Copyright

by

Paolo Ferraris

2007

The Dissertation Committee for Paolo Ferraris
certifies that this is the approved version of the following dissertation:

Expressiveness of Answer Set Languages

Committee:

____
Vladimir Lifschitz, Supervisor

____
Anna Gál

____
Warren A. Hunt, Jr.

____
Nicola Leone

____
Vijaya Ramachandran

Expressiveness of Answer Set Languages

by

Paolo Ferraris, B.S.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

August 2007

To my Parents

Acknowledgments

Studying for a PhD — and finally achieving it — has been a wonderful experience. I am usually not good at expressing gratitude, but I do really need to thank Vladimir Lifschitz for his advising, for his encouragements in starting this “adventure”, and for the freedom that he left me in doing the research that I wanted.

The experience of living for six years in another country is something that can be hardly described. Beside the initial difficulties, what you learn by living in a different environment, with different rules and different culture is priceless. This is especially true in a multi-cultural country such as the United States, and especially in a university with people from all parts of the world. The city of Austin has been a wonderful place to live, and I will miss it.

I need to thank some other people for my choice of studying in the United States. First of all, my *Laurea* Thesis supervisor Enrico Giunchiglia, who introduced me to research and who was the first of suggesting me the idea of a PhD in the United States. Professor Massimo Ancona, coach of the University of Genova for the ACM programming contests, also supported me in this choice. But the person who convinced me was Vittorio Badalassi, a friend from high school, with his stories as a PhD student in Santa Barbara, CA.

I need to thank all the people I did research with in these six years. Beside Vladimir Lifschitz, they are Yuliya Babovich/Lierler, Pedro Cabalar, Semra Doğandağ, Esra Erdem, Selim Erdoğan, Joohyung Lee and Wanwan Ren. With

most of them I also shared the life as a PhD student, with its highs and lows.

Participating in the world finals of the ACM programming contest has been a great and rewarding experience. This was possible only by having smart teammates such as Brian Peters, Andrew Hudson and Jeffrey Yasskin and great coaches such as Jean-Philippe "JP" Martin and Douglas Van Wieren. Thanks also go to the participants from other teams, who also helped us preparing for the contests.

I want to say Ciao to all the people that I have met at La Dolce Vita on Thursday nights. You are too many to be listed here, but I'd like to remember, in particular, our "presidentessa" Sylvia Leon Guerrero, Anthony Palomba, Paolo Bientinesi and Stefano e Chiara Masini.

I want to spend a few words to my Italian friends. Life evolves and priorities change. And distance makes things more difficult. I am really happy to know that I still have friends in Italy: (in no particular order) Manuela Marconi, Renato Cattani, Marco Lauciello, Chiara Iperti, Chiara Tixi, Paolo Predonzani, Tiziana Parodi and other people still in email contact and others that I may forget right now. Thanks a lot.

Le mie ultime parole — le più importanti, in italiano — le ho riservate ai miei genitori. Mi hanno sempre supportato nelle mie scelte, anche in una scelta difficile — soprattutto per loro — come quella di vivere a più di 8000 chilometri di distanza da loro. Molti miei amici hanno avuto le loro aspirazioni frenate dall'inconscio "egoismo" dei propri genitori, ma questo non e‘ stato il mio caso. Se ho ottenuto questo risultato è anche per merito loro. Non ci sono e non ci saranno mai parole sufficienti per esprimere la mia gratitudine.

(I have reserved the last words — the most important ones, in Italian — to my parents. They always supported me in my choices, even in a difficult choice — especially for them — such as living 5000+ miles away from them. Several friends of mine had their aspirations blocked by the unconscious "egoism" of their parents,

but this was not my case. I have achieved this result also because of them. There aren't and there will never be words that are sufficient to express my gratitude.)

PAOLO FERRARIS

The University of Texas at Austin
August 2007

Expressiveness of Answer Set Languages

Publication No. ______________

Paolo Ferraris, Ph.D.
The University of Texas at Austin, 2007

Supervisor: Vladimir Lifschitz

Answer set programming (ASP) is a form of declarative programming oriented towards difficult combinatorial search problems. It has been applied, for instance, to plan generation and product configuration problems in artificial intelligence and to graph-theoretic problems arising in VLSI design and in historical linguistics. Syntactically, ASP programs look like Prolog programs, but the computational mechanisms used in ASP are different: they are based on the ideas that have led to the development of fast satisfiability solvers for propositional logic.

ASP is based on the answer set/stable model semantics for logic programs, originally intended as a specification for query answering in Prolog. From the original definition of 1988, the semantics was independently extended by different research groups to more expressive kinds of programs, with syntax and semantics that are incompatible with each other. In this thesis we study how the various extensions are related to each other. In order to do that, we propose another definition of an answer set. This definition has three main characteristics: (i) it is very simple, (ii) its syntax is more general than the usual concept of a logic program, and (iii) strong theoretical tools can be used to reason on it. About (ii), we show that

our syntax allows constructs defined in many other extensions of the answer sets semantics. This fact, together with (iii), allows us to study the expressiveness of those constructs. We also compare the answer set semantics with another important formalism developed by Norm McCain and Hudson Turner, called causal logic.

Table of Contents

Acknowledgments **v**

Abstract **viii**

List of Figures **xv**

Chapter 1 Introduction **1**

Chapter 2 History **6**

2.1 The Origins of Answer Set Programming 6

2.1.1 The Answer Set Semantics for Traditional Programs 6

2.1.2 Relationship with Prolog 8

2.1.3 The Answer Set Programming Paradigm 8

2.2 Extensions to the Traditional Syntax 14

2.2.1 Propositional Extensions 15

2.2.2 Programs with Weight Constraints 17

2.2.3 Aggregates 19

2.3 Strong Equivalence 20

Chapter 3 Background **23**

3.1 Semantics of Programs with Nested Expressions 23

3.2 Semantics of Programs with Weight Constraints 25

3.3 PDB-aggregates 29
3.4 FLP-aggregates 31
3.5 Logic of Here-and-There 34
3.6 Equilibrium Logic 36
3.7 Proving Strong Equivalence 37

Chapter 4 Weight Constraints as Nested Expressions **39**
4.1 A Useful Abbreviation 41
4.2 Translations 42
4.2.1 Basic Translation 42
4.2.2 Nondisjunctive Translation 45
4.3 Strong Equivalence of Programs with Weight Constraints 46
4.4 Proof of Theorem 1 49

Chapter 5 Programs with Weight Constraints as Traditional Programs **55**
5.1 Eliminating Nested Expressions 56
5.2 Removing the Weights 62
5.2.1 Simplifying the Syntax of Weight Constraints 62
5.2.2 The Procedure 63
5.3 Proofs 65
5.3.1 Two Lemmas on Programs with Nested Expressions 65
5.3.2 Proof of Theorem 2 67
5.3.3 Proof of Theorem 3(b) 72
5.3.4 Proof of Theorem 3(a) 74

Chapter 6 Answer Sets for Propositional Theories **82**
6.1 Formulas, reducts and answer sets 83

6.2 Relationship to equilibrium logic and to the traditional definition of reduct 84
6.3 Properties of propositional theories 86
6.4 Computational complexity 88
6.5 Proofs 89
6.5.1 Proofs of Theorem 4 and Proposition 7 89
6.5.2 Proofs of Propositions 8–10 91
6.5.3 Proofs of Propositions 11 and 13 94
6.5.4 Proof of Proposition 12 97
6.5.5 Proof of Proposition 14 100

Chapter 7 A New Definition of an Aggregate **102**

7.1 Representing Aggregates 103
7.2 Aggregates as Propositional Formulas 105
7.3 Monotone Aggregates 106
7.4 Example 108
7.5 Computational Complexity 109
7.6 Other Formalisms 111
7.6.1 Programs with weight constraints 111
7.6.2 PDB-aggregates 112
7.6.3 FLP-aggregates 113
7.7 Proofs 113
7.7.1 Proof of Propositions 18 and 19 119
7.7.2 Proof of Theorem 5 121
7.7.3 Proof of Theorem 6 123
7.7.4 Proof of Proposition 20 124

Chapter 8 Modular Translations and Strong Equivalence **127**

8.1 Modular Transformations and Strong Equivalence 130

8.2 Applications: Negational Disjunctive Rules 132

8.3 Applications: Cardinality Constraints 134

8.4 Proofs 136

8.4.1 Properties of Strong Equivalence 136

8.4.2 Proof of Theorem 7 137

8.4.3 Proofs of Propositions 21 and 22 140

8.4.4 Proof of Theorem 8 141

8.4.5 Definition of a Tight Program 142

8.4.6 Proofs of Propositions 23 and 24 143

8.4.7 Proofs of Propositions 25–27 144

Chapter 9 Causal Theories as Logic Programs **147**

9.1 Introduction to Causal Theories 147

9.2 Syntax and Semantics of Causal Theories 149

9.3 Computational Methods 151

9.4 Almost Definite Causal Theories 153

9.5 Examples 155

9.5.1 Transitive Closure 155

9.5.2 Two Gears 159

9.6 Translation of causal theories in clausal form 161

9.7 Reducing the Size of the Translation of Causal Theories in Clausal Form 165

9.8 Clausifying a Causal Theory 167

9.9 Related work 169

9.10 Proofs 171

9.10.1 Proof of Theorem 9 171

9.10.2 Proof of Theorem 10 175
9.10.3 Proof of Proposition 28 177
9.10.4 Proof of Theorems 11 and 12 178
9.10.5 Proof of Theorem 13 186

Chapter 10 Conclusions **188**

Bibliography **190**

Vita **202**

List of Figures

1.1 Evolution of answer set languages. 2

1.2 New extensions to the answer set semantics. 4

2.1 An hitori puzzle and its solution 9

2.2 Encoding of the sample hitori puzzle 9

2.3 Rules that solve hitori puzzles 10

5.1 A translation that eliminates weight constraints in favor of cardinality constraints 63

7.1 A polynomial-time algorithm to find minimal models of special kinds of theories 120

8.1 A classification of logic programs under the answer set semantics. Here *a*, *b*, *c*, *d* stand for propositional atoms. *F*, *G* stand for nested expressions without classical negation (that is, expressions formed from atoms, $\top$ and $\bot$, using conjunction (,), disjunction (;) and negation as failure (*not*), see Section 2.2.1), and *H* stands for an arbitrary propositional formula. 128

9.1 Translation of causal theory (9.14)–(9.18). 157

9.2 Robby's apartment is a 3×3 grid, with a door between every pair of adjacent rooms. Initially Robby is in the middle, and all doors are locked. The goal of making every room accessible from every other can be achieved by unlocking 8 doors, and the robot will have to move to other rooms in the process. 158
9.3 Two gears domain. 159
9.4 Translation of the two gears domain. 160

Chapter 1

Introduction

Answer set programming (ASP) is a form of declarative programming oriented towards difficult combinatorial search problems. It has been applied, for instance, to plan generation and product configuration problems in artificial intelligence and to graph-theoretic problems arising in VLSI design and in historical linguistics. Syntactically, ASP programs look like Prolog programs, but the computational mechanisms used in ASP are different: they are based on the ideas that have led to the development of fast satisfiability solvers for propositional logic. ASP has emerged from interaction between two lines of research—on the semantics of negation in logic programming and on applications of satisfiability solvers to search problems. It was identified as a new programming paradigm in 1999.

ASP is based on the answer set (stable model) semantics for logic programs, originally intended as a specification for query answering in Prolog. The answer set semantics defines, for each logic program, a collection of sets of atoms. Each of these set is called an answer set for the logic program. Figure 1.1 shows how the Prolog-like syntax used in the original 1988 definition of an answer set has been extended by different research groups over the years, for different purposes. The left side of the figure traces the process of extending the traditional syntax towards expressions

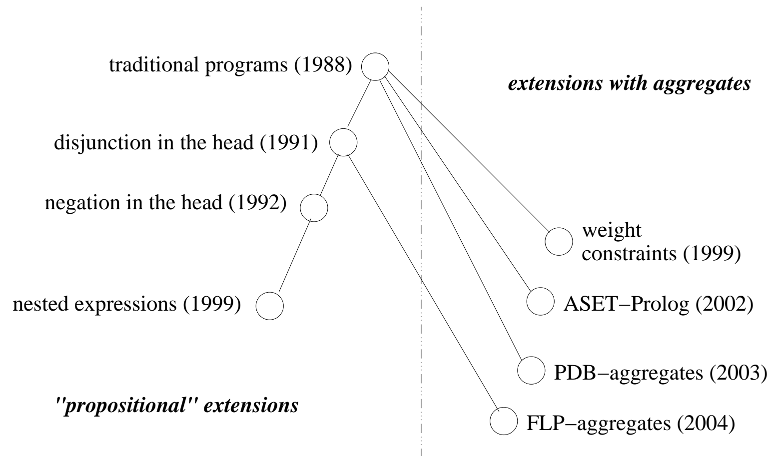

Figure 1.1: Evolution of answer set languages.

more and more similar to arbitrary propositional formulas. To a large degree, this was motivated by the desire to make the syntax of logic programs more uniform and elegant. On the right side, we see the introduction of the concept of an aggregate in logic programs. Aggregates are motivated by applications of ASP. They allow us, for instance, to talk about the number of atoms in a set that are true. In the figure, we mention three proposals of this kind that have been at least partially implemented. The four definitions differ from each other both syntactically and semantically. One reason for that is the lack of a common understanding of what an aggregate should be.

Part of this dissertation is devoted to comparing programs with weight constraints with the formalisms on the left side of Fig 1.1. The fact that this is possible is not surprising as, in case of PDB-aggregates, a relationship is well-known: the semantics of programs with PDB-aggregates is defined in terms of a translation into traditional programs.

First of all, we will show that weight constraints can be seen as abbreviations for nested expressions. Our theorem shows that if we replace, in any program with weight constraints, each weight constraint with a corresponding nested expression,

we obtain a program with nested expressions that has exactly the same answer sets of the original program. This also allows us to study properties of programs with weight constraints (such as strong equivalence, reviewed later in this introduction) using mathematical tools developed for programs with nested expressions.

We will also show how to rewrite a program with weight constraints as a traditional program. This procedure is similar to the procedure used in polynomial time clausifications of propositional formulas; the "signature" of the language is extended, the answer sets of the output program may contain auxiliary atoms, and the translation can be computed in polynomial-time in most cases of practical interest. Finally, we show that this conversion can always be done polynomially, by a more complicated procedure.

We will prove that nested expressions are not expressive enough to represent FLP-aggregates. In fact, it is generally not possible to replace FLP-aggregates with nested expressions without changing the answer sets, as we did for programs with weight constraints. This fact led us to extend the syntax of programs with nested expressions.

We will define the concept of an answer set for a "propositional theory". Syntactically, we discarded the idea that a "program" consists of "rules", by allowing arbitrary "propositional formulas" in it. We also propose a new definition of an aggregate inside propositional formulas. (The picture of the evolution of answer set semantics can be updated as in Figure 1.2.) For it we propose two equivalent semantics: one of them considers an aggregate as a primitive construct of the syntax, and the other as an abbreviation for a propositional formula. The first semantics is important computational-wise, the second because there are several theorems about propositional formulas that can be used for formulas with aggregates as well (see below). The new definition of an aggregate is more general than the definition of weight constraints and of PDB- and FLP-aggregates. This, for instance, allows us to

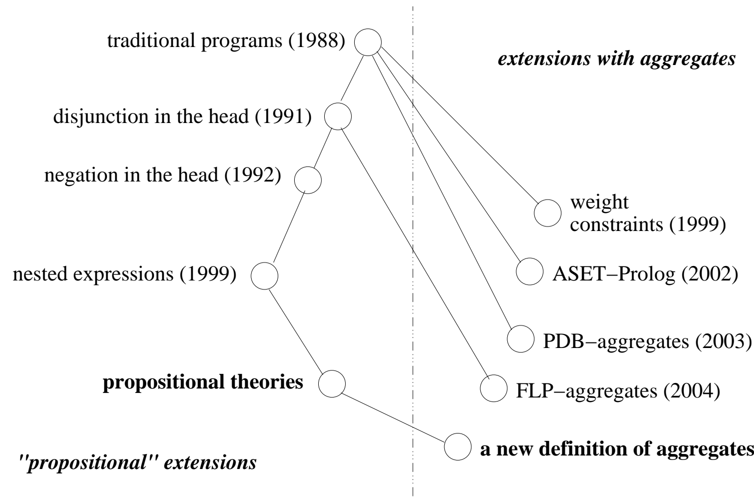

Figure 1.2: New extensions to the answer set semantics.

see under what conditions a weight constraint and an FLP-aggregate with the same intuitive meaning are actually equivalent to each other. Moreover, our definition of an aggregate seems to have the most intuitively correct properties.

Our definition of an answer set for propositional theories has other interesting properties. For instance, it is closely related to another formalism called equilibrium logic [Pearce, 1997]. This allows us to apply mathematical theorems about equilibrium logic to propositional theories and vice versa.

An important concept in the theory of logic programs is strong equivalence. Two logic programs are said to be strongly equivalent if they have the same answer sets even after we append any third logic program to both of them [Lifschitz *et al.*, 2001]. Strong equivalence between propositional theories can be defined similarly. We propose a method to check strong equivalence between propositional theories, similar to the one from [Turner, 2003] for logic programs, but simpler. We will extend other important theorems about programs with nested expressions to propositional theories as well.

We will also show that strong equivalence plays an important role if we

want to translate logic programs (or propositional theories) in one language into another, in a modular way (i.e., so that each rule/formula in the first language is independently replaced by a set of rules/formulas in the second language). In fact, it turns out that in most cases a transformation of this kind is guaranteed to preserve the answer sets if and only if each rule/formula is replaced by a strongly equivalent set of rules/formulas. We will use this fact to compare the expressiveness — in terms of modular translations — of some of the languages of Figure 1.1 and some of their subclasses, verifying if strongly equivalent transformations exist between them. As there are ways to check strong equivalence, we will use the above property to study the expressiveness of languages in terms of the presence of modular transformations.

Finally, we compare the expressiveness of answer set languages with another important formalism: causal logic [McCain and Turner, 1997], [Giunchiglia *et al.*, 2004a]. Causal logic defines the models of "causal theories", and has been used to encode several domains involving actions and their effects. A method of translating causal theories into logic programs was known for causal theories of a very simple form. We show how we can translate every causal theory into a logic program that is not much larger that the original causal theory.

After a historical overview in Chapter 2 and the necessary background information in Chapter 3, Chapters 4 and 5 describe how we can translate programs with weight constraints into programs with nested expressions and traditional programs, respectively.

Next we discuss the definition of an answer set for propositional theories (Chapter 6), how we can represent aggregates under this new syntax, and the relationship with FLP-aggregates (Chapter 7).

Finally, in Chapter 8 we investigate modular translations between logic programs and the relationship with strong equivalence, and in Chapter 9 we show how we can translate causal theories into logic programs.

Chapter 2

History

2.1 The Origins of Answer Set Programming

2.1.1 The Answer Set Semantics for Traditional Programs

The answer set (stable model) semantics was defined in [Gelfond and Lifschitz, 1988] for logic programs with *rules* of the form

$$A_0 \leftarrow A_1, \ldots, A_m, not\ A_{m+1}, \ldots, not\ A_n \quad (2.1)$$

where $n \geq m \geq 0$ and $A_0, \ldots, A_n$ are propositional atoms. The symbol *not* is called *negation as failure*. The part before the arrow (A_0) is the *head* of the rule, while the part after the arrow is called its *body*. When the body is empty (i.e., $m = n = 0$), the rule is called a *fact* and it is usually identified with its head. We call rules of the form (2.1) *traditional*. A *traditional (logic) program* is a set of traditional rules. Several other kinds of rules will be introduced in Section 2.2.

The answer set semantics defines when a set of atoms is an answer set for a logic program. First answer sets are defined for programs without negation as failure, i.e., when each rule has the form

$$A_0 \leftarrow A_1, \ldots, A_m \quad (2.2)$$

(such programs are called *positive*). We say that a set X of atoms is *closed* under a positive program Π if, for each rule (2.2) in Π, $A_0 \in X$ whenever $A_1, \dots, A_m \in X$. It is easy to see that there is a unique minimal set of atoms closed under Π. That is, there is exactly one set X of atoms such that

- X is closed under Π, and
- every proper subset of X is not closed under Π.

This set is defined to be the only *answer set* for Π. The definition resembles the semantics for positive programs given in [van Emden and Kowalski, 1976].

To define the answer sets for a generic traditional program Π, we first define a positive program Π^X, called the *reduct* of Π relative to a set of atoms X: Π^X is obtained from Π by dropping

- each rule (2.1) such that $\{A_{m+1}, \dots, A_n\} \cap X \neq \emptyset$, and
- the part $not\ A_{m+1}, \dots, not\ A_n$ from every other rule (2.1).

For instance, the reduct of

$$\begin{aligned} &p \leftarrow not\ q \\ &q \leftarrow not\ r \end{aligned} \tag{2.3}$$

relative to $\{q\}$ is

$$q \tag{2.4}$$

A set X of atoms is an *answer set* for Π if X is the answer set for Π^X. For instance, $\{q\}$ is an answer set for program (2.3), because $\{q\}$ is an answer set for (2.4). It is easy to check that no other set of atoms is an answer set for (2.3).

From the point of view of computational complexity, the problem of the existence of an answer set for a traditional logic program is NP-complete [Marek and Truszczyński, 1991].

Set of rules are often described by "schematic rules" that involve variables. Each schematic rule can be seen as an abbreviation for rules without variables, obtained by replacing each variable with one of the constants occurring in the program. This process is called grounding. For instance, the expression

$$\begin{aligned} &p(X) \leftarrow q(X) \\ &q(a) \\ &r(b) \end{aligned} \tag{2.5}$$

stands for logic program

$$\begin{aligned} &p(a) \leftarrow q(a) \\ &p(b) \leftarrow q(b) \\ &q(a) \\ &r(b) \end{aligned} \tag{2.6}$$

whose only answer set is $\{p(a), q(a), r(b)\}$.

2.1.2 Relationship with Prolog

The answer set semantics was originally proposed as a semantics for Prolog. A well-written (unambiguous and consistent) Prolog program has — after grounding — a unique answer set. This answer set is the collection of all atoms that are answered "yes" by Prolog queries. For instance, if we write (2.5) as a Prolog program, the only atoms that are answered "yes" in queries are $p(a)$, $q(a)$ and $r(b)$; these are the three atoms in the only answer set for (2.6).

2.1.3 The Answer Set Programming Paradigm

The answer set programming paradigm was proposed in [Marek and Truszczyński, 1999] and [Niemelä, 1999]. A combinatorial search problem is encoded in a logic program so that the program's answer sets are the solutions of the problem. An example of a puzzle game that can be solved using answer set programming is hitori.

4	8	1	6	3	2	5	7
3	6	7	2	1	6	5	4
2	3	4	8	2	8	6	1
4	1	6	5	7	7	3	5
7	2	3	1	8	5	1	2
3	5	6	7	3	1	8	4
6	4	2	3	5	4	7	8
8	7	1	4	2	3	5	6

	8		6	3	2		7
3	6	7	2	1		5	4
	3	4		2	8	6	1
4	1		5	7		3	
7		3		8	5	1	2
	5	6	7		1	8	
6		2	3	5	4	7	8
8	7	1	4		3		6

Figure 2.1: An hitori puzzle and its solution

$$
\begin{array}{cccc}
a(1,1,4) & a(2,1,8) & \cdots & a(8,1,7) \\
a(1,2,3) & a(2,2,6) & \cdots & a(8,2,4) \\
\vdots & \vdots & & \vdots \\
a(1,8,8) & a(2,8,7) & \cdots & a(8,8,6)
\end{array}
$$

Figure 2.2: Encoding of the sample hitori puzzle

Figure 2.1 (from `http://en.wikipedia.org/wiki/Hitori`) shows a hitori puzzle and its solution. You are given a square grid (usually 8x8) filled with numbers, and the goal is to darken (cancel out) some of the cells in such a way that the following conditions are satisfied:

- among uncancelled cells, the same number cannot occur twice in the same row or column; (for instance, in Figure 2.1, a 4 and a 3 had to be cancelled from the first column)
- there cannot be two horizontal or vertical adjacent cells both canceled; (for instance, we cannot cancel out both cells (1,1) and (1,2))
- all undarkened cells are connected to each other, through horizontal and vertical connections.

$$
\begin{array}{rl}
1 & \{dark(X,Y)\} \leftarrow a(X,Y,V) \\
\\
2 & \leftarrow dark(X,Y), dark(X,Y+1) \\
3 & \leftarrow dark(X+1,Y), dark(X,Y) \\
\\
4 & \leftarrow not\ dark(X,Y), not\ dark(X,Y_1), Y < Y_1, a(X,Y,V), a(X,Y_1,V) \\
5 & \leftarrow not\ dark(X,Y), not\ dark(X_1,Y), X < X_1, a(X,Y,V), a(X_1,Y,V) \\
\\
6 & conn(X,Y,X,Y) \leftarrow not\ dark(X,Y), a(X,Y,V) \\
7 & conn(X,Y,X_1,Y_1) \leftarrow not\ dark(X,Y), conn(X,Y+1,X_1,Y_1), a(X,Y,V) \\
8 & conn(X,Y,X_1,Y_1) \leftarrow not\ dark(X,Y), conn(X,Y-1,X_1,Y_1), a(X,Y,V) \\
9 & conn(X,Y,X_1,Y_1) \leftarrow not\ dark(X,Y), conn(X+1,Y,X_1,Y_1), a(X,Y,V) \\
10 & conn(X,Y,X_1,Y_1) \leftarrow not\ dark(X,Y), conn(X-1,Y,X_1,Y_1), a(X,Y,V) \\
\\
11 & \leftarrow not\ dark(X,Y), not\ dark(X_1,Y_1), not\ conn(X,Y,X_1,Y_1), \\
 & \qquad\qquad a(X,Y,V), a(X_1,Y_1,V_1)
\end{array}
$$

Figure 2.3: Rules that solve hitori puzzles

In order to solve a hitori puzzle (or, in general, combinatorial search problems of this kind) in answer set programming we need

1. an encoding of the problem instance, and

2. a set of rules that "solve" the problem.

An encoding of the puzzle of Figure 2.1 is shown in Figure 2.2. It consists of facts of the form $a(X, Y, V)$, which means that the cell with coordinates (X, Y) $((1, 1)$ is the top-leftmost cell) has value V.

A set of rules that "solve" hitori puzzles is in Figure 2.3. The syntax is richer than the one of traditional programs. Functions such as the sum in line 2 are evaluated when rules are grounded. The relation symbol of line 4 tells us that, in the process of grounding, we use only values of variables such that $Y \leq Y_1$, and similarly for line 5.

The first line is called a "choice rule": it says that, for each cell (X, Y), we can either choose that $dark(X, Y)$ holds or not. More precisely, if n is the number of cells, it generates 2^n candidate answer sets, each of them containing a different collection of atoms of the form $dark(X, Y)$. Each of those "candidate" answer sets corresponds to a set of darkened cells. The goal of the remaining lines is to remove the answer sets that do not satisfy the three conditions in the definition of a valid hitori puzzle solution.

Line 2 is an example of a "constraint" [1]. It tells us that $dark(X, Y)$ and $dark(X, Y + 1)$ cannot both belong to an answer set. That is, two horizontally adjacent cells cannot be both darkened. Similarly, line 3 expresses the same concept for vertically adjacent cells. Lines 4 and 5 prohibits that two undarkened cells on the same column or row contain the same value.

To encode the last condition on a hitori puzzle solution — that all undarkened cells are connected — we introduce, in our program, a predicate expressing that two undarkened cells (X, Y) and (X_1, Y_1) are connected. We define this concept using atoms of the form $conn(X, Y, X_1, Y_1)$, by the following recursive definition:

- each undarkened cell is connected to itself (line 6), and
- if a cell adjacent to an undarkened cell (X, Y) is connected to (X_1, Y_1) then (X, Y) is connected to (X_1, Y_1) as well (lines 7–10).

Finally, line 11 imposes the condition that every two undarkened cells are connected to each other.

By merging the 11 lines above with any puzzle description as in Figure 2.2, we get a program whose answer sets is in a 1–1 correspondence to a solution to the puzzle. Indeed, each solution of the puzzle corresponds to the answer set that contains an atom of the form $dark(X, Y)$ iff (X, Y) is a darkened cell.

[1] Not to be confused with cardinality or weight constraints.

We can notice that, unlike most other programming languages (including Prolog and many implementations of constraint programming), answer set programming is completely declarative: in particular, the order of rules in a program and the order of elements in the body of each rule are completely irrelevant. Moreover, the semantics is well-defined and simple; this makes it easier to develop mathematical tools for proving the correctness of a logic program. Some of the tools discussed in this dissertation are strong equivalence and the splitting set theorem.

Answer set programming is possible because there are systems, called *answer set solvers*, that compute the answer sets of logic programs. An incomplete list of the currently developed answer set solvers is

- ASET-SOLVER [2] [Heidt, 2001]
- ASSAT [3] [Lin and Zhao, 2002]
- CMODELS [4] [Lierler and Maratea, 2004]
- CSMODELS [5] [Sabuncu *et al.*, 2004]
- DLV [6] [Eiter *et al.*, 1998]
- GNT [7] [Janhunen *et al.*, 2003]
- NoMoRe [8] [Anger *et al.*, 2002]
- SMODELS [9] [Niemelä and Simons, 2000]

[2] `http://www.cs.ttu.edu/~mellarko/aset.html` .
[3] `http://assat.cs.ust.hk/` .
[4] `http://www.cs.utexas.edu/users/tag/cmodels.html` .
[5] `http://www.ceng.metu.edu.tr/~orkunt/csmodels/` .
[6] `http://www.dbai.tuwien.ac.at/proj/dlv/` .
[7] `http://www.tcs.hut.fi/Software/gnt/` .
[8] `http://www.cs.uni-potsdam.de/~linke/nomore/` .
[9] `http://www.tcs.hut.fi/Software/smodels/` .

Answer set programming is similar to the use of satisfiability to solve combinatorial search problems: in that approach, a problem is first encoded as a propositional formula whose models (interpretations that satisfy the formula) correspond to the solutions of the problem ([Kautz and Selman, 1992]). Then a satisfiability solver is used to find such models. Both formalisms are declarative, and answer sets can be seen as truth assignments to atoms just like models in propositional logic. Moreover, the computational mechanism of answer set solvers doesn't rely on the definition of an answer set directly but it is similar to the one used in satisfiability solvers.

On the other hand, answer set programming is different from satisfiability in a number of ways.

- The language of answer set programming allows the use of variables. The language accepted by satisfiability solvers (propositional formulas, usually clauses) doesn't allow variables, and higher level languages that can express variables are generally domain-dependent and not compatible with each other.

- We can easily express recursive definitions, such as the one of *conn* in the hitori example, in answer set programming. Recursive definitions are usually difficult to express in other formalisms. For instance, it is commonly believed the definition of *conn* in Figure 2.3 can be expressed in classical propositional logic — in view of [Spira, 1971], under some commonly believed conjecture in computational complexity, and without the use of auxiliary atoms — only with an exponentially large formula. On the other hand, after the elimination of variables, lines 6-10 of Figure 2.3 turn into a polynomially large set of rules.

- It is easy to express, in answer set programming, concepts such as defaults ("a proposition holds unless there is a reason for not being true") and commonsense inertia ("an object doesn't change its properties unless there is a reason for such a change"). For instance, we can express that a door with a spring is

normally closed, or that a piece on a chess-board remains in the same position unless it moves or its position is taken by an opponent piece. Those concepts are important, for instance, in commonsense reasoning and in planning.

Answer set programming has been used to solve combinatorial search problems in various fields, such as planning [10] [Dimopoulos *et al.*, 1997; Lifschitz, 1999], diagnosis [Eiter *et al.*, 1999; Gelfond and Galloway, 2001], model checking [Liu *et al.*, 1998; Heljanko and Niemelä, 2001], reachability analysis [Heljanko, 1999], product configuration [Soininen and Niemelä, 1998], dynamic constraint satisfaction [Soininen *et al.*, 1999], logical cryptanalysis [Hietalahti *et al.*, 2000], network inhibition analysis [Aura *et al.*, 2000], workflow specification [Trajcevski *et al.*, 2000; Koksal *et al.*, 2001], learning [Sakama, 2001], reasoning about policies [Son and Lobo, 2001], circuit design [Balduccini *et al.*, 2000; Balduccini *et al.*, 2001], wire routing problems [Erdem *et al.*, 2000], phylogeny reconstruction problems [Erdem *et al.*, 2003], query answering [Baral *et al.*, 2004], puzzle generation [Truszczynski *et al.*, 2006], data mining [Ruffolo *et al.*, 2006] and spacial reasoning [Cabalar and Santos, 2006].

2.2 Extensions to the Traditional Syntax

In the introduction, in Figure 1.1, we showed that the original syntax of the answer set semantics has been extended several times. In this section we present the syntax of the propositional extensions and of programs with weight constraints and the concept of an aggregate. The semantics of weight constraints, as well as the syntax and semantics of PDB- and FLP-aggregates are given in the next chapter.

[10] The idea of relating planning to answer sets was first proposed in [Subrahmanian and Zaniolo, 1995].

2.2.1 Propositional Extensions

The first important generalizations of traditional rules were given in [Gelfond and Lifschitz, 1991]. An alternative negation symbol $\neg$ (classical negation) was introduced. A *literal* is either an atom A or $\neg A$. A rule, as defined in [Gelfond and Lifschitz, 1988], is extended to have the form

$$L_0 \leftarrow L_1, \ldots, L_m, \mathit{not}\ L_{m+1}, \ldots, \mathit{not}\ L_n \tag{2.7}$$

where $n \geq m \geq 0$ and $L_0, \ldots, L_n$ are literals. Even in presence of classical negation, we say that a rule of this kind is traditional. The concept of classical negation in logic programs is "orthogonal" to the other syntactical extensions of a logic program that we review, in the sense that classical negation can be introduced in all such extensions. (In our definitions, unless otherwise specified, we always allow it.) What changes, in the presence of classical negation, is the basic concept of an answer set: if classical negation is not allowed, an answer set is a collection of atoms: otherwise, it is a consistent set of literals, i.e., a set of literals that doesn't include both A and $\neg A$ for the same atom A.

Another important extension proposed in [Gelfond and Lifschitz, 1991] allows the head of each rule to be the disjunction (represented by a semicolon) of several atoms. A rule has the form

$$L_1; \ldots; L_p \leftarrow L_{p+1}, \ldots, L_m, \mathit{not}\ L_{m+1}, \ldots, \mathit{not}\ L_n \tag{2.8}$$

where $n \geq m \geq p \geq 0$ and $L_1, \ldots, L_n$ are literals. Rules of this kind are important from two points of view. First of all, when we allow $p > 1$, the problem of the existence of an answer set for a program moves up in the polynomial hierarchy from NP to Σ_2^P [Eiter and Gottlob, 1993, Corollary 3.8]. Second, when $p = 0$, rule (2.8), which is usually written as

$$\leftarrow L_1, \ldots, L_m, \mathit{not}\ L_{m+1}, \ldots, \mathit{not}\ L_n, \tag{2.9}$$

is called a *constraint*, [11] and intuitively it imposes the requirement that at least one of $L_1, \ldots, L_m$, $not\ L_{m+1}, \ldots, not\ L_n$ be false. Lines 2–5 and 10 of Figure 2.3 are examples of rules of such a kind.

Negation as failure was first allowed in the head of a rule in [Lifschitz and Woo, 1992]: literals in the head can be preceded by the symbol *not*. As a result of this extension, answer sets for a program can be subsets of one another: for instance, the answer sets for the single rule program

$$p; not\ p \tag{2.10}$$

are $\emptyset$ and $\{p\}$. No program in the sense of [Gelfond and Lifschitz, 1991] has both $\emptyset$ and $\{p\}$ as its answer sets. One of the results of the research presented in Chapter 4 shows that there is a close relationship between choice rules, mentioned in Section 2.1.3 above, and rules with negation as failure in the head. For instance, the first line of Figure 2.3 can be alternatively written as

$$dark(X,Y); not\ dark(X,Y) \leftarrow a(X,Y,V).$$

Finally, programs with nested expressions were introduced in [Lifschitz *et al.*, 1999]. *Nested expressions* are built from literals and the symbols $\bot$ ("false") and $\top$ ("true") using the unary connective *not* and the binary connectives , (conjunction) and ; (disjunction). An example of a nested expression is

$$p, not(q; r).$$

A *rule with nested expressions* has the form

$$Head \leftarrow Body \tag{2.11}$$

where both *Body* and *Head* are nested expressions. The rule $\leftarrow Body$ is a shorthand for $\bot \leftarrow Body$, and the nested expression *Head* stands for rule $Head \leftarrow \top$. It is easy

[11] not to be confused with weight and cardinality constraints.

to see that rules with nested expressions are a generalization of all kinds of rules described so far.

A *program with nested expressions* is any set of rules with nested expressions.

2.2.2 Programs with Weight Constraints

Version 2.0 of answer set solver SMODELS supported a new construct, called weight constraints. Their syntax and semantics were first described in [Niemelä *et al.*, 1999]. Our presentation mostly follows [Niemelä and Simons, 2000]. A *rule element* is a literal (*positive* rule element) or a literal prefixed with *not* (*negative* rule element). A *weight constraint* is an expression of the form

$$L \leq \{c_1 = w_1, \ldots, c_m = w_m\} \leq U \tag{2.12}$$

where

- each of L, U is (a symbol for) a real number or one of the symbols $-\infty$, $+\infty$,
- $c_1, \ldots, c_m$ $(m \geq 0)$ are rule elements, and
- $w_1, \ldots, w_m$ are real numbers ("weights").

The intuitive meaning of (2.12) is that the sum of the weights w_i for all the c_i that are true is not lower than L and not greater than U. The part $L \leq$ can be omitted if $L = -\infty$; the part $\leq U$ can be omitted if $U = +\infty$. A *rule with weight constraints* is an expression of the form

$$C_0 \leftarrow C_1, \ldots, C_n \tag{2.13}$$

where $C_0, \ldots, C_n$ $(n \geq 0)$ are weight constraints. We call the rule elements of C_0 the *head elements* of rule (2.13).

Finally, a *program with weight constraints* is a set of rules with weight constraints.[12]

[12]In [Niemelä and Simons, 2000], programs are not allowed to contain classical negation. But classical negation is allowed in the input files of the current version of SMODELS.

This syntax becomes a generalization of traditional programs if we identify a rule element c with the weight constraint

$$1 \leq \{c = 1\}.$$

By

$$\leftarrow C_1, \ldots, C_n$$

we denote the rule

$$1 \leq \{\ \} \leftarrow C_1, \ldots, C_n.$$

A *cardinality constraint* is a weight constraint with all weights equal to 1. A cardinality constraint

$$L \leq \{c_1 = 1, \ldots, c_m = 1\} \leq U$$

can be abbreviated as

$$L \leq \{c_1, \ldots, c_m\} \leq U. \tag{2.14}$$

It becomes clear that the rules in Figure 2.3 are rules in the syntax of weight constraints. For instance, line 1 is, without abbreviations,

$$-\infty \leq \{dark(X, Y) = 1\} \leq +\infty \ \leftarrow \ 1 \leq \{a(X, Y, V) = 1\} \leq +\infty.$$

Intuitively, it seems possible to represent disjunctive programs also as programs with weight constraints. For instance, a disjunctive rule

$$l_1; \ldots; l_n \qquad\qquad (n > 0) \tag{2.15}$$

(intuitively: one of $l_1, \ldots, l_n$ must hold) seems to have the same meaning as the cardinality constraint

$$1 \leq \{l_1, \ldots, l_n\}. \tag{2.16}$$

However, this is not true. For instance, according to the definitions of answer sets in the next chapter, the answer sets for (2.15) are the singletons $\{l_1\}, \ldots, \{l_n\}$, while

the answer sets for (2.16) are arbitrary nonempty subsets of $\{l_1, \ldots, l_n\}$. Representing (2.15) as

$$1 \leq \{l_1, \ldots, l_n\} \leq 1 \tag{2.17}$$

is generally not adequate either. Indeed, if we append the two facts l_1, l_2 to (2.15) we get a program with the unique answer set $\{l_1, l_2\}$, while by appending the same facts to (2.17) we get a program that has no answer sets. The fact that programs with weight constraints don't generalize disjunctive programs has a theoretical justification: checking if a logic program with weight constraints has answer sets is an NP-complete problem, and not Σ_2^P as for disjunctive programs.

2.2.3 Aggregates

By a (ground) aggregate we understand an expression of the form

$$op\langle\{F_1 = w_1, \ldots, F_n = w_n\}\rangle \prec N \tag{2.18}$$

where

- *op* is (a symbol for) a function from multisets of $\mathcal{R}$ (real numbers) to $\mathcal{R} \cup \{-\infty, +\infty\}$ (such as sum, product, min, max, etc.),
- each of $F_1, \ldots, F_n$ is (a symbol for) some kind of "formula" (usually in the syntax of nested expressions), and $w_1, \ldots, w_n$ are (symbols for) real numbers ("weights"),
- $\prec$ is (a symbol for) a binary relation between real numbers, such as $\leq$ and $=$, and
- N is (a symbol for) a real number.

As an intuitive explanation of an aggregate, take the multiset W consisting of the weights w_i $(1 \leq i \leq n)$ such that F_i is "true". The aggregate is considered "true" if

$op(W) \prec N$. For example,

$$sum\langle\{p = 1, q = 1\}\rangle \neq 1.$$

intuitively expresses the condition that either both p are q are "true" or none of them is.

It is clear that a weight constraint of the form $L \leq S$ has the same intuitive meaning of aggregate $sum\langle S\rangle \geq L$, and that $S \leq U$ has the same intuitive meaning of aggregate $sum\langle S\rangle \leq U$.

The syntax and semantics of PDB- and FLP-aggregates are reviewed in Sections 3.3 and 3.4 respectively.

2.3 Strong Equivalence

In logic programming with the answer set semantics we distinguish between two kinds of equivalence between logic programs [13].

We say that two programs are *weakly equivalent* if they have the same answer sets. For instance, each of the one-rule programs

$$a \quad \text{and} \quad a \leftarrow not\ b$$

has a unique answer set: $\{a\}$, so they are weakly equivalent. However, if we append the rule b to each program then the answer sets don't remain the same for both programs: in the first case, we have the answer set $\{a, b\}$; in the second, $\{b\}$.

For this reason, another relationship between programs has been defined. We say that two programs with nested expressions Π_1 and Π_2 are *strongly equivalent* if, for any program Π with nested expressions, $\Pi_1 \cup \Pi$ and $\Pi_2 \cup \Pi$ have the same answer sets [Lifschitz *et al.*, 2001]. Strong equivalence implies weak equivalence: take Π to be empty.

[13] To be precise, other kinds of equivalences have been defined, but we don't discuss them in this thesis.

For instance, it can be proved that (2.10) is strongly equivalent to the rule

$$p \leftarrow \mathit{not}\ \mathit{not}\ p \tag{2.19}$$

Strong equivalence is an important tool for reasoning about answer sets, because it allows us to replace a part of a program with a strongly equivalent set of rules, with the guarantee that the answer sets for the whole program don't change. It is often used to simplify programs, or to rewrite them in a simpler syntax. For instance, a rule

$$F \leftarrow G; H$$

(F, G and H are nested expressions) is strongly equivalent to the pair of rules

$$F \leftarrow G$$
$$F \leftarrow H.$$

Similarly,

$$F, G \leftarrow H$$

is strongly equivalent to the pair of rules

$$F \leftarrow H$$
$$G \leftarrow H$$

The definition of strong equivalence and a condition characterizing when it holds were originally proposed in [Lifschitz *et al.*, 2001] for programs with nested expressions. In particular, in the definition, the third logic program Π can range over all programs with nested expressions. However, when we consider programs belonging to a subclass of programs with nested expressions — for instance, traditional programs — it would be natural to allow Π to range over such subclass of logic programs. It turns out, however, that the choice of the class of programs for which Π is taken is inessential. Indeed, [Lifschitz *et al.*, 2001] showed that, if any two programs with nested expressions are not strongly equivalent to each other, there is

always a counterexample Π that is a positive traditional program of a very simple form: the bodies of its rules consist of at most one element. Such programs, called *unary programs*, consists of facts and of rules of the form $l_1 \leftarrow l_2$, where l_1 and l_2 are literals. Consequently, whether two logic programs are strongly equivalent doesn't depend on the syntax of logic programs that we are considering, as long as all unary programs are allowed.

