation of problems (Simons et al., 2002). In many cases this also leads to faster solving times (Dell'Armi et al., 2003a). This is similar to the use of global constraints (which are the analog of aggregates) in constraint programming to obtain more efficient constraint propagation (Beldiceanu and Contejean, 1994). A more important reason is to be

able to model and solve problems that involve a type of quantitative reasoning which is very cumbersome to model in a language based on first-order logic.

In a logical language, aggregate relations are modelled as second-order predicates with a fixed interpretation. The input multiset of the aggregates is defined by a pair of a set and a function. An *aggregate atom* has the form

$$\text{R}(\lambda\mathbf{x}.\ u(\mathbf{x}), \{\mathbf{x} \mid \varphi(\mathbf{x})\}, t)$$

where R is a name of an aggregate relation, $\lambda\mathbf{x}.\ u(\mathbf{x})$ is a lambda expression whose interpretation is a function, $\{\mathbf{x} \mid \varphi(\mathbf{x})\}$ is a set expression whose interpretation is a set and t is a term. The input multiset of the aggregate is obtained by applying the function defined by the lambda expression $\lambda\mathbf{x}.\ u(\mathbf{x})$ to every element in the set defined by the set expression $\{\mathbf{x} \mid \varphi(\mathbf{x})\}$.

Aggregate atoms are very similar to generalized quantifiers (Lindström, 1966). One difference between the two concepts is that generalized quantifiers do not take lambda expressions as an input but instead may take several set expressions as an input. The more important difference is that unlike aggregates which have a fixed interpretation, generalized quantifiers are interpreted with a class of structures that is closed under isomorphisms. The fact that aggregates have a fixed interpretation is important for our study. For example, we present algorithms for computing aggregates on partially defined sets that exploit specific properties of the aggregate functions and of the domain of interpretation. In this respect programs with aggregates are very similar to constraint logic programming (Jaffar and Maher, 1994) and aggregate relations are closely related to global constraints (Beldiceanu and Contejean, 1994). In logic programming, the $setof/3$ and $findall/3$ predicates for computing the set of all solutions of a goal can be considered as an early predecessor of aggregates. In fact, they can be used to compute the value of a set expression $\{\mathbf{X} \mid \varphi(\mathbf{X})\}$ as a list of tuples S using the query $findall(\mathbf{X}, \varphi(\mathbf{X}), S)$.

1.2.1 Programs with Recursive Aggregates

The main challenge of our work was defining semantics of programs that contain recursion over aggregate atoms. This is the case when the membership of the elements in the input set of the aggregate atom depends on the value of the aggregate on the set. The following example contains a program that has recursion over aggregation.

Example 1.1 (Party Invitation I) A number of people are invited to a party. A person p will accept the invitation if and only if at least k of his (her) friends also accept the invitation. The friendship relation is given by a binary predicate $friend(X, Y)$ meaning that Y is a friend of X. The input also consists of a relation $thr(X, T)$ giving the lower bound T on the number of friends of X. The problem is modeled by the program consisting of the following single rule:

$$accept(X) \leftarrow thr(X, T), \text{CARD}_{\geq}(\{Y \mid friend(X, Y), accept(Y)\}, T).$$

The aggregate relation $\text{CARD}_{\geq}$ is obtained by the composition of the aggregate function CARD which returns the number of elements in the input multiset and the relation $\geq$ on natural numbers. So, $(M, d) \in \text{CARD}_{\geq}$ if and only if the number of elements in the multiset M is greater than or equal to d.

Note that $accept(X)$ appears in the condition of the set expression defining the input of the aggregate relation. □

Although problems with recursive aggregates are scarce, they come from diverse areas like computer science, mathematics, economics, and natural language.

One can define models for programs without recursion over aggregation using a well-known technique from logic programming known as *stratification* (Apt et al., 1988; Mumick et al., 1990). The rules in a program are partitioned in several strata such that the rules of the predicates used in a set expression of an aggregate atom are in a strictly lower level than the rule with the aggregate atom. The model of a stratified aggregate program is computed stratum by stratum. When computing the model of a given stratum, the interpretation of the predicates defining the input sets of all aggregate atoms in this stratum is completely defined and one can simply evaluate the aggregate atom. The process of computing models of stratified aggregate programs is very similar to the use of aggregates in the SQL language where every query may use either tables or views, the latter corresponding to a program at a lower stratum.

When studying examples of programs with recursion over aggregation we observed two principles which can be used to define a semantics. The first one is the principle of induction. An inductive definition is a a recipe for defining a set or a concept by a constructive process. The most well-known type of inductive definitions are monotone inductive definitions. A common way to define a monotone inductive definition is as a monotone function $f : \wp(X) \to \wp(X)$ where X is a set and $\wp(X)$ is the set of all subsets of X. The set which is inductively defined by f is obtained by an iterative computation which starts by applying f to the empty set and repeatedly applying f to the result of the last step. Because the function f is monotone we obtain a sequence of growing sets. After a (possibly transfinite) number of steps we reach a set S which cannot be improved any further. This is the set which is defined inductively by f. According to the fixpoint theory of monotone functions, S is also the smallest set such that $f(S) = S$, i.e. S is the least fixpoint of f.

Logic programs can be regarded as inductive definitions by associating an immediate consequence operator T_P with a program P which maps between interpretations (van Emden and Kowalski, 1976). For programs without negation in the body, called definite logic programs, the T_P operator is monotone. The set which is inductively defined by P is the least fixpoint of T_P. A standard example of a monotone inductive definition is the transitive closure of a graph.

Example 1.2 Let *edge* be a binary relation defining a directed graph: $(a, b) \in$ *edge* if and only if there is an edge from node a to node b. Consider the following

program defining the binary predicate *path*:

$$path(X,Y) \leftarrow edge(X,Y).$$
$$path(X,Y) \leftarrow edge(X,Z), path(Z,Y).$$

This is a definite logic program. In the least fixpoint of the associated T_P operator, $path(a,b)$ is true if there is a path between the nodes a and b. □

One of our results is a definition of the class of *definite aggregate programs* which also have a monotone immediate consequence operator and its least fixpoint defines the model of such program. For example, the program from Example 1.1 is a definite aggregate program, although we argue later that it should not be considered as a monotone inductive definition.

There are also examples of programs with recursive aggregates which should be considered as inductive definitions but for which the immediate consequence operator is not monotone. This is the case, for example, for the formulation of the problem of finding the shortest path between two nodes in a graph based on Dijkstra's algorithm. Denecker (Denecker, 1998; Denecker et al., 2001a) argues that the well-founded semantics of logic programs (Van Gelder et al., 1991) formalizes the principle underlying non-monotone inductive definitions. In this work we define several extensions of this semantics to aggregate programs.

The second way of interpreting programs with recursion over aggregation is not as inductive definitions but as *if-and-only-if* definitions in first-order logic. Under this view, a program may have several models which are considered as valid solutions. Such models do not have to be obtained by a constructive process as is the case for inductive definitions and can be self supported. The foundations of this semantics are provided by the program completion (Clark, 1978) or supported models (Apt et al., 1988). The party invitation problem from Example 1.1 is one example of a problem which should be considered as an *if-and-only-if* definition.

Example 1.3 The meaning of the program P from Example 1.1 is given by the following first-order theory $comp(P)$ known as the *completion* of P (Clark, 1978):

$$comp(P) = \{$$
$$\forall X.\ accept(X) \leftrightarrow thr(X, T \wedge \text{CARD}_{\geq}(\{Y \mid friend(X,Y) \wedge accept(Y)\}, T)\}.$$

Any interpretation of *accept* which satisfies this formula is considered as a solution of the problem.

Consider a situation where a person a will accept if and only if a person b will accept and vice versa. This is modeled by the following input:

$$D = \begin{cases} friend(a,b). & thr(a,1). \\ friend(b,a). & thr(b,1). \end{cases}$$

The first-order theory $comp(P) \cup D$ have two models. In one of them both $accept(a)$ and $accept(b)$ are false, thus none of the persons accept the invitation. In the other model both $accept(a)$ and $accept(b)$ are true, thus both persons accept the invitation. □

1.2.2 Overview of Existing Work

The existing work of semantics of aggregate programs follows more or less the same line of development as the semantics of logic programming. Mumick et al. (Mumick et al., 1990) define semantics for several classes of aggregate programs: least model semantics of monotonic aggregate programs extending the class of definite logic programs (van Emden and Kowalski, 1976); perfect model semantics of stratified and monotonic stratified programs extending the class of stratified logic programs (Przymusinski, 1988); and the perfect model semantics of group stratified programs which are similar to locally stratified programs (Przymusinski, 1988). A unique line of research in the semantics of aggregate programs which does not have a parallel in logic programming is to consider the least fixpoint semantics of programs with monotone immediate consequence operator under non-standard orders (Ross and Sagiv, 1997; Van Gelder, 1993). These orders are based on partial orders on the domain of the aggregation and may not form complete lattices. Monotone aggregate functions were also studied in the context of partial order programming (Osorio et al., 1999). A translation of partial order programs with aggregates to normal logic programs without aggregates is provided (Osorio and Jayaraman, 1999). However, the authors observe that the well-founded semantics is sometimes too weak and study different extensions (Dix et al., 2001).

The previous extensions of the well-founded semantics (Van Gelder et al., 1991) to aggregate programs (Kemp and Stuckey, 1991; Van Gelder, 1992) are too weak and often leave too many undefined atoms. (Ganguly et al., 1995) studied the well-founded semantics of programs containing only MIN and MAX aggregates. An interesting approach which uses three-valued multisets and aggregate functions and relations defined on three-valued multisets is the extension of the valid semantics to aggregate programs (Sudarshan et al., 1993). (Van Nuffelen and Denecker, 2000) report on extending an abductive solver to reason with aggregates under the semantics of Van Gelder(Van Gelder, 1992).

Most of the proposals for a stable semantics of aggregate programs (Gelfond, 2002; Kemp and Stuckey, 1991; Dell'Armi et al., 2003b) are based on a definition of a program reduct which treats aggregate atoms as negative literals. Although this approach is a straightforward extension of the stable semantics of normal logic programs (Gelfond and Lifschitz, 1988), programs may have non-minimal stable models and these semantics cannot model some problems with recursive aggregation. The most elaborate approach is the stable semantics of weight constraint rules (Simons et al., 2002) implemented by the SMODELS system. However, the syntax is limited only to specifying lower and upper bounds of the sum of the

input set and extension to other aggregates is not obvious.

(Simons et al., 2002) also introduced the use of aggregate-like expressions in the heads of rules. This has been further studied and generalized by (Marek and Remmel, 2002; Marek and Remmel, 2004; Marek et al., 2004). Although the syntax of these extensions generalizes the syntax of disjunctive logic programs, the semantics (intentionally) admits non-minimal models unlike the stable semantics of disjunctive logic programs (Gelfond and Lifschitz, 1991; Przymusinski, 1991).

Other approaches do not define directly well-founded or stable semantics of aggregate programs. Instead they "implement" aggregates within the program itself (Baral, 2003; Van Gelder, 1992; Zaniolo and Wang, 1999). This is done by encoding the input set of the aggregates as lists or auxiliary predicates and then defining the aggregates as predicates.

1.2.3 Overview of Results

In this thesis we lift the following semantics of logic programs to logic programs with aggregates:

- least model semantics of definite logic programs (van Emden and Kowalski, 1976);
- completion semantics (Clark, 1978) and supported models (Apt et al., 1988);
- standard model semantics of stratified logic programs (Apt et al., 1988);
- stable models semantics (Gelfond and Lifschitz, 1988);
- well-founded semantics (Van Gelder et al., 1991);
- three-valued stable models (Przymusinksi, 1990);
- ultimate three-valued stable semantics (Denecker et al., 2004).

One of the important contributions of our work is that most of the properties and relationships between all these semantics carry over to their extensions to aggregate programs. This is not the case with many of the previous proposals of semantics of aggregate programs or very little attention has been given to this aspect. One type of relationship is that the semantics of a larger class of programs coincides with the semantics of a smaller class of programs. For example, we show that stratified aggregate programs have a single two valued stable model which is also the well-founded model and coincides with the standard model. Tables 1.1, and 1.2 summarize this type of relationships. Every cell in the table represents a class of programs. The first column gives the names of the classes. The second column gives the name of the semantics for logic programs without aggregates which we extend and a reference where they are defined. The last column gives a reference to the section in the thesis which extends the semantics for aggregate

programs. The classes of programs in every row are a strict subset of the classes of programs in the next row. Also the semantics for every row agrees with the semantics in the previous row for the restricted class of programs. The same holds for columns: the semantics of aggregate programs which we define coincides with the semantics in the previous column for logic programs without aggregates.

programs	without aggregates	with aggregates
definite	least fixpoint (van Emden and Kowalski, 1976)	Section 3.3.1
stratified	standard model (Apt et al., 1988)	Section 3.3.2
general	well-founded (Van Gelder et al., 1991) stable models (Gelfond and Lifschitz, 1988) three-valued stable models (Przymusinksi, 1990)	Section 4.2

Table 1.1: Relationship between the Semantics I

programs	without aggregates	with aggregates
definite	least fixpoint (van Emden and Kowalski, 1976)	Section 3.3.1
monotone	least fixpoint	(Kemp and Stuckey, 1991)
general	ultimate three-valued stable semantics (Denecker et al., 2004)	Section 4.1

Table 1.2: Relationships between the Semantics II

The class of monotone programs on Table 1.2 is the class of programs which have a monotone immediate consequence operator and is strictly larger than the class of definite (aggregate) programs. The semantics of definite and stratified aggregate programs have been studied before as monotonic and stratified monotonic programs (Mumick et al., 1990). We argue that our definition is simpler and more natural than the one in (Mumick et al., 1990). The other reason to redefine these semantics is to show that one of the three-valued stable semantics for aggregate programs which we define extends the semantics of these two classes in the same way as the three-valued stable semantics of non-aggregate programs.

We do not define explicitly a well-founded and stable semantics. Instead we define a class of three-valued stable models in the sense of Przymusinski (Przymusinksi, 1990). The well-founded model is the least model in this class and the stable models are the set of total three-valued stable models. Also, we do not define a single three-valued stable semantics but a whole family, all of them extending the three-valued stable semantics of logic programs (Przymusinksi, 1990) (and consequently the stable (Gelfond and Lifschitz, 1988) and well-founded (Van Gelder

et al., 1991) semantics). These semantics are parametrized by the way aggregate functions and relations are extended to three-valued aggregate relations on three-valued multisets. Among the semantics in this family we study in detail the most precise one. There are several arguments why this semantics is interesting:

- This semantics has the most precise well-founded model and a larger number of total stable models than any other semantics in this family.
- For most standard aggregate functions and relations the complexity of this semantics remains the same as the complexity of logic programs without aggregates.
- The semantics extends and unifies the semantics of several special classes of aggregate programs including monotonic, stratified, and stratified monotonic programs (Mumick et al., 1990); logic programs with extrema predicates (Ganguly et al., 1995); and, for a large class of programs, the stable semantics of weight constraint rules (Simons et al., 2002).
- Several natural transformations of extrema aggregate relations, like MIN and SUP, which are equivalent in classical logic remain equivalent under this semantics.
- This semantics is precise enough to model correctly all examples with recursive aggregation which we consider except one problem of a monotone inductive definition (Example 4.7). Only the ultimate well-founded semantics is strong enough to model this example because it extends the theory of monotone inductive definitions.

Although we are concerned mostly with the study of the semantics of aggregate programs we provide several results which can be used in a practical implementation. Section 5.1.2 contains algorithms for computing three-valued aggregate relations (used in the three-valued stable semantics) of most standard aggregate relations. In Section 5.4 we discuss one approach for implementing the well-founded semantics in an existing Prolog system. Also, Section 6.3.1 shows how three-valued aggregate relations can be computed using constraint based techniques. This can be used, for example, in an ASP system extended with a constraint solver, e.g. (Pontelli et al., 2004).

Our final contribution is a complexity analysis of all semantics which we define. The complexity results are parametrized by the complexity class containing the problems of computing all aggregate relations (or, for some semantics, their extension to partial aggregate relations) used in the program.

The work contains an extensive collection of examples with recursive aggregates. Some of them (to the best of our knowledge) have not been discussed in the logic programming literature before.

1.2.4 Methodology

We follow a denotational approach for defining semantics in which a (set of) model(s) is defined as a (set of) fixpoint(s) of a suitable operator associated with the logic program. The least model semantics of definite logic programs, the completion semantics, and the standard semantics of stratified programs can all be defined as special types of fixpoints of the immediate consequence operator T_P (van Emden and Kowalski, 1976).

For the well-founded, stable, and, more generally, the three-valued stable semantics we use the framework of *Approximation Theory* (Denecker et al., 2000). This is an algebraic theory for approximating the fixpoints of non-monotone lattice operators. It combines ideas from the alternating fixpoint construction of the well-founded semantics (Van Gelder, 1989) and the setting of bilattices developed by Fitting (Fitting, 1991a; Fitting, 1991b; Fitting, 1993; Fitting, 2002). The central concept in this theory is that of an *approximating operator* $A : L^2 \to L^2$ of an operator $O : L \to L$ where L is a complete lattice. One of the requirements of an approximating operator is that it is monotone in a precision order on the bilattice L^2. By applying the Knaster-Tarski fixpoint theory of monotone operators, A has a least fixpoint (x, y). This fixpoint "approximates" all fixpoints of O in the sense that $x \leq z \leq y$ for every fixpoint z of O. For every approximating operator A of O, Approximation Theory defines a *stable operator* $S_A : L^2 \to L^2$ of O. The fixpoints of this operator are called *stable fixpoints*[1] of A. The stable operator S_A is also monotone and its least fixpoint is called *well-founded fixpoint*.

Very often an operator $O : L \to L$ has many approximating operators and each one of them may have a different set of stable and well-founded fixpoints. The relationship between approximating operators was studied in (Denecker et al., 2004) where the authors define a precision order between them. They show that more precise approximating operators have more precise well-founded fixpoints and a larger number of total stable fixpoints. Also, there exists a most precise approximating operator $U_O : L^2 \to L^2$ of O, called the *ultimate approximating operator*. An important property of this operator is that if O is a monotone operator, the well-founded fixpoint of U_P is total and coincides with the least fixpoint of O.

We use all the latest developments of Approximation Theory, including the characterization of three-valued stable models on consistent approximations (Denecker et al., 2004) and stratification of operators (Gilis and Denecker, 2002; Vennekens et al., 2003).

[1] Stable fixpoints correspond to four-valued stable models (Fitting, 2002).

1.3 Overview of the Thesis

In Chapter 2 we present the necessary technical background for understanding the rest of the results. In Chapter 3 we introduce the concept of aggregates and study several basic properties. We also present the semantics of aggregate programs based on two-valued logic: the least fixpoint semantics of definite aggregate programs, the standard model of stratified aggregate programs and the semantics based on program completion. In Chapter 4, we define several three-valued stable model semantics of aggregate programs based on Approximation theory. One of them is the ultimate three-valued stable semantics of aggregate programs (Denecker et al., 2001b) which extends the ultimate semantics of logic programs (Denecker et al., 2004). This is the most precise semantics in the sense of having the most precise well-founded model and the largest number of total stable models in the framework of approximation theory. We also study a family of semantics which extend the three-valued stable semantics of logic programs (Przymusinksi, 1990). In Chapter 5 we study in more detail the most precise semantics in this family. In the last chapter (Chapter 6) we use a propositional syntax of aggregate programs to study complexity of the various semantics and discuss related work.

Publications and Co-authors

The work on the semantics of aggregate programs is a joint work with Marc Denecker. Only part of the results are published. First appeared the work on the ultimate semantics (Section 4.1) at ICLP'01 (Denecker et al., 2001b). The three-valued stable semantics (Section 4.2) together with a discussion of the relationship with SMODELS (Section 6.3.2) and "Aggregates as Negative Literals" (Section 4.3.3) were published at the LPNMR'04 conference (Pelov et al., 2004). We used the propositional syntax of Chapter 6 to simplify the presentation and the discussion with related work. The translation of aggregate programs to normal logic programs (Section 5.3.6) was presented for a propositional language at the workshop on Answer Set Programming in Messina, Italy (Pelov et al., 2003) again for propositional programs. The algebraic characterization of the stable semantics of disjunctive logic programs (Section 4.4) is a result of the work during my visit of the University of Kentucky, Lexington, USA in collaboration with Mirosław Truszczyński and is submitted for review (Pelov and Truszczyński, 2004).

The thesis does not include the work on proving failure of queries for definite logic programs using XSB Prolog (Pelov and Bruynooghe, 1999) which I did together with Maurice Bruynooghe during my pre-doctoral year; an experimental work on comparing the performance of different ASP systems (Pelov et al., 2000a; Pelov et al., 2000b) which is a joint work with Emmanuel De Mot, Marc Denecker, and Maurice Bruynooghe; and the joint work with Maurice Bruynooghe on extending constraint logic programming with open functions (Pelov and Bruynooghe, 2000).

Chapter 2

Technical Background

In this chapter we present the necessary technical background for understanding the rest of the presentation.

2.1 Lattice Theory

In this section we recall some basic notions from lattice theory and fixpoint theory of monotone functions (Davey and Pristley, 1990).

Definition 2.1 (Partial Order) *Let P be a set. A* partial order *on P is a binary relation $\leq$ on P such that, for all $x, y, z \in P$,*

$$\begin{array}{ll} x \leq x & \textit{(reflexivity)} \\ x \leq y \wedge y \leq x \Rightarrow x = y & \textit{(antisymmetry)} \\ x \leq y \wedge y \leq z \Rightarrow x \leq z & \textit{(transitivity)} \end{array}$$

A *partially ordered set* or *poset* for short is a pair $\langle P, \leq \rangle$ of a set P and a partial order $\leq$ on P. When the partial order $\leq$ is clear from the context we denote a poset simply by P. A partial order relation $\leq$ gives rise to a *strict order* $<$ defined as: $x < y$ if and only if $x \leq y$ and $x \neq y$.

Example 2.1 Let X be a set. The *powerset* $\wp(X)$ of X consists of all subsets of X. The subset inclusion relation $\subseteq$ on $\wp(X)$ is a partial order relation and $\langle \wp(X), \subseteq \rangle$ is a poset. □

A poset P is a *chain* if for all $x, y \in P$, either $x \leq y$ or $y \leq x$. Sometimes, we also say that P is a *totally ordered set*. On the other hand P is an *antichain* if all elements in P are incomparable, i.e., if $x \leq y$ in P then $x = y$.

Let S be a subset of P. We say that an element $a \in S$ is a *maximal* element of S if for all $x \in S$, $a \leq x$ implies $a = x$. Similarly, an element $a \in S$ is a *minimal*

element of S if for all $x \in S$, $x \leq a$ implies $a = x$. The sets of maximal and minimal elements of S are denoted by $\max(S)$ and $\min(S)$ respectively.

A *well-founded order* is a partial order relation $\leq$ on P such that every non-empty subset $S \subseteq P$, S has a minimal element, i.e. $\min(S) \neq \emptyset$. A total well-founded order is called a *well-order*.

An element $a \in S$ is *greatest* (or *top*) if for all $x \in S$, $x \leq a$ and *least* (or *bottom*) if for all $x \in S$, $a \leq x$. The greatest and least elements of a set S are unique, if they exist. Consequently, they are denoted with $\text{TOP}(S)$ and $\text{BOT}(S)$. The top and bottom elements of the whole poset P (if they exist) are denoted with $\top$ and $\bot$.

Let S be a subset of P. We say that an element $a \in P$ is an *upper bound* of S if $x \leq a$ for all $x \in S$. Similarly, $a \in P$ is a *lower bound* of S if $a \leq x$ for all $x \in S$. The sets of upper and lower bounds of S are denoted with $\text{UB}(S)$ and $\text{LB}(S)$ respectively. Note that $\text{UB}(\emptyset) = P$ and $\text{LB}(\emptyset) = P$.

Finally, an element $a \in P$ is a *supremum* of S, denoted as $a = \bigvee S$ if a is the least upper bound of S. Dually, $a \in P$ is an *infimum* of S, denoted with $a = \bigwedge S$, if it is the greatest lower bound of S. Again if the infimum or supremum of a set exist then they are unique. Infima and suprema of two element sets are of special importance. So, we adopt the following notation: $x \vee y$ for the least upper bound of $\{x, y\}$ and $x \wedge y$ for the greatest lower bound of $\{x, y\}$.

Definition 2.2 *Let P be a non-empty poset. If $x \vee y$ and $x \wedge y$ exist for all $x, y \in P$ then P is called a* lattice. *If $\bigvee S$ and $\bigwedge S$ exist for all $S \subseteq P$ then P is called a* complete lattice.

Example 2.2 The powerset lattice $\langle \wp(X), \subseteq \rangle$ from Example 2.1 is an example of a complete lattice. Infimum and supremum of a set C of subsets of X are defined as the intersection and union of the sets in C. □

Example 2.3 Consider the poset $F = \langle fin(\wp(X)), \subseteq \rangle$ of all finite subsets of a set X. If X is finite then F is a complete lattice. However, if X is infinite then F is a lattice but not complete. This is because the union of an infinite number of finite sets may not be finite. □

2.1.1 Fixpoint Theory of Monotone Functions

We now present the fixpoint theory of monotone functions. The result that every monotone function on a complete lattice has a least fixpoint is usually attributed to Knaster and Tarski (Tarski, 1955). However, there are many versions of this theorem and its origins are difficult to trace precisely (Lassez et al., 1982). We give the weakest condition for the existence of least fixpoints of monotone functions. The following results can be found in (Cousout and Cousout, 1979; Markowsky, 1976; Davey and Pristley, 1990).

Definition 2.3 *Let $F\colon P \to Q$ be a map on posets P and Q. We say that F is* monotone *if $x \leq y$ implies $F(x) \leq F(y)$ for all $x, y \in P$ and* anti-monotone *if $x \leq y$ implies $F(y) \leq F(x)$ for all $x, y \in P$.*

Let $F\colon P \to P$ be a self-map on a poset P. An element $x \in P$ is a *fixpoint* of F if $F(x) = x$, *pre-fixpoint* if $F(x) \leq x$ and *post-fixpoint* if $x \leq F(x)$. The sets of fixpoints, pre-fixpoints, and post-fixpoints of F are denoted with $\mathrm{fp}(F)$, $\mathrm{pre}(F)$, and $\mathrm{post}(F)$ respectively. The least and the greatest fixpoints of F (whenever they exist) are denoted with $\mathrm{lfp}(F)$ and $\mathrm{gfp}(F)$ respectively.

An important question in fixpoint theory is the study of fixpoints of monotone functions. The simplest class of posets for which every monotone map has a least fixpoint are the chain-complete posets.

Definition 2.4 *A poset P is* chain-complete *if and only if every chain has a supremum.*

Because the empty set is also a chain and $\bigvee \emptyset = \bot$ then a chain-complete poset always have a bottom element.

Proposition 2.1 *Let $F\colon P \to P$ be a monotone function on a chain-complete set P. Then F has a least fixpoint.*

Chain-completeness is not only sufficient but also a necessary condition for the existence of a least fixpoint of every monotone function.

Theorem 2.2 ((Markowsky, 1976), Theorem 11) *Let P be a poset. Every monotone self-map $F\colon P \to P$ has a least fixpoint if and only if P is chain-complete.*

Another type of poset which we encounter is that of complete semilattices.

Definition 2.5 (Complete Semilattice) *Let P be a chain-complete poset*[1]*. If $\bigwedge S$ exists for all non-empty subsets S of P then P is called a* complete semilattice*.*

A complete semilattice P is almost a complete lattice — it may only lack a top element. In fact, adding a top element to P will make it a complete lattice. However, the opposite is not always true — removing the top element from a complete lattice may not results in a complete semilattice.

Example 2.4 Consider the set $\mathbb{N}^\infty = \mathbb{N} \cup \{\infty\}$. It is a complete lattice under the standard order $\leq$ on the natural numbers. However, $\mathbb{N}$ is not chain-complete (and consequently not a complete semilattice) because $\mathbb{N}$ is a chain but $\bigvee \mathbb{N}$ does not exists. □

[1]The definition of complete semilattice is usually based on a complete partial order instead of a chain-complete poset. However, the two notions are equivalent (Davey and Pristley, 1990, Theorem 8.11).

Let P be a chain-complete poset and $F\colon P \to P$ a monotone function. The least fixpoint of F can also be computed by a transfinite iteration of F starting from the bottom element.

Definition 2.6 *The* ordinal powers of F starting from x *are defined as follows:*

$$\begin{array}{lll} F^0(x) & = x & \\ F^{\alpha+1}(x) & = F(F^\alpha(x)) & \textit{for a successor ordinal } \alpha+1 \\ F^\alpha(x) & = \bigvee_{\beta<\alpha} F^\beta(x) & \textit{for a limit ordinal } \alpha \end{array}$$

The ordinal powers of a monotone function F are used to compute its least fixpoint.

Proposition 2.3 ((Cousout and Cousout, 1979)) *There exists an ordinal* ∞ *such that* $F^\infty(\bot) = \mathrm{lfp}(F)$.

The ordinal ∞ at which we reach the least fixpoint is called the *closure-ordinal* of F.

2.2 Computational Complexity

We present some basic concepts from complexity theory. A classical introduction to the subject is the book by Garey and Johnson (Garey and Johnson, 1979).

An *alphabet* is a finite set Σ of symbols. The set of all finite strings over Σ, including the empty string ϵ is denoted with Σ^*. A *language* $\mathcal{L}$ is a subset of Σ^*.

Definition 2.7 *A* decision problem *is a pair* $\langle \mathcal{L}, \mathcal{Y} \rangle$ *of a language* $\mathcal{L}$ *and a set* $\mathcal{Y} \subseteq \mathcal{L}$ *of* yes-instances. *A string* $s \in \mathcal{L}$ *is called an* instance *of a problem.*

A *class* is a set of decision problems. The *complement* co-$\mathcal{C}$ of a class $\mathcal{C}$ is defined as

$$\text{co-}\mathcal{C} = \{\langle \mathcal{L}, \mathcal{L} - \mathcal{Y} \mid \langle \mathcal{L}, \mathcal{Y} \rangle \in \mathcal{C}\}.$$

The standard model of computation for studying complexity is a Turing machine (TM). This is a finite state machine which consists of a read-write head and an infinite two-way tape. Initially, the head is pointing at the beginning of an input string written on the tape and the machine is in a pre-defined initial state. At every step of the computation, the machine reads the symbol on the tape at the current position of the head. Depending on the current state, the machine writes a new symbol at the current position, changes to a new state, and moves the head to the left or right.

Definition 2.8 *A* Turing machine M *is a tuple* $\langle Q, \Sigma, \delta, q_0, F \rangle$ *where* Q *is a finite set of* states, Σ *is a finite alphabet,* $q_0 \in Q$ *is the* initial state, $F \subseteq Q$ *is a set of* final states, *and* $\delta : (Q - F) \times \Sigma \to Q \times \Sigma \times \{-1, +1\}$ *is a* transition function.

A non-deterministic Turing machine (NDTM) is a Turing machine with a non-deterministic transition function. So, in every step of the computation the machine may have several possible choices how to continue. An input string s is accepted by a NDTM if there exists a computation which ends in a final state.

Definition 2.9 (P and NP) *The class P of polynomial-time computable decision problems consists of all decision problems $\langle \mathcal{L}, \mathcal{Y} \rangle$ for which there exists a deterministic Turing machine which for every instance $s \in \mathcal{L}$ decides in polynomial time in the size of s if $s \in \mathcal{Y}$. The class NP is defined in a similar way but using non-deterministic instead of deterministic Turing machines.*

The complexity of many of the problems discussed in our work belong to classes of the polynomial hierarchy. This is an infinite hierarchy of complexity classes, starting with P and NP, defined by using oracle machines.

An oracle machine is a Turing machine which can make calls to a subroutine — the *oracle* — deciding some fixed problem in constant time. Let $\mathcal{C}$ be a class of problems defined by some resource bounds and $\mathcal{A}$ a class of decision problems. By $\mathcal{C}^{\mathcal{A}}$ we denote the class of problems decidable by a Turing machine with the same resource bounds as $\mathcal{C}$ and an oracle for a problem in A.

Definition 2.10 *The* polynomial hierarchy *is defined as follows:*

$$\Delta_0^p = \Sigma_0^p = \Pi_0^p = P$$
$$\Delta_{k+1}^p = P^{\Sigma_k^p}, \quad \Sigma_{k+1}^p = NP^{\Sigma_k^p}, \quad \Pi_{k+1}^p = \textit{co-}NP^{\Sigma_k^p}$$

In particular, $NP = \Sigma_1^p$ and co-$NP = \Pi_1^p$. A problem is in the k-th level of the polynomial hierarchy if it is in the class Δ_{k+1}^p.

Next, we introduce the notion of complete problems. These are the "hardest" problems for a given class because all other problems can be reduced to them. The following definition makes precise the notion of reduction.

Definition 2.11 *A* polynomial transformation *from a problem $\Pi_1 = \langle \mathcal{L}_1, \mathcal{Y}_1 \rangle$ to a problem $\Pi_2 = \langle \mathcal{L}_2, \mathcal{Y}_2 \rangle$, denoted with $\Pi_1 \propto \Pi_2$, is a function $f \colon \mathcal{L}_1 \to \mathcal{L}_2$ which is computable by a polynomial time Turing machine and for every $x \in \mathcal{L}_1$, $x \in \mathcal{Y}_1$ if and only if $f(x) \in \mathcal{Y}_2$.*

Definition 2.12 (Completeness) *A decision problem Π is* complete *for a complexity class $\mathcal{C}$ of the polynomial hierarchy if $\Pi \in \mathcal{C}$ and $\Pi' \propto \Pi$ for all $\Pi' \in \mathcal{C}$.*

The standard complete problem for the k-th class Σ_k^p of the polynomial hierarchy is the k-QBF problem defined as: is the quantified boolean formula

$$\exists \mathbf{x} \forall \mathbf{y} \ldots F(\mathbf{x}, \mathbf{y}, \ldots)$$

with k quantifier alternations true? Complete problems for the classes Π_k^p are obtained by considering quantified boolean formulas with k quantifier alternations starting with a universal quantifier. A complete problem for the class NP $= \Sigma_1^p$ is to decide if a given closed propositional formula $\exists \mathbf{x}.\, F(\mathbf{x})$ is true which is equivalent to the problem SAT of satisfiability of $F(\mathbf{x})$.

2.3 First-order Logic

We start by defining many-sorted first-order languages. Let S be a set of *sorts* which are names of various domains under consideration.

Definition 2.13 (Signature) *A* signature Σ *is a triple* $\langle S;\ F;\ P\rangle$ *where*

- S *is a set of* sorts*;*
- F *is a set of pairs* $f\colon s_1 \times \cdots \times s_n \to w$ *where* $n \geq 0$*,* f *is a* function symbol*,* $s_1 \times \cdots \times s_n \to w$ *is the* type *of* f*, and* $s_1, \ldots, s_n, w \in S$*;*
- P *is a set of pairs* $p\colon s_1 \times \cdots \times s_n$ *where* $n \geq 0$*,* p *is a* predicate *symbol,* $s_1 \times \cdots \times s_n$ *is the* type *of* p*, and* $s_1, \ldots, s_n \in S$*.*

We use $Func(\Sigma)$ and $Pred(\Sigma)$ to denote the sets F and P of Σ. A function symbol with type $\to s$ is called a *constant* of type s.

Example 2.5 In this work we use frequently a signature with a single sort n containing the standard arithmetic function and predicate symbols

$$\Sigma = \langle \{n\};\ \{0\colon\ \to n, +, -, *, /\colon n \times n \to n\};\ \{=, \leq\colon n \times n\}\rangle. \qquad \square$$

Definition 2.14 (Terms) *Let* $V = \langle V_s \rangle_{s \in S}$ *be an* S*-indexed collection of sets of* variables *disjoint from the constants in* F*. For every sort* $s \in S$ *the set* $T_s(\Sigma, V)$ *of* terms of sort s *is inductively defined as follows:*

- *a variable* $x \in V_s$ *of type* s *is a term of sort* s*;*
- *if* $f\colon s_1 \times \cdots \times s_n \to w \in Func(\Sigma)$ *and* $t_1 \in T_{s_1}(\Sigma, V), \ldots, t_n \in T_{s_n}(\Sigma, V)$ *then* $f(t_1, \ldots, t_n) \in T_w(\Sigma, V)$ *is a term of sort* w*.*

An *atom* has the form $p(t_1, \ldots, t_n)$ where $p\colon s_1 \times \cdots \times s_n \in Pred(\Sigma)$ is a predicate symbol and $t_1, \ldots, t_n$ are terms of types $s_1, \ldots, s_n$ respectively. A *literal* is an atom (*positive literal*) of the form $p(t_1, \ldots, t_n)$ or the negation of an atom (*negative literal*) of the form $\neg p(t_1, \ldots, t_n)$. The *complement* $\overline{L}$ of a literal L is defined as $\overline{A} = \neg A$ for a positive literal A and $\overline{\neg A} = A$ for a negative literal $\neg A$. The set $\mathcal{L}_\Sigma$ of *first-order* Σ*-formulas* is defined in the usual way by using the logical connectives $\neg, \wedge, \vee$ and quantifiers $\forall, \exists$. A formula without negation is called *positive*.

Definition 2.15 *The set $FV(\cdot)$ of* free variables *for terms and formulas is defined as follows:*

$$\begin{aligned} FV(x) &= \{x\} && \text{for a variable } x \in V \\ FV(f(t_1, \ldots, t_n)) &= FV(t_1) \cup \cdots \cup FV(t_n) && \text{where } f \text{ is a function or predicate symbol} \\ FV(\neg\varphi) &= FV(\varphi) \\ FV(\varphi \circ \psi) &= FV(\varphi) \cup FV(\psi) && \text{where } \circ \in \{\vee, \wedge\} \\ FV(Qx.\ \varphi) &= FV(\varphi) - \{x\} && \text{where } Q \in \{\exists, \forall\} \end{aligned}$$

The existential and universal closures of a formula φ are denoted with $\tilde{\exists}\varphi$ and $\tilde{\forall}\varphi$ and defined as $\tilde{\exists}\varphi = \exists FV(\varphi).\ \varphi$ and $\tilde{\forall}\varphi = \forall FV(\varphi).\ \varphi$ respectively. Terms and formulas without variables are called *ground* and without free variables are called *closed*. A Σ-*theory* is a set of closed Σ-formulas.

Definition 2.16 (Structure) *A Σ-*structure *$\mathcal{D}$ consists of the following:*

- *for each sort $s \in S$ a domain D_s;*
- *for each function symbol $f\colon s_1 \times \cdots \times s_n \to w \in Func(\Sigma)$ a function $f^{\mathcal{D}}\colon D_{s_1}, \ldots, D_{s_n} \to D_w$;*
- *for each predicate symbol $p\colon s_1 \times \cdots \times s_n \in Pred(\Sigma)$ a relation $p^{\mathcal{D}} \subseteq D_{s_1} \times \cdots \times D_{s_n}$.*

Example 2.6 (Term Algebra) Let Σ be signature which contains at least one constant from each sort and no predicate symbols. The *term algebra* $\mathcal{T}$ (also known as a Herbrand pre-interpretation) is defined as follows. For every sort $s \in S$, the domain of s is the set of ground Σ-terms $T_s(\Sigma, \emptyset)$ of type s. The function symbols from Σ are interpreted as term constructors, i.e. $f^{\mathcal{T}}(t_1, \ldots, t_n) = f(t_1, \ldots, t_n)$. □

A *variable assignment* is a S-indexed family of functions $\sigma_s\colon V_s \to D_s$ which map every variable $x \in V_s$ of sort s to an element $\sigma_s(x) \in D_s$. With $\sigma_{x=d}$ we denote the variable assignment which maps x to d and maps all other variables as σ, i.e. $\sigma_{x=d}(x) = d$ and $\sigma_{x=d}(y) = \sigma(y)$ for $y \neq x$.

A variable assignment σ can be extended in a standard way to a *valuation function* of terms $[\![\cdot]\!]^{\sigma}_{\mathcal{D}}\colon T(\Sigma, V) \to D$ as follows:

$$\begin{aligned} [\![x]\!]^{\sigma}_{\mathcal{D}} &= \sigma(x) && \text{for a variable } x \in V \\ [\![f(t_1, \ldots, t_n)]\!]^{\sigma}_{\mathcal{D}} &= f^{\mathcal{D}}([\![t_1]\!]^{\sigma}_{\mathcal{D}}, \ldots, [\![t_n]\!]^{\sigma}_{\mathcal{D}}) && \text{for a term } f(t_1, \ldots, t_n) \in T(\Sigma, V) \end{aligned}$$

Definition 2.17 (Truth function) *Given a Σ-structure $\mathcal{D}$ and a variable assignment σ, the* truth-function *for first-order formulas $\mathcal{H}^{\sigma}_{\mathcal{D}}(\cdot) : \mathcal{L} \to \{\mathbf{f}, \mathbf{t}\}$ is*

defined as:

$$\begin{array}{ll}
\mathcal{H}^\sigma_\mathcal{D}(p(t_1,\ldots,t_n) & = \begin{cases} \mathbf{f}, & \text{if } ([\![t_1]\!]^\sigma_\mathcal{D},\ldots,[\![t_n]\!]^\sigma_\mathcal{D}) \notin p^\mathcal{D} \\ \mathbf{t}, & \text{if } ([\![t_1]\!]^\sigma_\mathcal{D},\ldots,[\![t_n]\!]^\sigma_\mathcal{D}) \in p^\mathcal{D} \end{cases} \\
\mathcal{H}^\sigma_\mathcal{D}(\neg\varphi) & = \neg\mathcal{H}^\sigma_\mathcal{D}(\varphi) \\
\mathcal{H}^\sigma_\mathcal{D}(\varphi \vee \psi) & = \mathcal{H}^\sigma_\mathcal{D}(\varphi) \vee \mathcal{H}^\sigma_\mathcal{D}(\psi) \\
\mathcal{H}^\sigma_\mathcal{D}(\varphi \wedge \psi) & = \mathcal{H}^\sigma_\mathcal{D}(\varphi) \wedge \mathcal{H}^\sigma_\mathcal{D}(\psi) \\
\mathcal{H}^\sigma_\mathcal{D}(\exists x.\ \varphi(x)) & = \bigvee_{d\in D} \mathcal{H}^\sigma_\mathcal{D}(\varphi(d)) \\
\mathcal{H}^\sigma_\mathcal{D}(\forall x.\ \psi(x)) & = \bigwedge_{d\in D} \mathcal{H}^\sigma_\mathcal{D}(\psi(d))
\end{array}$$

We also use the more conventional notion of a satisfiability relation — $\mathcal{D} \models_\sigma \varphi$ if and only if $\mathcal{H}^\sigma_\mathcal{D}(\varphi) = \mathbf{t}$. When the structure $\mathcal{D}$ is clear from the context, we drop the superscript $\mathcal{D}$ of the valuation function $[\![\cdot]\!]$ and truth function $\mathcal{H}$.

A *model* of a Σ-theory $\mathcal{T}$ is a Σ-structure $\mathcal{D}$ such that $\mathcal{D} \models \varphi$ for every $\varphi \in \mathcal{T}$.

2.4 Constraint Domains

In this section we recall the definition of constraint domains and their basic properties from (Jaffar and Maher, 1994; Jaffar et al., 1998).

A *constraint-domain* is a tuple $\mathcal{C} = \langle \Sigma, \mathcal{D}, \mathcal{L}, \mathcal{T} \rangle$ where:

- Σ is a first-order signature;
- $\mathcal{L}$ is a subset of the first-order Σ-language of allowed constraints;
- $\mathcal{D}$ is a Σ-structure which provides the intended interpretation of the constraints;
- $\mathcal{T}$ is a Σ-theory which describes the logical semantics of constraints;
- *solv* - a *solver* for the constraint domain.

A formula $C \in \mathcal{L}$ is called *constraint* and a Σ-atom is called a *primitive constraint*.

The signature Σ is assumed to contain the equality predicate $=$ which is interpreted as identity in $\mathcal{D}$ and that $\mathcal{T}$ contains the standard equality axioms for reflexivity, symmetry, transitivity and function and predicate substitution for the symbols in Σ.

We require the language $\mathcal{L}$ of the constraint domain $\mathcal{C}$ to contain all primitive constraints and to be closed under variable renaming and conjunction.

The Σ-structure $\mathcal{D}$ must be a model of $\mathcal{T}$.

Definition 2.18 *A* solution *of a constraint c is a variable assignment σ such that $\mathcal{D} \models_\sigma c$. The set of all solutions of c is defined as:*

$$sol(c) = \{\sigma \mid \mathcal{D} \models_\sigma c\}.$$

We say that two constraints c_1 *and* c_2 *are* equivalent *if* $sol(c_1) = sol(c_2)$.

The constraint solver *solv* transforms a constraint c to a set of constraints in a *solved form* which satisfy the following properties (Comon, 1991):

- *Solvability* - a constraint in solved form is either **f** or has at least one solution.
- *Simplicity* - every solution can be easily obtained from a solved form.
- *Completeness* - every problem is equivalent to a disjunction of solved forms.

A well-known solved form in the domain of equality of finite trees is the *unification solved form* which has the form $\bot$ or $x_1 = t_1 \wedge \ldots \wedge x_n = t_n$ where the x_i are variables that occur only once.

2.5 Logic Programming

We use a syntax and semantics of logic programs which is a combination of programs with first-order rule bodies (Lloyd and Topor, 1984) and Constraint Logic Programming (Jaffar and Maher, 1994; Jaffar et al., 1998). In particular, we distinguish between *pre-defined* symbols given by a first-order signature Σ and a set of *user-defined* predicate symbols Π. Let $\Sigma(\Pi) = \langle S;\ \ F;\ \ P \cup \Pi \rangle$ be a signature with a distinguished set of *user-defined* predicate symbols Π.

A $\Sigma(\Pi)$-*rule* is of the form

$$p(t_1, \ldots, t_n) \leftarrow \varphi$$

where $p\colon s_1 \times \cdots \times s_n \in \Pi$ is a user-defined predicate, $t_1, \ldots, t_n$ are Σ-terms of types $s_1, \ldots, s_n$, and φ is a $\Sigma(\Pi)$-formula. The atom $p(t_1, \ldots, t_n)$ is called the *head* of the rule and the formula φ the *body*. A *normal rule* is a rule whose body is a conjunction of $\Sigma(\Pi)$-literals. A $\Sigma(\Pi)$-*logic program* is a (possibly infinite) set of rules. A *normal logic program* is a logic program for which all rules are normal. A *definite logic program* is a logic program for which the bodies of all rules are positive formulas. A *definite normal logic program* is a normal logic program without negative literals in the body. For normal logic programs we use the symbol "," for conjunction and *not* for negation to distinguish the syntax from that of logic programs.

For the rest of the text we assume a fixed signature Σ and a fixed Σ-structure $\mathcal{D}$ interpreting the pre-defined symbols. For logic programs it is more convenient to give the interpretation of the user-defined symbols Π separately from the structure $\mathcal{D}$. The $\mathcal{D}$-*base* $base_{\mathcal{D}}(\Pi)$ of Π is defined as

$$base_{\mathcal{D}}(\Pi) = \{p(d_1, \ldots, d_n) \mid p\colon s_1 \times \cdots \times s_n \in \Pi, \text{ and } d_1 \in D_{s_1}, \ldots, d_n \in D_{s_n}\}.$$

An *interpretation* of the user-defined symbols Π is a mapping $I\colon base_{\mathcal{D}}(\Pi) \to \{\mathbf{f}, \mathbf{t}\}$. An interpretation I can also be specified by the subset $I \subseteq base_{\mathcal{D}}(\Pi)$ of the atoms which are assigned a value true. The set of all interpretations is denoted as $\mathcal{I} = \mathcal{P}(base_{\mathcal{D}}(\Pi))$. It forms a complete lattice under the subset inclusion order (see Example 2.2). An interpretation I is a *model* of a program P if $I \models \tilde{\forall} r$ for every rule $r \in P$ where the symbol $\leftarrow$ is interpreted as a logical implication.

A rule with free variables is normally understood as a notation for all instances obtained by grounding the free variables. We define the *grounding* of a program P over a structure $\mathcal{D}$ as follows:

$$ground_{\mathcal{D}}(P) = \{p([\![t_1]\!]^{\sigma}_{\mathcal{D}}, \ldots, [\![t_n]\!]^{\sigma}_{\mathcal{D}}) \leftarrow \sigma(\varphi) \mid p(t_1, \ldots, t_n) \leftarrow \varphi \in P \text{ and } \sigma \text{ is a variable assignment}\}.$$

The grounding of a logic program consists of a set of closed rules. We use the name "ground" because it is a standard terminology.

A central role for defining and studying the semantics of logic programs plays the T_P operator introduced by van Emden and Kowalski (van Emden and Kowalski, 1976).