Several characterizations of strong equivalence exist: for programs with nested expressions, we will review in Section 3.5 the original one from [Lifschitz *et al.*, 2001] and another equivalent from [Turner, 2003]. This last paper covers also programs with weight constraints (in this case, Π ranges over the set of programs with weight constraints). Finally, [Turner, 2004] defines the concept of strong equivalence between causal theories, presented in Chapter 9. Later in this thesis we will present yet another characterization of strong equivalence, based on the definition of an answer set for propositional theories.

Chapter 3

Background

3.1 Semantics of Programs with Nested Expressions

The syntax of programs with nested expressions [Lifschitz *et al.*, 1999] is described in Section 2.2.1. The semantics of these programs is characterized by defining when a consistent set X of literals is an answer set for a program Π. As a preliminary step, we define when a consistent set X of literals *satisfies* a nested expression F (symbolically, $X \models F$), as follows:

- for a literal l, $X \models l$ if $l \in X$
- $X \models \top$
- $X \not\models \bot$
- $X \models (F, G)$ if $X \models F$ and $X \models G$
- $X \models (F; G)$ if $X \models F$ or $X \models G$
- $X \models \textit{not}\ F$ if $X \not\models F$.

We say that X *satisfies* a program Π (symbolically, $X \models \Pi$) if, for every rule (2.11) in Π, $X \models \textit{Head}$ whenever $X \models\ \textit{Body}$.

The *reduct*[1] F^X of a nested expression F with respect to a consistent set X of literals is defined recursively as follows:

- if F is a literal, $\top$ or $\bot$, then $F^X = F$
- $(F, G)^X = F^X, G^X$
- $(F; G)^X = F^X; G^X$
- $(not\ F)^X = \begin{cases} \bot\ , & \text{if } X \models F, \\ \top\ , & \text{otherwise.} \end{cases}$

The *reduct* Π^X of a program Π with respect to X is the set of rules

$$Head^X \leftarrow Body^X$$

for each rule (2.11) in Π. For instance, the reduct of (2.19) with respect to X is

$$p \leftarrow \top \tag{3.1}$$

if $p \in X$, and

$$p \leftarrow \bot \tag{3.2}$$

otherwise.

The concept of an answer set is defined first for programs not containing negation as failure: a consistent set X of literals is an *answer set* for such a program Π if X is a minimal set (relative to set inclusion) satisfying Π. For an arbitrary program Π, we say that X is an *answer set* for Π if X is an answer set for the reduct Π^X.

For instance, the reduct of (2.19) with respect to $\{p\}$ is (3.1), and $\{p\}$ is a minimal set satisfying (3.1); consequently, $\{p\}$ is an answer set for (2.19). On the other hand, the reduct of (2.19) with respect to $\emptyset$ is (3.2), and $\emptyset$ is a minimal set satisfying (3.2); consequently, $\emptyset$ is an answer set for (2.19) as well.

[1]This definition of reduct is the same as the one in [Lifschitz *et al.*, 2001], except that the condition $X \models F^X$ is replaced with $X \models F$. It is easy to check by structural induction that the two conditions are equivalent.

3.2 Semantics of Programs with Weight Constraints

The syntax of programs with weight constraints [Niemelä and Simons, 2000] is reviewed in Section 2.2.2. The semantics of such programs was defined for programs whose weight constraints contain positive weight only. Indeed, [Niemelä and Simons, 2000] proposed to eliminate weight constraints with negative weights as follows: consider any weight constraint $L \leq S \leq U$ and an element $l = w$ of S where w is negative: this element can be replaced by $not\ l = |w|$ if we add w to both L and U. For instance,

$$0 \leq \{p = 2, p = -1\} \tag{3.3}$$

can be rewritten as

$$1 \leq \{p = 2, not\ p = 1\}. \tag{3.4}$$

(Recall that U is $+\infty$, so adding 1 to it is irrelevant.) Similarly, $not\ l = w$ can be replaced by $l = |w|$ if we again add w to both L and U. However, we find this way of considering negative weights not completely satisfactory: (3.3) intuitively has the same meaning of

$$0 \leq \{p = 1\},$$

which is different from (3.4): indeed,

$$p \leftarrow 1 \leq \{p = 2, not\ p = 1\} \tag{3.5}$$

has no answer sets, while

$$p \leftarrow 0 \leq \{p = 1\} \tag{3.6}$$

has answer set $\emptyset$. We will see later, in Section 7.6.1, a different way of considering negative weights that solves this problem.

The semantics of programs with weight constraints with nonnegative weights uses the following auxiliary definitions. A consistent set X of literals *satisfies* a weight constraint (2.12) if the sum of the weights w_j for all j such that $X \models c_j$ is

not less than L and not greater than U. For instance, X satisfies the cardinality constraint

$$1 \leq \{a = 1, b = 1\} \leq 1 \tag{3.7}$$

iff X contains exactly one of the atoms a, b. About a program Ω with weight constraints we say that X *satisfies* Ω if, for every rule (2.13) in Ω, X satisfies C_0 whenever X satisfies $C_1, \ldots, C_n$. As in the case of nested expressions, we will use $\models$ to denote the satisfaction relation for both weight constraints and programs with weight constraints.

The next part of the semantics of weight constraints is the definition of the reduct for weight constraints of the form

$$L \leq \{c_1 = w_1, \ldots, c_m = w_m\}.$$

The *reduct* $(L \leq S)^X$ of a weight constraint $L \leq S$ with respect to a consistent set X of literals is the weight constraint $L^X \leq S'$, where

- S' is obtained from S by dropping all pairs $c = w$ such that c is negative, and
- L^X is L minus the sum of the weights w for all pairs $c = w$ in S such that c is negative and $X \models c$.

For instance, the reduct of the constraint

$$1 \leq \{not\ a = 3, not\ b = 2\}$$

relative to $\{a\}$ is

$$-1 \leq \{\ \}.$$

The *reduct* of a rule

$$L_0 \leq S_0 \leq U_0 \leftarrow L_1 \leq S_1 \leq U_1, \ldots, L_n \leq S_n \leq U_n \tag{3.8}$$

with respect to a consistent set X of literals is

- the set of rules of the form

$$l \leftarrow (L_1 \leq S_1)^X, \ldots, (L_n \leq S_n)^X$$

  where l is a positive head element of (3.8) such that $X \models l$, if, for every i $(1 \leq i \leq n)$, $X \models S_i \leq U_i$;

- the empty set, otherwise.

The *reduct* Ω^X of a program Ω with respect to X is the union of the reducts of the rules of Ω.

Consider, for example, the one-rule program

$$1 \leq \{a = 2\} \leq 2 \leftarrow 1 \leq \{not\ a = 3, not\ b = 2\} \leq 4. \tag{3.9}$$

Since the only head element of (3.9) is a, the reduct of this rule with respect to a set X of atoms is empty if $a \notin X$. Consider the case when $a \in X$. Since

$$X \models \{not\ a = 3, not\ b = 2\} \leq 4,$$

the reduct consists of one rule

$$a \leftarrow (1 \leq \{not\ a = 3, not\ b = 2\})^X.$$

It is clear from the definition of the reduct of a program above that every rule in a reduct satisfies two conditions:

- its head is a literal, and

- every member of its body has the form $L \leq S$ where S does not contain negative rule elements.

A rule satisfying these conditions is called a *Horn rule*. If a program Ω consists of Horn rules then there is a unique minimal set X of literals such that $X \models \Omega$. This set is called the *deductive closure* of Ω and denoted by $cl(\Omega)$.

Finally, a consistent set X of literals is an *answer set* for a program Ω if $X \models \Omega$ and $cl(\Omega^X) = X$.

To illustrate this definition, assume that Ω is (3.7). Set $\{a, b\}$ is not an answer set for Ω because it does not satisfy Ω. Let us check that every proper subset of $\{a, b\}$ is an answer set. Clearly, every such subset satisfies Ω. It remains to show that each of these sets is the deductive closure of the corresponding reduct of Ω.

- $\Omega^{\emptyset}$ is empty, so that $cl(\Omega^{\emptyset}) = \emptyset$.
- $\Omega^{\{a\}}$ consists of the single rule a, so that $cl(\Omega^{\{a\}}) = \{a\}$.
- $\Omega^{\{b\}}$ consists of the single rule b, so that $cl(\Omega^{\{b\}}) = \{b\}$.

To give another example, let Ω be (3.9). Set $\{b\}$ is not an answer set for Ω because it does not satisfy Ω. The other subsets of $\{a, b\}$ satisfy Ω. Consider the corresponding reducts.

- $\Omega^{\emptyset}$ is empty, so that $cl(\Omega^{\emptyset}) = \emptyset$.
- $\Omega^{\{a\}}$ is

  $$a \leftarrow -1 \leq \{\}.$$

  Consequently, $cl(\Omega^{\{a\}}) = \{a\}$.
- $\Omega^{\{a,b\}}$ is

  $$a \leftarrow 1 \leq \{\}$$

  Consequently, $cl(\Omega^{\{a,b\}}) = \emptyset \neq \{a, b\}$.

We conclude that the answer sets for (3.9) are $\emptyset$ and $\{a\}$.

3.3 PDB-aggregates

The semantics of aggregates that we call "PDB" has been invented by Pelov, Deneker and Bruynooghe (2003).[2] We review the syntax of program with PDB-aggregates using the notation from Section 2.2.3. We also allow classical negation. A *(ground) PDB-aggregate* is an expression of the form (2.18) where each of $F_1, \ldots, F_n$ is a rule element (that is, a literal possibly prefixed by *not*).

A *program with PDB-aggregates* is a set of rules of the form

$$l \leftarrow A_1, \ldots, A_m,$$

where $m \geq 0$, l is a literal and $A_1, \ldots, A_m$ are PDB-aggregates. Programs with PDB-aggregates can seen as a generalization of traditional programs if we encode, in the body of each rule, a rule element c as $sum\langle\{c = 1\}\rangle > 0$.

The semantics of [Pelov *et al.*, 2003] for programs with PDB-aggregates consists in a procedure that transforms programs with such aggregates into a traditional program. In this section we show how we can convert a program with PDB-aggregates into a program with nested expressions. It is actually not hard to see that the program with nested expressions that we obtain is strongly equivalent to the traditional program result of translation of PDB-aggregates proposed in [Pelov *et al.*, 2003].

For the following definition, we will use the following notation: for a rule element c, by $\overline{c}$ we denote l, if c has the form *not* l, and *not* c, otherwise. The translation Π_{tr} of a PDB-program Π is the result of replacing each PDB-aggregate A of the form

$$op\langle\{c_1 = w_1, \ldots, c_n = w_n\}\rangle \prec N$$

[2] A semantics for such aggregates was proposed in [Denecker *et al.*, 2001], based on the approximation theory [Denecker *et al.*, 2002]. But the first characterization of PDB-aggregates in terms of answer sets is from [Pelov *et al.*, 2003].

with the following nested expression A_{tr} (W_I stands for the multiset $\{w_i : i \in I\}$)

$$\mathop{\Large ;}_{I_1,I_2:I_1\subseteq I_2\subseteq\{1,\dots,n\} \text{ and for all } I \text{ such that } I_1\subseteq I\subseteq I_2,\ op(W_I)\prec N} G_{(I_1,I_2)}$$

where $G_{(I_1,I_2)}$ stands for

$$\mathop{\Large ,}_{i\in I} c_i, \mathop{\Large ,}_{i\in\{1,\dots,n\}\setminus I_2} \overline{c_i}.$$

The use of the "big comma" and the "big semicolon" in the formulas above to represent a multiple conjunction and a multiple disjunction is similar to the familiar use of $\bigwedge$ and $\bigvee$. In particular, the empty conjunction is understood as $\top$, and the empty disjunction as $\bot$.

For instance, for a PDB-aggregate $A = sum\langle\{p = -1, q = 1\}\rangle \geq 0$, if we take $F_1 = p$, $F_2 = q$ then the pairs (I_1, I_2) that "contribute" to the disjunction of A_{tr} are

$$(\emptyset,\emptyset) \qquad (\{2\},\{2\}) \qquad (\{1,2\},\{1,2\}) \qquad (\emptyset,\{2\}) \qquad (\{2\},\{1,2\}).$$

The corresponding nested expressions $G_{(I_1,I_2)}$ are

$$not\ p, not\ q \qquad q, not\ p \qquad p, q \qquad not\ p \qquad q$$

and their disjunction is equivalent, in the logic of here and there, to $not\ p; q$.

PDB-aggregates seem to have the same problems of weight constraints in case of negative weights. For instance, program

$$p \leftarrow sum\langle\{p = 2, p = -1\}\rangle \geq 0 \tag{3.10}$$

(the way of writing (3.5) using a PDB-aggregate) has no answer sets, while

$$p \leftarrow sum\langle\{p = 1\}\rangle \geq 0 \tag{3.11}$$

which is intuitively equivalent to A, has answer set $\emptyset$.

In addition to this, PDB aggregates seem to give some other unintuitive results when negation as failure occurs in an aggregate. Consider the following Π:

$$\begin{aligned} & p \leftarrow sum\langle\{q = 1\}\rangle < 1 \\ & q \leftarrow not\ p \end{aligned} \tag{3.12}$$

and Π':

$$\begin{aligned} & p \leftarrow sum\langle\{not\ p = 1\}\rangle < 1 \rightarrow p \\ & q \leftarrow not\ p \end{aligned} \tag{3.13}$$

Intuitively, the two programs should have the same answer sets. Indeed, the operation of replacing q with $\neg p$ in the first rule of Π should not affect the answer sets since the second rule "defines" q as $\neg p$: it is the only rule with q in the head. However, under the semantics of [Pelov *et al.*, 2003], Π has answer $\{p\}$ and Π' has answer set $\{q\}$ also.

3.4 FLP-aggregates

A semantics for aggregates that we call "FLP" has been invented by Faber, Leone and Pfeifer (2004). Here again we use the notation from Section 2.2.3, and we allow classical negation, not allowed in the usual definition.

An *FLP-aggregate* is an expression of the form (2.18) where each of $F_1, \dots, F_n$ is a conjunction $l_1, \dots, l_m$ of literals. A *program with FLP-aggregates* is a set of rules of the form

$$l_1; \dots; l_n \leftarrow A_1, \dots, A_m, not\ A_{m+1}, \dots, not\ A_p \tag{3.14}$$

where $n \geq 0, 0 \leq m \leq p$, $l_1, \dots, l_n$ are literals and $A_1, \dots, A_p$ are FLP-aggregates. Programs with FLP-aggregates can seen as a generalization of disjunctive programs if we encode, in the body of each rule, a literal l as $sum\langle\{l = 1\}\rangle > 0$.

The semantics of [Faber *et al.*, 2004] defines whether a consistent set of literals[3] is an answer set for a program with FLP-aggregates.

[3]Similarly to [Niemelä and Simons, 2000] for programs with weight constraints, in the original

The satisfaction of an aggregate A of the form

$$op\langle\{F_1 = w_1, \dots, F_n = w_n\}\rangle \prec N$$

by a consistent set X of literals is defined as follows. Consider the multiset W consisting of all numbers w_i ($i = 1, \dots, n$) such that $X \models l_i$. Set X *satisfies* A if $op(W) \prec N$. This catches the intuitive meaning of an aggregate. The definition of satisfaction for aggregates is extended to bodies of rules and to programs with FLP-aggregates in the same way as in the case of programs with nested expressions (see Section 3.1).

The *reduct* Π^X of a program Π with FLP-aggregates consists of the rules of the form (3.14) such that X satisfies its body. Set X is an answer set for Π if X is a minimal set satisfying Π^X.

For instance, the only answer set for the following FLP-program Π.

$$p \leftarrow sum\langle\{p = 2\}\rangle \geq 0$$

is the empty set. Indeed, since the empty set doesn't satisfy the aggregate, $\Pi^\emptyset = \emptyset$, which has $\emptyset$ as the unique minimal model; we can conclude that $\emptyset$ is an answer set for Π. On the other hand, $\Pi^{\{p\}} = \Pi$ because $\{p\}$ satisfies the aggregate in Π. Since $\{p\} \models \Pi$, $\{p\}$ is not a minimal model of $\Pi^{\{p\}}$ and then it is not an answer set for Π.

This definition of a reduct is different from the other definitions of reducts for traditional programs, with nested expressions and with weight constraints, in the sense that it may leave negative rule elements in the body of a rule. For instance, the reduct of $a \leftarrow not\, b$ relative to $\{a\}$ is, accordingly to the other definitions, essentially the fact a. In the definition of this section, the reduct doesn't modify the rule. On the other hand, this definition of an answer set is equivalent to the definition of an

syntax of [Faber *et al.*, 2004] classical negation doesn't occur, and the word "literal" is used there to denote an atom possibly prefixed by *not*, i.e. what we call "rule element". We don't need such concept since we view rule elements as abbreviations for FLP-aggregates: see Section 2.2.3.

answer set in the sense of [Gelfond and Lifschitz, 1991] and of [Lifschitz *et al.*, 1999] when applied to disjunctive programs.

FLP-aggregates probably are the best proposal for a definition of an aggregate, as they don't have the same problems as weight constraints and PDB-aggregates in the case of sums with negative weights. For instance, both (3.10) and (3.11) have answer set $\{p\}$.

Under the semantics of [Faber *et al.*, 2004], an expression of the form

$$not\ op\langle S\rangle \prec N$$

has the same meaning of

$$op\langle S\rangle \not\prec N,$$

so that negation can be eliminated from programs with FLP-aggregates. We have to say that this negation is not really "negation as failure". Consider the following programs Π:

$$p \leftarrow not\ sum\langle\{q = 1\}\rangle < 1$$
$$q \leftarrow sum\langle\{q = 1\}\rangle < 1$$

and Π':

$$p \leftarrow not\ q$$
$$q \leftarrow sum\langle\{q = 1\}\rangle < 1$$

If the negation in the first rule of Π is negation as failure then the two programs should have the same answer sets. Indeed, the operation of replacing $sum\langle\{q = 1\}\rangle < 1$ with q in the first rule of Π should be safe since the second rule "defines" q as $sum\langle\{q = 1\}\rangle < 1$: it is the only rule with q in the head. However, under the semantics of [Faber *et al.*, 2004], Π has answer $\{p\}$ only, while Π' has answer set $\{q\}$ also.

3.5 Logic of Here-and-There

In this section we introduce the logic of here-and-there, which is needed for defining equilibrium logic and for a characterization of strong equivalence.

A *propositional formula* is any combination of atoms formed using the connectives $\bot$ (false), $\vee$, $\wedge$, and $\rightarrow$. An expression of the form $\neg F$ stands for $F \rightarrow \bot$, $\top$ for $\bot \rightarrow \bot$ and $F \leftrightarrow G$ for $(F \rightarrow G) \wedge (G \rightarrow F)$. As usual, a *propositional theory* is a (possibly infinite) set of propositional formulas. We identify an interpretation in classical logic with the set of atoms satisfied by it. That means that for every interpretation X and any atom a, X satisfies a iff $a \in X$. Satisfaction of non-atomic formulas is defined recursively in terms of truth-tables, as usual in classical logic.

The logic of here-and-there was originally defined in [Heyting, 1930]. The semantics of the logic of here-and-there is defined as follows. An *HT-interpretation* is a pair (X, Y) of sets of atoms (respectively called "here" and "there") such that $X \subseteq Y$. Each HT-interpretation intuitively "assigns" one of three possible values to each atom: atoms in X are considered to be true, atoms not in Y are considered to be false, and the rest $(Y - X)$ are thought to be undefined. We say that an HT-interpretation (X, Y) is *total* when $X = Y$ (no undefined atoms). In this section and all the others where the logic of here-and-there is discussed we will drop the suffix "HT-" from "HT-interpretation".

We recursively define[4] when an interpretation (X, Y) *satisfies* a formula F, written $(X, Y) \models F$, as follows:

- for any atom a, $(X, Y) \models a$ if $a \in X$,
- $(X, Y) \not\models \bot$,

[4] We have slightly simplified the definition in comparison with the usual definition of satisfaction in the logic of here-and-there which is typically provided in terms of a Kripke structure as in intuitionistic logic, but under the assumption that it consists of only two worlds. It can be easily seen that both definitions are equivalent.

- $(X, Y) \models F \land G$ if $(X, Y) \models F$ and $(X, Y) \models G$,
- $(X, Y) \models F \lor G$ if $(X, Y) \models F$ or $(X, Y) \models G$,
- $(X, Y) \models F \rightarrow G$ if $(X, Y) \models F$ implies $(X, Y) \models G$, and $Y \models F \rightarrow G$.

Although we use the same symbol '$\models$' for satisfaction in logic programs, classical logic and the logic of here-and-there, this will not lead to ambiguity if we note with object serve as the operands of $\models$. It is clear, for instance, that the expression $Y \models F \rightarrow G$ in the last line of the definition above refers to satisfaction in the sense of classical logic.

For instance, formula $F = (p \rightarrow q) \rightarrow q$ is satisfied by $(\{q\}, \{q\})$. Indeed, first we notice that, in the "there" world, $\{q\} \models F$. It remains to notice that $(\{q\}, \{q\})$ satisfies the consequent q of F. It is also easy to check that interpretation, $(\emptyset, \{q\})$ is not a model of F, because $(\emptyset, \{q\}) \models p \rightarrow q$ but $(\emptyset, \{q\}) \not\models q$.

As usual, an interpretation is a *model* of a theory T if it satisfies all the formulas in T. Two formulas (theories) are *equivalent* if they have the same models.

Note that when the interpretation is total ($X{=}Y$), $(Y, Y) \models F$ simply collapses into classical satisfaction $Y \models F$. Another interesting property is that $(X, Y) \models F$ implies $(Y, Y) \models F$ (that is, $Y \models F$). Finally, using the definition of $\neg F$ as $F \rightarrow \bot$, it also follows that $(X, Y) \models \neg F$ iff $Y \models \neg F$.

Axiomatically, the logic of here-and-there is intermediate between intuitionistic and classical logic. A natural deduction system for intuitionistic logic can be obtained from the corresponding classical system [Bibel and Eder, 1993, Table 3] by dropping the law of the excluded middle

$$F \lor \neg F$$

from the list of postulates. The logic of here-and-there, on the other hand, is the result of replacing the excluded middle in the classical system with the weaker axiom

schema [De Jongh and Hendriks, 2003]:

$$F \vee (F \rightarrow G) \vee \neg G. \tag{3.15}$$

In addition to all intuitionistically provable formulas, the set of theorems of the logic of here-and-there includes, for instance, the weak law of the excluded middle

$$\neg F \vee \neg\neg F$$

(note that $\neg\neg F$ is not equivalent to F) and De Morgan's law

$$\neg(F \wedge G) \leftrightarrow \neg F \vee \neg G$$

(the dual law can be proved even intuitionistically).

The logic of here-and-there differs from intuitionistic logic also as far as minimal adequate sets of connectives are concerned. In intuitionistic logic, disjunction cannot be expressed in terms of the other connectives; in the logic of here-and-there, a disjunction

$$F \vee G$$

is equivalent [Lukasiewicz, 1941] to

$$((F \rightarrow G) \rightarrow G) \wedge ((G \rightarrow F) \rightarrow F).$$

3.6 Equilibrium Logic

Equilibrium logic [Pearce, 1997] defines when a set Y of atoms (i.e., an interpretation in the sense of classical logic) is an equilibrium logic for a propositional theory Γ. A set Y is an *equilibrium model* of Γ if

$$\text{for all } X \subseteq Y,\ (X, Y) \text{ is a model for } \Gamma \text{ iff } X = Y.$$

For instance, consider a propositional theory Γ consisting of the single formula $F = (p \rightarrow q) \rightarrow q$. We have seen in the previous section that $(\{q\}, \{q\}) \models F$ and that $(\emptyset, \{q\}) \not\models F$. Consequently, $\{q\}$ is an equilibrium model of Γ.

A logic program with nested expressions without classical negation can be written as a propositional theory by substituting, in each nested expression, each comma with $\wedge$, each semicolon by $\vee$, the negation *not* by $\neg$; then we consider each rule $F \leftarrow G$ as the implication $G \rightarrow F$. The equilibrium models of the propositional theory obtained in this way are the answer sets of the original logic program [Pearce, 1997; Lifschitz *et al.*, 2001].

When classical negation occurs in the logic program, to translate it into equilibrium logic we first replace each occurrence of a negative literals $\neg a$ by a new atom $\sim a$. The symbol $\sim$ is called *strong negation*. We say that a set of atoms is *coherent* if it doesn't contain pairs of "complementary" atoms a, $\sim a$. When the logic program contains classical negation, coherent equilibrium models of the corresponding propositional theory become the answer sets of the logic program after having replaced each $\neg a$ with $\sim a$ [Pearce, 1997].

3.7 Proving Strong Equivalence

Recall that two logic programs Π_1 and Π_2 are said to be *strongly equivalent* to each other if, for every program Π, the union $\Pi_1 \cup \Pi$ has the same answer sets as $\Pi_2 \cup \Pi$ (Section 2.3).

The first characterization of strong equivalence is applicable to programs with nested expressions, and it is based on the logic of here-and-there. Let Π_1 and Π_2 be two programs with nested expressions, and S the set of negative literals occurring in any of the two programs. Let Γ_1 and Γ_2 be the two propositional theories obtained from Π_1 and Π_2 as explained in Section 3.6. Program Π_1 and Π_2 are strongly equivalent iff Γ_1 and Γ_2 are equivalent in the logic of here-and-there

under the set of hypotheses

$$\{\neg(a \wedge \sim a) : \neg a \in S\}$$

[Lifschitz *et al.*, 2001]. Note that if Π_1 and Π_2 don't contain classical negation then the set of hypotheses is empty.

In addition to this, [Lifschitz *et al.*, 2001] shows that equivalence between two propositional theories Γ_1 and Γ_2 in the logic of here-and-there characterizes strong equivalence in equilibrium logic as well, if we define this relation as follows: Γ_1 is *strongly equivalent* to Γ_2 if, for every propositional theory Γ, $\Gamma_1 \cup \Gamma$ has the same equilibrium models of $\Gamma_2 \cup \Gamma$.

The second characterization of strong equivalence between logic programs with nested expressions is in terms of satisfaction of the reduct [Turner, 2003], and we will rephrase it as follows. Let A be the set of atoms occurring in programs Π_1 and Π_2. Programs Π_1 and Π_2 are strongly equivalent iff, for every consistent set Y of literals subset of A,

- $Y \models \Pi_1^Y$ iff $Y \models \Pi_2^Y$, and
- if $Y \models \Pi_1^Y$ then, for each $X \subset Y$, $X \models \Pi_1^Y$ iff $X \models \Pi_2^Y$.

The same paper showed that this characterization of strong equivalence holds also if Π_1 and Π_2 are programs with weight constraints. The characterization of strong equivalence between causal theories (see Chapter 9) is similar [Turner, 2004].

Chapter 4

Weight Constraints as Nested Expressions

Weight constraints, in particular in the form of cardinality constraints, are an important construct for answer set programming, and are found in many logic programs. On the other hand, programs with nested expressions were introduced for more theoretical reasons.

It may appear that the two extensions of the basic syntax of logic programs — nested expressions and weight constraints — have little in common. The following observation suggests that it would not be surprising actually if these ideas were related to each other. The original definition of an answer set is known to have the "anti-chain" property: an answer set for a program cannot be a subset of another answer set for the same program. Examples (2.10) and (2.19) show that the anti-chain property is lost as soon as nested expressions are allowed in rules. Example

$$\{p\}$$

shows that in the presence of cardinality constraints the anti-chain property does not hold either.

We show that there is indeed a close relationship between these two forms of the answer set semantics: cardinality and weight constraints can be viewed as shorthand for nested expressions of a special form. We define a simple, modular translation that turns any program Ω with weight constraints into a program $[\Omega]$ with nested expressions that has the same answer sets as Ω. Furthermore, every rule of $[\Omega]$ can be equivalently replaced with a set of nondisjunctive rules: rules whose head is a literal or $\bot$. This will lead us to a nondisjunctive version $[\Omega]^{nd}$ of the basic translation.

The translations defined in this chapter can be of interest for several reasons. First, the definition of an answer set for programs with weight constraints (Section 3.2) is technically somewhat complicated. Instead of introducing that definition, we can treat any program Ω with weight constraints as shorthand for its translation $[\Omega]$.

Second, the definition of program completion from [Clark, 1978] has been extended to nondisjunctive programs with nested expressions [Lloyd and Topor, 1984], and this extension is known to be equivalent to the definition of an answer set whenever the program is "tight" [Erdem and Lifschitz, 2003]. In view of this fact, answer sets for a tight logic program can be generated by running a satisfiability solver on the program's completion [Babovich *et al.*, 2000]. Consequently, answer sets for a program Ω with weight constraints can be computed by running a satisfiability solver on the completion of the translation $[\Omega]^{nd}$ if it is tight. This idea has led to the development of the answer set solver CMODELS (see Chapter 5 below).

Third, this translation can be used to reason about strong equivalence. We will show that the translations $[\Omega_1]$ and $[\Omega_2]$ of two programs with weight constraints Ω_1 and Ω_2 are strongly equivalent iff Ω_1 and Ω_2 are strongly equivalent. Since strong equivalence of two programs with nested expressions can be expressed in terms of the logic of here-and-there, we can use this logic to reason about strong equivalence

between Ω_1 and Ω_2.

4.1 A Useful Abbreviation

The following abbreviation is used in the definition of the translation $[\Omega]$ in Section 4.2. For any nested expressions $F_1, \ldots, F_n$ and any set X of subsets of $\{1, \ldots, n\}$, by

$$\langle F_1, \ldots, F_n \rangle : X$$

we denote the nested expression

$$\mathop{;}_{I \in X} \big(\mathop{,}_{i \in I} F_i \big). \tag{4.1}$$

The use of the "big comma" and the "big semicolon" in (4.1) to represent a multiple conjunction and a multiple disjunction is similar to the familiar use of $\bigwedge$ and $\bigvee$. In particular, the empty conjunction is understood as $\top$, and the empty disjunction as $\bot$.

For instance, if X is the set of all subsets of $\{1, \ldots, n\}$ of cardinality ≥ 3, then (4.1) expresses, intuitively, that at least 3 of the nested expressions $F_1, \ldots, F_n$ are true. It is easy to check, for this X, that a consistent set Z of literals satisfies (4.1) iff Z satisfies at least 3 of the nested expressions $F_1, \ldots, F_n$. This observation can be generalized:

Proposition 1. *Assume that for every subset I of $\{1, \ldots, n\}$ that belongs to X, all supersets of I belong to X also. For any nested expressions $F_1, \ldots, F_n$ and any consistent set Z of literals,*

$$Z \models \langle F_1, \ldots, F_n \rangle : X \text{ iff } \{i : Z \models F_i\} \in X.$$

Proof.

$$\begin{aligned}
Z \models \langle F_1, \ldots, F_n \rangle : X \quad &\text{iff} \quad \text{for some } I \in X, \text{ for all } i, \text{ if } i \in I \text{ then } Z \models F_i \\
&\text{iff} \quad \text{for some } I \in X,\ I \subseteq \{i : Z \models F_i\} \\
&\text{iff} \quad \text{for some } I \in X,\ I = \{i : Z \models F_i\} \\
&\text{iff} \quad \{i : Z \models F_i\} \in X.
\end{aligned}$$

□

As a last remark, note that, by the absorption property of the logic of here-and-there, if we take a program containing a multiple disjunction of the form (4.1) and restrict this disjunction to the sets I that are minimal in X, then the answer sets of the program will remain the same.

4.2 Translations

4.2.1 Basic Translation

In this section, we give the description of a translation from the language of weight constraints to the language of nested expressions, and state a theorem about the soundness of this translation. The definition of the translation consists of 4 parts.

1. The translation of a constraint of the form

$$L \leq \{c_1 = w_1, \ldots, c_m = w_m\} \tag{4.2}$$

is the nested expression

$$\langle c_1, \ldots, c_m \rangle : \left\{ I : L \leq \textstyle\sum_{i \in I} w_i \right\} \tag{4.3}$$

where I ranges over the subsets of $\{1, \ldots, m\}$. We denote the translation of $L \leq S$ by $[L \leq S]$.

2. The translation of a constraint of the form

$$\{c_1 = w_1, \ldots, c_m = w_m\} \leq U \tag{4.4}$$

is the nested expression

$$not\ \left(\langle c_1,\ldots,c_m\rangle : \left\{I : U < \sum_{i\in I} w_i\right\}\right). \tag{4.5}$$

where I ranges over the subsets of $\{1,\ldots,m\}$. We denote the translation of $S \le U$ by $[S \le U]$.

3. The translation of a general weight constraint is defined by

$$[L \le S \le U] = [L \le S], [S \le U].$$

Recall that $L \le S$ is shorthand for $L \le S \le \infty$, and $S \le U$ is shorthand for $-\infty \le S \le U$; translations of weight constraints of these special types have been defined earlier. It is easy to see that the old definition of $[L \le S]$ gives a nested expression equivalent to $[L \le S \le \infty]$ in the logic of here-and-there, and similarly for $[S \le U]$.

4. For any program Ω with weight constraints, its translation $[\Omega]$ is the program with nested expressions obtained from Ω by replacing each rule (2.13) with

$$(l_1; not\ l_1),\ldots,(l_p; not\ l_p), [C_0] \leftarrow [C_1],\ldots,[C_n] \tag{4.6}$$

where $l_1,\ldots,l_p$ are the positive head elements of (2.13).

The conjunctive terms in $(l_1; not\ l_1),\ldots,(l_p; not\ l_p)$ express, intuitively, that we are free to decide about every positive head element of the rule whether or not to include it in the answer set.

To illustrate this definition, let us apply it first to program (3.7). The translation of the cardinality constraint $0 \le \{a, b\} \le 1$ is

$$[0 \le \{a, b\}], [\{a, b\} \le 1]. \tag{4.7}$$

The first conjunctive term is

$$\langle a, b\rangle : \{\emptyset, \{1\}, \{2\}, \{1, 2\}\}$$

which equals

$$\top; a; b; (a, b)$$

and is equivalent to $\top$. Similarly, the second conjunctive term is equivalent to *not* (a, b). Consequently, (4.7) can be written as *not* (a, b). It follows that the translation of program (3.7) can be written as

$$(a; not\ a), (b; not\ b), not\ (a, b). \tag{4.8}$$

Similarly, we can check that program (3.9) turns into

$$a \leftarrow (not\ a; not\ b), not\ (not\ a, not\ b).$$

The translation defined above is sound:

Theorem 1. *For any program Ω with weight constraints, Ω and $[\Omega]$ have the same answer sets.*

The proof of this theorem is presented in Section 4.4.

We will conclude this section with a few comments about translating weight constraints of the forms $L \leq S$ and $S \leq U$.

In Section 2.2.2 we have agreed to identify any rule element c with the cardinality constraint $1 \leq \{c\}$, and to drop the head of a rule with weight constraints when this head is $1 \leq \{\ \}$. It is easy to check that $[1 \leq \{c\}]$ is equivalent to c, and $[1 \leq \{\ \}]$ is equivalent to $\bot$.

If the weights $w_1, \ldots, w_m$ are integers then the inequality in (4.5) is equivalent to $\lfloor U \rfloor + 1 \leq \sum_{i \in I} w_i$. Consequently, in the case of integer weights (in particular, in the case of cardinality constraints), $[S \leq U]$ can be written as *not* $[\lfloor U \rfloor + 1 \leq S]$. This is similar to a transformation that is used by the preprocessor LPARSE of system SMODELS.

The sign $<$ in place of $\leq$ is not allowed in weight constraints. But sometimes it is convenient to write expressions of the form

$$[L < \{c_1 = w_1, \ldots, c_m = w_m\}]$$

understood as shorthand for

$$\langle c_1, \dots, c_m \rangle : \left\{ I : L < \textstyle\sum_{i \in I} w_i \right\}. \tag{4.9}$$

Using this notation, we can write $[S \leq U]$ as *not* $[U < S]$.

Finally, note that each of the sets X used in the expressions $\langle c_1, \dots, c_m \rangle : X$ in nested expressions (4.3), (4.5) and (4.9) satisfies the assumption of Proposition 1 (Section 4.1), because the weights w_i are nonnegative.

4.2.2 Nondisjunctive Translation

For any program Ω with weight constraints, its *nondisjunctive translation* $[\Omega]^{nd}$ is the nondisjunctive program obtained from Ω by replacing each rule (2.13) with $p+1$ rules

$$\begin{aligned} &l_j \leftarrow not\ not\ l_j, [C_1], \dots, [C_n] \qquad (1 \leq j \leq p), \\ &\bot \leftarrow not\ [C_0], [C_1], \dots, [C_n], \end{aligned} \tag{4.10}$$

where $l_1, \dots, l_p$ are the positive head elements of (2.13).

For example, if Π is (3.7) then $[\Pi]$, as we have seen, is (4.8); the nondisjunctive translation $[\Pi]^{nd}$ of the same program is

$$\begin{aligned} &a \leftarrow not\ not\ a, \\ &b \leftarrow not\ not\ b, \\ &\bot \leftarrow not\ not\ (a, b). \end{aligned} \tag{4.11}$$

Proposition 2. *For any program Ω with weight constraints, $[\Omega]^{nd}$ is strongly equivalent to $[\Omega]$.*

In combination with Theorem 1, this fact shows that the nondisjunctive translation is sound: Ω and $[\Omega]^{nd}$ have the same answer sets.

Its proof is based on the following well-known fact about intuitionistic logic:

Fact 1. *If F is a propositional combination of formulas $F_1, \dots, F_m$ then $F \vee \neg F$ is intuitionistically derivable from $F_1 \vee \neg F_1, \dots, F_m \vee \neg F_m$.*

Proof of Proposition 2. We will show that formula (4.6) is equivalent to the conjunction of the formulas (4.10) in the logic of here-and-there. By Fact 1, the formula

$$[C_0] \vee \neg [C_0] \tag{4.12}$$

is entailed by the formulas $c \vee \neg c$ for all head elements c of rule (2.13). For every negative c, $c \vee \neg c$ is provable in the logic of here-and-there. It follows that (4.12) is derivable in the logic of here-and-there from the formulas $c \vee \neg c$ for all positive head elements c, that is, from the formulas $l_1 \vee \neg l_1, \dots, l_p \vee \neg l_p$. Consequently, $\neg\neg[C_0] \leftrightarrow [C_0]$ is derivable from these formulas as well. Hence (4.6) is equivalent in the logic of here-and-there to the rule

$$(l_1; \mathit{not}\ l_1), \dots, (l_p; \mathit{not}\ l_p), \mathit{not}\ \mathit{not}\ [C_0] \leftarrow [C_1], \dots, [C_n]$$

which can be broken into the rules

$$\begin{array}{ll} l_j; \mathit{not}\ l_j \leftarrow [C_1], \dots, [C_n] & (1 \leq j \leq p), \\ \mathit{not}\ \mathit{not}\ [C_0] \leftarrow [C_1], \dots, [C_n]. & \end{array}$$

The first line is equivalent to the first line of (4.10) in the logic of here-and-there. The second line is intuitionistically equivalent to the second line of (4.10). □

4.3 Strong Equivalence of Programs with Weight Constraints

For programs with weight constraints, the definition of strong equivalence is similar to the definition given in Section 2.3 above: Ω_1 and Ω_2 are *strongly equivalent* to each other if, for every program Ω with weight constraints, the union $\Omega_1 \cup \Omega$ has the same answer sets as $\Omega_2 \cup \Omega$. The method of proving strong equivalence of programs with weight constraints discussed in this section is based on the following proposition:

Proposition 3. *Ω_1 is strongly equivalent to Ω_2 iff $[\Omega_1]$ is strongly equivalent to $[\Omega_2]$.*

Proof. Assume that $[\Omega_1]$ is strongly equivalent to $[\Omega_2]$. Then, for any program with weight constraints Ω, $[\Omega_1] \cup [\Omega]$ has the same answer sets as $[\Omega_2] \cup [\Omega]$. The first program equals $[\Omega_1 \cup \Omega]$, and, by Theorem 1, has the same answer sets as $\Omega_1 \cup \Omega$. Similarly, the second program has the same answer sets as $\Omega_2 \cup \Omega$. Consequently Ω_1 is strongly equivalent to Ω_2.

Assume now that $[\Omega_1]$ is not strongly equivalent to $[\Omega_2]$. Consider the corresponding programs $[\Omega_1]'$, $[\Omega_2]'$ without classical negation, formed as described at the end of Section 3.5, and let *Cons* be the set of nested expressions $\neg(a \wedge a')$ for all new atoms a' occurring in these programs. By Theorem 2 from [Lifschitz *et al.*, 2001], $[\Omega_1]' \cup \mathit{Cons}$ is not equivalent to $[\Omega_2]' \cup \mathit{Cons}$ in the logic of here-and-there. It follows by Theorem 1 from [Lifschitz *et al.*, 2001] that there exists a unary program Π such that $[\Omega_1]' \cup \mathit{Cons} \cup \Pi$ and $[\Omega_2]' \cup \mathit{Cons} \cup \Pi$ have different collections of answer sets. (A program with nested expressions is said to be unary if each of its rules is an atom or has the form $a_1 \leftarrow a_2$ where a_1, a_2 are atoms.) Let Π^* be the program obtained from Π by replacing each atom of the form a' by $\neg a$. In view of the convention about identifying any literal l with the weight constraint $1 \leq \{l = 1\}$ (Section 2.2.2), Π^* can be viewed as a program with weight constraints, and it's easy to check that $[\Pi^*]'$ is strongly equivalent to Π. Then, for $i = 1, 2$, the program $[\Omega_i]' \cup \mathit{Cons} \cup \Pi$ has the same answer sets as the program $[\Omega_i]' \cup \mathit{Cons} \cup [\Pi^*]'$, which can be rewritten as $[\Omega_i \cup \Pi^*]' \cup \mathit{Cons}$. By the choice of Π, it follows that the collection of answer sets of $[\Omega_1 \cup \Pi^*]' \cup \mathit{Cons}$ is different from the collection of answer sets of $[\Omega_2 \cup \Pi^*]' \cup \mathit{Cons}$. Consequently, the same can be said about the pair of programs $[\Omega_1 \cup \Pi^*]$ and $[\Omega_2 \cup \Pi^*]$, and, by Theorem 1, about $\Omega_1 \cup \Pi^*$ and $\Omega_2 \cup \Pi^*$. It follows that Ω_1 is not strongly equivalent to Ω_2. □

As an example, let us check that the program

$$
\begin{array}{l}
1 \leq \{p, q\} \leq 1 \\
p
\end{array}
\tag{4.13}
$$

is strongly equivalent to

$$
\begin{array}{l}
\leftarrow q \\
p.
\end{array}
\tag{4.14}
$$

Rules (4.13), translated into the language of nested expressions and written in the syntax of propositional formulas, become

$$
\begin{array}{l}
(p \vee \neg p) \wedge (q \vee \neg q) \wedge (p \vee q) \wedge \neg(p \wedge q) \\
(p \vee \neg p) \wedge p.
\end{array}
$$

Rules (4.14), rewritten in a similar way, become

$$
\begin{array}{l}
\neg q \\
(p \vee \neg p) \wedge p.
\end{array}
$$

It is clear that each of these sets of formulas is intuitionistically equivalent to $\{p, \neg q\}$.

The fact that programs (4.13) and (4.14) are strongly equivalent to each other can be also proved directly, using the definition of strong equivalence and the definition of an answer set for programs with weight constraints. But this proof would not be as easy as the one above. Generally, to establish that a program Ω_1 is strongly equivalent to a program Ω_2, we need to show that for every program Ω and every consistent set Z of literals,

(a$_1$) $Z \models \Omega_1 \cup \Omega$ and

(b$_1$) $cl((\Omega_1 \cup \Omega)^Z) = Z$

if and only if

(a$_2$) $Z \models \Omega_2 \cup \Omega$ and

(b$_2$) $cl((\Omega_2 \cup \Omega)^Z) = Z$.

Sometimes we may be able to check separately that (a$_1$) is equivalent to (a$_2$) and that (b$_1$) is equivalent to (b$_2$), but in other cases this may not work. For instance, if Ω_1 is (4.13) and Ω_2 is (4.14) then (b$_1$) may not be equivalent to (b$_2$).

An alternative method of establishing the strong equivalence of programs with weight constraints is proposed in [Turner, 2003, Section 6]. According to that approach, we check that for every consistent set Z of literals and every subset Z' of Z,

(a$_3$) $Z \models \Omega_1$ and

(b$_3$) $Z' \models \Omega_1^Z$

if and only if

(a$_4$) $Z \models \Omega_2$ and

(b$_4$) $Z' \models \Omega_2^Z$.

4.4 Proof of Theorem 1

Lemma 1. *For any weight constraint C and any consistent set Z of literals, $Z \models [C]$ iff $Z \models C$.*

Proof. It is sufficient to prove the assertion of the lemma for constraints of the forms $L \leq S$ and $S \leq U$. Let S be $\{c_1 = w_1, \dots, c_m = w_m\}$. Then, by Proposition 1 (Section 4.1),

$$\begin{array}{rll} Z \models [L \leq S] & \text{iff} & \{i : Z \models c_i\} \in \left\{I : L \leq \sum_{i \in I} w_i\right\} \\ & \text{iff} & L \leq \sum_{i : Z \models c_i} w_i \\ & \text{iff} & Z \models L \leq S. \end{array}$$

7

Similarly,

$$
\begin{aligned}
Z \models [S \le U] \quad & \text{iff} \quad \{i : Z \models c_i\} \notin \left\{I : U < \textstyle\sum_{i \in I} w_i\right\} \\
& \text{iff} \quad U \ge \textstyle\sum_{i : Z \models c_i} w_i \\
& \text{iff} \quad Z \models S \le U.
\end{aligned}
$$

□

Lemma 2. *For any constraint $L \le S$ and any consistent sets Z, Z' of literals,*

$$Z' \models [L \le S]^Z \text{ iff } Z' \models (L \le S)^Z.$$

Proof. Let S be $\{c_1 = w_1, \ldots, c_m = w_m\}$ and let I stand for $\{1, \ldots, m\}$. It is immediate from the definition of the reduct in Section 3.1 that

$$\big(\langle F_1, \ldots, F_n \rangle : X\big)^Z = \langle F_1^Z, \ldots, F_n^Z \rangle : X. \tag{4.15}$$

For any subset J of I, let ΣJ stand for $\sum_{i \in J} w_i$. Using (4.15) and Proposition 1, we can rewrite the left-hand side of the equivalence to be proved as follows:

$$
\begin{aligned}
Z' \models [L \le S]^Z \quad & \text{iff} \quad Z' \models \langle c_1^Z, \ldots, c_m^Z \rangle : \{J \subseteq I : L \le \Sigma J\} \\
& \text{iff} \quad \{i \in I : Z' \models c_i^Z\} \in \{J \subseteq I : L \le \Sigma J\} \\
& \text{iff} \quad L \le \Sigma\{i \in I : Z' \models c_i^Z\}
\end{aligned}
$$

Let I' be the set of all $i \in I$ such that the rule element c_i is positive, and let I'' be the set of all $i \in I \setminus I'$ such that $Z \models c_i$. It is clear that c_i^Z is c_i for $i \in I'$, $\top$ for

$i \in I''$, and $\bot$ for all other values of i. Consequently

$$\begin{aligned} Z' \models [L \leq S]^Z \quad & \text{iff} \quad L \leq \Sigma\{i \in I' : Z' \models c_i\} + \Sigma I'' \\ & \text{iff} \quad L - \Sigma I'' \leq \Sigma\{i \in I' : Z' \models c_i\} \\ & \text{iff} \quad Z' \models (L^Z \leq S') \end{aligned}$$

where L^Z and S' are defined as in Section 3.2. It remains to notice that $(L \leq S)^Z = (L^Z \leq S')$. □

Lemma 3. *For any constraint $S \leq U$ and any consistent set Z of literals,*

$$[S \leq U]^Z = \begin{cases} \top\ , & \textit{if } Z \models (S \leq U), \\ \bot\ , & \textit{otherwise.} \end{cases}$$

Proof. By the definition of the reduct in Section 3.1, $[S \leq U]^Z$ is

- $\top$, if $Z \not\models [U < S]$,
- $\bot$, otherwise.

It remains to notice that $Z \not\models [U < S]$ iff $Z \models [S \leq U]$, and then iff $Z \models S \leq U$ by Lemma 1. □

In Lemmas 4–7, Ω is an arbitrary program with weight constraints. Recall that, according to Section 4.2.2, the nondisjunctive translation $[\Omega]^{nd}$ of Ω consists of rules of two kinds:

$$l_j \leftarrow \textit{not not } l_j, [C_1], \ldots, [C_n] \tag{4.16}$$

and

$$\bot \leftarrow \textit{not } [C_0], [C_1], \ldots, [C_n]. \tag{4.17}$$

We will denote the set of rules (4.16) corresponding to all rules of Ω by Π_1, and the set of rules (4.17) corresponding to all rules of Ω by Π_2, so that

$$[\Omega]^{nd} = \Pi_1 \cup \Pi_2. \tag{4.18}$$

Lemma 4. *A consistent set Z of literals is an answer set for $[\Omega]^{nd}$ iff Z is an answer set for Π_1 and $Z \models \Pi_2$.*

In view of (4.18), this is an instance of a general fact, proved in [Lifschitz *et al.*, 1999] as Proposition 2, that can be restated as the following:

Fact 2. *Let Π_1, Π_2 be programs with nested expressions such that the head of every rule in Π_2 is $\bot$. A consistent set Z of literals is an answer set for $\Pi_1 \cup \Pi_2$ iff Z is an answer set for Π_1 and $Z \models \Pi_2$.*

Lemma 5. *For any consistent set Z of literals, $Z \models \Omega$ iff $Z \models \Pi_2$.*

Proof. It is sufficient to consider the case when Ω consists of a single rule (2.13). In this case, $Z \models \Omega$ iff

$$Z \models C_0 \text{ or, for some } i \ (1 \leq i \leq m),\ Z \not\models C_i.$$

On the other hand, $Z \models \Pi_2$ iff

$$Z \models [C_0] \text{ or, for some } i \ (1 \leq i \leq m),\ Z \not\models [C_i].$$

By Lemma 1, these conditions are equivalent to each other. □

Lemma 6. *For any consistent sets Z, Z' of literals, $Z' \models \Omega^Z$ iff $Z' \models \Pi_1^Z$.*

Proof. It is sufficient to consider the case when Ω consists of a single rule (3.8). Then Π_1^Z consists of the rules

$$l \leftarrow (not\ not\ l)^Z, [L_1 \leq S_1]^Z, [S_1 \leq U_1]^Z, \ldots, [L_n \leq S_n]^Z, [S_n \leq U_n]^Z \tag{4.19}$$

for all positive head elements l of (3.8).

Case 1: for every i $(1 \le i \le n)$, $Z \models S_i \le U_i$. Then, by Lemma 3, each of the formulas $[S_1 \le U_1]^Z, \ldots, [S_n \le U_n]^Z$ is $\top$. Note also that if $l \notin Z$ then $(not\ not\ l)^Z$ is $\bot$, so that (4.19) is satisfied by any consistent set of literals. Consequently Z' satisfies Π_1^Z iff, for each positive head element $l \in Z$,

$$Z' \models l \text{ or, for some } i\ (1 \le i \le m),\ Z' \not\models [L_i \le S_i]^Z. \tag{4.20}$$

On the other hand, according to the definition of the reduct from Section 3.2, Ω^Z is the set of rules

$$l \leftarrow (L_1 \le S_1)^Z, \ldots, (L_n \le S_n)^Z$$

for all positive head elements l satisfied by Z. Then $Z' \models \Omega^Z$ iff, for each positive head element $l \in Z$,

$$Z' \models l \text{ or, for some } i\ (1 \le i \le m),\ Z' \not\models (L_i \le S_i)^Z.$$

By Lemma 2, this condition is equivalent to (4.20).

Case 2: for some i, $Z \not\models S_i \le U_i$. Then, by Lemma 3, one of the formulas $[S_i \le U_i]^Z$ is $\bot$, so that each rule (4.19) is trivially satisfied by any Z'. On the other hand, in this case Ω^Z is empty. $\square$

Lemma 7. *If set $cl(\Omega^Z)$ is consistent then it is the only answer set for Π_1^Z; otherwise, Π_1^Z has no answer sets.*

Proof. Recall that $cl(\Omega^Z)$ is defined as the unique minimal set satisfying Ω^Z (Section 3.2). The answer sets for a program with nested expressions that does not contain negation as failure are defined as the minimal consistent sets satisfying that program (Section 3.1). It remains to notice that Ω^Z and Π_1^Z are satisfied by the same sets of literals (Lemma 6). $\square$

Theorem 1. *For any program Ω with weight constraints, Ω and $[\Omega]$ have the same answer sets.*

Proof. By the definition of an answer set for programs with weight constraints (Section 3.2), a consistent set Z of literals is an answer set for Ω iff

$$cl(\Omega^Z) = Z \text{ and } Z \models \Omega.$$

By Lemmas 7 and 5, this is equivalent to the condition

$$Z \text{ is an answer set for } \Pi_1^Z \text{ and } Z \models \Pi_2.$$

By the definition of an answer set for programs with nested expressions (Section 3.1) and by Lemma 4, this is further equivalent to saying that Z is an answer set for $[\Omega]^{nd}$. By Proposition 2, $[\Omega]^{nd}$ has the same answer sets as $[\Omega]$. □

Chapter 5

Programs with Weight Constraints as Traditional Programs

A nondisjunctive rule is *nonnested* if its body is a conjunction of literals, each possibly prefixed with *not*. A *nonnested program* is a program whose rules are nonnested. Thus the syntactic form of nonnested programs is the same as traditional programs, except that the head of a nonnested rule can be $\bot$. (This difference is not very essential: using an auxiliary atom we can rewrite any nonnested program as a traditional program.)

Since the answer sets for a nonnested program have the anti-chain property (see the introduction of Chapter 4), turning a program with weight constraints into a nonnested program with the same answer sets is, generally, impossible. But we can turn any program with weight constraints into its nonnested conservative extension—into a program that may contain new atoms; dropping the new atoms from the answer sets of the translation gives the answer sets for the original program.

In this chapter, we first show a "nonnested translation" $[\Omega]^{nn}$ of Ω that can

be seen as the result of eliminating nested expressions in $[\Omega]^{nd}$ (see Section 4.2.2) in favor of additional atoms. This translation is implemented by Yuliya Lierler in the answer set solver CMODELS to eliminate weight constraints [Giunchiglia *et al.*, 2004b].

The possibility of translating programs with cardinality constraints into the language of nonnested programs at the price of introducing new atoms was first established by Marek and Remmel [2002]. Our nonnested translation is more general, because it is applicable to programs with arbitrary weight constraints. Its other advantage is that, in the special case when all weights in the program are expressed by integers of a limited size (in particular, in the case of cardinality constraints) the translation can be computed in polynomial time. [1]

The second translation described in this chapter is limited to programs with integer weights. It reduces all weights in a program to 1. This can be done in polynomial time, so that, together with the previous translation, we can reduce every program with weight constraints into a nonnested program in polynomial time.

The first translation is described in Section 5.1, while the reduction to cardinality constraints is presented in Sections 5.2.1 and 5.2.

5.1 Eliminating Nested Expressions

Each of the new atoms introduced in the nonnested translation $[\Omega]^{nn}$ below is, intuitively, an "abbreviation" for some nested expression related to the nondisjunctive translation $[\Omega]^{nd}$. For instance, to eliminate the nesting of negations from the first line of the nondisjunctive translation (4.10), we will introduce, for every j, a new

[1] The fast algorithm for transforming such a program Ω into $[\Omega]^{nd}$ does not go through the intermediate step of constructing $[\Omega]^{nn}$: that program can be exponentially larger than Ω.

atom $q_{not\ l_j}$, and replace that line with the rules

$$q_{not\ l_j} \leftarrow not\ l_j,$$
$$l_j \leftarrow not\ q_{not\ l_j}, [C_1], \dots, [C_n]$$

($1 \leq j \leq p$). The first of these rules tells us that the new atom $q_{not\ l_j}$ is used to "abbreviate" the nested expression $not\ l_j$. The second rule is the first of rules (4.10) with this subexpression replaced by the corresponding atom. For instance, the nondisjunctive translation (4.11) of program (3.7) turns after this transformation into

$$\begin{aligned} & q_{not\ a} \leftarrow not\ a, \\ & a \leftarrow not\ q_{not\ a}, \\ & q_{not\ b} \leftarrow not\ b, \\ & b \leftarrow not\ q_{not\ b}, \\ & \bot \leftarrow not\ not\ (a; b). \end{aligned} \tag{5.1}$$

Introducing the atoms $q_{not\ l_j}$ brings us very close to the goal of eliminating nesting altogether, because every rule of the program obtained from $[\Omega]^{nd}$ by this transformation is strongly equivalent to a set of nonnested rules. One way to eliminate nesting is to convert the body of every rule to a "disjunctive normal form" using De Morgan's laws, the distributivity of conjunction over disjunction, and, in the case of the second line of (4.10), double negation elimination.[2] After that, we can break every rule into several nonnnested rules, each corresponding to one of the disjunctive terms of the body. For instance, the last rule of (5.1) becomes

$$\bot \leftarrow a; b$$

after the first step and

$$\bot \leftarrow a,$$
$$\bot \leftarrow b$$

[2] All these transformations are intuitionistically equivalent, and consequently preserve strong equivalence (Section 3.7). In particular, double negation elimination in the body of a rule with the head $\bot$ is intuitionistically valid.

after the second.

The definition of $[\Omega]^{nn}$ below follows a different approach to the elimination of the remaining nested expressions. Besides the "negation atoms" of the form $q_{not\ l_j}$, it introduces other new atoms, to make the translation of weight constraints more compact in some cases. These "weight atoms" have the forms $q_{w \leq S}$ and $q_{w<S}$, where w is a number and S is an expression of the form $\{c_1 = w_1, \dots, c_m = w_m\}$ for some rule elements $c_1, \dots, c_m$ and nonnegative numbers $w_1, \dots, w_m$. They "abbreviate" the nested expressions $[w \leq S]$ and $[w < S]$ respectively.

In the following definition, $\{c_1 = w_1, \dots, c_m = w_m\}'$, where $m > 0$, stands for $\{c_1 = w_1, \dots, c_{m-1} = w_{m-1}\}$. Consider a nonnested program Π that may contain atoms of the forms $q_{w \leq S}$ and $q_{w<S}$. We say that Π is *closed* if

- for each atom of the form $q_{w \leq S}$ that occurs in Π, Π contains the rule

  $$q_{w \leq S} \tag{5.2}$$

  if $w \leq 0$, and the pair of rules

  $$\begin{aligned} q_{w \leq S} &\leftarrow q_{w \leq S'}, \\ q_{w \leq S} &\leftarrow c_m, q_{w-w_m \leq S'} \end{aligned} \tag{5.3}$$

  if $0 < w \leq w_1 + \dots + w_m$;

- for each atom of the form $q_{w<S}$ that occurs in Π, Π contains the rule

  $$q_{w<S} \tag{5.4}$$

  if $w < 0$, and the pair of rules

  $$\begin{aligned} q_{w<S} &\leftarrow q_{w<S'}, \\ q_{w<S} &\leftarrow c_m, q_{w-w_m<S'} \end{aligned} \tag{5.5}$$

  if $0 \leq w < w_1 + \dots + w_m$.

We define the nonnested translation $[L \leq S \leq U]^{nn}$ of a weight constraint $L \leq S \leq U$ as the conjunction

$$q_{L \leq S}, \mathit{not}\ q_{U<S}.$$

Now we are ready to define the nonnested translation of a program. For any program Ω with weight constraints, $[\Omega]^{nn}$ is the smallest closed program that contains, for every rule

$$L_0 \leq S_0 \leq U_0 \leftarrow C_1, \ldots, C_n$$

of Ω, the rules

$$q_{\mathit{not}\ l} \leftarrow \mathit{not}\ l \tag{5.6}$$

and

$$l \leftarrow \mathit{not}\ q_{\mathit{not}\ l}, [C_1]^{nn}, \ldots, [C_n]^{nn} \tag{5.7}$$

for each of its positive head elements l, and the rules

$$\begin{aligned} \bot &\leftarrow \mathit{not}\ q_{L_0 \leq S_0}, [C_1]^{nn}, \ldots, [C_n]^{nn}, \\ \bot &\leftarrow q_{U_0 < S_0}, [C_1]^{nn}, \ldots, [C_n]^{nn}. \end{aligned} \tag{5.8}$$

For instance, if Ω is (3.7) then rules (5.6)–(5.8) are

$$\begin{aligned} & q_{\mathit{not}\ a} \leftarrow \mathit{not}\ a, \\ & a \leftarrow \mathit{not}\ q_{\mathit{not}\ a}, \\ & q_{\mathit{not}\ b} \leftarrow \mathit{not}\ b, \\ & b \leftarrow \mathit{not}\ q_{\mathit{not}\ b}, \\ & \bot \leftarrow \mathit{not}\ q_{0 \leq \{a,b\}}, \\ & \bot \leftarrow q_{1<\{a,b\}}. \end{aligned} \tag{5.9}$$

To make this program closed, we add to it the following "definitions" of the weight atoms $q_{0 \leq \{a,b\}}$ and $q_{1<\{a,b\}}$, and, recursively, of the weight atoms that are used in

these definitions:

$$
\begin{aligned}
& q_{0 \leq \{a,b\}}, \\
& q_{1 < \{a,b\}} \leftarrow q_{1 < \{a\}}, \\
& q_{1 < \{a,b\}} \leftarrow b, q_{0 < \{a\}}, \\
& q_{0 < \{a\}} \leftarrow q_{0 < \{\}}, \\
& q_{0 < \{a\}} \leftarrow a, q_{-1 < \{\}}, \\
& q_{-1 < \{\}}.
\end{aligned}
\tag{5.10}
$$

The nonnested translation of (3.7) consists of rules (5.9) and (5.10).

The following theorem describes the relationship between the answer sets for Ω and the answer sets for $[\Omega]^{nn}$. In the statement of the theorem, Q_Ω stands for the set of all new atoms that occur in $[\Omega]^{nn}$—both negation atoms $q_{not\ l}$ and weight atoms $q_{w \leq S}$, $q_{w < S}$.

Theorem 2. *For any program Ω with weight constraints, $Z \mapsto Z \setminus Q_\Omega$ is a 1–1 correspondence between the answer sets for $[\Omega]^{nn}$ and the answer sets for Ω.*

It is easy to see that the translation $\Omega \mapsto [\Omega]^{nn}$ is modular, in the sense that it can be computed by translating each rule of Ω separately.

Recall that the introduction of the new atoms $q_{w \leq S}$ and $q_{w < S}$ is motivated by the desire to make the translations of programs more compact. We will investigate now to what degree this goal has been achieved.

The basic translation $[C]$ of a weight constraint, as defined in Section 4.2.1, can be exponentially larger than C. For this reason, the basic and nondisjunctive translations of a program Ω are, generally, exponentially larger than Ω.

The nonnested translation of a program Ω consists of the rules (5.6)–(5.8) corresponding to all rules of Ω, and the additional rules (5.2)–(5.5) that make the program closed. The part consisting of rules (5.6)–(5.8) cannot be significantly larger than Ω, because each of the nested expressions $[C_i]^{nn}$ is short — it contains at most two atoms. The second part consists of the "definitions" of all weight atoms in $[\Omega]^{nn}$,

and it contains at most two short rules for every such atom. Under what conditions can we guarantee that the number of weight atoms is not large in comparison with the size of Ω?

The *length* of a weight constraint (2.12) is m, and its *weight* is $w_1 + \cdots + w_m$. We will denote the length of C by $L(C)$, and the weight of C by $W(C)$.

Proposition 4. *For programs Ω without non-integer weights, the number of weight atoms occurring in $[\Omega]^{nn}$ is $O\big(\sum L(C) \cdot W(C)\big)$, where the sum extends over all weight constraints C occurring in Ω.*

If the weights in Ω come from a fixed finite set of integers (for instance, if every weight constraint in Ω is a cardinality constraint) then $W(C) = O(L(C))$, and the proposition above shows that the number of weight atoms in $[\Omega]^{nn}$ is not large in comparison with the size of Ω. Consequently, in this case $[\Omega]^{nn}$ cannot be large in comparison with Ω either.

Proof of Proposition 4. Let Ω be a program without non-integer weights. About a rule from $[\Omega]^{nn}$ we will say that it is *relevant* if for every weight atom $w \leq S$ or $w < S$ occurring in that rule there is a weight constraint (2.12) in Ω such that S is $\{c_1 = w_1, \ldots, c_j = w_j\}$ for some $j \in \{0, \ldots, m\}$, and

$$w \in \{-\max(w_1, \ldots, w_m), \ldots, w_1 + \cdots + w_m\} \cup \{L, U\}.$$

It is clear that the number of weight atoms occurring in relevant rules can be estimated as $O\big(\sum L(C) \cdot W(C)\big)$. On the other hand, it is easy to see that the set of relevant rules contains the rules (5.6)–(5.8) corresponding to all rules of Ω, and that it is closed. Consequently, all rules in $[\Omega]^{nn}$ are relevant. $\square$

The fact that, in case of integer and bounded weights, $[\Omega]^{nn}$ can be computed from Ω in polynomial time, is not hard to check in view of Proposition 4.

5.2 Removing the Weights

5.2.1 Simplifying the Syntax of Weight Constraints

The second translation is limited to the case when all weights in the program are positive integers, and the bounds are integers. In addition to this, the translation requires the logic programs to satisfy some conditions:

- the head of each rule is an atom, $\bot$ or a cardinality constraints without lower and upper bound (that is, the rule is a choice rule),
- all weight constraints in the body have the form $L \leq S$ or $S \leq U$, and
- the bound conditions of weight constraints are not trivially true or false. (That is, lower bounds are positive and not greater than the sum of the weights, and upper bounds are nonnegative and lower than the sum of the weights.)

Those conditions can be satisfied by applying transformations similar to the ones computed by the preprocessor LPARSE of the answer set solver SMODELS. The first condition can be satisfied by rewriting each generic rule with weight constraints of the form (3.8) as 3 rules

$$\begin{aligned}
&S_0 \leftarrow F \\
&\bot \leftarrow F, S_0 \leq (L_0 - 1) \\
&\bot \leftarrow F, (U_0 + 1) \leq S_0
\end{aligned} \tag{5.11}$$

where F stands for

$$L_1 \leq S_1, S_1 \leq U_1, \ldots, L_n \leq S_n, S_n \leq U_n.$$

It is actually not hard to see that this transformation is correct, in view of Proposition 2 and Theorem 1.

For the second condition, we can rewrite each weight constraint $L \leq S \leq U$ in the body of a rule as two elements $L \leq S$ and $S \leq U$.

```
      function wc2cc(Ω)
 1        Ω' := ∅
 2        rewrite each expression of the form S ≤ U in Ω as not (U + 1) ≤ S
 3        foreach L ≤ S occurring in the body of a rule of Ω
 4            while a weight in S is greater than 1
                      (let S be {c_1 = w_1, ..., c_n = w_m})
 5                R := {c_i : i ∈ {1, ..., n}, w_i is odd}
 6                H := {c_i = ⌊w_i/2⌋ : i ∈ {1, ..., n}, w_i > 1}
 7                foreach i : 0 < i ≤ |R|, i + L is even
 8                    d := new atom
 9                    H := H ∪ {d}
10                    Ω' := Ω' ∪ {d ← i ≤ R}
11                end foreach
12                replace L ≤ S with ⌈L/2⌉ ≤ H
13            end while
14        end foreach
15        rewrite each expression of the form not L ≤ S in Ω as S ≤ (L − 1)
16        return Ω ∪ Ω'
```

Figure 5.1: A translation that eliminates weight constraints in favor of cardinality constraints

For the last condition, we drop each rule that contains a weight constraint where the upper bound is negative, or the lower bound is greater than the sum of weights (trivially false bounds). From the remaining rules, we drop every weight constraint in the body where the lower bound is not positive, or the upper bound is not lower than the sum of weights (trivially true bounds).

5.2.2 The Procedure

In Figure 5.1, we have a procedure `wc2cc` that eliminates, in a finite program with weight constraints in the syntax explained above, weight constraints in favor of cardinality constraints. In it, S, R and H are considered multisets.

Line 2 replaces each weight constraint of the form $S \leq U$ with *not* $((U+1) \leq S)$, and line 15 reverses the process. This allows $(U + 1) \leq S$ to be considered as an expression of the form $L \leq S$ in line 3. This transformation not only follows the

intuitive meaning of a weight constraint, but also $[S \leq U] = not\ [(U+1) \leq S]$. It is also easy to check that if $S \leq U$ has no trivial bounds conditions iff $(U+1) \leq S$ doesn't have them.

At each iteration of the **while** loop the bound L and the weights in S are halved. At line 5, set H contains the result of the integer division of weights in S by 2, and set R contains the "remainder" of the division. We need to include the contribution of the atoms in R to H, where each atom in R should count with a "weight" 1/2. The auxiliary atoms d added to H (and that are "defined" with rules added to Ω') have this role: intuitively, the number of these atoms that are "true" is about half the number of "true" elements of R.

Consider, for instance, the following program:

$$
\begin{aligned}
& p \leftarrow \{r = 3, q = 1\} \leq 3 \\
& q \\
& r \leftarrow not\ p
\end{aligned} \tag{5.12}
$$

At the first step, $\{r = 3, q = 1\} \leq 3$ becomes $not\ 4 \leq \{r = 3, q = 1\}$, so $4 \leq \{r = 3, q = 1\}$ is the only weight constraint $L \leq S$ with a weight greater than 1. For such values of L and S, $R = \{r, q\}$, H is initially computed as $\{r\}$, and i assumes value 2 only. Consequently, if d is the name of the new atom, H becomes $\{r, d\}$, $\{d \leftarrow 2 \leq \{r, q\}\}$ is added to Ω' and $4 \leq \{r = 3, q = 1\}$ is replaced by $2 \leq \{r, d\}$ in Ω. No other weight is greater than 1, so the procedure returns the current values of $\Omega \cup \Omega'$ after having converted $not\ 2 \leq \{r, d\}$ into $\{r, d\} \leq 1$: the output is the program is

$$
\begin{aligned}
& p \leftarrow \{r, d\} \leq 1 \\
& q \\
& r \leftarrow not\ p \\
& d \leftarrow 2 \leq \{r, q\}
\end{aligned} \tag{5.13}
$$

In the following theorem Ω is a program with weight constraints satisfying the conditions from Section 5.2.1. By the size of a number we mean its length in binary notation.

Theorem 3. *For any* Ω*,*

(a) the mapping $Z \mapsto Z \cap \sigma$ *is a 1–1 correspondence between the answer sets of* `wc2cc`(Ω) *and the answer sets of* Ω*, and*

(b) `wc2cc` *terminates in time polynomial in the size of the input.*

As an example of (a), the answer sets of (5.12) are $\{p, q\}$ and $\{q, r\}$, and the answer sets of the translation (5.13) are $\{p, q\}$ and $\{d, q, r\}$. The difference between the answer sets of the two programs is only in the presence of the atom d in one of the answer sets.

5.3 Proofs

The proof of Theorem 2 requires important properties of programs with nested expressions, which are introduced in the next section.

5.3.1 Two Lemmas on Programs with Nested Expressions

The idea of program completion [Clark, 1978] is that the set of rules of a program with the same atom q in the head is the "if" part of a definition of q; the "only if" half of that definition is left implicit. If, for instance, the rule

$$q \leftarrow F$$

is the only rule in the program whose head is q then that rule is an abbreviated form of the assertion that q is equivalent to F.

Since in a rule with nested expressions the head is allowed to have the same syntactic structure as the body, the "only if" part of such an equivalence can be expressed by a rule also:

$$F \leftarrow q.$$

The lemma below shows that adding such rules to a program does not change its answer sets.

An occurrence of a formula F in a formula or a rule is *singular* if the symbol before this occurrence of F is $\neg$; otherwise the occurrence is *regular* [Lifschitz *et al.*, 1999]. The expression

$$F \leftrightarrow G$$

stands for the pair of rules

$$F \leftarrow G$$

$$G \leftarrow F.$$

Proposition 5 (Completion Lemma)**.** *Let* Π *be a program with nested expressions, and let* Q *be a set of atoms that do not have regular occurrences in the heads of the rules of* Π*. For every* $q \in Q$*, let* $Def(q)$ *be a formula. Then the program*

$$\Pi \cup \{q \leftarrow Def(q) \ : \ q \in Q\}$$

has the same answer sets as the program

$$\Pi \cup \{q \leftrightarrow Def(q) \ : \ q \in Q\}.$$

In the special case when Q is a singleton this fact was first proved by Esra Erdem (personal communication).

In the statement of the completion lemma, if the atoms from Q occur neither in Π nor in the formulas $Def(q)$ then adding the rules $q \leftarrow Def(q)$ to Π extends the program by "explicit definitions" of "new" atoms. According to the lemma below, such an extension is conservative: the answer sets for Π can be obtained by dropping the new atoms from the answer sets for the extended program.

Proposition 6 (Lemma on Explicit Definitions)**.** *Let* Π *be a program with nested expressions, and let* Q *be a set of atoms that do not occur in* Π*. For every* $q \in Q$*, let* $Def(q)$ *be a formula that contains no atoms from* Q*. Then* $Z \mapsto Z \setminus Q$ *is a 1–1 correspondence between the answer sets for* $\Pi \cup \{q \leftarrow Def(q) \ : \ q \in Q\}$ *and the answer sets for* Π*.*

The completion lemma and the lemma on explicit definitions will be generalized to arbitrary propositional theories in Section 6.3, and the proofs of the more general statements are provided in Section 6.5.

5.3.2 Proof of Theorem 2

Let Ω be a program with weight constraints. Consider the subset Δ of its nonnested translation $[\Omega]^{nn}$ consisting of the rules whose heads are atoms from Q_Ω. The rules included in Δ have the forms (5.2)–(5.6); they "define" the atoms in Q_Ω. The rest of $[\Omega]^{nn}$ will be denoted by Π; the rules of Π have the forms (5.7) and (5.8). The union of these two programs is $[\Omega]^{nn}$:

$$[\Omega]^{nn} = \Pi \cup \Delta. \tag{5.14}$$

The idea of the proof of Theorem 2 is to transform $\Pi \cup \Delta$ into a program with the same answer sets so that Π will turn into $[\Omega]^{nd}$ and Δ will turn into a set of explicit definitions in the sense of Section 5.3.1, and then use the lemma on explicit definitions.

For every atom $q \in Q_\Omega$, define the formula $Def(q)$ as follows:

$$Def(q_{\text{not } l}) = not\ l$$

$$Def(q_{w \le S}) = \begin{cases} \top, & \text{if } w \le 0, \\ q_{w \le S'}; (c_m, q_{w-w_m \le S'}), & \text{if } 0 < w \le w_1 + \cdots + w_m, \\ \bot, & \text{otherwise} \end{cases}$$

$$Def(q_{w < S}) = \begin{cases} \top, & \text{if } w < 0, \\ q_{w < S'}; (c_m, q_{w-w_m < S'}), & \text{if } 0 \le w < w_1 + \cdots + w_m, \\ \bot, & \text{otherwise} \end{cases}$$

Lemma 8. *Program $[\Omega]^{nn}$ has the same answer sets as*

$$\Pi \cup \{q \leftrightarrow Def(q) \ : \ q \in Q_\Omega\}.$$

Proof. From the definitions of $[\Omega]^{nn}$ and Q_Ω we conclude that Δ consists of the following rules:

- rule (5.2) for every atom of the form $q_{w \le S}$ in Q_Ω such that $w \le 0$;
- rules (5.3) for every atom of the form $q_{w \le S} \in Q_\Omega$ such that

$$0 < w \le w_1 + \cdots + w_m;$$

- rule (5.4) for every atom of the form $q_{w < S}$ in Q_Ω such that $w < 0$;
- rules (5.5) for every atom of the form $q_{w < S}$ in Q_Ω such that

$$0 \le w < w_1 + \cdots + w_m;$$

- rule (5.6) for every atom of the form $q_{\text{not } l}$ in Q_Ω.

Consequently Δ is strongly equivalent to $\{q \leftarrow Def(q) \ : \ q \in Q_\Omega\}$. Then, by (5.14), program $[\Omega]^{nn}$ has the same answer sets as $\Pi \cup \{q \leftarrow Def(q) \ : \ q \in Q_\Omega\}$. The assertion to be proved follows by the completion lemma. ☐

Lemma 9. *Let* S *be* $\{c_1 = w_1, \dots, c_m = w_m\}$*. In the logic of here-and-there,*

$$[w \le S] \leftrightarrow \begin{cases} \top, & \text{if } w \le 0, \\ [w \le S'];(c_m, [w - w_m \le S']), & \text{if } 0 < w \le w_1 + \dots + w_m, \\ \bot, & \text{otherwise.} \end{cases}$$

$$[w < S] \leftrightarrow \begin{cases} \top, & \text{if } w < 0, \\ [w < S'];(c_m, [w - w_m < S']), & \text{if } 0 \le w < w_1 + \dots + w_m, \\ \bot, & \text{otherwise.} \end{cases}$$

Proof. Recall that $[w \le S]$ is an expression of the form (4.3), which stands for a disjunction of conjunctions (4.1). If $w \le 0$ then the set after the : sign in (4.3) has the empty set as one of its elements, so that one of the disjunctive terms of this formula is the empty conjunction $\top$. If $w > w_1 + \dots + w_m$ then the set after the : sign in (4.3) is empty, so that the formula is the empty disjunction $\bot$. Assume now that $0 < w_1 + \dots + w_m \le w$. Let I stand for $\{1, \dots, m\}$ and let I' be $\{1, \dots, m-1\}$.

For any subset J of I, by ΣJ we denote the sum $\sum_{i\in J} w_i$. Then

$$
\begin{aligned}
[w \leq S] &= \mathop{;}\limits_{J\subseteq I \,:\, \Sigma J\geq w} \Big(\mathop{,}\limits_{i\in J} c_i \Big) \\
&\leftrightarrow \mathop{;}\limits_{J\subseteq I' \,:\, \Sigma J\geq w} \Big(\mathop{,}\limits_{i\in J} c_i \Big);\ \mathop{;}\limits_{J\subseteq I \,:\, m\in J, \Sigma J\geq w} \Big(\mathop{,}\limits_{i\in J} c_i \Big) \\
&= [w \leq S'];\ \mathop{;}\limits_{J\subseteq I' \,:\, \Sigma J + w_m\geq w} \Big(\mathop{,}\limits_{i\in J\cup\{m\}} c_i \Big) \\
&\leftrightarrow [w \leq S'];\big(c_m,\ \mathop{;}\limits_{J\subseteq I' \,:\, \Sigma J\geq w - w_m} \Big(\mathop{,}\limits_{i\in J} c_i \Big)\big) \\
&= [w \leq S'];(c_m, [(w-w_m)\leq S']).
\end{aligned}
$$

The proof of the second equivalence is similar. □

Lemma 10. *Program*

$$\{q \leftrightarrow Def(q) \ :\ q\in Q_\Omega\} \tag{5.15}$$

is strongly equivalent to

$$
\begin{aligned}
&\{q_{not\ l} \leftrightarrow not\ l \ :\ q_{not\ l}\in Q_\Omega\}\cup \\
&\{q_{w\leq S} \leftrightarrow [w\leq S] \ :\ q_{w\leq S}\in Q_\Omega\}\cup \\
&\{q_{w<S} \leftrightarrow [w< S] \ :\ q_{w<S}\in Q_\Omega\}.
\end{aligned} \tag{5.16}
$$

Proof. The rules of (5.16) can be obtained from the rules of (5.15) by replacing $Def(q_{w\leq S})$ with $[w\leq S]$ for the atoms $q_{w\leq S}$ in Q_Ω, and $Def(q_{w<S})$ with $[w<S]$ for the atoms $q_{w<S}$ in Q_Ω. Consequently, it is sufficient to show that, for every atom of the form $q_{w\leq S}$ in Q_Ω, the equivalences

$$Def(q_{w\leq S}) \leftrightarrow [w\leq S] \tag{5.17}$$

are derivable in the logic of here-and-there both from (5.15) and from (5.16), and similarly for atoms of the form $q_{w<S}$. The proofs for atoms of both kinds are similar, and we will only consider $q_{w\leq S}$. Let S be $\{c_1 = w_1, \ldots, c_m = w_m\}$.

The definition of $Def(q_{w\leq S})$ and the statement of Lemma 9 show that the right-hand side of (5.17) is equivalent to the result of replacing $q_{w\leq S'}$ in the left-hand side with $[w \leq S']$, and $q_{w-w_m\leq S'}$ with $[w - w_m \leq S']$. Since $q_{w\leq S'}$ and $q_{w-w_m\leq S'}$ belong to Q_Ω, this observation implies the derivability of (5.17) from (5.16).

The derivability of (5.17) from (5.15) will be proved by strong induction on m. If $w \leq 0$ or $w > w_1 + \cdots + w_m$ then, by the definition of $Def(q_{w\leq S})$ and by Lemma 9, (5.17) is provable in the logic of here-and-there. Assume that $0 < w \leq w_1 + \cdots + w_m$. Then $q_{w\leq S'}$ and $q_{w-w_m\leq S'}$ belong to Q_Ω, and, by the induction hypothesis, the equivalences

$$Def(q_{w\leq S'}) \leftrightarrow [w \leq S']$$

and

$$Def(q_{w-w_m\leq S'}) \leftrightarrow [w - w_m \leq S']$$

are derivable from (5.15). Consequently, the equivalences

$$q_{w\leq S'} \leftrightarrow [w \leq S']$$

and

$$q_{w-w_m\leq S'} \leftrightarrow [w - w_m \leq S']$$

are derivable from (5.15) as well. By Lemma 9, this implies the derivability of (5.17).

□

Theorem 2. *For any program Ω with weight constraints, $Z \mapsto Z \setminus Q_\Omega$ is a 1–1 correspondence between the answer sets for $[\Omega]^{nn}$ and the answer sets for Ω.*

Proof. From Lemmas 8 and 10 we see that $[\Omega]^{nn}$ has the same answer sets as the union of Π and (5.16). Furthermore, this union is strongly equivalent to the union of $[\Omega]^{nd}$ and (5.16). Indeed, Π consists of the rules

$$\begin{aligned}
&l \leftarrow \textit{not}\ q_{\textit{not}\ l}, [C_1]^{nn}, \ldots, [C_n]^{nn},\\
&\bot \leftarrow \textit{not}\ q_{L_0 \leq S_0}, [C_1]^{nn}, \ldots, [C_n]^{nn},\\
&\bot \leftarrow q_{U_0 < S_0}, [C_1]^{nn}, \ldots, [C_n]^{nn}
\end{aligned}$$

for every rule

$$L_0 \leq S_0 \leq U_0 \leftarrow C_1, \ldots, C_n$$

in Ω and every positive head element l of that rule; $[\Omega]^{nd}$ consists of the rules

$$\begin{aligned}
&l \leftarrow \textit{not not}\ l, [C_1], \ldots, [C_n],\\
&\bot \leftarrow \textit{not}\ [L_0 \leq S_0, S_0 \leq U_0], [C_1], \ldots, [C_n].
\end{aligned}$$

It is easy to derive each of these two programs from the other program and (5.16) in the logic of here-and-there. Consequently, $[\Omega]^{nn}$ has the same answer sets as the union of $[\Omega]^{nd}$ and (5.16). By the completion lemma, it follows that $[\Omega]^{nn}$ has the same answer sets as the union of $[\Omega]^{nd}$ and the program

$$\begin{aligned}
&\{q_{\textit{not}\ l} \leftarrow \textit{not}\ l \ : \ q_{\textit{not}\ l} \in Q_\Omega\} \cup\\
&\{q_{w \leq S} \leftarrow [w \leq S] \ : \ q_{w \leq S} \in Q_\Omega\} \cup\\
&\{q_{w < S} \leftarrow [w < S] \ : \ q_{w < S} \in Q_\Omega\}.
\end{aligned}$$

The assertion of Theorem 2 follows now by the lemma on explicit definitions. □

5.3.3 Proof of Theorem 3(b)

In the following lemma, $L \leq S$ is a cardinality constraint such that S contains only positive integer weights; let $f(S)$ be the maximum between the cardinality $|S|$ of S and twice the number of weights in S that are greater than 1. For instance, if $S = \{p, \textit{not}\ q = 3, r = 2\}$, $f(S) = \max\{3, 2 \cdot 2\} = 4$. Function $f(S)$ offers an upper and lower bound for $|S|$: in fact, it is easy to verify that $f(S) \leq |S| \leq 2f(S)$.

Lemma 11. *At each execution of lines 5–12 in Figure 5.1, $f(S)$ never increases.*

Proof. We essentially need to compare $f(S)$ and $f(H)$ before H is assigned to S at line 12. Since S cannot contain more than $f(S)/2$ weights greater than 1, H is initially assigned — at line 6 — at most $f(S)/2$ elements. Then line 9 adds at most $|R|/2 \leq |S|/2 \leq f(S)/2$ auxiliary atoms (their weights are implicitly 1) to H. Consequently, at the end, $|H| \leq f(S)$, and the number of weights greater than 1 in H doesn't exceed $f(S)/2$. We can conclude that $f(H) \leq f(S)$. □

Theorem 3(b). *For any program Ω,* `wc2cc` *terminates in time polynomial in the size of the input.*

Proof. Consider Figure 5.1. The algorithm can be divided into three parts: lines 1–2, lines 3–14, and lines 15–16. It is sufficient to show that each part terminates in a time polynomial in the size of its input (in this case, the value of Ω and Ω' before the execution of that part of code). This is easy to be verified for lines 1–2 and 15–16. It remains to show that lines 3–14 require polynomial time.

The external **foreach** loop is clearly executed a linear number of times. Inside that loop, the **while** loop is executed a number of times linear to the the size of the largest weight in S, since that weight is halved at each iteration. (Recall that the size of a number is the number of bits required to memorize the number.) It remains to consider the time needed in each iteration of lines 5–12.

Let $L_0 \leq S_0$ be the value of $L \leq S$ before the first iteration of lines 5–12. Each iteration of such lines is polynomial in the size of the current $L \leq S$. The rest of the proof consists in showing that $L \leq S$ doesn't become more than polynomially larger than $L_0 \leq S_0$, so that the time of an execution of lines 5–12 is polynomially bounded by the size of $L_0 \leq S_0$.

Since L always decreases at each iteration, it remains to consider the size of S only. The size of S is polynomially bounded by

(i) the number $|S|$ of rules elements in it,

(ii) the size of the atom in each rule element, and

(iii) the total size of weights (we omit "$= w$" in $c = w$ when $w = 1$).

For part (iii), the weights always decrease, so their total size decrease. For part (i), we know that $f(S)$ — an upper bound for the value of $|S|$ — never increases by Lemma 11. It remains to notice that the value of $f(S)$ before the first iteration of lines 5–12 is not more than twice the initial value of $|S|$. □

5.3.4 Proof of Theorem 3(a)

For this proof, we extend the syntax of a weight constraint $L \leq S \leq U$ in the body of a rule, by allowing each element in S to be of the form $F = w$, where F is an arbitrary nested expression instead of a rule element. We call those weight constraints *extended weight constraints*. Clearly, the original definition of an answer set for programs with weight constraints in not applicable. However, we can easily extend our definition of $[L \leq S \leq U]$ to this new kind of weight constraints, and define the concept of an answer set for programs with extended weight constraints in terms of nested expressions.

Here are some lemmas about (extended) weight constraints. The first two are immediate from the definition of $[L \leq S]$.

Lemma 12. *For any two extended weight constraints $L_1 \leq S$ and $L_2 \leq S$ with $L_1 \geq L_2$, $[L_1 \leq S]$ entails $[L_2 \leq S]$.*

Lemma 13. *For any extended weight constraint $L \leq S$ where L and all weights in S are even, $[L \leq S] = [(L-1) \leq S]$.*

Lemma 14. *For any extended weight constraints $L_1 \leq S, \ldots, L_n \leq S$ ($n \geq 1$) where $L_1 \leq L_2 \leq \cdots \leq L_n$, and any integer $i = 1, \ldots, n$,*

$$[i \leq \{[L_1 \leq S], \ldots, [L_n \leq S]\}] \leftrightarrow [L_i \leq S].$$

Proof. Let F be the left-hand side of the claim. By Lemma 12,

$$F = \mathop{;}_{X \subseteq \{1,\dots,n\}\,:\,i \leq |X|} \Big(\mathop{,}_{j \in X} [L_j \leq S] \Big) \leftrightarrow \mathop{;}_{X \subseteq \{1,\dots,n\}\,:\,i \leq |X|} [L_{\max(X)} \leq S]$$

Since $|X| \geq i$, the value of $\max(X)$ ranges from i (when $X = \{1, \dots, i\}$) through n (when $n \in X$). Consequently, by Lemma 12 again, the disjunction in the last expression above can be simplified into $[L_i \leq S]$. □

For any weight constraint $L \leq S$, we denote by ΣS the sum of the weights in S.