Definition 2.19 *The* two-valued immediate consequence operator $T_P\colon \mathcal{I} \to \mathcal{I}$ *for a logic program P is defined as*

$$T_P(I) = \{A \in base_{\mathcal{D}}(\Pi) \mid A \leftarrow B \in ground(P) \text{ and } \mathcal{D}, I \models B\}.$$

Most of the semantics of logic programs correspond to different types of fixpoints of the T_P operator. For a definite normal program P the T_P operator is monotone. In this case the least fixpoint of T_P coincides with the least model of P (van Emden and Kowalski, 1976).

The immediate consequence operator of logic programs with negation may not be monotone. There is, however, a correspondence between pre-fixpoints of T_P and models of P.

Proposition 2.4 *$I \models P$ if and only if $T_P(I) \subseteq I$.*

One of the first semantics of logic programs with negation is Clark's completion semantics (Clark, 1978). It interprets the collection of all rules with the same predicate symbol p in the head as the *if-and-only-if* definition of p.

Definition 2.20 *The* completion *$comp(P)$ of a program P is a first-order theory defined as follows. First, every rule in P of the form*

$$p(t_1, \ldots, t_n) \leftarrow \varphi$$

is rewritten as

$$p(x_1, \ldots, x_n) \leftarrow \exists \mathbf{y}.\ x_1 = t_1 \wedge \ldots \wedge x_n = t_n \wedge \varphi$$

where $\mathbf{y} = FV(t_1) \cup \ldots \cup FV(t_n)$. *Then, for every predicate symbol* $p \in \Pi$ *we replace the (possibly empty) set of all rules with* p *in the head*

$$p(x_1, \ldots, x_n) \leftarrow \varphi_1.$$
$$\vdots$$
$$p(x_1, \ldots, x_n) \leftarrow \varphi_m.$$

with the formula

$$\tilde{\forall} p(x_1, \ldots, x_n) \leftrightarrow (\varphi_1 \vee \ldots \vee \varphi_m).$$

The original definition of the completion also includes the axioms of the equality predicate and the unique name axioms. However, these are incorporated in the structure $\mathcal{D}$ of the pre-defined symbols. Models of the theory $comp(P)$ formalize the intuition of supported models. A model I of P is *supported* if for every atom $A \in I$ there exists a rule $A \leftarrow \varphi \in ground(P)$ such that $I \models \varphi$. Finally, supported models correspond to fixpoints of the T_P operator. So, we have the following result.

Proposition 2.5 ((Apt et al., 1988)) *The following statements are equivalent:*

1. I *is a supported model of* P*;*
2. $I \models comp(P)$*;*
3. $T_P(I) = I$.

2.6 Approximation Theory

Approximation Theory (AT) is an algebraic theory which studies approximations of the fixpoints of non-monotone lattice functions. (Denecker et al., 2000; Denecker et al., 2002; Denecker et al., 2004).

2.6.1 Bilattices

The basic concept in approximation theory is that of a bilattice. In the literature there are different definitions of a bilattice (Ginsberg, 1988; Fitting, 1991a). For our purposes it suffices to consider a bilattice as the product of a lattice endowed with two partial orders:

Definition 2.21 (Bilattice) *Let* $\langle L, \leq \rangle$ *be a lattice. A* bilattice *is a structure* $\langle L^2, \leq_t, \leq_p \rangle$ *where* $\leq_t$ *and* $\leq_p$ *are partial orders called the* truth order *and the* precision order *respectively and defined as follows:*

truth order:	$(x, y) \leq_t (x_1, y_1)$ *if and only if* $x \leq x_1$ *and* $y \leq y_1$
precision order:	$(x, y) \leq_p (x_1, y_1)$ *if and only if* $x \leq x_1$ *and* $y_1 \leq y$

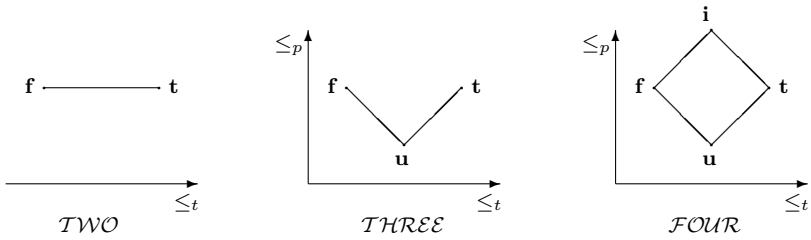

Figure 2.1: The structures $\mathcal{TWO}$, $\mathcal{THREE}$, and $\mathcal{FOUR}$.

We denote a bilattice simply with L^2. The restriction of L^2 to pairs (x, y) such that $x \leq y$ is denoted with L^c.

We call the elements (x, y) of the bilattice L^2 *approximations*. The element x is called the *under-estimate* and the element y is called the *over-estimate* of the approximation (x, y). A pair (x, y) such that $x \leq y$ is called *consistent* and if $x = y$ it is called *exact*. A consistent pair $(x, y) \in L^c$ can be seen as an approximation of all elements in the interval $[x, y] = \{z \in L \mid x \leq z \leq y\}$ which is non-empty. The precision order $\leq_p$ corresponds to the precision of the approximation, that is $(x, y) \leq_p (x_1, y_1)$ if and only if $[x, y] \supseteq [x_1, y_1]$. Exact pairs (x, x) approximate a single element x and represent the embedding of L in L^2 (or L^c).

Proposition 2.6 *The posets $\langle L^2, \leq_t \rangle$ and $\langle L^2, \leq_p \rangle$ are lattices. If L is complete lattice then so are $\langle L^2, \leq_t \rangle$ and $\langle L^2, \leq_p \rangle$.*

The structure of the set of consistent elements L^c is given by the following proposition.

Proposition 2.7 ((Fitting, 1991a)) *If L is complete lattice then $(L^c, \leq_t)$ is a complete lattice and $(L^c, \leq_p)$ is a complete semilattice.*

Example 2.7 Consider the lattice $\mathcal{TWO} = \{\mathbf{f}, \mathbf{t}\}$ of classical truth values ordered as $\mathbf{f} < \mathbf{t}$. We denote the bilattice $\mathcal{TWO}^2$ of $\mathcal{TWO}$ with $\mathcal{FOUR}$ and the set of consistent approximations $\mathcal{TWO}^c$ with $\mathcal{THREE}$ (Figure 2.1).

The exact pairs $(\mathbf{f}, \mathbf{f})$ and $(\mathbf{t}, \mathbf{t})$ are the embedding of the classical truth values $\mathbf{f}$ and $\mathbf{t}$ respectively. Only the pair $(\mathbf{f}, \mathbf{t})$ approximates more than one truth value, namely the set $\{\mathbf{f}, \mathbf{t}\}$, and corresponds to the value of *undefined*, denoted with $\mathbf{u}$. The pair $(\mathbf{t}, \mathbf{f})$ does not approximate any element and corresponds to the truth value of *inconsistent*, denoted with $\mathbf{i}$.

The truth order $\leq_t$ is used to define conjunction and disjunction in $\mathcal{FOUR}$ which are interpreted as greatest lower bound $\wedge_t$ and least upper bound $\vee_t$ respectively. Negation in $\mathcal{FOUR}$ is defined as $\neg(x, y) = (\neg y, \neg x)$. In particular,

$\neg\mathbf{f} = \mathbf{t}$, $\neg\mathbf{t} = \mathbf{f}$, $\neg\mathbf{u} = \mathbf{u}$, and $\neg\mathbf{i} = \mathbf{i}$. This interpretation of the logical connectives in the bilattice $\mathcal{FOUR}$ is the same as Belnap's four-valued logic (Belnap, 1997).

By Proposition 2.7, the poset $\langle \mathcal{THREE}, \leq_t \rangle$ is a complete lattice. Consequently, the operations $\wedge_t$ and $\vee_t$ are well-defined. They have the same interpretation as the Kleene's strong three-valued logic. □

Example 2.8 Let $L = \wp(D)$ be the powerset lattice of some domain D. The bilattice L^2 consists of pairs of sets (S_1, S_2). A set $S \in L$ can be viewed as a two-valued function $S\colon D \to \mathcal{TWO}$. Sometimes it is convenient to look at a pair of two-valued functions (S_1, S_2) as a *four-valued function* $\tilde{S}\colon D \to \mathcal{FOUR}$ defined as $\tilde{S}(x) = (S_1(x), S_2(x))$.

The set of consistent approximations L^c of L consists of all pairs of sets (S_1, S_2) such that $S_1 \subseteq S_2 \subseteq L$. We refer to such pairs as *three-valued sets* or *three-valued relations*. A three-valued set $\tilde{S} = (S_1, S_2)$ can be viewed as a pair of functions $S_1, S_2\colon D \to \mathcal{TWO}$ such that $S_1(x) \leq S_2(x)$ for all $x \in D$ or as a function $\tilde{S}\colon D \to \mathcal{THREE}$ defined as $\tilde{S}(x) = (S_1(x), S_2(x))$. □

2.6.2 Approximating Operators

Bilattices are used to approximate fixpoints on lattice operators. To this end we introduce the concept of approximating operator.

Definition 2.22 (Approximating Operator) *Let $O\colon L \to L$ be an operator on a complete lattice L. We say that $A\colon L^2 \to L^2$ is an* approximating operator *of $O\colon L \to L$ if A satisfies the following conditions:*

- *A* extends *O, i.e. $A(x, x) = (O(x), O(x))$ for all $x \in L$;*
- *A is $\leq_p$-monotone;*
- *A is* symmetric*, i.e. if $A(x, y) = (u, v)$ then $A(y, x) = (v, u)$.*

We denote the projection of $A(x, y)$ on the first and second components with A^1 and A^2, that is if $A(x, y) = (u, v)$ then $A^1(x, y) = u$ and $A^2(x, y) = v$.

From the $\leq_p$-monotonicity of an approximating operator A follows the existence of a $\leq_p$ least-fixpoint. We refer to this fixpoint as *Kripke-Kleene* and denote with $KK(A)$. It approximates all fixpoints of the original operator O.

Proposition 2.8 *Let $A\colon L^2 \to L^2$ be an approximating operator of $O\colon L \to L$. Then $KK(A)$ is consistent and $KK(A) \leq_p (x, x)$ for every fixpoint x of O.*

With every approximating operator A we can associate a stable operator in the following way.

Definition 2.23 ((Exact) Stable Operator) *Let $A\colon L^2 \to L^2$ be an approximating operator of $O\colon L \to L$.*

- *The* exact stable operator $C_A \colon L \to L$ *of* A *is defined as* $C_A(y) = \mathrm{lfp}(A^1(\cdot, y))$ *(or equivalently* $C_A(x) = \mathrm{lfp}(A^2(x, \cdot))$*).*
- *The* $S_A \colon L^2 \to L^2$ *of* A *is defined as* $S_A(x, y) = (C_A(y), C_A(x))$.

Because for every $y \in L$, the operators $A^1(\cdot, y)$ and $A^2(y, \cdot)$ are monotone, the operators C_A and S_A are well-defined.

The fixpoints of the stable operator S_A of A are called *stable fixpoints* of A. The exact stable fixpoints of A are also the fixpoints of the exact stable operator C_A and their set is denoted with $ST(A)$.

Proposition 2.9 *Let* $A \colon L^2 \to L^2$ *be an approximating operator of* $O \colon L \to L$. *The stable operator* $S_A \colon L^2 \to L^2$ *of* A *is an approximating operator of the exact stable operator* $C_A \colon L \to L$.

As a consequence, the stable operator S is also $\leq_p$-monotone and has a least fixpoint which we call the *well-founded fixpoint* of A and denote with $WF(A)$. It has the same property and relationship with the exact stable fixpoints of A as the Kripke-Kleene fixpoint in Proposition 2.8. In particular it is consistent and for every exact stable fixpoint $x \in ST(A)$, $WF(A) \leq_p (x, x)$.

The well-founded fixpoint of A is at least as precise as the Kripke-Kleene fixpoint.

Proposition 2.10 $KK(A) \leq_p WF(A)$ *for every approximating operator* A.

Proposition 2.11 *Let* $A \colon L^2 \to L^2$ *be an approximating operator of* $O \colon L \to L$. *Every stable fixpoint of* A *is a* $\leq_t$*-minimal fixpoint of* A.

This proposition implies that the stable fixpoints of A are a subset of the fixpoints of A. Also, the exact stable fixpoints of A are exact fixpoints of A which are also fixpoints of O. So we have the following corollary.

Corollary 2.12 *Let* $A \colon L^2 \to L^2$ *be an approximating operator of* $O \colon L \to L$. *The exact stable fixpoints of* A *are minimal fixpoints of* O.

For an operator $O : L \to L$ which is monotone or anti-monotone we can define an approximating operator based entirely on O.

Proposition 2.13 *Let* $O \colon L \to L$ *be a monotone operator on a complete lattice* L. *The operator* $A \colon L^2 \to L^2$ *defined as* $A(x, y) = (O(x), O(y))$ *is an approximating operator of* O. *Moreover,* $WF(A) = (\mathrm{lfp}(O), \mathrm{lfp}(O))$.

Proposition 2.14 *Let* $O \colon L \to L$ *be an anti-monotone operator on a complete lattice* L. *The operator* $A \colon L^2 \to L^2$ *defined as* $A(x, y) = (O(y), O(x))$ *is an approximating operator of* O.

2.6.3 Consistent Approximations

One of the important contributions of Approximation Theory is that the different classes of fixpoints of an approximating operator $A\colon L^2 \to L^2$ of O depend entirely on the set of consistent elements L^c (Denecker et al., 2002; Denecker et al., 2004). So, we consider the restriction of an approximating operator to the set of consistent elements L^c. It is defined in the same way as in Definition 2.22 except that we do not need the condition of symmetry.

Definition 2.24 (Consistent Approximating Operator) *Let $O\colon L \to L$ be an operator on a complete lattice L. We say that $A\colon L^c \to L^c$ is a* consistent approximating operator *of O if the following conditions are satisfied:*

- *A* extends *O, i.e. $\forall x \in L\colon A(x,x) = (O(x), O(x))$;*
- *A is $\leq_p$-monotone.*

We now give an alternative definition of the stable operator S_P which has the same set of consistent stable fixpoints as S_P. Recall the definition of the stable operator: $S_A(a,b) = (\mathrm{lfp}(A^1(\cdot.b)), \mathrm{lfp}(A^2(a,\cdot)))$. For consistent approximations, the operator $A^1(\cdot,b)$ is defined only on the domain $[\bot,b]$. However, it is possible that for some element $x \in [\bot,b]$, $A^1(x,b) \notin [\bot,b]$. Similarly, the operator $A^2(a,\cdot)$ is defined on the domain $[a,\top]$ but it is possible that for some element $y \in [a,\top]$, $A^2(a,y) \notin [a,\top]$. A set of consistent approximations for which the operators $A^1(\cdot,b)$ and $A^2(a,\cdot)$ are well-defined on the domains $[\bot,b]$ and $[a,\top]$ is the set $\mathrm{post}(A) = \{(a,b) \mid (a,b) \leq_p A(a,b)\}$ of post-fixpoint of A. Such approximations are called *A-reliable*.

Proposition 2.15 *If $(a,b) \in \mathrm{post}(A)$ then for every $x \in [\bot,b]$, $A^1(x,b) \in [\bot,b]$ and for every $y \in [a,\top]$, $A^2(a,y) \in [a,\top]$.*

Definition 2.25 (Consistent Stable Operator) *The* lower *and* upper stable operators *$C^l_A : L \to L$ and $C^u_A : L \to L$ are defined as follows:*

$$C^l_A(y) = \mathrm{lfp}(A^1|_{[\bot,y]}(\cdot,y))$$
$$C^u_A(x) = \mathrm{lfp}(A^2|_{[x,\top]}(x,\cdot))$$

The consistent stable operator *$S^c_A : \mathrm{post}(A) \to L^c$ is defined as*

$$S^c_A(x,y) = (C^l_A(y), C^u_A(x)).$$

Proposition 2.16 *Let $A^c\colon L^c \to L^c$ be the restriction of an approximating operator $A\colon L^2 \to L^2$ to the set of consistent elements L^c.*

- *a consistent pair (x,y) is a fixpoint of A if and only if (x,y) is a fixpoint of A^c;*

- *a consistent pair (x, y) is a fixpoint of S_A if and only if (x, y) is a fixpoint of S_A^c.*

We now show a simpler characterization of exact fixpoints of the consistent stable operator S_A^c.

Proposition 2.17 *Let $A : L^c \to L^c$ be a consistent approximating operator of $O : L \to L$. An element $x \in L$ is an exact fixpoint of the consistent stable operator S_A^c if*

1. *x is a fixpoint of O, i.e. $x = O(x)$;*
2. *$x = C_A^l(x)$.*

2.6.4 Precision of Approximations

We now introduce a precision order between consistent approximating operators and study the relationship between the fixpoints of operators with different precision. We also show that for every operator O, there exists a most precise consistent approximating operator, called an *ultimate approximation*. This operator plays an important role in the definition of the various semantics.

Definition 2.26 *Let $A\colon L^c \to L^c$ and $B\colon L^c \to L^c$ be two consistent approximating operators. We say that A is* less precise *than B, denoted with $A \leq_p B$, if $A(x, y) \leq_p B(x, y)$ for all $(x, y) \in L^c$.*

Clearly, if $A \leq_p B$ and A approximates an operator O then B also approximates O. Let $Appx(O)$ denote the set of all consistent approximating operators of O.

Proposition 2.18 *$\langle Appx(O), \leq_p \rangle$ is a complete lattice.*

The bottom element of this lattice is the *trivial approximation* T_O defined as $T_O(x, x) = (O(x), O(x))$ for exact approximations and $T_O(x, y) = (\bot, \top)$ for all other approximations $(x < y)$. More interesting is the most precise approximating operator of O called the *ultimate approximating operator* of O and denoted with U_O. It can be given the following constructive characterization.

Proposition 2.19 *Let $O\colon L \to L$ be an operator on a complete lattice L. Then*

$$U_O(x, y) = \Big(\bigwedge_{z \in [x,y]} O(z), \bigvee_{z \in [x,y]} O(z) \Big)$$

When we increase the precision of an approximating function we obtain more precise Kripke-Kleene and well-founded fixpoints and more exact stable fixpoints.

Proposition 2.20 *Let $A, B \in Appx(O)$. If $A \leq_p B$ then:*

- $KK(A) \leq_p KK(B)$;
- $WF(A) \leq_p WF(B)$;
- $ST(A) \subseteq ST(B)$.

Consequently, the ultimate approximating operator of O has the most precise Kripke-Kleene and well-founded fixpoints and the largest number of exact stable fixpoints compared with any other approximating operator of O.

The ultimate approximating operator U_O of a monotone lattice operator O has the form of Proposition 2.13

Proposition 2.21 *Let $O\colon L \to L$ be a monotone operator. Then $U_O(x, y) = (O(x), O(y))$.*

Together with Proposition 2.13 about the stable fixpoints of operators of approximating operators of the form $A(x, y) = (O(x), O(y))$ we obtain the following result.

Proposition 2.22 *Let $O\colon L \to L$ be a monotone operator on a complete lattice L. Then $WF(U_O) = (\mathrm{lfp}(O), \mathrm{lfp}(O))$.*

The ultimate approximating operator of an anti-monotone operator has the form of Proposition 2.14.

Proposition 2.23 *Let $O\colon L \to L$ be an anti-monotone operator. Then $U_O(x, y) = (O(y), O(x))$.*

2.6.5 Stratification of Operators

Other results from Logic Programming which were lifted to the algebraic setting of Approximation Theory are the standard model semantics of stratified logic programs (Apt et al., 1988) and the splitting theorem for answer set semantics (Lifschitz and Turner, 1994). Both results are based on the idea that the domain L of an operator $O : L \to L$ can be split in two parts L_1 and L_2 such that $L \cong L_1 \times L_2$ and for every element $(x_1, x_2) \in L_1 \times L_2$, the value y_1 of $(y_1, y_2) = O(x_1, x_2)$ does not depend on x_2 but only on x_1. In other words, the restriction of O to L_1 is a well-defined operator. We apply the stratification theory of operators to define standard model semantics of stratified aggregate programs (Section 3.3.2) and we use the stratification theory of approximating operators to show a splitting theorem for aggregate programs (Section 4.2.2). Our presentation follows the unpublished draft (Gilis and Denecker, 2002; Vennekens et al., 2003).

Let $\langle L_i \rangle_{i \in W}$ be a collection of posets $\langle L_i, \leq_i \rangle$ indexed by W. The *product* $L = \bigotimes_{i \in W} L_i$ is defined as

$$\bigotimes_{i \in W} L_i = \{x : W \to \bigcup_{i \in W} L_i \mid x(i) \in L_i\}.$$

If $W = \{1, \ldots, n\}$ is a finite index set then the product $\bigotimes_{i \in W} L_i$ is the same as the cartesian product $L_1 \times \cdots \times L_n$:

$$\bigotimes_{i \in \{1,\ldots,n\}} L_i = L_1 \times \cdots \times L_n = \{\langle x_1, \ldots, x_n \rangle \mid x_i \in L_i\}.$$

The partial orders $\leq_i$ on the posets L_i induce an order $\leq$ on L as follows:

$$x \leq y \quad \text{if and only if} \quad \forall i \in W \colon x(i) \leq_i y(i).$$

Definition 2.27 *A* complete lattice isomorphism *between two posets L_1 and L_2 is a bijective function $h : L_1 \to L_2$ which preserves the operations $\bigwedge$ and $\bigvee$ whenever they are defined, i.e.*

$$h(\bigwedge_{x \in X}{}_1\, x) = \bigwedge_{x \in X}{}_2\, h(x)$$

and similarly for $\bigvee$. If such function exists we say that L_1 and L_2 are isomorphic *and denote with $L_1 \cong L_2$.*

Let $\preceq$ be a well-founded order on W. For an element $i \in W$ we denote with $\preceq i$ the set $\preceq i = \{j \in W \mid j \preceq i\}$ and similarly, with $\prec i$ the set $\prec i = \{j \in W \mid j \prec i\}$.

Definition 2.28 (Stratification) *Let $O : L \to L$ be an operator on a poset L. A* collection $\langle L_i \rangle_{i \in W}$ *is a* stratification *of O if $L \cong \bigotimes_{i \in W} L_i$ and for every $x, y \in L$ and $i \in W$, if $x|_{\preceq i} = y|_{\preceq i}$ then $O(x)|_{\preceq i} = O(y)|_{\preceq i}$.*

The following proposition gives an alternative characterization of stratification of operators which is often more convenient to work with.

Proposition 2.24 *The collection $\langle L_i \rangle_{i \in W}$ is a* stratification *of $O : L \to L$ if and only if for each $i \in W$ and $u \in L|_{\prec i}$ there exists an operator $O_i^u : L_i \to L_i$ such that for every $x \in L$ if $x|_{\prec i} = u$ then $(O(x))(i) = O_i^u(x(i))$.*

Clearly, if the operators O_i^u exist then they are unique. We call these operators *components* of O.

The first property of a stratification of an operator O is that fixpoints of O correspond to fixpoints of its components. More precisely:

Proposition 2.25 *Let $\langle L_i \rangle_{i \in W}$ be a stratification of $O : L \to L$. Then*

$$x \in \mathrm{fp}(O) \quad \text{if and only if} \quad \forall i \in W \colon x(i) \in \mathrm{fp}(O_i^{x|_{\prec i}}).$$

The components of a stratifiable monotone operator are also monotone.

Proposition 2.26 *Let $O : L \to L$ be a monotone operator and $\langle L_i \rangle_{i \in W}$ be a stratification of O. For every $i \in W$ and $u \in L|_{\prec i}$, the component $O_i^u : L_i \to L_i$ is a monotone operator.*

It is possible that a non-monotone stratifiable operator O still has monotone components. In such case we define inductively a fixpoint of O which is the least fixpoint of every component.

Definition 2.29 (Standard Fixpoint) *Let $\langle L_i \rangle_{i \in W}$ be a stratification of $O : L \to L$ such that for every $i \in W$ and $u \in L|_{\prec} i$, the component O_i^u is monotone. The* standard fixpoint *of O is an element $s \in L$ such that*

$$\forall i \in W : s(i) = \mathrm{lfp}(O_i^{s|_{\prec i}}).$$

We can show that the standard fixpoint is a minimal fixpoint.

Proposition 2.27 *Let $\langle L_i \rangle_{i \in W}$ be a stratification of $O : L \to L$ such that for every $i \in W$ and $u \in L|_{\prec i}$, the component O_i^u is monotone. Then the standard fixpoint s of O is a minimal fixpoint of O.*

We now look at stratification of approximations. Let $\langle L_i \rangle_{i \in W}$ be a stratification of $O : L \to L$ and $A : L^2 \to L^2$ be an approximating operator of O. We first point out that the bilattice $L^2 = (\bigotimes_{i \in W} L_i)^2$ is isomorphic to the product lattice $\bigotimes_{i \in W} L_i^2$. Similarly, the set of consistent approximations $L^c = (\bigotimes_{i \in W} L_i)^c$ is isomorphic to the product lattice $\bigotimes_{i \in W} L_i^c$. We say that $\langle L_i \rangle_{i \in W}$ is a *stratification* of $A : L^2 \to L^2$ if $\langle L_i^2 \rangle_{i \in W}$ is a stratification of A. Stratification of a consistent approximating operator L^c is defined in a similar way. Clearly, all previous results about stratified monotone operators apply to A.

The result on stratification of approximating operators which is relevant to our work is the following.

Proposition 2.28 *Let $A : L^2 \to L^2$ be a stratifiable approximating operator on a product bilattice $\bigotimes_{i \in W} L_i^2$. Then (x, y) is a consistent stable fixpoint of A if and only if for every $i \in W$, $(x, y)(i)$ is a consistent stable fixpoint of the component $A_i^{(x,y)|_{\prec i}}$.*

2.6.6 Approximations in Logic Programming

The major non-monotonic semantics of logic programs can be obtained as an instance of Approximation Theory.

Let $base_{\mathcal{D}}(\Pi)$ be the $\mathcal{D}$-base of a set of predicate symbols Π for a structure $\mathcal{D}$. The basic lattice structure is the set of all interpretations of a logic program $\mathcal{I} = \wp(base_{\mathcal{D}}(\Pi))$. The bilattice $\mathcal{I}^2$ consists of pairs of interpretations (I_1, I_2). We call such pairs *four-valued interpretations*. As discussed in Example 2.8 a four-valued interpretation (I_1, I_2) can be considered as a four-valued function $\tilde{I} : base_{\mathcal{D}}(\Pi) \to \mathcal{FOUR}$. The set $\mathcal{I}^c$ of consistent four-valued interpretations consists of pairs (I_1, I_2) such that $I_1 \subseteq I_2$. We call such elements *three-valued interpretations*. The atoms in I_1 are those assigned a value of true, the atoms in $I_2 - I_1$ are assigned a value of undefined, and the atoms in $base_{\mathcal{D}}(\Pi) - I_2$ are assigned a value of false.

In logic programming we are interested to approximate the immediate consequence operator T_P. It is useful to give an equivalent definition of T_P by considering interpretations as boolean functions instead of sets:

$$T_P(I)(A) = \bigvee\{\mathcal{H}_I(B) \mid A \leftarrow B \in \mathit{ground}(P)\} \tag{2.1}$$

An approximating operator of T_P has been defined in the same way as (2.1) by evaluating the bodies in four-valued logic (Fitting, 2002; Przymusinksi, 1990). The truth value of first-order formulas in four and three-valued interpretations is defined in a similar way as the two-valued case (Definition 2.17) by using the truth order $\leq_t$ to interpret the logical connectives. In addition to the structure $\mathcal{D}$ and a variable assignment σ, $\mathcal{H}$ takes a four-valued interpretation $\tilde{I}$ of the user-defined predicates.

Definition 2.30 (Three-valued truth function) *The* three-valued truth function *for first-order formulas $\lambda(\tilde{I}, F)$. $\mathcal{H}_{\tilde{I}}(F) \colon \mathcal{I}^2 \times \mathcal{L} \to \mathcal{FOUR}$ is defined as:*

$$\begin{array}{ll}
\mathcal{H}_{\tilde{I}}(p(t_1, \ldots, t_n)) & = \begin{cases} (\mathbf{f}, \mathbf{f}), & \text{if } ([\![t_1]\!], \ldots, [\![t_n]\!]) \notin p^{\mathcal{D}} \\ (\mathbf{t}, \mathbf{t}), & \text{if } ([\![t_1]\!], \ldots, [\![t_n]\!]) \in p^{\mathcal{D}} \end{cases} \\
 & \quad \text{for a pre-defined predicate } p \in \mathit{Pred}(\Sigma) \\
\mathcal{H}_{\tilde{I}}(p(t_1, \ldots, t_n)) & = \tilde{I}(p([\![t_1]\!], \ldots, [\![t_n]\!])) \text{ for a user-defined predicate } p \in \Pi \\
\mathcal{H}_{\tilde{I}}(\neg\varphi) & = \neg\mathcal{H}_{\tilde{I}}(\varphi) \\
\mathcal{H}_{\tilde{I}}(\varphi \vee \psi) & = \mathcal{H}_{\tilde{I}}(\varphi) \vee_t \mathcal{H}_{\tilde{I}}(\psi) \\
\mathcal{H}_{\tilde{I}}(\varphi \wedge \psi) & = \mathcal{H}_{\tilde{I}}(\varphi) \wedge_t \mathcal{H}_{\tilde{I}}(\psi) \\
\mathcal{H}_{\tilde{I}}(\exists x.\, \varphi(x)) & = \bigvee_{t\, d \in D} \mathcal{H}_{\tilde{I}}(\varphi(d)) \\
\mathcal{H}_{\tilde{I}}(\forall x.\, \psi(x)) & = \bigwedge_{t\, d \in D} \mathcal{H}_{\tilde{I}}(\psi(d))
\end{array}$$

Definition 2.31 *The* four-valued immediate consequence operator $\Psi_P \colon \mathcal{I}^2 \to \mathcal{I}^2$ *for a logic program is defined as*

$$\Psi_P(\tilde{I})(A) = \bigvee_t\{\mathcal{H}_{\tilde{I}}(B) \mid A \leftarrow B \in \mathit{ground}(P)\}.$$

The restriction of Ψ_P to the set $\mathcal{I}^c$ of consistent elements is the three-valued immediate consequence operator *denoted with $\Phi_P \colon \mathcal{I}^c \to \mathcal{I}^c$.*

The Kripke-Kleene fixpoint of Φ_P (obviously) is equal to the Kripke-Kleene model of P (Fitting, 1985).

Exact stable fixpoints of Φ_P correspond to stable models (Gelfond and Lifschitz, 1988). To see this, let $GL(P;\ I)$ denote the Gelfond-Lifschitz transformation of a program P with respect to an interpretation I. It is easy to show that $\Psi^1_P(I_1, I_2) = T_{GL(P;\ I_2)}(I_1)$. Thus $\mathrm{lfp}(\Psi^1_P(\cdot, I))$ is equal to the least model of $GL(P;\ I)$. Consequently, I is an exact stable fixpoint of Ψ_P if and only if I is a stable model of P.

The definition of the well-founded fixpoint of Ψ_P in the bilattice $\mathcal{I}^2$ and its computation using ordinal powers of the stable operator S_{Ψ_P} is the same as the alternating fixpoint definition of the well-founded semantics (Van Gelder, 1989). The computation of the well-founded fixpoint based on the ordinal powers of the consistent stable operator $S^c_{\Phi_P}$ is very similar to the bottom-up evaluation technique based on the doubled program of Kemp, Srivastava, and Stuckey (Kemp et al., 1995).

Finally, consistent stable fixpoints correspond to three-valued stable models, as defined by Fitting (Fitting, 1991b; Fitting, 1993; Fitting, 1997) and Przymusinksi (Przymusinksi, 1990).

The semantics of logic programs obtained from the ultimate approximating operator U_P of T_P has been considered by (Denecker et al., 2004). For some programs this operator is strictly more precise than the Φ_P operator.

Example 2.9 Consider the program P_1:

$$p \leftarrow p.$$
$$p \leftarrow not\, p.$$

Its Kripke-Kleene and well-founded fixpoint are $(\{\}, \{p\})$ (i.e. p is undefined) and it has no stable models. On the other hand, the T_P operator of this program is monotone. It coincides with the immediate consequence operator of the program $P_2 = \{p.\}$. Hence, they have the same ultimate models. Since the Kripke-Kleene model of P_2 is the 2-valued interpretation $\{p\}$, this is also the ultimate Kripke-Kleene, ultimate well-founded and the unique ultimate stable model of P_1. □

Chapter 3

Aggregate Programs and Two-valued Semantics

In this chapter we introduce the syntax of aggregate atoms and aggregate programs. We also extend several two-valued semantics of logic programs to aggregate programs: the least fixpoint semantics of definite normal logic programs (van Emden and Kowalski, 1976), the completion semantics (Clark, 1978), and the standard model semantics of stratified normal logic programs (Apt et al., 1988).

The definitions of the classes of definite aggregate programs and weakly stratified aggregate programs are very similar to monotonic and stratified monotonic aggregate programs (Mumick et al., 1990). However, our definitions are almost syntactic (they depend on monotonicity properties of aggregate relations) while the definitions of (Mumick et al., 1990) are semantic. The main reason to introduce these semantics is to show that the (ultimate) three-valued stable semantics of aggregate programs which we define in Chapter 4 extend the semantics of definite and weakly stratified aggregate programs in the same way the three-valued stable semantics of normal logic programs (Przymusinksi, 1990) extends the semantics of definite and stratified normal logic programs. This property is not satisfied by many of the previous proposals for well-founded (Kemp and Stuckey, 1991) and stable semantics (Dell'Armi et al., 2003b; Gelfond, 2002; Kemp and Stuckey, 1991).

The definitions of definite and weakly stratified aggregate programs in this chapter are given for aggregate programs with full first-order bodies. They were presented for a restricted propositional language in (Pelov et al., 2003; Pelov et al., 2004). However, the semantics of weakly stratified aggregate programs was given indirectly by a translation to a stratified non-aggregate program. This chapter contains a direct definition of a standard model which is based on the stratification theory of operators (Gilis and Denecker, 2002; Vennekens et al., 2003).

3.1 Aggregates

In this section we give a precise definition of aggregates and study some basic properties like monotonicity.

An aggregate is typically understood as a function from a multiset to a single element. A multiset (also called a *bag*) is similar to a set except that an object can occur multiple times.

Definition 3.1 (Multiset) *A* multiset M *on domain* D *is a function* $M\colon D \to Card$ *where* $Card$ *is the class of cardinal numbers.*

For every element x, $M(x)$ gives the multiplicity of x. We use the notation $x \in M$ whenever $M(x) > 0$. We denote the collection of all multisets on D with $\mathcal{M}(D)$. Of special interest are multisets with a finite number of elements.

Definition 3.2 (Finite Multiset) *A* multiset with finite multiplicity *is a function* $M\colon D \to \mathbb{N}$ *where* $\mathbb{N}$ *is the set of natural numbers. A* finite multiset *is a multiset with finite multiplicity such that* $M(x) > 0$ *only for a finite number of elements.*

We denote the set of all finite multisets on D with $\mathcal{FM}(D)$. The *size* of a finite multiset M is defined as $|M| = \sum_{x \in D} M(x)$.

We denote finite multisets using a set notation where an element is repeated as many times as its multiplicity. For example the multiset M containing the elements a and b with multiplicity $M(a) = 2$ and $M(b) = 1$ is denoted with $M = \{a, a, b\}$.

The subset relation between multisets is defined as follows: $M_1 \subseteq M_2$ if and only if $M_1(x) \leq M_2(x)$ for all $x \in D$. The *additive union* $M = M_1 \uplus M_2$ of M_1 and M_2 is defined as $M(x) = M_1(x) + M_2(x)$ for all $x \in D$. Finally, the *multiset difference* $M = M_1 - M_2$ of M_1 and M_2 is defined as $M(x) = M_1(x) - M_2(x)$ if $M_1(x) \geq M_2(x)$ and $M(x) = 0$ otherwise. A common way to construct multisets is as the image $M = f(S)$ of a set $S \subseteq D_1$ under $f\colon D_1 \to D_2$ defined as $M(y) = |\{x \in S \mid f(x) = y\}|$. A set S is a special case of a multiset where the multiplicity of every element $x \in S$ is one.

For a multiset M and a set S we define M^S as the restriction of M to the elements of S, i.e.,

$$M^S(x) = \begin{cases} M(x), & \text{if } x \in S \\ 0, & \text{if } x \notin S \end{cases}.$$

For a multiset M on (a subset of) $\mathbb{R}$, let $M^- = M^{(-\infty,0]}$ and $M^+ = M^{[0,\infty)}$.

Aggregates are typically functions. However, for our work it is more convenient to treat an aggregate as a relation between a multiset and an element.

Definition 3.3 (Aggregate Functions and Relations) *Let* D_1 *and* D_2 *be sets. An* aggregate function *is any function* $\mathrm{F}\colon \mathcal{M}(D_1) \to D_2$. *An* aggregate relation *is any relation* $\mathrm{R} \subseteq \mathcal{M}(D_1) \times D_2$.

We use F to denote an aggregate function and R to denote an aggregate relation. Whenever convenient we use an aggregate function F in a context which requires an aggregate relation. In such situations F is understood as its *graph* $\mathrm{G_F}$ which is an aggregate relation defined as $(M, d) \in \mathrm{G_F}$ if and only if $\mathrm{F}(M) = d$.

3.1.1 Standard Aggregates

We now define a number of standard aggregate functions and relations which we use throughout the text. We start with aggregates on finite multisets of numbers.

Definition 3.4

- CARD: $\mathcal{FM}(D) \to \mathbb{N}$ *defined as* $\mathrm{CARD}(M) = \sum_{x \in M} M(x)$;
- SUM: $\mathcal{FM}(\mathbb{R}) \to \mathbb{R}$ *defined as* $\mathrm{SUM}(M) = \sum_{x \in M} x.\, M(x)$;
- PROD: $\mathcal{FM}(\mathbb{R}) \to \mathbb{R}$ *defined as* $\mathrm{PROD}(M) = \prod_{x \in M} x^{M(x)}$;
- AVG: $\mathcal{FM}(\mathbb{R}) \times \mathbb{R}$ - *a partial aggregate function defined only for non-empty multisets as* $(M, d) \in \mathrm{AVG}$ *if* $d = \mathrm{SUM}(M)/\mathrm{CARD}(M)$.

All these aggregate functions become aggregate relations when extended to infinite multisets — they are false for any infinite multiset M and real number n.

Example 3.1 We give the result of applying the aggregate functions in Definition 3.4 for some multisets:

$$\begin{aligned} \mathrm{CARD}(\{1,1,2,2\}) &= 4 \\ \mathrm{SUM}(\{1,1,2,2\}) &= 6 \\ \mathrm{PROD}(\{1,1,2,2\}) &= 4 \\ \mathrm{AVG}(\{1,1,2,2\}) &= 1.5 \end{aligned}$$

□

The aggregate relations below are defined for a poset $\langle D, \leq \rangle$. For these relations the multiplicity of the elements in the multiset is not significant.

Definition 3.5

- INF, SUP $\subseteq \mathcal{M}(D) \times D$ - *partial aggregate functions defined as* $(M, d) \in \mathrm{INF}$ *if and only if* d *is the greatest lower bound of* M *and* $(M, d) \in \mathrm{SUP}$ *is true if* d *is the least upper bound of* M.
- INF, SUP: $\mathcal{M}(D) \to D$ *where* $\langle D, \leq \rangle$ *is a complete lattice are total aggregate functions.*

- $\text{LB}, \text{UB} \subseteq \mathcal{M}(D) \times D$ - *aggregate relations defined as* $(M, d) \in \text{LB}$ *if and only if* d *is a lower bound of* M *and* $(M, d) \in \text{UB}$ *is true if* d *is an upper bound of* M*.*

- $\text{MIN}, \text{MAX} \subseteq \mathcal{M}(D) \times D$ - $(M, d) \in \text{MIN}$ *if and only if* d *is a minimal element of* M *and* $(M, d) \in \text{MAX}$ *if and only if* d *is a maximal element of* M*. For general posets a given multiset can have none, one, or more minimal element.*

On a totally ordered set D, MIN and MAX represent partial functions.

3.1.2 Derived Aggregate Relations

One advantage of representing aggregates as relations is that we can obtain new aggregate relations by composition of an existing aggregate with another relation.

Definition 3.6 *The* composition *of an aggregate function* $\text{F}\colon \mathcal{M}(D_1) \to D_2$ *with a binary relation* $P \subseteq D_2 \times D_3$ *is an aggregate relation* $\text{F}_P \subseteq \mathcal{M}(D_1) \times D_3$ *defined as* $(M, d) \in \text{F}_P$ *if and only if* $(\text{F}(M), d) \in P$*. The* composition *of an aggregate relation* $\text{R} \subseteq \mathcal{M}(D_1) \times D_2$ *with a binary relation* $P \subseteq D_2 \times D_3$ *is an aggregate relation* $\text{R}_P \subseteq \mathcal{M}(D_1) \times D_3$ *defined as* $(M, d) \in \text{R}_P$ *if and only if there exists* $a \in D_2$ *such that* $(M, a) \in \text{R}$ *and* $(a, d) \in P$*.*

Typically, the binary relation P is some partial order relation on the domain D. For example the $\text{CARD}_{\geq}$ aggregate relation means that the number of the elements in the multiset is greater than or equal to the second argument. Another useful composition is $\text{CARD}_{(n)}$ where (n) is the "modulo n" relation defined as $(x, y) \in (n)$ if $x \mod n = y$. It can be used to implement the EVEN an ODD aggregates: $(M, 0) \in \text{CARD}_{(2)}$ if and only if $|M|$ is even and $(M, 1) \in \text{CARD}_{(2)}$ if and only if $|M|$ is odd.

An aggregate relation can also be composed from the left with a relation on multisets. We consider only a composition with the submultiset relation.

Definition 3.7 *The* subset aggregate $\text{R}_{\subseteq}$ *of* R *is defined as the composition of the superset relation on multisets* $\supseteq$ *and* R*:* $(M, d) \in \text{R}_{\subseteq}$ *if and only if there exists a multiset* M' *such that* $M \supseteq M'$ *and* $(M', d) \in \text{R}$*.*

3.1.3 Monotone Aggregates

For an aggregate function $\text{F}\colon \mathcal{M}(D_1) \to D_2$ and a poset $\langle D_2, \leq \rangle$, we can apply the notion of monotonicity and anti-monotonicity of functions (Section 2.1): F is monotone if $M_1 \subseteq M_2$ implies $\text{F}(M_1) \leq \text{F}(M_2)$ and F is anti-monotone if $M_1 \subseteq M_2$ implies $\text{F}(M_1) \geq \text{F}(M_2)$. The next two propositions list standard aggregate functions which are monotone with respect to some partial order.

Proposition 3.1 *Let $\langle D, \leq \rangle$ be a complete lattice. The aggregate function* INF *is anti-monotone with respect to $\leq$ and the aggregate function* SUP *is monotone with respect to $\leq$.*

aggregate function	domain	partial order
INF	lattice $\langle D, \leq \rangle$	$\leq$
SUP	lattice $\langle D, \leq \rangle$	$\geq$
CARD	$\mathbb{N}$	$\geq$
SUM	$\mathbb{R}^+$	$\geq$
SUM	$\mathbb{R}^-$	$\leq$
PROD	$\mathbb{R}^{[1,\infty)}$	$\geq$
PROD	$\mathbb{R}^{[0,1)}$	$\leq$

Table 3.1: Monotone Aggregate Functions on Finite Multisets

Proposition 3.2 *The aggregate functions on Table 3.1 are monotone with respect to the given partial order for finite multisets on the given domain.*

Monotonicity and anti-monotonicity of aggregate relations is defined in the following way.

Definition 3.8 *Let* $\mathrm{R} \subseteq \mathcal{M}(D_1) \times D_2$ *be an aggregate relation. We say that* R *is:*

- monotone *if* $(M_1, d) \in \mathrm{R}$ *and* $M_1 \subseteq M_2$ *implies* $(M_2, d) \in \mathrm{R}$*;*
- anti-monotone *if* $(M_2, d) \in \mathrm{R}$ *and* $M_1 \subseteq M_2$ *implies* $(M_1, d) \in \mathrm{R}$.

The next proposition gives aggregate relations which are anti-monotone.

Proposition 3.3 *The aggregate relations* $\mathrm{LB}, \mathrm{UB} \subseteq \mathcal{M}(D) \times D$ *on a poset* $\langle D, \leq \rangle$ *are anti-monotone.*

We point out that a monotone aggregate function is not a monotone aggregate relation according to Definition 3.8. Instead, the composition of an aggregate function F with the inverse of the order with respect to which it is monotone results in a monotone aggregate relation.

Proposition 3.4 *Let* $\mathrm{F}\colon \mathcal{M}(D_1) \to D_2$ *be an aggregate function which is monotone with respect to a partial order relation $\leq$ on D_2. Then* $\mathrm{F}_{\geq}$ *is a monotone aggregate relation.*

For example, the aggregate relation $\text{CARD}_{\geq}$ is monotone.

There is a simple way to turn an arbitrary aggregate relation into a monotone one. As soon as an aggregate relation R is true for some pair (M, d) we define the completion of R to be true for any pair (M', d) where $M \subseteq M'$.

Definition 3.9 (Monotone Completion of Aggregate Relations) *The* completion *of an aggregate relation* $\text{R}\colon \mathcal{M}(D_1) \times D_2$ *is the aggregate relation* $\text{R}^{\uparrow} \subseteq \mathcal{M}(D_1) \times D_2$ *defined as the smallest relation satisfying:*

1. $\text{R} \subseteq \text{R}^{\uparrow}$;
2. *if* $(M, d) \in \text{R}$ *and* $M \subseteq M'$ *then* $(M', d) \in \text{R}^{\uparrow}$.

The next proposition gives some basic properties of the completion of an aggregate.

Proposition 3.5

1. $\text{R}^{\uparrow}$ *is a monotone aggregate relation,*
2. *if* R *is a monotone aggregate relation then* $\text{R}^{\uparrow} = \text{R}$.

One useful application of the monotone completion of aggregate relations is to extend a monotone aggregate relation on finite multisets to infinite multisets.

Example 3.2 Consider the aggregate function $\text{CARD}\colon \mathcal{FM}(D) \to \mathbb{N}$ which is monotone with respect to $\geq$. The derived aggregate relation $\text{CARD}_{\geq} \subseteq \mathcal{FM}(D) \times \mathbb{N}$ is a monotone aggregate relation. If we extend $\text{CARD}_{\geq}$ to infinite multisets then it is false for any infinite multiset. The completion $\text{CARD}^{\uparrow}_{\geq} \subseteq \mathcal{M}(D) \times \mathbb{N}$ of $\text{CARD}_{\geq} \subseteq \mathcal{M}(D) \times \mathbb{N}$ is an aggregate relation defined as $(M, d) \in \text{CARD}^{\uparrow}_{\geq}$ if and only if M is finite and $|M| \geq d$ or M is infinite. Note that the restriction of $\text{CARD}^{\uparrow}_{\geq}$ to finite multisets coincides with $\text{CARD}_{\geq}$. □

Finally, it is interesting to note that taking the subset aggregate $\text{R}_{\subseteq}$ of R is just another way to construct the completion of R.

Proposition 3.6 $\text{R}_{\subseteq} = \text{R}^{\uparrow}$

Proof. $\subseteq$) $(M, d) \in \text{R}_{\subseteq}$ implies that there exists $M' \subseteq M$ such that $(M', d) \in \text{R}$. Because $\text{R} \subseteq \text{R}^{\uparrow}$ then $(M', d) \in \text{R}^{\uparrow}$. Finally, because $\text{R}^{\uparrow}$ is the monotone closure of R then $(M, d) \in \text{R}^{\uparrow}$.

$\supseteq$) $(M, d) \in \text{R}^{\uparrow}$ implies that there exists $M' \subseteq M$ such that $(M', d) \in \text{R}$ which implies that $(M, d) \in \text{R}_{\subseteq}$. ■

3.2 Aggregate Atoms and Programs

We now introduce the syntax and semantics of aggregate programs. We do this by extending a first-order language $\mathcal{L}$ with aggregate atoms. Many authors combine the denotation of multisets and aggregates in a single syntactic expression. However, we find that there is a clear distinction between the two concepts and introduce separate denotations. The denotation of multisets is further separated into a set expression and a lambda expression. The multiset is obtained as the image of the denotation of the set expression (which is a set) under the denotation of the lambda expression (which is a function).