Lemma 15. *Consider any extended weight constraint $L \leq S$ where all weights are positive integers and L is a nonnegative integer. Let S_1 and S_2 be any partitioning of S in two (multi)sets. Then*

$$[L \leq S] \leftrightarrow \mathop{;}_{i=0,\dots,\Sigma S_1} ([i \leq S_1], [L - i \leq S_2]).$$

Proof. Let F and G be the left-hand side and the right-hand side of the above equivalence. Let S be $\{F_1 = w_1, \dots, F_{|S|} = w_{|S|}\}$ We assume, without losing in generality, that the elements of S_1 are the first $|S_1|$ elements of S. In the following formulas, we omit mentioning that $X \subseteq \{1, \dots, |S|\}$, $X_1 \subseteq \{1, \dots, |S_1|\}$ and $X_2 \subseteq$

$\{|S_1|+1,\ldots,|S|\}$. For each X, by W_X we denote $\sum_{i\in X} w_i$.

$$F = [L \le S] = \mathop{;}\limits_{X\ :\ L\le W_X} \big(\mathop{,}\limits_{j\in X} F_j\big)$$

$$\leftrightarrow \mathop{;}\limits_{X_1,X_2\ :\ L\le W_{X_1}+W_{X_2}} \big(\mathop{,}\limits_{j\in X_1} F_j, \mathop{,}\limits_{j\in X_2} F_j\big)$$

$$\leftrightarrow \mathop{;}\limits_{X_1} \big(\mathop{;}\limits_{X_2\ :\ L-W_{X_1}\le W_{X_2}} \big(\mathop{,}\limits_{j\in X_1} F_j, \mathop{,}\limits_{j\in X_2} F_j\big)\big)$$

$$\leftrightarrow \mathop{;}\limits_{X_1} \big(\big(\mathop{,}\limits_{j\in X_1} F_j\big), \big(\mathop{;}\limits_{X_2\ :\ L-W_{X_1}\le W_{X_2}} \big(\mathop{,}\limits_{j\in X_2} F_j\big)\big)\big)$$

$$\leftrightarrow \mathop{;}\limits_{X_1} \big(\big(\mathop{,}\limits_{j\in X_1} F_j\big), [(L-W_{X_1}) \le S_2]\big).$$

On the other hand, considering that $\Sigma S_1 \ge W_{X_1}$ for all X_1,

$$G = \mathop{;}\limits_{i=0,\ldots,\Sigma S_1} \big(\big(\mathop{;}\limits_{X_1\ :\ i\le W_{X_1}} \big(\mathop{,}\limits_{j\in X_1} F_j\big)\big), [L-i \le S_2]\big)$$

$$\leftrightarrow \mathop{;}\limits_{\substack{X_1\\ i=0,\ldots,W_{X_1}}} \big(\big(\mathop{,}\limits_{j\in X_1} F_j\big), [L-i \le S_2]\big)$$

$$\leftrightarrow \mathop{;}\limits_{X_1} \big(\big(\mathop{,}\limits_{j\in X_1} F_j\big), \big(\mathop{;}\limits_{i=0,\ldots,W_{X_1}} [L-i \le S_2]\big)\big).$$

It remains to notice that

$$\mathop{;}\limits_{i=0,\ldots,W_{X_1}} [L-i \le S_2] \leftrightarrow [L-W_{X_1} \le S_2]$$

by Lemma 12. □

Lemma 16. *For every extended weight constraint $L \leq S$ where all weights are positive integers and L is a nonnegative integer, replacing two elements $F = w_1$ and $F = w_2$ in S with a single element $F = w_1 + w_2$ preserves strong equivalence of $[L \leq S]$.*

This assertion is actually true even if we allow the weights to be arbitrary nonnegative numbers. However, the less general form is sufficient for our purposes and it is easier to prove.

Proof of Lemma 16. Let S_1 be $\{F = w_1, F = w_2\}$ and S_2 be $\{F = w_1 + w_2\}$. It is easy to verify that, for any integer i,

$$[i \leq S_1] \leftrightarrow [i \leq S_2].$$

Then, for any integer L and any set expression S', by Lemma 15,

$$\begin{aligned} [L \leq (S_1 \cup S')] &\leftrightarrow \mathop{;}_{i=0,\ldots,\Sigma S_1} ([i \leq S_1], [N - i \leq S']) \\ &\leftrightarrow \mathop{;}_{i=0,\ldots,\Sigma S_2} ([i \leq S_2], [N - i \leq S']) \\ &\leftrightarrow [L \leq (S_2 \cup S')] \end{aligned}$$

□

Given a weight constraint $L \leq S$ where $S = \{c_1 = w_1, \ldots, c_n = w_m\}$, all weights are positive integers and L is a nonnegative integer, let H' be

$$\{[i \leq R] : 0 < i \leq |R|, i + L \text{ is even}\} \cup \{c_i = \lfloor w_i/2 \rfloor : i \in \{1, \ldots, n\}, w_i > 1\}, \quad (5.18)$$

where R is the set as computed in line 5 of Figure 5.1.

Lemma 17.

$$[\lceil L/2 \rceil \leq H'] \leftrightarrow [L \leq S]$$

Proof. We consider the case in which L is even. The other case is similar. Let H'_1 and H'_2 be the first and second term of the union (5.18) respectively. By Lemma 15,

$$[\lceil L/2 \rceil \leq H'] \leftrightarrow \mathop{;}\limits_{i=0,\ldots,|H'_1|} ([i \leq H'_1], [L/2 - i \leq H'_2])$$

For $i = 0$ then both $i \leq H'_1$ and $2i \leq R$ are clearly equivalent to $\top$. For $i > 0$, we use Lemma 14 to rewrite $i \leq H'_1$ as $2i \leq R$. Consequently,

$$[\lceil L/2 \rceil \leq H'] \leftrightarrow \mathop{;}\limits_{i=0,\ldots,\lfloor |R|/2 \rfloor} ([2i \leq R], [L/2 - i \leq H'_2])$$

Let T be H'_2 with all weights doubled. Then

$$[\lceil L/2 \rceil \leq H'] \leftrightarrow \mathop{;}\limits_{i=0,\ldots,\lfloor |R|/2 \rfloor} ([2i \leq R], [L - 2i \leq T])$$

$$\leftrightarrow \mathop{;}\limits_{i=0,\ldots,|R|,\ i \text{ is even}} ([i \leq R], [L - i \leq T])$$

Consider each disjunctive term above. In view of Lemma 12, $[i + 1 \leq R]$ entails $[i \leq R]$; moreover, since $L - i$ is even, by Lemma 13, $[L - i \leq T] = [L - (i + 1) \leq T]$. Consequently, $([(i + 1) \leq R], [L - (i + 1) \leq T])$ entails $([i \leq R], [L - i \leq T])$. Consequently, we can add disjunctive terms $([i \leq R], [L - i \leq T])$ relative to odd i's while preserving strong equivalence. It follows, by Lemma 15 again, that

$$[\lceil L/2 \rceil \leq H'] \leftrightarrow \mathop{;}\limits_{i=0,\ldots,|R|} ([i \leq R], [L - i \leq T])$$

$$\leftrightarrow [L \leq (R \cup T)].$$

It remains to notice that $[L \leq (R \cup T)]$ is strongly equivalent to $[L \leq S]$ by Lemma 16.

□

Recall that, at any step in our algorithm, Ω may not be a program with weight constraints, since it may contain occurrences of weight constraints $L \leq S$ preceded by negation *not*. By $\hat{\Omega}$ we denote the program with weight constraints obtained from Ω by replacing each expression of the form *not* $(L \leq S)$ by $S \leq U$. We agree to write $[\hat{\Omega}]$ simply as $[\Omega]$: this doesn't lead to ambiguity when Ω is a program with weight constraints, since, in this case, $\hat{\Omega} = \Omega$.

Lemma 18. *Let Ω_2 be Ω with one occurrence of a weight constraint $L \leq S$ (possibly prefixed by not) in a body of a rule replaced by another weight constraint $L' \leq S'$. Then $[\Omega_2]$ coincides with the result of replacing, in $[\Omega]$, one occurrence of $[L \leq S]$ with $[L' \leq S']$.*

Proof. If the occurrence of $L \leq S$ in Ω is not prefixed by *not* then it is not hard to see that the same relationship between Ω and Ω_2 holds between $\hat{\Omega}$ and $\hat{\Omega}_2$. If we consider that the transformation into nested expressions replaces $L \leq S$ with $[L \leq S]$ and $L' \leq S'$ with $[L' \leq S']$ then the claim is obvious. Otherwise, $\hat{\Omega}_2$ is $\hat{\Omega}$ with one occurrence of $S \leq (U + 1)$ replaced by one occurrence of $S' \leq (U' + 1)$. It remains to notice that $S \leq (U + 1)$ is translated, in $[\Omega]$, as *not* $[L \leq S]$, and $S' \leq (U' + 1)$ is translated, in $[\Omega_2]$, as *not* $[L' \leq S']$. □

Lemma 19. *Consider one execution of lines 5–12 of the algorithm in Figure 5.1. Let Ω_1 and Ω'_1 be the initial values of Ω and Ω' respectively, and let Ω_2 and Ω'_2 be the same sets after one execution of lines 5–12. Let D be the set of auxiliary atoms d introduced at line 8. Then $X \mapsto X \setminus D$ is a 1–1 correspondence between the answer sets of $\hat{\Omega}_2 \cup \Omega'_2$ and the answer sets of $\hat{\Omega}_1 \cup \Omega'_1$.*

Proof. For each $d \in D$, let $Def(d)$ be the formula $[i \leq R]$ for which rule $d \leftarrow i \leq R$ has been added to Ω'_1 at line 10. In view of Theorem 1, we can compare the answer sets of $[\Omega_2 \cup \Omega'_2] = [\Omega_2] \cup [\Omega'_2]$ and the answer sets of $[\Omega_1 \cup \Omega'_1] = [\Omega_1] \cup [\Omega'_1]$. The difference between Ω_1 and Ω_2 is that Ω_2 contains $\lceil L/2 \rceil \leq H$ in place of $L \leq S$.

Then, by Lemma 18, the difference between $[\Omega_1]$ and $[\Omega_2]$ is that $[\Omega_2]$ contains $[\lceil L/2 \rceil \leq H]$ in place of $[L \leq S]$. Moreover, $[\Omega'_2]$ is essentially $[\Omega'_2]$ plus rules of the form (after a few simplifications)

$$d \leftarrow Def(d). \tag{5.19}$$

Let Δ be the set of rules

$$Def(d) \leftarrow d$$

for all $d \in D$. Since each of such atoms $d \in D$ occurs in the head of a rule of $[\Omega_2] \cup [\Omega'_2]$ uniquely in (5.19), then, by the Completion Lemma, $[\Omega_2] \cup [\Omega'_2]$ has the same answer sets as $[\Omega_2] \cup [\Omega'_2] \cup \Delta$.

Let Γ be $[\Omega_1]$ with the occurrence of $[L \leq S]$ replaced by $[\lceil L/2 \rceil \leq H']$, where H' is defined as (5.18). Program Γ can also be seen as the result of replacing, in $[\Omega'_1]$ every auxiliary atom $d \in D$ with $Def(d)$. Since $[\Omega'_2] \cup \Delta$ contains

$$d \leftrightarrow [i \leq S],$$

for each $d \in D$, we can strongly equivalently rewrite $[\Omega_2] \cup [\Omega'_2] \cup \Delta$ as $\Gamma \cup [\Omega'_2] \cup \Delta$. This program can also be rewritten as $[\Omega] \cup [\Omega'_2] \cup \Delta$, since Γ is $[\Omega]$ with the occurrence of $[L \leq S]$ replaced by the strongly equivalent formula (by Lemma 17) $[\lceil L/2 \rceil \leq H']$. Finally, we can drop Δ from program $[\Omega_1] \cup [\Omega'_2] \cup \Delta$ by the Completion Lemma.

To sum up, we showed that $[\Omega_2] \cup [\Omega'_2]$ has the same answer sets of $[\Omega_1] \cup [\Omega'_2]$. It remains to notice that $X \mapsto X \setminus D$ is a 1–1 correspondence between the answer set for $[\Gamma] \cup [\Omega'_2]$ and for $[\Gamma] \cup [\Omega']$ by the Lemma on Explicit definitions, since the only occurrences of auxiliary atoms $d \in D$ in $[\Gamma] \cup [\Omega'_2]$ are the heads of rules $[\Omega'_2] \setminus [\Omega'_1]$. □

Theorem 3(a). *For any Ω, the mapping $Z \mapsto Z \cap \sigma$ is a 1–1 correspondence between the answer sets of* `wc2cc`(Ω) *and the answer sets of* Ω.

Proof. Consider Figure 5.1. It is easy to verify that the value of $\hat{\Omega} \cup \Omega'$ after the execution of line 2 equals the value of Ω given as a parameter to the procedure. Moreover, the program with nested expression returned by the procedure is identical to the value of $\hat{\Omega} \cup \Omega'$ before the execution of line 15. Consequently, it is sufficient to show that $Z \mapsto Z \cap \sigma$ is a 1–1 correspondence between the answer sets of $\hat{\Omega} \cup \Omega'$ after line 2 and the answer sets of $\hat{\Omega} \cup \Omega'$ before line 15.

Between lines 2 and 14, programs Ω and Ω' are modified only inside the **while** loop (lines 5–12). Lemma 19 shows that the answer sets of $\hat{\Omega} \cup \Omega'$ are preserved by each iteration of lines 5–12, with the possible addition — to each answer set — of auxiliary atoms. The claim is then immediate. □

Chapter 6

Answer Sets for Propositional Theories

Recall (see Section 3.6) that a rule with nested expression is seen, in equilibrium logic, as a propositional formula — over atoms that may contain strong negation — of a special kind: it is an implication whose antecedent and consequent don't contain other implications [Pearce, 1997].

In this section we present an extension of the definition of an answer set, which allows every "rule" to be any arbitrary propositional formula.

This new definition of an answer set turns out to be equivalent to the concept of an equilibrium model. This fact is important for several reasons. First of all, theorems about equilibrium models — for instance the characterization of strong equivalence in terms of the logic of here-and-there — hold for the new definition of an answer set. Second, this new definition of an answer set provides a simpler characterization of an equilibrium model, originally defined in terms of Kripke models.

We will also show how we can extend many useful theorems about logic programs to arbitrary propositional theories.

In the next chapter we will show how we can use this new definition of an

answer set to represent aggregates.

Proofs of the theorems stated in this chapter are presented in Section 6.5.

6.1 Formulas, reducts and answer sets

As in equilibrium logic, we consider (propositional) formulas formed from atoms of the form a and $\sim a$ (with strong negation) and connectives $\bot$, $\vee$, $\wedge$ and $\rightarrow$. [1]

A rule with nested expression can be seen as a formula as in Section 3.6. A *theory* is a set of formulas. In view of the relationship between theories and logic programs, we will sometimes call theories as programs. In the rest of the chapter, F and G denote formulas, Γ a theory, X and Y coherent sets of atoms (that is, sets of atoms that don't contain a and $\sim a$ for the same a), and $\otimes$ a binary connective.

As we did in Section 3.5, we identify an interpretation with the set of atoms satisfied by it, and we write $X \models F$ ($X \models \Gamma$) if X satisfies F (or Γ) in the sense of classical logic.

The *reduct* F^X of F relative to X is defined recursively:

- if $X \not\models F$ then $F^X = \bot$,
- if $X \models a$ (a is an atom) then $a^X = a$, and
- if $X \models F \otimes G$ then $(F \otimes G)^X = F^X \otimes G^X$.

This definition of reduct is similar to a transformation proposed in [Osorio *et al.*, 2004, Section 4.2].

The reduct F^X can be alternatively defined as the formula obtained from F by replacing every outermost subformula not satisfied by X with $\bot$ (this alternative definition applies even if we treat $\neg$, $\top$ and $\leftrightarrow$ as primitive connectives).

[1] $\neg F$ stands for $F \rightarrow \bot$; $\top$ stands for $\bot \rightarrow \bot$; $F \leftrightarrow G$ stands for $(F \rightarrow G) \wedge (G \rightarrow F)$.

For instance, if X contains p but not q then

$$((p \rightarrow q) \vee (q \rightarrow p))^X = \bot \vee (\bot \rightarrow p).$$

It is easy to see that, for every X, Y, $\otimes$, F and G,

$$Y \models (F \otimes G)^X \text{ iff } X \models F \otimes G \text{ and } Y \models F^X \otimes G^X. \tag{6.1}$$

The *reduct* Γ^X of Γ relative to X is $\{F^X : F \in \Gamma\}$. A set X is an *answer set* for Γ if X is a minimal set satisfying Γ^X.

For instance, let Γ be $\{(p \rightarrow q) \vee (q \rightarrow p), p\}$. Set $\{p\}$ is an answer set for Γ because $\{p\}$ is a minimal model satisfying the reduct $\{\bot \vee (\bot \rightarrow p), p\}$. It is not difficult to see that no other set of atoms is an answer set for Γ.

6.2 Relationship to equilibrium logic and to the traditional definition of reduct

Theorem 4. *For any theory, its models in the sense of equilibrium logic are identical to its answer sets.*

Since in application to programs with nested expressions equilibrium logic is equivalent to the semantics defined in [Lifschitz *et al.*, 1999], Theorem 4 implies that our definition of an answer set extends the corresponding definition from that paper.

In the language of propositional formulas, a nested expression can be seen as a formula that contains no implications $F \rightarrow G$ with $G \neq \bot$, and no equivalences. In a program with nested expressions, the "rules" are implications with nested expressions in the antecedent and the consequent. In application to programs with nested expressions, our definition of the reduct is quite different from the traditional

definition [Lifschitz *et al.*, 1999]. Consider, for instance, the following program:

$$p \leftarrow not\ q$$
$$q \leftarrow not\ r$$

According to [Lifschitz *et al.*, 1999], its reduct relative to $\{r\}$ is

$$p \leftarrow \top$$
$$q \leftarrow \bot;$$

under our definition, it is

$$\bot$$
$$q \leftarrow \bot.$$

The first reduct is satisfied, for instance, by $\{p\}$, while the second is unsatisfiable. However, some similarities between these formalisms exist. For instance, it is easy to see that for any formula F, $(\neg F)^X$, according to the new definition, is $\top$ when $X \not\models F$, and $\bot$ otherwise, as with the traditional definition. Indeed, if $X \models F$ then $X \not\models F \rightarrow \bot$ and consequently

$$(\neg F)^X = (F \rightarrow \bot)^X = \bot.$$

Otherwise, $X \models F \rightarrow \bot$, so that

$$(\neg F)^X = (F \rightarrow \bot)^X = F^X \rightarrow \bot = \bot \rightarrow \bot = \top.$$

The following proposition states a more general relationship between the new definition of the reduct and the traditional one. We denote by $F^{\underline{X}}$ the reduct of a nested expression F relative to X according to the definition from [Lifschitz *et al.*, 1999], and similarly for the reduct of a program.

Proposition 7. *For any program* Π *with nested expressions and any set* X *of atoms,* Π^X *is equivalent, in the sense of classical logic,*

- *to* $\bot$*, if* $X \not\models \Pi$*, and*

- *to the program obtained from* $\Pi^{\underline{X}}$ *by replacing all atoms that do not belong to* X *by* $\bot$*, otherwise.*

Corollary 1. *Given two sets of atoms* X *and* Y *with* $Y \subseteq X$ *and any program* Π*,* $Y \models \Pi^X$ *iff* $X \models \Pi$ *and* $Y \models \Pi^{\underline{X}}$*.*

This corollary suggests another way to verify that the definition of an answer set proposed in this paper is equivalent to the usual one in the case of programs with nested expressions. If $X \not\models \Pi$ then X is not an answer set for Π under either semantics. Otherwise, for every subset Y of X, $Y \models \Pi^X$ iff $Y \models \Pi^{\underline{X}}$ by Corollary 1.

6.3 Properties of propositional theories

Several theorems about answer sets for logic programs can be extended to propositional theories.

Two theories Γ_1 and Γ_2 are *strongly equivalent* if, for every theory Γ, $\Gamma_1 \cup \Gamma$ and $\Gamma_2 \cup \Gamma$ have the same answer sets.

Proposition 8. *For any two theories* Γ_1 *and* Γ_2*, the following conditions are equivalent:*

(i) Γ_1 *is strongly equivalent to* Γ_2*,*

(ii) Γ_1 *is equivalent to* Γ_2 *in the logic of here-and-there, and*

(iii) for each set X *of atoms,* Γ_1^X *is equivalent to* Γ_2^X *in classical logic.*

The two characterizations of strong equivalence stated in this theorem are similar to the ones between logic programs reviewed in Section 3.7. The equivalence between (i) and (ii) is essentially Lemma 4 from [Lifschitz *et al.*, 2001]. The equivalence between (i) and (iii) is similar to Theorem 1 from [Turner, 2003]. However, ours is not only more general — applicable to arbitrary propositional formulas — but also simpler.

To state several theorems below, we need the following definitions. An occurrence of an atom in a formula is *positive* if it is in the antecedent of an even number of implications. An occurrence is *strictly positive* if such number is 0, and *negative* if it odd. For instance, in a formula $(p \to r) \to q$, the occurrences of p and q are positive, the one of r is negative, and the one of q is strictly positive.

The following proposition is an extension of the property that in each answer set of a program, each atom occurs in the head of a rule of that program [Lifschitz, 1996, Section 3.1]. An atom is an head *head atom* of a theory Γ if it has a strictly positive occurrence in Γ. [2]

Proposition 9. *Each answer set for a theory* Γ *consists of heads atoms of* Γ.

In a logic program, adding constraints (rules whose heads are $\bot$) to a program Π removes the answer sets of Π that don't satisfy the constraints. As a formula, a constraint has the form $\neg F$, a typical formula without head atoms. Next theorem generalizes the property of logic programs stated above to propositional theories.

Proposition 10. *For every two propositional theories* Γ_1 *and* Γ_2 *such that* Γ_2 *has no head atoms, a consistent set* X *of atoms is an answer set for* $\Gamma_1 \cup \Gamma_2$ *iff* X *is an answer set for* Γ_1 *and* $X \models \Gamma_2$.

The following two propositions are generalizations of propositions stated in Section 5.3.1 in the case of logic programs. We say that an occurrence of an atom is in the scope of negation when it occurs in an implication $F \to \bot$.

Proposition 11 (Lemma on Explicit Definitions)**.** *Let* Γ *be any propositional theory, and* Q *a set of atoms without strong negation such that, for each* $q \in Q$*, neither* q *nor* $\sim q$ *occur in* Γ*. For each* $q \in Q$*, let* $Def(q)$ *be a formula that doesn't contain any atoms from* Q*. Then* $X \mapsto X \setminus Q$ *is a 1–1 correspondence between the answer sets for* $\Gamma \cup \{Def(q) \to q : q \in Q\}$ *and the answer sets for* Γ.

[2] In case of programs with nested expressions, it is easy to check that head atoms are atoms that occur in the head of a rule outside the scope of negation.

Proposition 12 (Completion Lemma)**.** *Let* Γ *be any propositional theory, and* Q *a set of atoms without strong negation such that all positive occurrences of atoms from* Γ *in* Γ *are in the scope of negation. For each* $q \in Q$*, let* $Def(q)$ *be a formula such that all negative occurrences of atoms from* Q *in* $Def(q)$ *are in the scope of negation. Then* $\Gamma \cup \{Def(q) \rightarrow q : q \in Q\}$ *and* $\Gamma \cup \{Def(q) \leftrightarrow q : q \in Q\}$ *have the same answer sets.*

The following proposition is essentially a generalization of the splitting set theorem from [Lifschitz and Turner, 1994] and [Erdoğan and Lifschitz, 2004], which allows to break logic programs/propositional theories into parts and compute the answer sets separately.

Proposition 13 (Splitting Set Theorem)**.** *Let* Γ_1 *and* Γ_2 *be two theories such that* Γ_1 *doesn't contain head atoms of* Γ_2*. A set* X *of atoms is an answer set for* $\Gamma_1 \cup \Gamma_2$ *iff there is an answer set* Y *of* Γ_1 *such that* X *is an answer set for* $\Gamma_2 \cup Y$*.*

6.4 Computational complexity

Since the concept of an answer set is equivalent to the concept of an equilibrium model, checking the existence of an answer set for a propositional theory is a Σ_2^P-complete problem as for equilibrium models [Pearce *et al.*, 2001].

In a logic program, if the head of each rule is an atom or $\bot$ then the existence of an answer set is NP-complete [Marek and Truszczyński, 1991]. We may wonder if the existence of an answer set for a theory consisting of implications of the form

$$F \rightarrow a$$

(a is an atom or $\bot$) is still in class NP. The answer is negative: the following proposition shows that as soon as we allow implications (that are not negations) in formula F then we have the same expressivity — and complexity — of disjunctive rules.

Proposition 14. *Rule*

$$l_1 \wedge \cdots \wedge l_m \to a_1 \vee \cdots \vee a_n$$

($n > 0, m \geq 0$) where $a_1, \ldots, a_n$ are atoms and $l_1, \ldots, l_m$ are literals, is strongly equivalent to the set of n implications ($i = 1, \ldots, n$)

$$(l_1 \wedge \cdots \wedge l_m \wedge (a_1 \to a_i) \wedge \cdots \wedge (a_n \to a_i)) \to a_i. \tag{6.2}$$

We will see, in the next chapter, that the conjunctive terms in the antecedent of (6.2) can equivalently be replaced by aggregates of a simple kind, thus showing that adding aggregates to the language of nondisjunctive programs increases the complexity.

6.5 Proofs

6.5.1 Proofs of Theorem 4 and Proposition 7

Lemma 20. *For any formulas $F_1, \ldots, F_n$ ($n \geq 0$), any set X of atoms, and any connective $\otimes \in \{\vee, \wedge\}$, $(F_1 \otimes \cdots \otimes F_n)^X$ is classically equivalent to $F_1^X \otimes \cdots \otimes F_n^X$.*

Proof. **Case 1**: $X \models F_1 \wedge \cdots \wedge F_n$. Then, by the definition of reduct, $(F_1 \wedge \cdots \wedge F_n)^X = F_1^X \wedge \cdots \wedge F_2^X$. **Case 2**: $X \not\models F_1 \wedge \cdots \wedge F_n$. Then $(F_1 \otimes \cdots \otimes F_n)^X = \bot$; moreover, one of $F_1, \ldots, F_n$ is not satisfied by X, so that one of $F_1^X, \ldots, F_n^X$ is $\bot$. The case of disjunction is similar. □

Lemma 21. *For any X and Y such that $X \subseteq Y$ and any theory Γ,*

$$X \models \Gamma^Y \text{ iff } (X, Y) \models \Gamma.$$

Proof. It is sufficient to consider the case when Γ is a singleton $\{F\}$. The proof is by induction on F.

- F is $\bot$. $X \not\models \bot$ and $(X, Y) \not\models \bot$.

- F is an atom a. $X \models a^Y$ iff $Y \models a$ and $X \models a$. Since $X \subseteq Y$, this means iff $X \models a$, which is the condition for which $(X, Y) \models a$.

- F has the form $G \wedge H$. $X \models (G \wedge H)^Y$ iff $X \models G^Y \wedge H^Y$ by Lemma 20, and then iff $X \models G^Y$ and $X \models H^Y$. This is equivalent, by induction hypothesis, to say that $(X, Y) \models G$ and $(X, Y) \models H$, and then that $(X, Y) \models G \wedge H$.

- The proof for disjunction is similar to the proof for conjunction.

- F has the form $G \rightarrow H$. $X \models (G \rightarrow H)^Y$ iff $X \models G^Y \rightarrow H^Y$ and $Y \models (G \rightarrow H)$, and then iff

  $$X \models G^Y \text{ implies } X \models H^Y, \text{ and } Y \models G \rightarrow H.$$

  This is equivalent, by the induction hypothesis, to

  $$(X, Y) \models G \text{ implies } (X, Y) \models H, \text{ and } Y \models G \rightarrow H,$$

  which is the definition of $(X, Y) \models G \rightarrow H$.

□

Theorem 4. *For any theory, its models in the sense of equilibrium logic are identical to its answer sets.*

Proof. A set Y is an equilibrium model of Γ iff

$$(Y, Y) \models \Gamma \text{ and, for all proper subsets } X \text{ of } Y,\ (X, Y) \not\models \Gamma.$$

In view of Lemma 21, this is equivalent to the condition

$$Y \models \Gamma^Y \text{ and, for all proper subsets } X \text{ of } Y,\ X \not\models \Gamma^Y.$$

which means that Y is an answer set for Γ. □

Lemma 22. *The reduct F^X of a nested expression F is equivalent, in the sense of classical logic, to the nested expression obtained from $F^{\underline{X}}$ by replacing all atoms that do not belong to X by $\bot$.*

Proof. The proof is by structural induction on F.

- When F is $\bot$ then $F^X = \bot = F^{\underline{X}}$.
- For an atom a, $a^{\underline{X}} = a$. The claim is immediate.
- Let F be a negation $\neg G$ (for instance, $\neg\bot = \top$). If $X \models G$ then $F^X = \bot = F^{\underline{X}}$; otherwise, $F^X = \neg\bot = \top = F^{\underline{X}}$.
- for $\otimes \in \{\vee, \wedge\}$, $(G \otimes H)^{\underline{X}}$ is $G^{\underline{X}} \otimes H^{\underline{X}}$, and, by Lemma 20, $(G \otimes H)^X$ is equivalent to $G^X \otimes H^X$. The claim now follows by the induction hypothesis.

□

Proposition 7. *For any program Π with nested expressions and any set X of atoms, Π^X is equivalent, in the sense of classical logic,*

- *to $\bot$, if $X \not\models \Pi$, and*
- *to the program obtained from $\Pi^{\underline{X}}$ by replacing all atoms that do not belong to X by $\bot$, otherwise.*

Proof. If $X \not\models \Pi$ then clearly Π^X contains $\bot$. Otherwise, Π^X consists of formulas $F^X \to G^X$ for each rule $G \leftarrow F \in \Pi$, and consequently for each rule $G^{\underline{X}} \leftarrow F^{\underline{X}} \in \Pi^{\underline{X}}$. Since each F and G is a nested expression, the claim is immediate by Lemma 22. □

6.5.2 Proofs of Propositions 8–10

Proposition 8. *For any two theories Γ_1 and Γ_2, the following conditions are equivalent:*

(i) Γ_1 *is strongly equivalent to* Γ_2*,*

(ii) Γ_1 *is equivalent to* Γ_2 *in the logic of here-and-there, and*

(iii) for each set X *of atoms,* Γ_1^X *is equivalent to* Γ_2^X *in classical logic.*

Proof. We will prove the equivalence between (i) and (ii) and between (ii) and (iii). We start with the former. Lemma 4 from [Lifschitz *et al.*, 2001] tells that, for any two theories, the following conditions are equivalent:

(a) for every theory Γ, theories $\Gamma_1 \cup \Gamma$ and $\Gamma_2 \cup \Gamma$ have the same equilibrium models, and

(b) Γ_1 is equivalent to Γ_2 in the logic of here-and-there.

Condition (b) is identical to (ii). Condition (a) can be rewritten, by Theorem 4, as

(a′) for every theory Γ, theories $\Gamma_1 \cup \Gamma$ and $\Gamma_2 \cup \Gamma$ have the same answer sets,

which means that Γ_1 is strongly equivalent to Γ_2.

It remains to prove the equivalence between (ii) and (iii). We have that

$$\Gamma_1 \text{ is equivalent to } \Gamma_2 \text{ in the logic of here-and-there}$$

iff, for every Y,

$$\text{for every } X \text{ such that } X \subseteq Y,\ (X,Y) \models \Gamma_1 \text{ iff } (X,Y) \models \Gamma_2.$$

This condition is equivalent to

$$\text{for every } X \subseteq Y,\ (X,Y) \models \Gamma_1 \text{ iff } (X,Y) \models \Gamma_2$$

and, by Lemma 21, to

$$\text{for every } X \subseteq Y,\ X \models \Gamma_1^Y \text{ iff } X \models \Gamma_2^Y.$$

Since Γ_1^Y and Γ_2^Y contain atoms from Y only (the other atoms are replaced by $\bot$ in the reduct), this last condition expresses equivalence between Γ_1^Y and Γ_2^Y. □

Lemma 23. *For any theory Γ, let S be a set of atoms that contains all head atoms of Γ. For any set X of atoms, if $X \models \Gamma$ then $X \cap S \models \Gamma^X$.*

Proof. It is clearly sufficient to prove the claim for Γ that is a singleton $\{F\}$. The proof is by induction on F.

- If $F = \bot$ then $X \not\models F$, and the claim is trivial.
- For an atom a, if $X \models a$ then $a^X = a$, so that, since $a \in S$, $X \cap S \models a^X$.
- If $X \models G \wedge H$ then $X \models G$ and $X \models H$. Consequently, by induction hypothesis, $X \cap S \models G^X$ and $X \cap S \models H^X$. It remains to notice that $(G \wedge H)^X = G^X \wedge H^X$.
- The case of disjunction is similar to the case of conjunction.
- If $X \models G \rightarrow H$ then $(G \rightarrow H)^X = G^X \rightarrow H^X$. Assume that $X \cap S \models G^X$. Then $X \models G$. Consequently, since $X \models G \rightarrow H$, $X \models H$. Since S contains all head atoms of H, the claim follows by the induction hypothesis.

□

Lemma 24. *For any theory Γ and any set X of atoms, $X \models \Gamma^X$ iff $X \models \Gamma$.*

Proof. Reduct Γ^X is obtained from Γ by replacing some subformulas that are not satisfied by X with $\bot$. □

Proposition 9. *Each answer set for a theory Γ consists of heads atoms of Γ.*

Proof. Consider any theory Γ, the set S of head atoms of Γ, and an answer set X of Γ. By Lemma 24, $X \models \Gamma$, so that, by Lemma 23, $X \cap S \models \Gamma^X$. Since $X \cap S \subseteq X$ and no proper subset of X satisfies Γ^X, it follows that $X \cap S = X$, and consequently that $X \subseteq S$. □

Proposition 10. *For every two propositional theories Γ_1 and Γ_2 such that Γ_2 has no head atoms, a consistent set X of atoms is an answer set for $\Gamma_1 \cup \Gamma_2$ iff X is an answer set for Γ_1 and $X \models \Gamma_2$.*

Proof. If $X \models \Gamma_2$ then Γ_2^X is satisfied by every subset of X by Lemma 23, so that $(\Gamma_1 \cup \Gamma_2)^X$ is classically equivalent to Γ_1^X; then clearly X is an answer set for $\Gamma_1 \cup \Gamma_2$ iff it is an answer set for Γ_1. Otherwise, Γ_2^X contains $\bot$, and X cannot be an answer set for $\Gamma_1 \cup \Gamma_2$. □

6.5.3 Proofs of Propositions 11 and 13

We start with the proof of Proposition 13. Some lemmas are needed.

Lemma 25. *If X is a answer set for Γ then Γ^X is equivalent to X.*

Proof. Since all atoms that occur in Γ^X belong to X, it is sufficient to show that the formulas are satisfied by the same subsets of X. By the definition of an answer set, the only subset of X satisfying Γ^X is X. □

Lemma 26. *Let S be a set of atoms that contains all atoms that occur in a theory Γ_1 but does not contain any head atoms of a theory Γ_2. For any set X of atoms, if X is a answer set for $\Gamma_1 \cup \Gamma_2$ then $X \cap S$ is an answer set for Γ_1.*

Proof. Since X is an answer set for $\Gamma_1 \cup \Gamma_2$, $X \models \Gamma_1$, so that $X \cap S \models \Gamma_1$, and, by Lemma 24, $X \cap S \models \Gamma_1^{X \cap S}$. It remains to show that no proper subset Y of $X \cap S$ satisfies $\Gamma_1^{X \cap S}$. Let S' be the set of head atoms of Γ_2, and let Z be $X \cap (S' \cup Y)$. We will show that Z has the following properties:

(i) $Z \cap S = Y$;

(ii) $Z \subset X$;

(iii) $Z \models \Gamma_2^X$.

To prove (i), note that since S' is disjoint from S, and Y is a subset of $X \cap S$,

$$Z \cap S = X \cap (S' \cup Y) \cap S = X \cap Y \cap S = (X \cap S) \cap Y = Y.$$

To prove (ii), note that set Z is clearly a subset of X. It cannot be equal to X, because otherwise we would have, by (i),

$$Y = Z \cap S = X \cap S;$$

this is impossible, because Y is a proper subset of $X \cap S$. Property (iii) follows from Lemma 23, because $X \models \Gamma_2$, and $S' \cup Y$ contains all head atoms of Γ_2.

Since X is a answer set for $\Gamma_1 \cup \Gamma_2$, from property (ii) we can conclude that $Z \not\models (\Gamma_1 \cup \Gamma_2)^X$. Consequently, by property (iii), $Z \not\models \Gamma_1^X$. Since all atoms that occur in Γ_1 belong to S, $\Gamma_1^X = \Gamma_1^{X \cap S}$, so that we can rewrite this formula as $Z \not\models \Gamma_1^{X \cap S}$. Since all atoms that occur in $\Gamma_1^{X \cap S}$ belong to S, it follows that $Z \cap S \not\models \Gamma_1^{X \cap S}$. By property (i), we conclude that $Y \not\models \Gamma_1^{X \cap S}$. □

Proposition 13. *Let Γ_1 and Γ_2 be two theories such that Γ_1 doesn't contain head atoms of Γ_2. A set X of atoms is an answer set for $\Gamma_1 \cup \Gamma_2$ iff there is an answer set Y of Γ_1 such that X is an answer set for $\Gamma_2 \cup Y$.*

Proof. Take theories Γ_1 and Γ_2 such that Γ_1 does not contain any head atoms of Γ_2, and let S be the set of atoms that occur in Γ_1. Observe first that if a set X of atoms is an answer set for $\Gamma_2 \cup Y$ then $X \cap S = Y$. Indeed, by Lemma 26 with Y as Γ_1, $X \cap S$ is an answer set for Y, and the only answer set of Y is Y. Consequently, the assertion to be proved can be reformulated as follows: a set X of atoms is an answer set for $\Gamma_1 \cup \Gamma_2$ iff

(i) $X \cap S$ is an answer set for Γ_1, and

(ii) X is an answer set for $\Gamma_2 \cup (X \cap S)$.

If $X \cap S$ is not an answer set for Γ_1 then X is not an answer set for $\Gamma_1 \cup \Gamma_2$ by Lemma 26. Now suppose that $X \cap S$ is an answer set for Γ_1. Then, by Lemma 25, $\Gamma_1^{X \cap S}$ is equivalent to $X \cap S$. Consequently,

$$\begin{aligned}(\Gamma_1 \cup \Gamma_2)^X \;=\; \Gamma_1^X \cup \Gamma_2^X \;=\; \Gamma_1^{X\cap S} \cup \Gamma_2^X \;\leftrightarrow\; X \cap S \;\cup\; \Gamma_2^X \\ =\; (X \cap S)^X \cup \Gamma_2^X \;=\; \big((X \cap S) \cup \Gamma_2\big)^X\end{aligned}$$

We can conclude that X is an answer set for $\Gamma_1 \cup \Gamma_2$ iff X is an answer set for $\Gamma_2 \cup (X \cap S)$. ☐

Proposition 11. *Let Γ be any propositional theory, and Q a set of atoms without strong negation such that, for each $q \in Q$, neither q nor $\sim q$ occur in Γ. For each $q \in Q$, let $Def(q)$ be a formula that doesn't contain any atoms from Q. Then $X \mapsto X \setminus Q$ is a 1–1 correspondence between the answer sets for $\Gamma \cup \{Def(q) \rightarrow q : q \in Q\}$ and the answer sets for Γ.*

Proof. Let Γ_2 be $\{Def(q) \rightarrow q : q \in Q\}$. In view of the splitting set theorem (Proposition 13), a set X is an answer set for $\Gamma \cup \Gamma_2$ iff

there is an answer set Y for Γ such that X is an answer set for $\Gamma_2 \cup Y$.

By reasoning similar to one used in the proof of Proposition 13, the only set Y such that X is an answer set for $\Gamma_2 \cup Y$ is $X \setminus Q$. Then the splitting set theorem (Proposition 13) tells us that X is an answer set for $\Gamma \cup \Gamma_2$ iff

$X \setminus Q$ is an answer set for Γ and X is answer set for $\Gamma_2 \cup (X \setminus Q)$.

Clearly, if X is an answer set for $\Gamma \cup \Gamma_2$ then $X \setminus Q$ is an answer set for Γ. Now take any answer set Y for Γ. We need to show that there is exactly one answer set X of $\Gamma \cup \Gamma_2$ such that $X \setminus Q = Y$. In view of the splitting set theorem, it is sufficient to show that

$$Z = \{q \in Q : X \setminus Q \models Def(q)\} \cup Y$$

is the only answer set X for $\Gamma_2 \cup Y$ such that $X \setminus Q = Y$. If X doesn't contain an element $q \in Q$ such that $X \models Def(q)$ then $X \not\models \Gamma_2$, so that X is not an answer set for $\Gamma_2 \cup Y$. In all other cases, $X \models \Gamma_2$, so that $(\Gamma_2 \cup Y)^X$ is equivalent to

$$\{Def(q)^X \to q : q \in Q, X \models Def(q)\} \cup Y.$$

In this reduct, each $Def(q)^X$ is classically equivalent to $\top$. Indeed, first of all, since $Y = X \setminus Q$, all atoms that occur in $Def(q)^X$ belong to Y, and then we can replace such occurrences with $\top$, obtaining a formula that is either equivalent to $\top$ or $\bot$. It remains to notice that since $X \models Def(q)$ then $X \models Def(q)^X$.

Theory $(\Gamma_2 \cup Y)^X$ is then equivalent to the set Z defined above. We can conclude that the only answer set X for $\Gamma_2 \cup Y$ such that $X \models Def(q)$ is Z. ☐

6.5.4 Proof of Proposition 12

In order to prove the Completion Lemma, we will need the following lemma.

Lemma 27. *Take any three sets X, Y and S of atoms such that $Y \subseteq X$. For any formula F,*

(a) if each positive occurrence of an atom from S in F is in the scope of negation and $Y \models F^X$ then $Y \setminus S \models F^X$, and

(b) if each negative occurrence of an atom from S in F is in the scope of negation and $Y \setminus S \models F^X$ then $Y \models F^X$.

Proof. We prove the two claims simultaneously by induction on F.

- If $X \not\models F$ then $F^X = \bot$, and the claim is trivial. This covers the case in which $F = \bot$.
- If $X \models F$ and F is an atom a then claim (b) holds because if $a \in Y \setminus S$ then $a \in Y$. For claim (a), if $a \notin S$ and $a \in Y$ then $a \in Y \setminus S$.

- If $X \models F$ and they are a conjunction or a disjunction, the claim is almost immediate by Lemma 20 and induction hypothesis.
- The case in which $X \models F$ and F has the form $G \rightarrow H$ remains. Clearly, $(G \rightarrow H)^X = G^X \rightarrow H^X$. We describe a proof of claim (a). The proof for (b) is similar. **Case 1**: $H = \bot$. Then, since $X \models F$, $F^X = \top$, and the claim clearly follows. **Case 2**: $H \neq \bot$. Assume that no atom from S has a positive occurrence in $G^X \rightarrow H^X$ outside the scope of the negation, and that $Y \setminus S \models G^X$. We want to show that $Y \setminus S \models H^X$. Notice that no atom from S has a negative occurrence in G^X outside the scope of negation; consequently, by the induction hypothesis (claim (b)), $Y \models G^X$. On the other hand, $Y \models (G \rightarrow H)^X$, so that $Y \models H^X$. Since no atom from S has a positive occurrence in H^X outside the scope of negation, we can conclude that $Y \setminus S \models H^X$ by induction hypothesis (claim (a)).

□

Proposition 12. *Let Γ be any propositional theory, and Q a set of atoms without strong negation such that all positive occurrences of atoms from Γ in Γ are in the scope of negation. For each $q \in Q$, let $Def(q)$ be a formula such that all negative occurrences of atoms from Q in $Def(q)$ are in the scope of negation. Then $\Gamma \cup \{Def(q) \rightarrow q : q \in Q\}$ and $\Gamma \cup \{Def(q) \leftrightarrow q : q \in Q\}$ have the same answer sets.*

Proof. Let Γ_1 be $\Gamma \cup \{Def(q) \rightarrow q \ : \ q \in Q\}$ and Let Γ_2 be $\Gamma_1 \cup \{q \rightarrow Def(q) \ : \ q \in Q\}$. We want to prove that a set X is an answer set for both theories or for none of them. Since $\Gamma_1^X \subseteq \Gamma_2^X$, Γ_2^X entails Γ_1^X. If the opposite entailment holds also then we clearly have that Γ_2^X and Γ_1^X are satisfied by the same subsets of X, and the claim immediately follows. Otherwise, for some $Y \subseteq X$, $Y \not\models \Gamma_2^X$ and $Y \models \Gamma_1^X$. First of all, that means that $X \models \Gamma_1$, so that Γ_1^X is equivalent to

$$\Gamma^X \cup \{Def(q)^X \rightarrow q \ : \ q \in Q \cap X\}.$$

Secondly, set Y is one of the sets Y' having the following properties:

(i) $Y' \setminus Q = Y \setminus Q$, and

(ii) $Y' \models Def(q)^X \rightarrow q$ for all $q \in Q \cap X$.

Let Z be the intersection of such sets Y', and let Δ be $\{q \rightarrow Def(q)^X \ : \ q \in Q \cap X\}$. Set Z has the following properties:

(a) $Z \subseteq Y$,

(b) $Z \models \Gamma_1^X$, and

(c) $Z \models \Delta$.

Indeed, claim (a) holds since Y is one of the elements Y' of the intersection. To prove (b), first of all, we observe that $Z \setminus Q = Y \setminus Q$, so that, by (a), there is a set $S \subseteq Q$ such that $Z = Y \setminus S$; as $Y \models \Gamma^X$ and Γ has all positive occurrences of atoms from $S \subseteq Q$ in the scope of negation, it follows that $Z \models \Gamma^X$ by Lemma 27(a). It remains to show that, for any q, if $Z \models Def(q)^X$ then $q \in Z$. Assume that $Z \models Def(q)^X$. Then, since $Def(q)$ has all negative occurrences of atoms from Q in the scope of negation, and since all Y' whose intersection generate Z are superset of Z with $Y' \setminus Z \subseteq Q$, all those Y' satisfy $Def(q)^X$ by Lemma 27. By property (ii), we have that $q \in Y'$ for all Y', and then $q \in Z$.

It remains to prove claim (c). Take any $q \in Z$. Set $Y' = Z \setminus \{q\}$ satisfies condition (i), but it cannot satisfy (ii), because sets Y' that satisfy (i) and (ii) are supersets of Z by construction of Z. Consequently, $Y' \not\models Def(q)^X$. Since all positive occurrences of atom q in $Def(q)$ are in the scope of negation and $Y' = Z \setminus \{q\}$, we can conclude that $Z \not\models Def(q)^X$ by Lemma 27 again.

Now consider two cases. If $X \not\models \Gamma_2$ then clearly X is not an answer for Γ_2. It is not an answer set for Γ_1 as well. Indeed, since $X \models \Gamma_1$, we have that, for some $q \in Q \cap X$, $X \not\models Def(q)$. Consequently, $Def(q)^X = \bot$ and then $X \not\models \Delta$, but, since

$Z \models \Delta$ by (c) and $Z \subseteq Y \subseteq X$ by (a), Z is a proper subset of X. Since $Z \models \Gamma_1^X$ by (b), X is not an answer set for Γ_1.

In the other case ($X \models \Gamma_2$) it is not hard to see that Γ_2^X is equivalent to $\Gamma_1^X \cup \Delta$. We have that $Z \models \Gamma_1^X$ by (b), and then $Z \models \Gamma_2^X$ by (c). Since $Y \not\models \Gamma_2^X$, $Z \neq Y$. On the other hand, $Z \subseteq Y \subseteq X$ by (a). This means that Z is a proper subset of X that satisfies Γ_1^X and Γ_2^X, and we can conclude that X is not an answer set for any of Γ_1 and Γ_2. □

6.5.5 Proof of Proposition 14

Proposition 14. *Rule*

$$l_1 \wedge \cdots \wedge l_m \to a_1 \vee \cdots \vee a_n \tag{6.3}$$

($n > 0, m \geq 0$) where $a_1, \dots, a_n$ are atoms and $l_1, \dots, l_m$ are literals, is strongly equivalent to the set of n implications ($i = 1, \dots, n$)

$$(l_1 \wedge \cdots \wedge l_m \wedge (a_1 \to a_i) \wedge \cdots \wedge (a_n \to a_i)) \to a_i. \tag{6.4}$$

Proof. Let F be (6.3) and G_i ($i = 1, \dots, n$) be (6.4). We want to prove that F is strongly equivalent to $\{G_1, \dots, G_n\}$ by showing that F^X is classically equivalent to $\{G_1^X, \dots, G_n^X\}$. Let H be $l_1 \wedge \cdots \wedge l_m$.

Case 1: $X \not\models H$. Since H is a conjunctive term of F and all G_i's, it is easy to verify that their reducts relative to X are all equivalent to $\bot$. **Case 2**: $X \models H$ and $X \not\models F$. Then clearly $F^X = \bot$. But Γ^X is $\bot$: indeed, since $X \not\models F$, $X \not\models l_i$ for all $i = 1, \dots, m$. It follows that the consequent of each G_i is not satisfied by X, but the antecedent is satisfied, because $X \models H$ and in each implication $a_j \to a_i$ in G_i, the antecedent is not satisfied. **Case 3**: $X \models H$ and $X \models F$. This means that some of $a_1, \dots, a_n$ belong to X. Assume, for instance, that $a_1, \dots, a_p$ ($0 < p \leq n$) belong

to X, and $a_{p+1}, \ldots, a_n$ don't. Then F^X is equivalent to $H^X \rightarrow (a_1 \vee \cdots \vee a_p)$. Now consider formula G_i. If $i > p$ then the consequent a_i is not satisfied by X, but also the antecedent is not: it contains an implication $a_1 \rightarrow a_i$; consequently G_i^X is $\top$. On the other hand, if $i \leq p$ then the consequent a_i is satisfied by X, as well as each implication $a_j \rightarrow a_i$ in the antecedent of G_i. After a few simplifications, we can rewrite G_i^X as

$$(H^X \wedge (a_1 \rightarrow a_i) \wedge \cdots \wedge (a_p \rightarrow a_i)) \rightarrow a_i.$$

It is not hard to see that this formula is classically equivalent to F^X, so that the claim easily follows. $\square$

Chapter 7

A New Definition of an Aggregate

Aggregates are an important construct in answer set programming 2.2.3. In this section, we provide a new definition of an aggregate. They are introduced in the syntax of propositional theories (Chapter 6) in a natural way, essentially behaving — in some sense — as propositional connectives.

This definition has three nice properties. First of all, it is very general and uniform. It allows, for instance, both disjunctive rules and choice rules. It also allows aggregates nested into each other. It is unclear if nested aggregates may be useful in an ASP-program, but they can be useful for theoretical purposes: for instance, we used nested weight constraints to prove Theorem 3(a).

Second, our definition of an aggregate doesn't give the unintuitive results of weight constraints and PDB-aggregates that were mentioned in Sections 3.2 and 3.3. We will see that when aggregates are monotone or antimonotone (see Section 7.3), all definitions of an aggregates are essentially equivalent to each other; in the most general case, our definition of an aggregate is a generalization of FLP-aggregates.

Finally, it turns out (Section 7.2) that an arbitrary aggregate can be rewritten

as a propositional formula, reducing the syntax to the one of propositional theories proposed in Chapter 6. This provides a way to apply theorems about propositional theories (Section 6.3) to programs with aggregates. For instance, the proof of Proposition 17 uses several of such theorems.

7.1 Representing Aggregates

A *formula with aggregates* is defined recursively as follows:

- atoms of the form a and $\sim a$, and $\bot$ are formulas with aggregates,
- propositional combinations formed from formulas with aggregates using connectives $\vee$, $\wedge$ and $\rightarrow$ are formulas with aggregates, and
- any expression of the form

$$op\langle\{F_1 = w_1, \dots, F_n = w_n\}\rangle \prec N \tag{7.1}$$

  where

  - op is (a symbol for) a function from multisets of real numbers to $\mathcal{R} \cup \{-\infty, +\infty\}$ (such as sum, product, min, max, etc.),
  - $F_1, \dots, F_n$ are formulas with aggregates, and $w_1, \dots, w_n$ are (symbols for) real numbers ("weights"),
  - $\prec$ is (a symbol for) a binary relation between real numbers, such as $\leq$ and $=$, and
  - N is (a symbol for) a real number,

  is a formula with aggregates.

A *(ground) aggregate* is a formula with aggregates of the form (7.1). A *theory with aggregates* is a set of formulas with aggregates.

Recall that a PDB-aggregate was introduced in Section 3.3 as an expression of the form (7.1) also, except that $F_1, \dots, F_n$ are literals; similarly, in FLP-aggregates (Section 3.4), $F_1, \dots, F_n$ are conjunctions of atoms.

We extend the recursive definition of satisfaction for propositional formulas to formulas with aggregates, by adding a clause for aggregates: a coherent set X of atoms *satisfies* an aggregate (7.1) if $op(W) \prec N$ for the multiset W consisting of the weights w_i $(1 \leq i \leq n)$ such that $X \models F_i$. As usual, we say that X satisfies a theory Γ if X satisfies all formulas in Γ.

For example,

$$sum\langle\{p = 1, q = 1\}\rangle \neq 1 \tag{7.2}$$

is satisfied by sets of atoms that contain both p and q or none of them.

The definition of reduct for formulas with aggregates extends the one of propositional formulas (Section 6.1), with the case of aggregates: for an aggregate A of the form (7.1),

$$A^X = \begin{cases} op\langle\{F_1^X = w_1, \dots, F_n^X = w_n\}\rangle \prec N, & \text{if } X \models A, \\ \bot, & \text{otherwise.} \end{cases}$$

This is similar to the clause for binary connectives:

$$(F \otimes G)^X = \begin{cases} F^X \otimes G^X, & \text{if } X \models F \otimes G, \\ \bot, & \text{otherwise.} \end{cases}$$

As in the case of propositional formulas, we can see the reduct process for a formula with aggregates as the result of replacing with $\bot$ each maximal subformula not satisfied by X.

As usual, the *reduct* Γ^X of a theory Γ with aggregates relative to a coherent set X of atoms is the set of reducts F^X for all $F \in \Gamma$. Set X is an *answer set* for Γ if X is a minimal model of Γ^X.

Consider, for instance, the theory Γ consisting of one formula

$$sum\langle\{p = -1, q = 1\}\rangle \geq 0 \to q. \tag{7.3}$$

Set $\{q\}$ is an answer set for Γ. Indeed, since both the antecedent and consequent of (7.3) are satisfied by $\{q\}$, $\Gamma^{\{q\}}$ is

$$sum\langle\{\bot = -1, q = 1\}\rangle \geq 0 \to q.$$

The antecedent of the implication above is satisfied by any set of atoms, so the whole formula is equivalent to q. Consequently, $\{q\}$ is the minimal model of $\Gamma^{\{q\}}$, and then an answer set for Γ.

7.2 Aggregates as Propositional Formulas

The introduction of the concept of a formula/theory with aggregates is actually not necessary. In fact, we can identify (7.1) with the formula

$$\bigwedge_{I \subseteq \{1,\ldots,n\}\ :\ op(\{w_i\ :\ i \in I\}) \not\prec N} \big((\bigwedge_{i \in I} F_i) \to (\bigvee_{i \in \overline{I}} F_i) \big), \tag{7.4}$$

where $\overline{I}$ stands for $\{1, \ldots, n\} \setminus I$, and $\not\prec$ is the negation of $\prec$.

For instance, if we consider aggregate (7.2), the conjunctive terms in (7.4) correspond to the cases when the sum of weights is 1, that is, when $I = \{1\}$ and $I = \{2\}$. The two implications are $q \to p$ and $p \to q$ respectively, so that (7.2) is

$$(q \to p) \wedge (p \to q). \tag{7.5}$$

Similarly,

$$sum\langle\{p = 1, q = 1\}\rangle = 1 \tag{7.6}$$

is

$$(p \vee q) \wedge \neg(p \wedge q). \tag{7.7}$$

Even though (7.2) can be seen as the negation of (7.2), the negation of (7.7) is not strongly equivalent to (7.5) (although they are classically equivalent). This shows that it is generally incorrect to "move" a negation from a binary relation symbol (such as $\neq$) in front of the aggregate as the unary connective $\neg$.

Next proposition shows that this understanding of aggregates as propositional formulas is equivalent to the semantics for theories with aggregates of the previous section.

Proposition 15. *Let A be an aggregate of the form (7.1), and let G be the corresponding formula (7.4). For any coherent sets X and Y of atoms,*

(a) $X \models G$ iff $X \models A$, and

(b) $Y \models G^X$ iff $Y \models A^X$.

It can be shown using Proposition 8 from Section 6.3 that if we want to identify (7.1) with a formula so that Proposition 15 holds then (7.4) is the only choice, modulo strong equivalence.

Treating aggregates as propositional formulas allows us to apply many properties of propositional theories presented in the previous chapter to theories with aggregates also. We then have the concept of an head atom, of strong equivalence, we can use the completion lemma and so on. We will use several of those properties to prove Proposition 17 below. In the rest of the chapter we will often make no distinctions between the two ways of defining the semantics of aggregates discussed here.

7.3 Monotone Aggregates

An aggregate $op\langle\{F_1 = w_1, \ldots, F_n = w_n\}\rangle \prec N$ is *monotone* if, for each pair of multisets W_1, W_2 such that $W_1 \subseteq W_2 \subseteq \{w_1, \ldots, w_n\}$, $op(W_2) \prec N$ is true whenever

$op(W_1) \prec N$ is true. The definition of an *antimonotone* aggregate is similar, with $W_1 \subseteq W_2$ replaced by $W_2 \subseteq W_1$.

For instance,

$$sum\langle\{p = 1, q = 1\}\rangle > 1. \tag{7.8}$$

is monotone, and

$$sum\langle\{p = 1, q = 1\}\rangle < 1. \tag{7.9}$$

is antimonotone. An example of an aggregate that is neither monotone nor antimonotone is (7.2).

Proposition 16. *For any aggregate $op\langle\{F_1 = w_1, \dots, F_n = w_n\}\rangle \prec N$, formula (7.4) is strongly equivalent to*

$$\bigwedge_{I \subseteq \{1,\dots,n\}\ :\ op(\{w_i\ :\ i \in I\}) \not\prec N} \big(\bigvee_{i \in \overline{I}} F_i\big) \tag{7.10}$$

if the aggregate is monotone, and to

$$\bigwedge_{I \subseteq \{1,\dots,n\}\ :\ op(\{w_i\ :\ i \in I\}) \not\prec N} \big(\neg \bigwedge_{i \in I} F_i\big) \tag{7.11}$$

if the aggregate is antimonotone.

In other words, if $op\langle S\rangle \prec N$ is monotone then the antecedents of the implications in (7.4) can be dropped. Similarly, in case of antimonotone aggregates, the consequents of these implications can be replaced by $\bot$. In both cases, (7.4) is turned into a nested expression, if $F_1, \dots, F_n$ are nested expressions.

For instance, the monotone aggregate (7.8) is

$$(p \vee q) \wedge (p \rightarrow q) \wedge (q \rightarrow p),$$

which is equivalent, in the logic of here and there, to

$$(p \vee q) \wedge q \wedge p$$

and then to $q \wedge p$. In the case of the antimonotone aggregate (7.9), the formula

$$((p \wedge q) \rightarrow \bot) \wedge (p \rightarrow q) \wedge (q \rightarrow p)$$

is equivalent, in the logic of here-and-there, to

$$\neg(p \wedge q) \wedge \neg p \wedge \neg q,$$

and then to $\neg p \wedge \neg q$.

On the other hand, if an aggregate is neither monotone nor antimonotone, it may be not possible to find a nested expression equivalent, in the logic of here-and-there, to (7.4), even if $F_1, \ldots, F_n$ are nested expressions. This is the case for (7.2). Indeed, let A denote (7.2). Considering that this expression stands for (7.5), it is easy to check that $(\{p\}, \{p, q\}) \not\models A$ and $(\emptyset, \{p, q\}) \models A$. On the other hand, for any nested expression F, if $(\{p\}, \{p, q\}) \not\models F$ then $(\emptyset, \{p, q\}) \not\models F$ (easily provable by structural induction.)

In some uses of ASP, aggregates that are neither monotone nor antimonotone are essential, as discussed in the next Section.

7.4 Example

We consider the following variation of the combinatorial auction problem [Baral and Uyan, 2001], which can be naturally formalized using an aggregate that is neither monotone nor antimonotone.

Joe wants to move to another town and has the problem of removing all his bulky furniture from his old place. He has received some bids: each bid may be for one piece or several pieces of furniture, and the amount offered can be negative (if the value of the pieces is lower than the cost of removing them). A junkyard will take any object not sold to bidders, for a price. The goal is to find a collection of bids for which Joe doesn't lose money, if there is any.

Assume that there are n bids, denoted by atoms $b_1, \ldots, b_n$. We express by the formulas

$$b_i \vee \neg b_i \tag{7.12}$$

($1 \leq i \leq n$) that Joe is free to accept any bid or not. Clearly, Joe cannot accept two bids that involve the selling of the same piece of furniture. So, for every such pair i, j of bids, we include the formula

$$\neg(b_i \wedge b_j). \tag{7.13}$$

Next, we need to express which pieces of the furniture have not been given to bidders. If there are m objects we can express that an object i is sold by bid j by adding the rule

$$b_j \rightarrow s_i \tag{7.14}$$

to our theory.

Finally, we need to express that Joe doesn't lose money by selling his items. This is done by the aggregate

$$sum\langle\{b_1 = w_1, \ldots, b_n = w_n, \neg s_1 = -c_1, \ldots, \neg s_m = -c_m\}\rangle \geq 0, \tag{7.15}$$

where each w_i is the amount of money (possibly negative) obtained by accepting bid i, and each c_i is the money requested by the junkyard to remove item i. Note that (7.15) is neither monotone nor antimonotone.

Proposition 17. *$X \mapsto X \cap \{b_1, \ldots, b_n\}$ is a 1–1 correspondence between the answer sets of the theory consisting of formulas (7.12)–(7.15) and the sets of accepted bids $b_1, \ldots, b_n$ that are solutions of this problem.*

7.5 Computational Complexity

Since theories with aggregates generalize disjunctive problems, the problem of the existence of an answer set for a theory with aggregates clearly is Σ_2^P-hard. We need

to check in which class of the computational hierarchy this problem belongs.

If we represent aggregates as formulas (7.4) then a theory with aggregates is just a propositional theory, and we saw in Section 6.4 that this problem is in class Σ_2^P. However, we should consider that a formula (7.4) can be exponentially larger that the original aggregate, so this is generally not a good way of representing aggregates from a computational point of view.

We can avoid the growth in size by using the semantics of Section 7.1, and it turns out that the computation is not harder than for propositional theories.

Proposition 18. *If, for every aggregate, computing $op(W) \prec N$ requires polynomial time, then the existence of an answer set for a theory with aggregates is a Σ_2^P-complete problem.*

For a logic program with nested expressions, if the heads are atoms or $\bot$ then the existence of an answer set is NP-complete. If we allow nonnested aggregates in the body, for instance by allowing rules

$$A_1 \wedge \cdots \wedge A_n \rightarrow a$$

($A_1, \ldots, A_n$ are aggregates and a is an atom or $\bot$) then the complexity increases to Σ_2^P. This follows from Proposition 14, since, in (6.2), each formula l_i is the propositional representation of $sum\langle\{l_i = 1\}\rangle \geq 1$; similarly, each $a_j \rightarrow a_i$ is the propositional representation of $sum\langle\{a_j = -1, a_i = 1\}\rangle \geq 0$.

However, if we allow monotone and antimonotone aggregates only — even nested — in the antecedent, we are in class NP.

Proposition 19. *Consider theories with aggregates consisting of formulas of the form*

$$F \rightarrow a,$$

where a is an atom or $\bot$, and F contains monotone and antimonotone aggregates only, and no implications other than negations. If, for every aggregate, computing

$op(W) \prec N$ requires polynomial time then the problem of the existence of an answer set for theories of this kind is an NP-complete problem.

Similar results have been independently proven in [Calimeri *et al.*, 2005] for FLP-aggregates.

7.6 Other Formalisms

7.6.1 Programs with weight constraints

Recall that a weight constraint $L \leq S$ has the same intuitive meaning of $sum\langle S\rangle \geq L$, and $S \leq U$ has the same intuitive meaning of $sum\langle S\rangle \leq U$. Next theorem shows that there is indeed a relationship between them.

Theorem 5. *If $L \leq S$ and $S \leq U$ are weight constraints where all weights are nonnegative,*

(a) $[L \leq S]$ is strongly equivalent to $sum\langle S\rangle \geq L$, and

(b) $[S \leq U]$ is strongly equivalent to $sum\langle S\rangle \leq U$.

The theorem above and Theorem 1 shows that our concept of a general aggregate captures the concept of weight constraints defined in [Niemelä and Simons, 2000], when all weights are nonnegative. When we consider negative weights, however, such correspondence doesn't hold. (As mentioned in the introduction of this chapter, our view of negative weights is equivalent to the one proposed in [Faber *et al.*, 2004].) For instance, we don't consider (3.3) to be the same as (3.4), while — under our semantics — program

$$p \leftarrow 0 \leq \{p = 2, p = -1\}$$

has the same answer set $\emptyset$ of (3.6).

While weight constraints with positive weights only are either monotone or antimonotone, this is not the case when negative weights are allowed as in (7.15). In particular, it may not be possible to represent an aggregate of this kind by a nested expression.

Under the semantics of [Niemelä and Simons, 2000], the existence of an answer set for a program Ω with weight constraints is an NP-complete problem even in presence of negative weights, since they reduce nonmonotone aggregates into monotone and antimonotone ones. On the other hand, under our semantics, the same problem is in class Σ_P^2.

7.6.2 PDB-aggregates

We reviewed the syntax and semantics of PDB-aggregates [Pelov *et al.*, 2003] in Section 3.3. Under the syntax of propositional formulas with aggregates, a PDB-aggregate has the form (7.1) where $F_1, \dots, F_n$ are literals. A program with PDB-aggregates has the form

$$A_1 \wedge \cdots \wedge A_m \rightarrow a$$

where $A_1, \dots, A_m$ are PDB-aggregates and a is an atom.

In case of monotone and antimonotone PDB-aggregates and in the absence of negation as failure, the semantics of Pelov *et al.* is equivalent to ours. The theorem refers to definitions given in Section 3.3.

Proposition 20. *For any monotone or antimonotone PDB-aggregates A of the form (7.1) where $F_1, \dots, F_n$ are atoms, A_{tr} is strongly equivalent to (7.4).*

The claim above is not necessarily true when either the aggregates are not monotone or antimonotone, or when some formula in the aggregate is a negative literal. Programs (3.10) and (3.12) are examples of programs of PDB-aggregates of those two kinds where the semantics of Pelov *et al.* and ours give different answer

sets. For those programs, our semantics seems to give more intuitive results: for us, (3.10) has the same answer sets of (3.11), and (3.12) has the same answer sets of (3.13).

7.6.3 FLP-aggregates

We will now show that our semantics of aggregates is an extension of the semantics proposed by Faber, Leone and Pfeifer [2004]. Under the syntax of propositional formulas extended with aggregates, an FLP-program is a set of formulas

$$A_1 \wedge \cdots \wedge A_m \wedge \neg A_{m+1} \wedge \cdots \wedge \neg A_p \rightarrow a_1 \vee \cdots \vee a_n \tag{7.16}$$

($n, m \geq 0$), where $a_1, \dots, a_n$ are atoms and $A_1, \dots, A_p$ are FLP-aggregates. An FLP-program is *positive* if, in each formula (7.16), $p = m$.

Theorem 6. *The answer sets for a positive FLP-program under our semantics are identical to its answer sets in the sense of [Faber* et al.*, 2004].*

The theorem doesn't apply to arbitrary FLP-aggregates for the different meaning that has negation $\neg$ in front of an aggregate. In case of [Faber *et al.*, 2004], $\neg op\langle S\rangle \prec N$ is semantically the same as $op\langle S\rangle \not\prec N$, while we have seen that this fact doesn't always hold in our semantics. As a program with FLP-aggregate can be easily rewritten as a positive program with FLP-aggregate, our definition of an aggregate essentially generalizes the one of [Faber *et al.*, 2004].

7.7 Proofs

Lemma 28. *Let F be formula (7.1). If $X \models F$ then F^X is classically equivalent to*

$$\bigwedge_{I \subseteq \{1,\dots,n\}\,:\, op(\{w_i\,:\, i \in I\}) \not\prec N} \left(\left(\bigwedge_{i \in I} F_i^X \right) \rightarrow \left(\bigvee_{i \in \overline{I}} F_i^X \right) \right). \tag{7.17}$$

Proof. Formula F^X is classically equivalent, in view of Lemma 20, to

$$\bigwedge_{I\subseteq\{1,\ldots,n\}\ :\ op(\{w_i\ :\ i\in I\})\not\prec N} ((\bigwedge_{i\in I} F_i) \rightarrow (\bigvee_{i\in\overline{I}} F_i))^X.$$

Notice that all implications in (7.1) are satisfied by X because $X \models F$. Consequently, F^X is classically equivalent to

$$\bigwedge_{I\subseteq\{1,\ldots,n\}\ :\ op(\{w_i\ :\ i\in I\})\not\prec N} ((\bigwedge_{i\in I} F_i)^X \rightarrow (\bigvee_{i\in\overline{I}} F_i)^X),$$

and then, by Lemma 20 again, to (7.17). □

Proposition 15. *Let A be an aggregate of the form (7.1), and let G be the corresponding formula (7.4). For any coherent sets X and Y of atoms,*

(a) $X \models G$ iff $X \models A$, and

(b) $Y \models G^X$ iff $Y \models A^X$.

Proof. We start with part (a). Consider formula H_I (where $I \subseteq \{1,\ldots,n\}$):

$$(\bigwedge_{i\in I} F_i) \rightarrow (\bigvee_{i\in\overline{I}} F_i).$$

For each coherent set X of atoms there is exactly one set I such that $X \not\models H_i$: the set I_X that consists of the i's such that $X \models F_i$. Consequently,

$$\begin{aligned} X \models G \quad &\text{iff} \quad H_{I_X} \text{ is not a conjunctive term of } G \\ &\text{iff} \quad op(\{w_i\ :\ i \in I_X\}) \prec N \\ &\text{iff} \quad op(\{w_i\ :\ X \models F_i\}) \prec N \\ &\text{iff} \quad X \models A. \end{aligned}$$

For (b), if $X \not\models A$ then $X \not\models G$ by (a) and then both reducts are $\bot$. Otherwise, by (a) again, $X \models G$, and then G^X is equivalent, in view of Lemma 28, to (7.17). By (a) again, this formula is satisfied by the same consistent sets of atoms that satisfy

$$op\langle\{F_1^X = w_1,\ldots,F_n^X = w_n\}\rangle \prec N,$$

which is A^X. □

Lemma 29. *For any aggregate* $op\langle\{F_1 = w_1, \ldots, F_n = w_n\}\rangle \prec N$*, formula (7.4) is classically equivalent to*

$$\bigwedge_{I \subseteq \{1,\ldots,n\} \,:\, op(\{w_i \,:\, i \in I\}) \not\prec N} \Big(\bigvee_{i \in \overline{I}} F_i\Big) \tag{7.18}$$

if the aggregate is monotone, and to

$$\bigwedge_{I \subseteq \{1,\ldots,n\} \,:\, op(\{w_i \,:\, i \in I\}) \not\prec N} \Big(\neg \bigwedge_{i \in I} F_i\Big)$$

if the aggregate is antimonotone.

Proof. Consider the case of a monotone aggregate first. Let G be (7.4), and H be (7.18). It is easy to verify that H entails G. The opposite direction remains. Assume H, and we want to derive every conjunctive term

$$\bigvee_{i \in \overline{I}} F_i \tag{7.19}$$

in G. For every conjunctive term D of the form (7.19) in G, $op(\{w_i \,:\, i \in I\}) \not\prec N$. As the aggregate is monotone then, for every subset I' of I, $op(\{w_i \,:\, i \in I'\}) \not\prec N$, so that the implication

$$\Big(\bigwedge_{i \in I'} F_i\Big) \rightarrow \Big(\bigvee_{i \in \overline{I'}} F_i\Big)$$

is a conjunctive term of H for all $I' \subseteq I$. Then, since $\overline{I'} = \overline{I} \cup (I \setminus I')$, ("⇒" denotes entailment, and "⇔" equivalence)

$$H \Rightarrow \bigwedge_{I' \subseteq I} ((\bigwedge_{i \in I'} F_i) \to (\bigvee_{i \in \overline{I'}} F_i))$$

$$\Leftrightarrow \bigwedge_{I' \subseteq I} (((\bigwedge_{i \in I'} F_i) \wedge \bigwedge_{i \in I' \setminus I} \neg F_i) \to (\bigvee_{i \in \overline{I}} F_i))$$

$$\Leftrightarrow (\bigvee_{I' \subseteq I} ((\bigwedge_{i \in I'} F_i) \wedge \bigwedge_{i \in I' \setminus I} \neg F_i)) \to D.$$

The antecedent of the implication is a tautology: for each interpretation X, the disjunctive term relative to $I' = \{i \in I : X \models F_i\}$ is satisfied by X. We can conclude that H entails D.

The proof for antimonotone aggregates is similar. □

Proposition 16. *For any aggregate* $op\langle\{F_1 = w_1, \dots, F_n = w_n\}\rangle \prec N$, *formula (7.4) is strongly equivalent to*

$$\bigwedge_{I \subseteq \{1,\dots,n\} \,:\, op(\{w_i \,:\, i \in I\}) \not\prec N} (\bigvee_{i \in \overline{I}} F_i)$$

if the aggregate is monotone, and to

$$\bigwedge_{I \subseteq \{1,\dots,n\} \,:\, op(\{w_i \,:\, i \in I\}) \not\prec N} (\neg \bigwedge_{i \in I} F_i)$$

if the aggregate is antimonotone.

Proof. Consider the case of a monotone aggregate first. Let G be (7.4), and H be (7.18). In view of Proposition 8, it is sufficient to show that G^X is equivalent to H^X in classical logic for all sets X. If $X \not\models H$ then also $X \not\models G$ by Lemma 29, so that both reducts are $\bot$. Otherwise ($X \models H$), by the same lemma, $X \models G$. Then,

by Lemma 28, G^X is classically equivalent to (7.17). On the other hand, it is easy to verify, by applying Lemma 20 to H^X twice, that H^X is classically equivalent to

$$\bigwedge_{I\subseteq\{1,\ldots,n\}\ :\ op(\{w_i\ :\ i\in I\})\not\prec N} (\bigvee_{i\in\overline{I}} F_i^X).$$

The claim now follows by Lemma 29.

The reasoning for nonmonotone aggregates is similar. □

For the proof of Proposition 17, let Γ be the theory consisting of formulas (7.12)–(7.15).

Lemma 30. *For any answer set X of Γ, X contains an atom s_i iff X contains an atom b_j such that bid j involves selling object i.*

Proof. Consider Γ as a propositional theory. We notice that

- formulas (7.14) can be strongly equivalently grouped as m formulas ($i = 1,\ldots,m$)

  $$(\bigwedge_{j=1,\ldots,n:\ \text{object } i \text{ is part of bid } j} b_j) \rightarrow s_i,$$

  and

- no other formula of Γ contains atoms of the form s_i outside the scope of negation.

Consequently, by the Completion lemma (Proposition 12), formulas (7.14) in Γ can be replaced by by m formulas ($i = 1,\ldots,m$)

$$(\bigwedge_{j=1,\ldots,n:\ \text{object } i \text{ is part of bid } j} b_j) \leftrightarrow s_i. \tag{7.20}$$

preserving the answer sets. It follows that every answer set for Γ must satisfy formulas (7.20), and the claim immediately follows. □

A solution of Joe's problem is a set of atoms b_i such that

(a) the relative bids involve selling disjoint sets of items,

(b) the sum of the money earned from the bids is greater than the money spent giving away the remaining items.

Proposition 17. *$X \mapsto X \cap \{b_1, \dots, b_n\}$ is a 1–1 correspondence between the answer sets of the theory consisting of formulas (7.12)–(7.15) and the sets of accepted bids $b_1, \dots, b_n$ that are solutions of this problem.*

Proof. Take any answer set X of Γ. Since X satisfies rules (7.13) of Γ, condition (a) is satisfied. Condition (b) is satisfies as well, because X contains exactly all atoms s_i sold in some bids by Lemma 30, and since X satisfies aggregate (7.15) that belongs to Γ.

Now consider a solution of Joe's problem. This determines which atoms of the form b_i belongs to a possible corresponding answer set X. Consequently, Lemma 30 determines also which atoms of the form s_j belong to X, reducing the candidate answer sets X to one. We need to show that this X is indeed an answer set for Γ. The reduct Γ^X consists of (after a few simplifications)

(i) all atoms b_i that belong to X (from (7.12)),

(ii) $\top$ from (7.13) since (a) holds,

(iii) (by Lemma 30) implications (7.14) such that both b_j and s_i belong to X, and

(iv) the reduct of (7.15) relative to X.

Notice that (i)–(iii) together are equivalent to X, so that every every proper subset of X doesn't satisfy Γ^X. It remains to show that $X \models \Gamma^X$. Clearly, X satisfies (i)–(iii). To show that X satisfies (iv) it is sufficient, by Lemma 24 (consider (7.15) as a propositional formula), to show that X satisfies (7.15): it does that by hypothesis (b). $\square$

7.7.1 Proof of Propositions 18 and 19

Lemma 31. *If, for every aggregate, computing* $op(W) \prec N$ *requires polynomial time, then*

(a) checking satisfaction of a theory with aggregates requires polynomial time, and

(b) computing the reduct of a theory with aggregates requires polynomial time.

Proof. Part (a) is easy to verify by structural induction. Computing the reduct essentially consists of checking satisfaction of subexpressions of each formula of the theory. Each check doesn't require too much time by (a). It remains to notice that each formula with aggregates has a linear number of subformulas. □

Proposition 18. *If, for every aggregate, computing* $op(W) \prec N$ *requires polynomial time, then the existence of an answer set for a theory with aggregates is a* Σ_2^P*-complete problem.*

Proof. Hardness follows from the fact that theories with aggregates are a generalization of theories without aggregates. To prove inclusion, consider that the existence of an answer set for a theory Γ is equivalent to:

$$\textbf{exists } X \text{ such that } \textbf{for all } Y, \text{ if } Y \subseteq X \text{ then } Y \models \Gamma^X \text{ iff } X = Y$$

It remains to notice that, in view of Lemma 31, checking

$$\text{if } Y \subseteq X \text{ then } Y \models \Gamma^X \text{ iff } X = Y$$

requires polynomial time. □

Lemma 32. *Let* F *be a formula with aggregates containing monotone and anti-monotone aggregates only, and no implications different from negations. For any consistent sets* X, Y *and* Z *such that* $Y \subseteq Z$, *if* $Y \models F^X$ *then* $Z \models F^X$.

```
function verifyAS(Γ, X}
    if X ⊭ Γ then return false
    Δ := {F^X → a : F → a ∈ Γ and X ⊨ a}
    Y := ∅
    while there is a formula G → a ∈ Δ such that Y ⊨ G and a ∉ X
        Y := Y ∪ {a}
    end while
    if Y = X then return true
    return false
```

Figure 7.1: A polynomial-time algorithm to find minimal models of special kinds of theories

Proof. Let G be F with each monotone aggregate replaced by (7.10) and each antimonotone aggregate replaced by (7.11). It is easy to verify that G is a nested expression. Nested expressions have all negative occurrences of atoms in the scope of negation, so if $Y \models G^X$ then $Z \models G^X$ by Lemma (27). It remains to notice that F^X and G^X are satisfied by the same sets of atoms by Propositions 16 and 15. ☐

Proposition 19. *Consider theories with aggregates consisting of formulas of the form*

$$F \to a, \tag{7.21}$$

where a is an atom or $\bot$, and F contains monotone and antimonotone aggregates only, and no implications other than negations. If, for every aggregate, computing $op(W) \prec N$ requires polynomial time then the problem of the existence of an answer set for theories of this kind is an NP-complete problem.

Proof. NP-hardness follows from the fact that theories with aggregates are a generalization of traditional programs, for which the same problem is NP-complete. For inclusion in NP, it is sufficient to show that the time required to check if a coherent set X of atoms is an answer set for Γ. An algorithm that does this test is in Figure 7.1. It is easy to verify that it is a polynomial time algorithm. It remains to

prove that it is correct. If $X \not\models \Gamma$ then it is trivial. Now assume that $X \models \Gamma$. It is sufficient to show that

(a) Δ is classically equivalent to Γ^X, and

(b) the last value of Y (we call it Z) is the unique minimal model of Δ.

Indeed, for part (a), we notice that, since $X \models \Gamma$, Γ^X is

$$\{F^X \to a^X \ : \ F \to a \in \Gamma \text{ and } X \models a\} \cup \{F^X \to a^X \ : \ F \to a \in \Gamma \text{ and } X \not\models a\}.$$

The first set is Δ. The second set (which includes the case in which $a = \bot$) is a set of $\bot \to \bot$. Indeed, each $a^X = \bot$, and since $X \models \Gamma$, X doesn't satisfy any F and then $F^X = \bot$.

For part (b) it is easy to verify that the **while** loop iterates as long as $Y \not\models \Delta$, so that $Z \models \Delta$. Now assume, in sake of contradiction, that there is a set Z' that satisfies Δ and that is not a superset of Z. Then consider, in the algorithm, the last value of Y that is a subset of Z', and the atom a added to that Y (that is, $a \notin Z'$). This means that Δ contains a formula $G \to a$ such that $Y \models G$. Recall that G stands for a formula of the form F^X, where F is a formula with aggregates with monotone and antimonotone aggregates only and without implications that are not negations. Consequently, by Lemma 32, $Z' \models G$. On the other hand, $a \notin Z'$, so $Z' \not\models G \to a$, contradicting the hypothesis that Z' is a model of Δ. $\square$

7.7.2 Proof of Theorem 5

Lemma 33. *Let $F(\mathbf{u})$ and $G(\mathbf{u})$ be two formulas that are AND-OR combinations of $\top$, $\bot$ and atoms from the list of atoms $\mathbf{u}$. If, for any list $\mathbf{H}$ of formulas of the same arity of $\mathbf{u}$, $F(\mathbf{H})$ is classically equivalent to $G(\mathbf{H})$, then, for the same lists $\mathbf{H}$ of formulas, $F(\mathbf{H})$ is strongly equivalent to $G(\mathbf{H})$.*

Proof. In view of Proposition 8, it is sufficient to show that, for every coherent set X of atoms, $(F(\mathbf{H}))^{\mathbf{X}}$ is classically equivalent to $(G(\mathbf{H}))^{\mathbf{X}}$. By Lemma 20, $(F(\mathbf{H}))^{\mathbf{X}}$ is classically equivalent to $F(\mathbf{H}^{\mathbf{X}})$, where $\mathbf{H}^{\mathbf{X}}$ is the result of applying the reduct operation to each element of $\mathbf{H}$. Similarly, $(G(\mathbf{H}))^{\mathbf{X}}$ is classically equivalent to $G(\mathbf{H}^{\mathbf{X}})$. It remains to notice that $F(\mathbf{H}^{\mathbf{X}})$ is classically equivalent to $G(\mathbf{H}^{\mathbf{X}})$ by hypothesis. $\square$

Lemma 34. *For every weight constraints $L \le S$ and $S \le U$ and any coherent set X of atoms,*

(a) $X \models [L \le S]$ iff $X \models sum\langle S\rangle \ge L$, and

(b) $X \models [S \le U]$ iff $X \models sum\langle S\rangle \le U$.

Proof. Immediate by the definition of satisfaction of aggregates (Section 7.1), the definition of $[L \le S]$ and $[S \le U]$, and Lemma 1. $\square$

Theorem 5. *If $L \le S$ and $S \le U$ are weight constraints where all weights are nonnegative,*

(a) $[L \le S]$ is strongly equivalent to $sum\langle S\rangle \ge L$, and

(b) $[S \le U]$ is strongly equivalent to $sum\langle S\rangle \le U$.

Proof. Let S be $\{F_1 = w_1, \ldots, F_n = w_n\}$. We start with (a). We know, by Lemmas 34 and 15(a), that, for any formulas $F_1, \ldots, F_n$, $[L \le S]$ is classically equivalent to formula (7.1) corresponding to aggregate $sum\langle S\rangle \ge L$. In view of Lemma 33, it is then sufficient to show that both formulas can be written as AND-OR combinations of $F_1, \ldots, F_n, \top$ and $\bot$. Formula $[L \le S]$ is already a formula of such kind. It remains to notice that, since $sum\langle S\rangle \ge L$ is monotone, (7.1) is strongly equivalent to (7.10) by Lemma 16.

For part (b), the reasoning is similar, except that we want $[S \leq U]$ and (7.1) (corresponding to aggregate $sum\langle S\rangle \leq U$) to be written as AND-OR combinations of $not\ F_1, \ldots, not\ F_n$, $\top$ and $\bot$. Formula $[S \leq U]$, which has the form (written as a propositional formula)

$$\neg\Big(\bigvee_{I\subseteq\{1,\ldots,n\}:\cdots}\big(\bigwedge_{i\in I} F_i\big)\Big)$$

is strongly equivalent to

$$\bigwedge_{I\subseteq\{1,\ldots,n\}:\cdots}\big(\bigvee_{i\in I}\neg F_i\big).$$

It remains to show that, since $sum\langle S\rangle \leq U$ is an antimonotone aggregate, (7.1) is strongly equivalent to (7.11), which can be strongly equivalently written as

$$\bigwedge_{I\subseteq\{1,\ldots,n\}\ :\ op(\{w_i\ :\ i\in I\})\not\prec N}\big(\bigvee_{i\in I}\neg F_i\big).$$

$\square$

7.7.3 Proof of Theorem 6

We observe, first of all, that the definition of satisfaction of FLP-aggregates and FLP-programs in [Faber *et al.*, 2004] is equivalent to ours. The definition of a reduct is different, however. We denote the reduct of a program Π with FLP-aggregates relative to X in the sense of [Faber *et al.*, 2004] as $\Pi^{\underline{X}}$.

Next lemma is easily provable by structural induction.

Lemma 35. *For any nested expression F without negations and any two sets X and Y of atoms such that $Y \subseteq X$, $Y \models F^X$ iff $Y \models F$.*

Lemma 36. *For any FLP-aggregate A and any set X of atoms, if $X \models A$ then*

$$Y \models A^X \text{ iff } Y \models A.$$

Proof. Let A have the form (7.1). Since $X \models A$, A^X has the form

$$op\langle\{F_1^X = w_1, \ldots, F_n^X = w_n\}\rangle \prec N.$$

In case of FLP-aggregates, each F_i is a conjunction of atoms. Then, by Lemma 35, $Y \models F_i^X$ iff $Y \models F_i$. The claim immediately follows from the definition of satisfaction of aggregates. $\square$

Theorem 6. *The answer sets for a positive FLP-program under our semantics are identical to its answer sets in the sense of [Faber* et al.*, 2004].*

Proof. It is easy to see that if $X \not\models \Pi$ then $X \not\models \Pi^X$ and $X \not\models \Pi^{\underline{X}}$, so that X is not an answer set under either semantics. Now assume that $X \models \Pi$. We will show that the two reducts are satisfied by the same subsets of X. It is sufficient to consider the case in which Π contains only one rule

$$A_1 \wedge \cdots \wedge A_m \rightarrow a_1 \vee \cdots \vee a_n. \tag{7.22}$$

If $X \not\models A_1 \wedge \cdots \wedge A_m$ then $\Pi^{\underline{X}} = \emptyset$, and Π^X is the tautology

$$\bot \rightarrow (a_1 \vee \cdots \vee a_n)^X.$$

Otherwise, $\Pi^{\underline{X}}$ is rule (7.22), and Π^X is

$$A_1^X \wedge \cdots \wedge A_m^X \rightarrow (a_1 \vee \cdots \vee a_n)^X.$$

These two reducts are satisfied by the same subsets of X by Lemmas 35 and 36. $\square$

7.7.4 Proof of Proposition 20

This proof refers to the semantics of PDB-aggregates reviewed in Section 3.3. Given a PDB-aggregate of the form (7.1) and a coherent set X of literals, by I_X we denote the set $\{i \in \{1, \ldots, n\} \ : \ X \models F_i\}$.