A *first-order set expression* or *set expression* for short of type $s_1 \times \cdots \times s_n$ has the form $\{(x_1, \ldots, x_n) \mid \varphi\}$ where φ is a first-order formula called the *condition* of the set expression and for each $i = 1, \ldots, n$ x_i is a variable of sort s_i. The variables x_i are local for the expression, i.e. $FV(\{(x_1, \ldots, x_n) \mid \varphi\}) = FV(\varphi) - \{x_1, \ldots, x_n\}$. In practice $\{x_1, \ldots, x_n\}$ is a subset of the free variables of φ. A *lambda expression of type* $s_1 \times \cdots \times s_n \to w$ has the form $\lambda(x_1, \ldots, x_n).\ u$ where for each $i = 1, \ldots, n$ x_i is a variable of sort s_i and u is a term of sort w. Again, the variables x_i are local for the lambda expression, i.e. $FV(\lambda(x_1, \ldots, x_n).\ u) = FV(u) - \{x_1, \ldots, x_n\}$.

Definition 3.10 (Aggregate Signature) *An* aggregate signature Σ *is a tuple of the form* $\langle S;\ F;\ P;\ A\rangle$ *where* $\langle S;\ F;\ P\rangle$ *is a signature and* A *is a set of pairs* $\mathrm{R} : \{s_1\} \times s_2$ *where* $s_1, s_2 \in S$, R *is an* aggregate symbol*, and* $\{s_1\} \times s_2$ *is the* type *of* R.

We use $Aggr(\Sigma)$ to denote the set A of aggregate symbols. For $\mathrm{R} : \{s_1\} \times s_2$, the sort s_1 is the type of the multiset argument and the sort s_2 is the type of the result of the aggregate.

Definition 3.11 *An* aggregate atom *has the form* $\mathrm{R}(f, s, t)$ *where* $\mathrm{R}\colon \{m\} \times w \in Aggr(\Sigma)$ *is an aggregate symbol,* f *is a lambda expression of type* $s_1 \times \cdots \times s_n \to m$*,* s *is a set expression of type* $s_1 \times \cdots \times s_n$*, and* t *is a term of sort* w*.*

The next problem is an example where the multiplicity of the elements in a multiset is important.

Example 3.3 (Power Plant Maintenance) A power plant has a number of units which have to be scheduled for maintenance. There is a restriction on the total capacity of the units in maintenance. The relation $capacity(U, C)$ defines that a unit U has a capacity C. The predicate $in_maint(U, S, E)$ specifies that unit U is in maintenance during the period starting at time point S (inclusive) and ending at time point E (exclusive). The total capacity TC of the units in maintenance at time point T can be defined with the following aggregate atom:

$$\begin{aligned}&\textsc{sum}(\lambda(U, C).\ C,\\ &\quad \{(U, C) \mid \exists S, E.\ in_maint(U, S, E) \wedge S \leq T < E \wedge capacity(U, C)\},\\ &\quad TC).\end{aligned}$$

Because several units may have the same capacity we pair the capacity with the unit name. The lambda expression then projects on the second argument with the capacity which is summed. □

For making proofs about aggregate programs it is convenient to use a simpler notation of aggregate atoms without lambda expressions of the form $\text{R}(s,t)$. In this case we assume that the aggregate is applied on the multiset obtained from the projection of the set denoted by s on the first argument, i.e. $\text{R}(s,t)$ is a shorthand for $\text{R}(\lambda(x_1,\ldots,x_n).\, x_1, s, t)$. Every aggregate atom

$$\text{R}(\lambda(x_1,\ldots x_n).\, u, \{(x_1,\ldots,x_n) \mid \varphi\}, t)$$

can be rewritten in this simpler notation as

$$\text{R}(\{(z, x_1,\ldots,x_n) \mid \varphi \wedge z = u\}, t).$$

For example, the aggregate atom from Example 3.3 can be represented in this notation as:

$$\text{SUM}(\{(C,U) \mid \exists S,E.\, in_maint(U,S,E) \wedge S \leq T < E \wedge capacity(U,C)\}, TC).$$

Let $\mathcal{L}^{aggr}$ be the first-order language with aggregates built over an aggregate signature Σ.

3.2.1 Semantics

Aggregate structures are structures which in addition provide interpretation of the aggregate symbols as aggregate relations.

Definition 3.12 (Aggregate Structure) *A* Σ-aggregate structure $\mathcal{D}$ *consists of the following:*

- *for each sort* $s \in S$ *a domain* D_s*;*
- *for each function symbol* $f\colon s_1 \times \cdots \times s_n \to w \in Func(\Sigma)$ *a function* $f^{\mathcal{D}}\colon D_{s_1},\ldots,D_{s_n} \to D_w$*;*
- *for each predicate symbol* $p\colon s_1 \times \cdots \times s_n \in Pred(\Sigma)$ *a relation* $p^{\mathcal{D}} \subseteq D_{s_1} \times \cdots \times D_{s_n}$*;*
- *for each aggregate symbol* $\text{R}\colon \{s_1\} \times s_2 \in Aggr(\Sigma)$ *an aggregate relation* $\text{R}^{\mathcal{D}} \subseteq \mathcal{M}(D_{s_1}) \times D_{s_2}$*.*

Given an aggregate structure $\mathcal{D}$ to define the semantics of a first-order language with aggregate $\mathcal{L}^{aggr}$ we define a valuation function for set expressions and lambda expressions, denoted with $[\![\cdot]\!]$, and extend the truth function $\mathcal{H}$ to aggregate atoms.

The value $[\![\{x_1, \ldots, x_n \mid \varphi\}]\!]^\sigma_\mathcal{D}$ of a set expressions is a set. It consists of all tuples of elements $(d_1, \ldots, d_n) \in D^n$ such that the condition φ is true in $\mathcal{D}$ after substituting $d_1, \ldots, d_n$ for $x_1, \ldots, x_n$:

$$[\![\{x_1, \ldots, x_n \mid \varphi\}]\!]^\sigma_\mathcal{D} = \{(d_1, \ldots, d_n) \in D^n \mid \mathcal{D} \models_{\sigma\{x_1=d_1,\ldots,x_n=d_n\}} \varphi\}$$

The value $[\![\lambda x_1, \ldots, x_n.\ u]\!]^\sigma_\mathcal{D}$ of a lambda expression is a function $f\colon D^n \to D$ defined as $f(x_1, \ldots, x_n) = [\![u]\!]^{\sigma\{x_1=d_1,\ldots,x_n=d_n\}}_\mathcal{D}$.

Let $\mathrm{R}(f, s, t)$ be an aggregate atom and M be the multiset $M = [\![f]\!]([\![s]\!])$. Then $\mathcal{H}^\sigma_\mathcal{D}(\mathrm{R}(f, s, t)) = \mathbf{t}$ if and only if $(M, [\![t]\!]^\sigma_\mathcal{D}) \in \mathrm{R}^\mathcal{D}$. For an aggregate atom $\mathrm{R}(s, t)$ without lambda expression, the semantics is defined as $\mathcal{H}^\sigma_\mathcal{D}(\mathrm{R}(s, t)) = \mathbf{t}$ if and only if $(\pi_1([\![s]\!]^\sigma_\mathcal{D}), [\![t]\!]^\sigma_\mathcal{D}) \in \mathrm{R}^\mathcal{D}$. By a slight abuse of notation we also drop the projection function π_1 and simply write $([\![s]\!]^\sigma_\mathcal{D}, [\![t]\!]^\sigma_\mathcal{D}) \in \mathrm{R}^\mathcal{D}$.

There are problems which require aggregates and that can be modeled entirely in the first-order logic with aggregates which we just defined. One example is the well-known magic square problem.

Example 3.4 (Magic Square) Given is a $n \times n$ grid which has to be filled with the integer numbers from 1 to n^2 such that the sum of the numbers in all rows, columns, and two diagonals is equal to the same number $M(n)$, known as the *magic constant*:

$$M(n) = \frac{n(n^2+1)}{2}$$

Let Σ be an aggregate signature consisting of the following sorts: *pos* for the positions of the table, *num* for the numbers in the table, and *nat* for the sum of the numbers. We use a function symbol $f\colon pos \times pos \to num$ specifying the number in the corresponding row and column and a constant $mc\colon num$ giving the magic number. In addition we use the aggregate symbol $\mathrm{SUM}\colon \{num\} \times nat$ The problem is modeled by the following theory T:

$$\begin{aligned}
&mc = n * (n * n + 1)/2.\\
&\forall(x_1, x_2, y).\ x_1 = x_2 \vee f(x_1, y) \neq f(x_2, y).\\
&\forall(x, y_1, y_2).\ y_1 = y_2 \vee f(x, y_1) \neq f(x, y_2).\\
\\
&\forall y.\ \mathrm{SUM}(\lambda x.\ f(x, y), \{x \mid \mathbf{t}\}, mc).\\
&\forall x.\ \mathrm{SUM}(\lambda y.\ f(x, y), \{y \mid \mathbf{t}\}, mc).\\
&\mathrm{SUM}(\lambda(x, y).\ f(x, y), \{(x, y) \mid x = y\}, mc).\\
&\mathrm{SUM}(\lambda(x, y).\ f(x, y), \{(x, y) \mid x + y = n + 1\}, mc).
\end{aligned}$$

The first line computes the value of the magic constant. The second and third sentences specify that the numbers in all columns and rows have to be different. The last four sentences specify that the sum of the numbers in all rows, columns,

and two diagonals must be equal to the magic constant mc. To find a solution of a magic square instance of size n we consider an aggregate Σ-structure $\mathcal{D}$ which associates with the sort pos the domain $D_{pos} = \{1, \ldots, n\}$, with the sort num the domain $D_{num} = \{1, \ldots, n^2\}$ and with the sort nat the domain $D_{nat} = \mathbb{N}$. If $\mathcal{D} \models T$ then the interpretation of f given by $\mathcal{D}$ is a solution of the problem. □

3.2.2 Aggregate Programs

An *aggregate program* is a program in which the bodies of the rules are aggregate formulas. A *normal aggregate program* is an aggregate program in which the bodies of all rules are conjunctions of literals and aggregate atoms. For a rule r of a normal aggregate program we use $body(r)$ to denote the body of r, $pos(r)$ to denote the set of all positive literals in the body of r, $neg(r)$ to denote the set of all negative literals in the body of r, and $aggr(r)$ to denote the set of all aggregate atoms in the body of r.

Having defined two-valued satisfiability of aggregate atoms we can define a two-valued immediate consequence operator T_P^{aggr} for aggregate programs.

Definition 3.13 *The* two-valued immediate consequence operator $T_P^{aggr} : \mathcal{I} \to \mathcal{I}$ *for an aggregate program P is defined as:*

$$T_P^{aggr}(I) = \{A \mid A \leftarrow B \in ground(P) \text{ and } I \models B\}.$$

3.3 Two-valued Semantics

We present several semantics of aggregate programs which are based on the T_P^{aggr} operator and can be defined entirely in two-valued logic. The semantics of definite and (weakly) stratified aggregate programs are defined only on restricted classes of aggregate programs while the completion semantics is defined for any aggregate program.

3.3.1 Definite Aggregate Programs

Definite aggregate programs are the simplest class of aggregate programs. Their main property is that they have a monotone T_P^{aggr} operator and that its least fixpoint is considered as the intended model.

We define the syntactic notions of positive and negative formulas. For aggregate atoms the definition depends not only on the syntax of the atom but also on the interpretation of the aggregate symbol. We can do that because aggregate symbols always have a fixed interpretation given by an aggregate structure $\mathcal{D}$. Taking into account the monotonicity properties of aggregate relations allows us to give a more precise definition. In particular we distinguish the cases when the aggregate symbol is interpreted as a monotone or anti-monotone aggregate relation.

We start by defining the notion of positive and negative occurrences of atoms in a formula.

Definition 3.14 (Positive and Negative Occurrences of Atoms)

- *an atom* A occurs positively *in A;*
- *if an atom A occurs positively (resp. negatively) in a formula φ then A* occurs positively *(resp.* negatively*) in $\exists x.\ \varphi$, $\forall x.\ \varphi$, $\varphi \vee \psi$, and $\varphi \wedge \psi$;*
- *if A occurs positively (resp. negatively) in φ then A* occurs negatively *(resp.* positively*) in $\neg\varphi$;*
- *if A occurs positively (resp. negatively) in φ and $\mathrm{R}^{\mathcal{D}}$ is a monotone aggregate relation then A* occurs positively *(resp.* negatively*) in $\mathrm{R}(f, \{\mathbf{x} \mid \varphi\}, t)$;*
- *if A occurs positively (resp. negatively) in φ and $\mathrm{R}^{\mathcal{D}}$ is an anti-monotone aggregate relation then A* occurs negatively *(resp.* positively*) in $\mathrm{R}(f, \{\mathbf{x} \mid \varphi\}, t)$;*

We note that an atom can occur both positively and negatively in the same formula, for example p in $p \vee \neg p$. Also if a formula φ does not contain non-monotone aggregate relations then every atom in φ occurs either positively or negatively (or both) in φ. However, for an aggregate atom A with a non-monotone aggregate relation, the atoms in A occur neither positively nor negatively in A.

A simple way to determine the polarity of an atom (or a sub-formula) ψ is to count the number of negations on top of ψ. If they are even then ψ occurs positively. If they are odd then ψ occurs negatively. Aggregate atoms with anti-monotone aggregate relations count as negations and aggregate atoms with non-monotone aggregate relations "block" the counting and they occur neither positively nor negatively.

Definition 3.15 (Positive and Negative Aggregate Formulas)
An aggregate formula φ is positive *(resp.* negative*) if all atoms in φ occur only positively (resp. only negatively) in φ.*

If the formula φ is an aggregate atom of the form $\mathrm{R}(f, \{\mathbf{x} \mid \varphi\}, t)$ there are two cases in which it can be positive. The first one is when the aggregate symbol R is interpreted as a monotone aggregate relation $\mathrm{R}^{\mathcal{D}}$ and φ is a positive formula. The second case is when R is interpreted as an anti-monotone aggregate relation and φ is a negative formula. Similarly, the aggregate atom $\mathrm{R}(f, \{\mathbf{x} \mid \varphi\}, t)$ is negative if $\mathrm{R}^{\mathcal{D}}$ is a monotone aggregate relation and φ is negative or $\mathrm{R}^{\mathcal{D}}$ is an anti-monotone aggregate relation and φ is positive.

Proposition 3.7 *Satisfiability of positive aggregate formulas is monotone and satisfiability of negative aggregate formulas is anti-monotone.*

Proof. Let ψ be an aggregate formula and I and J be two interpretations such that $I \subseteq J$. We show by induction on the structure of ψ that if it is positive then $I \models \psi$ implies $J \models \psi$ and if ψ is negative then $J \models \psi$ implies $I \models \psi$. For positive and negative formulas without aggregates this property is a standard result in the theory of first-order logic. We consider only the case when ψ is an aggregate atom of the form $\mathrm{R}(f, \{\mathbf{x} \mid \varphi\}, t)$. Let $S_I = [\![\{\mathbf{x} \mid \varphi\}]\!]^I$ and $S_J = [\![\{\mathbf{x} \mid \varphi\}]\!]^J$.

First, let φ be a positive aggregate atom. We can distinguish two cases.

1. Let R be interpreted by a monotone aggregate relation $\mathrm{R}^{\mathcal{D}}$. In this case φ must be a positive formula and so is $\forall \mathbf{x}.\ \varphi$. By the inductive hypothesis $I \models \forall \mathbf{x}.\ \varphi$ implies $J \models \forall \mathbf{x}.\ \varphi$. Consequently, $S_I \subseteq S_J$. Finally, because $\mathrm{R}^{\mathcal{D}}$ is a monotone aggregate relation then $(S_I, d) \in \mathrm{R}^{\mathcal{D}}$ implies $(S_J, d) \in \mathrm{R}^{\mathcal{D}}$ thus $I \models \varphi$ implies $J \models \varphi$.

2. Let R be interpreted by an anti-monotone aggregate relation $\mathrm{R}^{\mathcal{D}}$. In this case φ must be a negative formula and so is $\forall \mathbf{x}.\ \varphi$. By the inductive hypothesis $J \models \forall \mathbf{x}.\ \varphi$ implies $I \models \forall \mathbf{x}.\ \varphi$. Consequently, $S_J \subseteq S_I$. Finally, because $\mathrm{R}^{\mathcal{D}}$ is an anti-monotone aggregate relation then $(S_I, d) \in \mathrm{R}^{\mathcal{D}}$ implies $(S_J, d) \in \mathrm{R}^{\mathcal{D}}$ thus $I \models \varphi$ implies $J \models \varphi$.

The proof of anti-monotonicity of negative aggregate atoms is similar and is omitted. ∎

We point out that the classes of positive and negative formulas are a strict subset of the classes for which the satisfiability relation is monotone (resp. anti-monotone). For example the formula $p \vee \neg p$ is a tautology and hence it is monotone, however it is neither positive nor negative.

Definition 3.16 *A* definite aggregate program *is an aggregate program such that the bodies of all rules are positive aggregate formulas.*

The class of definite aggregate programs is an extension of the class of definite logic programs and has a monotone immediate consequence operator.

Theorem 3.8 *If P is a definite aggregate program then T_P^{aggr} is monotone.*

Proof. Follows immediately from Proposition 3.7. ∎

For programs with a monotone immediate consequence operator it is generally accepted that the intended semantics is given by the least fixpoint of this operator.

The Company Controls problem (Kemp and Stuckey, 1991; Mumick et al., 1990; Ross and Sagiv, 1997; Van Gelder, 1992) is a typical example of a definite aggregate program.

Example 3.5 (Company Controls) A sort *comp* represents companies and a sort *shares* interpreted over the real interval [0..1] represents a fraction of the

shares. The interpreted predicate *owns_stock* has type $comp \times comp \times shares$ and $owns_stock(X, Y, S)$ means that a company X owns a fraction S of the stock of a company Y. The defined predicate *controls* has type $comp \times comp$; $controls(X, Y)$ denotes that company X has a controlling interest in a company Y. This is the case when X owns (directly or through an intermediate company that X controls) more than 50% of the stock of Y. The predicate $cv(X, Z, Y, S)$ means X controls a fraction S of the stocks of Y through an intermediate company Z.

$$cv(X, X, Y, S) \leftarrow owns_stock(X, Y, S).$$
$$cv(X, Z, Y, S) \leftarrow controls(X, Z) \wedge owns_stock(Z, Y, S).$$

$$controls(X, Y) \leftarrow \text{SUM}_{>}(\lambda(Z, S).\ S, \{(Z, S) \mid cv(X, Z, Y, S)\}, 0.5).$$

For numbers in the interval [0..1], the SUM aggregate function is monotone and $cv(X, Z, Y, S)$ is a positive formula, so the aggregate atom in the third rule is monotone. Since none of the bodies contain negation this is a definite aggregate program with a monotone T_P^{aggr} operator. Moreover, its least fixpoint provides the intended interpretation. This is because the *controls* predicate has to be interpreted as the transitive closure over $\text{SUM}_{\geq}$ of the *owns_stock* relation. □

3.3.2 Stratified Aggregate Programs

We now define the classes of stratified and weakly stratified[1] aggregate programs. Both classes extend the class of stratified logic programs (Apt et al., 1988; Lloyd, 1987). For stratified aggregate programs aggregate atoms are treated in the same way as negative literals: all predicate symbols in the set expression of an aggregate atom have to be defined in a strictly lower level than the predicate symbol in the head of the rule. So, this class of programs does not include aggregate programs with recursion over aggregation. On the other hand, weakly stratified aggregate program is a weaker notion of stratification. The conditions for the predicates in a single stratum are essentially the same as for definite aggregate programs: it may contain positive aggregate atoms with predicates in the set expression defined in the same level.

A *stratification* of an aggregate program P is a partition $\langle \Pi_i \rangle_{i \in W}$ of the user-defined predicate symbols of Π where W is some index set. For every predicate symbol $p \in \Pi$ the *level* $\|p\|$ of p is the index i for which $p \in \Pi_i$.

Definition 3.17 ((Weakly) Stratified Aggregate Program) *An aggregate program P is* weakly stratified *if there exists a stratification $\langle \Pi_i \rangle_{i \in W}$ of Π and a well-order $\preceq$ on W such that for every rule $p(\mathbf{t}) \leftarrow \varphi \in P$ and for every predicate symbol q in φ:*

[1] Our notion of weak stratification is not related with weakly stratified logic programs (Przymusinska and Przymusinski, 1990).

- $\|q\| \preceq \|p\|$ *if all atoms with* q *occur positively in* φ*;*
- $\|q\| \prec \|p\|$ *if some atom with* q *does not occur positively in* φ*.*

A stratified aggregate program *must, in addition, satisfy:*

- $\|q\| \prec \|p\|$ *if* q *occurs in a set expression of an aggregate atom in* φ*.*

The class of weakly stratified aggregate programs is strictly larger than the class of stratified aggregate programs.

Let $P_i \subseteq P$ be the sub-program of P containing only rules with predicate symbols of level i. Note that the rules of P_i use predicate symbols only of level i or lower levels. Let $\mathcal{I}_i = \wp(base_{\mathcal{D}}(\Pi_i))$.

Proposition 3.9 *If* P *is a (weakly) stratified aggregate program with stratification* $\langle \Pi_i \rangle_{i \in W}$ *then* $\langle \mathcal{I}_i \rangle_{i \in W}$ *is a stratification of* T_P^{aggr}*.*

Proof. We use the characterization of stratifiability of Proposition 2.24. Fix a level $i \in W$, and an interpretation $U \in \mathcal{I}|_{\prec i}$. Let $(T_P^{aggr})_i^U : \mathcal{I}_i \to \mathcal{I}_i$ be the operator defined as follows $(T_P^{aggr})_i^U(J) = T_{P_i}^{aggr}(U \cup J)$. Let $I \in \mathcal{I}$ be an interpretation which agrees with U, i.e. $I|_{\prec i} = U$.

$$
\begin{aligned}
T_P^{aggr}(I)|_i &= T_{P_i}^{aggr}(I) \\
&= T_{P_i}^{aggr}(I|_{\preceq i}) && \text{because } P_i \text{ contains only predicates of} \\
& && \text{level } i \text{ or lower} \\
&= T_{P_i}^{aggr}(U \cup (I|_i)) && \text{because } I|_{\prec i} = U \\
&= (T_P^{aggr})_i^U(I|_i) && \text{by definition of } (T_P^{aggr})_i^U
\end{aligned}
$$

■

Lemma 3.10 *Let* P *be a (weakly) stratified aggregate program and* $\langle \Pi_i \rangle_{i \in W}$ *be a stratification of* P*. Then for every* $i \in W$ *and* $U \in \mathcal{I}|_{\prec i}$*, the operator* $(T_P^{aggr})_i^U(J) = T_{P_i}^{aggr}(U \cup J)$ *is monotone.*

Now, we can apply the definition of standard fixpoint (Definition 2.29) and define a standard model M_P of a stratified aggregate program. We can also apply Proposition 2.27 and obtain a result that the standard model is a minimal fixpoint of T_P^{aggr} and consequently a minimal supported model of P. For logic programs without aggregates, our definition agrees with the one from standard logic programs (Apt et al., 1988; Lloyd, 1987).

Note that definite aggregate programs are weakly stratified — the entire program can be put in a single stratum. Thus, the Company Controls program from Example 3.5 is also a weakly stratified aggregate program. However, it is not stratified. The next example contains a program which is stratified but not definite.

Example 3.6 (Shortest Path) Let *nodes* be a sort representing nodes and c a sort representing weights, interpreted with some set of real numbers $D_c \subseteq \mathbb{R}$. The graph is defined by a predicate $edge\colon n \times n \times c$. A tuple $edge(X, Y, L)$ represents an edge from X to Y with weight L. Consider the following formulation of the problem of finding the length of the shortest path between two nodes which can be found in (Van Gelder, 1992, Example 4.1).

$$sp(X, Y, S) \leftarrow \text{MIN}(\lambda C.\, C, \{C \mid cp(X, Y, C)\}, S).$$

$$cp(X, Y, C) \leftarrow edge(X, Y, C).$$
$$cp(X, Y, C_1 + C_2) \leftarrow cp(X, Z, C_1) \wedge edge(Z, Y, C_2).$$

The aggregate relation MIN is non-monotone, so the aggregate atom MIN(...) in the first rule is neither positive nor negative. Consequently, the program is not definite. However, the program is stratified. The first stratum contains the $cp/3$ predicate and the sub-program P_{cp} containing only the definition of $cp/3$ is a definite program. The predicate $cp/3$ computes the transitive closure of the graph and $cp(X, Y, C)$ is true in the least model of P_{cp} if and only if there is a path between X and Y with length C. The second stratum contains only the definition of $sp/3$ and $sp(a, b, w)$ is true in the standard model of the program if and only if the shortest path between a and b has a length w. Note that if the graph contains a path from a node a to itself with a negative length then $sp(a, a, w)$ will be false for all values $w \in D_c$ in the cost domain. □

3.3.3 Supported Models

Although the extension of the supported model semantics to aggregate programs is straightforward we still discuss it shortly because it provides the intended semantics for several examples with "recursion" over aggregates. The definition of supported models does not require any change for aggregate programs. We also have the same characterization of supported models as Proposition 2.5 for logic programs without aggregates.

Proposition 3.11 *The following statements are equivalent:*

1. *I is a supported model of P;*
2. *$I \models comp(P)$;*
3. *$T_P^{aggr}(I) = I$.*

As a first example of a problem under the completion semantics we consider the following version of the Party Invitation problem (Ross and Sagiv, 1997; Zaniolo and Wang, 1999).

Example 3.7 (Party Invitation I) A number of people are invited to a party. A person p will accept the invitation if and only if at least k of his (her) friends also accept the invitation. The friendship relation is given by a binary predicate $friend(X, Y)$ meaning that Y is a friend of X. The input also consists of a relation $thr(X, T)$ giving the lower bound T on the number of friends of X. The problem is modeled by the program consisting of the following single rule:

$$accept(X) \leftarrow thr(X, T) \wedge \text{CARD}_{\geq}(\lambda Y.\ 1, \{Y \mid friend(X, Y) \wedge accept(Y)\}, T).$$

Note that the aggregate relation $\text{CARD}_{\geq}$ is monotone and the aggregate atom in the rule is also monotone. So, this is a definite aggregate program with a monotone T_P^{aggr} operator. However, taking the least fixpoint of this operator may not be the appropriate semantics of this problem. In fact the statement of the problem suggest that the completion semantics may be more appropriate.

Consider an input consisting of two persons a and b such that a will accept if and only if b does and vice versa. The precise input is:

$$\begin{array}{ll} friend(a, b). & thr(a, 1). \\ friend(b, a). & thr(b, 1). \end{array}$$

The least fixpoint of the T_P^{aggr} operator is $\emptyset$, i.e. none of the persons accept the invitation. However, $comp(P)$ has a second model, namely $\{a, b\}$ which can also be considered as a correct solution. The statement of the problem does not exclude the possibility that a and b communicate with each other about their decisions to attend the party. □

In a more elaborate version of the problem one can specify a degree of friendship which can be both positive and negative (Van Gelder, 1992, Example 6.3). This implies that the resulting program will have a non-monotone T_P^{aggr} operator.

Example 3.8 (Party Invitation II)
The sort *pers* represents persons. The pre-defined predicate *comp* has a type $pers \times pers \times real$ and $comp(X, Y, P)$ means that person X has a compatibility measure P with person Y. If P is a negative number that means that X dislikes Y with degree P. A person P will accept the invitation if and only if the sum of the compatibilities of all other people who accept the invitation is greater than or equal than some threshold K, given in a relation $thr(P, K)$ of type $pers \times real$. The program for this version consists of the following rule.

$$\begin{aligned} accept(X) \leftarrow & thr(X, K) \wedge \\ & \text{SUM}_{\geq}(\lambda(Y, P).\ P, \{(Y, P) \mid comp(X, Y, P) \wedge accept(Y)\}, K). \end{aligned}$$

Now, because the input multiset of the $\text{SUM}_{\geq}$ aggregate may contain both positive and negative numbers the aggregate atom is not monotone and the T_P^{aggr} operator is non-monotone.

Again, we argue that Clark's completion semantics agrees with the intended semantics of the problem. However, unlike the Party Invitation I problem where there is always at least one solution, this version may not have solutions for some inputs.

Consider a situation where a person a will accept if and only if b accepts and b will accept if and only if a does not. This input can be defined as:

$$\begin{array}{ll} comp(a, b, 1). & thr(a, 1). \\ comp(b, a, -1). & thr(b, 0). \end{array}$$

Instantiating and simplifying the program with this input we obtain the program P_1:

$$\begin{array}{l} accept(a) \leftarrow \text{SUM}_{\geq}(\lambda(Y, P).\ P, \{(Y, P) \mid accept(b) \wedge Y = b \wedge P = 1\}, 1). \\ accept(b) \leftarrow \text{SUM}_{\geq}(\lambda(Y, P).\ P, \{(Y, P) \mid accept(a) \wedge Y = a \wedge P = -1\}, 0). \end{array}$$

In classical logic, satisfiability of the aggregate atoms in the bodies of the two rules is the same as the satisfiability of $accept(a)$ and $\neg accept(b)$ respectively. So, the program P_1 is equivalent to P_2:

$$\begin{array}{l} accept(a) \leftarrow accept(b). \\ accept(b) \leftarrow \neg accept(a). \end{array}$$

Finally, the completion $comp(P_2)$ is the following first-order theory:

$$\begin{array}{l} accept(a) \leftrightarrow accept(b). \\ accept(b) \leftrightarrow \neg accept(a). \end{array}$$

Clearly, $comp(P_2)$ does not have any models. □

Another problem for which the completion semantics is sufficient is the following puzzle[2] from (Hurley, 1971) suggested to me by Maarten Mariën.

Example 3.9 (Hundred I) There is a piece of paper with 100 sentences written on it:

1. Exactly one sentence on this page is false.
2. Exactly two sentences on this page are false.

⋮

99. Exactly 99 sentences on this page are false.
100. Exactly 100 sentences on this page are false.

[2] `http://rec-puzzles.org/new/sol.pl/logic/hundred`

How many sentences are false and how many are true?

Let n be a sort interpreted with the interval of natural numbers [1..100]. Let $s\colon n$ be a predicate denoting that sentence i is true. The problem is modeled with the following one rule program.

$$s(X) \leftarrow \text{CARD}(\{Y \mid not\, s(Y)\}, X).$$

To find the solution of the problem first observe that all statements are mutually exclusive, so at most one can be true. If they are all false then $s(100)$ must be true which is a contradiction. The other possibility is that 99 of the statements are false so $s(99)$ must be true.

Clearly, classical models of Clark's completion of the program provide the intended semantics. This is because there is no reason to exclude self-supported models. Indeed, P has a single supported model in which $s(99)$ is the only true atom which agrees with the solution above. □

Example 3.10 (Hundred II) One variation of the Hundred I example is obtained be replacing "exactly" by "at least" in the question. The corresponding program is

$$s(X) \leftarrow \text{CARD}_{\geq}(\{Y \mid not\, s(Y)\}, X).$$

To find a solution to this variant, suppose that x statements are false. Then statements $s(1)$ to $s(x)$ will be true and $s(x+1)$ to $s(100)$ will be false. We obtain the equation $x = 100 - x$, so $x = 50$.

Again, the completion semantics is the right semantics of this problem. Indeed, $comp(P)$ has a single model which coincides with this solution.

Note that this time the T_P^{aggr} operator is anti-monotone because the aggregate atom $\text{CARD}_{\geq}(\{Y \mid not\, s(Y)\}, X)$ is a negative formula. Anti-monotone operators have a $\leq_p$-maximal oscillating pair (I_1, I_2), i.e. $T_P^{aggr}(I_1) = I_2$, $T_P^{aggr}(I_2) = I_1$, and for any other oscillating pair (J_1, J_2) of T_P^{aggr}, $(I_1, I_2) \leq_p (J_1, J_2)$. For this program the maximal oscillating pair of T_P^{aggr} is $(\emptyset, \{s(x) \mid x \in [1..100]\}$. □

3.4 Related Work

In logic programming, aggregate expressions have been denoted in different ways. A common way (Kemp and Stuckey, 1991; Mumick et al., 1990) is a *group_by* expression of the form:

$$group_by(p(\mathbf{x}, \mathbf{y}), [\mathbf{x}], C = \text{F}(u(\mathbf{x}, \mathbf{y})))$$

where $\mathbf{x}$ are the grouping variables which together with C are the free variables for the expression and C is the result of computing the aggregate function F. It is equivalent in our notation to:

$$\text{F}(\lambda \mathbf{y}.\, u(\mathbf{x}, \mathbf{y}), \{\mathbf{y} \mid p(\mathbf{x}, \mathbf{y})\}, C).$$

Some authors (Sudarshan et al., 1993; Van Gelder, 1992) argue that it is convenient to allow the computation of aggregates on sub-multisets of the input multiset. This can be accomplished within our framework without introducing any additional syntax by means of the subset aggregate $\mathrm{R}_{\subseteq}$. A more detailed study on the use of the subset aggregate is given in Section 5.3.3.

The class of monotonic aggregate programs (Mumick et al., 1990) is very similar to the class of definite aggregate programs. A monotonic programs is a program in which every rule is monotonic. A monotonic rule is a rule r such that $I \models \mathit{body}(r)$ and $I \subseteq J$ implies $J \models r$ for any pair of interpretations I and J. Although this is a semantic definition of monotonicity, the authors introduce a sufficient syntactic condition for monotonicity of a rule. Essentially, an aggregate atom can appear only in formulas of the form

$$\exists z.\ \mathrm{R}(\lambda \mathbf{x}.\ u, \{\mathbf{x} \mid q(\mathbf{x})\}, z) \wedge p(z, t) \tag{3.1}$$

where p is a pre-defined relation. Moreover, the satisfiability of this formula must be monotone. In our syntax (3.1) can be expressed as the aggregate atom

$$\mathrm{R}_P(\lambda \mathbf{x}.\ u, \{\mathbf{x} \mid q(\mathbf{x})\}, t) \tag{3.2}$$

with the derived aggregate relation R_P. Then, the condition that the satisfiability of (3.1) is monotone is equivalent to the condition that R_P is a monotone aggregate relation (and consequently (3.2) is a positive aggregate atom). The notion of positive aggregate atoms is simpler and, in our opinion, more natural than the condition of monotonic literals of (Mumick et al., 1990). In the same way we extend the least fixpoint semantics of definite aggregate programs to the standard model semantics of weakly stratified aggregate programs, (Mumick et al., 1990) extend the semantics of monotonic programs to stratified monotonic programs.

Finally, we mention a line of research which considers the least fixpoint semantics of aggregate programs with a monotone T_P^{aggr} operator under non-standard orders between interpretations (Ross and Sagiv, 1997; Van Gelder, 1993) and does not have an analog in logic programming. (Denecker et al., 2001b) contains some discussion on the relationship with this approach.

3.5 Conclusions

In this chapter we introduced the syntax of aggregate atoms and programs and defined several semantics which can be obtained from the two-valued immediate consequence operator T_P^{aggr}. Instead of studying only aggregate functions we considered arbitrary aggregate relations. An important idea in our approach is the construction of derived aggregate relations as the composition of aggregate functions and relations on their domains. This allowed us to define monotonicity and anti-monotonicity of aggregate relations and atoms in a natural way which is

independent on any domain specific order relation. Based on these properties, we defined the semantics of three restricted classes of aggregate programs: definite, stratified, and weakly stratified. These classes of programs and their semantics are extensions of the corresponding classes and semantics in logic programming (van Emden and Kowalski, 1976; Apt et al., 1988).

One interesting result from our study of examples is that the programs of several examples which have recursion over aggregation should not be considered as recursive definitions but as *if-and-only-if* definitions. For such programs classical models of Clark's completion of the program (Clark, 1978) provide the intended semantics. To our knowledge this semantics of aggregate programs has not been considered in the literature before.

Chapter 4

Three-Valued Stable Semantics of Aggregate Programs

In this chapter we define several three-valued stable semantics of aggregate programs. These semantics are obtained as stable fixpoints of several consistent approximating operators of the immediate consequence operator T_P^{aggr} of aggregate programs. The first semantics which we study is obtained from the ultimate approximating operator of T_P^{aggr} and extends the ultimate semantics of logic programs without aggregates (Denecker et al., 2004). This semantics is interesting because it is the most precise semantics according to Approximation Theory. This means that it has the most precise well-founded model and the number of total stable models is larger than that of any other semantics obtained from a consistent approximating operator of T_P^{aggr}. One concern with the ultimate semantics is that even for logic programs without aggregates it is more expensive to compute the ultimate well-founded and stable models than the standard well-founded and stable models.

The other three-valued stable semantics which we define (Section 4.2) are all extensions of the three-valued stable semantics of logic programs (Przymusinksi, 1990) (and consequently of the stable (Gelfond and Lifschitz, 1988) and well-founded (Van Gelder et al., 1991) semantics). According to Approximation Theory, the three-valued semantics of logic programs (Przymusinksi, 1990) can be obtained from the three-valued operator Φ_P defined by Fitting (Fitting, 1985). We show how this operator can be extended to a consistent approximating operator Φ_P^{aggr} of T_P^{aggr}. The operator Φ_P^{aggr} is based on extending aggregate relations to three-valued relations on three-valued multisets. We give conditions for these extensions that guarantee that the Φ_P^{aggr} operator is a consistent approximating

operator of T_P^{aggr}. Such extensions are called approximating aggregate relations. Each of them gives rise to a different three-valued stable semantics.

In Section 4.3 we give one possible definition of approximating aggregate relations which is defined in a uniform way for all aggregate relations. It is based on the ideas and constructions of ultimate approximations. Consequently, the three-valued semantics which it defines is the most precise three-valued stable semantics in this family. We show that this semantics extend the least fixpoint semantics of definite aggregate programs and the standard model semantics of weakly stratified aggregate programs.

The last section in this chapter (Section 4.4) presents some preliminary results on extending the stable semantics of disjunctive logic programs (Przymusinski, 1991) to programs with aggregates. It is based on a novel algebraic characterization of stable models based on non-deterministic operators.

The ultimate well-founded and stable semantics of aggregate programs (Section 4.1) have been published in (Denecker et al., 2001b) and the three-valued stable semantics (Section 4.2) in (Pelov et al., 2004).

4.1 Ultimate Three-Valued Stable Semantics

The first semantics of aggregate programs based on approximation theory which we study is the one based on the ultimate approximating operator U_P^{aggr} of T_P^{aggr} (Denecker et al., 2001b). Using Proposition 2.19 it can be defined as follows.

Definition 4.1 *The* ultimate approximating operator $U_P^{aggr} : \mathcal{I}^c \to \mathcal{I}^c$ *of* $T_P^{aggr} : \mathcal{I} \to \mathcal{I}$ *is defined as:*

$$U_P^{aggr}(I_1, I_2) = (\bigcap_{I \in [I_1, I_2]} T_P^{aggr}(I), \bigcup_{I \in [I_1, I_2]} T_P^{aggr}(I)).$$

Definition 4.2 *The set of* ultimate three-valued stable models *of an aggregate program P is defined as the set of fixpoints of the consistent stable operator of U_P^{aggr}. We call the well-founded fixpoint of U_P^{aggr} the* ultimate well-founded model *of P and the exact stable fixpoints of U_P^{aggr},* ultimate stable models.

The U_P^{aggr} operator has all the properties of an ultimate approximating operator. The following result is an instance of Proposition 2.22.

Proposition 4.1 *If the T_P^{aggr} operator is monotone then the program P has a two-valued ultimate well-founded model which is also the unique ultimate stable model and is equal to* $\mathrm{lfp}(T_P^{aggr})$.

In particular definite aggregate programs have a monotone T_P^{aggr} operator, so the ultimate semantics agrees with the least model semantics of such programs. There are, however, programs with a monotone immediate consequence operator

which are not definite and for which the well-founded model is three-valued while the ultimate well-founded model is two-valued. One such program is given in Example 2.9. The following example, suggested by John Schlipf, is an example of an aggregate program which is not definite but has a monotone T_P^{aggr} operator. The ultimate well-founded model of this program agrees with the intended interpretation while the less precise semantics of aggregate programs which we define later are too weak (Example 4.7).

Example 4.1 (Borel Sets) Let $\mathbb{R}$ be the set of real numbers. *Borel sets* are defined by the following monotone inductive definition:

- any open set of real numbers is a Borel set;
- for any countable set C of Borel sets, $\bigcap C$ and $\bigcup C$ are Borel sets;
- if B is a Borel set then $\mathbb{R} - B$ is a Borel set.

To model this definition as an aggregate program consider the following aggregate signature

$$\Sigma = \langle \{s\};\ \{open\colon s\};\ \{compl : s \to s\};\ \{\text{INF}_\omega, \text{SUP}_\omega \colon \{s\} \times s\}\rangle.$$

The Σ-structure $\mathcal{D}$ interprets the sort s with the set $\wp(\mathbb{R})$ of all subsets of the real numbers, the predicate *open* is interpreted with the set of open sets, and the function *compl* is interpreted as set complement: $compl^{\mathcal{D}}(S) = \mathbb{R} - S$. The aggregate relations INF_ω and SUP_ω are the restrictions of INF and SUP to countable input multisets. The program defining Borel sets contains a single user-defined predicate $borel\colon s$.

$$\begin{aligned} borel(S) &\leftarrow open(S).\\ borel(compl(S)) &\leftarrow borel(S).\\ borel(S) &\leftarrow \text{INF}_\omega(\{B \mid borel(B)\}, S).\\ borel(S) &\leftarrow \text{SUP}_\omega(\{B \mid borel(B)\}, S). \end{aligned}$$

Note that the aggregate relations INF_ω and SUP_ω are non-monotone, so the program is not definite. However the T_P^{aggr} operator is monotone and $borel(S) \in lfp(T_P^{aggr})$ if and only if S is a Borel set. By Proposition 4.1, the program has a two-valued ultimate well-founded model which is equal to $lfp(T_P^{aggr})$. □

The ultimate semantics also extends the standard model semantics of stratified aggregate programs.

Proposition 4.2 *A stratified aggregate program P has a two-valued ultimate well-founded model which is also the unique ultimate stable model and is equal to the standard model of P.*

We do not provide a direct proof of this proposition. Instead, in Section 4.3.2 we show that the three-valued stable semantics agrees with the semantics of stratified aggregate programs. Since the ultimate well-founded model is more precise than the well-founded fixpoint of any other approximating operator of T_P^{aggr} (Proposition 2.20) the result follows.

Next, we present a translation from an aggregate program to a ground normal logic program which preserves the ultimate semantics.

Definition 4.3 *The translation $lp(P)$ of P is defined as follows:*

$$lp(P) = \{A \leftarrow L \mid A \leftarrow F \in ground(P) \text{ and } L \text{ is a set of literals with atoms from } base_{\mathcal{D}}(\Pi) \text{ such that } L \models F\}.$$

The $lp(\cdot)$ translation preserves the two-valued immediate consequence operators.

Proposition 4.3 $T_P^{aggr}(I) = T_{lp(P)}^{aggr}(I)$

Proof. $\supseteq$) Suppose that $A \in T_{lp(P)}^{aggr}(I)$. This means that there exists a rule $A \leftarrow L \in ground(lp(P)) = lp(P)$ such that $I \models L$. By definition of $lp(P)$ there exists a rule $A \leftarrow F \in ground(P)$ such that $L \models F$. This implies that $I \models F$ and consequently, $A \in T_P^{aggr}(I)$.

$\subseteq$) Suppose that $A \in T_P^{aggr}(I)$. This means that there exists a rule $A \leftarrow F \in ground(P)$ such that $I \models F$. Let L be the set of all literals which are true in I and defined as

$$L = \{A \mid A \in I\} \cup \{\neg A \mid A \notin I\}.$$

Clearly $L \models F$ and so $A \leftarrow L \in lp(P)$ and $A \in T_{lp(P)}^{aggr}(I)$. ∎

If the T_P^{aggr} operators of P and $lp(P)$ are the same then the two programs will have the same ultimate approximating operator U_P^{aggr}. So, we have:

Proposition 4.4 *P and $lp(P)$ have the same set of ultimate three-valued stable models.*

4.2 Three-Valued Stable Model Semantics

One of the concerns with the ultimate three-valued stable semantics is that it has a high computational complexity even for programs without aggregates (Denecker et al., 2004). Computing the ultimate well-founded model is co-NP-hard and deciding the existence of a two-valued ultimate stable model is Σ_2^p-complete. In this section we investigate a family of less precise consistent approximating operators of T_P^{aggr}. Our goal is to extend the three-valued stable semantics of aggregate programs defined by Przymusinski (Przymusinksi, 1990). According

to Approximation Theory (Denecker et al., 2000) this semantics can be obtained as consistent stable fixpoints of the three-valued Fitting operator Φ_P^{aggr} (Fitting, 1985) which is a consistent approximating operator of T_P^{aggr}. By extending this operator to a consistent approximating operator of aggregate programs we obtain a three-valued stable semantics which extends the one of logic programs without aggregates (Przymusinksi, 1990).

We first extend the valuation function of set expressions for three-valued interpretations. The value of a set expression is a three-valued multiset.

Definition 4.4 (Three-Valued Multiset) *A* three-valued multiset *is a pair of multisets* (M_1, M_2) *such that* $M_1 \subseteq M_2$*. A three-valued multiset* (M_1, M_2) *is* finite *if* M_2 *is a finite multiset.*

This definition also covers finite sets and finite three-valued sets because they are special cases of multisets and three-valued multisets.

Definition 4.5 *Let* $\{x_1, \ldots, x_n \mid \varphi\}$ *be a first-order set expression and* $\tilde{I}$ *a three-valued interpretation. The value* $[\![\{x_1, \ldots, x_n \mid \varphi\}]\!]_{\tilde{I}}$ *is a three-valued set* $\tilde{S}\colon D_{s_1} \times \cdots \times D_{s_n} \to \mathcal{THREE}$ *defined as* $\tilde{S}(d_1, \ldots, d_n) = \mathcal{H}_{\tilde{I}}^{\sigma_{x_1=d_1,\ldots,x_n=d_n}}(\varphi)$.

Lemma 4.5 *For every set expression* s *if* $\tilde{I} \leq_p \tilde{J}$ *then* $[\![s]\!]_{\tilde{I}} \leq_p [\![s]\!]_{\tilde{J}}$ *and for every 2-valued interpretation* I*,* $[\![s]\!]_{(I,I)} = ([\![s]\!]_I, [\![s]\!]_I)$.

The value of a lambda expressions $\lambda \mathbf{x}.\ e(x)$ depends only on the interpretation $\mathcal{D}$ of the pre-defined symbols and does not change in three-valued logic. In three-valued logic the combination of a set expression s and a lambda expression f define a three-valued multiset (M_1, M_2) obtained by applying the function $F = [\![f]\!]_{\mathcal{D}}$ to the two components of the three-valued set $(S_1, S_2) = [\![s]\!]_{\tilde{I}}$, i.e. $(M_1, M_2) = (F(S_1), F(S_2))$.

In three-valued logic aggregate symbols are interpreted by three-valued aggregate relations. Instead of providing immediately a definition of three-valued aggregate relations we first give the basic properties which such three-valued aggregate relations have to satisfy such that the corresponding Φ_P^{aggr} operator will approximate T_P^{aggr}. Depending on the particular choice of three-valued aggregate relations we obtain different semantics.

Definition 4.6 (Approximating Aggregate Relation) *Let* $\mathrm{R} \subseteq \mathcal{M}(D_1) \times D_2$ *be an aggregate relation. We say that the three-valued relation* $A_{\mathrm{R}}\colon \mathcal{M}(D_1)^c \times D_2 \to \mathcal{THREE}$ *is an* approximating aggregate relation *of* R *if it satisfies the following properties:*

- A_{R} *extends* R*, i.e.* $A_{\mathrm{R}}((M, M), d) = (\mathrm{R}(M, d), \mathrm{R}(M, d))$*;*
- A_{R} *is* $\leq_p$*-monotone in the first argument, i.e. if* $(M_1, M_2) \leq_p (N_1, N_2)$ *then* $A_{\mathrm{R}}((M_1, M_2), d) \leq_p A_{\mathrm{R}}((N_1, N_2), d)$.

Definition 4.7 (Three-Valued truth function) *The three-valued truth function for first-order formulas* $\lambda(\tilde{I}, \varphi)$. $\mathcal{H}_{\tilde{I}}(\varphi)\colon \mathcal{I}^c \times \mathcal{L} \to \mathcal{THREE}$ *(Definition 2.30) is extended to a* three-valued truth function *for first-order aggregate formulas* $\lambda(\tilde{I}, \varphi)$. $\mathcal{H}^{aggr}_{\tilde{I}}(\varphi)\colon \mathcal{I}^c \times \mathcal{L}^{aggr} \to \mathcal{THREE}$ *as follows:*

$$\mathcal{H}^{aggr}_{\tilde{I}}(\mathrm{R}(\lambda \mathbf{x}.\ e, \{\mathbf{x} \mid \varphi\}, t)) = A_{\mathrm{R}}([\![\lambda \mathbf{x}.\ e]\!](\tilde{I}), [\![t]\!]_{\mathcal{D}})$$

where $A_{\mathrm{R}}\colon \mathcal{M}(D)^c \times D \to \mathcal{THREE}$ *is an approximating aggregate relation of* $\mathrm{R}\colon \mathcal{M}(D) \times D \to \mathcal{TWO}$.

The three-valued truth function $\mathcal{H}^{aggr}_{\tilde{I}}$ is $\leq_p$-monotone and extends the two-valued truth function for aggregate formulas $\mathcal{H}^{aggr}_{I}$.

Lemma 4.6 *For every aggregate formula* φ *if* $\tilde{I} \leq_p \tilde{J}$ *then* $\mathcal{H}^{aggr}_{\tilde{I}}(\varphi) \leq_p \mathcal{H}^{aggr}_{\tilde{J}}(\varphi)$ *and for every 2-valued interpretation* I, $\mathcal{H}^{aggr}_{(I,I)}(\varphi) = (\mathcal{H}^{aggr}_{I}(\varphi), \mathcal{H}^{aggr}_{I}(\varphi))$.

Proof. (Sketch) We only need to prove the statement for aggregate atoms. It follows from Lemma 4.5 and the definition of approximating aggregates. ∎

Using the simplified notation of aggregate atoms without lambda expressions, the above definition simplifies to

$$\mathcal{H}_{\tilde{I}}(\mathrm{R}(s, t)) = A_{\mathrm{R}}([\![s]\!]_{\tilde{I}}, [\![t]\!]_{\mathcal{D}}).$$

We now define an approximating operator Φ^{aggr}_P of T^{aggr}_P as follows.

Definition 4.8 *The* three-valued immediate consequence operator $\Phi^{aggr}_P\colon \mathcal{I}^c \to \mathcal{I}^c$ *for an aggregate program* P *is defined as:*

$$\Phi^{aggr}_P(\tilde{I})(A) = \bigvee\nolimits_t \{\mathcal{H}^{aggr}_{\tilde{I}}(B) \mid A \leftarrow B \in ground(P)\}.$$

Proposition 4.7 *The* Φ^{aggr}_P *operator is a consistent approximating operator of* T^{aggr}_P.

Proof. Follows from Lemma 4.6. ∎

Definition 4.9 *The set of* three-valued stable models *(or* partial stable models*) of an aggregate program* P *are defined as the set of fixpoints of the consistent stable operator* $S^c_{\Phi^{aggr}_P}$ *of* Φ^{aggr}_P *(see Section 2.6.3).*

For logic programs without aggregates the Φ^{aggr}_P operator coincides with the Fitting Φ_P operator (Fitting, 1985) and the Ψ_P operator defined by Przymusinski (Przymusinksi, 1990). So, the three-valued stable semantics of aggregate programs is an extension of the three-valued stable models of normal logic programs (Przymusinksi, 1990).

4.2.1 Aggregate Programs with Finite Multisets

All standard aggregates MIN, MAX, CARD, SUM, PROD, and AVG are not well-defined for most multisets of infinite size. On the other hand, with the exception of AVG on the empty multiset, all these aggregates are total functions on finite multisets. So, the algorithms, properties, and program transformations of these aggregates which we study are valid only for finite multisets. In this section we investigate properties of aggregate programs which guarantee that the input of the aggregates are finite multisets and, consequently, we can apply the results which we have obtained.

Definition 4.10 *An* aggregate atom with finite multisets *is a* $(\Sigma(\Pi), \mathcal{D})$*-aggregate atom of the form* $\mathrm{R}(\{\mathbf{x} \mid \varphi\}, t)$ *such that for every three-valued interpretation* $\tilde{I}$*,* $[\![\{\mathbf{x} \mid \varphi\}]\!]_{\tilde{I}}$ *is a finite three-valued set. An* aggregate program with finite multisets *is an aggregate program* P *such that every aggregate atom is an aggregate atom with finite multisets.*

The conditions of the set expressions of the aggregate atoms of most of typical aggregate programs have a common form.

Example 4.2 Consider the program of the Party Invitation II problem from Example 3.8:

$$accept(X) \leftarrow thr(X, K) \wedge \\ \mathrm{SUM}_{\geq}(\lambda(Y, P).\ P, \{(Y, P) \mid comp(X, Y, P) \wedge accept(Y)\}, K).$$

Note that $comp\colon pers \times pers \times real$ is a pre-defined relation which is always finite. This is because the third argument is functionally dependent on the first two arguments and the sort $pers$ is interpreted with a finite domain. Note also that $accept\colon pers$ is a user-defined predicate which is again interpreted over a finite domain. Thus the interpretation of the set expression $\{(Y, P) \mid comp(X, Y, P) \wedge accept(Y)\}$ is always a finite three-valued set. □

The intuition from this example is formalized by the following proposition.

Proposition 4.8 *An aggregate atom* $\mathrm{R}(\{\mathbf{x} \mid \varphi\}, t)$ *is an aggregate atom with finite multisets if every predicate symbol* $p\colon s_1 \times \cdots \times s_n$ *of the condition* φ *is either a pre-defined predicate symbol with finite extension (i.e.* $p^{\mathcal{D}}$ *is finite) or* p *is a user-defined predicate and all sorts* $s_1, \ldots, s_n$ *are interpreted with finite domains.*

The programs of most of the problems we have considered fall in the class of aggregate programs with finite multisets. This is the case for the two versions of the Party Invitation problem (Example 3.7 and Example 3.8). Some other programs needs to be formulated in a slightly different way to become aggregate programs with finite multisets.

Example 4.3 The program of the Company Controls problem from Example 3.5 is not an aggregate program with finite multisets. To see this consider the rule containing the aggregate atom:

$$controls(X,Y) \leftarrow \text{SUM}_{>}(\lambda(Z,S).\ S, \{(Z,S) \mid cv(X,Z,Y,S)\}, 0.5).$$

The user-defined predicate cv has a type $comp \times comp \times comp \times shares$. So, in the least interpretation $\tilde{I} = (\emptyset, base_{\mathcal{D}}(\Pi))$ in the precision order $\leq_p$,

$$[\![\{(Z,S) \mid cv(X,Z,Y,S)\}]\!]_{\tilde{I}} = (\emptyset, \{(c,x) \mid c \in D_{comp}, x \in D_{shares} = [0,1]\}).$$

The over-estimate of the three-valued set is obviously an infinite set. □

The program can be reformulated in a such a way as to have only finite multisets. The idea is simply to drop the last argument of the cv predicate.

Example 4.4 (Company Controls with Finite Multisets) Consider the following formulation of the Company Control problem from Example 3.5. The user-defined predicate cv has a type $comp \times comp \times comp$ and $controls$ has a type $comp \times comp$.

$$\begin{aligned} cv(X,X,Y) &\leftarrow \exists S.\ owns_stock(X,Y,S). \\ cv(X,Z,Y) &\leftarrow controls(X,Z) \wedge \exists S.\ owns_stock(Z,Y,S). \end{aligned}$$

$$\begin{aligned} controls(X,Y) \leftarrow \text{SUM}_{>}(\ & \{\lambda(Z,S).\ S, \\ & (Z,S) \mid cv(X,Z,Y) \wedge owns_stock(Z,Y,S)\} \\ & 0.5). \end{aligned}$$

This formulation of the problem is clearly an aggregate program with finite multisets. □

Finally, there are problems which cannot be formulated as aggregate programs with finite multisets. This is the case, for example, with the Shortest Path problem.

Example 4.5 Consider the rule for the $sp\colon n \times n \times c$ predicate from the programs of the Shortest Path problem from Example 3.6 and Example 4.6.

$$sp(X,Y,S) \leftarrow \text{MIN}(\lambda C.\ C, \{C \mid cp(X,Y,C)\}, S).$$

If the cost-domain c is interpreted with an infinite domain then for the interpretation $\tilde{I} = (\emptyset, base_{\mathcal{D}}(\Pi))$,

$$[\![\{C \mid cp(X,Y,C)\}]\!]_{\tilde{I}} = (\emptyset, \{n \mid n \in D_c\}).$$

In fact, it is possible to have graphs with infinitely many paths if the graph is infinitely large or if there is a cycle with non-negative length in a path between two nodes. □

4.2.2 Splitting Theorem

In Section 3.3.2 we applied the theory of stratification of operators to the T_P^{aggr} operator to define a standard model semantics of (weakly) stratified aggregate programs. We now develop a different application of this theory by considering a stratification of the consistent approximating operator Φ_P^{aggr}. It is analogous to the splitting theorem in logic programming (Lifschitz and Turner, 1994). Similarly to the stratification of an aggregate program, the splitting of a program agrees with the dependency graph. However, the conditions are weaker and a single stratum may contain all types of cycles. We also restrict our attention to index sets which are well-ordered instead of well-founded. The splitting theorem for aggregate programs which we show is used to prove that the weakly stratified aggregate programs have a single three-valued stable model which is equal to the standard model. The results hold for any choice of an approximating aggregate.

Definition 4.11 *Let P be a $\Sigma(\Pi)$-aggregate program. A* splitting sequence *of length μ of P is a collection $\langle \Pi_\alpha \rangle_{\alpha<\mu}$ of disjoint subsets of Π such that $\bigcup_{\alpha<\mu} \Pi_\alpha = \Pi$ and for every rule $p(\mathbf{t}) \leftarrow \varphi \in P$ if $p \in \Pi_\alpha$ then the set of predicate symbols of φ are in $\bigcup_{\beta\leq\alpha} \Pi_\beta$.*

For a splitting sequence $\langle P_\alpha \rangle_{\alpha<\mu}$ of P let $\mathcal{I}_\alpha = \wp(base_\mathcal{D}(\Pi_\alpha))$. Clearly $\mathcal{I} = \bigotimes_{\alpha<\mu} \mathcal{I}_\alpha$. Also, for a three-valued interpretation $\tilde{I} = (I_1, I_2)$ let $\tilde{I}|_\alpha = (I_1 \cap base_\mathcal{D}(\Pi_\alpha), I_2 \cap base_\mathcal{D}(\Pi_\alpha))$, i.e. the restriction of $\tilde{I}$ only to atoms with predicates in Π_α.

Lemma 4.9 *Let $\langle \Pi_\alpha \rangle_{\alpha<\mu}$ be a splitting sequence of an aggregate program P. Then $\langle \mathcal{I}_\alpha \rangle_{\alpha<\mu}$ is a stratification of the approximating operator $\Phi_P^{aggr} : \mathcal{I}^c \to \mathcal{I}^c$.*

Proof. We use the characterization of stratifiability of Proposition 2.24. Let $\alpha < \mu$ and $\tilde{U} \in \mathcal{I}^c|_{<\alpha}$. Define the component $(\Phi_P^{aggr})_\alpha^{\tilde{U}} : \mathcal{I}_\alpha^c \to \mathcal{I}_\alpha^c$ as follows

$$(\Phi_P^{aggr})_\alpha^{\tilde{U}}(\tilde{J}) = \Phi_{P_\alpha}^{aggr}(\tilde{U} \otimes \tilde{J}).$$

Let $\tilde{I} \in \mathcal{I}^c$ be a three-valued interpretation which agrees with $\tilde{U}$, i.e. $\tilde{I}|_{\prec\alpha} = \tilde{U}$.

$$\begin{aligned}
\Phi_P^{aggr}(\tilde{I})|_\alpha &= \Phi_{P_\alpha}^{aggr}(\tilde{I}) \\
&= \Phi_{P_\alpha}^{aggr}(\tilde{I}|_{\preceq i}) \\
&= \Phi_{P_\alpha}^{aggr}(\tilde{U} \otimes \tilde{I}|_\alpha) && \text{because } \tilde{I}|_{\prec\alpha} = \tilde{U} \\
&= (\Phi_P^{aggr})_\alpha^{\tilde{U}}(\tilde{I}|_\alpha) && \text{by definition of } (\Phi_P^{aggr})_\alpha^{\tilde{U}}
\end{aligned}$$ ∎

We note that for normal logic programs in which the rules are conjunctions of literals the component $(\Phi_P^{aggr})_\alpha^{\tilde{U}}$ can be defined as $\Phi_{P/\tilde{U}}^{aggr}$ where $P/\tilde{U}$ is the program obtained from $ground(P)$ by substituting all atoms at strata lower than α with

their truth values given by $\tilde{U}$. However, this is not possible for logic programs which contain first-order formulas in the rule bodies. In particular, if a formula contains quantifiers or set expressions then some atoms may have variables in their arguments in $ground(P)$. So, such atoms cannot be substituted with their truth value.

Theorem 4.10 (Splitting Theorem) *Let $\langle \Pi_\alpha \rangle_{\alpha \leq \mu}$ be a splitting sequence of an aggregate program P. A three-valued interpretation $\tilde{I}$ is a three-valued stable model of P if and only if for every $\alpha < \mu$, $\tilde{I}|_\alpha$ is a consistent stable fixpoint of the component $(\Phi_P^{aggr})_\alpha^{\tilde{I}|_{\prec\alpha}}$.*

Proof. Follows form Lemma 4.9 and Proposition 2.28. ∎

To explain how the splitting theorem is used to characterize three-valued stable models let $\langle \Pi_\alpha \rangle_{\alpha<\mu}$ be a splitting sequence of an aggregate program P. In the first step we compute the set of consistent stable fixpoints of the operator $(\Phi_P^{aggr})_0$ which correspond to three-valued stable models of the program P_0. The three-valued stable models of a stratum $\alpha+1$ are defined as the union of the three-valued stable fixpoints of the $(\Phi_P^{aggr})_{\alpha+1}^{\tilde{U}}$ operator for every three-valued stable model $\tilde{U}$ of P_α.

Unlike stratified programs for which every stratum has a single stable model (which is the least fixpoint of the T_P^{aggr} operator on this stratum), the strata defined by a splitting sequence $\langle \Pi_\alpha \rangle_{\alpha<\mu}$ may have several three-valued stable models.

4.2.3 Related Work

A definition of a well-founded and valid semantics based on three-valued multisets and three-valued aggregate functions has already been proposed by Sudarshan et al. (Sudarshan et al., 1993). Our definition of three-valued multisets is equivalent to the one given in (Sudarshan et al., 1993). Also, the way three-valued multisets are computed in (Sudarshan et al., 1993) is the same as the valuation of a set expression in a three-valued interpretation. Finally, aggregate functions and relations in (Sudarshan et al., 1993) are defined for three-valued multisets and have to be monotone. So, they are similar to the approximating aggregate relations which we consider

The syntax of logic programs with first-order rule bodies which we use is similar to the syntax of logic programs with nested expressions (Lifschitz et al., 1999). The first visible difference is that the later is purely propositional and does not support quantifiers. On the other hand programs with nested expressions can have disjunction and nested formulas also in the head of the rules. A more important difference is that the logical foundations of the stable semantics of the two languages are very different. While our semantics is based on evaluating rule bodies in

Kleene's strong three-valued logic $\mathcal{THREE}$, the one of nested expression is based on the logic of here-and-there, HT (Lifschitz et al., 2001). A simple example of the difference between the two logics is that $\neg\neg p$ is equivalent in $\mathcal{THREE}$ to p while it is not in the logic of here-and-there.

We postpone the discussion of several recent extensions of the stable semantics of logic programs with weight constraints (Simons et al., 2002), set constraints (Marek and Remmel, 2004) and monotone cardinality atoms (Marek et al., 2004) to Chapter 6 because these extensions are defined for a propositional language.

4.3 The Three-valued Stable Semantics based on the Ultimate Approximating Aggregate

The three-valued truth function of aggregate atoms (Definition 4.7) and consequently, the three-valued stable semantics of aggregate programs are parametrized by the interpretation of aggregate symbols by approximating aggregate relations. In the next chapter we investigate in detail the semantics obtained by interpreting aggregate symbols with the most precise approximating aggregate relation, called the *ultimate approximating aggregate*. This three-valued aggregate relation is defined for all aggregate relations in a uniform way using the construction of ultimate approximations (Section 2.6.4). In this section we show how this semantics is related with the semantics of definite and stratified aggregate programs.

Definition 4.12 (Ultimate Approximating Aggregate) *Let* $\mathrm{R} \subseteq \mathcal{M}(D) \times D$ *be an aggregate relation. The* ultimate approximating aggregate *of* R *is a function* $U_{\mathrm{R}} \colon \mathcal{M}(D)^c \times D \to \mathcal{THREE}$. *It is defined as*

$$\mathrm{R}((M_1, M_2), d) = (U^1_R((M_1, M_2), d), U^2_{\mathrm{R}}((M_1, M_2), d))$$

where $U^1_{\mathrm{R}}, U^2_{\mathrm{R}} \colon \mathcal{M}(D)^c \times D \to \mathcal{TWO}$ *are the projections of* U_{R} *on the first and second component, defined as follows:*

$$U^1_{\mathrm{R}}((M_1, M_2), d) = \mathbf{t} \text{ if and only if } \forall M \in [M_1, M_2] \colon (M, d) \in \mathrm{R}$$
$$U^2_{\mathrm{R}}((M_1, M_2), d) = \mathbf{t} \text{ if and only if } \exists M \in [M_1, M_2] \colon (M, d) \in \mathrm{R}$$

Proposition 4.11 U_{R} *is an approximating aggregate relation of* R.

We first restate some results which have already been proven for the ultimate approximating operator.

The ultimate approximating aggregate U_{R} is the most precise in the $\leq_p$-order among all possible approximating aggregate relations.

Proposition 4.12 $A_{\mathrm{R}}((M_1, M_2), d) \leq_p U_{\mathrm{R}}((M_1, M_2), d)$ *for every approximating aggregate relation* A_{R} *of* R, *three-valued multiset* (M_1, M_2), *and element* $d \in D$.

For monotone and anti-monotone aggregates the truth value can be computed directly on the boundary multisets.

Proposition 4.13 *Let* $\mathrm{R}\colon \mathcal{M}(D_1) \times D_2$ *be a monotone aggregate relation. Then* $((M_1, M_2), d) \in U^1_{\mathrm{R}}$ *if and only if* $(M_1, d) \in \mathrm{R}$ *and* $((M_1, M_2), d) \in U^2_{\mathrm{R}}$ *if and only if* $(M_2, d) \in \mathrm{R}$.

Proposition 4.14 *Let* $\mathrm{R}\colon \mathcal{M}(D_1) \times D_2$ *be an anti-monotone aggregate relation. Then* $((M_1, M_2), d) \in U^1_{\mathrm{R}}$ *if and only if* $(M_2, d) \in \mathrm{R}$ *and* $((M_1, M_2), d) \in U^2_{\mathrm{R}}$ *if and only if* $(M_1, d) \in \mathrm{R}$.

4.3.1 Examples

The following formulation of the shortest path problem is a natural example of a program which is not stratified and uses recursion over non-monotone aggregate relation.

Example 4.6 (Shortest Path) Let $\Sigma = \langle \{n, c\};\ \{edge\colon n \times n \times c\};\ \emptyset \rangle$ be the signature of directed weighted graphs from Example 3.6. A tuple $edge(X, Y, L)$ represents an edge from X to Y with length L. Consider the following formulation of the problem of finding the shortest path which can be found in (Van Gelder, 1992, Example 4.1).

$$sp(X, Y, S) \leftarrow \textsc{min}(\lambda C.\ C, \{C \mid cp(X, Y, C)\}, S).$$

$$cp(X, Y, C) \leftarrow edge(X, Y, C).$$
$$cp(X, Y, C_1 + C_2) \leftarrow sp(X, Z, C_1) \wedge edge(Z, Y, C_2).$$

The only difference between this program and the formulation of the shortest path in Example 3.6 is that we have replaced the call to $cp/3$ in the second clause of $cp/3$ with a call to $sp/3$. We have incorporated the knowledge that any shortest path of length $n+1$ must be an extension of a shortest path of length n. This fact is the basis of Dijkstra's algorithm. However, the program is no longer stratified because the $sp/3$ predicate depends on itself through the MIN aggregate relation which is non-monotone. Moreover, the program may have a three-valued well-founded model. In particular if there exists a path between a node a and a node b which contains a loop with a negative length then $sp(a, b, w)$ will be undefined in the well-founded model for any path between a and b with length w. This is also the case for the ultimate well-founded model.

Even graphs with positive weights can contain a three-valued well-founded model. There is an example of an infinite graph in (Ross and Sagiv, 1997) in which for two nodes a and b the set

$$W_{ab} = \{w \mid \text{there is a path between } a \text{ and } b \text{ with length } w\}$$

is unfounded. In such case all atoms $sp(a, b, w)$ with $w \in W_{ab}$ will be undefined in the well-founded model of the above program. □

The next example shows that program defining Borel sets which has a monotone T_P^{aggr} operator but is not definite, even the three-valued stable semantics based on the ultimate approximating aggregate is too weak.

Example 4.7 (Borel Sets) Reconsider the program from Example 4.1:

$$\begin{aligned} borel(S) &\leftarrow open(S). \\ borel(compl(S)) &\leftarrow borel(S). \\ borel(S) &\leftarrow \text{INF}_\omega(\{B \mid borel(B)\}, S). \\ borel(S) &\leftarrow \text{SUP}_\omega(\{B \mid borel(B)\}, S). \end{aligned}$$

The well-founded model of the program is (I_1, I_2) where

$$\begin{aligned} I_1 &= \{borel(S) \mid S \text{ is an open set or a complement of an open set}\}, \\ I_2 &= \{borel(S) \mid S \text{ is a Borel set}\}. \end{aligned}$$

The under-estimate I_1 is the least fixpoint of the definite program consisting only of the first two rules. Let $(S_1, S_2) = [\![\{B \mid borel(B)\}]\!]_{(I_1,I_2)}$ and $(J_1, J_2) = \Phi_P^{aggr}(I_1, I_2)$. To see why the model (I_1, I_2) cannot be improved, i.e. $(I_1, I_2) = (J_1, J_2)$, consider a subset $s \subseteq \mathbb{R}$ in the half-open interval $(S_1, S_2]$. We have that $borel(s) \in J^1$ if and only if

$$\mathcal{H}^1_{(I_1,I_2)}(\text{INF}_\omega(\{B \mid borel(B)\}, s)) = U^1_{\text{INF}_\omega}((S_1, S_2), s) = \mathbf{t}$$

or

$$\mathcal{H}^1_{(I_1,I_2)}(\text{SUP}_\omega(\{B \mid borel(B)\}, s)) = U^1_{\text{SUP}_\omega}((S_1, S_2), s) = \mathbf{t}.$$

However, both $U^1_{\text{INF}_\omega}((S_1, S_2), s)$ and $U^1_{\text{SUP}_\omega}((S_1, S_2), s)$ are false because it is not the case that for all countable sets $S \in [S_1, S_2]$, $\bigcap S = s$ or $\bigcup S = s$. □

4.3.2 Definite and Weakly Stratified Aggregate Programs

We now show that definite aggregate programs have a single three-valued stable model which is total.

The key to the proof of this result is that in a three-valued interpretation (I_1, I_2), the under-estimate $\mathcal{H}^1_{(I_1,I_2)}(\varphi)$ of the truth-value of a positive formula φ depends only on I_1 and the over-estimate $\mathcal{H}^2_{(I_1,I_2)}(\varphi)$ depends only on I_2.

Lemma 4.15 *Let (I_1, I_2) be a three-valued interpretation. If φ is a positive aggregate formula then $\mathcal{H}_{(I_1,I_2)}(\varphi) = (\mathcal{H}_{I_1}(\varphi), \mathcal{H}_{I_2}(\varphi))$. If φ is a negative aggregate formula then $\mathcal{H}_{(I_1,I_2)}(\varphi) = (\mathcal{H}_{I_2}(\varphi), \mathcal{H}_{I_1}(\varphi))$.*

Proof. By induction on the structure of φ. We give the proof only for positive aggregate formulas. The proof for negative aggregate formulas is symmetric.

- for a pre-defined atom A, $\mathcal{H}_{(I_1,I_2)}(A)$ does not depend on (I_1, I_2);
- for a user defined atom $p(t_1, \ldots, t_n)$:

$$\begin{aligned}\mathcal{H}_{(I_1,I_2)}(p(t_1,\ldots,t_n)) &= (I_1,I_2)(p([\![t_1]\!],\ldots,[\![t_m]\!])) \\ &= (I_1(p([\![t_1]\!],\ldots,[\![t_m]\!])), I_2(p([\![t_1]\!],\ldots,[\![t_m]\!])));\end{aligned}$$

- for a formula with negation $\neg\varphi$:

$$\begin{aligned}\mathcal{H}_{(I_1,I_2)}(\neg\varphi) &= \neg\mathcal{H}_{(I_1,I_2)}(\varphi) && \text{by Definition 2.30} \\ &= \neg(\mathcal{H}_{I_2}(\varphi), \mathcal{H}_{I_1}(\varphi)) && \text{by inductive hypothesis} \\ &= (\neg\mathcal{H}_{I_1}(\varphi), \neg\mathcal{H}_{I_2}(\varphi)) && \text{by definition of } \neg \text{ (Ex. 2.7)} \\ &= (\mathcal{H}_{I_1}(\neg\varphi), \mathcal{H}_{I_2}(\neg\varphi)); && \text{by Definition 2.17}\end{aligned}$$

- for a conjunction $\varphi \wedge \psi$:

$$\begin{aligned}\mathcal{H}_{(I_1,I_2)}(\varphi\wedge\psi) &= \mathcal{H}_{(I_1,I_2)}(\varphi) \wedge_t \mathcal{H}_{(I_1,I_2)}(\psi) \\ &= (\mathcal{H}^1_{(I_1,I_2)}(\varphi) \wedge \mathcal{H}^1_{(I_1,I_2)}(\psi), \mathcal{H}^2_{(I_1,I_2)}(\varphi) \wedge \mathcal{H}^2_{(I_1,I_2)}(\psi)) \\ &= (\mathcal{H}_{I_1}(\varphi) \wedge \mathcal{H}_{I_1}(\psi), \mathcal{H}_{I_2}(\varphi) \wedge \mathcal{H}_{I_2}(\psi)) \\ &= (\mathcal{H}_{I_1}(\varphi\wedge\psi), \mathcal{H}_{I_2}(\varphi\wedge\psi));\end{aligned}$$

- the proofs for formulas of the form $\varphi \vee \psi$, $\exists x.\ \varphi$, and $\forall x.\ \varphi$ are analogous;
- for an aggregate atom $\mathrm{R}(\{\mathbf{x} \mid \varphi\}, t)$ we consider only the case when $\mathrm{R}^{\mathcal{D}}$ is a monotone aggregate relation and φ is a positive aggregate formula. First, note that $\forall\mathbf{x}.\ \varphi$ is also positive, so

$$\mathcal{H}_{(I_1,I_2)}(\forall\mathbf{x}.\ \varphi) = (\mathcal{H}_{I_1}(\forall\mathbf{x}.\ \varphi), \mathcal{H}_{I_2}(\forall\mathbf{x}.\ \varphi)).$$

Consequently,

$$[\![\{\mathbf{x} \mid \varphi\}]\!]_{(I_1,I_2)} = ([\![\{\mathbf{x} \mid \varphi\}]\!]_{I_1}, [\![\{\mathbf{x} \mid \varphi\}]\!]_{I_2}).$$

Let $d = [\![t]\!]$. We have:

$$\begin{aligned}\mathcal{H}_{(I_1,I_2)}(\mathrm{R}(\{\mathbf{x} \mid \varphi\}, t)) &= U_{\mathrm{R}}([\![\{\mathbf{x} \mid \varphi\}]\!]_{(I_1,I_2)}, d) \\ &= U_{\mathrm{R}}(([\![\{\mathbf{x} \mid \varphi\}]\!]_{I_1}, [\![\{\mathbf{x} \mid \varphi\}]\!]_{I_2}), d) \\ \text{(by Proposition 4.13)} \quad &= (\mathrm{R}([\![\{\mathbf{x} \mid \varphi\}]\!]_{I_1}, d), \mathrm{R}([\![\{\mathbf{x} \mid \varphi\}]\!]_{I_2}, d)) \\ &= (\mathcal{H}_{I_1}(\mathrm{R}(\{\mathbf{x} \mid \varphi\}, t)), \mathcal{H}_{I_2}(\mathrm{R}(\{\mathbf{x} \mid \varphi\}, t))).\end{aligned}$$

■

Theorem 4.16 *Let P be a definite aggregate program. Then P has a single three-valued stable model (M, M) which is two-valued and is the well-founded model. Moreover $M = \mathrm{lfp}(T_P^{aggr})$.*

Proof. From Lemma 4.15 follows that if P is a positive aggregate program then $\Phi_P^{aggr}(I_1, I_2) = (T_P^{aggr}(I_1), T_P^{aggr}(I_2))$. The statement of the theorem follows from Proposition 2.13. ∎

Next we show that the three-valued stable semantics extends the semantics of weakly stratified aggregate programs.

Theorem 4.17 *Let P be a weakly stratified aggregate program. Then P has a single three-valued stable model (M_P, M_P) which is two-valued and is the well-founded model where M_P is the standard model of P.*

Proof. (Sketch) By a combination of Theorem 4.10 (the Splitting Theorem) and Theorem 4.16. ∎

4.3.3 Related Work

The idea of ultimate approximations is closely related to the work on supervaluations initiated by van Fraassen. Aggregate relations are very similar to generalized quantifiers. An extensive study of generalized quantifiers in three-valued logic is done by van Eijck (van Eijck, 1996). Our definition of ultimate approximating aggregates is the same as supervaluation quantifiers of (van Eijck, 1996).

Aggregate Atoms as Negative Literals

A common way to define stable semantics of aggregate programs is by a reduct of a program where aggregate atoms are treated as negative literals. This was first done by Kemp and Stuckey (Kemp and Stuckey, 1991) for normal logic programs and later used for the semantics of ASet-Prolog (Gelfond, 2002; Heidt, 2001) for logic programs with subset and aggregate atoms and for the stable semantics of disjunctive logic programs used by the **dlv** system (Dell'Armi et al., 2003b). We make a comparison for the common subset of our syntax and that of the other works, namely normal aggregate programs.

The *reduct* of a normal aggregate program P with respect to an interpretation I is a program P^I obtained from $ground(P)$ by:

- deleting all rules $r \in ground(P)$ for which a negative literal or an aggregate atom in the body of r is false in I;
- deleting all negative literals and aggregate in the remaining rules.

The reduct P^I of P is thus a definite normal logic program with a monotone immediate consequence operator. An interpretation I is a *R-stable model*[1] of P if and only if $I = \text{lfp}(T_{P^I})$.

As pointed our already by (Kemp and Stuckey, 1991) a program can have non-minimal stable models.

Example 4.8 Consider the following (definite) aggregate program:

$$p(a) \leftarrow \text{CARD}_{\geq}(\{X \mid p(X)\}, 1).$$

This program has two R-stable models: $\emptyset$ and $\{p(a)\}$, one of which ($\{p(a)\}$) is not minimal. On the other hand under our semantics it has a single stable model $\emptyset$. Note also that this is a definite aggregate program with monotone T_P^{aggr} operator with a least fixpoint $\emptyset$. □

Thus, for programs with monotone T_P^{aggr} operator for which the least fixpoint of T_P^{aggr} is the intended model, R-stable models are not adequate semantics.

Example 4.9 Consider the Company Controls problem from Example 3.5 which has a monotone T_P^{aggr} operator. Consider an input with three companies $D_{comp} = \{a, b, c\}$ where every company owns 30% shares of the other two companies, i.e. the graph defined by the *owns_stock* relation is complete. Clearly, no company controls any other company. The program has a single three-valued stable model which is two-valued and all atoms are false. On the other hand the program has 8 R-stable models. For every company $x \in D_{comp}$ we have one R-stable model in which $controls(x, y)$ is false for all $y \in D_{comp}$ and another R-stable model in which $controls(x, y)$ is true for all companies $y \in D_{comp}$. For example, the interpretation of the *controls* predicate by one R-stable model is

$$\{controls(a, a), controls(a, b), controls(a, c)\}.$$

Clearly, this is not an intended model. □

As the above examples illustrates the stable semantics which we define and that of (Dell'Armi et al., 2003b; Heidt, 2001; Kemp and Stuckey, 1991) can disagree even for definite aggregate programs. However, we can establish a correspondence for aggregate programs which use only negative aggregate atoms (as in Definition 3.15).

Proposition 4.18 *If an aggregate program P contains only negative aggregate atoms then exact stable models coincide with R-stable models.*

[1] We call stable models of (Dell'Armi et al., 2003b; Heidt, 2001; Kemp and Stuckey, 1991) R-stable (R from reduct) to distinguish the notion from stable models which we define.

Proof. (Sketch) Let (I_1, I_2) be a three-valued interpretation. We will show that $T_{P^{I_2}}(I_1) = (\Phi_P^{aggr})^1(I_1, I_2)$:

$$A \in T_{P^{I_2}}(I_1) \qquad (*)$$

if and only if there exists a rule $r \in ground(P)$ such that $A = head(r)$, $I_2 \models aggr(r) \wedge neg(r)$, and $I_1 \models pos(r)$. Because $aggr(r)$ consists only of negative aggregate atoms then by Lemma 4.15, the last two statements are equivalent to $\mathcal{H}^1_{(I_1,I_2)}(pos(r) \wedge aggr(r) \wedge neg(r)) = \mathbf{t}$. Continuing the proof we obtain that (*) is true if and only if $\bigvee\{\mathcal{H}^1_{(I_1,I_2)}(body(r)) \mid A = head(r) \text{ and } r \in ground(P)\}$ if and only if $A \in (\Phi_P^{aggr})^1(I_1, I_2)$. ∎

4.4 Stable Semantics of Disjunctive Aggregate Programs

In this section we extend the stable model semantics for disjunctive logic programs (Gelfond and Lifschitz, 1991; Przymusinski, 1991) to programs with aggregates. Unfortunately, approximation theory (Denecker et al., 2000; Denecker et al., 2004) cannot be applied to this class of programs. We develop a novel algebraic characterization of the minimal model and stable semantics of disjunctive logic programs based on non-deterministic operators.

4.4.1 Preliminaries

A *disjunctive aggregate rule* has the form

$$A_1 \vee \cdots \vee A_m \leftarrow L_1 \wedge \cdots \wedge L_n$$

where $m \geq 1$, $n \geq 0$, A_i are atoms with user-defined predicate symbols from Π and L_i are $\Sigma(\Pi)$-literals. A disjunctive aggregate program is a (possibly infinite) set of *disjunctive aggregate rules*. We also use the notation $head(r)$ and $body(r)$ to denote the head and the body of a disjunctive rule r. We say that a disjunctive aggregate program P is *definite* if it contains only positive literals and positive aggregate atoms in the rule bodies. A *positive clause* over a set of atoms At is an expression $A_1 \vee \ldots \vee An$ where $Ai \in At$. The *disjunctive base* $base^{\vee}_{\mathcal{D}}(\Pi)$ of a program P is defined as the set of all finite positive clauses with atoms from $base_{\mathcal{D}}(\Pi)$. For a set of clauses C, let $At(C)$ denote the set of all atoms in C.

An interpretation I *satisfies* a positive clause $A_1 \vee \ldots \vee A_n$, denoted with $I \models A_1 \vee \ldots \vee A_n$, if $A_i \in I$ for some i between 1 and n. An interpretation I *satisfies* a ground rule r, denoted with $I \models r$, if $I \models body(r)$ implies $I \models head(r)$. An interpretation I is a *model* of a program P, denoted with $I \models P$, if $I \models r$ for every rule $r \in ground(P)$. A model I of P is *minimal* if there does not exist a model N of P such that $N \subset M$. The set of all models of a program P is

denoted with $Mod(P)$ and of all minimal models with $MM(P)$. We say that an interpretation I *covers* an interpretation J if $I \subseteq J$.

4.4.2 Non-deterministic Operators

Let L be a poset. A *non-deterministic operator* on L is an operator $N : L \to \wp(L)$.

There are many possible ways to extend the partial order relation $\leq$ on L to an order of subsets of L. The one which is relevant for us is the *Smyth pre-order* (Smyth, 1978) denoted with $\leq^S$ and defined as follows:

$$\textit{Smyth pre-order:} \quad A \leq^S B \text{ if and only if } \forall b \in B.\ \exists a \in A.\ a \leq b$$

If $A \leq^S B$ we say that A *covers* B. If $\leq^S$ is restricted to sets which are anti-chains (contain only minimal elements) then it is a partial order relation, otherwise it is a pre-order, i.e. it is not anti-symmetric. This is illustrated by the following example.

Example 4.10 Consider the poset $\langle \mathbb{N}, \leq \rangle$ where $\leq$ is the standard order on the set of natural numbers $\mathbb{N}$. Take the sets $A = \{1\}$ and $B = \{1, 2\}$. It is the case that $A \leq^S B$ and $B \leq^S A$ but clearly $A \neq B$. □

A *fixpoint* of a non-deterministic operator N is an element x such that $x \in N(x)$. The set of all fixpoints of N is denoted with $\mathrm{fp}(N)$ and the set of all minimal fixpoints of N with $\mathrm{mfp}(N)$. A *pre-fixpoint* of N (in the Smyth-order $\leq^S$) is an element x such that $N(x) \leq^S \{x\}$, i.e. there exists $y \in N(x)$ such that $y \leq x$. The set of all such pre-fixpoints is denoted with $\mathrm{pre}^S(N)$.

The following basic result about pre-fixpoints of $\leq^S$-monotone non-deterministic operators is used later.

Lemma 4.19 *Let $N : L \to 2^L$ be a $\leq^S$-monotone non-deterministic operator on a poset L. If x is a minimal pre-fixpoint of N then x is also a fixpoint of N.*

Proof. Let x be a pre-fixpoint of N, i.e. there exists $y \in N(x)$ such that $y \leq x$. If we assume that x is not a fixpoint of N then $x \notin N(x)$ which implies $y < x$. From the $\leq^S$-monotonicity of N and $y < x$ follows that for $y \in N(x)$ there exists $z \in N(y)$ such that $z \leq y$. So, y is also a pre-fixpoint of N which is strictly smaller than x. This contradicts with the assumption that x is a minimal pre-fixpoint. ■

4.4.3 Immediate Consequence Operators of Disjunctive Programs

Our first task is to study immediate consequence operators for disjunctive aggregate programs and establish a correspondence between different types of fixpoints of the operators and different types of models of the program. The presence of

disjunction in the head of rules means that when the body of a rule is satisfied by an interpretation there is a choice between all atoms in the head. This suggests that we have to consider non-deterministic operators which for a single interpretation return a set of interpretations. Instead of giving a precise definition of such operator we study a family of operators which satisfy some common requirements. For every interpretation in the result of the operator we leave open the choice how many atoms from the heads of the satisfied rules are selected as long as for every atom in every head there is at least one interpretation which covers it. It turns out that all operators which satisfy our requirements have the same basic properties.

The operators are defined in two steps. In the first step we collect the set of the heads of all rules whose bodies are satisfied by an input interpretation I.

Definition 4.13 *The* head reduct *of a disjunctive aggregate program P is a function*

$$R_P^{aggr} : \mathcal{I} \to \wp(base_{\mathcal{D}}^{\vee}(\Pi))$$

defined as:

$$R_P^{aggr}(I) = \{head(r) \mid r \in ground(P) \text{ and } I \models body(r)\}$$

In the second step, we use a function which returns some set of models of $R_P^{aggr}(I)$ satisfying certain conditions.

Definition 4.14 (Selection Function) *A* selection function *is a function*

$$Sel\colon \wp(base_{\mathcal{D}}^{\vee}(\Pi)) \to \wp(\mathcal{I})$$

which for every set of positive clauses C returns a set of interpretations $Sel(C)$ satisfying the following conditions:

$$Sel(C) \subseteq Mod(C) \tag{P1}$$

$$Sel(C) \leq^S Mod(C) \tag{P2}$$

$$\forall I \in Sel(C)\colon I \subseteq At(C) \tag{P3}$$

The first condition ensures that the selection function returns only models of C and the second condition ensures that all models of C are covered by some interpretation in $Sel(C)$. Note that an empty set of clauses has a single minimal model which is the empty interpretation, so $Sel(\emptyset) = \{\emptyset\}$ for any selection function Sel.

It is easy to show that the value $Sel(C)$ of any selection function Sel satisfying the above conditions always includes the set of minimal models of C.

Proposition 4.20 *$MM(C) \subseteq Sel(C)$, for any selection function Sel.*

Proof. Let $I \in MM(C) \subseteq Mod(C)$. By (P2), there exists $J \in Sel(C)$ such that $J \subseteq I$. By (P1), $J \in Mod(C)$ and since I is a minimal model then $J = I$. ∎

In fact, the set of minimal models $MM(C)$ is one possible selection function.

Proposition 4.21 *$MM(C)$ is a selection function.*

Proof. (P1) follows from Proposition 4.20 and (P2) follows from Corollary 4.28 of Proposition 4.27. ∎

Another possible selection function is the set $Mod(C)$ of all models of C using only atoms from C. The selection functions MM and Mod are the two extreme cases of selection functions in the sense that $MM(C) \subseteq Sel(C) \subseteq Mod(C)$ for any selection function Sel. A more interesting selection function is the *exactly one* function EO. For a set of positive clauses C it returns all interpretations obtained by selecting one atom $A \in D$ from every clause $H \in C$. Formally, a *satisfier* of a set of positive clauses C is a function $s\colon C \to At(C)$ such that $s(H) \in H$ for every clause $H \in C$. The exactly one selection function EO is defined as:

$$EO(C) = \{Img(s) \mid s \text{ is a satisfier of } C\}$$

where $Img(s) = \{s(D) \mid D \in C\}$ is the image of s. If C is empty we assume that there exists a unique function $s : \emptyset \to \emptyset$. So, $EO(\emptyset) = \{\emptyset\}$.

We now show that any selection function Sel is $\leq^S$-monotone.

Lemma 4.22 *Let C_1 and C_2 be two sets of positive clauses. If $C_1 \subseteq C_2$ then $Sel(C_1) \leq^S Sel(C_2)$.*

Proof. Let $J_2 \in Sel(C_2)$. By property (P1) $J_2 \models C_2$. Since $C_1 \subseteq C_2$ then also $J_2|_{At(C_1)} \models C_1$. By (P2) there exists $J_1 \in Sel(C_1)$ such that $J_1 \subseteq J_2|_{At(C_1)} \subseteq J_2$. ∎

The definition of a non-deterministic operator based on a selection function is straightforward.

Definition 4.15 (Non-deterministic Operator) *Let Sel be a selection function. The non-deterministic operator $N_P^{aggr}\colon \mathcal{I} \to \wp(\mathcal{I})$ based on Sel is defined as:*

$$N_P^{aggr}(I) = Sel(R_P^{aggr}(I)).$$

The results in the rest of this section hold for any choice of selection function.

For logic programs without disjunction in the head, any non-deterministic operator coincides with the van Emden-Kowalski immediate consequence operator.

Proposition 4.23 *Let P be an aggregate program without disjunction. Then $N_P^{aggr}(I) = \{T_P^{aggr}(I)\}$.*

Proof. For normal programs $R_P^{aggr}(I)$ consists only of atoms. Consequently, it has a single model using atoms from $At(R_P^{aggr})$ (which is also minimal) and is $R_P^{aggr}(I)$ itself. So, $N_P^{aggr}(I) = Sel(R_P^{aggr}(I)) = \{R_P^{aggr}(I)\} = \{T_P^{aggr}(I)\}$. ∎

Our next task is to establish a correspondence between different types of fixpoints of N_P^{aggr} and different types of models of P. The following simple result about the head reduct will be useful in the subsequent proofs.

Lemma 4.24 *If $I \models P$ then $I \models R_P^{aggr}(I)$.*

Proof. For every $C \in R_P^{aggr}(I)$ there exists a rule $r \in P$ such that $head(r) = C$ and $I \models body(r)$. From $I \models r$ follows that $I \models A$ for some $A \in C$ which implies $I \models C$. ∎

The first result is a characterization of models of P as pre-fixpoints of N_P^{aggr}.

Proposition 4.25 *$I \models P$ if and only if there exists $J \in N_P^{aggr}(I)$ such that $J \subseteq I$, i.e., $Mod(P) = \text{pre}^S(N_P^{aggr})$.*

Proof. $\Leftarrow$) For every rule $r \in P$ if $I \not\models body(r)$ then $I \models r$. If $I \models body(r)$ then $C = head(r) \in R_P^{aggr}(I)$. Suppose that there exists some $J \in N_P^{aggr}(I)$ such that $J \subseteq I$. By (P1) we know that every such J is a model of $R_P^{aggr}(I)$ and in particular $J \models C$. Consequently $I \models C$.

$\Rightarrow$) If $I \models P$ then by Lemma 4.24 $I \models R_P^{aggr}(I)$. By (P2) there exists $J \in Sel(R_P^{aggr}(I))$ such that $J \subseteq I$, so $J \in N_P^{aggr}(I)$. ∎

4.4.4 Definite Disjunctive Programs

Definite aggregate programs without disjunction have a single minimal model. Moreover, the T_P^{aggr} operator is monotone and its least fixpoint is equal to the minimal model of P. On the other hand, definite disjunctive aggregate programs can have several minimal models. However, we can still establish a correspondence between the minimal models of P and the minimal fixpoints of N_P^{aggr} which is the goal of this subsection.

We first look at monotonicity of N_P^{aggr}. Recall that the result of this operator is a set of interpretations.

Proposition 4.26 *If P is a definite disjunctive aggregate program then N_P^{aggr} is $\leq^S$-monotone, i.e. $I \leq J$ implies $N_P^{aggr}(I) \leq^S N_P^{aggr}(J)$.*

Proof. If $I \subseteq J$ then $R_P^{aggr}(I) \subseteq R_P^{aggr}(J)$ and, by Lemma 4.22,

$$Sel(R_P^{aggr}(I)) \leq^S Sel(R_P^{aggr}(J))$$

which implies

$$N_P^{aggr}(I) \leq^S N_P^{aggr}(J).$$ ∎

A non-trivial result about definite disjunctive programs is that the set of minimal models of a program P covers the set of all models of P (Seipel et al., 1997). This property depends on the fact the the heads of rules are finite disjunctions and it does not hold if we allow infinite disjunctions.

Proposition 4.27 *Let P be a definite disjunctive aggregate program. Then*

$$MM(P) \leq^S Mod(P).$$

Proof. The proof of this result in (Seipel et al., 1997) is easily lifted to aggregate programs. ■

Corollary 4.28 *If C is a set of positive clauses then $MM(C) \leq^S Mod(C)$.*

The next example demonstrates that Proposition 4.27 does not hold if we allow arbitrary first-order formulas or closed aggregate atoms which are not positive in the bodies of rules.

Example 4.11 Consider the following normal aggregate program P:

$$r\colon p(x) \leftarrow \text{CARD}(\{z \mid p(z)\}, y). \tag{4.1}$$

Suppose that the predicate p is interpreted over the set $\mathbb{N}$ of natural numbers. Obviously, an interpretation I satisfies a ground instance $\text{CARD}(\{z \mid p(z)\}, n)$ of the aggregate atom in the body of r if and only if $|I|$ is finite. We show that I is a model of P if and only if $|I|$ is infinite.

1. Let I be an interpretation of infinite size. Then the size of the set $[\![\{z \mid p(z)\}]\!]_I$ is also infinite. So, I does not satisfy the body of any rule in $ground(P)$. Hence $I \models ground(P)$.

2. Suppose that P has a finite model M with size $n = |I|$. But then for every $a \in \mathbb{N}$, M satisfies the body of the rule $p(a) \leftarrow \text{CARD}(\{z \mid p(z)\}, n) \in ground(P)$. So, for every $a \in \mathbb{N}$, $p(a)$ must be true in M. This implies that M has an infinite size which contradicts with the assumption.

Because the set of all subsets of $\mathbb{N}$ does not have minimal sets, the program P does not have minimal models. Also note that the T_P^{aggr} operator of (4.1) is non-monotone because $T_P^{aggr}(I) = \{p(n) \mid n \in \mathbb{N}\}$ if I is finite and $T_P^{aggr}(I) = \emptyset$ if I is infinite.

A program which has the same properties as (4.1) can be constructed without aggregates:

$$r_1\colon p(x) \leftarrow \exists y.\, \neg p(y) \wedge \forall z.\, y < z \rightarrow \neg p(z).$$

Again, we can show that $I \models body(r_1)$ if and only if I is finite. This implies that interpretation I of this program is a model if and only if I has an infinite size. □

The main result about non-deterministic operators is that for a definite disjunctive aggregate program the set of minimal models is equal to the set of minimal fixpoints of N_P.

Theorem 4.29 *If P is a definite disjunctive aggregate program then*

$$MM(P) = \text{mfp}(N_P^{aggr}).$$

Proof. $\Rightarrow$) By Proposition 4.25 there is a one to one correspondence between models of P and pre-fixpoints of N_P^{aggr}. So, minimal models are minimal pre-fixpoints and by Lemma 4.19 they are fixpoints.