Lemma 37. *For each PDB-aggregate of the form (7.1), a set X of atoms satisfies a formula of the form $G_{(I_1,I_2)}$ iff $I_1 \subseteq I_X \subseteq I_2$.*

Proof.

$$
\begin{aligned}
X \models G_{(I_1,I_2)} \quad &\text{iff} \quad X \models F_i \text{ for all } i \in I_1, \text{ and } X \not\models F_i \text{ for all } i \in \{1,\dots,n\} \setminus I_2 \\
&\text{iff} \quad X \models F_i \text{ for all } i \in I_1, \text{ and for every } i \text{ such that } X \models F_i, i \in I_2 \\
&\text{iff} \quad I_1 \subseteq I_X \text{ and } I_X \subseteq I_2.
\end{aligned}
$$

□

Lemma 38. *For every PDB-aggregate A, A_{tr} is classically equivalent to (7.4).*

Proof. Consider a coherent set X of atoms. By Lemma 37,

$$
\begin{aligned}
X \models A_{tr} \quad &\text{iff} \quad X \text{ satisfies one of the disjunctive terms } G_{(I_1,I_2)} \text{ of } A_{tr} \\
&\text{iff} \quad \text{for a disjunctive term } G_{(I_1,I_2)} \text{ of } A_{tr}, I_1 \subseteq I_X \subseteq I_2.
\end{aligned}
$$

It is easy to verify that A_{tr} contains a disjunctive term $G_{(I_1,I_2)}$ with $I_1 \subseteq I_X \subseteq I_2$ iff A_{tr} contains disjunctive term $G_{(I_X,I_X)}$. Consequently,

$$
\begin{aligned}
X \models A_{tr} \quad &\text{iff} \quad A_{tr} \text{ contains disjunctive term } G_{(I_X,I_X)} \\
&\text{iff} \quad op(W_{I_X}) \prec N.
\end{aligned}
$$

We have essentially found that $X \models A_{tr}$ iff $X \models A$. The claim now follows by Proposition 15(a). □

Lemma 39. *For any PDB-aggregate A, A_{tr} is strongly equivalent to*

(a)

$$\bigvee_{I \in \{1,\dots,n\} : op(W_I) \prec N} G_{(I,\{1,\dots,n\})}$$

if A is monotone, and to

(b)

$$\bigvee_{I \in \{1,\dots,n\} : op(W_I) \prec N} G_{(\emptyset,I)}$$

if it is antimonotone.

Proof. To prove (a), assume that A is monotone. Then, if A_{tr} contains a disjunctive term $G_{(I_1,I_2)}$ then it contains the disjunctive term $G_{(I_1,\{1,\dots,n\})}$ as well. Consider also that formula $G_{(I_1,\{1,\dots,n\})}$ entails $G_{(I_1,I_2)}$. Then we can drop all disjunctive terms of the form $G_{(I_1,I_2)}$ with $I_2 \neq \{1,\dots,n\}$. Formula A_{tr} becomes

$$\bigvee_{I_1\subseteq\{1,\dots,n\}:\text{ for all } I \text{ such that } I_1 \subseteq I \subseteq \{1,\dots,n\},\ op(W_I) \prec N} G_{(I_1,\{1,\dots,n\})}.$$

It remains to notice that, since A is monotone, if $op(W_{I_1}) \prec N$ then $op(W_I) \prec N$ for all I superset of I_1.

The proof for (b) is similar. □

Proposition 20 *For any monotone or antimonotone PDB-aggregates A of the form (7.1) where $F_1,\dots,F_n$ are atoms, A_{tr} is strongly equivalent to (7.4).*

Proof. Let S be $\{F_1 = w_1,\dots,F_n = w_n\}$. Lemma 38 says that A_{tr} is classically equivalent to (7.4) for every formulas $F_1,\dots,F_n$ in S. We can prove the claim using Lemma 33, by showing that both A_{tr} and (7.4) can be strongly equivalently rewritten as AND-OR combinations of

- $F_1,\dots,F_n,\top,\bot$, if A is monotone, and
- *not* $F_1,\dots,$ *not* $F_n,\top,\bot$, if A is antimonotone.

About A_{tr}, this follows by Lemma 39. Indeed, each $G_{(I,\{1,\dots,n\})}$ is a (possibly empty) conjunction of terms of the form F_i, and each $G_{(\emptyset,I)}$ is a (possibly empty) conjunction of terms of the form F_i. About (7.4), this has already been shown in the proof of Theorem 5. □

Chapter 8

Modular Translations and Strong Equivalence

In this chapter we consider logic programs/propositional theories without classical/strong negation. Under this hypothesis, Figure 8.1 summarizes the syntax of rules in traditional programs and "propositional extensions" of this class, as well as some of its subclasses. The language in each line of the table contains the languages shown in the previous lines.

When we compare the expressiveness of two classes of rules R and R', several criteria can be used. First, we can ask whether for any R-program (that is, a set of rules of the type R) one can find an R'-program that has exactly the same answer sets. (That means, in particular, that the R'-program does not use "auxiliary atoms" not occurring in the given R-program.) From this point of view, the classes of rules shown in Figure 8.1 can be divided into three groups: a UR- or PR-program has a unique answer set; TR-, TRC- and DR-programs may have many answer sets, but its answer sets always have the "anti-chain" property (one cannot be a proper subset of another); a NDR-, RNE or PF-program can have an arbitrary collection of sets of atoms as its collection of answer sets.

class of rules	syntactic form
UR	unary rules: $a \leftarrow$ (also written as simply a) and $a \leftarrow b$
PR	positive rules: $a \leftarrow b_1, \dots, b_n$
TR	traditional rules: $a \leftarrow b_1, \dots, b_n, not\ c_1, \dots, not\ c_m$
TRC	TRs + constraints: TRs and $\leftarrow b_1, \dots, b_n, not\ c_1, \dots, not\ c_m$
DR	disjunctive rules: $a_1; \dots; a_p \leftarrow b_1, \dots, b_n, not\ c_1, \dots, not\ c_m$
NDR	negational disjunctive rules: $a_1, \dots; a_p; not\ d_1; \dots; not\ d_q \leftarrow b_1, \dots, b_n, not\ c_1, \dots, not\ c_m$
RNE	rules with nested expressions: $F \leftarrow G$
PF	propositional formulas: H

Figure 8.1: A classification of logic programs under the answer set semantics. Here a, b, c, d stand for propositional atoms. F, G stand for nested expressions without classical negation (that is, expressions formed from atoms, $\top$ and $\bot$, using conjunction (,), disjunction (;) and negation as failure (*not*), see Section 2.2.1), and H stands for an arbitrary propositional formula.

Another comparison criterion is based on the computational complexity of the problem of the existence of an answer set. We pass, in the complexity hierarchy, from P in case of UR- and PR-programs, to NP in case of TR- and TRC-programs [Marek and Truszczyński, 1991], and finally to Σ_2^P for more complex kinds of programs [Eiter and Gottlob, 1993].

A third criterion consists in checking whether every rule in R is strongly equivalent [Lifschitz *et al.*, 2001] (see Section 3.7) to an R'-program. From this point of view, PR is essentially more expressive than UR: we will see at the end of Section 8.2 that $a \leftarrow b, c$ is not strongly equivalent to any set of unary rules. Furthermore, TRC and DR are essentially different from each other, since no program in TRC is strongly equivalent to the rule $p; q$ in DR [Turner, 2003, Proposition 1].

A fourth comparison criterion is based on the existence of a translation from R-programs to R'-programs that is not only sound (that is, preserves the program's answer sets) but is also modular: it can be applied to a program rule-by-rule. For instance, Janhunen (2000) showed that there is no modular translation from PR to UR, and no modular translation from TR to PR.[1] On the other hand, RNE can be translated into NDR by a modular procedure similar to converting formulas to conjunctive normal form [Lifschitz *et al.*, 1999].

The main theorem of this chapter shows that under some general conditions, the last two criteria — the one based on strong equivalence and the existence of a sound modular translation — are equivalent to each other. This offers a method to prove that there is no modular translation from R to R' by finding a rule in R that is not strongly equivalent to any R'-program. For instance, in view of the Proposition 1 from [Turner, 2003] mentioned above, no modular translation exists from DR to TRC.

To apply the main theorem to other cases, we need to learn more about the

[1] His results are actually stronger, see Section 8.1 below.

strong equivalence relations between a single rule of a language and a set of rules. We show that for many rules r in NDR, any NDR-program that is strongly equivalent to r contains a rule that is at least as "complex" as r. This fact will allow us to conclude that all classes UR, PR, TR, TRC, DR and NDR are essentially different from each other in terms of strong equivalence. In view of the main theorem, it follows that they are essentially different from each other in the sense of the modular translation criterion as well.

Finally, we show how to apply our main theorem to programs with weight constraints [Simons *et al.*, 2002] (Section 2.2.2). As a result, we find that it is not possible to translate programs with weight constraints into programs with monotone cardinality atoms [Marek *et al.*, 2004] in a modular way (unless the translation introduces auxiliary atoms).

The chapter continues with the statement of our main theorem (Section 8.1). In Section 8.2, we study the expressiveness of subclasses of NDR in terms of strong equivalence and modular translations. We move to the study of cardinality constraints in Section 8.3. The proofs of all claims of this chapter are in Section 8.4.

8.1 Modular Transformations and Strong Equivalence

A *(modular) transformation* is a function f such that

- $\text{Dom}(f) \subseteq \text{PT}$, and
- for every formula $r \in \text{Dom}(f)$, $f(r)$ is a theory such that every atom occurring in it occurs in r also.

A transformation f is *sound* if, for every theory $\Pi \subseteq \text{Dom}(f)$, Π and $\bigcup_{r\in\Pi} f(r)$ have the same answer sets.

For example, the transformation defined in the proof of Proposition 7 from [Lifschitz *et al.*, 1999], which eliminates nesting from a program with nested expressions,

is a sound transformation. For instance, for this transformation f,

$$f(a \leftarrow b; c) = \{a \leftarrow b,\ a \leftarrow c\}.$$

As another example of a sound transformation, consider the transformation f with $\text{Dom}(f) = \text{NDR}$, where

$$f(not\ d_1; \dots; not\ d_q \leftarrow b_1, \dots, b_n, not\ c_1, \dots, not\ c_m) =$$
$$\{\leftarrow b_1, \dots, b_n, d_1, \dots, d_q, not\ c_1, \dots, not\ c_m\}$$

and $f(r) = \{r\}$ for the rules r in NDR that contains at least one nonnegated atom in the head. On the other hand, the familiar method of eliminating constraints from a program that turns $\leftarrow p$ into $q \leftarrow p, not\ q$ is not a transformation in the sense of our definition, because it introduces an atom q that doesn't occur in $\leftarrow p$.

Other definitions of a modular transformation allow the the introduction of auxiliary atoms. This is, for instance, the case of the definitions in [Janhunen, 2000], and of the translation $[\Omega]^{nn}$ of a program with nested expressions into a traditional program (see Section 5.1). These two works are also different from the work described in this chapter in that they take into account the computation time of translation algorithms.

This is the theorem that relates strong equivalence and modular transformations:

Theorem 7. *For every transformation f such that* $\text{Dom}(f)$ *contains all unary rules, f is sound iff, for each $r \in \text{Dom}(f)$, $f(r)$ is strongly equivalent to r.*

Our definition of transformation requires that all atoms that occur in $f(r)$ occur in r also. The following counterexample shows that without this assumption the assertion of Theorem 7 would be incorrect. Let p and q be two atoms, and, for

each rule $r = F \leftarrow G$ in RNE, let $f_1(r)$ be

$$F \leftarrow G, not\ p$$
$$F \leftarrow G, not\ q$$
$$F_{p\leftrightarrow q} \leftarrow G_{p\leftrightarrow q}, not\ not\ p, not\ not\ q.$$

where $F_{p\leftrightarrow q}$ and $G_{p\leftrightarrow q}$ stand for F and G with all the occurrences of p replaced by q and vice versa. Note that f_1 is not a transformation as defined in this chapter since q occurs in $f_1(p \leftarrow \top)$. It can also be shown that $p \leftarrow \top$ and $f_1(p \leftarrow \top)$ are not strongly equivalent. However, the "transformation" is sound:

Proposition 21. *For any program* Π*,* Π *and* $\bigcup_{r\in\Pi} f_1(r)$ *have the same answer sets.*

Without the assumption that UR $\subseteq$ Dom(f), Theorem 7 would not be correct either. We define a transformation f_2 such that Dom(f_2) consists of all rules (of DR) of the form

$$a_1; \ldots; a_p \leftarrow not\ c_1, \ldots, not\ c_m. \tag{8.1}$$

with $p > 0$ and where $a_1, \ldots, a_p$ are all distinct. We denote the set of rules of this form by NBR. For each rule r of the form (8.1), $f_2(r)$ is defined as

$$\{a_i \leftarrow not\ a_1, \ldots, not\ a_{i-1}, not\ a_{i+1}, \ldots, not\ a_p, not\ c_1, \ldots, not\ c_m \ :\ 1 \leq i \leq p\}.$$

For instance, $f_2(p; q) = \{p \leftarrow not\ q\ ,\ q \leftarrow not\ p\}$. It is easy to see that this program is not strongly equivalent to $p; q$. However, this transformation is sound:

Proposition 22. *For any program* $\Pi \subseteq$ *NBR,* Π *and* $\bigcup_{r\in\Pi} f_2(r)$ *have the same answer sets.*

8.2 Applications: Negational Disjunctive Rules

In order to apply Theorem 7 to modular translations, we first need to study some properties of strong equivalence. We focus on the class NDR.

If r is

$$a_1, \ldots ; a_p ; not\ d_1 ; \ldots ; not\ d_q \leftarrow b_1, \ldots, b_n, not\ c_1, \ldots, not\ c_m$$

define

$$head^+(r) = \{a_1, \ldots, a_p\} \qquad head^-(r) = \{d_1, \ldots, d_q\}$$
$$body^+(r) = \{b_1, \ldots, b_n\} \qquad body^-(r) = \{c_1, \ldots, c_m\}.$$

We say that r is *basic* if any two of these sets, except possibly for the pair $head^+(r)$, $head^-(r)$, are disjoint.

Any nonbasic rule can be easily simplified: it is either strongly equivalent to the empty program or contains redundant terms in the head. A basic rule, on the other hand, cannot be simplified if it contains at least one nonnegated atom in the head:

Theorem 8. *Let r be a basic rule in NDR such that $head^+(r) \neq \emptyset$. Every program subset of NDR that is strongly equivalent to r contains a rule r' such that*

$$head^+(r) \subseteq head^+(r') \qquad body^+(r) \subseteq body^+(r')$$
$$head^-(r) \subseteq head^-(r') \qquad body^-(r) \subseteq body^-(r')$$

This theorem shows us that for most basic rules r (the ones with at least one positive element in the head), every program strongly equivalent to r must contain a rule that is at least as "complex" as r.

Given two subsets R and R' of RNE, a *(modular) translation* from R to R' is a transformation f such that $\mathrm{Dom}(f) = R$ and $f(r)$ is a subset of R' for each $r \in \mathrm{Dom}(f)$. Using Theorems 7 and 8, we can differentiate between the classes of rules in Figure 8.1 in terms of modular translations:

Proposition 23. *For any two languages R and R' among UR, PR, TRC, TR, DR and NDR such that $R' \subset R$, there is no sound translation from R to R'.*

Theorems 7 and 8 allow us also to differentiate between subclasses of NDR described in terms of the sizes of various parts of the rule. Define, for instance PBR_i ("positive body of size i") as the set of rules r of NDR such that $|body^+(r)| \leq i$. We can show that, for every $i \geq 0$, there is no sound translation from PBR_{i+1} to PBR_i (or even from $\mathrm{PBR}_{i+1} \cap \mathrm{PR}$ to PBR_i). Similar properties can be stated in terms of the sizes of $body^-(r)$, $head^+(r)$ and $head^-(r)$.

Another consequence of Theorem 8 is in terms of (absolute) tightness of a program [Erdem and Lifschitz, 2003; Lee and Lifschitz, 2003]. Tightness is an important property of logic programs: if a program is tight then its answer sets can be equivalently characterized by the satisfaction of a set of propositional formulas of about the same size. Modular translations usually don't make nontight programs tight:

Proposition 24. *Let R be any subset of NDR that contains all unary rules, and let f be any sound translation from R to NDR. For every nontight program $\Pi \subseteq R$ consisting of basic rules only, $\bigcup_{r\in\Pi} f(r)$ is nontight.*

Note that Theorem 8 doesn't relate classes of rule/formulas more general than NDR. In terms of strong equivalence and modular translations, they are no formulas more general than NDR. Indeed, every propositional formula can be strongly equivalently written as a set of NDR rules [Cabalar and Ferraris, 2007].

Criteria for strong equivalence, in part related to Theorem 8, are proposed in [Lin and Chen, 2005].

8.3 Applications: Cardinality Constraints

Let CCR denote the set of all rules with cardinality constraints (Section 2.2.2). A straightforward generalization of the definition of a transformation allows us to talk about sound translations between subclasses of CCR, and also between a sub-

class of CCR and a subclass of PT. The concept of (modular) transformations and translations can be extended to programs with cardinality constraints: we can have translations between subclasses of CCR, and from/to subclasses of PT. The definition of the soundness for those translations is straightforward as well. For instance, the two translations $\Omega \mapsto [\Omega]$ and $\Omega \mapsto [\Omega]^{nd}$ from CCR to RNE defined in Chapter 4 are modular and sound.

Another class of programs similar to programs with cardinality constraints — programs with monotone cardinality atoms — has been defined in [Marek *et al.*, 2004]. The results of that paper show that rules with monotone cardinality atoms are essentially identical to rules with cardinality constraints that don't contain negation as failure; we will denote the set of all such rules by PCCR ("positive cardinality constraints").

First of all, we show how programs with cardinality constraints are related to the class NDR. Let SNDR (Simple NDR) be the language consisting of rules of the form

$$a; not\ d_1; \ldots; not\ d_q \leftarrow b_1, \ldots, b_n, not\ c_1, \ldots, not\ c_m \tag{8.2}$$

and

$$\leftarrow b_1, \ldots, b_n, not\ c_1, \ldots, not\ c_m. \tag{8.3}$$

Proposition 25. *There exist sound translations from SNDR to CCR, and back.*

If we don't allow negation in cardinality constraints, another relationship holds. We define the class VSNDR (Very Simple NDR) consisting of rules of the form

$$a; not\ a \leftarrow b_1, \ldots, b_n, not\ c_1, \ldots, not\ c_m \tag{8.4}$$

and of the form (8.3).

Proposition 26. *There exist sound translations from VSNDR to PCCR, and back.*

Using Theorems 7 and 8, we will prove:

Proposition 27. *There is no sound translation from CCR to PCCR.*

Since the class PCCR is essentially identical to the class of rules with monotone cardinality atoms, we conclude that programs with cardinality constraints are essentially more expressive than programs with monotone cardinality atoms.

8.4 Proofs

8.4.1 Properties of Strong Equivalence

In this section — unless otherwise specified — we refer to the definition of a reduct for propositional theories of Section 6.1. The main criterion that we use to check strong equivalence is one slightly modified version of a characterization from Proposition 8.

Lemma 40. *Let A be the set of atoms occurring in theories Γ_1 and Γ_2. Γ_1 and Γ_2 are strongly equivalent iff, for each $Y \subseteq A$, Γ_1^Y and Γ_2^Y.*

Recall also that Γ_1^Y contains atoms from Y only, and similarly for Γ_2^Y.

We will also use the following two lemmas. The first one is immediate from the characterization of strong equivalence based on equivalence in the logic of here-and-there from [Lifschitz *et al.*, 2001]. The second comes from Lemma 4 from [Lifschitz *et al.*, 2001].

Lemma 41. *Let P_1 and P_2 be sets of theories. If each theory in P_1 is strongly equivalent to a theory in P_2 and vice versa, then $\bigcup_{\Gamma\in P_1}\Gamma$ and $\bigcup_{\Gamma\in P_2}\Gamma$ are strongly equivalent.*

Lemma 42. *Two theories Γ_1 and Γ_2 are strongly equivalent iff, for every program $\Pi \subseteq UR$, $\Gamma_1 \cup \Pi$ and $\Gamma_2 \cup \Pi$ have the same answer sets.*

8.4.2 Proof of Theorem 7

Theorem 7. *For every transformation f such that* $\mathrm{Dom}(f)$ *contains all unary rules, f is sound iff, for each $r \in \mathrm{Dom}(f)$, $f(r)$ is strongly equivalent to r.*

The proof from right to left is a direct consequence of Lemma 41: if $f(r)$ is strongly equivalent to r for every formula $r \in R$, then for any $\Gamma \subseteq \mathrm{Dom}(f)$, Π and $\bigcup_{r\in\Gamma} f(r)$ are strongly equivalent, and consequently have the same answer sets.

In the proof from left to right, we first consider the case when r is a unary rule, and then extend the conclusion to arbitrary formulas. In the rest of this section, f is an arbitrary sound transformation such that $\mathrm{Dom}(f)$ contains all unary rules. By a and b we denote distinct atoms.

Lemma 43. *For every fact a, $\{a\}$ and $f(a)$ are strongly equivalent.*

Proof. In view of Lemma 40, we need to show that (i) $f(a)^{\{a\}}$ is equivalent to $\{a\}$, and (ii) $f(a)^{\emptyset}$ is equivalent to $\bot$. Since $\{a\}$ is an answer set for $\{a\}$, it is an answer set for $f(a)$, so that (i) follows from Lemma 25. Since $\emptyset$ is not an answer set for $f(a)$, it follows we get that $\emptyset \not\models f(a)^{\emptyset}$. Since $f(a)^{\emptyset}$ doesn't contain atoms, (ii) must hold. □

Lemma 44. *For every formula r and fact a, $\{r, a\}$ and $f(r) \cup \{a\}$ have the same answer sets.*

Proof. In view of Lemma 43, $f(r) \cup \{a\}$ and $f(r) \cup f(a)$ have the same answer sets, and the same holds for $\{r, a\}$ and $f(r) \cup f(a)$ by hypothesis. □

Lemma 45. *For every rule r of the form $a \leftarrow a$,*

(i) $f(r)^{\emptyset}$ is classically equivalent to $\{r\}^{\emptyset}$, and

(ii) $f(r)^{\{a\}}$ is classically equivalent to $\{r\}^{\{a\}}$.

Proof. First of all, since the empty set in the only answer set for $\{r\}$ and then for $f(r)$, (i) is clearly true by Lemma 25. For (ii), we need to show that $f(r)^{\{a\}}$ is satisfied by both $\emptyset$ and $\{a\}$. Consider the theory consisting of rule r plus fact a. Since $\{a\}$ is an answer set for $\{r, a\}$, it is an answer set for $f(r) \cup \{a\}$ also by Lemma 44. Consequently, by Lemma 25, $f(r)^{\{}a\} \cup \{a\}$ is classically equivalent to $\{a\}$ from which we derive that $\{a\} \models f(r)^{\{}a\}$. On the other hand, since $\{a\}$ is not an answer set for $f(r)$, so also $\emptyset \models f(r)^{\{}a\}$. □

Lemma 46. *For every rule r of the form $a \leftarrow b$,*

(i) $f(r)^{\emptyset}$ is classically equivalent to $\{r\}^{\emptyset}$,

(ii) $f(r)^{\{a\}}$ is classically equivalent to $\{r\}^{\{a\}}$, and

(iii) $f(r)^{\{b\}}$ is classically equivalent to $\{r\}^{\{b\}}$.

Proof. The proof of the first two claims is similar to the one of Lemma 45. To prove (iii), it is sufficient to show that $\{b\} \not\models f(r)$. Since $\{b\}$ is not an answer set for $\{r, b\}$, it is not an answer set for $f(r) \cup \{b\}$ either by Lemma 44. But $\emptyset \not\models (f(r) \cup \{b\})^{\{b\}}$ because $\emptyset \not\models \{b\}^{\{b\}}$; consequently $\{b\} \not\models (f(r) \cup \{b\})^{\{b\}}$. Since $\{b\} \models \{b\}^{\{b\}}$, it follows that $\{b\} \not\models f(r)^{\{b\}}$, and then that $\{b\} \not\models f(r)$ by Lemma 24. □

Lemma 47. *For every rule r of the form $a \leftarrow b$, $f(r)^{\{a,b\}}$ is classically equivalent to $\{r\}^{\{a,b\}}$.*

Proof. We need to show that the only subset of $\{a, b\}$ not satisfying $f(r)^{\{a,b\}}$ is $\{b\}$, that is

(i) $\{b\} \not\models f(r)^{\{a,b\}}$,

(ii) $\{a, b\} \models f(r)^{\{a,b\}}$,

(iii) $\{a\} \models f(r)^{\{a,b\}}$, and

(iv) $\emptyset \models f(r)^{\{a,b\}}$.

First of all, set $\{a, b\}$ is an answer set for $\{r, b\}$, and consequently for $f(r) \cup \{b\}$ also by Lemma 44. Consequently, by Lemma 25,

$$\{\text{a,b}\} \text{ is classically equivalent to } (f(r) \cup \{b\})^{\{a,b\}}. \tag{8.5}$$

From it we derive that $\{b\} \not\models (f(r) \cup \{b\})^{\{a,b\}}$, and consequently (i). From (8.5) we also derive (ii) and then that $\{a, b\} \models (f(r) \cup \{a\})^{\{a,b\}}$. Notice that $\{a, b\}$ is not an answer set for $f(r) \cup \{a\}$ because it is not an answer set for $\{r, a\}$ and by Lemma 44. Consequently there is a proper subset of $\{a, b\}$ that satisfies $(f(r) \cup \{a\})^{\{a,b\}}$. Such subset can only be $\{a\}$ because it is the only one that satisfies $\{a\}^{\{a,b\}}$. Claim (iii) immediately follows.

The proof of (iv) remains. Let r' be $b \leftarrow a$. Claims (i) and (ii) determine which subsets of $\{a, b\}$ satisfy $(f(r) \cup f(r'))^{\{a,b\}}$: from part (i) applied to both r and r' we get that $\{a, b\}$ satisfies this theory, while $\{b\}$ (by part (ii) applied to r) and $\{a\}$ (by part (ii) applied to r') don't. On the other hand $\{a, b\}$ is not an answer set for $\{r, r'\}$ and then for $f(r) \cup f(r')$ by the soundness hypothesis. We can conclude that $\emptyset \models (f(r) \cup f(r'))^{\{a,b\}}$ from which (iv) follows. □

Lemma 48. *For every unary program* Π*,* Π *and* $\bigcup_{r \in \Pi} f(r)$ *are strongly equivalent.*

Proof. In view of Lemma 41, it is sufficient to show that for each unary rule r, $\{r\}$ and $f(r)$ are strongly equivalent. For rules that are facts, this is proven Lemma 43. For rules r of the form $a \leftarrow a$, it follows by Lemma 45 and Lemma 40. Similarly, for rules of the form $a \leftarrow b$, it follows by Lemmas 46, 47 and Lemma 40. □

Now we are ready to prove the second part of the main theorem: for any formula $r \in \text{Dom}(f)$, $f(r)$ and $\{r\}$ are strongly equivalent. By Lemma 42, it is sufficient to show that, for each unary program Π, $\Pi \cup \{r\}$ and $\Pi \cup f(r)$ have the same answer sets. First we notice that $\Pi \cup \{r\}$ and $\bigcup_{r' \in \Pi \cup \{r\}} f(r')$ have the same

answer sets since $\Pi \cup \{r\} \subseteq \mathrm{Dom}(f)$ and for the soundness of the transformation. Then we can see that $\bigcup_{r' \in \Pi \cup \{r\}} f(r') = \bigcup_{r' \in \Pi} f(r') \cup f(r)$. Finally, $\bigcup_{r' \in \Pi} f(r') \cup f(r)$ and $\Pi \cup f(r)$ have the same answer sets because, by Lemma 48, programs Π and $\bigcup_{r' \in \Pi} f(r')$ are strongly equivalent.

8.4.3 Proofs of Propositions 21 and 22

Proposition 21. *For any program Π, Π and $\bigcup_{r \in \Pi} f_1(r)$ have the same answer sets.*

Proof. Consider any set of atoms X. If $\{p, q\} \not\subseteq X$ then $f(F \leftarrow G)^X$ is essentially $\{F \leftarrow G\}^X$, and the claim easily follows. Otherwise $f_1(F \leftarrow G)^X$ is essentially $\{F_{p \leftrightarrow q} \leftarrow G_{p \leftrightarrow q}\}$. If we extend the notation of the subscript $p \leftrightarrow q$ to both programs and sets of atoms, $(\bigcup_{r \in \Pi} f_1(r))^X$ can be seen as $(\Pi^X)_{p \leftrightarrow q}$. A subset Y of X satisfies Π^X iff $Y_{p \leftrightarrow q}$ satisfies $(\Pi^X)_{p \leftrightarrow q}$. Since $Y_{p \leftrightarrow q} \subseteq X$ and $|Y_{p \leftrightarrow q}| = |Y|$, we can conclude that X is a minimal set satisfying $(\bigcup_{r \in \Pi} f_1(r))^X$ iff it is a minimal set satisfying Π^X. □

Proposition 22. *For any program $\Pi \subseteq NBR$, Π and $\bigcup_{r \in \Pi} f_2(r)$ have the same answer sets.*

Proof. Let Π' be $\bigcup_{r \in \Pi} f_2(r)$. First of all, it is easy to verify that Π and Π' are satisfied by the same sets of atoms, so that if $X \not\models \Pi$ then X is not an answer set for neither programs. Assume now that $X \models \Pi$. Then, $X \models \Pi'$. It is then not hard to check that $(Pi')^X$ is equivalent to a set of atoms Y, consisting of atoms a of X such that there is a rule r in Π with $head^+(r) \cap X = \{a\}$ and $body^-(r) \cap X = \emptyset$. So X is an answer set for Π' iff $Y = X$. It remains to show that X is an answer set for Π' iff $Y = X$. It is easy to verify that Π^X is Y plus a set of disjunctions of two or more atoms of X. If $Y = X$ then clearly Π^X is equivalent to Y. If not, let a be an atom of X but not of X, and let Z be $X \setminus \{a\}$. It is not hard to verify that

$Z \models \Pi^X$, since all elements of Y belong to Z, and, for each disjunction of two or more atoms of X in Π^X, at least one of them belongs to Z. From this we conclude that X is an answer set for Π iff $Y = X$. □

8.4.4 Proof of Theorem 8

For simplicity, we consider the definition of an answer set for NDR programs as defined in [Lifschitz and Woo, 1992], in which the reduct Π^X consists of the rule

$$\mathit{head}^+(r) \leftarrow \mathit{body}^+(r) \tag{8.6}$$

($\mathit{head}^+(r)$ here stands for the disjunction of its elements, $\mathit{body}^+(r)$ for their conjunction) for every rule r in Π such that $\mathit{head}^-(r) \subseteq X$ and $\mathit{body}^-(r) \cap X = \emptyset$. It is easy to check that this definition of a reduct is equivalent to the definition of a reduct for programs with nested expressions (they are satisfied by the same sets of atoms, see Section 3.1). Consequently, the characterization of strong equivalence based on the reduct of [Turner, 2003] reviewed in Section 3.7 also is correct with this definition of a reduct:

Lemma 49. *Two programs Π_1 and Π_2 subsets of NDR are strongly equivalent iff, for every consistent set Y of literals,*

- *$Y \models \Pi_1^Y$ iff $Y \models \Pi_2^Y$, and*
- *if $Y \models \Pi_1^Y$ then, for each $X \subset Y$, $X \models \Pi_1^Y$ iff $X \models \Pi_2^Y$.*

Theorem 8. *Let r be a basic rule in NDR such that $\mathit{head}^+(r) \neq \emptyset$. Every program subset of NDR that is strongly equivalent to r contains a rule r' such that*

$$\mathit{head}^+(r) \subseteq \mathit{head}^+(r') \qquad \mathit{body}^+(r) \subseteq \mathit{body}^+(r')$$
$$\mathit{head}^-(r) \subseteq \mathit{head}^-(r') \qquad \mathit{body}^-(r) \subseteq \mathit{body}^-(r')$$

Proof. Let Π be a program strongly equivalent to r. Let X be $head^+(r) \cup head^-(r) \cup body^+(r)$, and let Y be $body^+(r)$. Then r^X is (8.6). By Lemma 49, since $X \models r^X$ (recall that $head^+(r)$ is nonempty by hypothesis) then $X \models \Pi^X$, and since $Y \not\models r^X$ it follows that $Y \not\models \Pi^X$. Consequently, there is a rule of Π — the rule r' of the theorem's statement — such that $X \models (r')^X$ and $Y \not\models (r')^X$. From this second fact, $(r')^X$ is nonempty so it is

$$head^+(r') \leftarrow body^+(r'), \tag{8.7}$$

and also $Y \models body^+(r')$ and $Y \not\models head^+(r')$.

To prove that $head^+(r) \subseteq head^+(r')$, take any atom $a \in head^+(r)$. The set $Y \cup \{a\}$ satisfies r^X, so it satisfies Π^X by Lemma 49, and then $(r')^X$ also. On the other hand, since $Y \models body^+(r')$ and $Y \subseteq Y \cup \{a\}$, we have that $Y \cup \{a\} \models body^+(r')$. Consequently, $Y \cup \{a\} \models head^+(r')$. Since $Y \not\models head^+(r')$, we can conclude that a is an element of $head^+(r')$.

The proof that $body^+(r) \subseteq body^+(r')$ is similar to the previous part of the proof, by taking any $a \in body^+(r)$ and considering the set $Y \setminus \{a\}$ instead of $Y \cup \{a\}$.

To prove that $head^-(r) \subseteq head^-(r')$, take any atom $a \in head^-(r)$. Since $r^{X \setminus \{a\}}$ is empty, it is satisfied, in particular, by Y and $X \setminus \{a\}$. Consequently, $Y \models (r')^{X \setminus \{a\}}$ by Lemma 49. On the other hand, $Y \not\models (r')^X$, so $(r')^{X \setminus \{a\}}$ is not (8.7), and then it is empty. The only case in which $(r')^{X \setminus \{a\}}$ is empty and $(r')^X$ is not is if $a \in head^-(r')$.

The proof that $body^-(r) \subseteq body^-(r')$ is similar to the previous part of the proof, by taking any $a \in body^-(r)$ and considering the reduct $r^{X \cup \{a\}}$. □

8.4.5 Definition of a Tight Program

Here we provide the definition of a tight program for programs in NDR [Lee and Lifschitz, 2003].[2] This class of logic programs is important because there is a poly-

[2]The definition of tightness of that paper is slightly more general.

nomial time reduction from tight NDR-programs into a set of propositional formulas (called the "completion" of the program [Clark, 1978; Lee and Lifschitz, 2003]) such that the answer sets of the program are identical to the models of the completion. That is, the problems of the existence of an answer set for a tight program is in class NP, even if the program contains disjunction in the head.

For any program Π $\in$NDR, consider functions g from the atoms occurring in Π into the set of natural number. Program Π is *tight* if there is a function g such that, for every rule $r \in \Pi$, for each element $a \in \mathit{body}^+(r)$ and each $b \in \mathit{head}^+(r)$, $g(a) < g(b)$.

For instance,

$$p \leftarrow q$$
$$q \leftarrow r$$

is tight: take $g(p) = 0$, $g(q) = 1$ and $g(r) = 2$. Program

$$p \leftarrow q$$
$$q \leftarrow p$$

is clearly not tight.

8.4.6 Proofs of Propositions 23 and 24

Proposition 23. *For any two languages R and R' among UR, PR, TR, TRC, DR and NDR such that $R' \subset R$, there is no sound translation from R to R'.*

Proof. First of all, Theorem 8 shows us that

- no subset of UR is strongly equivalent to PR rule $a \leftarrow b, c$,
- no subset of PR is strongly equivalent to TR rule $a \leftarrow b, \mathit{not}\ c$,
- no subset of TRC is strongly equivalent to DR rule $a; b \leftarrow c$, and
- no subset of DR is strongly equivalent to NDR rule $a; \mathit{not}\ b \leftarrow c$.

By Theorem 7, we can conclude the claim between UR and PR, PR and TR, TRC and DR, and finally between DR and NDR.

The proof between TR and TRC remains. Consider the TRC rule r with an empty head and an empty body. By Theorem 7, it is sufficient to show that no subset of TR that contains atoms from r only is strongly equivalent to r. Notice that r doesn't contain atoms, and that the only subset of TR that doesn't contain atoms is the empty set; this set is not strongly equivalent to r. □

Proposition 24. *Let R be any subset of NDR that contains all unary rules, and let f be any sound translation from R to NDR. For every nontight program $\Pi \subseteq R$ consisting of basic rules only, $\bigcup_{r \in \Pi} f(r)$ is nontight.*

Proof. Consider any sound translation f and program Π_1 consisting of rules of Π with at least one positive element in the head. Consider also a program Π' consisting, for each rule $r \in \Pi_1$, of a rule $r' \in f(r)$ such that

$$head^+(r) \subseteq head^+(r') \qquad \text{and} \qquad body^+(r) \subseteq body^+(r'). \tag{8.8}$$

(Such rule r' exists by Theorem 8.) Clearly, $\Pi' \subseteq \bigcup_{r \in \Pi} f(r)$. In view of the definition of tightness, it is then sufficient to show that Π' is not tight. Assume, in sake of contradiction, that Π' is tight. Then, there is a function g such that, for every rule $r' \in \Pi'$, for each element $a \in body^+(r')$ and each $b \in head^+(r')$, $g(a) < g(b)$. In view of (8.8), it is easy to check that g makes Π_1 tight. As the rules of $\Pi \setminus \Pi_1$ don't characterize tightness of Π, we conclude that Π is tight, which contradicts our hypotheses. □

8.4.7 Proofs of Propositions 25–27

All those proofs rely on the sound translations $\Omega \mapsto [\Omega]$ and $\Omega \mapsto [\Omega]^{nd}$ of Section 4.2.

Proposition 25. *There exist sound translations from SNDR to CCR, and back.*

Proof. We define a translation f from SNDR to CCR as follows: if r has the form (8.2) then $f(r)$ is

$$1 \leq \{a\} \leftarrow 1 \leq \{b_1\}, \ldots, 1 \leq \{b_n\}, \{c_1\} \leq 0, \ldots, \{c_m\} \leq 0, \\ \{not\ d_1\} \leq 0, \ldots, \{not\ d_q\} \leq 0,$$

and, if r has the form (8.3), then $f(r)$ is

$$1 \leq \{\} \leftarrow 1 \leq \{b_1\}, \ldots, 1 \leq \{b_n\}, \{c_1\} \leq 0, \ldots, \{c_m\} \leq 0, 1 \leq \{d_1\}, \ldots, 1 \leq \{d_q\}.$$

It is easy to check that $[f(r)]$ is strongly equivalent to $\{r\}$.

For the opposite direction, we take a rule r in CCR and consider $[\{r\}]^{nd}$. We can eliminate nesting from each of those rules by converting each body to a "disjunctive normal form" using De Morgan's laws and the distributivity of conjunction over disjunction. Then we can break each rule into several rules of the form

$$a \leftarrow b_1, \ldots, b_n, not\ c_1, \ldots, not\ c_m, not\ not\ d_1, \ldots; not\ not\ d_q$$

and

$$\bot \leftarrow b_1, \ldots, b_n, not\ c_1, \ldots, not\ c_m, not\ not\ d_1, \ldots; not\ not\ d_q.$$

In rules of the first kind, we can move each term $not\ not\ d_i$ to the head as a disjunctive term $not\ d_i$, obtaining (8.2); in rules of the second kind, we can eliminate double negation, obtaining a formula of the form (8.3). □

Proposition 26. *There exist sound translations from VSNDR to NCCR, and back.*

Proof. We define a translation f from VSNDR to NCCR as follows: if r has the form (8.4) then $f(r)$ is

$$\{a\} \leftarrow 1\{b_1\}, \ldots, 1\{b_n\}, \{c_1\}0, \ldots, \{c_m\}0$$

and, if r has the form (8.3), then $f(r)$ is

$$1 \leq \{\} \leftarrow 1 \leq \{b_1\}, \ldots, 1 \leq \{b_n\}, \{c_1\} \leq 0, \ldots, \{c_m\} \leq 0, 1 \leq \{d_1\}, \ldots, 1 \leq \{d_q\}.$$

It is easy to check that $[f(r)]$ is strongly equivalent to $\{r\}$.

For the opposite direction, we take a rule r in NCCR and consider $[\{r\}]^{nd}$. If we rewrite the first of rules (4.10) as

$$l_j; not\ l_j \leftarrow [C_1], \ldots, [C_n],$$

the body of each rule $[\{r\}]^{nd}$ is the conjunction of terms of the form $[C]$, where C is a cardinality constraint without negation as failure. In this case, $[C]$ doesn't contain negations nested in another negation, so that the rewriting of the body in "disjunctive" normal form doesn't contain terms of the form *not not a*. Consequently, $[\{r\}]^{nd}$ can be written as terms of the form

$$l_j; not\ l_j \leftarrow b_1, \ldots, b_n, not\ c_1, \ldots, not\ c_m$$

and

$$\bot \leftarrow b_1, \ldots, b_n, not\ c_1, \ldots, not\ c_m.$$

The rules of the first kind have the form (8.4), the other rules have the form (8.3). □

Proposition 27. *There is no sound translation from CCR to PCCR.*

Proof. Assume, in sake of contradiction, that a sound translation from CCR to PCCR exists. Then, in view of Propositions 25 and 26, a sound translation from SNDR to VSNDR exists. This is impossible in view of Theorems 7 and 8. □

Chapter 9

Causal Theories as Logic Programs

9.1 Introduction to Causal Theories

Causal logic [McCain and Turner, 1997] is a formalism for knowledge representation, especially suited for representing effects of actions. Causal theories are syntactically simple but also very general: they consist of causal rules of the form

$$F \Leftarrow G \tag{9.1}$$

where F and G are propositional formulas. Intuitively, rule (9.1) says that there is a cause for F to be true if G is true. For instance, the causal rule

$$p_{t+1} \Leftarrow a_t \tag{9.2}$$

can be used to describe the effect of an action a on a Boolean fluent p: if a is executed at time t then there is a cause for p to hold at time $t+1$. Other important concepts in commonsense reasoning can be easily expressed by rules of this kind too. For instance, a rule of the form

$$F \Leftarrow F$$

("if F is true then there is a cause for this") expresses, intuitively, that F is true by default. In particular, the causal rule

$$p_t \Leftarrow p_t \tag{9.3}$$

says that Boolean fluent p is normally true. The frame problem [McCarthy and Hayes, 1969] is solved in causal logic using the rules

$$\begin{aligned} p_{t+1} &\Leftarrow p_t \wedge p_{t+1} \\ \neg p_{t+1} &\Leftarrow \neg p_t \wedge \neg p_{t+1}. \end{aligned} \tag{9.4}$$

These rules express inertia: if a fluent p is true (false) at time t then normally it remains true (false) at time $t+1$.

The equivalence of two fluents or actions can be expressed by equivalences in the head. For instance, to express that two actions constants a and a' are synonymous, we can use causal rule

$$a_t \leftrightarrow a'_t \Leftarrow \top.$$

Rules of this kind are used to semantically characterize the relationship between modules in the Modular Action Description language MAD [Lifschitz and Ren, 2006]. For instance, [Erdoğan and Lifschitz, 2006] used an equivalence of this kind to state that pushing the box in the Monkey and Bananas domain is a specialization of a more general action "move".

The language of causal theories has been extended in [Giunchiglia *et al.*, 2004a] to handle multi-valued constants. In the extended language, a constant may assume values from an arbitrary finite set, not necessarily "true" and "false". For instance, we can express the fact that object x is in location l with $loc(x) = l$. One advantage of using $loc(x) = l$ instead of the Boolean fluent $loc(x, l)$ is that $loc(x) = l$ implicitly expresses the commonsense fact that every object is exactly in one position.

In many useful causal rules, such as (9.2)–(9.4), the formula before the "$\Leftarrow$" is a Boolean literal, a non-Boolean atom (such as $loc(x) = l$) or $\bot$. Rules of this kind are called definite. Such rules are important because a causal theory consisting of definite rules can be converted into an equivalent set of propositional formulas [McCain and Turner, 1997; Giunchiglia *et al.*, 2004a], so that its models can be computed using a satisfiability solver. That translation is used in an implementation of the definite fragment of causal logic, called the Causal Calculator, or CCalc.[1] The Causal Calculator has been applied to several problems in the theory of commonsense reasoning [Lifschitz *et al.*, 2000; Lifschitz, 2000; Akman *et al.*, 2004; Campbell and Lifschitz, 2003; Lee and Lifschitz, 2006].

9.2 Syntax and Semantics of Causal Theories

Causal theories were originally introduced in [McCain and Turner, 1997]. We review the more general syntax and semantics of causal theories — which allow multi-valued constants — from [Giunchiglia *et al.*, 2004a].

A *(multi-valued) signature* is a set σ of symbols c, called *constants*, with a set of symbols $Dom(c)$ (the *domain* of c) associated to each of them. A *(multi-valued) atom* is a string of the form $c = v$, where $c \in \sigma$ and $v \in Dom(c)$. A *(multi-valued) formula* is built from atoms using the connectives $\wedge$, $\vee$, $\neg$, $\top$ and $\bot$. Formulas of the forms $F \rightarrow G$ and $F \leftrightarrow G$ can be seen as abbreviations in the usual way.

A *(multi-valued) causal rule* is an expression of the form $F \Leftarrow G$, where F and G are formulas. These formulas are called the *head* and the *body* of the rule respectively. A *(multi-valued) causal theory* is a set of causal rules.

A *(multi-valued) interpretation* over σ is a (total) function that maps each constant c of σ to an element of $Dom(c)$. An interpretation I *satisfies* (or is a *model of*) an atom $c = v$ if $I(c) = v$. The definition of satisfaction and model of formulas

[1] `http://www.cs.utexas.edu/~tag/ccalc/` .

of more general form follows the usual rules of propositional logic.

The semantics of causal theories of [Giunchiglia *et al.*, 2004a] defines when an interpretation I of σ is a model of a causal theory T, as follows. The *reduct* T^I of T relative to I is the set of the heads of the rules of T whose bodies are satisfied by I. We say that I is a *model* of T if I is the only model of T^I. It is clear that replacing the head or the body of a causal rule by an equivalent formula doesn't change the models of a causal theory.

Take, for instance, the following causal theory with $Dom(c) = \{1, 2, 3\}$:

$$\begin{aligned} &\neg(c = 1) \vee c = 2 \Leftarrow \top \\ &\neg(c = 2) \vee c = 1 \Leftarrow \top. \end{aligned} \tag{9.5}$$

The reduct relative to any I is always

$$\{\neg(c = 1) \vee c = 2, \neg(c = 2) \vee c = 1\},$$

which is equivalent to $c = 1 \leftrightarrow c = 2$. The only model of the reduct is the interpretation J such that $J(c) = 3$. It is then clear that J is a model of (9.5), while no other interpretation I is a model of this causal theory because I is not a model of the reduct.

A rule of the form

$$l_1 \vee \cdots \vee l_n \Leftarrow G,$$

where $n \geq 0$ and $l_1, \ldots, l_n$ are literals, is said to be in *clausal form*. It is also called *semi-definite* if $n \leq 1$, and *definite* if either the head is $\bot$ ($n = 0$) or an atom. A causal theory is *in clausal form* (*semi-definite*, *definite*) if all its rules are in clausal form (respectively *semi-definite*, *definite*).

A constant c is *binary* if $|Dom(c)| = 2$. It is also called *Boolean* if $Dom(c) = \{\mathbf{t}, \mathbf{f}\}$. Signatures, formulas, causal rules and causal theories are *binary* (*Boolean*), if they contain binary (respectively, Boolean) constants only. In case of a binary

signature, the difference between definite and semi-definite causal rules is not essential, because every negative literal can be rewritten as an atom. For instance, if the underlying signature is Boolean then $\neg(c = \mathbf{t})$ is equivalent to $c = \mathbf{f}$. In case of Boolean constants c, we will often write $c = \mathbf{t}$ simply as c. If a causal theory of a Boolean signature doesn't contain atoms of the form $c = \mathbf{f}$ then the heads and bodies of its rules are essentially classical, as in the original definition of causal theories [McCain and Turner, 1997]. We call such theories MCT theories.

Take, for instance, the following MCT theory T of signature $\{p, q\}$:

$$p \vee \neg q \Leftarrow \top$$
$$q \Leftarrow p.$$

The interpretation I defined by $I(p) = I(q) = \mathbf{t}$ is a model of T. Indeed, in this case $T^I = \{p \vee \neg q, q\}$, and its only model is I. No other interpretation is a model of T: if $I(p) = \mathbf{t}$ and $I(q) = \mathbf{f}$ then I is not a model of the reduct $T^I = \{p \vee \neg q, q\}$, while if $I(p) = \mathbf{f}$ then the reduct $T^I = \{p \vee \neg q\}$ has more than one model.

9.3 Computational Methods

We are interested in methods for computing the models of a causal theory T other than by using the definition of a model directly.

The Causal Calculator uses a technique limited to definite causal theories. A finite definite causal theory can be easily turned into an equivalent set of propositional formulas using the "literal completion" procedure defined in [McCain and Turner, 1997; Giunchiglia *et al.*, 2004a]. Then the models of the causal theory can be found by running a SAT solver on this set of formulas.

An alternative computational procedure is to translate the given causal theory into a nondisjunctive logic program (the head of each rule is a literal or $\bot$) under the answer set semantics as in Proposition 6.7 from [McCain, 1997], and then invoke

an answer set solver, such as SMODELS. This is discussed in [Doğandağ *et al.*, 2001]. McCain's translation is more limited than literal completion, in the sense that, in addition to be applicable to definite theories only, those must be MCT-theories.

In this chapter we define two extensions of McCain's translation. The first one is applicable to the class of "almost definite" causal theories, a category of MCT-theories that includes all definite theories and covers some other interesting cases. One kind of interesting almost definite theories has to do with the idea of transitive closure, or reachability in a directed graph, which plays an important role in formal commonsense reasoning. The result of the translation is generally a program with nested expressions, but, in the examples mentioned above, we obtain a nondisjunctive program as with McCain's translation.

The second translation is applicable to causal theories — even multi-valued ones — that are in clausal form: the head of each rule is a disjunction of literals. This class of causal theories is very general, as every causal theory can be rewritten in clausal form. The output of this translation is a program with nested expressions with the same "models". We also show how we can modify the translation to produces smaller logic programs. This translation, together with a polynomial time "clausification" process described in Section 9.8, produces a program with nested expressions from arbitrary causal theories in polynomial time. Nested expressions can be eliminated in favor of disjunctive programs in the sense of [Gelfond and Lifschitz, 1991], by a system such as NLP.[2] We can then use answer set solvers that accept disjunctive logic programs as input (such as CMODELS, DLV and GNT) to find the models of arbitrary causal theories.

[2] `http://www.cs.uni-potsdam.de/~torsten/nlp/` .

9.4 Almost Definite Causal Theories

In this chapter we will sometimes identify the logic program connectives , and ; with the symbols used in propositional formulas $\wedge$ and $\vee$, (but, in contrast to a similar convention of Section 3.5, we don't identify negation as failure *not* with $\neg$). Under this convention, nested expressions that do not contain negation as failure are identical to propositional formulas that are formed from literals using the connectives $\wedge$, $\vee$, $\top$ and $\bot$. Such propositional formulas are said to be *in standard form*, and the literals they are formed from will be called their *component literals.*

By T we denote a causal theory whose rules have the form

$$G \rightarrow H \Leftarrow F, \tag{9.6}$$

where F, G and H are in standard form. When G is $\top$, we will identify the head $G \rightarrow H$ of this rule with H, so that definite rules (with the body written in standard form) can be viewed as a special case of (9.6).

Let $\bar{l}$ stand for the literal complementary to l. We say that l is *default false* if T contains the rule

$$\bar{l} \Leftarrow \bar{l}. \tag{9.7}$$

We say that T is *almost definite* if, in each of its rules (9.6),

- the component literals of G are default false, and
- the component literals of H are default false, or H is a conjunction of literals.

Definite theories are almost definite: in each of their rules, G is $\top$, and H is a literal or the empty conjunction $\bot$. The pair of rules

$$\begin{aligned} &\neg q \rightarrow p \Leftarrow \top \\ &q \Leftarrow q \end{aligned} \tag{9.8}$$

is an example of an almost definite theory that is not definite.

Now we will describe a translation that turns any almost definite causal theory T into a logic program Π_T. For any standard formula F, by F_{not} we denote the nested expression obtained from F by replacing each component literal l with $not\ \bar{l}$. For instance,

$$(\neg p \wedge q)_{not} = not\ p, not\ \neg q.$$

For any almost definite causal theory T, the logic program Π_T is defined as the set of the program rules

$$H \leftarrow G, F_{not} \tag{9.9}$$

for all causal rules (9.6) in T.

For instance, the translation of

$$\begin{array}{r} p \vee \neg q \Leftarrow \top \\ q \Leftarrow p, \end{array} \tag{9.10}$$

is the program

$$\begin{array}{r} p \leftarrow \top, not\ \neg q \\ q \leftarrow \top, not\ \neg q \\ \neg q \leftarrow \top, not\ q \end{array}$$

which is strongly equivalent to

$$\begin{array}{l} p \leftarrow not\ \neg q \\ q \leftarrow not\ \neg q \\ \neg q \leftarrow not\ q. \end{array} \tag{9.11}$$

The translation of (9.8) can be similarly written as

$$\begin{array}{l} p \leftarrow \neg q \\ q \leftarrow not\ \neg q. \end{array} \tag{9.12}$$

The theorem below expresses the soundness of this translation. We identify each interpretation with the set of literals that are satisfied by it. Clearly this set is complete: for each atom a, it contains either a or $\neg a$.

Theorem 9. *An interpretation is a model of an almost definite causal theory T iff it is an answer set for Π_T.*

Note that this theorem does not say anything about the answer sets for the translation Π_T that are not complete. For instance, the first of the answer sets

$$\{p, q\},\ \{\neg q\}$$

for program (9.11) is complete, and the second is not; in accordance with Theorem 9, the first answer set is identical to the only model of the corresponding causal theory (9.10). The only answer set for the translation (9.12) of causal theory (9.8) is $\{q\}$; it is incomplete, and accordingly (9.8) has no models. The incomplete answer sets of a logic program can be eliminated by adding the constraints

$$\bot \leftarrow \mathit{not}\ a, \mathit{not}\ \neg a \tag{9.13}$$

for all atoms a.

Rules of Π_T can be more complex than allowed by the preprocessor LPARSE of the system SMODELS. (The syntax of LPARSE does not permit disjunctions in the bodies of rules, as well as conjunctions and disjunctions in the heads of rules.) If, however, the formulas F and G in a rule (9.6) are conjunctions of literals, and H is a literal or $\bot$, then the translation (9.9) of that rule can be processed by LPARSE. These conditions are satisfied in many interesting cases, including the examples of almost definite causal theories discussed in the next section.

9.5 Examples

9.5.1 Transitive Closure

The transitive closure of a binary relation P on a set A can be described by an almost definite causal theory as follows. For any $x, y \in A$ such that xPy, the theory

includes the causal rule

$$p(x, y) \Leftarrow \top. \tag{9.14}$$

In the remaining causal rules, x, y and z stand for arbitrary elements of A. By default, P does not hold:

$$\neg p(x, y) \Leftarrow \neg p(x, y). \tag{9.15}$$

If xPy then there is a cause for (x, y) to satisfy the transitive closure of P:

$$tc(x, y) \Leftarrow p(x, y), \tag{9.16}$$

and there is a cause for the implication $tc(y, z) \rightarrow tc(x, z)$ to hold:

$$tc(y, z) \rightarrow tc(x, z) \Leftarrow p(x, y). \tag{9.17}$$

Finally, by default, the transitive closure relation does not hold:

$$\neg tc(x, y) \Leftarrow \neg tc(x, y). \tag{9.18}$$

The following theorem expresses that this representation of transitive closure in causal logic is adequate. By P^* we denote the transitive closure of P.

Theorem 10. *Causal theory (9.14)–(9.18) has a unique model. In this model M, an atom is true iff it has the form $p(x, y)$ where xPy, or the form $tc(x, y)$ where xP^*y.*

If we attempt to make the almost definite causal theory (9.14)–(9.18) definite by replacing (9.17) with the definite rule

$$tc(x, z) \Leftarrow p(x, y) \wedge tc(y, z)$$

then the assertion of Theorem 10 will become incorrect [Giunchiglia *et al.*, 2004a, Section 7.2].

The logic program corresponding to causal theory (9.14)–(9.18) is shown in Figure 9.1.

$$
\begin{array}{rcll}
p(x,y) & \leftarrow & \top & \text{if } xPy \\
\neg p(x,y) & \leftarrow & not\ p(x,y) & \\
tc(x,y) & \leftarrow & not\ \neg p(x,y) & \\
tc(x,z) & \leftarrow & tc(y,z), not\ \neg p(x,y) & \\
\neg tc(x,y) & \leftarrow & not\ tc(x,y) &
\end{array}
$$

Figure 9.1: Translation of causal theory (9.14)–(9.18).

Rules (9.16) and (9.17) above can be replaced with

$$p(x,y) \rightarrow tc(x,y) \Leftarrow \top$$

and

$$p(x,y) \wedge tc(y,z) \rightarrow tc(x,z) \Leftarrow \top.$$

The assertion of Theorem 10 holds for the modified theory also. Since the atoms $p(x,y)$ are default false, the modified theory is almost definite, so that Theorem 9 can be used to turn it into a logic program. Its translation differs from the one shown in Figure 9.1 in that it does not have the combination *not* $\neg$ in the third and forth lines.

The fact that the atoms $p(x,y)$ are assumed to be defined by rules of the forms (9.14) and (9.15) is not essential for the validity of Theorem 10, or for the claim that the theory is almost definite; the relation P can be characterized by any definite causal theory with a unique model. But the modification described above would not be almost definite in the absence of rules (9.15).

Transitive closure often arises in work on formalizing commonsense reasoning. For instance, Erik Sandewall's description of the Zoo World[3] says about the neighbor relation among positions that it "is symmetric, of course, and the transitive closure of the neighbor relation reaches all positions." Because of the difficulty with expressing transitive closure in the definite fragment of causal logic, this part of the specification

[3] `http://www.ida.liu.se/ext/etai/lmw/` .

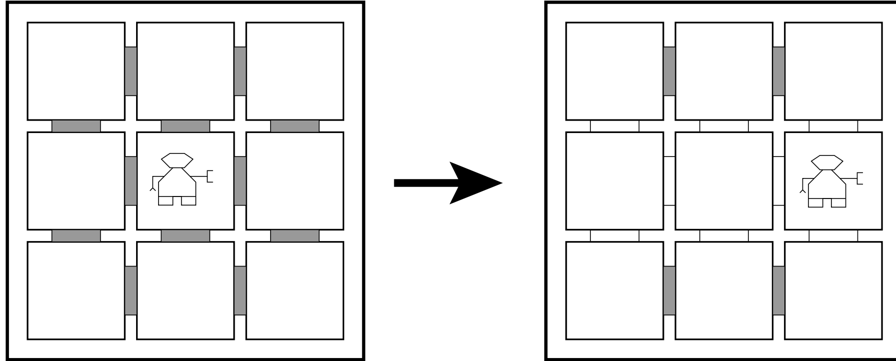

Figure 9.2: Robby's apartment is a 3×3 grid, with a door between every pair of adjacent rooms. Initially Robby is in the middle, and all doors are locked. The goal of making every room accessible from every other can be achieved by unlocking 8 doors, and the robot will have to move to other rooms in the process.

of the Zoo World is disregarded in the paper by Akman *et al.* [2004] where the Zoo World is formalized in the input of language of CCALC.

Here is another example of this kind. The apartment where Robby the Robot lives consists of several rooms connected by doors, and Robby is capable of moving around and of locking and unlocking the doors. This is a typical action domain of the kind that are easily described by definite causal theories. But the assignment given to Robby today is to unlock enough doors to make any room accessible from any other (Figure 9.2). To express this goal in the language of causal logic we need, for any time instant t, the transitive closure of the relation "there is currently an unlocked door connecting Room i with Room j." In the spirit of the representation discussed above, the transitive closure can be defined by the causal rules

$$\begin{aligned} \mathit{Accessible}(i,j)_t &\Leftarrow \mathit{Unlocked}(i,j)_t \\ \mathit{Accessible}(j,k)_t \rightarrow \mathit{Accessible}(i,k)_t &\Leftarrow \mathit{Unlocked}(i,j)_t \\ \neg \mathit{Accessible}(i,j)_t &\Leftarrow \neg \mathit{Accessible}(i,j.)_t \end{aligned} \tag{9.19}$$

The causal theories in both examples are almost definite.

$$
\begin{array}{rcl}
\neg MotorOn(G(i))_{t+1} & \Leftarrow & Toggle(S(i))_t \wedge MotorOn(G(i))_t \\
MotorOn(G(i))_{t+1} & \Leftarrow & Toggle(S(i))_t \wedge \neg MotorOn(G(i))_t \\
\neg Connected_{t+1} & \Leftarrow & Push_t \wedge Connected_t \\
Connected_{t+1} & \Leftarrow & Push_t \wedge \neg Connected_t \\
MotorOn(G(i))_{t+1} & \Leftarrow & MotorOn(G(i))_{t+1} \wedge MotorOn(G(i))_t \\
\neg MotorOn(G(i))_{t+1} & \Leftarrow & \neg MotorOn(G(i))_{t+1} \wedge \neg MotorOn(G(i))_t \\
Connected_{t+1} & \Leftarrow & Connected_{t+1} \wedge Connected_t \\
\neg Connected_{t+1} & \Leftarrow & \neg Connected_{t+1} \wedge \neg Connected_t \\
MotorOn(G(i))_0 & \Leftarrow & MotorOn(G(i))_0 \\
\neg MotorOn(G(i))_0 & \Leftarrow & \neg MotorOn(G(i))_0 \\
Connected_0 & \Leftarrow & Connected_0 \\
\neg Connected_0 & \Leftarrow & \neg Connected_0 \\
Toggle(S(i))_t & \Leftarrow & Toggle(S(i))_t \\
\neg Toggle(S(i))_t & \Leftarrow & \neg Toggle(S(i))_t \\
Push_t & \Leftarrow & Push_t \\
\neg Push_t & \Leftarrow & \neg Push_t
\end{array}
$$

$(i = 1, 2;\ t = 0, \dots, n-1)$;

$$
\begin{array}{rcl}
Turning(G(i))_t & \Leftarrow & MotorOn(G(i))_t \\
Turning(G(1))_t \leftrightarrow Turning(G(2))_t & \Leftarrow & Connected_t \\
\neg Turning(G(i))_t & \Leftarrow & \neg Turning(G(i))_t
\end{array}
$$

$(i = 1, 2;\ t = 0, \dots, n)$.

Figure 9.3: Two gears domain.

9.5.2 Two Gears

This domain, invented by Marc Denecker, is described in [McCain, 1997, Section 7.5.5] as follows:

> Imagine that there are two gears, each powered by a separate motor. There are switches that toggle the motors on and off, and a button that moves the gears so as to connect or disconnect them from one another. The motors turn the gears in opposite (i.e., compatible) directions. A gear is caused to turn if either its motor is on or it is connected to a gear that is turning.

$$
\begin{array}{rcl}
\neg MotorOn(G(i))_{t+1} & \leftarrow & not\ \neg Toggle(S(i))_t, not\ \neg MotorOn(G(i))_t \\
MotorOn(G(i))_{t+1} & \leftarrow & not\ \neg Toggle(S(i))_t, not\ MotorOn(G(i))_t \\
Connected_{t+1} & \leftarrow & not\ \neg Push_t, not\ Connected_t \\
\neg Connected_{t+1} & \leftarrow & not\ \neg Push_t, not\ \neg Connected_t \\
MotorOn(G(i))_{t+1} & \leftarrow & not\ \neg MotorOn(G(i))_{t+1}, not\ \neg MotorOn(G(i))_t \\
\neg MotorOn(G(i))_{t+1} & \leftarrow & not\ MotorOn(G(i))_{t+1}, not\ MotorOn(G(i))_t \\
Connected_{t+1} & \leftarrow & not\ \neg Connected_{t+1}, not\ \neg Connected_t \\
\neg Connected_{t+1} & \leftarrow & not\ Connected_{t+1}, not\ Connected_t \\
MotorOn(G(i))_0 & \leftarrow & not\ \neg MotorOn(G(i))_0 \\
\neg MotorOn(G(i))_0 & \leftarrow & not\ MotorOn(G(i))_0 \\
Connected_0 & \leftarrow & not\ \neg Connected_0 \\
\neg Connected_0 & \leftarrow & not\ Connected_0 \\
Toggle(S(i))_t & \leftarrow & not\ \neg Toggle(S(i))_t \\
\neg Toggle(S(i))_t & \leftarrow & not\ Toggle(S(i))_t \\
Push_t & \leftarrow & not\ \neg Push_t \\
\neg Push_t & \leftarrow & not\ Push_t
\end{array}
$$

$(i = 1, 2;\ t = 0, \dots, n-1)$,

$$
\begin{array}{rcl}
Turning(G(i))_t & \leftarrow & not\ \neg MotorOn(G(i))_t \\
Turning(G(2))_t & \leftarrow & Turning(G(1))_t, not\ \neg Connected_t \\
Turning(G(1))_t & \leftarrow & Turning(G(2))_t, not\ \neg Connected_t \\
\neg Turning(G(i))_t & \leftarrow & not\ Turning(G(i))_t
\end{array}
$$

$(i = 1, 2;\ t = 0, \dots, n)$.

Figure 9.4: Translation of the two gears domain.

McCain's representation of this domain as a causal theory is shown in Figure 9.3. The first 4 lines describe the direct effects of actions. The next 4 lines have the form (9.4) and express the commonsense law of inertia. The 8 lines that follow say that the initial values of fluents and the execution of actions are "exogenous." The last 3 lines express that a gear's motor being on causes the gear to turn, that the gears being connected causes them to turn (and not to turn) together, and that by default the gears are assumed not to turn.

Because of the second line from the end, this theory is not definite. But we can make it almost definite by replacing that line with

$$\begin{array}{rcl} Turning(G(1))_t \rightarrow Turning(G(2))_t & \Leftarrow & Connected_t \\ Turning(G(2))_t \rightarrow Turning(G(1))_t & \Leftarrow & Connected_t. \end{array}$$

By Proposition 4(i) from [Giunchiglia *et al.*, 2004a], this transformation does not change the set of models.

The corresponding logic program is shown in Figure 9.4. Many occurrences of the combination *not* $\neg$ in this program can be dropped without changing the answer sets.

9.6 Translation of causal theories in clausal form

In this section we will use some terminology from Section 9.4.

Recall, from Section 9.2, that a causal theory (non necessarily a MCT-theory) in clausal form consists of rules (in clausal form) of the form

$$l_1 \vee \cdots \vee l_n \Leftarrow G, \tag{9.20}$$

where $l_1, \ldots, l_n$ ($n \geq 0$) are literals. We will also assume that G is in standard form.

Given any causal theory T in clausal form, we define Λ_T as the program with nested expressions obtained from T

- by replacing each causal rule (9.20) by

$$l_1; \ldots; l_n \leftarrow G_{not}, (\overline{l_1}; not\ \overline{l_1}), \ldots, (\overline{l_n}; not\ \overline{l_n}) \tag{9.21}$$

  where each $\overline{l_i}$ stands for the literal complementary to l_i, and

- by adding, for every constant $c \in \sigma$ and every distinct $v, v' \in Dom(c)$, rules

$$c = v \leftrightarrow \underset{w \in Dom(c) \setminus \{v\}}{,} \neg(c = w) \tag{9.22}$$

$$\neg(c = v); \neg(c = v') \leftarrow not\ (c = v), not\ (c = v') \tag{9.23}$$

where the expression of the form $F \leftrightarrow G$ stands for two rules $F \leftarrow G$ and $G \leftarrow F$.

According to this definition, each rule (9.21) of Λ_T can be obtained from the corresponding rule of T in three steps: by

- replacing each $\wedge$ and $\vee$ with the corresponding "logic program connective",
- replacing each component literal l in the body of each rule with $not\ \bar{l}$, and
- adding some "excluded middle hypotheses" to the body of the rule.

This last step "compensates" the replacement of $\vee$ in the head of a rule with the corresponding "stronger" logic program connective. It is clear that this translation is linear if there is an upper bound on the size of the domain for each constant in T (for instance, when T is binary).

Rules (9.22) and (9.23) relate literals containing the same constant. They are needed to establish, for each interpretation I, a 1–1 relationship between the models of T^I and the subsets of I (where I is seen as the set of literals satisfied by it) that satisfy $(\Lambda_T)^I$.

For instance, if T is

$$\begin{aligned} \neg(c = 1) \vee c = 2 &\Leftarrow \top \\ \neg(c = 2) \vee c = 1 &\Leftarrow \top. \end{aligned} \tag{9.24}$$

then Λ_T is

$$\begin{array}{c}
\neg(c = 1); c = 2 \leftarrow \top, (c = 1; not\ (c = 1)), (\neg(c = 2); not\ \neg(c = 2)) \\
\neg(c = 2); c = 1 \leftarrow \top, (c = 2; not\ (c = 2)), (\neg(c = 1); not\ \neg(c = 1)) \\
c = 1 \leftrightarrow \neg(c = 2), \neg(c = 3) \\
c = 2 \leftrightarrow \neg(c = 1), \neg(c = 3) \\
c = 3 \leftrightarrow \neg(c = 1), \neg(c = 2) \\
\neg(c = 1); \neg(c = 2) \leftarrow not\ (c = 1), not\ (c = 2) \\
\neg(c = 1); \neg(c = 3) \leftarrow not\ (c = 1), not\ (c = 3) \\
\neg(c = 2); \neg(c = 3) \leftarrow not\ (c = 2), not\ (c = 3).
\end{array} \tag{9.25}$$

If T is (9.10) then Λ_T is

$$\begin{array}{c}
p; \neg q \leftarrow \top, (\neg p; not\ \neg p), (q; not\ q) \\
q \leftarrow not\ \neg p, (\neg q; not\ \neg q) \\
p \leftrightarrow \neg(p = \mathbf{f}) \\
p = \mathbf{f} \leftrightarrow \neg p \\
q \leftrightarrow \neg(q = \mathbf{f}) \\
q = \mathbf{f} \leftrightarrow \neg q \\
\neg p; \neg(p = \mathbf{f}) \leftarrow not\ p, not\ (p = \mathbf{f}) \\
\neg q; \neg(q = \mathbf{f}) \leftarrow not\ q, not\ (q = \mathbf{f}).
\end{array} \tag{9.26}$$

The theorem below expresses the soundness of this translation. We identify each interpretation with the (complete) set of literals over σ that are satisfied by the interpretation.

Theorem 11. *For any causal theory T in clausal form, the models of T are identical to the answer sets for Λ_T.*

For instance, the only answer set for (9.25) is $\{\neg(c = 1), \neg(c = 2), c = 3\}$, and indeed it is the only model of (9.24). The only answer set for (9.26) is

$$\{p, \neg(p = \mathbf{f}), q, \neg(q = \mathbf{f})\},$$

which is the only model of (9.10).

For each causal rule (9.20) that has the form $l_1 \Leftarrow G$ (i.e., $n = 1$), we can drop the "excluded middle hypothesis" from the corresponding rule (9.21) of Λ_T.

Proposition 28. *For any literal l and any nested expression F, the one-rule logic program*

$$l \leftarrow F, \left(\bar{l}; not\ \bar{l}\right)$$

is strongly equivalent to

$$l \leftarrow F.$$

For instance, the second rule of (9.26) can be rewritten as

$$q \leftarrow not\ \neg p$$

and the answer sets don't change.

However, dropping terms of the form $\overline{l_i}; not\ \overline{l_i}$ from (9.21) is usually not sound when $n > 1$. Take, for instance, the one-rule MCT causal theory:

$$p \vee \neg p \Leftarrow \top,$$

which has no models. As we expect, the corresponding logic program Λ_T:

$$\begin{aligned} p; \neg p &\leftarrow \top, (\neg p; not\ \neg p), (p; not\ p) \\ p &\leftrightarrow \neg(p = \mathbf{f}) \\ p = \mathbf{f} &\leftrightarrow \neg p \\ \neg p; \neg(p = \mathbf{f}) &\leftarrow not\ p, not\ (p = \mathbf{f}) \end{aligned} \tag{9.27}$$

has no answer sets. If we drop the two disjunctions in the body of the first rule of (9.27) we get a logic program with two answer sets $\{p, \neg(p = \mathbf{f})\}$ and $\{\neg p, p = \mathbf{f}\}$ instead.

9.7 Reducing the Size of the Translation of Causal Theories in Clausal Form

Our simplification of Λ_T depends on two parameters:

- a set S of atoms of σ such that every atom occurring in T belongs to S, and
- a set C of constants of σ such that every rule of T containing a constant from C in the head is semi-definite (see Section 9.2).

For each constant c, let N_c denote the number of atoms containing c that do not occur in S. We define the logic program $\Delta_T(S, C)$ as obtained from Λ_T by:

- dropping all rules (9.23) such that $c \in C$ or $\{c = v, c = v'\} \not\subseteq S$,
- replacing, for each constant c such that $N_c > 0$, rules (9.22) with the set of rules

  $$\underset{w \,:\, c=w \in S, w \neq v}{,} \neg(c = w) \leftarrow c = v \tag{9.28}$$

  for all $v \in Dom(c)$ such that $c = v \in S$, and
- adding

  $$\bot \leftarrow \underset{w \,:\, c=w \in S}{,} not\ (c = w) \tag{9.29}$$

  for each constant c such that $N_c > 1$.

We will denote $\Delta_T(S, \emptyset)$ by $\Delta_T(S)$. We can easily notice that $\Delta_T(S, \emptyset)$ contains atoms from S only. Clearly, when S contains all atoms of the underlying signature, $\Delta_T(S) = \Lambda_T$. Taking S smaller and C larger makes $\Delta_T(S, C)$ contain smaller and simpler rules.

Rules (9.28) impose a condition similar to the left-to-right half of (9.22), but they are limited to atoms of S. Rule (9.29) expresses, in the translation, the following fact about causal theories: if neither of two distinct atoms $c = v_1$ and

$c = v_2$ occurs in a causal theory T then no model of T maps c to v_1 or v_2. For instance, if $Dom(c) = \{1, 2, 3\}$ and only $c = 1$ occurs in T then every model of T maps c to 1. However, if $c = 2$ occurs in T as well then c can be mapped to 3, as shown by example (9.24).

For instance, if T is (9.24) then $\Delta_T(\{c = 1, c = 2\})$ is

$$
\begin{array}{r}
\neg(c = 1); c = 2 \leftarrow \top, (c = 1; \mathit{not}\ (c = 1)), (\neg(c = 2); \mathit{not}\ \neg(c = 2)) \\
\neg(c = 2); c = 1 \leftarrow \top, (c = 2; \mathit{not}\ (c = 2)), (\neg(c = 1); \mathit{not}\ \neg(c = 1)) \\
\neg(c = 2) \leftarrow c = 1 \\
\neg(c = 1) \leftarrow c = 2 \\
\neg(c = 1); \neg(c = 2) \leftarrow \mathit{not}\ (c = 1), \mathit{not}\ (c = 2)
\end{array}
\tag{9.30}
$$

If S is a set of atoms, a subset of $\{a, \neg a : a \in S\}$ is *complete over* S if it contains exactly one of the two literals a or $\neg a$ for each $a \in S$.

Theorem 12. *Let T be a causal theory over σ. Let S be a set of atoms of σ such that every atom occurring in T belongs to S, and let C be a set of constants of σ such that every rule of T containing a constant from C in the head is semi-definite. Then $I \mapsto I \cap \{a, \neg a \ : \ a \in S\}$ is a 1–1 correspondence between the models of T and the answer sets of $\Delta_T(S, C)$ that are complete over S.*

We get the models of the original causal theory by looking at the unique interpretation that satisfies each complete answer set for $\Delta_T(S, C)$. (The uniqueness of the interpretation is guaranteed by the theorem.) For instance, $\{\neg(c = 1), \neg(c = 2)\}$ is the only complete answer set for (9.30); it corresponds to the interpretation that maps c to 3, and this is indeed the only model of (9.24). Also this translation $\Delta_T(S, C)$ may have incomplete answer sets.

Similarly to Π_T, program $\Delta_T(S, C)$ may have incomplete answer sets, which don't correspond to any model of T. Also in this case, they can be eliminated from the collection of answer sets $\Delta_T(S, C)$ by adding

We can notice that no constant $c \in C$ occurs in the head of "intrinsically disjunctive" rules of $\Delta_T(S, C)$, in the following sense. If $c \in C$ then each rule (9.21) with c in the head is nondisjunctive because it comes from a semi-definite causal rule, and $\Delta_T(S, C)$ doesn't contain rules (9.23) whose head contains c. Moreover, rules (9.22) and (9.29) can be strongly equivalently rewritten as nondisjunctive rules. In particular, it is possible to translate semi-definite causal theories into nondisjunctive programs of about the same size. As a consequence, the problem of the existence of a model of a semi-definite causal theory is in class NP.

When, for a binary constant c, only one of the two atoms belongs to S, all rules (9.22) and (9.23) in Λ_T for such constant c are replaced in $\Delta_T(S, C)$ by a single rule (9.28) whose head is $\top$, which can be dropped. In particular, an MCT theory T over σ can be translated into logic program $\Delta_T(\sigma)$, essentially consisting just of rules (9.21) for all rules (9.20) in T.

For instance, if T is (9.10) then $\Delta_T(\{p, q\})$ is

$$p; \neg q \leftarrow \top, (\neg p; \mathit{not}\ \neg p), (q; \mathit{not}\ q)$$

$$q \leftarrow \mathit{not}\ \neg p, (\neg q; \mathit{not}\ \neg q)$$

whose only complete answer set is $\{p, q\}$ as expected.

9.8 Clausifying a Causal Theory

As we mentioned in the introduction, the translations from the previous sections can also be applied to arbitrary causal theories, by first converting them into clausal form. One way to do that is by rewriting the head of each rule in conjunctive normal form, and then by breaking each rule

$$C_1 \wedge \cdots \wedge C_n \Leftarrow G, \tag{9.31}$$

where $C_1, \ldots, C_n$ $(n \geq 0)$ are clauses, into n rules

$$C_i \Leftarrow G \tag{9.32}$$

$(i = 1, \ldots, n)$ [Giunchiglia *et al.*, 2004a, Proposition 4]. However, this reduction may lead to an exponential increase in size unless we assume an upper bound on the number of atoms that occur in the head of each single rule.

We propose a reduction from an arbitrary causal theory to a causal theory where the head of each rule has at most three atoms. This translation can be computed in polynomial time and requires the introduction of auxiliary Boolean atoms. The translation is similar to the one for logic programs from [Pearce *et al.*, 2002] mentioned in the introduction.

We denote each auxiliary atom by d_F, where F is a formula. For any causal theory T, the causal theory T' is obtained by T by

- replacing the head of each rule $F \Leftarrow G$ in T by d_F, and
- adding, for each subformula F that occurs in the head of rules of T,
  - $d_F \leftrightarrow F \Leftarrow \top$, if F is an atom, $\top$ and $\bot$,
  - $d_F \leftrightarrow \neg d_G \Leftarrow \top$, if F has the form $\neg G$, and
  - $d_F \leftrightarrow d_G \otimes d_H \Leftarrow \top$, if F has the form $G \otimes H$.

  ($\otimes$ denotes a binary connective.)

Intuitively, the equivalences in the heads of the rules above recursively define each atom d_F occurring in T' to be equivalent to F. This translation is clearly modular.

If T is an MCT theory then T' is an MCT theory also. For instance, MCT rule

$$p \vee (q \wedge \neg r) \Leftarrow r$$

is transformed into the following 7 MCT rules:

$$d_{p\vee(q\wedge\neg r)} \Leftarrow r$$

$$d_{p\vee(q\wedge\neg r)} \leftrightarrow d_p \vee d_{q\wedge\neg r} \Leftarrow \top$$

$$d_{q\wedge\neg r} \leftrightarrow d_q \wedge d_{\neg r} \Leftarrow \top$$

$$d_{\neg r} \leftrightarrow \neg d_r \Leftarrow \top$$

$$d_a \leftrightarrow a \Leftarrow \top \qquad (a \in \{p, q, r\})$$

Theorem 13. *For any causal theory T over a signature σ, $I \mapsto I|_\sigma$ is a 1–1 correspondence between the models of T' and the models of T.*

We can see that every rule in causal theories of the form T' is either already in clausal form, or has the body $\top$ and at most three atoms in the head. It is not hard to see that the clausification process described at the beginning of the section is linear when applied to T'.

9.9 Related work

Theorems 9, 11 and 12 extend Proposition 6.7 from [McCain, 1997]. McCain's translation transforms each rule of an MCT-theory T

$$l \Leftarrow l_1 \wedge \cdots \wedge l_n$$

($l_1, \ldots, l_n$ are literals and l is a literal or $\bot$) into a logic program rule

$$l \leftarrow not\ \overline{l_1}, \ldots, not\ \overline{l_n}.$$

It is easy to check that Π_T is identical to McCain's translation of T. Translation $\Delta_T(\sigma)$ (where σ is the set of atoms in T) differs from Π_T only for the presence of terms of the form $\overline{l_1}; not\ \overline{l_1}$ in the body of rules, and those can be eliminated by Proposition 28.

Another special case of Theorem 9 that was known before is the case when

- every atom in the language of T is default false, that is to say, T contains the causal rule $\neg a \Leftarrow \neg a$ for every atom a, and
- in every other rule (9.6) of T, F is a conjunction of negative literals, G is a conjunction of atoms, and H is a disjunction of atoms.

This special case is covered essentially by the lemma from [Giunchiglia *et al.*, 2004a, Section 7.3]. What is interesting about this case is that by forming the translation Π_T of a causal theory T satisfying the conditions above we can get an *arbitrary* set of rules of the form

$$a_1; \dots; a_m \leftarrow b_1, \dots, b_n, \mathit{not}\ c_1, \dots, \mathit{not}\ c_p \tag{9.33}$$

where $a_1, \dots, a_m, b_1, \dots, b_n, c_1, \dots, c_p$ are atoms, plus the "closed world assumption" rules

$$\neg a \leftarrow \mathit{not}\ a$$

for all atoms a. Since the problem of existence of an answer set for a finite set of rules of the form (9.33) is Σ_2^P-hard [Eiter and Gottlob, 1993, Corollary 3.8], it follows that the problem of existence of a model for an almost definite causal theory is Σ_2^P-hard also. This fact shows that from the complexity point of view almost definite causal theories are as general as arbitrary causal theories.

On the other hand, if the formula H in every rule (9.6) of an almost definite causal theory T is a literal or $\bot$ then the corresponding logic program Π_T is nondisjunctive. Consequently, the problem of existence of a model for the almost definite causal theories satisfying this condition is in class NP, just as for definite causal theories. This condition is satisfied, for instance, for both examples discussed in Section 9.5.

If a causal theory T is definite then the corresponding logic program Π_T is tight in the sense of [Erdem and Lifschitz, 2003]. The answer sets for a finite tight

program can be computed by eliminating classical negation from it in favor of additional atoms and then generating the models of the program's completion [Babovich *et al.*, 2000]. This process is essentially identical to the use of literal completion mentioned in the introduction. If T is almost definite but not definite then the program Π_T, generally, is not tight.

Even in the case MCT-theories, the classes of almost definite causal theories and of causal theories in clausal form partially overlap, neither is a subset of the other. For a causal theory T that is both almost definite and in clausal form, the two programs Π_T and Λ_T are strongly equivalent to each other.

9.10 Proofs

We begin with a comment about notation. The semantics of propositional logic defines when an interpretation satisfies a propositional formula; on the other hand, we also defined when a set of literals satisfies a nested expression. Since we have agreed to identify any interpretation with a set of literals, and to identify any standard formula with a nested expression, these two definitions of satisfaction overlap when applied to an interpretation (a complete set of literals) and a standard formula (a nested expression without negation as failure). It is easy to see, however, that the two definitions are equivalent to each other in this special case, so that we can safely use the same symbol $\models$ for both relations.

9.10.1 Proof of Theorem 9

For any nested expression F we denote by $lit(F)$ the set of literals that have regular occurrences in F. (An occurrence is *regular* if it is not an occurrence of an atom within a negative literal [Lifschitz *et al.*, 1999],) In particular, if F is a standard formula then $lit(F)$ is the set of all component literals of F. The following fact is easy to check by structural induction:

Fact 3. *For any nested expression F and any consistent set X of literals,*

$$X \models F \textit{ iff } X \cap lit(F) \models F.$$

Consider an almost definite causal theory T. For any interpretation I, the reduct T^I consists of implications $G \to H$ where G and H are standard. We will denote by $\Theta(I)$ the set of program rules

$$H \leftarrow G \tag{9.34}$$

for all implications $G \to H$ in T^I. It is clear that for any interpretation J,

$$J \models \Theta(I) \text{ iff } J \models T^I. \tag{9.35}$$

By D we denote the set of default false literals of T. Since T is almost definite, for any rule (9.34) in $\Theta(I)$

$$lit(G) \subseteq D \tag{9.36}$$

and

$$lit(H) \subseteq D \text{ or } H \text{ is a conjunction of literals.} \tag{9.37}$$

Lemma 50. *For any interpretations I, J, if $I \models \Theta(I)$ then*

$$J \models \Theta(I) \textit{ iff } I \cap J \models \Theta(I).$$

In the proof we use the following fact, which is easy to prove by structural induction.

Fact 4. *Let F be a nested expression without negation as failure, and let X, Y be consistent sets of literals. If $X \models F$ and $X \subseteq Y$ then $Y \models F$.*

Proof of Lemma 50. Case 1: $J \cap D \not\subseteq I$. Take a literal l such that $l \in J \cap D$ and $l \notin I$. Since $l \in D$, T contains the rule $\bar{l} \Leftarrow \bar{l}$. Since $l \notin I$, it follows that the formula $\bar{l}$

belongs to T^I, so that $\Theta(I)$ contains the program rule $\overline{l} \leftarrow \top$. Consequently any set of literals that satisfies $\Theta(I)$ contains $\overline{l}$. But $l \in J$, so that $\overline{l} \notin J$, which implies that neither J nor $I \cap J$ satisfies $\Theta(I)$.

Case 2: $J \cap D \subseteq I$.

Left to right. Assume that I and J satisfy $\Theta(I)$, and take any rule (9.34) in $\Theta(I)$ such that $I \cap J \models G$. We need to check that $I \cap J \models H$. By Fact 4, $I \models G$ and $J \models G$. Since I and J satisfy $\Theta(I)$, it follows that $I \models H$ and $J \models H$. According to (9.37), there are two possibilities. One is that $lit(H) \subseteq D$. Then, by the assumption of Case 2,

$$J \cap lit(H) \subseteq J \cap D \subseteq I \cap J. \tag{9.38}$$

By Fact 3, from $J \models H$ we can conclude that $J \cap lit(H) \models H$. By (9.38) and Fact 4, it follows that $I \cap J \models H$. The other possibility is that H is a conjunction of literals $l_1, \dots, l_n$. Since I and J both satisfy H, each of these literals belongs both to I and to J, so that $l_1, \dots, l_n \in I \cap J$, and we arrive at the same conclusion $I \cap J \models H$.

Right to left. Assume that $I \cap J \models \Theta(I)$, and take any rule (9.34) in $\Theta(I)$ such that $J \models G$. We need to check that $J \models H$. By Fact 3, from $J \models G$ can conclude that $J \cap lit(G) \models G$. On the other hand, by (9.36) and the assumption of Case 2,

$$J \cap lit(G) \subseteq J \cap D \subseteq I \cap J.$$

By Fact 4, it follows that $I \cap J \models G$. Since $I \cap J \models \Theta(I)$, we can conclude that $I \cap J \models H$. By Fact 4, it follows that $J \models H$. $\square$

Lemma 51. *For any consistent set X of literals and any interpretation I,*

$$X \models \Theta(I) \text{ iff } X \models (\Pi_T)^I.$$

In the proof we use the following fact, which is easy to prove by structural induction.