$\Leftarrow$) Let M be a minimal fixpoint of N_P^{aggr}. By Proposition 4.25, M is a model of P. Suppose that M is not a minimal model. Then, by Proposition 4.27, there exists a minimal model M' of P which is strictly smaller than M. But then M' is a fixpoint of N_P^{aggr} (by the opposite direction of this proof) so M is not a minimal fixpoint which is a contradiction. ■

The next example shows that the above result does not hold for general disjunctive programs for any selection function.

Example 4.12 Consider the following disjunctive logic program P:

$$a \vee b.$$
$$c \leftarrow not\ d.$$

The interpretation $I = \{a, d\}$ is a minimal model of P. However, I is not a fixpoint of N_P^{aggr} for any selection function. Let $C = R_P^{aggr}(I) = \{a \vee b\}$. Using the set of all models Mod as a selection function we have $(N_P^{aggr})^{Mod}(I) = Mod(C) = \{\{a\}, \{b\}, \{a, b\}\} \not\ni I$. The result holds for any selection function Sel since $Sel(C) \subseteq Mod(C)$. □

Theorem 4.29 generalizes a result for definite aggregate programs without disjunction about the equivalence between the least fixpoint of the T_P^{aggr} operator and the least model of P — $\text{mfp}(T_P^{aggr}) = MM(P)$. One result from the theory of monotone operators which we could not generalize is the computation of the least fixpoint by ordinal powers as in Proposition 2.3.

4.4.5 Approximating Operators and Stable Models

The goal of this section is to define a set of total stable models of disjunctive programs with aggregates. We exploit ideas from approximation theory and define the notion of a non-deterministic approximating operator A_P^{aggr} of the N_P^{aggr} operator. Unlike deterministic approximating operators where both the domain and the range of the T_P operator are approximated, here we approximate only the domain of the N_P^{aggr} operator. Consequently, we define only a two-valued stable semantics and not the entire class of three and four valued stable models (which include the well-founded model).

The definition of A_P^{aggr} is very similar to that of N_P^{aggr} but to compute the head reduct we evaluate rule bodies in three-valued logic.

Definition 4.16 *The* extended head reduct *for a disjunctive aggregate program P is a function $E_P^{aggr} : \mathcal{I}^c \to \wp(base_{\mathcal{D}}^{\vee}(\Pi))$ defined as:*

$$E_P^{aggr}(I_1, I_2) = \{head(r) \mid r \in ground(P) \text{ and } (\mathcal{H}_{(I_1,I_2)}^{aggr})^1(body(r)) = \mathbf{t}\}$$

Note that we use only the under-estimate of the truth value of the body. The second step in the definition of A_P^{aggr} is exactly the same as for N_P^{aggr}.

Definition 4.17 *Let P be an aggregate program, Sel a selection function and $N_P^{aggr} : \mathcal{I} \to \wp(\mathcal{I})$ a non-deterministic operator based on Sel. The* approximating operator $A_P^{aggr} : \mathcal{I}^c \to \wp(\mathcal{I})$ *of N_P^{aggr} is defined as:*

$$A_P^{aggr}(I_1, I_2) = Sel(E_P^{aggr}(I_1, I_2)).$$

Example 4.13 We compute the value of the A_P^{aggr} operator for the program P from Example 4.12. Let $I = \{a, c\}$. We have

$$A_P^{aggr}(\emptyset, I) = Sel(E_P^{aggr}(\emptyset, I)) = Sel(\{a \vee b, c\}).$$

As a selection function take the set of all models Mod. We have $Mod(\{a \vee b, c\}) = \{\{a, c\}, \{b, c\}, \{a, b, c\}\}$ which is also the result of $(A_P^{aggr}(\emptyset, I))^{Mod}$. □

The operator A_P^{aggr} has similar properties and relationship with N_P^{aggr} as an approximating operator according to Approximation Theory.

Proposition 4.30

- A_P^{aggr} extends N_P^{aggr}, *i.e.* $A_P^{aggr}(I, I) = N_P^{aggr}(I)$.
- A_P^{aggr} *is monotone in the following sense:*

$$(I_1, I_2) \leq_p (J_1, J_2) \text{ implies } A_P^{aggr}(I_1, I_2) \leq^S A_P^{aggr}(J_1, J_2).$$

The proposition is a consequence of the analogous properties of the extended head reduct and Lemma 4.22.

Lemma 4.31

- $E_P^{aggr}(I, I) = R_P^{aggr}(I)$;
- $(I_1, I_2) \leq_p (J_1, J_2)$ *implies* $E_P^{aggr}(I_1, I_2) \subseteq E_P^{aggr}(J_1, J_2)$.

Stable models of disjunctive aggregate programs are defined as follows.

Definition 4.18 *An interpretation I is a* stable model *of P if $I \in \text{mfp}(A_P^{aggr}(\cdot, I))$.*

We note that the operator $A_P^{aggr}(\cdot, I)$ is defined only on the domain $[\emptyset, I]$. So, if I is a stable model of P then it is the single fixpoint of $A_P^{aggr}(\cdot, I)$.

The notion of a stable model is well defined as it does not depend on the selection function used to define the operator A_P^{aggr}. That property follows directly from our next result.

Proposition 4.32 *Let P be a disjunctive program and let Sel be a selection function. Then* $\mathrm{mfp}((A_P^{aggr})^{Sel}(\cdot, I)) = \mathrm{mfp}((A_P^{aggr})^{MM}(\cdot, I))$.

Example 4.14 Reconsider the program P from Example 4.12:

$$a \vee b.$$
$$c \leftarrow not\ d.$$

It has four minimal models $MM(P) = \{\{a, c\}, \{b, c\}, \{a, d\}, \{b, d\}\}$. However, only two of them, $\{a, c\}$ and $\{b, c\}$, are stable models of P. To see why take, for example, $I = \{a, c\}$. To verify that I is a stable model of P we have to show that no subset $I' \subseteq I$ is a fixpoint of the operator $A_P^{aggr}(\cdot, I)$ that is $I' \notin A_P^{aggr}(I', I)$. We demonstrate this for $I' = \emptyset$. In Example 4.13 we already computed the value $\mathcal{M}$ of $(A_P^{aggr})^{Mod}(\emptyset, I)$ which is $\mathcal{M} = \{\{a, c\}, \{b, c\}, \{a, b, c\}\}$ and we have $I' \notin \mathcal{M}$. So, $I' = \emptyset$ is not a fixpoint of $A_P^{aggr}(\cdot, I)$. □

Theorem 4.33 *Stable models of disjunctive aggregate programs are minimal models.*

Before proving this theorem we need some intermediate results.

Lemma 4.34 *M is a model of $E_P^{aggr}(M, I)$ if and only if M is a pre-fixpoint of $A_P^{aggr}(\cdot, I)$.*

Proof. Similar to the proof of Proposition 4.25.

The next result is a property of the pre-fixpoints of the $A_P^{aggr}(\cdot, I)$ operator which is similar to Proposition 4.27.

Proposition 4.35 *If M is a pre-fixpoint of $A_P^{aggr}(\cdot, I)$ then there exists a minimal pre-fixpoint N of $A_P^{aggr}(\cdot, I)$ such that $N \subseteq M$.*

Proof. The proof is adapted from the proof of Proposition 4.27 from (Seipel et al., 1997).

Let $\mathrm{pre}_M = \{J \in \mathrm{pre}(A_P^{aggr}(\cdot, I)) \mid J \subseteq M\}$. We prove below that every chain $\mathcal{K} \subseteq \mathrm{pre}_M$ has a lower bound M^* in pre_M. Using Zorn's lemma this implies that pre_M has a minimal element N which satisfies the condition of the proposition. Let $\mathcal{K} \subseteq \mathrm{pre}_M$ be a chain of pre-fixpoints and let

$$M^* = \bigwedge \mathcal{K}.$$

Obviously, M^* is a lower bound of $\mathcal{K}$. We only need to show that M^* is a pre-fixpoint. Let $C^* = E_P^{aggr}(M^*, I)$ and fix a clause

$$A_1 \vee \ldots \vee A_k \in C^*.$$

Since $\mathcal{K}$ is a chain, we get that

$$\mathcal{J} = \{J \cap \{A_1, \ldots, A_k\} \mid J \in \mathcal{K}\}$$

is a chain too. Because all elements in $\mathcal{J}$ have a finite cardinality then there exists $J' \in \mathcal{J}$ with minimal cardinality and $J' \subseteq J$ for every $J \in \mathcal{J}$. Let $M' \in \mathcal{K}$ be the interpretation such that $J' = M' \cap \{A_1, \ldots, A_k\}$. From $M^* \subseteq M'$ then

$$C^* = E_P^{aggr}(M^*, I) \subseteq E_P^{aggr}(M', I) = C'$$

and consequently

$$A_1 \vee \ldots \vee A_k \in C'.$$

Since M' is a pre-fixpoint of $A_P^{aggr}(\cdot, I)$ then (by Lemma 4.34) M' is a model of C' and so $M' \models A_1 \vee \ldots \vee A_k$, i.e. $J' = M' \cap \{A_1, \ldots, A_k\} \neq \emptyset$. Moreover,

$$\forall J \in \mathcal{K}\colon J' \subseteq J$$

which implies that $J' \subseteq M^*$. So M^* is also a model of C^* and, by Lemma 4.34, a pre-fixpoint of $A_P^{aggr}(\cdot, I)$. ∎

Proof. (Theorem 4.33) Let $I \in \mathrm{mfp}(A_P^{aggr}(\cdot, I))$ be a stable model of P. In particular I is a fixpoint of $A_P^{aggr}(\cdot, I)$, i.e. $I \in A_P^{aggr}(I, I)$. Because A_P^{aggr} extends N_P^{aggr} (Proposition 4.30), $I \in N_P^{aggr}(I)$ and by Proposition 4.25 I is a model (pre-fixpoint) of P.

To prove minimality suppose that there exists an interpretation J such that

$$J \text{ is a model of } P \qquad (*)$$

and $J \subset I$. By Proposition 4.25, (*) implies that there exists K such that $K \in N_P^{aggr}(J)$ and $K \subseteq J$. Because $J \subset I$ then (J, I) is a three-valued interpretation and, moreover, $(J, I) \leq_p (J, J)$. By the monotonicity of A_P^{aggr} (Proposition 4.30) this implies

$$A_P^{aggr}(J, I) \leq^S A_P^{aggr}(J, J) = N_P^{aggr}(J) \ni K.$$

By the definition of $\leq^S$ there exists $M \in A_P^{aggr}(J, I)$ such that $M \subseteq K$. Since $K \subseteq J$ then $M \subseteq J$, so J is a pre-fixpoint of $A_P^{aggr}(\cdot, I)$. Next, by Proposition 4.35, there exists a minimal pre-fixpoint J' of $A_P^{aggr}(\cdot, I)$ such that $J' \subseteq J \subset I$. Finally, by Lemma 4.19, J' is also a fixpoint of $A_P^{aggr}(\cdot, I)$ but this contradicts with the fact that I is a minimal fixpoint. ∎

4.4.6 Comparison with Other Semantics

We show that the stable semantics of disjunctive aggregate programs extends both the stable semantics of disjunctive logic programs (Gelfond and Lifschitz, 1991; Przymusinski, 1991) and exact stable models of aggregate programs without disjunction as defined earlier.

First, we recall the definition of the stable semantics of disjunctive logic programs without aggregates. The *reduct* P^I of a disjunctive logic program P without aggregates with respect to an interpretation I is a program P^I obtained from $ground(P)$ as follows:

1. delete all rules for which a negative literal in the body is not satisfied by I;
2. delete all negative literals from the remaining rules.

Definition 4.19 *An interpretation I is a* stable model *of P if $I \in MM(P^I)$. The set of all stable models of P is denoted with $SM(P)$.*

For a disjunctive logic programs without aggregates we drop the ^{aggr} index from the names of head reducts and operators. Taking into account the fact that the bodies of all rules are conjunctions of literals, the definition of the extended head reduct can be simplified as:

$$E_P(I_1, I_2) = \{head(r) \mid r \in ground(P),\ I_1 \models pos(r),\ \text{and } I_2 \not\models neg(r)\}$$

For programs without aggregates, the extended head reduct E_P and the approximating operator A_P are well-defined for four-valued interpretations and not only for three-valued interpretations.

We first show how the A_P operator is related with the N_P operator on a reduct of P.

Lemma 4.36 $A_P(I_1, I_2) = N_{P^{I_2}}(I_1)$.

We are now ready to show the correspondence.

Proposition 4.37 *$I \in SM(P)$ if and only if $I \in \text{mfp}(A_P^{aggr}(\cdot, I))$.*

Proof. $I \in SM(P)$ if and only if $I \in MM(P^I)$ by definition of $SM(P)$, if and only if $I \in \text{mfp}(N_{P^I})$ by Theorem 4.29, if and only if $I \in \text{mfp}(A_P(\cdot, I))$ by Lemma 4.36.

Remember that the A_P operator is defined on four-valued interpretations while A_P^{aggr} is defined on three-valued interpretations. So, we need to show that $I \in \text{mfp}(A_P(\cdot, I))$ if and only if $I \in \text{mfp}(A_P^{aggr}(\cdot, I))$. An interpretation I is a least fixpoint of $A_P(\cdot, I)$ if (i) $I \in A_P(I, I)$ and (ii) there does not exist an interpretation $J \subset I$ such that $J \in A_P(J, I)$. Clearly both conditions depend only on the value of A_P on consistent interpretations (J, I). ∎

We now show that the stable semantics of non-disjunctive logic programs based on non-deterministic approximating operators agrees with the semantics based on the Φ_P^{aggr} operator.

Proposition 4.38 *If P is an aggregate program without disjunction then*

$$A_P^{aggr}(I_1, I_2) = \{(\Phi_P^{aggr})^1(I_1, I_2)\}.$$

Corollary 4.39 *If P is an aggregate program without disjunction then I is a stable model of P (according to Definition 4.18) if and only if P is an exact stable model of P.*

Some instances of the two-valued non-deterministic operator N_P could be used to characterize some other semantics of disjunctive logic programming. For example, the possible model semantics (Sakama, 1989). We have not investigated further this line of research because it is not directly relevant to our goals.

Most of the operators associated with disjunctive logic programs which have been studied so far are deterministic. They operate either on (closed) sets of positive clauses, called *states*, or sets of interpretations. In either case such operators are not relevant for us, as we are interested in characterizing stable models which are single interpretations. Most of the operators which are based on sets of interpretations use the Smyth pre-order relation. This should not come as a surprise considering that this order was originally introduced to model non-deterministic computations (Smyth, 1978).

Non-deterministic operators were studied in the context of programs with monotone cardinality atoms (mc-atoms) (Marek et al., 2004) and set constraint atoms (sc-atoms) (Marek and Remmel, 2004). Sc-atoms are similar to aggregate atoms and mc-atoms to aggregate atoms with the $\text{CARD}_{\geq}$ aggregate relation. The non-determinism comes from the fact that sc-atoms and mc-atoms also appear in the head of rules. The main difference with our approach is that we focus on minimality of stable models while (Marek et al., 2004; Marek and Remmel, 2004) do not. However, there are some similarities. For example (Marek et al., 2004) has the same characterization of models as pre-fixpoints as in Proposition 4.25. It is possible to apply the framework and results in this section to sc-programs and mca-programs or to propositional aggregate programs with aggregate atoms in the head and obtain a stable model semantics in which stable models are always minimal. The important question is whether such semantics have any useful applications and advantages over the semantics of (Marek et al., 2004; Marek and Remmel, 2004).

4.5 Conclusions

According to Approximation Theory any consistent approximating operator of T_P^{aggr} gives rise to a three-valued stable semantics. In this chapter we proposed

and investigated two such operators — the ultimate approximating operator and a family of operators extending the Φ_P operator of normal logic programs. From this family we proposed to use the operator interpreting aggregate symbols again with their ultimate approximations. However, during our investigations we considered several other approximating operators, all of them extending the Φ_P operator. In this light finding a criteria for selecting an approximating operator becomes an important issue.

The ultimate approximations of operators have several attractive properties which make them a preferred choice: they induce the most precise semantics; they have a constructive definition; they generalize the theory of monotone induction. One concern with the ultimate approximations is that generally they are more expensive to compute. The second concern is that their definition is not compositional. For example for the three-valued truth function $\mathcal{H}_{\tilde{I}}$ of first-order logic the truth value of a formula is a function of the truth values of its subformulas. This is not the case if we consider the ultimate valuation function $U_{\mathcal{H}}$ of the two-valued truth function $\mathcal{H}$.

The guiding principle in defining the three-valued stable semantics of aggregate programs and in particular defining the three-valued truth function for aggregate atoms has been this general principle of *compositionallity*. First of all, we have separated the denotation of sets from the denotation of aggregates. Then we defined the interpretation of aggregates (both in two and in three-valued logic) as a function of the interpretation of the set and lambda expressions. Further, the membership of each element in the denotation of a set expression is decided independently of the other elements. We want to point out that it is possible to define a truth function of aggregate atoms or an evaluation function of set expressions which are not compositional. The resulting approximating operator will be more precise than the Φ_P^{aggr} operator based on ultimate approximating aggregates and will still extend the Φ_P^{aggr} operator.

The result that the three-valued stable semantics based on the ultimate approximating aggregate extends the least fixpoint semantics of definite aggregate programs and the standard model of weakly stratified aggregate programs is specific only to aggregate atoms with monotone or anti-monotone aggregate relations. So, we want to point out that any semantics in which the approximating aggregate relations of monotone and anti-monotone aggregate relations coincide with the ultimate approximating aggregate will have the same property no matter how the non-monotone aggregate relations are extended to approximating aggregate relations.

The results on the stable model semantics of disjunctive logic programs based on non-deterministic operators was obtained using properties of disjunctive logic programs. A central role played the fact that programs have only finite disjunctions in the head. We think that it is interesting to provide a purely algebraic characterization of the semantics in a similar way as Approximation Theory. We believe that our results are an important step in this direction. We have identi-

fied several concepts and properties which may prove useful. For example, Smyth monotonicity and pre-fixpoints of non-deterministic operators. Another property which is used in several results, including the proof of minimality of stable models, is that $N_P(I)$ is covered by the set of minimal interpretations in $N_P(I)$. It is interesting to give a simple algebraic characterization of the structure of the set $N_P(I)$ which implies this property. One approach to this problem which uses results and intuitions from Domain Theory (Abramsky and Jung, 1994) is by Rounds and Zhang (Rounds and Zhang, 2001). They show that the function Mod which returns the set of all models for a set of positive clauses is an isomorphism between logically closed sets of disjunctions and Scott-compact saturated sets of interpretations.

The obvious next step is to extend our fixpoint characterization to the entire set of three-valued stable models (Przymusinski, 1991) and not just the total. This could be done by considering non-deterministic approximating operators of the form $A : L^c \to \wp(L^c)$.

Chapter 5

The Three-valued Stable Semantics Based on the Ultimate Approximating Aggregate

In Section 4.2 we defined a family of three-valued stable semantics of aggregate programs all of them extending the three-valued stable semantics of normal logic programs (Przymusinksi, 1990). The semantics depend on the interpretation of aggregate symbols with three-valued aggregate relations. We also considered the semantics obtained by interpreting aggregate symbols with the ultimate approximating aggregate relations. In this chapter we investigate in more depth this semantics. This semantics is interesting because it is the most precise semantics (according to Approximation Theory) from the family of three-valued stable semantics which we defined in Section 4.2.

In the first section we study basic algebraic properties of the ultimate approximating aggregate relation. We also present efficient algorithms for computing the approximating aggregate of most of the standard aggregate functions and relations. These algorithms are used for complexity analysis and they can also be used in a practical implementation of the semantics.

In Section 5.3 we study transformations of aggregate atoms to formulas which are equivalent in three-valued logic. For some aggregate relations, for example MIN and MAX, we can translate aggregate atoms to formulas without aggregate atoms. In other cases, for example $\text{SUM}_{\geq}$, we can translate an aggregate atom with a non-monotone aggregate relation to a formula which uses only monotone and anti-monotone aggregate relations. The transformations serve several purposes:

to study properties of aggregate programs; to transform an aggregate program to an equivalent program which is simpler and can be executed by a system with a limited support for aggregates; to discussing related work. At the end of the section we present a general translation which is defined for all aggregate atoms and does not exploit any properties of specific aggregate relations.

In Section 5.4 we show a prototype implementation of the well-founded semantics in XSB Prolog for aggregate programs using only monotone aggregate relations (Section 5.4). Together with the example in Section 5.5 this can be considered as the culmination of our research. Although we do not give formal proofs we provide strong evidence that the implementation is correct for the well-founded semantics of aggregate programs discussed in this section. So, we are able to travel the whole path from a model theoretic semantics based on solid algebraic foundations to a simple and intuitive implementation in Prolog.

Almost none of the material in this chapter has been published. Only the general translation of aggregate atoms in Section 5.3.6 has been published in (Pelov et al., 2003) for a propositional language.

5.1 Ultimate Approximating Aggregates

We now give a concrete definition of the approximating aggregate relation A_{R} for every aggregate relation R. We interpret R with its ultimate approximating operator U_{R}.

Definition 5.1 (Ultimate Approximating Aggregate) *Let* $\mathrm{R} \subseteq \mathcal{M}(D) \times D$ *be an aggregate relation. The* ultimate approximating aggregate *of* R *is a function* $U_{\mathrm{R}} \colon \mathcal{M}(D)^c \times D \to \mathcal{THREE}$. *It is defined as*

$$\mathrm{R}((M_1, M_2), d) = (U^1_R((M_1, M_2), d), U^2_{\mathrm{R}}((M_1, M_2), d))$$

where $U^1_{\mathrm{R}}, U^2_{\mathrm{R}} \colon \mathcal{M}(D)^c \times D \to \mathcal{TWO}$ *are the projections of* U_{R} *on the first and second component, defined as follows:*

$$U^1_{\mathrm{R}}((M_1, M_2), d) = \mathbf{t} \text{ if and only if } \forall M \in [M_1, M_2] \colon (M, d) \in \mathrm{R}$$
$$U^2_{\mathrm{R}}((M_1, M_2), d) = \mathbf{t} \text{ if and only if } \exists M \in [M_1, M_2] \colon (M, d) \in \mathrm{R}$$

5.1.1 Algebraic Properties

For monotone and anti-monotone aggregates the truth value can be computed directly on the boundary multisets.

Proposition 5.1 *Let* $\mathrm{R} \colon \mathcal{M}(D_1) \times D_2$ *be a monotone aggregate relation. Then* $((M_1, M_2), d) \in U^1_{\mathrm{R}}$ *if and only if* $(M_1, d) \in \mathrm{R}$ *and* $((M_1, M_2), d) \in U^2_{\mathrm{R}}$ *if and only if* $(M_2, d) \in \mathrm{R}$.

Proposition 5.2 *Let* $\mathrm{R}\colon \mathcal{M}(D_1) \times D_2$ *be an anti-monotone aggregate relation. Then* $((M_1, M_2), d) \in U^1_{\mathrm{R}}$ *if and only if* $(M_2, d) \in \mathrm{R}$ *and* $((M_1, M_2), d) \in U^2_{\mathrm{R}}$ *if and only if* $(M_1, d) \in \mathrm{R}$.

It is interesting to look at the definition of U_{F} for aggregate functions. We study under what conditions an aggregate function F can be replaced by a conjunction of $\mathrm{F}_{\leq}$ and $\mathrm{F}_{\geq}$. Because $\wedge$ is interpreted in the logic $\mathcal{THREE}$ as meet under the $\leq_t$ order we can study the equivalent question whether $U_{\mathrm{F}} = U_{\mathrm{F}_{\leq}} \cap_t U_{\mathrm{F}_{\geq}}$. First note that $U_{\mathrm{F}_{\leq}} \cap_t U_{\mathrm{F}_{\geq}}$ is an approximating aggregate of F. Because U_{F} is the most precise approximation of F we immediately obtain the inequality $U_{\mathrm{F}} \geq_p U_{\mathrm{F}_{\leq}} \cap_t U_{\mathrm{F}_{\geq}}$ and we only need to study the other direction. We look separately at the two components.

Proposition 5.3 *Let* $\mathrm{F}\colon \mathcal{M}(D_1) \to D_2$ *be an aggregate function and* $\leq$ *a partial order on* D_2. *Then* $U^1_{\mathrm{F}} = U^1_{\mathrm{F}_{\geq}} \cap U^1_{\mathrm{F}_{\leq}}$.

Proof.

$$
\begin{aligned}
&((M_1, M_2), d) \in U^1_{\mathrm{F}_{\geq}} \cap U^1_{\mathrm{F}_{\leq}} \\
\Leftrightarrow &((M_1, M_2), d) \in U^1_{\mathrm{F}_{\geq}} \wedge ((M_1, M_2), d) \in U^1_{\mathrm{F}_{\leq}} \\
\Leftrightarrow &(\forall M \in [M_1, M_2]\colon \mathrm{F}(M) \geq d) \wedge (\forall M \in [M_1, M_2] : \mathrm{F}(M) \leq d) \\
\Leftrightarrow &\forall M \in [M_1, M_2]\colon d \leq \mathrm{F}(M) \leq d \\
\Leftrightarrow &\forall M \in [M_1, M_2]\colon \mathrm{F}(M) = d \\
\Leftrightarrow &((M_1, M_2), d) \in U^1_{\mathrm{F}}
\end{aligned}
$$

■

Unfortunately, we do not have such equivalence for the second component.

Example 5.1 For the three-valued multiset $(\emptyset, \{1, 3\})$ we have that

$$U_{\mathrm{SUM}}((\emptyset, \{1, 3\}), 2) = (\mathbf{f}, \mathbf{f})$$

because $\mathrm{SUM}(M) \neq 2$ for all multisets $M \in [\emptyset, \{1, 3\}]$. On the other hand

$$
\begin{aligned}
U_{\mathrm{SUM}_{\geq}}((\emptyset, \{1, 3\}), 2) &= (\mathbf{f}, \mathbf{t}) \text{ and} \\
U_{\mathrm{SUM}_{\leq}}((\emptyset, \{1, 3\}), 2) &= (\mathbf{f}, \mathbf{t})
\end{aligned}
$$

and so their conjunction is also equal to $(\mathbf{f}, \mathbf{t})$. □

A sufficient condition to obtain equivalence for the second component is the aggregate function to be continuous. Let L be a lattice and $S \subseteq L$. We say that S is a *convex* if for all $x, y \in S$, $[x, y] \subseteq S$.

Definition 5.2 *An aggregate function* F *is* continuous *if for any convex of multisets* $\mathcal{A}$, *the image* $\mathrm{F}(\mathcal{A})$ *of* $\mathcal{A}$ *under* F *is a convex.*

Proposition 5.4 *If* F *is a continuous aggregate function then* $U^2_{\mathrm{F}} = U^2_{\mathrm{F}_\geq} \cap U^2_{\mathrm{F}_\leq}$.

Proof. The inclusion $\subseteq$ follows from the fact that U_{F} is the most precise approximation of F. So, we only need to prove the other direction, i.e. $U^2_{\mathrm{F}} \supseteq U^2_{\mathrm{F}_\geq} \cap U^2_{\mathrm{F}_\leq}$. Suppose that $((M_1, M_2), d) \in U^2_{\mathrm{F}_\geq} \cap U^2_{\mathrm{F}_\leq}$. Then there exists $M', M'' \in [M_1, M_2]$ such that $\mathrm{F}(M') \geq d$ and $\mathrm{F}(M'') \leq d$. Because F is continuous then there exists $M \in [M_1, M_2]$ such that $\mathrm{F}(M) = d$. Hence $((M_1, M_2), d) \in U^2_{\mathrm{F}}$. ∎

From the common aggregate functions only CARD is continuous.

Proposition 5.5 *The aggregate function* $\mathrm{CARD}\colon \mathcal{FM}(D) \to \mathbb{N}$ *is continuous for any domain* D.

Proof. Let $\mathcal{A} \subseteq \mathcal{FM}(D)$ be a convex set of multisets over D. We have to show that the set $C = \mathrm{CARD}(\mathcal{A})$ is a convex set. Take two numbers $c_1, c_2 \in C$ such that $c_1 \leq c_2$. Let $M_1, M_2 \in \mathcal{A}$ be multisets such that $\mathrm{CARD}(M_1) = c_1$ and $\mathrm{CARD}(M_2) = c_2$. Because $\mathcal{A}$ is a convex then $[M_1, M_2] \subseteq \mathcal{A}$. Clearly, for any $c \in [c_1, c_2]$ there exists a multiset $M \in [M_1, M_2]$ such that $\mathrm{CARD}(M) = c$. Because $M \in \mathcal{A}$ then $C = \mathrm{CARD}(M) \in X$. This implies that $[c_1, c_2] \subseteq C$. Thus C is a convex and CARD is a continuous function. ∎

The aggregate functions SUM and PROD are well defined on different domains, for example $\mathbb{N}$, $\mathbb{Z}$, and $\mathbb{R}$. To avoid showing continuity (or discontinuity) for many different domains we observe the following property. If an aggregate function is continuous on a larger domain, for example $\mathbb{R}$ it will also be continuous on a smaller domain, for example $\mathbb{N}$.

Proposition 5.6 *Let* $\mathrm{F} : \mathcal{FM}(D) \to D$ *be an aggregate function and* P *be a subset of* D *such that the aggregate function*

$$\mathrm{F}|_P : \mathcal{FM}(P) \to P\colon M \mapsto \mathrm{F}(M)$$

is a well-defined function, i.e., $\mathrm{F}(M) \in P$ *for any multiset* $M \in \mathcal{FM}(P)$. *If* F *is a continuous aggregate function then* $\mathrm{F}|_P$ *is also continuous.*

Proof. The key issue in the proof is that convexity is defined with respect to a set. So, if a set X is a convex in D, i.e.

$$\forall a, b \in X\colon a \leq b \to \forall z \in D\colon a \leq z \leq b \to z \in X$$

then X is also a convex for any subset of D and in particular in the set $P \subseteq D$:

$$\forall a, b \in X\colon a \leq b \to \forall z \in P\colon a \leq z \leq b \to z \in X.$$

So for a convex set $\mathcal{A} \subseteq \mathcal{FM}(P)$ of finite multisets over P if the set $\mathrm{F}(\mathcal{A})$ is a convex in D then it is also a convex in P. ∎

This proposition implies that if an aggregate function is not continuous for a smaller domain then it is not continuous for a larger domain. In the next proposition we show that SUM and PROD are not continuous for the domain $\mathbb{N}$ which implies that they are not continuous for any larger domain.

Proposition 5.7 *The aggregate functions* $\text{SUM}\colon \mathcal{FM}(\mathbb{N}) \to \mathbb{N}$, $\text{PROD}\colon \mathcal{FM}(\mathbb{N}) \to \mathbb{N}$, *and* $\text{AVG}\colon \mathcal{FM}(\mathbb{R}) \to \mathbb{R}$ *are not continuous.*

Proof. (SUM) Consider the convex set $\mathcal{A} = \{\{1\}, \{1,2\}\}$. The image of $\mathcal{A}$ under SUM is the set

$$\text{SUM}(\mathcal{A}) = \{\text{SUM}(\{1\}), \text{SUM}(\{1,2\})\} = \{1,3\}$$

which is not a convex because $2 \in [1,3]$ but $2 \notin \{1,3\}$.

(PROD) Consider the convex set $\mathcal{B} = \{\{1\}, \{1,3\}\}$. The image of $\mathcal{B}$ under PROD is the set $\text{PROD}(\mathcal{B}) = \{1,3\}$ which is again not a convex.

(AVG) The image of $\mathcal{B}$ under AVG is the set $\text{AVG}(\mathcal{B}) = \{1, 2.5\}$ which is not a convex. ∎

Proposition 5.8 *If* L *is a lattice containing a chain with three elements then the aggregate functions* $\text{INF}\colon \mathcal{FM}(L) \to L$ *and* $\text{SUP}\colon \mathcal{FM}(L) \to L$ *are not continuous.*

Proof. Consider the chain $\{a, b, c\} \subseteq L$ ordered as $a < b < c$. The image of the convex set $\mathcal{A} = \{\{c\}, \{a, c\}\}$ under INF is the set $\text{INF}(\mathcal{A}) = \{a, c\}$ which is not a convex because $b \in [a, c]$ but $b \notin \{a, c\}$. For the aggregate function SUP consider the image of the convex set $\mathcal{B} = \{\{a\}, \{a, c\}\}$ under SUP which is again the set $\{a, c\}$. ∎

Note that if L is a totally ordered set then $\text{INF}(M) = \text{MIN}(M)$ and $\text{SUP}(M) = \text{MAX}(M)$ for any finite multiset M on L. So, we have the following corollary.

Corollary 5.9 *The aggregate functions* $\text{MIN}, \text{MAX} : \mathcal{FM}(D) \to D$ *are not continuous for any set of numbers* $D \subseteq \mathbb{R}$.

5.1.2 Algorithms

In this section we present algorithms for computing the ultimate approximating aggregate of some common aggregate functions. Such algorithms are useful for several purposes. First of all, they can be used in a practical implementation of the three-valued stable semantics. Second, they are important for the complexity analysis of the semantics. The formulas for SUM are also used in discussing the relationship with the language of weight constraint rules.

For monotone and anti-monotone aggregate relations, for example LB and UB (Proposition 3.3), the algorithms are given by Proposition 5.1 and Proposition 5.2. Our first result concerns extrema aggregates defined on possibly infinite multisets.

Proposition 5.10 *The algorithms on Table 5.1 are correct.*

R	$((M_1, M_2), d) \in U^1_{\mathrm{R}}$ iff	$((M_1, M_2), d) \in U^2_{\mathrm{R}}$ iff
MIN	$d \in M_1 \wedge \textsc{min}(M_2, d)$	$d \in M_2 \wedge \textsc{min}(M_1 \uplus \{d\}, d)$
MAX	$d \in M_1 \wedge \textsc{max}(M_2, d)$	$d \in M_2 \wedge \textsc{max}(M_1 \uplus \{d\}, d)$
INF	$\textsc{inf}(M_1, d) \wedge \textsc{inf}(M_2, d)$	$\textsc{inf}(M_1 \uplus M_2^{[d,\top]}, d)$
SUP	$\textsc{sup}(M_1, d) \wedge \textsc{sup}(M_2, d)$	$\textsc{sup}(M_1 \uplus M_2^{[\bot,d]}, d)$

Table 5.1: Algorithms for Extrema Aggregates

Next, look at the aggregate functions CARD, SUM, and PROD defined on finite multisets.

Proposition 5.11 (U_{card})

$$((M_1, M_2), d) \in U^1_{\mathrm{CARD}} \textit{ if and only if } |M_1| = d = |M_2|$$
$$((M_1, M_2), d) \in U^2_{\mathrm{CARD}} \textit{ if and only if } |M_1| \leq d \leq |M_2|$$

For SUM and PROD we can only give efficient algorithms for the first component of their ultimate approximations. We show in Section 5.1.3 that deciding the second component of the ultimate approximating aggregates are NP-complete problems. So, it is unlikely to find efficient algorithms.

Proposition 5.12 (U^1_{sum} and U^1_{prod})

$$((M_1, M_2), d) \in U^1_{\mathrm{SUM}} \textit{ iff } \textstyle\sum M_1 = d \textit{ and } \forall x \in (M_2 - M_1) \colon x = 0$$
$$((M_1, M_2), d) \in U^1_{\mathrm{PROD}} \textit{ iff } \textstyle\prod M_1 = d \textit{ and } \forall x \in (M_2 - M_1) \colon x = 1$$

We now look at combined aggregate relations of the form $\mathrm{F}_{\geq}$ and $\mathrm{F}_{\leq}$ where $\mathrm{F} \colon \mathcal{FM}(D_1) \to D_2$ is an aggregate function on finite multisets and $\leq$ is a total order on D_2. For all three aggregate functions CARD, SUM, and PROD we can give efficient algorithms for $U^1_{\mathrm{F}_{\geq}}$ and $U^2_{\mathrm{F}_{\geq}}$. We start with the following general result. Let $\min_{\mathrm{F}}, \max_{\mathrm{F}} \colon \mathcal{FM}(D_1)^c \to \mathcal{FM}(D_1)$ be functions which take as an input a finite three-valued multiset (M_1, M_2) and return a multiset $M \in [M_1, M_2]$ such that $\mathrm{F}(M)$ is minimal (resp. maximal) over the set $[M_1, M_2]$, i.e.

$$\forall M \in [M_1, M_2] \colon \mathrm{F}(\min_{\mathrm{F}}(M_1, M_2)) \leq \mathrm{F}(M)$$
$$\forall M \in [M_1, M_2] \colon \mathrm{F}(\max_{\mathrm{F}}(M_1, M_2)) \geq \mathrm{F}(M)$$

Note that for a given aggregate function F and a three-valued multiset (M_1, M_2) there may be more than one multiset in the interval $[M_1, M_2]$ with a minimal value of F. For example $\min_{\mathrm{SUM}}(\emptyset, \{0\})$ can return either $\emptyset$ or $\{0\}$.

The values of $U_{\mathrm{F}_{\leq}}$ and $U_{\mathrm{F}_{\geq}}$ can be computed using the $\min_{\mathrm{F}}$ and $\max_{\mathrm{F}}$ functions in the following way.

Lemma 5.13 *Let* $\mathrm{F}\colon \mathcal{FM}(D_1) \to D_2$ *be an aggregate function and* $\leq$ *be a total order on* D_2*. Then:*

$$((M_1, M_2), d) \in U^1_{\mathrm{F}_\geq} \text{ if and only if } \mathrm{F}(\min_{\mathrm{F}}(M_1, M_2)) \geq d$$
$$((M_1, M_2), d) \in U^2_{\mathrm{F}_\geq} \text{ if and only if } \mathrm{F}(\max_{\mathrm{F}}(M_1, M_2)) \geq d$$

and similarly for $U_{\mathrm{F}_\leq}$*:*

$$((M_1, M_2), d) \in U^1_{\mathrm{F}_\leq} \text{ if and only if } \mathrm{F}(\max_{\mathrm{F}}(M_1, M_2)) \leq d$$
$$((M_1, M_2), d) \in U^2_{\mathrm{F}_\leq} \text{ if and only if } \mathrm{F}(\min_{\mathrm{F}}(M_1, M_2)) \leq d.$$

Proof. First note that since $\{\mathrm{F}(M) \mid M \in [M_1, M_2]\}$ is a finite totally ordered set it always has a minimal and a maximal element. We give the proof for $\mathrm{F}^1_\geq$:

$$\begin{aligned}
&((M_1, M_2), d) \in U^1_{\mathrm{F}_\geq}\\
\Leftrightarrow &\forall M \in [M_1, M_2]\colon (M, d) \in \mathrm{F}_\geq\\
\Leftrightarrow &\forall M \in [M_1, M_2]\colon \mathrm{F}(M) \geq d\\
\Leftrightarrow &\forall x \in \{\mathrm{F}(M) \mid M \in [M_1, M_2]\}\colon x \geq d\\
\Leftrightarrow &\mathrm{F}(\min_{\mathrm{F}}(M_1, M_2)) \geq d.
\end{aligned}$$

The proofs of the other cases are analogous. ∎

So, to decide the first and second components of $U_{\mathrm{F}_\geq}((M_1, M_2), d)$ we need to compute the values $\min_{\mathrm{F}}(M_1, M_2)$ and $\max_{\mathrm{F}}(M_1, M_2)$. For CARD it is clear that the minimal value is obtained for the multiset M_1 and the maximal value for the multiset M_2.

Proposition 5.14 $\min_{\mathrm{CARD}}(M_1, M_2) = M_1$ *and* $\max_{\mathrm{CARD}}(M_1, M_2) = M_2$.

Proposition 5.15

$$\min_{\mathrm{SUM}}(M_1, M_2) = M_1^+ \uplus M_2^-$$
$$\max_{\mathrm{SUM}}(M_1, M_2) = M_1^- \uplus M_2^+.$$

Proof. The multiset M with a minimal sum is obtained by taking all elements from M_1 and all negative numbers from $M_2 - M_1$, that is

$$\min_{\mathrm{SUM}}(M_1, M_2) = M_1 \uplus (M_2 - M_1)^- = M_1^+ \uplus M_1^- \uplus (M_2^- - M_1^-) = M_1^+ \uplus M_2^-. \quad ∎$$

We want to point out that there is a close relation between the formulas of SUM and the monotonicity properties of SUM. Recall that the aggregate function SUM is monotone on multisets of positive numbers and anti-monotone on multisets of negative numbers. If you look at the formulas from Proposition 5.15 they partition the multisets M_1 and M_2 to multisets with positive and negative numbers. The minimal and maximal elements are obtained by adding the sum of one partition from the under-estimate and another partition from the over-estimate.

Finally, we look at the derived aggregate relations $\text{PROD}_{\leq}$ and $\text{PROD}_{\leq}$. First, we give algorithms for the aggregate relations restricted to positive real numbers and then to integer numbers. Then we combine the results to obtain algorithms for PROD on the entire set of real numbers.

Observe that PROD is monotone for multisets on the interval $[1, \infty)$ and anti-monotone on the interval $[0, 1]$ and non-monotone on negative numbers. So, it has a similar algorithm as the SUM aggregate. For the aggregate function PROD we use a sup-script to denote its domain.

Proposition 5.16 *For the aggregate function* $\text{PROD}^{\mathbb{R}^+} : \mathcal{FM}(\mathbb{R}^+) \to \mathbb{R}^+$,

$$\min^{\mathbb{R}^+}_{\text{PROD}}(M_1, M_2) = M_1^{[1,\infty)} \uplus M_2^{[0,1)}$$
$$\max^{\mathbb{R}^+}_{\text{PROD}}(M_1, M_2) = M_1^{[0,1)} \uplus M_2^{[1,\infty)}.$$

On positive integer numbers, product is monotone. Thus $\min^{\mathbb{N}}_{\text{PROD}}(M_1, M_2) = M_1$ and $\max^{\mathbb{N}}_{\text{PROD}}(M_1, M_2) = M_2$. If there are negative numbers in M_1 or in M_2 then to obtain a multiset with minimal product we try to select a multiset which has an odd number of negative numbers.

Proposition 5.17 *The algorithms on Figure 5.1 and Figure 5.2 are correct for the aggregate function* $\text{PROD}^{\mathbb{Z}} : \mathcal{FM}(\mathbb{Z}) \to \mathbb{Z}$.

Proof.

Line 3) If $\prod M_2 < 0$ then clearly removing from M_2 any number different than 1 will only increase the product (since the numbers are integer).

Line 5) If $0 \in M_1$ then the product of any multiset $M \in [M_1, M_2]$ will be zero so we just return the smallest multiset.

Line 9) In this case $0 \in (M_2 - M_1)$. We compute the multiset M' with a minimal product in the interval $[M_1, M_2^0]$ where M_2^0 is obtained from M_2 by removing all zeros. If the product of M' is less than zero then we return it. Adding 0 to M' will increase the product and by the correctness of the rest of the algorithm on the input (M_1, M_2^0) follows that M' is the multiset with minimal product.

Line 10) In this case $0 \in (M_2 - M_1)$ and $\prod \min^{\mathbb{Z}}_{\text{PROD}}(M_1, M_2^0) > 0$. This implies that $M_2 - M_1$ contains only non-negative numbers so the multiset with minimal product is $M_1 \uplus \{0\}$.

Line 14) If $M_2 - M_1$ does not contain any negative numbers then the multiset with minimal product is M_1.

```
1  funct min^Z_PROD(M1, M2) =
2    let p2 := ∏ M2
3    if (p2 < 0) then M2
4    elsif (p2 = 0) then
5          if 0 ∈ M1 then M1
6          else
7                let M2^0 := M2 − M2^{0}
8                let M' := min^Z_PROD(M1, M2^0)
9                if ∏ M' < 0 then M'
10               else M1 ⊎ {0}
11               fi
12         fi
13   else
14         if (M2 − M1)^− = ∅ then M1
15         else
16               M2 − {max((M2 − M1)^−)}
17         fi
18   fi
```

Figure 5.1: Algorithm for $\min^{\mathbb{Z}}_{\text{PROD}}$ on integers.

```
1  funct max^Z_PROD(M1, M2) =
2    let p2 := ∏ M2
3    if (p2 > 0) then M2
4    elsif (p2 = 0) then
5          if 0 ∈ M1 then M1
6          else
7                let M2^0 := M2 − M2^{0}
8                let M' := max^Z_PROD(M1, M2^0)
9                if ∏ M' > 0 then M'
10               else M1 ⊎ {0}
11               fi
12         fi
13   else
14         if (M2 − M1)^− = ∅ then M1
15         else
16               M2 − {max((M2 − M1)^−)}
17         fi
18   fi
```

Figure 5.2: Algorithm for $\max^{\mathbb{Z}}_{\text{PROD}}$ on integers.

Line 16) If the product of M_2 is positive and $M_2 - M_1$ does contain negative numbers then by removing one negative number from $M_2 - M_1$ the product will become negative. If we remove a negative number with a minimal absolute value then the product of the remaining multiset will be minimal. ■

The only differences between the two algorithms for $\min^{\mathbb{Z}}_{\text{PROD}}$ and $\max^{\mathbb{Z}}_{\text{PROD}}$ are the comparisons on lines 3 and 9 and the recursive call on line 8. Note that after the recursive call on line 8 the multiset M_2 does not contain 0. Consequently, the product cannot be 0 and the case $p_2 = 0$ will not be considered again. So, the algorithms make at most one recursive call.

Finally, we present algorithms for $\min^{\mathbb{R}}_{\text{PROD}}$ and $\max^{\mathbb{R}}_{\text{PROD}}$ on the entire set of real numbers.

Proposition 5.18 *The algorithm on Figure 5.3 is correct for the aggregate function* $\text{PROD}^{\mathbb{R}}\colon \mathcal{FM}(\mathbb{R}) \to \mathbb{R}$.

Proof. The algorithms have the same general structure as the algorithms for $\min^{\mathbb{Z}}_{\text{PROD}}$ and $\max^{\mathbb{Z}}_{\text{PROD}}$ on integers on Figure 5.1 and Figure 5.2.

Line 5) We first remove from M_2, the multiset $M_{21}^{(0,1]}$. This is correct because the multiset with minimal product has a negative product. Thus adding to it any set of numbers in the interval $(0, 1)$ will only increase the product.

Line 6) If M_{21} does not contain any numbers in the interval $(-1, 0)$ then multiset in the interval $[M_1, M_2]$ with a minimal product is clearly M_2'.

Line 8) Let N be the multiset with minimal product. It cannot contain an even number of elements from the multiset $M_{21}^{(-1,0)}$ because removing all those numbers will yield a multiset with smaller product. Suppose that N contains an odd number of elements from the multiset $M_{21}^{(-1,0)}$. Because $\prod N < 0$ then N must contain all the elements in $M_{21}^{(-\infty,-1]}$ except the largest one x. However, we can obtain a multiset with smaller product by adding x to N and removing one element from the multiset $M_{21}^{(-1,0)}$.

Lines 12 and 14) In this case we still have to remove an even number of elements to keep the sign of the product negative. Obviously, removing all but the smallest element a from the multiset $M_{21}^{(-1,0)}$ will yield a multiset with smaller product (Line 14). However, if the multiset $M_{21}^{(-\infty,-1)}$ is not empty and its maximal element multiplied by a is smaller than 1 (Line 11) then removing these two elements will yield a multiset with even smaller product (Line 12).

Line 19) The proof is the same as in Proposition 5.17.

Line 21) If the product of M_2 is positive and M_{21} does not contain negative numbers then the multiset with minimal product is clearly equal to $M_1 \uplus M_{21}^{(0,1)}$.

Line 23 If the product of M_2 is positive and M_{21} does contain negative numbers then we can find a multiset such that its product is negative. In this case we remove all elements in the interval $(0, 1]$ which may only increase the product.

```
1  funct min^R_PROD(M_1, M_2) =
2    let p_2 := ∏ M_2
3    let M_21 := (M_2 − M_1)
4    if (p_2 < 0) then
5      let M'_2 := M_2 − M_21^(0,1]
6      if M_21^(−1,0) = ∅ then M'_2
7      else
8          ⌈if |M_21^(−1,0)| is even then M'_2 − M_21^(−1,0)
9           else
10              let a := min(M_21^(−1,0))
11              if M_21^(−∞,−1) ≠ ∅ and max(M_21^(−∞,−1)) ∗ a < 1 then
12                M'_2 − M_21^(−1,0) − {max(M_21^(−∞,−1))}
13              else
14                  M'_2 − M_21^(−1,0) ⊎ {a}
15              fi
16          fi⌋
17     fi
18   elsif (p_2 = 0) then
19       the same code as in Figure 5.1 for p_2 = 0
20   else
21       if M_21^− = ∅ then M_1 ⊎ M_21^(0,1)
22       else
23           let M'_2 := M_2 − M_21^(0,1]
24           if M_21^(−1,−0) = ∅ then M'_2 − max(M_21^(−∞,1])
25           else
26               ⌈if |M_21^(−1,−0)| is odd then M'_2 − M_21^(−1,0)
27                else
28                    let a := min(M_21^(−1,0))
29                    if M_21^(−∞,−1) ≠ ∅ and max(M_21^(−∞,−1)) ∗ a < 1 then
30                      M'_2 − M_21^(−1,0) − {max(M_21^(−∞,−1))}
31                    else
32                        M'_2 − M_21^(−1,0) ⊎ {a}
33                    fi
34                fi⌋
35           fi
36       fi
```

Figure 5.3: Algorithm for $\min^{\mathbb{R}}_{\text{PROD}}$ for product on reals.

Line 24) If all negative numbers are in the interval $(-\infty, -1]$, i.e. $M_{21}^{(-1,0)} = \emptyset$, then by removing one of them we obtain a multiset with a negative product. Moreover, if this is the largest element from the multiset $M_{21}^{(-1,0)} = \emptyset$ then we obtain a multiset with the smallest product. This line is analogous to Line 16 from the algorithm of $\min_{\text{PROD}}^{\mathbb{Z}}$ on Figure 5.1.

Finally, lines 26–34 are analogous to lines 8–16 but instead we want to remove an odd number of elements instead of even. ■

An algorithm for $\max_{\text{PROD}}^{\mathbb{R}}$ can be obtained from the algorithm for $\min_{\text{PROD}}^{\mathbb{R}}$ by simply changing the comparison on line 4 from $<$ to $>$ and on line 19 using the code from the algorithm for $\max_{\text{PROD}}^{\mathbb{Z}}$ on integers (Figure 5.2). We finally point out that the two algorithms can also be applied for product on rational numbers.

5.1.3 Complexity

In this section we study the complexity of ultimate approximating aggregates. The size of the input is measured by the size $|M|$ of the input multiset. From the definition we can immediately obtain upper bounds.

Proposition 5.19 *If the complexity of deciding* R *is in the class* Δ_k^p *then deciding* U_{R}^1 *is in the class* Π_{k+1}^p *and deciding* U_{R}^2 *is in the class* Σ_{k+1}^p.

Because most of the aggregate relations are polynomial then deciding U_{R}^1 and U_{R}^2 will be in the classes co-NP and NP respectively.

Using Proposition 5.1 we can show that for monotone aggregate relations the complexity does not increase.

Proposition 5.20 *If* R *is a monotone aggregate relation then the problems* U_{R}^1 *and* U_{R}^2 *are in the same complexity class as* R.

Finally, we give generic results for subset aggregates.

Proposition 5.21 *If the complexity of deciding* R *is in the class* Δ_k^p *then the decision problems* $\text{R}_\subseteq$, $U_{\text{R}_\subseteq}^1$, *and* $U_{\text{R}_\subseteq}^2$ *are all in the class* Σ_{k+1}^p.

Proof. (Sketch) We only need to show the complexity of $\text{R}_\subseteq$. The complexities of $U_{\text{R}_\subseteq}^1$ and $U_{\text{R}_\subseteq}^2$ follow from Proposition 5.20 because $\text{R}_\subseteq$ is a monotone aggregate relation. To decide if $(M, d) \in \text{R}_\subseteq$ we have to guess a subset $M' \subseteq M$ such that $(M', d) \in \text{R}$. So, if $\text{R} \in \Delta_k^p$ then $\text{R}_\subseteq \in \Sigma_{k+1}^p$. ■

Note that in contrast to Proposition 5.19 both the first and the second components are in the Σ_{k+1}^p class.

For many specific aggregate functions the complexity remains polynomial:

Proposition 5.22 *If* R *is any of the aggregate relations* CARD, MIN, MAX, INF, SUP, $\text{SUM}_{\leq}^{\mathbb{Q}}$, $\text{SUM}_{\geq}^{\mathbb{Q}}$, $\text{PROD}_{\leq}^{\mathbb{Q}}$, $\text{PROD}_{\geq}^{\mathbb{Q}}$ *then* $U_{\text{R}}^1, U_{\text{R}}^2 \in P$.

Proof. (Sketch) The algorithms for CARD are given in Proposition 5.11.

The algorithms for MIN, MAX, INF, and SUP are given in Table 5.1. For MIN and MAX we assume that the partial order relation is polynomial time computable. For INF we assume that the meet operation is polynomial time computable and for SUP that join is polynomial time computable.

The algorithms for $\text{SUM}^{\mathbb{Q}}_{\geq}$ and $\text{SUM}^{\mathbb{Q}}_{\leq}$ are given by Lemma 5.13 and Proposition 5.15 and for $\text{PROD}^{\mathbb{Q}}_{\geq}$ and $\text{PROD}^{\mathbb{Q}}_{\leq}$ by Lemma 5.13 and Proposition 5.18. ∎

For SUM and PROD there is a difference in the complexity of the first and the second component.

Proposition 5.23 *$U^1_{\text{SUM}} \in P$ and $U^1_{\text{PROD}} \in P$.*

Deciding the second component of the ultimate approximation of PROD and SUM are generalizations of the following NP-complete problems (Garey and Johnson, 1979, problems SP13 and SP14):

SUBSET-SUM (SUBSET-PRODUCT)
Instance: A set $S = \{q_1, \ldots, q_n\}$ of positive numbers and integer k.
Question: Is there a subset S' of S such that $\sum S' = k$ ($\prod S' = k$)?

Proposition 5.24 *The problems U^2_{SUM} and U^2_{PROD} are NP-complete.*

5.2 Program Transformation

We introduce one transformation of aggregate programs which is used in conjunction with the transformation of aggregate atoms in the next section. The idea is to replace a subformula $\psi(\mathbf{x})$ in a body of a rule by a new atom $q(\mathbf{x})$ and add a new rule $q(\mathbf{x}) \leftarrow \psi(\mathbf{x})$. The transformation can be used, for example, to change a negative occurrence of a sub-formula into a positive or to simplify aggregate programs to normal aggregate programs.

Definition 5.3 (Folding) *Consider a $\Sigma(\Pi)$-aggregate program P. The* folding *of a formula ψ of a rule $r : p(\mathbf{t}) \leftarrow \varphi[\psi] \in P$ is the $\Sigma(\Pi \cup \{q\})$-program $fold^{\psi}_r(P)$ defined as follows:*

$$\begin{aligned} fold^{\psi}_r(P) = & P - \{r\} \cup \{ \\ & p(\mathbf{t}) \leftarrow \varphi[q(\mathbf{x})]. \\ & q(\mathbf{x}) \leftarrow \psi(\mathbf{x}).\} \end{aligned}$$

where $q \notin \Pi$ is a new predicate symbol and $\mathbf{x} = FV(\psi)$.

Example 5.2 Consider a program P consisting of the following rule

$$r\colon p(x) \leftarrow \neg\exists y.\ r(x, y).$$

One way to interpret this rule is that $r(x, y)$ encodes a partial function from x to y and $p(x)$ is true if this function is not defined for x. After folding r on the formula $\exists y.\ r(x, y)$ which is positive we obtain the following program $P' = fold_r^{\exists y.\ r(x,y)}(P)$:

$$p(x) \leftarrow \neg q(x).$$
$$q(x) \leftarrow \exists y.\ r(x, y). \qquad \square$$

One condition which guarantees that the $fold$ transformation preserves the set of partial stable models is the formula ψ to be positive.

Proposition 5.25 *If ψ is a positive formula then there is a one to one correspondence between the three-valued stable models of P and those of $fold_r^{\psi}(P)$.*

To see why the transformation does not preserve the set of three-valued stable models when the formula ψ is negative consider the following example.

Example 5.3 Let P be the program consisting of the rule

$$1 : p \leftarrow \neg\neg p.$$

The result of $fold_1^{\neg p}(P)$ is the program:

$$p \leftarrow \neg q.$$
$$q \leftarrow \neg p.$$

The first program has only one exact stable model $\emptyset$ while the second program has two: $\{p\}$ and $\{q\}$. $\square$

5.3 Transformations of Aggregate Atoms

In this section we study transformations of aggregate atoms to formulas which are equivalent in three-valued logic. Every transformation is specific to a class of aggregate atoms and exploits specific properties of the atoms in this class.

Definition 5.4 *For two formulas φ and ψ we denote that they are equivalent in two valued logic with $\varphi = \psi$ and that they are equivalent in three-valued logic with $\varphi =_3 \psi$, that is $\mathcal{H}_{\tilde{I}}(\varphi) = \mathcal{H}_{\tilde{I}}(\psi)$ for any three-valued interpretation $\tilde{I}$.*

Because two-valued interpretations are also three-valued interpretations equivalence in three-valued logic implies equivalence in two-valued logic. The opposite is not true. For example, there are no tautologies in three-valued logic. So, $p \wedge (q \vee \neg q) = p$ but $p \wedge (q \vee \neg q) \neq_3 p$.

By replacing a sub-formula φ in a body of a rule in a program P with a formula ψ such that $\varphi =_3 \psi$ we obtain a program P' such that $\Phi_P^{aggr} = \Phi_{P'}^{aggr}$. Consequently P and P' have the same set of three-valued stable models. We note that equivalence of the Φ_P^{aggr} operators of two programs implies strong equivalence of logic programs (Lifschitz et al., 2001): P_1 is *strongly equivalent* to P_2 if for any program Π, $P_1 \cup \Pi$ have the same set of exact stable models as $P_2 \cup \Pi$.

5.3.1 Complement and Negation

A normal aggregate program may contain only aggregate atoms but not the negation of aggregate atoms. The following proposition shows that this is not a limitation as long as the aggregate signature contains the complements of all aggregate relations.

Definition 5.5 *The* complement *of a relation $R \subseteq D_1 \times D_2$ is a relation $\overline{R} \subseteq D_1 \times D_2$ defined as $(a, b) \in \overline{R}$ if $(a, b) \notin R$.*

Proposition 5.26

$$\neg\mathrm{R}(\lambda\mathbf{x}.\ e(\mathbf{x}), \{\mathbf{x} \mid \varphi(\mathbf{x})\}, t) =_3 \overline{\mathrm{R}}(\lambda\mathbf{x}.\ e(\mathbf{x}), \{\mathbf{x} \mid \varphi(\mathbf{x})\}, t).$$

For a derived aggregate relation of the form F_r its complement is equivalent to the combination of F and the complement of r.

Proposition 5.27 *If* F *is a total aggregate function then* $\overline{\mathrm{F}_r} = \mathrm{F}_{\overline{r}}$.

Proof. $(S, d) \in \overline{\mathrm{F}_r}$ iff $(S, d) \notin \mathrm{F}_r$ iff $(\mathrm{F}(S), d) \notin r$ iff $(\mathrm{F}(S), d) \in \overline{r}$ iff $(S, d) \in \mathrm{F}_{\overline{r}}$. ■

By combining the above two propositions we can replace a negation of a derived aggregate atom by the aggregate atom obtained by the combination of the aggregate function and the complement of the relation. For example $\neg\mathrm{SUM}_{\geq}(s, t) =_3 \mathrm{SUM}_{<}(s, t)$. This will be useful for replacing an anti-monotone aggregate relation by the negation of a monotone aggregate.

5.3.2 Aggregates as Quantifiers

Let $\mathrm{FORALL} \subseteq \mathcal{M}(D_1) \times D_2$ and $\mathrm{EXISTS} \subseteq \mathcal{M}(D_1) \times D_2$ be aggregate relations corresponding to the quantifiers in classical logic. They are defined as $(M, d) \in \mathrm{FORALL}$ if and only if $M(x) > 0$ for every $x \in D_1$ and $(M, d) \in \mathrm{EXISTS}$ if and only if $M(x) > 0$ for some $x \in D_1$. We show that the ultimate approximations of FORALL and EXISTS are equal to the interpretation of the $\forall$ and $\exists$ quantifiers in three-valued logic (see Definition 2.30).

Proposition 5.28

$$\mathrm{EXISTS}(\lambda x.\ x, \{x \mid \varphi\}, t) =_3 \exists x.\ \varphi$$
$$\mathrm{FORALL}(\lambda x.\ x, \{x \mid \varphi\}, t) =_3 \forall x.\ \varphi$$

Proof. First, observe that EXISTS and FORALL are monotone aggregate relations. Fix a three-valued interpretation $\tilde{I}$ and let $(M_1, M_2) = [\![\{x \mid \varphi\}]\!]_{\tilde{I}}$. The next two

columns contain the proofs for the first and second components of $\mathcal{H}$ for the EXISTS quantifier.

$$
\begin{array}{ll}
\mathcal{H}^1_{\tilde{I}}(\text{EXISTS}(\lambda x.\ x, \{x \mid \varphi\}, t)) = \mathbf{t} & \mathcal{H}^2_{\tilde{I}}(\text{EXISTS}(\lambda x.\ x, \{x \mid \varphi\}, t)) = \mathbf{t} \\
\Leftrightarrow ((M_1, M_2), [\![t]\!]) \in U^1_{\text{EXISTS}} & \Leftrightarrow ((M_1, M_2), [\![t]\!]) \in U^2_{\text{EXISTS}} \\
\Leftrightarrow (M_1, [\![t]\!]) \in \text{EXISTS} & \Leftrightarrow (M_2, [\![t]\!]) \in \text{EXISTS} \\
\Leftrightarrow \exists d \in D_1.\ M_1(d) > 0 & \Leftrightarrow \exists d \in D_1.\ M_2(d) > 0 \\
\Leftrightarrow \exists d \in D_1.\ \mathcal{H}^1_{\tilde{I}}(\varphi(d)) = \mathbf{t} & \Leftrightarrow \exists d \in D_1.\ \mathcal{H}^2_{\tilde{I}}(\varphi(d)) = \mathbf{t} \\
\Leftrightarrow \mathcal{H}^1_{\tilde{I}}(\exists x.\ \varphi(x)) = \mathbf{t} & \Leftrightarrow \mathcal{H}^2_{\tilde{I}}(\exists x.\ \varphi(x)) = \mathbf{t}
\end{array}
$$

The proof for the FORALL aggregate is analogous. ∎

5.3.3 Derived Aggregate Relations

An interesting question about derived aggregate relations of the form R_P is whether they are more expressive than a conjunction of R and P. More precisely we want to study the following question:

$$\text{R}_P(s,t) =_3 \exists z.\ \text{R}(s,z) \wedge P(z,t)$$

It turns out that the answer to this question is negative and if using the derived aggregate relation R_P the truth value is at least as precise as when using a combination of R and P. The precise relationship is given by the following lemma.

Lemma 5.29

$$\mathcal{H}_{\tilde{I}}(\exists z.\ \text{R}(s,z) \wedge P(z,t)) \leq_t \mathcal{H}_{\tilde{I}}(\text{R}_P(s,t)).$$

Proof. Fix a three-valued interpretation $\tilde{I}$ and let $(M_1, M_2) = [\![s]\!]_{\tilde{I}}$ and $d = [\![t]\!]$.

$$
\begin{array}{ll}
& \mathcal{H}^1_{\tilde{I}}(\exists z.\ \text{R}(s,z) \wedge P(z,t)) = \mathbf{t} \\
\Leftrightarrow & \exists z.\ \mathcal{H}^1_{\tilde{I}}(\text{R}(s,z)) = \mathbf{t} \wedge \mathcal{H}^1_{\tilde{I}}(P(z,t)) = \mathbf{t} \\
\Leftrightarrow & \exists z.\ U^1_{\text{R}}((M_1, M_2), z) \wedge P^{\mathcal{D}}(z,d) \\
\Leftrightarrow & \exists z.\ \forall M \in [M_1, M_2].\ \text{R}(M,z) \wedge P^{\mathcal{D}}(z,d) \qquad (*) \\
\Rightarrow & \forall M \in [M_1, M_2].\ \exists z.\ \text{R}(M,z) \wedge P^{\mathcal{D}}(z,d) \\
\Leftrightarrow & \forall M \in [M_1, M_2].\ \text{R}_P(M,d) \\
\Leftrightarrow & U^1_{\text{R}_P}((M_1, M_2), d) \\
\Leftrightarrow & \mathcal{H}^1_{\tilde{I}}(\text{R}_P(s,t)) = \mathbf{t}
\end{array}
$$

The place where the proof cannot be reversed is after step (*) where the existential quantifier is pushed in the scope of a universal quantifier. There is no such a problem for the over-estimate of the truth value because the ultimate approximating

aggregate uses an existential quantifier. So, we can show equivalence for the two formulas:

$$\begin{aligned}
&\mathcal{H}^2_{\tilde{I}}(\exists z.\ \mathrm{R}(s,z) \wedge P(z,t)) = \mathbf{t} \\
\Leftrightarrow &\exists z.\ \mathcal{H}^2_{\tilde{I}}(\mathrm{R}(s,z)) = \mathbf{t} \wedge \mathcal{H}^2_{\tilde{I}}(P(z,t)) = \mathbf{t} \\
\Leftrightarrow &\exists z.\ U^2_{\mathrm{R}}((M_1,M_2),z) \wedge P^{\mathcal{D}}(z,d) \\
\Leftrightarrow &\exists z.\ \exists M \in [M_1,M_2].\ \mathrm{R}(M,z) \wedge P^{\mathcal{D}}(z,d) \\
\Leftrightarrow &\exists M \in [M_1,M_2].\ \exists z.\ \mathrm{R}(M,z) \wedge P^{\mathcal{D}}(z,d) \\
\Leftrightarrow &\exists M \in [M_1,M_2].\ \mathrm{R}_P(M,d) \\
\Leftrightarrow &U^2_{\mathrm{R}_P}((M_1,M_2),d) \\
\Leftrightarrow &\mathcal{H}^2_{\tilde{I}}(\mathrm{R}_P(s,t)) = \mathbf{t}
\end{aligned}$$

■

The next example demonstrates that the truth value of R_P can be strictly more precise than the combination of R and P.

Example 5.4 Consider the derived aggregate relation $\mathrm{SUM}_{\geq}$ and the three-valued multiset $(M_1,M_2) = (\{1\},\{1,2\})$. We have that $((M_1,M_2),1) \in U^1_{\mathrm{SUM}_{\geq}}$ because $\mathrm{SUM}(M_1) \geq 1$ and $\mathrm{SUM}(M_2) \geq 1$. However there does not exists a number z such that $\mathrm{SUM}(\{1\}) = z$ and $\mathrm{SUM}(\{1,2\}) = z$. Consequently the statement $\exists z.\ U^1_{\mathrm{SUM}}((M_1,M_2),z) \wedge z \geq 1$ is false. □

We can obtain an equivalent translation in the special case when R_P is a monotone aggregate relation. We also need to translate $\mathrm{R}_P(s,t)$ to $\exists z.\ \mathrm{R}_{\subseteq}(s,z) \wedge P(z,t)$ by using the subset aggregate $\mathrm{R}_{\subseteq}$ of R.

Proposition 5.30 *If R_P is a monotone aggregate relation then*

$$\exists z.\ \mathrm{R}_{\subseteq}(s,z) \wedge P(z,t) =_3 \mathrm{R}_P(s,t)$$

Proof. Fix a three-valued interpretation $\tilde{I}$ and let $(M_1,M_2) = [\![s]\!]_{\tilde{I}}$ and $d = [\![t]\!]$.

$$\begin{aligned}
&\mathcal{H}^1_{\tilde{I}}(\exists z.\ \mathrm{R}_{\subseteq}(s,z) \wedge P(z,t)) = \mathbf{t} \\
\Leftrightarrow &\exists z.\ \mathcal{H}^1_{\tilde{I}}(\mathrm{R}_{\subseteq}(s,z)) = \mathbf{t} \wedge \mathcal{H}^1_{\tilde{I}}(P(z,t)) = \mathbf{t} \\
\Leftrightarrow &\exists z.\ U^1_{\mathrm{R}_{\subseteq}}((M_1,M_2),z) \wedge P(z,d) \\
\Leftrightarrow &\exists z.\ \mathrm{R}_{\subseteq}(M_1,z) \wedge P(z,d) \\
\Leftrightarrow &\exists z.\ \exists M' \subseteq M_1.\ \mathrm{R}(M',z) \wedge P(z,d) \\
\Leftrightarrow &\exists M' \subseteq M_1.\ \exists z.\ \mathrm{R}(M',z) \wedge P(z,d) \\
\Leftrightarrow &\exists M' \subseteq M_1.\ \mathrm{R}_P(M',d) \\
\Leftrightarrow &\mathrm{R}_P(M_1,d) && \text{because } R_P \text{ is monotone} \\
\Leftrightarrow &U^1_{\mathrm{R}_P}((M_1,M_2),d) \\
\Leftrightarrow &\mathcal{H}^1_{\tilde{I}}(\mathrm{R}_P(s,t)) = \mathbf{t}
\end{aligned}$$

The proof for $\mathcal{H}^2$ was given in Lemma 5.29. ∎

Note that if a positive aggregate atom $R_P(s,t)$ with a monotone derived aggregate relation R_P is translated to $\exists z.\ \text{R}_\subseteq(s,z) \wedge P(z,t)$, the aggregate atom $R_\subseteq(s,z)$ and the entire formula $\exists z.\ \text{R}_\subseteq(s,z) \wedge P(z,t)$ are still positive. So, applying this transformation to a positive aggregate program P preserves the property of positiveness.

Example 5.5 Consider the following rule from the formulation of the Company Controls problem (Example 3.5):

$$\mathit{controls}(X,Y) \leftarrow \text{SUM}_>(\lambda(Z,S).\ S, \{(Z,S) \mid \mathit{cv}(X,Z,Y,S)\}, 0.5).$$

On positive numbers the derived aggregate relation $\text{SUM}_>$ is monotone. So, using Proposition 5.30, the aggregate can be rewritten as

$$\mathit{controls}(X,Y) \leftarrow \text{SUM}_\subseteq(\lambda(Z,S).\ S, \{(Z,S) \mid \mathit{cv}(X,Z,Y,S)\}, Z) \wedge Z > 0.5.$$

The same formulation for the Company Controls example with a subset aggregate was given by Van Gelder (Van Gelder, 1992) who makes the following comment:

> In this problem, a company A controls company C if some subset $\{B_i\}$ of the companies controlled by A own a combined portion ...
>
> Observe the phrase "some subset" in our wording. We believe this represents the intuition of the problem: once the combined ownership exceeds .50 we do not care about its exact value.

We believe that Proposition 5.30 gives a formal explanation to the intuitions of Van Gelder. As demonstrated by Example 5.4 if we use in the translation the SUM aggregate instead of $\text{SUM}_\subseteq$ the semantics may be too weak to model correctly the problem. □

5.3.4 Extrema Predicates

It is possible to transform all extrema aggregate relations entirely to formulas without aggregates which are equivalent in the 3-valued logic.

Proposition 5.31 *For any formula* φ

$$\begin{aligned}
\text{MIN}(\{x \mid \varphi(x)\}, t) &=_3 \varphi(t) \wedge \neg\exists x.\ (\varphi(x) \wedge x < t)\\
\text{MAX}(\{x \mid \varphi(x)\}, t) &=_3 \varphi(t) \wedge \neg\exists x.\ (\varphi(x) \wedge t < x)
\end{aligned}$$

$$\begin{aligned}
\text{LB}(\{x \mid \varphi(x)\}, t) &=_3 \forall x.\ \varphi(x) \rightarrow t \leq x\\
\text{UB}(\{x \mid \varphi(x)\}, t) &=_3 \forall x.\ \varphi(x) \rightarrow x \leq t
\end{aligned}$$

$$\begin{aligned}
\text{INF}(\{x \mid \varphi(x)\}, t) &=_3 \text{LB}(\{x \mid \varphi(x)\}, t) \wedge (\forall z.\ \text{LB}(\{x \mid \varphi(x)\}, z) \rightarrow z \leq t)\\
\text{SUP}(\{x \mid \varphi(x)\}, t) &=_3 \text{UB}(\{x \mid \varphi(x)\}, t) \wedge (\forall z.\ \text{UB}(\{x \mid \varphi(x)\}, z) \rightarrow t \leq z)
\end{aligned}$$

Proof. The proofs are straightforward and are omitted. ∎

Example 5.6 Consider the rule for the $sp/3$ predicate from the programs of the Shortest Path problem from Example 3.6 and Example 4.6:

$$sp(X, Y, S) \leftarrow \text{MIN}(\lambda C.\ C, \{C \mid cp(X, Y, C)\}, S).$$

Using the transformation of MIN from Proposition 5.31, it is is equivalent to:

$$sp(X, Y, S) \leftarrow cp(X, Y, S) \wedge \neg\exists C_1.\ cp(X, Y, C_1) \wedge C_1 < S$$

Because the formula $\exists C_1.\ cp(X, Y, C_1) \wedge C_1 < S$ is positive, the second line can be folded on it (Proposition 5.25) and we obtain the following normal program:

$$\begin{aligned} sp(X, Y, S) &\leftarrow cp(X, Y, S), not\ bp(X, Y, S). \\ bp(X, Y, S) &\leftarrow cp(X, Y, C_1), C_1 < S. \end{aligned}$$

This translation of the MIN aggregate relation has been used in several previous works on aggregates (Ganguly et al., 1995; Van Gelder, 1992). □

5.3.5 Summation

We now show how to translate an aggregate atom $\text{SUM}_{\geq}(\lambda\mathbf{x}.\ e, \{\mathbf{x} \mid \varphi\}, t)$ where the domain of the input multiset of $\text{SUM}_{\geq}$ is some subset of real numbers to a combination of monotone and anti-monotone aggregates. This transformation is used later for a practical implementation of problems using such aggregates. Recall that $\text{SUM}_{\geq}$ is monotone for positive numbers and anti-monotone on negative numbers. The idea is then to use two separate SUM-aggregates — one taking only positive numbers as input and the other only negative numbers. The tricky part is how to combine the results of the two atoms. First, define two formulas φ^+ and φ^- which compute only the subset of positive, respectively negative, elements from the set of the aggregate relation:

$$\begin{aligned} \varphi^+(\mathbf{x}) &\equiv \varphi(\mathbf{x}) \wedge e(\mathbf{x}) > 0 \\ \varphi^-(\mathbf{x}) &\equiv \varphi(\mathbf{x}) \wedge e(\mathbf{x}) < 0 \end{aligned}$$

Proposition 5.32

$$\begin{aligned} \text{SUM}_{\geq}(\lambda\mathbf{x}.\ e, \{\mathbf{x} \mid \varphi\}, t) =_3 \exists z.\ &\text{SUM}_{\subseteq}(\lambda\mathbf{x}.\ e, \{\mathbf{x} \mid \varphi^+\}, z) \wedge \\ &\text{SUM}_{\geq}(\lambda\mathbf{x}.\ e, \{\mathbf{x} \mid \varphi^-\}, t - z). \end{aligned}$$

Proof. We give the proof only for $\mathcal{H}^1$. The proof for $\mathcal{H}^2$ is analogous. Fix a three-valued interpretation $\tilde{I}$ and let $(M_1, M_2) = [\![\lambda \mathbf{x}.\ e]\!]([\![\{\mathbf{x} \mid \varphi\}]\!]_{\tilde{I}})$ and $d = [\![t]\!]$.

$$
\begin{aligned}
&\mathcal{H}^1_{\tilde{I}}(\text{SUM}_{\geq}(\lambda \mathbf{x}.\ e, \{\mathbf{x} \mid \varphi\}, t)) = \mathbf{t} \\
\Leftrightarrow &((M_1, M_2), d) \in \text{SUM}^1_{\geq} \\
\Leftrightarrow &\textstyle\sum M_1^+ + \sum M_2^- \geq d \\
\Leftrightarrow &\textstyle\sum M_1^+ \geq d - \sum M_2^- \\
\Leftrightarrow &\exists z.\ \textstyle\sum M_1^+ \geq z \wedge z \geq d - \sum M_2^- \\
\Leftrightarrow &\exists z.\ \textstyle\sum M_1^+ \geq z \wedge \sum M_2^- \geq d - z \qquad (*) \\
\Leftrightarrow &\exists z.\ \exists M' \subseteq M_1^+.\ \textstyle\sum M' = z \wedge \sum M_2^- \geq d - z \\
\Leftrightarrow &\exists z.\ \text{SUM}_{\subseteq}(M_1^+, z) \wedge \text{SUM}_{\geq}(M_2^-, d - z) \\
\Leftrightarrow &\exists z.\ ((M_1^+, M_2^+), z) \in U^1_{\text{SUM}_{\subseteq}} \wedge ((M_1^-, M_2^-), d - z) \in U^1_{\text{SUM}_{\geq}} \\
\Leftrightarrow &\exists z.\ \mathcal{H}^1_{\tilde{I}}(\text{SUM}_{\subseteq}(\lambda \mathbf{x}.\ e, \{\mathbf{x} \mid \varphi^+\}, z)) = \mathbf{t} \wedge \mathcal{H}^1_{\tilde{I}}(\text{SUM}_{\geq}(\lambda \mathbf{x}.\ e, \{\mathbf{x} \mid \varphi^-\}, t - z)) = \mathbf{t} \\
\Leftrightarrow &\mathcal{H}^1_{\tilde{I}}(\exists z.\ \text{SUM}_{\subseteq}(\lambda \mathbf{x}.\ e, \{\mathbf{x} \mid \varphi^+\}, z) \wedge \text{SUM}_{\geq}(\lambda \mathbf{x}.\ e, \{\mathbf{x} \mid \varphi^-\}, t - z)) = \mathbf{t}
\end{aligned}
$$

One of the crucial steps is after (*) where we turn $\sum M_1^+ \geq z$ into $\exists M' \subseteq M_1^+.\ \sum M' = z$. In the forward direction, suppose that (*) holds for some z. By choosing $M' = M_1^+$ and $z' = \sum M_1^+$ we have $z \leq z'$. Consequently, $d - z \geq d - z'$, so z' satisfies the second inequality. In the other direction, suppose that there is a multiset $M' \subseteq M_1^+$ such that $z = \sum M'$ and $\sum M_2^- \geq d - z$. Consequently, $\sum M_1^+ \geq \sum M' = z$, so (*) is also satisfied. ∎

In a resolution based system with a left to right selection rule like Prolog or an abductive system (Kakas et al., 2000; Kakas et al., 2001) the call to the first aggregate atom $\text{SUM}_{\subseteq}(\lambda \mathbf{x}.\ e, \{\mathbf{x} \mid \varphi^+\}, z)$ is used to compute a large enough value of z which can satisfy the second aggregate atom $\text{SUM}_{\geq}(\lambda \mathbf{x}.\ e, \{\mathbf{x} \mid \varphi^-\}, t - z)$. This may trigger backtracking over φ^+ to generate enough tuples for the input multiset of the aggregate. We point out that instead of the subset aggregate $\text{SUM}_{\subseteq}$ we could have used $\text{SUM}_{\geq}$. However, keeping the above execution mechanism in mind the subset aggregate $\text{SUM}_{\subseteq}$ is preferable. The reason is that for a given finite three-valued input multiset (M_1, M_2) there are only finitely many sub-multisets M' of M_1 and, consequently, finitely many numbers d such that $((M_1, M_2), d) \in U^1_{\text{SUM}_{\subseteq}}$. In fact, as we will see in the section on the implementation, it is not necessary to consider all of the sub-multisets of M_1 but only those which form an initial segment for some fixed ordering on the elements of M_1. On the other hand, there are infinitely many numbers x which satisfy $((M_1, M_2), x) \in U^1_{\text{SUM}_{\geq}}$. A correct implementation of $\text{SUM}_{\geq}$ should be able to return all of them.

Notice that the aggregate atom $\text{SUM}_{\geq}(\lambda \mathbf{x}.\ e, \{\mathbf{x} \mid \varphi^-\}, t - z)$ is anti-monotone because its input is a set of negative numbers. It can be translated to a monotone aggregate by introducing a double negation in front of the atom and switching

from $\neg\text{SUM}_{\geq}$ to $\text{SUM}_{<}$:

$$\text{SUM}_{\geq}(\lambda\mathbf{x}.\, e, \{\mathbf{x} \mid \varphi^{-}\}, t - z) =_3 \neg\text{SUM}_{<}(\lambda\mathbf{x}.\, e, \{\mathbf{x} \mid \varphi^{-}\}, t - z).$$

5.3.6 General Transformation

We now present a general transformation of aggregate atoms to formulas. We use it in the next section to give an argument about the correctness of our implementation.

Let A be a closed aggregate atom of the form $\text{R}(f, \{\mathbf{x} \mid \varphi(\mathbf{x})\}, t)$ where $\mathbf{x} = x_1, \ldots, x_n$ and for every $i = 1, \ldots, n$, the type of the variable x_i is s_i. The translation of A is a disjunction of the form $\bigvee_{(S_1,S_2)\in I} C^{\varphi}_{(S_1,S_2)}$ where I is an index set. Let $S = D_{s_1} \times \cdots \times D_{s_n}$. Every $C^{\varphi}_{(S_1,S_2)}$ is a conjunction of ground instances of φ and $\neg\varphi$. The instances are determined by the three-valued set (S_1, S_2) where $S_1 \subseteq S_2 \subseteq S$. The set S_1 contains the tuples $\mathbf{d}$ for which $C^{\varphi}_{(S_1,S_2)}$ contains an instance $\varphi(\mathbf{d})$ of φ and $S - S_2$ contains the tuples $\mathbf{d}$ for which $C^{\varphi}_{(S_1,S_2)}$ contains an instance $\neg\varphi(\mathbf{d})$ of φ:

$$C^{\varphi}_{(S_1,S_2)} = \bigwedge\{\varphi(\mathbf{d}) \mid \mathbf{d} \in S_1\} \wedge \bigwedge\{\neg\varphi(\mathbf{d}) \mid \mathbf{d} \in (S - S_2)\}.$$

The index set I is defined as

$$I = \{(S_1, S_2) \mid \forall S' \in [S_1, S_2] \colon (\llbracket f \rrbracket(S'), \llbracket t \rrbracket) \in \text{R}\}.$$

Note that the condition $(\llbracket f \rrbracket(S'), \llbracket t \rrbracket) \in \text{R}$ is equivalent to $(\llbracket f \rrbracket(S_1, S_2), \llbracket t \rrbracket) \in U^1_{\text{R}}$. The translation of an aggregate atom is defined as

$$\begin{aligned} tr(\text{R}(f, \{\mathbf{x} \mid \varphi\}, t)) &= \bigvee_{(S_1,S_2)\in I} C^{\varphi}_{(S_1,S_2)} \\ &= \bigvee\{C^{\varphi}_{(S_1,S_2)} \mid S_1 \subseteq S_2 \subseteq S \text{ and } \forall S' \in [S_1, S_2] \colon (\llbracket f \rrbracket(S'), \llbracket t \rrbracket) \in \text{R}\}. \end{aligned}$$

Proposition 5.33 *$A =_3 tr(A)$ for every closed aggregate atom A.*

Proof. Let A be a closed aggregate atom of the form $\text{R}(f, \{\mathbf{x} \mid \varphi\}, t)$ and let $d = \llbracket t \rrbracket$. We have to prove equivalence for the two components $\mathcal{H}^1$ and $\mathcal{H}^2$ of $\mathcal{H}$ and for each component we have to prove two directions. Fix a three-valued interpretation $\tilde{I}$ and let $(S_1, S_2) = \llbracket\{\mathbf{x} \mid \varphi\}\rrbracket_{\tilde{I}}$. In the proof we treat the disjunction $tr(A)$ as a set of formulas.

($\mathcal{H}^1$, $\Rightarrow$) Suppose that $\mathcal{H}^1_{\tilde{I}}(A) = \mathbf{t}$. This implies that $(\llbracket f \rrbracket(S_1, S_2), d) \in U^1_{\text{R}}$ and so $\forall S \in [S_1, S_2] \colon (\llbracket f \rrbracket(S), d) \in \text{R}$. So, $C^{\varphi}_{(S_1,S_2)} \in tr(A)$. Moreover, $\mathcal{H}^1_{\tilde{I}}(C^{\varphi}_{(S_1,S_2)}) = t$ and consequently $\mathcal{H}^1_{\tilde{I}}(tr(A)) = \mathbf{t}$.

($\mathcal{H}^1$, $\Leftarrow$) Suppose that $\mathcal{H}^1_{\tilde{I}}(tr(A)) = \mathbf{t}$. This means that there is a disjunct $C^{\varphi}_{(S_1',S_2')} \in tr(A)$ such that $\mathcal{H}^1_{\tilde{I}}(C^{\varphi}_{(S_1',S_2')}) = \mathbf{t}$. This means that $(S_1', S_2') \leq_p (S_1, S_2)$

(because S_1 contains all tuples $\mathbf{d}$ for which $\mathcal{H}_{\tilde{I}}(\varphi(\mathbf{d})) = \mathbf{t}$ and S_2 contains all tuples $\mathbf{d}$ for which $\mathcal{H}_{\tilde{I}}(\varphi(\mathbf{d})) \neq \mathbf{f}$) and so $[S_1, S_2] \subseteq [S_1', S_2']$. By definition of $tr(A)$, $C^{\varphi}_{(S_1', S_2')} \in tr(A)$ only if for all $S' \in [S_1', S_2']$, $(\llbracket f \rrbracket(S'), d) \in \text{R}$. Because $[S_1, S_2] \subseteq [S_1', S_2']$ then also for all $S' \in [S_1, S_2]$, $(\llbracket f \rrbracket(S'), d) \in \text{R}$ and hence $\mathcal{H}^1_{\tilde{I}}(A) = \mathbf{t}$.

($\mathcal{H}^2$, $\Rightarrow$) Assume that $\mathcal{H}^2_{\tilde{I}}(A) = \mathbf{t}$. This means that there exists $S' \in [S_1, S_2]$ such that $(\llbracket f \rrbracket(S'), d) \in \text{R}$. Consequently, $C^{\varphi}_{(S', S')} \in tr(A)$ and it has the form

$$C^{\varphi}_{(S', S')} = \underbrace{\bigwedge_{\mathbf{d} \in S_1} \varphi(\mathbf{d}) \wedge \bigwedge_{\mathbf{d} \in S - S_2} \neg\varphi(\mathbf{d})}_{= C^{\varphi}_{(S_1, S_2)}} \wedge \Big(\bigwedge_{\mathbf{d} \in S' - S_1} \varphi(\mathbf{d}) \wedge \bigwedge_{\mathbf{d} \in S_2 - S'} \neg\varphi(\mathbf{d}) \Big).$$

Clearly, $\mathcal{H}_{\tilde{I}}(C^{\varphi}_{(S_1, S_2)}) = \mathbf{t}$ for the first half of the formula. For the second half of the formula

$$\mathcal{H}_{\tilde{I}}(\bigwedge_{\mathbf{d} \in S' - S_1} \varphi(\mathbf{d}) \wedge \bigwedge_{\mathbf{d} \in S_2 - S'} \neg\varphi(\mathbf{d})) = \mathbf{u}$$

because $S' \in [S_1, S_2]$. So, $\mathcal{H}_{\tilde{I}}(C^{\varphi}_{(S', S')}) = \mathbf{u}$ which means that $\mathcal{H}^2_{\tilde{I}}(C^{\varphi}_{(S', S')}) = \mathbf{t}$. Consequently $\mathcal{H}^2_{\tilde{I}}(tr(A)) = \mathbf{t}$.

($\mathcal{H}^2$, $\Leftarrow$) Assume that

$$\mathcal{H}^2_{\tilde{I}}(A) = \mathbf{f} \quad \text{and} \quad \mathcal{H}^2_{\tilde{I}}(tr(A)) = \mathbf{t} \tag{5.1}$$

The first assumption implies that there does not exist a set $S' \in [S_1, S_2]$ such that $(\llbracket f \rrbracket(S'), d) \in \text{R}$. In a formal notation

$$\forall S' \in [S_1, S_2].\ (\llbracket f \rrbracket(S), d) \notin \text{R}. \tag{5.2}$$

The second assumption of (5.1) is true if and only if

$$\exists C^{\varphi}_{(S_1', S_2')} \in tr(A) \colon \mathcal{H}^2_{\tilde{I}}(C^{\varphi}_{(S_1', S_2')}) = \mathbf{t}. \tag{5.3}$$

By definition of $tr(A)$ for the formula $C^{\varphi}_{(S_1', S_2')}$ we have

$$\forall S' \in [S_1', S_2'].\ (\llbracket f \rrbracket(S'), d) \in \text{R}.$$

Together with (5.2) this implies

$$[S_1, S_2] \cap [S_1', S_2'] = \emptyset. \tag{5.4}$$

The next step is to show the following:

$$\neg(S_1' \subseteq S_2) \vee \neg(S_1 \subseteq S_2') \tag{5.5}$$

Assume the opposite, i.e. $S_1' \subseteq S_2$ and $S_1 \subseteq S_2'$ and let $U = S_1 \cup S_1'$. Because $S_1 \subseteq S_2$ and $S_1' \subseteq S_2$ then also $U \subseteq S_2$. Similarly, $U \subseteq S_2'$. Thus, $U \in [S_1, S_2]$ and $U \in [S_1', S_2']$ which is a contradiction with (5.4).

Finally, we do a case analysis of (5.5). Suppose that $S_1' \not\subseteq S_2$. This means that there exists a tuple $\mathbf{d} \in S_1'$ such that $\mathbf{d} \notin S_2$. For this tuple we have $\mathcal{H}^2_{\tilde{I}}(\varphi(\mathbf{d})) = \mathbf{f}$ and consequently $\mathcal{H}^2_{\tilde{I}}(C^{\varphi}_{(S_1',S_2')}) = \mathbf{f}$ which is a contradiction with (5.3) and consequently with (5.1). Now, consider the case when $S_1 \not\subseteq S_2'$. This means that there exists a tuple $\mathbf{d} \in S_1$ such that $\mathbf{d} \notin S_2'$. For this tuple $\mathcal{H}^1_{\tilde{I}}(\varphi(\mathbf{d})) = \mathbf{t}$ (by definition of S_1) and consequently $\mathcal{H}^2_{\tilde{I}}(\neg\varphi(\mathbf{d})) = \mathbf{f}$. On the other hand, $\mathbf{d} \notin S_2'$ implies $\mathbf{d} \in (S - S_2')$, so $\neg\varphi(\mathbf{d}) \in C^{\varphi}_{(S_1',S_2')}$. Because $\mathcal{H}^2_{\tilde{I}}(\neg\varphi(\mathbf{d})) = \mathbf{f}$ then $\mathcal{H}^2_{\tilde{I}}(C^{\varphi}_{(S_1',S_2')}) = \mathbf{f}$ which is again a contradiction with (5.3). ∎

For closed aggregate atoms with a monotone aggregate relation we can define a simpler translation tr_{mon} as follows:

$$tr_{mon}(\mathrm{R}(f,s,t)) = \bigvee\{C^{\varphi}_{(S_1,S)} \mid S = D_{s_1} \times \cdots \times D_{s_n}, S_1 \subseteq S, \text{ and } (\llbracket f \rrbracket(S_1), \llbracket t \rrbracket) \in \mathrm{R}\}.$$

Notice that this formula does not contain $\neg\varphi(\mathbf{d})$ instances of φ because $S - S = \emptyset$. In a similar way, for closed aggregate atoms with an anti-monotone aggregate relation, define

$$tr_{anti}(\mathrm{R}(f,s,t)) = \bigvee\{C^{\varphi}_{(\emptyset,S_2)} \mid S = D_{s_1} \times \cdots \times D_{s_n}, S_2 \subseteq S, \text{ and } (\llbracket f \rrbracket(S_2), \llbracket t \rrbracket) \in \mathrm{R}\}.$$

Likewise, $tr_{anti}(\mathrm{R}(f,s,t))$ does not contain $\varphi(\mathbf{d})$ instances of φ. Finally, let

$$tr_{ma}(\mathrm{R}(f,s,t)) = \begin{cases} tr_{mon}(\mathrm{R}(f,s,t)), & \text{if R is monotone} \\ tr_{anti}(\mathrm{R}(f,s,t)), & \text{if R is anti-monotone} \\ tr(\mathrm{R}(f,s,t)), & \text{otherwise} \end{cases}$$

Proposition 5.34 *Let A be a closed aggregate atom of the form $\mathrm{R}(f, \{\mathbf{x} \mid \varphi\}, t)$ and R be a monotone or an anti-monotone aggregate relation. Then $tr_{ma}(A) =_3 tr(A)$.*

Proof. Consider the case where R is a monotone aggregate relation. In this case $tr_{ma}(A) = tr_{mon}(A)$. We first show that $tr_{mon}(A) \subseteq tr(A)$ which implies that

$$\mathcal{H}_{\tilde{I}}(tr_{mon}(A)) \leq_t \mathcal{H}_{\tilde{I}}(tr(A)). \tag{5.6}$$

If $C^{\varphi}_{(S_1,S)} \in tr_{mon}(A)$ then $(\llbracket f \rrbracket(S_1), \llbracket t \rrbracket) \in \mathrm{R}$. Since R is a monotone aggregate relation then $\forall S' \in [S_1, S]$: $(\llbracket f \rrbracket(S'), \llbracket t \rrbracket) \in \mathrm{R}$ thus $C^{\varphi}_{(S_1,S)} \in tr(A)$.

Next, we show the opposite inequality of (5.6). Let $C^{\varphi}_{(S_1,S_2)} \in tr(A)$ be the disjunct with a maximum truth value $\mathcal{H}_{\tilde{I}}(C^{\varphi}_{(S_1,S_2)})$ in the $\leq_t$ order. For any such disjunct, $C^{\varphi}_{(S_1,S)} \in tr_{mon}(A)$. This is because $\forall S' \in [S_1, S_2]$: $\llbracket f \rrbracket(S', \llbracket t \rrbracket) \in \mathrm{R}$

implies $(\llbracket f \rrbracket(S_1), \llbracket t \rrbracket) \in \text{R}$. Moreover, $C^{\varphi}_{(S_1,S)} \subseteq C^{\varphi}_{(S_1,S_2)}$ (considering a conjunction as a set of literals) which implies $\mathcal{H}_{\tilde{I}}(C^{\varphi}_{(S_1,S)}) \geq_t \mathcal{H}_{\tilde{I}}(C^{\varphi}_{(S_1,S_2)})$. Thus $\mathcal{H}_{\tilde{I}}(tr_{mon}(A)) \geq_t \mathcal{H}_{\tilde{I}}(tr(A))$.

The proof for an anti-monotone aggregate relation and the tr_{anti} translation is analogous. ■

In general, both the tr and the tr_{ma} translations of an aggregate atom can be a formula with infinite disjunctions and conjunctions.

Example 5.7 Consider the domain $\mathbb{N}^{\infty} = \mathbb{N} \cup \{\infty\}$ with the usual order $\leq$ on the set of natural numbers extended to $\mathbb{N}^{\infty}$ as $n \leq \infty$ for all $n \in \mathbb{N}^{\infty}$. Let $\text{CARD}^{\infty}\colon D \to \mathbb{N}^{\infty}$ be the aggregate function defined as $\text{CARD}^{\infty}(M) = |M|$ for a finite multiset M and $\text{CARD}^{\infty}(M) = \infty$ otherwise. Note that CARD^{∞} is a monotone aggregate function with respect to $\leq$ and consequently $\text{CARD}^{\infty}_{\geq}$ is a monotone aggregate relation. Also note that $(M, \infty) \in \text{CARD}^{\infty}_{\geq}$ if and only if M is not a finite multiset.

Consider the aggregate atom A of the form $\text{CARD}^{\infty}(\lambda x.\ x, \{x \mid p(x)\}, \infty)$.

This atom is positive, so

$$tr_{ma}(A) = tr_{mon}(A) = \bigvee\{C^{p(x)}_{(S_1,\mathbb{N}^{\infty})} \mid S_1 \text{ is infinite subset of } \mathbb{N}^{\infty}\}.$$

So, every conjunction $C^{p(x)}_{(S_1,\mathbb{N}^{\infty})}$ is infinite and, moreover, $tr_{mon}(A)$ contains an infinite number of such conjunctions. □

If $tr_{ma}(A)$ is a finite formula, we can consider the formula $\min(tr_{ma}(A))$ which contains only minimal conjunctions with respect to set inclusion:

$$\min(tr_{ma})(A) = \bigvee\{C^{\varphi}_{(S_1,S_2)} \in tr_{ma}(A) \mid \neg\exists C^{\varphi}_{(S_1',S_2')} \in tr_{ma}(A)\colon [S_1', S_2'] \subset [S_1, S_2]\}.$$

Proposition 5.35 *For every aggregate atom A, if $tr_{ma}(A)$ is finite then*

$$tr_{ma}(A) =_3 \min(tr_{ma}(A)).$$

It is tempting to use the translation of aggregate atoms defined above as a way of implementing the various semantics of aggregate programs. This can be done as a pre-processing step to an existing answer set programming system. In this respect it is important to study the question about the size complexity of the translation. We measure the size $|F|$ of a formula F in disjunctive normal form by the number of disjuncts in F. Although the total number of atoms in F is a more precise measure, the number of disjuncts is an under-estimate which is sufficient for our analysis. We study the size of the translation as a function of the size of the domain of the set expression of an aggregate atom:

$$size(\{x_1, \ldots, x_n \mid \varphi\}) = |D_{s_1} \times \cdots \times D_{s_n}|$$

where s_i is the type of the variable x_i. We show that in the worst case the size even of the $\min(tr_{mon})$ translation can be exponential.

Example 5.8 Consider the (positive) aggregate atom A of the form

$$\text{CARD}_{\geq}(\lambda x.\ x, \{x \mid p(x)\}, n/2)$$

where the predicate symbol p has type s and $|D_s| = n$ is even. The translation $\min(tr_{mon})$ contains all conjunctions $C^{p(x)}_{(S_1,D_s)}$ where $|S_1| = n/2$. The number of these conjunctions is equal to the number of ways of picking $n/2$ objects out of n. This is given by the central binomial coefficient $C(n) = \binom{n}{n/2}$. Using Stirling's approximation of the factorial function $n! \approx n^n e^{-n}\sqrt{2\pi n}$ we obtain the following approximation of $C(n)$:

$$C(n) \approx \frac{1}{\sqrt{(\pi n)/2}} \cdot 2^n.$$

□

So, the translation is useful mostly for studying theoretical properties of aggregates and not for a practical implementation.

5.4 Implementing the Well-founded Semantics in XSB Prolog

In this section we look at one possibility of implementing the semantics of aggregate programs. Instead of building a complete answer set programming system, we present a simple meta interpreter written in Prolog for solving monotone aggregate atoms. In principle the code does not use any features specific to a particular Prolog system and should be able to run on any Prolog. However, to be able to solve problems with recursive aggregates we use XSB Prolog (Rao et al., 1997) which provides an efficient implementation of the well-founded semantics. The implementation is written in such a way as to exploit automatically the tabling and loop detection facilities of XSB.

The implementation makes the following assumptions on the syntax and instantiation of variables of an aggregate atom $\text{R}(\lambda\mathbf{x}.\ u, \{\mathbf{x} \mid \varphi\}, t)$ when it is selected for execution:

1. $\mathbf{x} \subseteq FV(\varphi)$;
2. the formula φ does not contain quantifiers;
3. the free variables of u and φ when the atom is selected are a subset of $\mathbf{x}$;
4. all answer substitutions of the goal $\leftarrow \varphi(\mathbf{x})$ instantiate all variables $\mathbf{x}$ to ground terms.

The first two conditions are not serious limitations. Either the implementation can be elaborated to deal with them or one can use the fold transformation on the set expression which is a positive formula. We believe that the third condition can also be lifted if we make the implementation to instantiate and backtrack on the free variables in u and φ which are not among $\mathbf{x}$ in the same way as the $setof/3$ predicate in Prolog. The final condition is necessary because we evaluate the term u for all answers of $\leftarrow \varphi(\mathbf{x})$. It can be lifted if we use a constraint based Prolog system.

If all aggregate atoms in an aggregate program satisfy the first three conditions then when an aggregate atom is selected for execution the set of variables $\mathbf{x}$ are equal to the set of free variables of φ. So, we use a simplified syntax for aggregate atoms by dropping the variable prefix for function and set expressions. Thus, an aggregate atom $\text{R}(\lambda\mathbf{x}.\ u, \{\mathbf{x} \mid \varphi\}, t)$ is written as $\text{R}(u, \varphi, t)$ where the term u and the formula φ use the same names for the local variables $\mathbf{x}$ and these variables are not used in any other literal in the same clause. This syntax is very similar to the notation of the $setof/3$ family of "all solution" predicates.

Example 5.9 Reconsider the following rule from the formulation of the Company Controls problem with finite multisets (Example 4.4):

$$\begin{aligned} controls(X,Y) \leftarrow \text{SUM}_{>}(\ & \{\lambda(Z,S).\ S, \\ & (Z,S) \mid cv(X,Z,Y) \wedge owns_stock(Z,Y,S)\} \\ & 0.5). \end{aligned}$$

In the simplified Prolog syntax, the rule is written as follows:

```
controls(X,Y) :- sum_gt(S,(cv(X,Z,Y),owns_stock(Z,Y,S)),0.5).
```

The implementation is correct only if $controls(X,Y)$ is called with X and Y ground. □

The intended use of the implementation below is to compute aggregate atoms with monotone aggregate relations. The idea is to collect enough answers of the condition of the set expression to satisfy the aggregate relation.

```
aggr_rel(Func,Goal,Res,Aggr) :-
      findall_prefix(El, (Goal, El is Func), MultiSet),
      AggrCall =.. [Aggr,MultiSet,Res],
      call(AggrCall).

findall_prefix(Templ,Goal,Bag) :-
      findall(Templ,Goal,L1),
      append(Bag,_,L1).
```

To illustrate how the predicate *aggr_rel* is called consider the following implementation of the *sum_gt* aggregate from Example 5.9:

```
sum_gt(Func,Goal,Res) :- aggr_rel(Func,Goal,Res,sum_gt_rel).
sum_gt_rel(L,Res) :- sumlist(L,S), S > Res.
```

The clause for *sum_gt_rel* implements the aggregate relation $\text{SUM}_{>}$ and the predicate $sumlist(L, S)$ is true if L is a list of integers whose sum is S.

The call to $append(Bag, _, L1)$ in the implementation of the $findall_prefix/3$ predicate above is used to backtrack over all initial segments Bag of the list $L1$ of all answers of *Goal*. To be able to solve problems with recursion over aggregates the predicate $findall_prefix/3$ must be implemented (either by the user or directly by the Prolog system) such that it returns after every new answer of *Goal* and does not first compute all answers and then returns all prefixes, as in the implementation above. One way to implement this behavior is the following:

```
findall_prefix(Templ,Goal,Bag) :-
      inc_var0(ctr,Reg),        % create a unique counter name

      (Bag = [];
       call(Goal),
       get_var(Reg,OldBag),
       Bag = [Templ|OldBag]),

      set_var(Reg,Bag).
```

It uses non-backtrackable global registers. In XSB Prolog they are provided by the `storage` module. A call to $inc_var0(Name, Value)$ increments the value of the register $Name$ and returns the result in $Value$. If $Name$ does not exists then it is first initialized to 0. The actual call to $inc_var0(ctr, Reg)$ above is used to generate a unique counter name Reg for every call to the predicate $findall_prefix/3$. This counter is used to hold the current list of answers to the goal. The first time the predicate $findall_prefix/3$ is called the register Reg is set to the empty list which is also returned as a result. On subsequent backtracking, the second disjunct calls the goal *Goal* and the answer is appended to the current list which is again returned.

Example 5.10 Reconsider the Party Invitation I problem from Example 3.7. The input program is as follows:

```
:- table accept/1.

accept(X) :-
      thr(X,K),
      count_ge((friend(X,Y),accept(Y)),K).

count_ge(Goal,Res) :- aggr_rel(1,Goal,Res,count_ge_rel).
count_ge_rel(L,R) :- length(L,N), N >= R.
```

In this program, the predicate $count_ge(Goal, Res)$ implements the aggregate atom $\text{CARD}_{\geq}(1, Goal, Res)$ while $count_ge_rel/2$ implements the aggregate relation $\text{CARD}_{\geq}$. The call to $thr(X, K)$ will instantiate the variable X and when the aggregate atom is selected only Y will be free.

Consider the following input database:

```
friend(a,b).      thr(a,1).
friend(b,a).      thr(b,1).
```

Intuitively, a will accept if and only if b does and vice versa. Using the transformation of aggregate programs to normal logic programs, the program with the input is equivalent to:

```
accept(a) :- accept(b).
accept(b) :- accept(a).
```

Thus, in the well-founded model both $accept(a)$ and $accept(b)$ are false. Indeed, running the query $?- accept(a)$ in XSB returns `no`. □

Our next task is to establish correctness of the $aggr_rel$ implementation for positive aggregate atoms in XSB. We do this in several steps. In every step, we replace the definition of the $aggr_rel$ in the previous step with a new program which has the same procedural behavior as the previous program. We do not give full proofs of the correctness of some of the transformations because this is beyond the scope of this work.

Consider an atom A of the form $aggr_rel(F, G, Res, Aggr)$. Let $\theta = \theta_1, \theta_2, \ldots$ be the sequence of all answer substitutions θ_i of the query $\leftarrow G$ such that θ_1 is the first answer substitution, θ_2 is the second one, and so on. Let P_A^{is} be the program consisting of the set of rules

$$aggr_rel(F, G, d, Aggr) \leftarrow G\theta_1, \ldots, G\theta_n.$$

for every initial segment $\theta_1, \ldots, \theta_n$ of θ and $d \in D$ which satisfies:

$$(\{[\![F\theta_1]\!], \ldots, [\![F\theta_n]\!]\}, d) \in Aggr.$$

We leave the following proposition without proof.

Proposition 5.36 *Let P be an aggregate program using only aggregates interpreted by monotone aggregate relations. Let P' be the program obtained from P by removing the definition of aggr_rel and adding P_A^{is} for every atom A with the aggr_rel predicate symbol. Then XSB will compute the same answers for P and P'.*

Now, consider the program P_A^{mon} obtained by defining $aggr_rel$ by using the translation tr_{mon}. It consists of the set of rules

$$aggr_rel(F, G, d, Aggr) \leftarrow G\delta_1, \ldots, G\delta_m.$$

for every $d \in D$ and every sequence $\delta_1, \ldots, \delta_m$ of substitutions of the variables of G such that $(\{[\![F\delta_1]\!], \ldots, [\![F\delta_m]\!]\}, d) \in Aggr$. The only difference with the program P_A^{is} is that instead of considering only initial segments of the set of answer substitutions of G, we consider all possible sets of substitutions of G which satisfy the aggregate relation. Clearly $P_A^{is} \subseteq P_A^{mon}$.

Proposition 5.37 *Let P be an aggregate program using only aggregates interpreted by monotone aggregate relations. Let P' be the program obtained from P by removing the definition of aggr_rel and adding P_A^{is} for every atom A with the aggr_rel predicate symbol and P'' be the program obtained in the same way but using P_A^{mon}. Then XSB will compute the same answers for P' and P''.*

Proof. (Sketch) Because $P_A^{is} \subseteq P_A^{mon}$ then answers computed for the program P' can be computed for the program P''. In the opposite direction, suppose that during the proof the following rule from P_A^{mon} is used:

$$aggr_rel(F, G, d, Aggr) : -G\delta_1, \ldots, G\delta_m.$$

This means that all of $\delta = \delta_1, \ldots, \delta_m$ are answer substitutions of G. Let $\theta = \theta_1, \ldots, \theta_n$ be the smallest initial sequence of answer substitutions of G which contains all of δ_i. Because the aggregate relation of A is monotone then P_A^{is} contains a rule

$$aggr_rel(F, G, d, Aggr) \leftarrow G\theta_1, \ldots, G\theta_n.$$

Moreover, the proof procedure can use this rule instead of the original rule because to generate the answers δ the proof procedure has already generated all answers θ. ■

5.4.1 Incremental Aggregate Functions

In the implementation of *aggr_rel*, every time a new value is added to the list representing the multiset the aggregate function is recomputed for the entire list. However, most of the standard aggregate functions are incremental in the sense that $\textsc{f}(M \uplus \{x\})$ can be computed from the value of $\textsc{f}(M)$ and x. For example $\textsc{sum}(M \uplus \{x\}) = \textsc{sum}(M) + x$. So, instead of keeping the current set of answers we can keep the value of the aggregate function on the current multiset and update it every time a new element is added to the list. We first formalize the notion of incremental aggregates.

Definition 5.6 (Incremental Aggregate Function) *An* incremental aggregate function *is an aggregate function $\textsc{f}\colon \mathcal{M}(D_1) \to D_2$ such that there exists an* identity *element $e \in D_2$ and a binary* update function *$op\colon D_2 \times D_1 \to D_2$ such that $\textsc{f}(\emptyset) = e$ and $\textsc{f}(M \uplus \{x\}) = op(\textsc{f}(M), x)$ for every multiset M on D_1 and $x \in D_2$.*

The following table gives the identity element and the update function of common aggregate functions.

aggregate	identity	update $op(x,y) =$
CARD	0	$x+1$
SUM	0	$x+y$
PROD	1	$x*y$
MIN	$+\infty$	$min(x,y)$
MAX	$-\infty$	$max(x,y)$

The implementation of incremental aggregate functions is given below. Instead of a name of an aggregate relation, the last argument is a pair (e, op) of identity element e and the name of a ternary predicate op which implements the update function. The program is similar to the implementation of *aggr_rel* with two differences: (i) the call to $findall_prefix$ is unfolded and (ii) we keep in the register the value of the aggregate function on the current multiset instead of the multiset itself.

```
inc_aggr_func(Func,Goal,Res,(Id,Op)) :-
      inc_var0(ctr,Reg),

      (Res=Id;

       call(Goal),
       get_var(Reg,Old),
       El is Func,
       Expr =.. [Op,Old,El,Res],
       call(Expr)),

       set_var(Reg,Res).
```

Although the implementation is limited only to monotone aggregate atoms, by using the transformations from Section 5.3 many aggregate atoms can be transformed to a combination of aggregate atoms with monotone aggregate relations.

5.5 Example

In this section we combine most of the results developed in this and the previous chapters. We take the one-line program of the Party Invitation II problem (Example 3.8). Although, we argued the the completion semantics is the right semantics for this example, it illustrates many of the techniques developed in this chapter. Using a series of program transformations we obtain a Prolog program

which can be executed by our implementation in XSB Prolog. It has a simple and elegant formulation and at the same time a solid theoretical foundation.

Example 5.11 Consider the program of the Party Invitation II problem from Example 3.8. It consists of the single rule

$$\begin{aligned} accept(X) \leftarrow & \exists K.\ thr(X,K) \wedge \\ & \textsc{sum}_{\geq}(\lambda(Y,C).\ C, \{(Y,C) \mid comp(X,Y,C) \wedge accept(Y)\}, K). \end{aligned}$$

First, we use the transformation of $\textsc{sum}_{\geq}$ on general numbers from Section 5.3.5:

$$\begin{aligned} \textsc{sum}_{\geq}(\lambda\mathbf{x}.\ e, \{\mathbf{x} \mid \varphi\}, t) =_3 & \\ & \exists z.\ \textsc{sum}_{\subseteq}(\lambda\mathbf{x}.\ e, \{\mathbf{x} \mid \varphi^+\}, z) \wedge \neg\textsc{sum}_{<}(\lambda\mathbf{x}.\ e, \{\mathbf{x} \mid \varphi^-\}, t - z). \end{aligned}$$

Applying it to the aggregate atom in the program above we obtain:

$$\begin{aligned} & accept(X) \leftarrow \\ & \quad \exists K.\ thr(X,K) \wedge \\ & \quad \textsc{sum}_{\subseteq}(\lambda(Y,C).\ C, \{(Y,C) \mid comp(X,Y,C) \wedge accept(Y) \wedge C > 0\}, Pos) \wedge \\ & \quad \neg\textsc{sum}_{<}(\lambda(Y,C).\ C, \{(Y,C) \mid comp(X,Y,C) \wedge accept(Y) \wedge C < 0\}, K - Pos). \end{aligned}$$

The $\textsc{sum}_{\subseteq}$ aggregate atom does not depend on K and can be taken outside the existential quantifier:

$$\begin{aligned} & accept(X) \leftarrow \\ & \quad \textsc{sum}_{\subseteq}(\lambda(Y,C).\ C, \{(Y,C) \mid comp(X,Y,C) \wedge accept(Y) \wedge C > 0\}, Pos) \wedge \\ & \quad \exists K.\ thr(X,K) \wedge \\ & \quad \neg\textsc{sum}_{<}(\lambda(Y,C).\ C, \{(Y,C) \mid comp(X,Y,C) \wedge accept(Y) \wedge C < 0\}, K - Pos). \end{aligned}$$

Because $thr(X,K)$ encodes a total function from X to K it can be "pushed" inside the negation as:

$$\begin{aligned} & accept(X) \leftarrow \\ & \quad \textsc{sum}_{\subseteq}(\lambda(Y,C).\ C, \{(Y,C) \mid comp(X,Y,C) \wedge accept(Y) \wedge C > 0\}, Pos) \wedge \\ & \quad \neg(\exists K.\ thr(X,K) \wedge \\ & \quad \textsc{sum}_{<}(\lambda(Y,C).\ C, \{(Y,C) \mid comp(X,Y,C) \wedge accept(Y) \wedge C < 0\}, K - Pos)). \end{aligned}$$

The formula $\exists K.\ thr(X,K) \wedge \textsc{sum}_{<}(\ldots)$ is positive, so, by Proposition 5.25, it can

be folded obtaining the following normal aggregate program:

$$\begin{aligned}
&accept(X) \leftarrow \\
&\quad \text{SUM}_{\subseteq}(\lambda(Y,C).\ C, \{(Y,C) \mid comp(X,Y,C), accept(Y), C > 0\}, Pos), \\
&\quad not(too_bad(X, Pos)).
\end{aligned}$$

$$\begin{aligned}
&too_bad(X, Pos) \leftarrow \\
&\quad thr(X,K), \\
&\quad \text{SUM}_{<}(\lambda(Y,C).\ C, \{(Y,C) \mid comp(X,Y,C), accept(Y), C < 0\}, K - Pos).
\end{aligned}$$

Next, the derived aggregate relation $\text{SUM}_{<}$ can be replaced by a combination of $\text{SUM}_{\subseteq}$ and $<$ as in Proposition 5.30. So, we obtain the following program:

$$\begin{aligned}
&accept(X) \leftarrow \\
&\quad \text{SUM}_{\subseteq}(\lambda(Y,C).\ C, \{(Y,C) \mid comp(X,Y,C), accept(Y), C > 0\}, Pos), \\
&\quad not(too_bad(X, Pos)).
\end{aligned}$$

$$\begin{aligned}
&too_bad(X, Pos) \leftarrow \\
&\quad thr(X,K), \\
&\quad \text{SUM}_{\subseteq}(\lambda(Y,C).\ C, \{(Y,C) \mid comp(X,Y,C), accept(Y), C < 0\}, Neg), \\
&\quad Neg < K - Pos.
\end{aligned}$$

Finally, we give the program in XSB Prolog syntax. The $\text{SUM}_{\subseteq}$ aggregates are replaced with calls to $inc_aggr_rel(\ldots, (0,+))$. Moving the tests $C > 0$ and $C < 0$ before the call to $accept(Y)$ in the conditions of the set expressions of the two aggregate atoms is done for efficiency reasons. The program runs correctly if the tests are the last calls, as in the formulation above.

```
:- table accept/1.
accept(X) :-
      sum(C,(comp(X,Y,C),C > 0,accept(Y)),Pos),
      tnot(too_bad(X,Pos)).

:- table too_bad/2.
too_bad(X,Pos) :-
      thr(X,K),
      sum(C,(comp(X,Y,C),C < 0,accept(Y)),Neg),
      Pos + Neg < K.

sum(Func,Goal,Res) :- inc_aggr_func(Func,Goal,Res,(0,plus)).
plus(X,Y,Z) :- Z is X + Y.
```

The name `tnot` comes for "tabled" `not` and is an implementation of the negation operator under the well-founded semantics by XSB Prolog using the tabling mechanism.

Consider the following (awkward) input:

```
comp(a,b,1).        thr(a,1).
comp(b,a,-1).       thr(b,0).
```

Intuitively, a will accept if and only if b accepts and b will accept if and only if a does not. Obviously, this problem does not have a solution. Running the query $?- accept(a)$ under XSB results in the following residual program:

```
accept(a) :- accept(b).
accept(b) :- tnot(too_bad(b,0)).
too_bad(b,0) :- accept(a).
```

So, both $accept(a)$ and $accept(b)$ are undefined in the well-founded model. □

5.6 Conclusions

In this chapter we studied the three-valued stable semantics of aggregate programs obtained by interpreting aggregate symbols with the ultimate approximating aggregate. As higher precision usually comes at a price, one of the important questions is what is the complexity of this semantics. The full complexity analysis of the semantics is done only in Chapter 6. In this chapter we showed that the complexity of the ultimate approximating aggregates of most standard aggregate functions and relations does not increase. So, for all those aggregates, the complexity of the semantics is the same as the complexity of the corresponding semantics of non-aggregate programs. There is an increase in complexity only for the aggregates SUM and PROD. However, we have not found any example with recursive aggregation which uses these aggregates. One possible approximating aggregate of SUM (and PROD) which is polynomial time computable can be defined as $A_{\text{SUM}} = U_{\text{SUM}_{\geq}} \cap_t U_{\text{SUM}_{\leq}}$. As Example 5.1 demonstrates A_{SUM} is a strictly less precise approximating aggregate relation of SUM than U_{SUM}. This rises an interesting open question for giving a uniform (possibly the most precise) definition of an approximating aggregate relation for all aggregate relations which is polynomial time computable. Note that the A_{SUM} depends on a partial order relation $\leq$ on the domain of the aggregate.

Although we have not developed a full featured implementation of any of our semantics, we provide several results in this direction. For example, the algorithms for computing the ultimate approximating aggregate relations in Section 5.1.2 can be used in an answer set programming system which generates models of a ground logic program. Also, the idea behind the implementation of aggregate atoms with monotone aggregate relations in Section 5.4 can be used to implement

aggregates in an abductive answer set programming system based on a top-down proof procedure like ACLP (Kakas et al., 2000) or A-System (Kakas et al., 2001). A third approach is outlined in Section 6.3.1 where the ultimate approximating aggregates are computed using constraint programming techniques.

Chapter 6

Propositional Aggregate Programs

In this chapter we restrict the syntax of aggregate programs to a propositional language. The main difference for aggregate atoms is that instead of defining a multiset using a combination of a first-order set expression of the form $\{\mathbf{x} \mid \varphi(\mathbf{x})\}$ and a lambda function $\lambda\mathbf{x}.\ u$ we use a propositional set expression of the form $\{L_1 = w_1, \ldots, L_n = w_n\}$ where L_i are literals and w_i are weights associated with the literals. Obviously, such language is not suitable for modeling directly real-world problems. However, it provides a simple framework for theoretical studies of aggregate programs. The three-valued stable semantics of aggregate programs from Section 4.2 and the translation of aggregate programs to normal logic programs from Section 5.3.6 were originally developed and presented for this propositional language (Pelov et al., 2004; Pelov et al., 2003). The results were subsequently lifted to the first-order language used in this work.

There are two results for which using such propositional language is more convenient than a first-order language. The first one is complexity analysis of the different semantics. The second one is a comparison with the extensions of logic programs with weight constraints (Simons et al., 2002), monotone cardinality atoms (Marek et al., 2004), and set constraints (Marek and Remmel, 2004) which are presented for a propositional language.

The relationship with the SMODELS system (Section 6.3.2) was presented in (Pelov et al., 2004). None of the results on the complexity have been published.

6.1 Preliminaries

Aggregate programs are built over a set of propositional atoms At, a domain D, and a set $\mathcal{R}$ of aggregate relations over D. A *literal* is an atom a (*positive literal*)

or the negation of an atom *not a* (*negative literal*). The *complement* $\overline{L}$ of a literal L is defined as $\overline{L} = not\, a$ if $L = a$ and $\overline{L} = a$ if $L = not\, a$.

A *weight literal* is an expression $L = w$ where L is a literal and $w \in D$ is the *weight* associated with L. A *propositional set expression* is a finite set of weight literals. Syntactically, we denote a set expression as $\{L_1 = w_1, \ldots, L_n = w_n\}$. The set of all set expression is denoted with $\mathcal{SE}$. It forms a lattice under the subset order. However, since we consider only finite set expressions, this lattice is not complete (see Example 2.3). For a set expression s, $w(s)$ denotes the multiset consisting of the weights of all weight literals in s. An *aggregate atom* has the form $\mathrm{R}(s, d)$ where R is an aggregate relation, s is a set expression, and $d \in D$.

A *rule* has the form $A \leftarrow B_1 \wedge \cdots \wedge B_n$ where A is an atom called the *head* of the rule and every B_i is a a literal or an aggregate atom. The set $B = \{B_1, \ldots, B_n\}$ is called the *body* of the rule. It can be partitioned in three sets namely, positive literals $pos(B)$, negative literals $neg(B)$, and aggregate atoms $aggr(B)$. A *propositional aggregate program* is a (possibly infinite) set of rules. A *normal propositional logic program* is a set of rules which does not contain aggregate atoms. For the rest of this chapter we drop the word "propositional" from the names of programs.

In the following example we give a formulation of the Party Invitation II problem (Example 3.8) as a propositional aggregate program.

Example 6.1 (Party Invitation II, (Pelov et al., 2004)) The input of the problem is given by a set of people S and a set of compatibility measures $\langle w_{pq} \rangle_{p,q \in S}$. For every person p we have a rule

$$a_p \leftarrow \mathrm{SUM}_{\geq}(\{a_{q_1} = w_{pq_1}, \ldots, a_{q_m} = w_{pq_m}\}, k_p).$$

where $q_1, \ldots, q_m$ are all persons with whom p has a non-zero compatibility measure, i.e. $w_{pq_i} \neq 0$ for all $i = 1, \ldots, m$. □

An *interpretation* is a function $I : At \to \mathcal{TWO}$. As before an interpretation is also viewed as the subset of At containing the atoms which are mapped to true. The set of all interpretation is denoted with $\mathcal{I} = \wp(At)$. For a set expression s and an interpretation I we denote with s^I the subset expression of s which contains only the literals which are true in I that is $s^I = \{L = w \in s \mid I \models L\}$. The value of a set expression s for an interpretation I is defined as $[\![s]\!]_I = w(s^I)$ that is the multiset of weights of all literals in s which are true in I. Satisfiability of an aggregate atom $\mathrm{R}(s, d)$ is defined in the same way as the first-order case: $I \models \mathrm{R}(s, d)$ if and only if $([\![s]\!]_I, d) \in \mathrm{R}$.

Example 6.2 An example of an aggregate atom is $\mathrm{SUM}_{\geq}(\{a = 1, b = 2\}, 2)$. It is true in the interpretation $I = \{b\}$ because $[\![\{a = 1, b = 2\}]\!]_I = \{2\}$ and $(\{2\}, 2) \in \mathrm{SUM}_{\geq}$. On the other hand $\{a\} \not\models \mathrm{SUM}_{\geq}(\{a = 1, b = 2\}, 2)$. □

6.1.1 Three-Valued Stable Semantics

We now specialize the three-valued stable semantics of aggregate programs from Section 4.2 to propositional aggregate programs. The valuation function for propositional set expressions for three-valued interpretations is defined in a very similar way as for first-order set expressions.

Definition 6.1 *The value* $[\![s]\!]_{(I_1,I_2)}$ *of the set expression* s *for the three-valued interpretation* (I_1, I_2) *is a three-valued multiset defined as*

$$[\![s]\!]_{(I_1,I_2)} = (w(s^{I_1}), w(s^{I_2})).$$

The definition of three-valued truth function $\mathcal{H}_{\tilde{I}}$ is the same as for the first-order language:

Definition 6.2 *The* three-valued truth function *for a propositional language with aggregate atoms is defined as follows:*

$$\begin{array}{lll} \mathcal{H}^{aggr}_{\tilde{I}}(a) & = \tilde{I}(a) & \text{for an atom } a \in At \\ \mathcal{H}^{aggr}_{\tilde{I}}(\neg\varphi) & = \neg\mathcal{H}_{\tilde{I}}(\varphi) & \\ \mathcal{H}^{aggr}_{\tilde{I}}(\varphi \wedge \psi) & = \mathcal{H}_{\tilde{I}}(\varphi) \wedge_t \mathcal{H}_{\tilde{I}}(\psi) & \\ \mathcal{H}^{aggr}_{\tilde{I}}(\mathrm{R}(s,d)) & = A_{\mathrm{R}}([\![s]\!]_{\tilde{I}}), d) & \end{array}$$

where $A_{\mathrm{R}} \colon \mathcal{M}(D)^c \times D \to \mathcal{THREE}$ *is an approximating aggregate relation of* $\mathrm{R} \colon \mathcal{M}(D) \times D \to \mathcal{TWO}$.

Finally, the three-valued immediate consequence operator $\Phi^{aggr}_P \colon \mathcal{I}^c \to \mathcal{I}^c$ is defined as:

$$\Phi^{aggr}_P(\tilde{I})(A) = \bigvee_t \{\mathcal{H}^{aggr}_{\tilde{I}}(B) \mid A \leftarrow B \in P\}.$$

Three-valued stable models of an aggregate program P are defined as the consistent stable fixpoints of Φ^{aggr}_P.

6.2 Complexity of the Semantics

From complexity point of view there is a correspondence between aggregate relations and oracles. So, the complexity classes of the different semantics are parametrized by the complexity of deciding two or three-valued aggregate relations. For a set $\mathcal{R}$ of aggregate relations let $\tilde{\mathcal{R}}$ be the set containing some approximating aggregate relation A_{R} for every $\mathrm{R} \in \mathcal{R}$. Moreover, let $\tilde{\mathcal{R}}^1$ and $\tilde{\mathcal{R}}^2$ be the sets containing the first and second components of the approximating aggregate relations in $\tilde{\mathcal{R}}$:

$$\tilde{\mathcal{R}}^1 = \{A^1_{\mathrm{R}} \mid A_{\mathrm{R}} \in \tilde{\mathcal{R}}\}$$
$$\tilde{\mathcal{R}}^2 = \{A^2_{\mathrm{R}} \mid A_{\mathrm{R}} \in \tilde{\mathcal{R}}\}$$

When talking about the complexity of deciding $\tilde{\mathcal{R}}$ we refer to the complexity of deciding the relations in $\tilde{\mathcal{R}}^1 \cup \tilde{\mathcal{R}}^2$.

We study the following decision problems.

LEAST($\mathcal{R}$)
Instance: An atom a and a definite aggregate program P.
Question: Is a true in the least model of P?

STD($\mathcal{R}$)
Instance: An atom a and a weakly stratified aggregate program P.
Question: Is a true in the standard model of P?

The well-founded semantics of an aggregate program P based on the Φ_P^{aggr} operator is parametrized by a set $\tilde{\mathcal{R}}$ of approximating aggregate relations.

WF($\tilde{\mathcal{R}}$)
Instance: An atom a and an aggregate program P.
Question: Is a true in the well-founded model of P?

ST($\tilde{\mathcal{R}}^1$)
Instance: An aggregate program P.
Question: Does P has a two-valued stable model?

Note that for the stable semantics the complexity depends only on the complexity of the first component of the approximating aggregate relations. If we allow negation of aggregate atoms this will not be the case. Also note that for some aggregate functions, for example SUM and PROD, the complexity of computing the first component is polynomial while that of the second component it is NP-complete, so this distinction is crucial to obtain more precise complexity analysis.

The ultimate Kripke-Kleene, well-founded, and exact stable semantics depend only on two valued aggregate relations.

ULT-WF($\mathcal{R}$)
Instance: An atom a and an aggregate program P.
Question: Is a true in the ultimate well-founded model of P?

ULT-ST($\mathcal{R}$)
Instance: An aggregate program P.
Question: Does P has a two-valued ultimate stable model?

Table 6.1 gives a summary of the results for membership and Section 6.2.1 contains the proofs. The first column gives the name of the problem. The last column gives the complexity class for this decision problem under the assumption that all problems in the parameter are in the class Δ_k^p (the $k-1$ level in the polynomial hierarchy). In most cases the complexity of the parameters will be in a Σ_k^p or a Π_k^p class. However, the complexity of the semantics depend only on the

level of the polynomial hierarchy which contains the (approximating) aggregate relations and no lower bounds can be obtained for different classes within a level in the hierarchy.

Problem	Complexity of the parameter	Complexity of the problem
LEAST($\mathcal{R}$) STD($\mathcal{R}$)	Δ_k^p	Δ_k^p
WF($\tilde{\mathcal{R}}$)	Δ_k^p	Δ_k^p
ST($\tilde{\mathcal{R}}^1$)	Δ_k^p	Σ_k^p
ULT-WF($\mathcal{R}$)	Δ_k^p	Δ_{k+1}^p
ULT-ST($\mathcal{R}$)	Δ_k^p	Σ_{k+1}^p

Table 6.1: Complexity of the Semantics

For the least fixpoint semantics of definite aggregate programs, the standard model semantics of weakly stratified aggregate programs the complexity is determined only by the complexity of computing the aggregate relations. For the well-founded semantics the complexity is determined by the complexity of computing the first and second components of the approximating aggregate relations. For the stable semantics there is an expected increase in the complexity which is caused by the non-determinism of the semantics. For the ultimate well-founded model there is an increase in complexity by one level and for the ultimate stable semantics by two levels in the polynomial hierarchy.

For all approximating aggregate relations which we study, the complexity of deciding R or A_{R} is either polynomial (the class Δ_1^p) or it is in the class Δ_2^p. Table 6.2 is obtained by instantiating the results in Table 6.1 for these two classes, i.e. taking $k = 1$ and $k = 2$. For comparison, the second column includes complexity results for logic programs without aggregates which can be found in (Dantsin et al., 2001) and for the ultimate semantics in (Denecker et al., 2004).

Problem	no aggr.	$X \subseteq$ P	$X \subseteq \Delta_2^p$
LEAST(X) STD(X)	P	P	Δ_2^p
WF(X)	P	P	Δ_2^p
ST(X)	NP	NP	Σ_2^p
ULT-WF(X)	Δ_2^p	Δ_2^p	Δ_3^p
ULT-ST(X)	Σ_2^p	Σ_2^p	Σ_3^p

Table 6.2: Complexity of the Semantics of Aggregate Programs

The next proposition is one of the main conclusions of our complexity analysis.

It is applicable for all standard aggregate relations MIN, MAX, CARD, SUM, PROD, and AVG which are polynomial time computable in the size of the input multiset.

Proposition 6.1 *If all aggregate relations $\mathcal{R}$ in the program are polynomial time computable then the complexity of computing the least fixpoint of definite aggregate programs, the standard model of weakly stratified aggregate programs, and the ultimate well-founded and ultimate stable model is the same as for programs without aggregates.*

The complexity of the the well-founded and stable semantics depend on the complexity of computing the first and second component of the approximating aggregate relations. Proposition 5.22 tells us for which aggregate relations computing the first and second component of the ultimate approximating aggregates are polynomial and so we stay in the same complexity classes as for programs without aggregates.

Proposition 6.2 *If an aggregate program uses only some the following aggregate relations* CARD, MIN, MAX, INF, SUP, $\text{SUM}^{\mathbb{Q}}_{\leq}$, $\text{SUM}^{\mathbb{Q}}_{\geq}$, $\text{PROD}^{\mathbb{Q}}_{\leq}$, $\text{PROD}^{\mathbb{Q}}_{\geq}$ *then the complexity of computing the well-founded (*WF*) and stable semantics (*ST*) based on the ultimate approximating aggregate is the same as for programs without aggregates (P and NP respectively).*

For programs without aggregates, the complexity of the ultimate well-founded and ultimate stable semantics is one level higher in the polynomial hierarchy than the complexity of the standard well-founded and stable semantics. However, in the presence of aggregate atoms the complexity of the two semantics may become equal. This is because the former semantics depends on the complexity of the two-valued aggregate relations while the latter depends on the complexity of the ultimate approximating aggregate relations which may be higher. This is the case, for example, for the SUM and PROD aggregate relations. They are polynomial time computable, however U^2_{SUM} and U^2_{PROD} are NP-complete problems (Proposition 5.24). So, the complexity of both the ULT-WF($\{$SUM$\}$) problem and WF($\{U_{\text{SUM}}\}$) problem is Δ^p_2. In such cases there is no computational advantage of using the weaker three-valued stable semantics.

Our next task is to look at lower bounds of complexity of semantics for which the complexity of computing the (approximating) aggregate relations is in the class Δ^p_2. Table 6.3 summarizes the results. The first column gives the names of the problems. The second column gives the smallest complexity class containing the decision problems of the (approximating) aggregate which is used as a parameter. For example the complexity of computing $U^1_{\text{SUM}_{\neq}}$ is co-NP (follows from Proposition 5.24) and the complexity of computing $U^2_{\text{SUM}_{\neq}}$ is P (follows from Proposition 5.23). So, the smallest complexity class containing $U^1_{\text{SUM}_{\neq}}$ and $U^2_{\text{SUM}_{\neq}}$ is co-NP. In brackets we also give the smallest Δ^P_k class containing this class (which is Δ^p_2 in this case). The third column gives the class for which the problem in the

Problem(s)	Complexity of the parameter	Hardness (lower bound)	Membership (upper bound)
$\text{LEAST}(\{\text{SUM}_{\subseteq}\})$	NP ($\subseteq \Delta_2^p$)	NP	Δ_2^p
$\text{LEAST}(\{\text{PROD}_{\subseteq}\})$	NP ($\subseteq \Delta_2^p$)	NP	Δ_2^p
$\text{WF}(\{U_{\text{SUM}_{\neq}}\})$	co-NP ($\subseteq \Delta_2^p$)	co-NP	Δ_2^p
$\text{WF}(\{U_{\text{PROD}_{\neq}}\})$			
$\text{WF}(\{U_{\text{SUM}_{\subseteq}}\})$	NP ($\subseteq \Delta_2^p$)	co-NP	Δ_2^p
$\text{WF}(\{U_{\text{PROD}_{\subseteq}}\})$		and NP	
$\text{ST}(\{U^1_{\text{SUM}_{\neq}}\})$	co-NP ($\subseteq \Delta_2^p$)	Σ_2^p	Σ_2^p

Table 6.3: Lower bounds for complexity of the semantics

first column is hard. The proofs of these results are given in Section 6.2.2. In the last column we give the complexity class to which the problem belongs (obtained from Table 6.1).

The results in Table 6.3 should be interpreted as: the complexity of the problem in the first column is between the class in the third column and the class in the fourth (last) column. If these two classes are equal then the problem is complete for this class. This is the case only with the $\text{ST}(\{U^1_{\text{SUM}_{\neq}}\})$ problem which is complete for the class Σ_2^p. For the subset aggregate relations $\text{SUM}_{\subseteq}$ and $\text{PROD}_{\subseteq}$ the problems of "computing" the well-founded model are both NP-hard and co-NP-hard but we were unable to obtain completeness for the class Δ_2^p.

6.2.1 Proofs of Membership

The rest of the section contains proofs of the results in Table 6.1.

Proposition 6.3 *If all problems in the set $\mathcal{R}$ are in the class Δ_k^p then the problems $\text{LEAST}(\mathcal{R})$ and $\text{STD}(\mathcal{R})$ are also in the class Δ_k^p.*

Proof. (Sketch) Computing the least or the standard model of P requires at most $|At|$ applications of the T_P^{aggr} operator. Each application of the operator requires at most $|P|$ calls to the oracle machine for deciding an aggregate relation $\text{R} \in \mathcal{R}$. So, the problem is in the class $P^{\Delta_k^p} = \Delta_k^p$. For the standard model of a weakly stratified aggregate program we can compute a stratification in polynomial time (assuming that aggregate relations which are monotone or anti-monotone are explicitly labeled as such). ■

Proposition 6.4 *If all problems $\tilde{\mathcal{R}}$ are in the class Δ_k^p then the problem $\text{WF}(\tilde{\mathcal{R}})$ is also in the class Δ_k^p.*

Proof. (Sketch) Computing the well-founded fixpoint of P requires at most $|At|$ applications of the operator Φ_P^{aggr}. Each application of the operator requires at

most $|P|$ calls to the oracle machine for deciding A^1_{R} or A^2_{R}. So, the problem is in the class $P^{\Delta^p_k} = \Delta^p_k$. ∎

Proposition 6.5 *If all problems in the set $\tilde{\mathcal{R}}^1$ are in the class Δ^p_k then the problem* $\textsc{st}(\tilde{\mathcal{R}})$ *is in the class Σ^p_k.*

Proof. (Sketch) First guess an interpretation I and then verify if it is a stable model. The guessing step is done by a non-deterministic Turing machine and the verification is done with polynomially many calls to the oracle deciding the approximating aggregate relation (see proof of the previous proposition). So, the problem is in the class

$$\mathrm{NP}^{\Delta^p_k} = \mathrm{NP}^{(P^{\Sigma^p_{k-1}})} = \mathrm{NP}^{\Sigma^p_{k-1}} = \Sigma^p_k. \qquad \blacksquare$$

Proposition 6.6 *If all problems $\mathcal{R}$ are in the class Δ^p_k then the problem* $\textsc{ult-wf}(\mathcal{R})$ *is in the class Δ^p_{k+1}.*

Proof. (Sketch) Let p be an atom and P be an aggregate program. Our first task is to show the complexity of the one application of the U^{aggr}_P operator. We show that the complexity of deciding

$$p \notin (U^{aggr}_P(I_1, I_2))^1 \tag{6.1}$$

for some three-valued interpretation (I_1, I_2) is in the class Σ^p_k. Recall the definition of the U^{aggr}_P operator:

$$U^{aggr}_P(I_1, I_2) = \Big(\bigcap_{I \in [I_1, I_2]} T^{aggr}_P(I), \bigcup_{I \in [I_1, I_2]} T^{aggr}_P(I) \Big).$$

So, $p \in (U^{aggr}_P(I_1, I_2))^1$ if and only if $\forall I \in [I_1, I_2] \colon p \in T^{aggr}_P(I)$. Consequently, $p \notin (U^{aggr}_P(I_1, I_2))^1$ if and only if

$$\exists I \in [I_1, I_2] \colon p \notin T^{aggr}_P(I). \tag{6.2}$$

This problem can be decided by a non-deterministic Turing machine which guesses I and then checks that $p \notin T^{aggr}_P(I)$. Finally, one application of the T^{aggr}_P operator requires at most $|P|$ calls to the oracle machine for deciding an aggregate relation $\mathrm{R} \in \mathcal{R}$. So, the complexity of deciding $p \notin T^{aggr}_P(I)$ is Δ^p_k and the complexity of (6.2) and (6.1) is

$$\mathrm{NP}^{\Delta^p_k} = \mathrm{NP}^{(P^{\Sigma^p_{k-1}})} = \mathrm{NP}^{\Sigma^p_{k-1}} = \Sigma^p_k.$$

With a similar argument we can show that the complexity of deciding

$$p \in (U^{aggr}_P(I_1, I_2))^2$$

is in the same class.

Computing the well-founded model of P requires a polynomial number of applications of the U_P^{aggr} (in the number of atoms) that is polynomial number of calls to an oracle in Σ_k^p. So, the problem ULT-WF($\mathcal{R}$) is in the class $\mathrm{P}^{\Sigma_k^p} = \Delta_{k+1}^p$.■

Proposition 6.7 *If all problems $\mathcal{R}$ are in the class Δ_k^p then the problem* ST($\mathcal{R}$) *is in the class Σ_{k+1}^p.*

Proof. (Sketch) First guess an interpretation I and then verify if it is a stable model. The guessing step is done by a non-deterministic Turing machine and the verification is done with an Oracle machine in the class Δ_{k+1}^p (see proof of the previous proposition). So, the complexity of the problem is

$$\mathrm{NP}^{\Delta_{k+1}^p} = \mathrm{NP}^{(P^{\Sigma_k^p})} = \mathrm{NP}^{\Sigma_k^p} = \Sigma_{k+1}^p. \qquad \blacksquare$$

6.2.2 Proofs of Hardness

This section contains the proofs for the hardness results on Table 6.3.

Proposition 6.8 *If $\mathcal{R}$ contains* SUM$_\subseteq$ *or* PROD$_\subseteq$ *aggregate relation then the problems* LEAST($\mathcal{R}$) *and* STD($\mathcal{R}$) *are NP-hard.*

Proof. (Sketch) Consider an instance $\{q_1, \ldots, q_n\}$ and k of the SUBSET-SUM problem which is NP-complete. Let P be the following definite aggregate program:

$$a_1. \quad \ldots \quad a_n.$$

$$p \leftarrow \text{SUM}_\subseteq(\{a_1 = q_1, \ldots, a_n = q_n\}, k).$$

The least fixpoint of P is obtained after two applications of the T_P^{aggr} operator. In the first application

$$T_P^{aggr}(\emptyset) = I_1 \supseteq \{a_1, \ldots, a_n\}$$

and $p \in I_1$ if and only if $k = 0$. For the second application let

$$T_P^{aggr}(I_1) = I_2.$$

We have that

$$\begin{aligned}
& p \in I_2 \\
\Leftrightarrow\ & I_1 \models \text{SUM}_\subseteq(\{a_1 = q_1, \ldots, a_n = q_n\}, k) \\
\Leftrightarrow\ & (\llbracket\{a_1 = q_1, \ldots, a_n = q_n\}\rrbracket_{I_1}, k) \in \text{SUM}_\subseteq \\
\Leftrightarrow\ & (\{q_1, \ldots, q_n\}, k) \in \text{SUM}_\subseteq \\
\Leftrightarrow\ & \exists S\colon S \subseteq \{q_1, \ldots, q_n\} \text{ and } (S, k) \in \text{SUM} \\
\Leftrightarrow\ & \text{the answer of SUBSET-SUM is ``yes''.}
\end{aligned}$$

The proof of NP-hardness for the $\text{PROD}_{\subseteq}$ aggregate can be obtained by substituting $\text{SUM}_{\subseteq}$ with $\text{PROD}_{\subseteq}$ in the above program. ■

Proposition 6.9 *If $\mathcal{R}$ contains one of the aggregate relations $\text{SUM}_{\neq}$ or $\text{PROD}_{\neq}$ then the problem $\text{WF}(\tilde{\mathcal{R}})$ is co-NP-hard.*

Proof. (Sketch) By reduction from the complement of SUBSET-SUM. Consider an instance of the SUBSET-SUM problem consisting of the set $\{q_1, \ldots, q_n\}$ and the number k. Let P be the following program:

$$\begin{array}{lll} a_i & \leftarrow \mathit{not}\, b_i. & \text{for } i = 1, \ldots, n \\ b_i & \leftarrow \mathit{not}\, a_i. & \text{for } i = 1, \ldots, n \end{array}$$

$$p \quad \leftarrow \text{SUM}_{\neq}(\{a_1 = q_1, \ldots, a_n = q_n\}, k).$$

We show that the atom p is true in the well-founded model of P if and only if the answer of the SUBSET-SUM problem is "no". First note that we can split the program P in two strata: the first one containing the atoms a_i and b_i and the second one containing the atom p. So, by the Splitting Theorem, the well-founded model of P can be obtained by computing the well-founded model of each stratum based on the well-founded model of the previous stratum. In the well-founded model of the first stratum all atoms a_i and b_i are undefined. In the well-founded model of the second stratum, p is true if and only if the answer to the SUBSET-SUM problem is "no".

The proof for the PROD aggregate is obtained by substituting SUM with PROD in the program. ■

Proposition 6.10 *If $\tilde{\mathcal{R}}$ contains one of the ultimate approximating aggregates $U_{\text{SUM}_{\subseteq}}$ or $U_{\text{PROD}_{\subseteq}}$ then the problem $\text{WF}(\tilde{\mathcal{R}})$ is both NP-hard and co-NP-hard.*

Proof. (Sketch) NP-hardness follows from Proposition 6.8 because for a definite aggregate program P the well-founded model of P is equal to the least model of P.

(*co-NP-hardness*) Consider an instance $\{q_1, \ldots, q_n\}$ and k of the SUBSET-SUM problem. Let P be the following program:

$$\begin{array}{lll} a_i & \leftarrow \mathit{not}\, b_i. & \text{for } i = 1, \ldots, n \\ b_i & \leftarrow \mathit{not}\, a_i. & \text{for } i = 1, \ldots, n \end{array}$$

$$\begin{array}{ll} q & \leftarrow \text{SUM}_{\subseteq}(\{a_1 = q_1, \ldots, a_n = q_n\}, k). \\ p & \leftarrow \mathit{not}\, q. \end{array}$$

We can show that p is true in the well-founded model of P if and only if the answer to the SUBSET-SUM problem is "no". The argument is analogous to the one in Proposition 6.9.

The proof of co-NP hardness for the $\text{PROD}_{\subseteq}$ aggregate can be obtained by substituting $\text{SUM}_{\subseteq}$ with $\text{PROD}_{\subseteq}$ in the above program. ■

For the Σ_2^p-completeness of ST we use a generalized subset sum problem with alternating existential and universal quantifiers (Berman et al., 1997).

Σ_2^p-SUBSET-SUM
Instance: Two vectors $\vec{u}$ and $\vec{v}$ of integers and an integer k.
Question: Is $\exists\vec{x}.\ \forall\vec{y}.\ \vec{u}\vec{x} + \vec{v}\vec{y} \neq k$ true where $\vec{x}$ and $\vec{y}$ are vectors of boolean variables of the same length as $\vec{u}$ and $\vec{v}$ respectively.

Proposition 6.11 *The problem* $\text{ST}(\{U^1_{\text{SUM}_\neq}\})$ *is* Σ_2^p*-complete.*

Proof. (Sketch) We use a reduction which is similar to the one used in the proof of the Σ_2^p-completeness of the existence of ultimate two-valued stable model for normal logic programs (Denecker et al., 2004). Let $m = |\vec{u}|$ and $n = |\vec{v}|$. Let P be the following program:

$$
\begin{array}{lll}
x_i & \leftarrow not\ \overline{x}_i. & \text{for } i = 1, \ldots, m \\
\overline{x}_i & \leftarrow not\ x_i. & \text{for } i = 1, \ldots, m \\
 & & \\
y_j & \leftarrow r. & \text{for } j = 1, \ldots, n \\
r & \leftarrow \text{SUM}_\neq(\{x_1 = u_1, \ldots, x_m = u_m, y_1 = v_1, \ldots, y_n = v_n\}, k). & \\
 & \leftarrow not\ r. &
\end{array}
$$

We can show that the above program has a two-valued stable model if and only if the answer to the Σ_2^p-SUBSET-SUM problem is "yes". ■

6.3 Related Work

The discussion of the related work in this section is for the three-valued stable semantics interpreting aggregates with their ultimate approximations.

6.3.1 Aggregates and Constraint Programming

It is possible to represent the problem of computing the first and second components of the ultimate aggregate relation as entailment and disentailment of a finite domain aggregate constraint. The syntax of a primitive aggregate constraint is exactly the same as aggregate atoms with propositional set expressions. Let (I_1, I_2) be a three-valued interpretation and $\text{R}(s, d)$ be an aggregate atom (primitive constraint). Consider the following constraint:

$$C = (\bigwedge_{p \in I_1} p = 1) \wedge (\bigwedge_{p \in At - I_2} p = 0) \wedge \text{R}(s, d).$$

We have the following correspondence between the three-valued truth value of the aggregate atom $\text{R}(s, t)$ and satisfiability of the constraint C.

Proposition 6.12
$\mathcal{H}_{(I_1,I_2)}(\mathrm{R}(s,t)) = \mathbf{t}$ *if and only if* $\tilde{\forall}C$ *is true and*
$\mathcal{H}_{(I_1,I_2)}(\mathrm{R}(s,t)) = \mathbf{f}$ *if and only if* $\tilde{\forall}C$ *is false.*

Testing satisfiability of the constraint $\tilde{\forall}C$ is possible only if the constraint solver supports aggregate constraints. For some aggregate functions and relations it is possible to translate an aggregate constraint to other primitive constraints.

Example 6.3 Consider an aggregate constraint of the form:

$$\mathrm{R}(\{a_1 = w_{a_1}, \ldots, a_m = w_{a_m}, not\, b_1 = w_{b_1}, \ldots, not\, b_n = w_{b_n}\}, d)$$

where $op \in \{\leq, <, =, \neq, \geq, >\}$. If R is the aggregate function CARD it can be translated to the constraint:

$$a_1 + \cdots + a_m + (1 - b_1) + \cdots + (1 - b_n)\ \ op\ \ d.$$

If R is the aggregate function SUM we obtain the constraint:

$$a_1 * w_{a_1} + \cdots + a_m * w_{a_m} + (1 - b_1) * w_{b_1} + \cdots + (1 - b_n) * w_{b_n}\ \ op\ \ d. \qquad \square$$

Pontelli, Son, and Elkabani have integrated the SMODELS ASP system with the finite domain CLP solver of ECL^iPS^e Prolog (Pontelli et al., 2004). Currently, their system is not based on any formal semantics. We believe that the correspondence between the truth value of aggregate atoms and finite domain constraints which we showed above can be used to implement the three-valued stable model semantics based on the ultimate approximating aggregate in the framework of (Pontelli et al., 2004).

6.3.2 Weight Constraint Rules

A closely related work is the stable model semantics of weight constrain rules defined by Simons, Niemelä, and Soininen (Simons et al., 2002) and implemented by the well-known SMODELS system. A *weight constraint* is an expression of the form $l \leq s \leq u$ where s is a set expression and l, u are numbers. The intended interpretation of such expression is that the sum of the weights of the atoms satisfied by an interpretation has to be between the lower bound l and the upper bound u. In our syntax, a weight constraint $l \leq s \leq u$ corresponds to the conjunction of aggregate atoms $\mathrm{SUM}_{\geq}(s, l) \wedge \mathrm{SUM}_{\leq}(s, u)$.

In (Simons et al., 2002), a weight constraint can also appear in the head of a rule. Because we do not allow aggregates as heads of rules we make a comparison for a subset of the language of weight constraint rules where all weight constraints in the head are of the form $1 \leq \{p = 1\}$ which corresponds to the atom p.

To study the relationship between the two semantics we give an alternative definition of the Φ_P^{aggr} operator for programs containing only $\mathrm{SUM}_{\geq}$ and $\mathrm{SUM}_{\leq}$

aggregates. We do this by defining reducts of aggregate atoms and programs in a similar fashion as weight constraints and programs (Simons et al., 2002). The first step is to put the set expressions of $\text{SUM}_{\geq}$ and $\text{SUM}_{\leq}$ atoms in a normal form containing only positive or negative weights. By switching the signs of the weight and the associated literal in a weight literal and also updating the result of the aggregate we obtain an equivalent aggregate atom. For example the aggregate atom $\text{SUM}(\{a = 5, b = -2\}, 3))$ is equivalent to $\text{SUM}(\{a = 5, not\, b = 2\}, 5)$. Let $sw_{L=w}$ be the transformation defined as follows:

$$sw_{L=w}(\text{SUM}(s, d)) = \text{SUM}((s \setminus \{L = w\}) \cup \{\overline{L} = -w\}, d - w)$$

Proposition 6.13 *$I \models \text{SUM}(s, d)$ if and only if $I \models sw_{L=w}(\text{SUM}(s, d))$, for any two-valued interpretation I and weight literal $L = w$.*

Proof. Let s' be the set expression of the aggregate atom $sw_{L=w}(\text{SUM}(s, d))$: $s' = s \setminus \{L = w\}) \cup \{\overline{L} = -w\}$. It is sufficient to show that for any interpretation I,

$$\sum([\![s]\!]_I) = \sum([\![s']\!]_I) + w. \tag{6.3}$$

Let $s_0 = s \setminus \{L = w\}$ and $M_0 = [\![s_0]\!]_I$. If $I \models l$ then $\sum([\![s]\!]_I) = (\sum M_0) + w$ and $\sum([\![s']\!]_I) = M_0$ from which (6.3) follows. If $I \not\models l$ then $\sum([\![s]\!]_I) = \sum M_0$ and $\sum([\![s']\!]_I) = \sum M_0 + (-w)$ and again (6.3) follows. ∎

Using this proposition we can show that the transformation $sw_{L=w}$ is also equivalent in three-valued logic for any aggregate relation derived from SUM.

Corollary 6.14 *$\text{SUM}_P(s, d) =_3 sw_{L=w}(\text{SUM}_P(s, d))$ for any relation $P \subseteq D \times D$.*

Definition 6.3 *Let A be an aggregate atom of the form $\text{SUM}_{\geq}(s, d)$ and I an interpretation. The* reduct *A^I of A under I is defined as follows. First, apply the $sw_{L=w}$ transformation on A for all weight literals $L = w \in s$ with negative weights. The result is an equivalent aggregate atom A^+ which contains only positive weights:*

$$A^+ = \text{SUM}_{\geq}(\{a_1 = w_{a_1}, \ldots, a_m = w_{a_m}, not\, b_1 = w_{b_1}, \ldots, not\, b_n = w_{b_n}\}, d').$$

The reduct A^I of A is defined as

$$A^I = \text{SUM}_{\geq}(\{a_1 = w_{a_1}, \ldots, a_m = w_{a_m}\}, d' - \sum_{b_i \notin I} w_{b_i}) \tag{6.4}$$

We note that A^I contains only positive weight literals with positive weights and the aggregate relation $\text{SUM}_{\geq}$ is monotone for positive numbers. So A^I is a monotone aggregate atom.

The reduct of an aggregate atom B with a $\text{SUM}_{\leq}$ aggregate relation is defined by first transforming B using $sw_{L=w}$ to an equivalent aggregate atom B^- containing

only negative weights. The reduct B^I of B is defined in exactly the same way as (6.4) (but using B^- instead of B^+). Again B^I is a monotone aggregate atom because it contains only positive literals with negative numbers and $\text{SUM}_{\leq}$ is a monotone aggregate relation on multisets of negative numbers.

The reduct S^I of a set S of aggregate atoms is defined as the set of the reducts of all atoms in S.

Three-valued satisfiability of A is related with two-valued satisfiability of the reduct A^I of A in the following way.

Lemma 6.15 *Let A be a $\text{SUM}_{\geq}$ or $\text{SUM}_{\leq}$ aggregate atom. Then $\mathcal{H}^1_{(I_1,I_2)}(A) = \mathbf{t}$ if and only if $I_1 \models A^{I_2}$ and $\mathcal{H}^2_{(I_1,I_2)}(A) = \mathbf{t}$ if and only if $I_2 \models A^{I_1}$.*

Proof. Fix a three-valued interpretation $\tilde{I} = (I_1, I_2)$. Let A be an aggregate atom of the form $\text{SUM}_{\geq}(s, d)$ with only positive weights and the set expression s has the form:

$$s = \{a_1 = w_{a_1}, \ldots, a_m = w_{a_m}, not\, b_1 = w_{b_1}, \ldots, not\, b_n = w_{b_n}\}.$$

Let $[\![s]\!]_{\tilde{I}} = (M_1, M_2)$. In particular $M_1 = \{\!|w_{a_i} \mid a_i \in I_1|\!\} \uplus \{\!|w_{b_i} \mid b_i \notin I_2|\!\}$.

By definition of satisfiability of aggregate atoms $\mathcal{H}^1_{\tilde{I}}(A) = \mathbf{t}$ if and only if $((M_1, M_2), d) \in U^1_{\text{SUM}_{\geq}}$. By Lemma 5.13 and Proposition 5.15 $((M_1, M_2), d) \in U^1_{\text{SUM}_{\geq}}$ if and only if $\sum M_1^+ + \sum M_2^- \geq d$. Because there are no negative weights then $\mathcal{H}^1_{\tilde{I}}(A) = \mathbf{t}$ if and only if $\sum M_1 \geq d$.

The reduct A^{I_2} is of A defined as

$$A^{I_2} = \text{SUM}_{\geq}(\{a_1 = w_{a_1}, \ldots, a_m = w_{a_m}\}, d - \sum_{b_i \notin I_2} w_{b_i})$$

By definition $I_1 \models A^{I_2}$ if and only if

$$\text{SUM}([\![\{a_1 = w_{a_1}, \ldots, a_m = w_{a_m}\}]\!]_{I_1}) \geq d - \sum_{b_i \notin I_2} w_{b_i}$$

if and only if

$$\sum_{a_i \in I_1} w_{a_i} \geq d - \sum_{b_i \notin I_2} w_{b_i}$$

which is true if and only if $\sum M_1 \geq d$.

The proofs for $\mathcal{H}^2$ and $\text{SUM}_{\leq}$ are analogous. ∎

Let P be a program containing only $\text{SUM}_{\leq}$ and $\text{SUM}_{\geq}$ aggregates. We extend the standard definition (Gelfond and Lifschitz, 1988) of *reduct* P^I of a program P with respect to an interpretation I to aggregate programs. Let $r \in P$ be a rule from P. Then if all negative literals of r are satisfied by I, i.e. $neg(r) \cap I = \emptyset$, then P^I contains the rule $head(r) \leftarrow pos(r), aggr(r)^I$. The program reduct relates with the Φ^{aggr}_P operator in the following way.

Proposition 6.16 *Let P be an aggregate program containing only* $\text{SUM}_{\leq}$ *and* $\text{SUM}_{\geq}$ *aggregates. Then* $(\Phi_P^{aggr})^1(I_1, I_2) = T_{P^{I_2}}^{aggr}(I_1)$.

We can establish a correspondence between the two semantics only for a restricted class of weight constraint rules.

Theorem 6.17 *Stable models of a program with weight constraints without upper bounds and without weight constraints in the heads of rules coincide with exact stable models of aggregate programs.*

Proof. We first show that the definitions of reducts for aggregate programs and for programs with weight constraints coincide:

- The definition of a reduct of a $\text{SUM}_{\geq}(s, d)$ aggregate atom is the same as the definition of the reduct of a weight constraint $d \leq s$.
- For a positive literal a, the reduct $(1 \leq \{a = 1\})^I$ is again $1 \leq \{a = 1\}$ for any interpretation I.
- For a negative literal $not\, a$, the reduct $(1 \leq \{not\, a = 1\})^I$ is $0 \leq \{\}$ if $a \notin I$. The weight constraint $0 \leq \{\}$ is always true and can be removed from the body of the rule. If $a \in I$ then $(1 \leq \{not\, a = 1\})^I$ is $1 \leq \{\}$ which is false for every interpretation and so the whole rule can be removed.

Using the alternative characterization of two-valued stable models (Proposition 2.17) we have that I is a stable model of a program P if and only if (i) $T_P^{aggr}(I) = I$ and (ii) $I = \text{lfp}(T_{P^I}^{aggr})$. Condition (i) is necessary to guarantee that the operator in condition (ii) is well-defined. However, for programs with $\text{SUM}_{\leq}$ and $\text{SUM}_{\geq}$ aggregates, the $T_{P^I}^{aggr}(J)$ operator is well-defined for any pair of interpretations I and J. Thus condition (i) can be dropped (since it is implied by (ii)). Finally, an interpretation satisfies condition (ii) if and only if it is the deductive closure of P^I, as defined in (Simons et al., 2002).

In the definition of stable models of weight constraints (Simons et al., 2002) there is an extra condition that an interpretation should also be a model of the program P. However, this condition is relevant only for programs with weight constraints in the head. According to approximation theory, any exact stable fixpoint is a fixpoint of T_P^{aggr} and consequently a model of P. So, this condition is always satisfied. ∎

For weight constraints with an upper bound the two semantics do not correspond in general.

Example 6.4 Consider the following weight constraint program P_{wc}:

$$a \leftarrow \{not\, a = 1\} \leq 0.$$

Intuitively, the rule expresses the fact that a is true if $not\, a$ is false or equivalently, a is true if a is true. The corresponding aggregate program $P = \{a \leftarrow \text{SUM}_{\leq}(\{not\, a = 1\}, 0)\}$ is definite and the T_P^{aggr} operator is monotone. Its least fixpoint is the empty set which is also equal to the well-founded and the single stable model. However, under the semantics of weight constraints a rule with an upper bound is translated to a rule with a lower bound by introducing an intermediate atom:

$$a \leftarrow not\, b.$$
$$b \leftarrow 1 \leq \{not\, a = 1\}.$$

This program is equivalent to the program P':

$$a \leftarrow not\, b.$$
$$b \leftarrow not\, a.$$

which has two stable models $\{a\}$ and $\{b\}$. So the stable models of the original program are $\{a\}$ and $\emptyset$ which disagrees with the intuitive reading of the program.□

As the above example illustrates our semantics assigns a more intuitive meaning to programs with weight constraints with upper bounds. In particular, the exact stable models as defined in our approach are always minimal models of the program, which is not the case in the SMODELS approach.

If under our semantics the transformation of weight constraints with upper bounds to weight constraints with lower bounds as used by the SMODELS system preserves the set of stable models then the two semantics are equivalent. The next proposition gives sufficient conditions for that.

Proposition 6.18 *If every negative weight literal in a weight constraint with upper bound does not depend on the atom in the head of the corresponding rule then the two semantics coincide.*

Because such examples rarely occur in practice the difference in the two approaches is not so serious.

We point out that the definition of a reduct of a $\text{SUM}_{\leq}$ aggregate atom can be applied to weight constraints with an upper bound. This results in a definition of stable semantics of weight constraint rules different from the one currently used by SMODELS but equivalent to ours. With our approach, the only difference between a reduct of a weight constraint with a lower bound and a weight constraint with an upper bound is that the first one has to be put in a normal form containing only positive weights while the second one has to be put in a normal form containing only negative weights. So, we believe that it is feasible to implement our alternative semantics in the SMODELS system.

6.3.3 Other

Programs with monotone cardinality atoms (mc-atoms) (Marek et al., 2004) and set-constraint atoms (sc-atoms) (Marek and Remmel, 2004) are two variations of the language of weight constraints (Simons et al., 2002). Mc-atoms are similar to weight constraints with lower bounds with the following two restrictions: they can express a lower bound only on the cardinality of the input set (thus no sum) and negative literals are not allowed in the set expression. To express an upper limit on the cardinality of a set one can use a negative mc-atom. For this restricted language the stable semantics of weight constraints is the same as the stable semantics of mca-programs. So, we can apply the results from Section 6.3.2 on the relationship between our semantics and the semantics of weight constraint rules. However, because mca-programs do not have negative literals in set expressions stable models of an mca-program correspond to exact stable models of the corresponding aggregate program (assuming that there are no mc-atoms in the heads of rules).

Set constraint atoms, sc-atoms for short, (Marek and Remmel, 2004) are very similar to aggregate atoms, however the interpretation of the aggregate relations defined explicitly in the program. A sc-atom is an expression of the form $\langle X, \mathcal{F} \rangle$ where $X \subseteq At$ is a set of atoms and $\mathcal{F} \subseteq \wp(X)$ is a family of subsets of X. A sc-atom $\langle X, \mathcal{F} \rangle$ is satisfied by an interpretation I, denoted with $I \models \langle X, \mathcal{F} \rangle$ if $X \cap I \subseteq \mathcal{F}$. So, an aggregate atom of the form $\mathrm{R}(\{a_1 = w_1, \ldots, a_n = w_n\}, d)$ without negative weight literals where R is an arbitrary aggregate relation can be represented as the following sc-atom:

$$\langle \{a_1, \ldots, a_n\}, \{S \subseteq \{a_1, \ldots, a_n\} \mid S \models \mathrm{R}(\{a_1 = w_1, \ldots, a_n = w_n\}, d)\} \rangle.$$

Clearly satisfiability of the aggregate atom $\mathrm{R}(\{a_1 = w_1, \ldots, a_n = w_n\}, d)$ and the satisfiability of the above sc-atom are equivalent for two-valued interpretations. The stable semantics of sc-programs is defined as an extension of the stable semantics of programs with weight constraint and is therefore, in general, different than our approach. More research is necessary to give precise relationship between the two semantics.

A translation of weight constraints to nested expressions which preserves the set of answer sets is given in (Ferraris and Lifschitz, 2004). The semantics of weight constraints and consequently the translation of (Ferraris and Lifschitz, 2004) is defined only for set expressions with non-negative weights. For multisets of such numbers, the aggregate relation $\mathrm{SUM}_{\geq}$ is monotone. For weight constraints of the form $l \leq s$, the translation of (Ferraris and Lifschitz, 2004) is exactly the same as tr_{mon}. However, because of the different semantics of weight constraints with upper bounds the translation which we give and the one from (Ferraris and Lifschitz, 2004) are different.

6.3.4 Complexity

Concerning the complexity analysis of the different semantics, our view of aggregate relations as oracle machines is very similar to the relationship between generalized quantifiers and oracles (Makowsky and Pnueli, 1994). Another closely related work is the result on the complexity of model checking of first-order logic extended with generalized quantifiers by Gottlob (Gottlob, 1997). The result on complexity of the stable semantics for the aggregate relations $\text{SUM}_{\geq}$ and $\text{SUM}_{\leq}$ (Proposition 6.2) have been shown before for programs with weight constraint rules (Simons et al., 2002).

6.4 Conclusions

The main focus in this chapter was the study of the complexity of the various semantics of aggregate programs. One of the interesting results is that extending the ultimate well-founded and stable semantics to logic programs which use polynomial time computable aggregate relations does not increase complexity. The other important outcome is that for most standard aggregate relations (which are polynomial) the complexity of the well-founded and stable semantics based on the ultimate approximating aggregate does not increase. We argue below that for the aggregate relations for which it increases (SUM and PROD) the language is not suitable for expressing complete problems in these classes. So, in practice one will use approximating aggregate relations of PROD and SUM which are less precise than the ultimate approximating aggregates but polynomial time computable.

The results on lower bounds of complexity of the semantics which use (approximating) aggregate relations in the class Δ_2^P and which are not known to be polynomial (Table 6.3) can possibly be improved, hopefully obtaining completeness for the class Δ_2^P. However, even for the existing lower bounds (co-NP, NP, and Σ_2^p), we think that the language is not very suitable for expressing typical problems in these classes (Garey and Johnson, 1979; Schaefer and Umans, 2002). The only way to encode a complete problem for some of these classes as an aggregate program is to use aggregate atoms as an interface to an oracle machine (unless P=NP). However, this interface is very restrictive: the input consists only of a set and an element from the same domain as the set. Moreover the oracles can answer only questions of a very limited form — if all subsets of the input set satisfy certain relation with the element.

Chapter 7

Conclusions

In the last chapter of the thesis we summarize our main results and contributions and outline potential topics for future research.

The goal of the thesis was to do an extensive study of semantics of logic programs extended with arbitrary aggregate atoms in the rule bodies. We extended the major semantics of normal logic programs: the minimal model semantics of definite programs (van Emden and Kowalski, 1976); Clark's completion semantics (Clark, 1978); the standard model of stratified programs (Apt et al., 1988); the three-valued stable semantics (Przymusinksi, 1990) which includes as special cases both the well-founded (Van Gelder et al., 1991) and the stable semantics (Gelfond and Lifschitz, 1988) of normal logic programs; and the ultimate well-founded and ultimate stable semantics (Denecker et al., 2004).

The main contribution of the thesis is a study of several three-valued stable semantics of aggregate programs. To define such semantics we used the theory of approximating operators (Denecker et al., 2000). One argument why this theory is appropriate for our work is that it unifies the semantics of several non-monotonic logics including Default Logic, Autoepistemic Logic, and Logic Programming (Denecker et al., 2003). To define a three-valued stable model semantics using this theory, one has to define a suitable approximating operator of the immediate consequence operator T_P^{aggr}. However, the conditions of an approximating operator are relatively weak. So, there are many possible ways to define an approximating operator of T_P^{aggr} and consequently many possible versions of a three-valued stable semantics. The two main criteria for selecting an approximating operator are precision and complexity. (Denecker et al., 2004) defined a precision order between approximating operators and showed that more precise approximating operators have more precise well-founded models and a larger number of total stable models. Moreover, there exists a most precise approximating operator called *ultimate* which has the most precise well-founded fixpoint and the largest number of total stable fixpoints. This precision comes at a cost: computing the well-founded and

total stable models is one level higher in the polynomial hierarchy than the corresponding semantics of logic programs without aggregates (Denecker et al., 2004). This increase in complexity is not caused by aggregates atoms. In fact we showed that if all aggregate relations in the program are polynomial time computable then there is no further increase in complexity.

To define a three-valued stable semantics with lower complexity we extended the three-valued stable semantics of logic programs without aggregates (Przymusinksi, 1990). For this semantics the source of complexity is isolated to aggregate atoms. According to approximation theory this semantics can be obtained from the three-valued immediate consequence operator Φ_P defined by Fitting (Fitting, 1985) which we extended to an approximating operator Φ_P^{aggr} of T_P^{aggr}. This was done by extending the evaluation function of set expressions and interpreting aggregate relations with approximating aggregate relations which are a special type of three-valued relations which take as input three-valued multisets. The definition of approximating aggregate relations is application specific and the main criteria is again a balance between precision and complexity.

In Section 4.3, Chapter 5, and Section 6.2 we investigated the properties of the three-valued stable semantics based on the ultimate approximating aggregate (UAA). For most of the standard aggregate relations our results indicate that using the UAA is appropriate. This is certainly the case for monotone and anti-monotone aggregate relations for which the three-valued semantics based on the UAA extends the least model semantics of definite aggregate programs and the standard model of weakly stratified aggregate programs. For the aggregate relations MIN, MAX using the UAA results in a semantics which extends previous proposals for well-founded semantics (Ganguly et al., 1995; Van Gelder, 1992). Similarly, two-valued stable models of aggregate programs using $\text{SUM}_{\geq}$ and $\text{SUM}_{\leq}$ aggregates coincide with the stable semantics of programs with weight constraints (Simons et al., 2002) for a large class of programs. Also for most standard aggregate relations the complexity of computing the ultimate approximating aggregate remains polynomial. The only exceptions are SUM and PROD for which there is an increase in complexity. At the end of Chapter 5 we discussed how to define less precise approximating aggregate relations for SUM and PROD which are polynomial time computable.

We also studied many examples with recursion over aggregation. Some of them have to be interpreted as inductive definitions and an extension of the well-founded or stable semantics agrees with the intended interpretation. However, we presented several examples for which non-minimal models are valid solutions. We argued that the programs of these examples should be interpreted as *if-and-only-if* definitions and an extension of the program completion semantics (Clark, 1978) is the appropriate semantics.

All examples which we discussed in this work use only the aggregate relations MIN, MAX, CARD, and SUM. They have been or can be successfully modeled by previous approaches. However, our results are much more general. We not only support arbitrary relations between multisets and simple values but the results can

be extended in a straightforward way to arbitrary second-order relations with a fixed interpretation which take as input any number of sets, functions, and simple values. So, an important topic for future research is to look for novel applications of our semantics.

Another interesting direction for future work is to study extensions of other non-monotonic logics with aggregates, for example Default Logic or Autoepistemic Logic. The major semantics of these logics can be also obtained as instances of Approximation Theory (Denecker et al., 2003) so we can possibly follow a similar methodology as for the three-valued stable semantics.

Bibliography

Abramsky, S. and Jung, A. (1994). Domain theory. In Abramsky, S., Gabbay, D., and Maibaum, T. S. E., editors, *Handbook of Logic in Computer Science*, volume 3, pages 1–168. Oxford University Press.

Apt, K. R., Blair, H. A., and Walker, A. (1988). Towards a theory of declarative knowledge. In Minker, J., editor, *Foundations of Deductive Databases and Logic Programming*, chapter 2, pages 89–148. Morgan Kaufmann.

Apt, K. R. and Bol, R. N. (1994). Logic programming and negation: A survey. *Journal of Logic Programming*, 19/20:9–71.

Baral, C. (2003). *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press.

Baral, C. and Gelfond, M. (1994). Logic programming and knowledge representation. *Journal of Logic Programming*, 19/20:73–148.

Beldiceanu, N. and Contejean, E. (1994). Introducing global constraints in CHIP. *Mathematical and Computer Modelling*, 20(12):97–123.

Belnap, N. D. (1997). A useful four-valued logic. In Michael Dunn, J. and Epstein, G., editors, *Modern Uses of Multiple-valued Logic*, pages 8–37. D. Reidel.

Berman, P., Karpinski, M., Larmore, L. L., Plandowski, W., and Rytter, W. (1997). On the complexity of pattern matching for highly compressed two-dimensional texts. In *Proc. 8th Annual Symposium on Combinatorial Pattern Matching*, volume 1264 of *Lecture Notes in Computer Science*. Springer.

Clark, K. L. (1978). Negation as failure. In Gallaire, H. and Minker, J., editors, *Logic and Data Bases*, pages 293–322. Plenum Press.

Comon, H. (1991). Disunification: A survey. In Lassez, J.-L. and Plotkin, G., editors, *Computational Logic - Essays in Honor of Alan Robinson*, pages 322–359. MIT Press.

Cousout, P. and Cousout, R. (1979). Constructive versions of Tarski's fixed point theorems. *Pacific Journal of Mathematics*, 82(1):43–57.

Dantsin, E., Eiter, T., Gottlob, G., and Voronkov, A. (2001). Complexity and expressive power of logic programming. *ACM Computing Surveys*, 33(3):374–425.

Davey, B. A. and Pristley, H. A. (1990). *Introduction to Lattices and Order*. Cambridge University Press.

Dell'Armi, T., Faber, W., Ielpa, G., Leone, N., and Pfeifer, G. (2003a). Aggregate functions in disjunctive logic programming: Semantics, complexity, and implementation in DLV. In Gottlob, G., editor, *Proceedings of the 18th International Joint Conference on Artificial Intelligence*, pages 847–852. Morgan Kaufmann.

Dell'Armi, T., Faber, W., Ielpa, G., Leone, N., and Pfeifer, G. (2003b). Aggregate functions in DLV. In De Vos, M. and Provetti, A., editors, *Answer Set Programming: Advances in Theory and Implementation*, volume 78 of *CEUR Workshop proceedings*, pages 274–288. online CEUR-WS.org/Vol-78/.

Denecker, M. (1998). The well-founded semantics is the principle of inductive definition. In Dix, J., del Cerro, L. F., and Furbach, U., editors, *Logics in Artificial Intelligence, European Workshop*, volume 1489 of *LNAI*, pages 1–16, Dagstuhl, Germany. Springer.

Denecker, M. (2000). Extending classical logic with inductive definitions. In Lloyd, J. et al., editors, *1st International Conference on Computational Logic*, volume 1861 of *Lecture Notes in Artificial Intelligence*, pages 703–717. Springer.

Denecker, M. (2004). What's in a model? epistemological analysis of logic programming. In *Proceedings of the 9th International Conference on Principles of Knowledge Representation and Reasoning*. To appear.

Denecker, M., Bruynooghe, M., and Marek, V. (2001a). Logic programming revisited: logic programs as inductive definitions. *ACM Transactions on Computational Logic*, 2(4):623–654.

Denecker, M. and De Schreye, D. (1998). SLDNFA: an abductive procedure for abductive logic programs. *Journal of Logic Programming*, 34(2):201–226.

Denecker, M., Marek, V., and Truszczyński, M. (2000). Approximating operators, stable operators, well-founded fixpoints and applications in non-monotonic reasoning. In Minker, J., editor, *Logic-based Artificial Intelligence*, pages 127–144. Kluwer Academic Publishers.

Denecker, M., Marek, V., and Truszczyński, M. (2002). Ultimate approximations in nonmonotonic knowledge representation systems. In *Principles of Knowledge Representation and Reasoning*, pages 177–188. Morgan Kaufmann.

Denecker, M., Marek, V., and Truszczyński, M. (2003). Uniform semantic treatment of default and autoepistemic logics. *Artificial Intelligence*, 143(1):79–122.

Denecker, M., Marek, V., and Truszczyńsky, M. (2004). Ultimate approximations and its application in nonmonotonic knowledge representation. *Information and Computation*. Accepted.

Denecker, M., Pelov, N., and Bruynooghe, M. (2001b). Ultimate well-founded and stable model semantics for logic programs with aggregates. In Codognet, P., editor, *17th International Conference on Logic Programming*, volume 2237 of *LNCS*, pages 212–226. Springer.

Dix, J., Osorio, M., and Zepeda, C. (2001). A general theory of confluent rewriting systems for logic programming and its applications. *Annals of Pure and Applied Logic*, 108(1–3):153–188.

Eiter, T., Faber, W., Leone, N., and Pfeifer, G. (2000). Declarative problem solving using the DLV system. In Minker, J., editor, *Logic-Based Artificial Intelligence*, pages 79–103. Kluwer Academic Publishers.

Ferraris, P. and Lifschitz, V. (2004). Weight constraints as nested expressions. *Theory and Practice of Logic Programming*. To appear.

Fitting, M. (1985). A Kripke-Kleene semantics for logic programs. *Journal of Logic Programming*, 2(4):295–312.

Fitting, M. (1991a). Bilattices and the semantics of logic programming. *Journal of Logic Programming*, 11(1,2):91–116.

Fitting, M. (1991b). Well-founded semantics, generalized. In Saraswat, V. A. and Ueda, K., editors, *Logic Programming, Proceedings of the International Symposium, San Diego, California*, pages 71–84. MIT Press.

Fitting, M. (1993). The family of stable models. *Journal of Logic Programming*, 17(2–4):197–225.

Fitting, M. (1997). A theory of truth that prefers falsehood. *Journal of Philosophical Logic*, 26:477–500.

Fitting, M. (2002). Fixpoint semantics for logic programming a survey. *Theoretical Computer Science*, 278(1–2):25–51.

Fung, T. H. and Kowalski, R. (1997). The IFF procedure for abductive logic programming. *Journal of Logic Programming*, 33(2):151–165.

Ganguly, S., Greco, S., and Zaniolo, C. (1995). Extrema predicates in deductive databases. *Journal of Computer and System Sciences*, 51(2):244–259.

Garey, M. R. and Johnson, D. S. (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company.

Gelfond, M. (2002). Representing knowledge in A-Prolog. In Kakas, A. C. and Sadri, F., editors, *Computational Logic: Logic Programming and Beyond, Essays in Honour of Robert A. Kowalski, Part II*, volume 2408 of *Lecture Notes in Computer Science*, pages 413–451. Springer.

Gelfond, M. and Lifschitz, V. (1988). The stable model semantics for logic programming. In Kowalski, R. A. and Bowen, K. A., editors, *Logic Programming, Proc. of the 5th International Conference and Symposium*, pages 1070–1080. MIT Press.

Gelfond, M. and Lifschitz, V. (1991). Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9(3/4):365–386.

Gilis, D. and Denecker, M. (2002). Compositionality results for stratified nonmonotone operators. In Benferhat, S. and Giunchiglia, E., editors, *Proceedings of the 9th International Workshop on Non-Monotonic reasoning*, pages 51–56.

Ginsberg, M. L. (1988). Multivalued logics: A uniform approach to reasoning in artificial intelligence. *Computational Intelligence*, 4(3):256–316.

Gottlob, G. (1997). Relativized logspace and generalized quantifiers over finite ordered structures. *Journal of Symbolic Logic*, 62(2):545–574.

Heidt, M. L. (2001). Developing an inference engine for ASET-Prolog. Master's thesis, University of Texas at El Paso.

Hurley, J. F. (1971). *Litton's Problematical Recreations*. New York Van Nostrand Reinhold Co.

Jaffar, J. and Maher, M. (1994). Constraint logic programming: A survey. *Journal of Logic Programming*, 19/20:503–581.

Jaffar, J., Maher, M., Marriott, K., and Stuckey, P. (1998). The semantics of constraint logic programs. *Journal of Logic Programming*, 37(1–3):1–46.

Kakas, A. C., Kowalski, R. A., and Toni, F. (1998). The role of abduction in logic programming. In Gabbay, D. M., Hogger, C., and Robinson, J., editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 5, pages 235–324. Oxford University Press.

Kakas, A. C., Michael, A., and Mourlas, C. (2000). ACLP: Abductive constraint logic programming. *Journal of Logic Programming*, 44(1–3):129–177.

Kakas, A. C., Van Nuffelen, B., and Denecker, M. (2001). A-system: Problem solving through abduction. In Nebel, B., editor, *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence*, pages 591–596. Morgan Kaufmann.

Kemp, D. B., Srivastava, D., and Stuckey, P. J. (1995). Bottom-up evaluation and query optimization of well-founded models. *Theoretical Computer Science*, 146(1–2):145–184.

Kemp, D. B. and Stuckey, P. J. (1991). Semantics of logic programs with aggregates. In Saraswat, V. A. and Ueda, K., editors, *Proceedings of the International Logic Programming Symposium*, pages 387–401. MIT Press.

Kunen, K. (1987). Negation in logic programming. *Journal of Logic Programming*, 4:289–308.

Lassez, J.-L., Nguyen, V. L., and Sonenberg, E. A. (1982). Fixed point theorems and semantics: a folk tale. *Information Processing Letters*, 14(3):112–116.

Lierler, Y. and Maratea, M. (2004). Cmodels-2: SAT-based answer set solver enhanced to non-tight programs. In Lifschitz, V. and Niemelä, I., editors, *7th International Conference on Logic Programming and Nonmonotonic Reasoning*, volume 2923 of *LNCS*, pages 346–350. Springer. System Description.

Lifschitz, V. (2002). Answer set programming and plan generation. *Journal of Articial Intelligence*, 138:39–54.

Lifschitz, V., Pearce, D., and Valverde, A. (2001). Strongly equivalent logic programs. *ACM Transactions on Computational Logic*, 2(4):526–541.

Lifschitz, V., Tang, L. R., and Turner, H. (1999). Nested expressions in logic programs. *Annals of Mathematics and Artificial Intelligence*, 25(3-4):369–389.

Lifschitz, V. and Turner, H. (1994). Splitting a logic program. In *International Conference on Logic Programming*, pages 23–37. MIT Press.

Lindström, P. (1966). First order predicate logic with generalized quantifiers. *Theoria,*, 32:186–195.

Lloyd, J. W. (1987). *Foundations of Logic Programming*. Springer-Verlag, second edition.

Lloyd, J. W. and Topor, R. W. (1984). Making Prolog more expressive. *Journal of Logic Programming*, 1(3):225–240.

Makowsky, J. A. and Pnueli, Y. B. (1994). Oracles and quantifiers. In Börger, E., Gurevich, Y., and Meinke, K., editors, *Selected Papers from CSL'93*, volume 832 of *Lecture Notes in Computer Science*, pages 189–222. Springer.

Marek, V., Niemelä, I., and Truszczynski, M. (2004). Logic programs with monotone cardinality atoms. In Lifschitz, V. and Niemelä, I., editors, *7th International Conference on Logic Programming and Nonmonotonic Reasoning*, volume 2923 of *LNCS*, pages 155–166. Springer.

Marek, V. and Remmel, J. (2002). On logic programs with cardinality constraints. In *9th International Workshop on Non-Monotonic Reasoning*, pages 219–228.

Marek, V. and Remmel, J. (2004). Set constraints in logic programming. In Lifschitz, V. and Niemelä, I., editors, *7th International Conference on Logic Programming and Nonmonotonic Reasoning*, volume 2923 of *LNCS*, pages 167–179. Springer.

Marek, V. and Truszczyński, M. (1999). Stable models and an alternative logic programming paradigm. In Apt, K., Marek, V. W., Truszczyński, M., and Warren, D., editors, *The Logic Programming Paradigm: a 25 Years Perspective*, pages 375–398. Springer-Verlag.

Markowsky, G. (1976). Chain-complete posets and directed sets with applications. *Algebra Universalis*, 6:53–68.

Mumick, I. S., Pirahesh, H., and Ramakrishnan, R. (1990). The magic of duplicates and aggregates. In McLeod, D., Sacks-Davis, R., and Schek, H.-J., editors, *16th International Conference on Very Large Data Bases*, pages 264–277. Morgan Kaufmann.

Niemelä, I. (1999). Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, 25(3,4):241–273.

Osorio, M. and Jayaraman, B. (1999). Aggregation and negation-as-failure. *New Generation Computing*, 17(3):255–284.

Osorio, M., Jayaraman, B., and Plaisted, D. A. (1999). Theory of partial-order programming. *Science of Computer Programming*, 34(3):207–238.

Pelov, N. and Bruynooghe, M. (1999). Proving failure of queries for definite logic programs using XSB-Prolog. In Ganzinger, H., McAllester, D., and Voronkov, A., editors, *6th International Conference on Logic for Programming and Automated Reasoning*, volume 1705 of *LNAI*, pages 358–375. Springer.

Pelov, N. and Bruynooghe, M. (2000). Extending constraint logic programming with open functions. In Gabbrielli, M. and Pfenning, F., editors, *2nd International Conference on Principles and Practice of Declarative Programming*, pages 235–244. ACM Press.

Pelov, N., De Mot, E., and Bruynooghe, M. (2000a). A comparison of logic programming approaches for representation and solving of constraint satisfaction problems. In Denecker, M., Kakas, A., and Toni, F., editors, *8th International Workshop on Non-Monotonic Reasoning*.

Pelov, N., De Mot, E., and Denecker, M. (2000b). Logic programming approaches for representing and solving constraint satisfaction problems: a comparison. In Parigot, M. and Voronkov, A., editors, *Logic for Programming and Automated Reasoning, 7th International Conference*, volume 1955 of *LNAI*, pages 225–239. Springer.

Pelov, N., Denecker, M., and Bruynooghe, M. (2003). Translation of aggregate programs to normal logic programs. In De Vos, M. and Provetti, A., editors, *Answer Set Programming: Advances in Theory and Implementation*, volume 78 of *CEUR Workshop Proceedings*, pages 29–42. online CEUR-WS.org/Vol-78/.

Pelov, N., Denecker, M., and Bruynooghe, M. (2004). Partial stable semantics for logic programs with aggregates. In Lifschitz, V. and Niemelä, I., editors, *7th International Conference on Logic Programming and Nonmonotonic Reasoning*, volume 2923 of *LNCS*, pages 207–219. Springer.

Pelov, N. and Truszczyński, M. (2004). Semantics of disjunctive programs with monotone aggregates — an operator-based approach. In *10th International Workshop on Non-Monotonic Reasoning*.

Pontelli, E., Son, T. C., and Elkabani, I. (2004). Smodels with CLP — a treatment of aggregates in ASP. In Lifschitz, V. and Niemelä, I., editors, *7th International Conference on Logic Programming and Nonmonotonic Reasoning*, volume 2923 of *LNCS*, pages 356–360. Springer. System Description.

Przymusinksi, T. (1990). The well-founded semantics coincides with the three-valued stable semantics. *Fundamenta Informaticae*, 13(4):445–464.

Przymusinska, H. and Przymusinski, T. (1990). Weakly stratified logic programs. *Fundamenta Informaticae*, 13:51–65.

Przymusinski, T. (1988). On the declarative semantics of deductive databases and logic programs. In Minker, J., editor, *Foundations of Deductive Databases and Logic Programming*, chapter 5, pages 193–216. Morgan Kaufmann.

Przymusinski, T. (1991). Stable semantics for disjunctive programs. *New Generation Computing*, 9:401–424.

Rao, P., Sagonas, K., Swift, T., Warren, D., and Freire, J. (1997). XSB: A system for efficiently computing well-founded semantics. In *Proc. of the 4th Int. Conference on Logic Programming and Nonmonotonic Reasoning*, volume 1265 of *LNCS*, pages 430–440. Springer.

Ross, K. A. and Sagiv, Y. (1997). Monotonic aggregation in deductive databases. *Journal of Computer and System Sciences*, 54(1):79–97.

Rounds, W. C. and Zhang, G.-Q. (2001). Clausal logic and logic programming in algebraic domains. *Information and Computation*, 171(2):183–200.

Sakama, C. (1989). Possible model semantics for disjunctive databases. In *Proc. of the First International Conference on Deductive and Object Oriented Databases*, pages 1055–1060. Elsevier.

Schaefer, M. and Umans, C. (2002). Completeness in the polynomial-time hierarchy: A compendium. *SIGACT News*, 33(3).

Seipel, D., Minker, J., and Ruiz, C. (1997). Model generation and state generation for disjunctive logic programs. *Journal of Logic Programming*, 32(1):49–69.

Simons, P., Niemelä, I., and Soininen, T. (2002). Extending and implementing the stable model semantics. *Artificial Intelligence*, 138(1–2):181–234.

Smyth, M. B. (1978). Power domains. *Journal of Computer and System Sciences*, 16:23–36.

Sudarshan, S., Srivastava, D., Ramakrishnan, R., and Beeri, C. (1993). Extending the well-founded and valid semantics for aggregation. In Miller, D., editor, *International Logic Programming Symposium*, pages 590–608. MIT Press.

Tarski, A. (1955). A lattice-theoretic fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5(2):285–309.

van Eijck, J. (1996). Quantifiers and partiallity. In van der Does, J. and van Eijck, J., editors, *Quantifiers, Logic, and Language*, pages 105–144. CSLI.

van Emden, M. H. and Kowalski, R. A. (1976). The semantics of predicate logic as a programming language. *Journal of the ACM*, 23(4):733–742.

Van Gelder, A. (1989). The alternating fixpoint of logic programs with negation. In *Proceedings of the Eight ACM Symposium on Principles of Database Systems*, pages 1–10. ACM Press.

Van Gelder, A. (1992). The well-founded semantics of aggregation. In *11th ACM Symposium on Principles of Database Systems*, pages 127–138. ACM Press.

Van Gelder, A. (1993). Foundations of aggregation in deductive databases. In Ceri, S., Tanaka, K., and Tsur, S., editors, *3rd International Conference on Deductive and Object-Oriented Databases*, volume 760 of *LNCS*, pages 13–34. Springer.

Van Gelder, A., Ross, K. A., and Schlipf, J. S. (1991). The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650.

Van Nuffelen, B. (2004). *Abductive Constraint Logic Programming: Implementation and Applications*. PhD thesis, K.U.Leuven, Belgium.

Van Nuffelen, B. and Denecker, M. (2000). Problem solving in ID-logic with aggregates: some experiments. In Mark Denecker, A. K., editor, *8th International Workshop on Nonmonotonic Reasoning, special track on Abductive Reasoning*, Breckenridge, Colorado, USA.

Vennekens, J., Gilis, D., and Denecker, M. (2003). Stratification in non-monotonic logic: A unified approach. Draft.

Zaniolo, C. and Wang, H. (1999). Logic-based user-defined aggregates for the next generation of database systems. In Apt, K., Marek, V., Truszczynski, M., and Warren, D., editors, *The Logic Programming Paradigm: Current Trends and Future Directions*. Springer.

List of Notation

P, 17
NP, 17
Δ_k^p, 17
Σ_k^p, 17
Π_k^p, 17
SUBSET-PRODUCT, 97
SUBSET-SUM, 97
Σ_2^p-SUBSET-SUM, 129

$=_3$, 98
$FV(\varphi)$, 19
$\mathcal{FOUR}$, 24
$\mathcal{THREE}$, 24
$\mathcal{TWO}$, 24
$\mathcal{D}$, 19
$\mathcal{H}^{aggr}$, 60
$\mathcal{L}^{aggr}$, 42
Σ, 18
$[\![t]\!]$, 19

L^2, 23
L^c, 23
$\wp(X)$, 13
$\bigvee S$, 14
$\bigwedge S$, 14
$\bot$, 14
$\leq^S$, 72
$\leq_p$, 23
$\leq_t$, 23
$\max(S)$, 14
$\min(S)$, 14
$\top$, 14
$\mathrm{fp}(F)$, 15
$\mathrm{gfp}(F)$, 15
$\mathrm{lfp}(F)$, 15
$\mathrm{post}(F)$, 15
$\mathrm{pre}(F)$, 15
$KK(A)$, 25
$ST(A)$, 26
$WF(A)$, 26

$\mathcal{M}(D)$, 36
$\mathcal{FM}(D)$, 36
$|M|$, 36
$M_1 \uplus M_2$, 36
M^S, 36
M^+, 36
M^-, 36
R_P, 38
$\mathrm{R}_\subseteq$, 38
$\mathrm{R}^\uparrow$, 40

C_A, 26
S_A, 26
S_A^c, 27
T_P, 22
T_P^{aggr}, 44
U_P, 33
U_P^{aggr}, 56
Φ_P, 32
Φ_P^{aggr}, 60

$\mathcal{I}$, 22
Π, 21
$base_{\mathcal{D}}(\Pi)$, 21
$comp(P)$, 22
$fold_r^\psi(P)$, 97
$ground(P)$, 22

Index

aggregate
  atom, 41
    with finite multisets, 61
  function, 36
    continuous, 87
    incremental, 113
  program, 44
    (weakly) stratified, 47
    definite, 46
  relation, 36
    anti-monotone, 39
    approximating, 59
    derived, 38
    monotone, 39
    monotone completion of, 40
    subset, 38
  signature, 41
  structure, 42
antichain, 13
approximating operator, 25
  ultimate, 28
approximation, 24
  exact, 24

bilattice, 23

chain, 13
closure ordinal, 16
consistent
  approximating operator, 27
  approximation, 24
  stable operator, 27
constraint, 20
constraint domain, 20

fixpoint, 15
  Kripke-Kleene, 25
  standard, 31
  well-founded, 26
folding, 97

infimum, 14

lambda expression, 41

multiset, 36
  finite, 36

negative aggregate formula, 45

order
  partial, 13
  precision, $\leq_p$, 23
  Smyth, $\leq^S$, 72
  total, 13
  truth, $\leq_t$, 23
  well-founded, 14
  well-order, 14
over-estimate, 24

positive aggregate formula, 45
post-fixpoint, 15
powerset, 13
pre-fixpoint, 15

set expression, 41
signature, 18
stable operator, 26
  consistent, 27
  exact, 26

lower, 27
upper, 27
standard model, 48
stratification, 30
of an aggregate program, 47
structure, 19
supported model, 23
supremum, 14

three-valued
interpretation, 31
multiset, 59
relation, 25
set, 25
stable models, 60
truth function, 32
truth function, 19

ultimate
approximating aggregate, 86
approximating operator, 28
stable model, 56
three-valued stable models, 56
well-founded model, 56
under-estimate, 24

valuation function, 19
of set expressions, 59
variable assignment, 19

Curriculum Vitae

Nikolay Pelov was born on 19 November 1974 in Sofia, Bulgaria. He finished the Sofia High School of Mathematics in 1992. He obtained a Master degree in Computer Science from the University of Sofia in 1997. His master thesis is "Metrics on terms and clauses and application in inductive machine learning" (in Bulgarian) under the supervision of Prof. Zdravko Markov. From 1998 he is a pre-doctoral and from 1999 he is a doctoral student at the Katholieke Universiteit Leuven, Belgium under the supervision of Prof. Maurice Bruynooghe and Prof. Marc Denecker.

He is a co-author of 6 papers published at international conferences.