Fact 5. *For any standard formula F, any interpretation I and any consistent set X of literals,*

$$X \models (F_{not})^I \textit{ iff } I \models F.$$

Proof of Lemma 51. The condition $X \models (\Pi_T)^I$ means that X satisfies the reduct with respect to I of the translation (9.9) of every rule (9.6) of T. That reduct can be written as

$$H \leftarrow G, (F_{not})^I \tag{9.39}$$

Consequently, by Fact 5, the condition $X \models (\Pi_T)^I$ can be expressed by saying that, for every rule (9.6) of T, if $I \models F$ and $X \models G$ then $X \models H$. This is equivalent to $X \models \Theta(I)$. □

Theorem 9. *An interpretation is a model of an almost definite causal theory T iff it is an answer set for Π_T.*

Proof. Let I be an interpretation. The condition

$$I \text{ is a model of } T$$

means that

$$I \models T^I \text{ and, for any interpretation } J, \text{ if } J \models T^I \text{ then } J = I.$$

In view of (9.35), this is equivalent to the condition

$$I \models \Theta(I) \text{ and, for any interpretation } J, \text{ if } J \models \Theta(I) \text{ then } J = I$$

and then, by Lemma 50, to

$$I \models \Theta(I) \text{ and, for any interpretation } J, \text{ if } I \cap J \models \Theta(I) \text{ then } J = I.$$

The last condition can be expressed by saying that

$$\text{for any interpretation } J, I \cap J \models \Theta(I) \text{ iff } J = I.$$

This is further equivalent to the assertion

$$I \text{ is the only subset of } I \text{ that satisfies } \Theta(I),$$

because $J \mapsto I \cap J$ is a 1–1 correspondence between the set of all interpretations and the set of all the subsets of I. By Lemma 51, this assertion is equivalent to the claim that I is minimal among the sets satisfying $(\Pi_T)^I$, which means that I is an answer set for Π_T. ☐

9.10.2 Proof of Theorem 10

Let P be a binary relation on a set A, P^* its transitive closure, and T the causal theory (9.14)–(9.18). Define the interpretation M by

$$\begin{aligned} M \;=\; & \{p(x,y) \,:\, xPy\} \cup \{\neg p(x,y) \,:\, \text{ not } xPy\} \\ & \cup \{tc(x,y) \,:\, xP^*y\} \cup \{\neg tc(x,y) \,:\, \text{ not } xP^*y\}. \end{aligned}$$

In this notation, the theorem to be proved can be stated as follows:

Theorem 10. *M is the only model of T.*

Lemma 52. *Let I be an interpretation such that T^I is consistent. For any $x, y \in A$, if xP^*y then $T^I \models tc(x,y)$.*

Proof. Observe that

$$\begin{aligned} T^I \;=\; & \{p(x,y) \,:\, xPy\} \\ & \cup \{\neg p(x,y) \,:\, I \not\models p(x,y)\} \\ & \cup \{tc(x,y) \,:\, I \models p(x,y)\} \cup \{tc(y,z) \to tc(x,z) \,:\, I \models p(x,y)\} \\ & \cup \{\neg tc(x,y) \,:\, I \not\models tc(x,y)\}. \end{aligned} \tag{9.40}$$

Since T^I is consistent, from lines 1 and 2 of (9.40) we see that $I \models p(x,y)$ whenever xPy. Consequently, T^I contains

$$\{tc(x,y) \,:\, xPy)\} \cup \{tc(y,z) \to tc(x,z) \,:\, xPy\}.$$

It remains to notice that these formulas entail $tc(x,y)$ for all x, y such that xP^*y. ☐

Lemma 53. *For any interpretation I, if T^I is consistent and complete then $M \models T^I$.*

Proof. According to (9.40), the set of formulas in T^I that contain the atoms $p(x, y)$ is

$$\{p(x,y) \,:\, xPy\} \cup \{\neg p(x,y) \,:\, I \not\models p(x,y)\}.$$

Since T^I is consistent and complete, for any $x, y \in A$

$$xPy \text{ iff } I \models p(x,y).$$

It follows that (9.40) can be rewritten as

$$\begin{aligned} T^I \;=\;& \{p(x,y) \,:\, xPy\} \\ & \cup \{\neg p(x,y) \,:\, \text{ not } xPy\} \\ & \cup \{tc(x,y) \,:\, xPy\} \cup \{tc(y,z) \rightarrow tc(x,z) \,:\, xPy\} \\ & \cup \{\neg tc(x,y) \,:\, I \not\models tc(x,y)\}. \end{aligned} \tag{9.41}$$

M clearly satisfies the formulas in the first three lines of (9.41). To prove that M satisfies the formulas in line 4, take any x, y such that $I \not\models tc(x,y)$; we need to check that $M \not\models tc(x,y)$, or, in other words, that xP^*y doesn't hold. Assume xP^*y. Then, by Lemma 52, $T^I \models tc(x,y)$; since T^I contains the formula $\neg tc(x,y)$, this contradicts the consistency of T^I. □

Proof of Theorem 10. First we need to check that M is the only interpretation satisfying T^M. By formula (9.40) applied to $I = M$,

$$\begin{aligned} T^M \;=\;& \{p(x,y) \,:\, xPy\} \\ & \cup \{\neg p(x,y) \,:\, \text{ not } xPy\} \\ & \cup \{tc(x,y) \,:\, xPy\} \cup \{tc(y,z) \rightarrow tc(x,z) \,:\, xPy\} \\ & \cup \{\neg tc(x,y) \,:\, \text{ not } xP^*y)\}. \end{aligned} \tag{9.42}$$

It is clear that M satisfies all these formulas. Take any interpretation I satisfying T^M and any $x, y \in A$. From the first two lines of (9.42) we see that

$$I \models p(x,y) \text{ iff } xPy \text{ iff } M \models p(x,y).$$

If xP^*y then, by Lemma 52 applied to $I = M$, $T^M \models tc(x, y)$, and consequently $I \models tc(x, y)$. Otherwise, from line 4 of (9.42) we see that $I \models \neg tc(x, y)$. Thus $I = M$.

To show that an interpretation I different from M cannot be a model of T, notice that for such I, by Lemma 53, T^I either is inconsistent, or is incomplete, or is satisfied by an interpretation different from I. In each of these cases, I cannot be the only interpretation satisfying T^I. □

9.10.3 Proof of Proposition 28

We use the following property about logic programs, easily provable by induction.

Fact 6. *For any logic program* Π*, and any consistent set* X *of literals,*

$$X \models \Pi^X \text{ iff } X \models \Pi.$$

In view of this fact, we can rewrite the characterization of strong equivalence in Section 3.7, which is a rephrasing of the characterization of strong equivalence of [Turner, 2003], in the following way: two logic programs Π_1 and Π_2 are strongly equivalent iff for every consistent set Y of literals,

(a) $Y \models \Pi_1$ iff $Y \models \Pi_2$, and

(b) if $Y \models \Pi_1^Y$ then, for each $X \subset Y$, $X \models \Pi_1^Y$ iff $X \models \Pi_2^Y$.

We also use the following property about logic programs, easily provable by induction.

Proposition 28. *For any literal* l *and any nested expression* F*, the one-rule logic program*

$$l \leftarrow F, (\bar{l}; not\ \bar{l}) \tag{9.43}$$

is strongly equivalent to

$$l \leftarrow F. \tag{9.44}$$

Proof. Let Π_2 be (9.43), and Π_1 be (9.44). It is clear that condition (a) above holds. We still need to prove (b). Take any consistent set Y of literals, and assume that Y satisfies the reduct Π_1^Y:

$$l \leftarrow F^Y.$$

Case 1: $\bar{l} \notin Y$. Then $\left(\bar{l}; not\,\bar{l}\right)^Y = \left(\bar{l}; \top\right)$, which is equivalent to $\top$. Consequently $(\Pi_2)^Y$ is essentially identical to $(\Pi_1)^Y$, so that (b) holds. **Case 2:** $\bar{l} \in Y$. Then $l \notin Y$ (Y is consistent), and then, since $Y \models \Pi_1^Y$, $Y \not\models F^Y$. Consequently, since F^Y is a nested expression without negation as failure *not*, F^Y is not satisfied by any subset of Y. Consequently, all subset of Y satisfies both $(\Pi_1)^Y$ and $(\Pi_2)^Y$, because they both contain F^Y as a conjunctive term in the body. ☐

9.10.4 Proof of Theorems 11 and 12

We are going to prove Theorem 12 only, as the statement of Theorem 11 is a special case of Theorem 12: recall that $\Lambda_T = \Delta(S, \emptyset)$ where S contains all literals allowed by the signature. We will use the following properties about logic programs.

Fact 7. *Let Π be any logic program, and let Z be a set of literals not occurring in Π. Then, for any two consistent sets X and Y of literals that*

$$Y \models \Pi^X \text{ iff } Y \cap Z \models \Pi^{X \cap Z}.$$

Fact 8. *For any nested expression F and any two sets of literals X and Y such that $Y \subseteq X$,*

$$Y \models \{\bot \leftarrow F\}^X \text{ iff } Y \models \{\bot \leftarrow F\}.$$

Let T be a causal theory over σ, and let S be a set of atoms over σ that do not occur in T, and C a set of constants which do not occur in the head of nonsemi-definite rules. Let S' be $S \cup \{\neg a \ : \ a \in S\}$. By any set X of literals (for

instance, interpretations and S') and any constant c, by X_c we denote the set of literals of X that are over c.

Let Γ_1 be the set of rules (9.21) of $\Delta(S, C)$, and Γ_2 the other rules of $\Delta_T(S, C)$. We say that an interpretation is a *candidate model* over S if, for each constant c, atom $c = I(c) \in S$ whenever two or more atoms of the form $c = v$ ($v \in Dom(c) \setminus \{I(c)\}$) don't belong to S (or, alternatively, whenever $N_c \geq 2$).

Lemma 54. *Every model of T is a candidate model over S.*

Proof. Take any interpretation I that is not a candidate model. This means that, for some constant c and two values $v_1, v_2 \in Dom(c)$, $I(c) = v_1$, and $c = v_1, c = v_2$ don't belong to S. This means that those two atoms don't occur neither in T, not in T^I. Let I' be identical to I except that $I'(c) = v_2$. Consequently, if I is a model of T^I then I' is a model of T^I, so that I cannot be the unique model of T^I. We conclude that I is not a model of T.

Lemma 55. *$I \mapsto I \cap S'$ is a 1–1 correspondence between the candidate models over S and the consistent and complete set of literals over S satisfying Γ_2.*

Proof. We start by proving the right direction. It is easy to check that every interpretation satisfies rules (9.22), (9.23) and (9.28) of Γ_2. Now consider any rule (9.29) which occurs in Γ_2. This means that $N_c > 1$, so that at least two atoms of the form $c = v$ don't occur in T. If an interpretation I is a candidate model then $c = I(c)$ belongs to S, which is the condition imposed by (9.29).

Now take any consistent and complete set X of literals over S that satisfies Γ_2. Consequently,

- X doesn't contain more than one atom $c = v$ (rules (9.28) and right direction of (9.22)), and
- X contains an atom of the form $c = v$ if $N_c = 0$ (left direction of (9.22)) or $N_c > 1$ (rule (9.29)).

(Rules (9.23) are always satisfied by complete sets of literals.) It is easy to see that there is just one interpretation I such that $I \cap S' = X$: it is the one that maps each constant c to

(a) the value of v such that $c = v \in X$, if such value exists, and

(b) the only value of $v \in Dom(c)$ such that $c = v \notin S$, otherwise. (The value of v is unique, because, in this case, $N_c = 1$.)

This interpretation I is also a candidate model over S. Indeed, when $N_c > 1$ we are in case (a), and in this case $c = I(c) \in X \subseteq S$.

Lemma 56. *For any two interpretations I and J, and any causal theory T in clausal form, $I \cap J \models \Gamma_1^I$ iff J is a model of T^I.*

Proof. It is sufficient to prove the claim for the case when T is a single rule (9.20), and Γ_1 rule (9.21).

Case 1: I is not a model for G. Then $T^I = \emptyset$, and, in view of Fact 5, $(G_{not})^I$ is not satisfied by any subset of I. Consequently, since $(G_{not})^I$ is a conjunctive term in the body of Γ_1^I, every subset of I satisfies Γ_1^I. The claim immediately follows. **Case 2:** I is a model for G. Then T^I is

$$l_1 \vee \cdots \vee l_n.$$

Consider the reduct of (9.21) relative to I. We notice that the body of that rule contains a conjunctive term $(G_{not})^I$, but, in view of Fact 5, it is satisfied by all sets of literals and then it can be dropped. Also, the terms of the form $(\bar{l}_i; not\ \bar{l}_i)^X$ in the body of the reduct can be written as $\bar{l}_i$ if $\bar{l}_i \in X$, and dropped otherwise. Consequently, we can write Γ_1^I as

$$l_1; \ldots; l_n \leftarrow \underset{l \in \{l_1,\ldots,l_n\}:\bar{l} \in I}{,} \bar{l}$$

where the "big comma" is similar to $\bigwedge$ (in particular, it is $\top$ if there are no conjunctive terms). Consequently, since interpretations are complete sets of literals, (l ranges over $l_1, \dots, l_n$)

$$
\begin{aligned}
I \cap J \models \Gamma_1^I &\text{ iff } l \in I \cap J \text{ for some } l \text{ whenever } \bar{l} \in I \cap J \text{ for all } \bar{l} \in I \\
&\text{ iff } l \in J \text{ for some } l \in I \text{ or } \bar{l} \notin I \cap J \text{ for some } \bar{l} \in I \\
&\text{ iff } l \in J \text{ for some } l \in I \text{ or } \bar{l} \notin J \text{ for some } \bar{l} \in I \\
&\text{ iff } l \in J \text{ for some } l \in I \text{ or } l \in J \text{ for some } l \notin I \\
&\text{ iff } l \in J \text{ for some } l \\
&\text{ iff } J \text{ is a model of } T^I.
\end{aligned}
$$

□

Lemma 57. *For any two interpretations I and J such that I is a candidate model over S, $I \cap J \cap S' \models \Delta_T(S, C)^{I \cap S'}$ iff J is a model of T^I.*

Proof. In view of Lemma 56 and Fact 7, we have that $I \cap J \cap S' \models \Gamma_1^{I \cap S'}$ iff J is a model of T^I. It remains to show that $I \cap J \cap S' \models \Gamma_2^{I \cap S'}$. By Lemma 55 and Fact 8, $I \cap J$ satisfies the reduct of rules (9.29). The other rules of $\Gamma_2^{I \cap S'}$ are (9.22), (9.28), and rules of the form

$$\neg(c = v); \neg(c = v') \leftarrow \top \tag{9.45}$$

such that both $\neg(c = v)$ and $\neg(c = v')$ belong to S' and I. (Some trivial rules are dropped). We notice that, for each constant c and each pair of literals $\neg(c = v), \neg(c = v')$ that belong to I, at least one of them belongs to J also, so that (9.45) is satisfied by $I \cap J \cap S'$. Consider any rule (9.22) of $\Gamma_2^{I \cap S'}$. This means that $N_c = 0$, so that $(I \cap J \cap S')_c = (I \cap J)_c$. If we consider that either $(I \cap J)_c$ is I_c or a set of $|Dom(c) - 2|$ literals of the form $\neg(c = v)$ then clearly (9.22) is satisfied by $I \cap J \cap S'$. Finally, take any $c = v$ that belongs to $I \cap J \cap S'$, and consider the corresponding rule (9.28). Since $c = v \in I \cap J \cap S'$ then $I_c = J_c$, so that $\neg(c = v') \in I \cap J$ for all $v' \neq v$, and in particular it contains all conjunctive terms in the head of (9.28). □

For any interpretation I and any subset X of I, we define interpretation $J = f(I, X)$ as follows: for each constant c, $J(c)$ is

(i) $I(c)$, if either $c = I(c) \in X$, or $\neg(c = v) \in X$ for all $v \in Dom(c) \setminus \{I(c)\}$, and

(ii) an arbitrary value $v \in Dom(c) \setminus \{I(c)\}$ such that $\neg(c = v) \notin X$ and possibly such that $c = v \in S$ (if such value exists), otherwise.

Lemma 58. *For any interpretation I and any subset X of I, $J = f(I, J)$ has the following properties:*

(a) for each constant c, if $J(c) = I(c)$ then $X_c = (I \cap S')_c$,

(b) $X \subseteq J$

(c) for each constant $c \notin C$, $(I \cap J \cap S')_c = X_c$, and

(d) $I \cap J \cap S' \models \Gamma_1^{I \cap S'}$.

Proof. To prove (a), first notice that, since $X \subset I \cap S'$, $X_c \subseteq (I \cap S')_c$. It remains to show that $(I \cap S')_c \subseteq X_c$. Considering which elements belong to I_c, it is sufficient to prove the following.

(x) if $c = I(c) \in S'$ then $c = I(c) \in X$, and

(y) for all $v \in Dom(c) \setminus \{I(c)\}$, if $\neg(c = v) \in S'$ then $\neg(c = v) \in X$.

Assume that $J(c) = I(c)$, then one of the two conditions of (i) holds. **Case 1:** $c = I(c) \in X$. Claim (x) clearly holds. It is not too hard to see that (y) holds if X satisfies rule (9.28) with $v = I(c)$, or the left-to-right direction of (9.22) with the same substitution. It remains to notice that $\Delta_T(S, C)$ contains one of those two rules, that the reduct operation doesn't modify them (they don't contain negation as failure) and that $X \models (\Delta_T(S, C))^{I \cap S'}$ by hypothesis. **Case 2:** $\neg(c = v) \in X$ for all $v \in Dom(c) \setminus \{I(c)\}$. Then (y) clearly holds. To prove (x), assume that

$c = I(c) \in S'$. Notice that $X_c \subseteq S'$, so $\neg(c = v) \in S$ for all $v \in Dom(c) \setminus \{I(c)\}$. Consequently, $N_c = 0$. By the definition of Δ, $\Delta_T(S, C)$ (and then $(\Delta_T(S, C))^{I \cap S'}$) contains rules (9.22) for that constant c. Since $X \models (\Delta_T(S, C))^{I \cap S'}$ and by the hypothesis of this case, it follows that $c = I(c) \in X$.

To prove (b), first we notice that, for each c such that $J(c) = I(c)$, $X_c \subseteq J_c$ because $X \subset I$. Now consider constants c such that $J(c) \neq I(c)$, i.e., (ii) was applied. Since $X \subseteq I$, and the only atom of the form $c = v \in I$ is $c = I(c)$, X doesn't contain any atom of the form $c = v$. Consider now any literal of the form $\neg(c = v) \in X$. By the choice of J, $J(c) \neq v$, so that J contains $\neg(c = v)$.

To prove (c), consider any constant $c \notin C$. The fact that $X_c \subseteq (I \cap J \cap S')_c$ follows from the hypothesis that $X \subset I \cap S'$ and from (b). It remains to show that $(I \cap J \cap S')_c \subseteq X_c$. **Case 1**: $J(v) = I(v)$. Then $J_c = I_c$ and, by (a), $X_c = (I \cap S')_c$. Consequently,

$$(I \cap J \cap S')_c = I_c \cap J_c \cap S'_c = I_c \cap S'_c = (I \cap S')_c = X_c.$$

Case 2: $J(c) \neq I(c)$. Then I_c and J_c share only negative literals. Let $\neg(c = w)$ be one element of I_c, J_c and S'. The goal is to prove that $\neg(c = w) \in X$. Consider also that $J(c)$ has been selected using (ii). **Case 2a**: $c = J(c) \notin S$. Since we wanted $c = J(c)$ to be in S if possible, that means that

$$\text{for all } v \in Dom(c) \setminus \{I(c)\} \text{ such that } \neg(c = v) \notin X,\ c = v \notin S$$

which can be written as

$$\text{for all } v \in Dom(c) \setminus \{I(c)\}, \text{ if } c = v \in S \text{ then } \neg(c = v) \in X,$$

and then as

$$\text{if } \neg(c = v) \in I_c \text{ and } \neg(c = v) \in S'_c \text{ then } \neg(c = v) \in X_c.$$

It remains to notice that $v = w$ satisfies the "if" part, so $\neg(c = w) \in X_c$. **Case 2b**: $c = J(c) \in S$. Since both $c = w$ and $c = J(c)$ belong to S, and $c \notin C$, $\Delta_T(S, C)$

contains rule (9.23) with $v = J(c)$. Note also that I doesn't contain either $c = J(c)$ (since $J(c) \neq I(c)$) nor $c = w$ (since $\neg(c = w) \in I_c$), consequently $(\Delta_T(S, C))^{I \cap S'}$ contains rule

$$\neg(c = J(c)); \neg(c = w) \leftarrow \top.$$

Since $X \models (\Delta_T(S, C))^{I \cap S'}$ by hypothesis but $X \not\models \neg(c = J(c))$ by (ii). We can then conclude that $\neg(c = w) \in X$.

Now the proof of (d). Since the transformation $T \mapsto \Gamma_1$ is modular, it is sufficient to consider a single causal rule r of T, and the corresponding logic program rule r' of Γ_1, and prove that $I \cap J \cap S' \models (r')^{I \cap S'}$. Since $(r')^{I \cap S'} \in \Gamma_1^{I \cap S'}$ and $X \models \Gamma_1^{I \cap S'}$ then $X \models (r')^{I \cap S'}$. **Case 1:** the head of r contains an occurrence of constant $c \in C$. Then, by the definition of C, the head of r is a literal. Then r' has the form

$$l \leftarrow F_{not}, (\bar{l}; not\ \bar{l})$$

and it can be simplified, by Proposition 28, into

$$l \leftarrow F_{not}.$$

Consequently, the reduct $(r')^{I \cap S'}$ is

$$l \leftarrow (F_{not})^{I \cap S'}.$$

The only literal occurring in $\Gamma_1^{I \cap S'}$ is l in the head, so, since $X \models (r')^{I \cap S'}$, and $X \subseteq I \cap J \cap S'$ by hypothesis and (b), the claim follows. **Case 2:** the head of r doesn't contain occurrences of constants of $c \in C$. All occurrences of constants in the body of r are in the scope of negation *not* in r', then, in r', all constants occurring in $(r')^{I \cap S'}$ don't belong to C. The claim now follows from (c) and the fact that $X \models (r')^{I \cap S'}$. □

Theorem 12. *Let T be a causal theory over σ, let S be a set of atoms over σ and C a set of constants subset of σ. If S contains all atoms occurring in T and*

elements of C do not occur in the head of rules of T that are not semi-definite then $I \mapsto I \cap (S \cup \{\neg a \ : \ a \in S\})$ is a 1–1 correspondence between the models of T and the complete (over S) answer sets of $\Delta_T(S, C)$.

Proof. By Lemma 54, only interpretations that are candidate models over S can be models of T. Similarly, among the complete sets X of literals over S, only the ones that satisfy Γ_2 can be answer sets of $\Delta_T(S, C)$. Indeed, if $X \not\models \Gamma_2$ then $X \not\models \Delta_T(S, C)$, and then by Fact 6, $X \not\models (\Delta_T(S, C))^X$; we can conclude that X is not an answer set for $\Delta_T(S, C)$. Consequently, by Lemma 55, it is sufficient to show that a candidate model I over S is a model of T iff $I \cap S'$ is an answer set for $\Delta_T(S, C)$.

For the right-to-left direction, assume that $I \cap S'$ is an answer set for $\Delta_T(S, C)$. Consequently, $I \cap S' \models \Delta_T(S, C)^{I \cap S'}$ and then $I \models T^I$ by Lemma 57 (take J to be I). It remains to show that, for any interpretation $J \neq I$, J is not a model of T^I. Since $J \neq I$, there is a constant c such that $I \cap J$ doesn't contain the two literals $c = I(c), \neg(c = J(c))$ of I. If $c = I(c) \notin S$ then $c = J(c) \in S$, because I is a candidate model. Consequently, either $c = I(c)$ or $\neg(c = J(c))$ belong to S'; also, both belong to I but none to J, so that $I \cap J \cap S' \subset I \cap S'$, and then $I \cap J \cap S' \not\models (\Delta_T(S, C))^{I \cap S'}$ by the definition of an answer set. We can conclude, by Lemma 57, that $J \not\models T^I$.

For the left-to-right direction, assume that I is a model of T. This means that $I \models T^I$, so that $I \cap S' \models (\Delta_T(S, C))^{I \cap S'}$ by Lemma 57. Now take any proper subset X of $I \cap S'$, and assume, in sake of contradiction, that $X \models (\Delta_T(S, C))^{I \cap S'}$. By Lemma 58, there is a interpretation J such that $I \cap J \cap S' \models \Gamma_1^{I \cap S'}$ (claim (d)), and then that satisfies T^I by Lemma 56. Moreover, $J \neq I$ by claim (a) and the hypothesis that $X \subset J$. We get the contradictory conclusion that I is not a model of T. ∎

9.10.5 Proof of Theorem 13

Let T be a causal theory with signature σ. Let Δ be the propositional theory $\{d_F \leftrightarrow \mathit{def}(F) : F \in \mathit{form}(T)\}$, and Δ' be $\{d_F \leftrightarrow F : F \in \mathit{form}(T)\}$. Next lemma is provable by induction.

Lemma 59. *Δ is equivalent to Δ'.*

Lemma 60. *$I \mapsto I|_\sigma$ is a 1–1 correspondence between the models of $(T')^I$ and the models of $T^{I|_\sigma}$.*

Proof. Consider that $(T')^I$ is

$$\{d_F : F \Leftarrow G \in T, I \models G\} \cup \Delta.$$

In view of Lemma 59 and the fact that the body of each rule of T is over σ, $(T')^I$ is equivalent to

$$\{d_F : F \Leftarrow G \in T, I|_\sigma \models G\} \cup \Delta'$$

and then to

$$T^{I|_\sigma} \cup \Delta'.$$

We can notice that Δ' is a set of equivalences that define auxiliary atoms in terms of atoms from σ, and that $T^{I|_\sigma}$ doesn't contain auxiliary atoms. Consequently, the 1–1 correspondence between models follows from the fact that the truth value of auxiliary atoms is determined in terms of atoms from σ in a unique way.

Theorem 13. *For any causal theory T over σ, $I \mapsto I|_\sigma$ is a 1–1 correspondence between the models of T' and the models of T.*

Proof. If I is a model of T' then I is the only model of $(T')^I$. In view of Lemma 60, $I|_\sigma$ is the only model of $T^{I|_\sigma}$, and then of T. Now take any model J of T. It remains to show that, among all the interpretations I of the signature of T' such that $J = I|_\sigma$, exactly one is a model of T'. First we notice that, since T and the bodies of the rules of T' are over σ,

(i) each $T^{I|_\sigma}$ is identical to T^J, and

(ii) all $(T')^I$ are identical to each other.

From (i) we get that J is the only model of every $T^{I|_\sigma}$. Consequently, by Lemma 60, $(T')^I$ has a unique model M, with the following property: $M|_\sigma = J$. Moreover, this M is the same for all $I's$ by (ii). Note that since $M|_\sigma = J$, M is one of the interpretations I that we are considering, so that M is the only model of $(T')^M$ and consequently a model of T'. No $I \neq M$ is a model of T' because M is a model of $(T')^I$.

Chapter 10

Conclusions

We saw that answer set programming is a declarative programming paradigm with a clear semantics. Many mathematical tools for proving properties of ASP programs have been devoloped, such as the concept of strong equivalence. Extending the language to allow aggregates made many of such methods generally not applicable. We hope that this work will help reasoning about programs with aggregates, and that it clarifies the relationship between various definitions of an aggregate.

All this dissertation was about programs without variables, as the answer set semantics requires eliminating them as a preprocessing step (called grounding). On the other hand, there are several reasons why we want to have a definition of an answer set that doesn't require this preliminary step.

One of them is that variable elimination is becoming more and more time consuming compared to the newer and faster search procedure implemented in answer set solvers. For this reason, one line of research in answer set programming is in figuring out if we can avoid grounding the whole program. For instance, if we are interested in the truth value of a predicate only (say, *dark* in the Hitori example in Section 2.1.3) it may be possible to ground only the rules that "influence" such predicate. Research in this area may become simpler if we are able to reason about

programs with variables.

In this direction, we have recently proposed, in [Ferraris *et al.*, 2007], an extension of the concept of an answer set from propositional theories to first-order formulas. Its definition, however, is not based on the concept of a reduct: the answer sets of a first order formula F are the models of a second-order formula built from it. In the same paper we also proposed a characterization of strong equivalence for the new definition of an answer set. We still need to extend other theorems defined for propositional formulas, and to represent aggregates under this extended syntax.

Possible applications of this new definition of an answer set for first-order formulas are related the fact that the answer set semantics for first-order formulas is more expressive than classical first-order logic: for instance, we can define recursive definitions such as reachability, and express defaults. There are commonsense knowledge representation languages, such as the situation calculus [McCarthy and Hayes, 1969], whose meaning is defined in terms of classical first-order logic. However, in classical first-order logic, it is difficult to express the commonsense law of inertia. This problem could be avoided by redefining such languages as first-order formulas under the answer set semantics.

Bibliography

[Akman *et al.*, 2004] Varol Akman, Selim Erdoğan, Joohyung Lee, Vladimir Lifschitz, and Hudson Turner. Representing the Zoo World and the Traffic World in the language of the Causal Calculator. *Artificial Intelligence*, 153(1–2):105–140, 2004.

[Anger *et al.*, 2002] Christian Anger, Kathrin Konczak, and Thomas Linke. NoMoRe: Non-monotonic reasoning with logic programs. In *Lecture Notes in Computer Science*, volume 2424, pages 521–524. Springer-Verlag, 2002.

[Aura *et al.*, 2000] Tuomas Aura, Matt Bishop, and Dean Sniegowski. Analizing single-server network inhibition. In *Proceedings of the 13th EEEI Computer Security Foundation Workshop*, pages 108–117, 2000.

[Babovich *et al.*, 2000] Yuliya Babovich, Esra Erdem, and Vladimir Lifschitz. Fages' theorem and answer set programming.[1] In *Proceedings of International Workshop on Nonmonotonic Reasoning (NMR)*, 2000.

[Balduccini *et al.*, 2000] Marcello Balduccini, Michael Gelfond, and Monica Nogueira. A-Prolog as a tool for declarative programming. In *Proceeding of the 12th International Conference on Software Engineering and Knowledge Engineering (SEKE'2000)*, 2000.

[1] `http://arxiv.org/abs/cs.ai/0003042` .

[Balduccini *et al.*, 2001] Marcello Balduccini, Michael Gelfond, Monica Nogueira, Richard Watson, and Matthew Barry. An A-Prolog decision support system for the Space Shuttle. In *Working Notes of the AAAI Spring Symposium on Answer Set Programming*, 2001.

[Baral and Uyan, 2001] Chitta Baral and Cenk Uyan. Declarative specification and solution of combinatorial auctions using logic programming. *Lecture Notes in Computer Science*, 2173:186–199, 2001.

[Baral *et al.*, 2004] Chitta Baral, Michael Gelfond, and Richard Scherl. Using answer set programming to answer complex queries. In *Workshop on Pragmatics of Question Answering at HLT-NAAC2004*, 2004.

[Bibel and Eder, 1993] Wolfgang Bibel and Elmar Eder. A survey of logical calculi. In D.M. Gabbay, C.J. Hogger, and J.A. Robinson, editors, *The Handbook of Logic in AI and Logic Programming*, volume 1, pages 67–182. Oxford University Press, 1993.

[Cabalar and Ferraris, 2007] Pedro Cabalar and Paolo Ferraris. Propositional theories are strongly equivalent to logic programs.[2] *Theory and Practice of Logic Programming*, 2007. To appear.

[Cabalar and Santos, 2006] Pedro Cabalar and Paulo Santos. Strings and holes: An exercise on spatial reasoning. In *IBERAMIA-SBIA*, pages 419–429, 2006.

[Calimeri *et al.*, 2005] Francesco Calimeri, Wolfgang Faber, Nicola Leone, and Simona Perri. Declarative and computational properties of logic programs with aggregates. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*, 2005.

[2] `http://www.cs.utexas.edu/users/otto/papers.html` .

[Campbell and Lifschitz, 2003] Jonathan Campbell and Vladimir Lifschitz. Reinforcing a claim in commonsense reasoning. In *Working Notes of the AAAI Spring Symposium on Logical Formalizations of Commonsense Reasoning*, 2003.

[Clark, 1978] Keith Clark. Negation as failure. In Herve Gallaire and Jack Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, New York, 1978.

[De Jongh and Hendriks, 2003] Dick De Jongh and Lex Hendriks. Characterization of strongly equivalent logic programs in intermediate logics. *Theory and Practice of Logic Programming*, 3:250–270, 2003.

[Denecker *et al.*, 2001] Marc Denecker, Nikolay Pelov, and Maurice Bruynooghe. Ultimate well-founded and stable semantics for logic programs with aggregates. In *Proc. ICLP*, pages 212–226, 2001.

[Denecker *et al.*, 2002] Marc Denecker, V. Wiktor Marek, and Miroslaw Truszczynski. Ultimate approximations in nonmonotonic knowledge representation systems. In *Proc. KR*, pages 177–190, 2002.

[Dimopoulos *et al.*, 1997] Yannis Dimopoulos, Bernhard Nebel, and Jana Koehler. Encoding planning problems in non-monotonic logic programs. In Sam Steel and Rachid Alami, editors, *Proceedings of European Conference on Planning*, pages 169–181. Springer-Verlag, 1997.

[Doğandağ *et al.*, 2001] Semra Doğandağ, Ferda N. Alpaslan, and Varol Akman. Using stable model semantics (SMODELS) in the Causal Calculator (CCALC). In *Proc. 10th Turkish Symposium on Artificial Intelligence and Neural Networks*, pages 312–321, 2001.

[Eiter and Gottlob, 1993] Thomas Eiter and Georg Gottlob. Complexity results for disjunctive logic programming and application to nonmonotonic logics. In

Dale Miller, editor, *Proceedings of International Logic Programming Symposium (ILPS)*, pages 266–278, 1993.

[Eiter *et al.*, 1998] Thomas Eiter, Nicola Leone, Cristinel Mateis, Gerald Pfeifer, and Francesco Scarcello. The KR system DLV: Progress report, comparisons and benchmarks. In Anthony Cohn, Lenhart Schubert, and Stuart Shapiro, editors, *Proceedings of International Conference on Principles of Knowledge Representation and Reasoning (KR)*, pages 406–417, 1998.

[Eiter *et al.*, 1999] Thomas Eiter, Wolfgang Faber, Nicola Leone, and Gerald Pfeifer. Diagnosis frontend of the DLV system. *The European Journal of Artificial Intelligence*, 12(1–2):99–111, 1999.

[Erdem and Lifschitz, 2003] Esra Erdem and Vladimir Lifschitz. Tight logic programs. *Theory and Practice of Logic Programming*, 3:499–518, 2003.

[Erdem *et al.*, 2000] Esra Erdem, Vladimir Lifschitz, and Martin Wong. Wire routing and satisfiability planning. In *Proceedings of International Conference on Computational Logic*, pages 822–836, 2000.

[Erdem *et al.*, 2003] Esra Erdem, Vladimir Lifschitz, Luay Nakhleh, and Donald Ringe. Reconstructing the evolutionary history of Indo-European languages using answer set programming. In *Proceedings of International Symposium on Practical Aspects of Declarative Languages (PADL)*, pages 160–176, 2003.

[Erdoğan and Lifschitz, 2004] Selim T. Erdoğan and Vladimir Lifschitz. Definitions in answer set programming. In Vladimir Lifschitz and Ilkka Niemelä, editors, *Proceedings of International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, pages 114–126, 2004.

[Erdoğan and Lifschitz, 2006] Selim T. Erdoğan and Vladimir Lifschitz. Actions

as special cases. In *Proceedings of International Conference on Principles of Knowledge Representation and Reasoning (KR)*, pages 377–387, 2006.

[Faber *et al.*, 2004] Wolfgang Faber, Nicola Leone, and Gerard Pfeifer. Recursive aggregates in disjunctive logic programs: Semantics and complexity. In *Proceedings of European Conference on Logics in Artificial Intelligence (JELIA)*, 2004. Revised version: `http://www.wfaber.com/research/papers/jelia2004.pdf`.

[Ferraris *et al.*, 2007] Paolo Ferraris, Joohyung Lee, and Vladimir Lifschitz. A new perspective on stable models. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*, pages 372–379, 2007.

[Gelfond and Galloway, 2001] Michael Gelfond and Joel Galloway. Diagnosing dynamic systems in aprolog. In *Working Notes of the AAAI Spring Symposium on Answer Set Programming*, 2001.

[Gelfond and Lifschitz, 1988] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In Robert Kowalski and Kenneth Bowen, editors, *Proceedings of International Logic Programming Conference and Symposium*, pages 1070–1080, 1988.

[Gelfond and Lifschitz, 1991] Michael Gelfond and Vladimir Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9:365–385, 1991.

[Giunchiglia *et al.*, 2004a] Enrico Giunchiglia, Joohyung Lee, Vladimir Lifschitz, Norman McCain, and Hudson Turner. Nonmonotonic causal theories. *Artificial Intelligence*, 153(1–2):49–104, 2004.

[Giunchiglia *et al.*, 2004b] Enrico Giunchiglia, Yuliya Lierler, and Marco Maratea. SAT-based answer set programming. In *Proceedings of National Conference on Artificial Intelligence (AAAI)*, pages 61–66, 2004.

[Heidt, 2001] Mary Lynn Heidt. Developing an inference engine for ASET-prolog.[3] Master's thesis, University of Texas at El Paso, 2001.

[Heljanko and Niemelä, 2001] Keijo Heljanko and Ilkka Niemelä. Answer set programming and bounded model checking. In *Working Notes of the AAAI Spring Symposium on Answer Set Programming*, 2001.

[Heljanko, 1999] Keijo Heljanko. Using logic programs with stable model semantics to solve deadlock and reachability problems for 1-safe Petri nets. In *Proc. Fifth Int'l Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 218–223, 1999.

[Heyting, 1930] Arend Heyting. Die formalen Regeln der intuitionistischen Logik. *Sitzunsberichte der Preussischen Akademie von Wissenschaften. Physikalisch-mathematische Klasse*, pages 42–56, 1930.

[Hietalahti *et al.*, 2000] Maarit Hietalahti, Fabio Massacci, and Nielelä Ilkka. a challenge problem for nonmonotonic reasoning systems. In *Proceedings of the 8th International Workshop on Non-Monotonic Reasoning*, 2000.

[Janhunen *et al.*, 2003] Tomi Janhunen, Ilkka Niemelä, Dietmar Seipel, Patrik Simons, and Jia-Huai You. Unfolding partiality and disjunctions in stable model semantics. *CoRR*, cs.AI/0303009, 2003.

[Janhunen, 2000] Tomi Janhunen. Comparing the expressive powers of some syntactically restricted classes of logic programs. In *Proc. 1st International Conference on Computational Logic*, volume 1861, pages 852–866, 2000.

[Kautz and Selman, 1992] Henry Kautz and Bart Selman. Planning as satisfiability. In *Proceedings of European Conference on Artificial Intelligence (ECAI)*, pages 359–363, 1992.

[3] `http://www.krlab.cs.ttu.edu/papers/download/hei01.pdf` .

[Koksal *et al.*, 2001] Pinar Koksal, Kesim Cicekli, and I. Hakki Toroslu. Specification of wrokflow processes using the action description language $\rfloor$. In *Working Notes of the AAAI Spring Symposium on Answer Set Programming*, 2001.

[Lee and Lifschitz, 2003] Joohyung Lee and Vladimir Lifschitz. Loop formulas for disjunctive logic programs. In *Proceedings of International Conference on Logic Programming (ICLP)*, pages 451–465, 2003.

[Lee and Lifschitz, 2006] Joohyung Lee and Vladimir Lifschitz. A knowledge module: buying and selling. In *Working Notes of the AAAI Symposium on Formalizing Background Knowledge*, 2006.

[Lierler and Maratea, 2004] Yuliya Lierler and Marco Maratea. Cmodels-2: SAT-based answer set solver enhanced to non-tight programs. In *Procedings of International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, pages 346–350, 2004.

[Lifschitz and Ren, 2006] Vladimir Lifschitz and Wanwan Ren. A modular action description language. In *Proceedings of National Conference on Artificial Intelligence (AAAI)*, pages 853–859, 2006.

[Lifschitz and Turner, 1994] Vladimir Lifschitz and Hudson Turner. Splitting a logic program. In Pascal Van Hentenryck, editor, *Proceedings of International Conference on Logic Programming (ICLP)*, pages 23–37, 1994.

[Lifschitz and Woo, 1992] Vladimir Lifschitz and Thomas Woo. Answer sets in general nonmonotonic reasoning (preliminary report). In Bernhard Nebel, Charles Rich, and William Swartout, editors, *Proceedings of International Conference on Principles of Knowledge Representation and Reasoning (KR)*, pages 603–614, 1992.

[Lifschitz *et al.*, 1999] Vladimir Lifschitz, Lappoon R. Tang, and Hudson Turner. Nested expressions in logic programs. *Annals of Mathematics and Artificial Intelligence*, 25:369–389, 1999.

[Lifschitz *et al.*, 2000] Vladimir Lifschitz, Norman McCain, Emilio Remolina, and Armando Tacchella. Getting to the airport: The oldest planning problem in AI. In Jack Minker, editor, *Logic-Based Artificial Intelligence*, pages 147–165. Kluwer, 2000.

[Lifschitz *et al.*, 2001] Vladimir Lifschitz, David Pearce, and Agustin Valverde. Strongly equivalent logic programs. *ACM Transactions on Computational Logic*, 2:526–541, 2001.

[Lifschitz, 1996] Vladimir Lifschitz. Foundations of logic programming. In Gerhard Brewka, editor, *Principles of Knowledge Representation*, pages 69–128. CSLI Publications, 1996.

[Lifschitz, 1999] Vladimir Lifschitz. Answer set planning. In *Proc. ICLP-99*, pages 23–37, 1999.

[Lifschitz, 2000] Vladimir Lifschitz. Missionaries and cannibals in the Causal Calculator. In *Proceedings of International Conference on Principles of Knowledge Representation and Reasoning (KR)*, pages 85–96, 2000.

[Lin and Chen, 2005] Fangzhen Lin and Yin Chen. Discovering classes of strongly equivalent logic programs. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*, 2005. To appear.

[Lin and Zhao, 2002] Fangzhen Lin and Yuting Zhao. ASSAT: Computing answer sets of a logic program by SAT solvers. In *Proceedings of National Conference on Artificial Intelligence (AAAI)*, pages 112–117, 2002.

[Liu *et al.*, 1998] Xinxin. Liu, C. R. Ramakrishnan, and Scott A. Smolka. Fully local and efficient evaluation of alternating fixed points. In *Proc. Fourth Int'l Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 5–19, 1998.

[Lloyd and Topor, 1984] John Lloyd and Rodney Topor. Making Prolog more expressive. *Journal of Logic Programming*, 3:225–240, 1984.

[Lukasiewicz, 1941] J. Lukasiewicz. Die logik und das grundlagenproblem. *Les Entries de Zürich sur les Fondaments et la Méthode des Sciences Mathématiques*, 12(6-9 (1938)):82–100, 1941.

[Marek and Remmel, 2002] Victor Marek and Jeffrey Remmel. On logic programs with cardinality constraints. In *Proc. NMR-02*, pages 219–228, 2002.

[Marek and Truszczyński, 1991] Victor Marek and Mirosław Truszczyński. Autoepistemic logic. *Journal of ACM*, 38:588–619, 1991.

[Marek and Truszczyński, 1999] Victor Marek and Mirosław Truszczyński. Stable models and an alternative logic programming paradigm. In *The Logic Programming Paradigm: a 25-Year Perspective*, pages 375–398. Springer Verlag, 1999.

[Marek *et al.*, 2004] V. Wiktor Marek, Ilkka Niemelä, and Miroslaw Truszczynski. Logic programs with monotone cardinality atoms. In *LPNMR*, pages 154–166, 2004.

[McCain and Turner, 1997] Norman McCain and Hudson Turner. Causal theories of action and change. In *Proceedings of National Conference on Artificial Intelligence (AAAI)*, pages 460–465, 1997.

[McCain, 1997] Norman McCain. *Causality in Commonsense Reasoning about Actions.*[4] PhD thesis, University of Texas at Austin, 1997.

[4] `ftp://ftp.cs.utexas.edu/pub/techreports/tr97-25.ps.gz` .

[McCarthy and Hayes, 1969] John McCarthy and Patrick Hayes. Some philosophical problems from the standpoint of artificial intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence*, volume 4, pages 463–502. Edinburgh University Press, Edinburgh, 1969.

[Niemelä and Simons, 2000] Ilkka Niemelä and Patrik Simons. Extending the Smodels system with cardinality and weight constraints. In Jack Minker, editor, *Logic-Based Artificial Intelligence*, pages 491–521. Kluwer, 2000.

[Niemelä *et al.*, 1999] Ilkka Niemelä, Patrik Simons, and Timo Soininen. Stable model semantics for weight constraint rules. In *Logic Programming and Non-monotonic Reasoning: Proc. Fifth Int'l Conf. (Lecture Notes in Artificial Intelligence 1730)*, pages 317–331, 1999.

[Niemelä, 1999] Ilkka Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, 25:241–273, 1999.

[Osorio *et al.*, 2004] Mauricio Osorio, Juan Antonio Navarro, and José Arrazola. Safe beliefs for propositional theories. Accepted to appear at Annals of Pure and Applied Logic, 2004.

[Pearce *et al.*, 2001] David Pearce, Hans Tompits, and Stefan Woltran. Encodings for equilibrium logic and logic programs with nested expressions. In *Proceedings of Portuguese Conference on Artificial Intelligence (EPIA)*, pages 306–320, 2001.

[Pearce *et al.*, 2002] David Pearce, Torsten Schaub, Vladimir Sarsakov, Hans Tompits, and Stefan Woltran. A polynomial translation of logic programs with nested expressions into disjunctive logic programs. In *Proc. NMR-02*, 2002.

[Pearce, 1997] David Pearce. A new logical characterization of stable models and answer sets. In Jürgen Dix, Luis Pereira, and Teodor Przymusinski, editors,

Non-Monotonic Extensions of Logic Programming (Lecture Notes in Artificial Intelligence 1216), pages 57–70. Springer-Verlag, 1997.

[Pelov *et al.*, 2003] Nikolay Pelov, Marc Denecker, and Maurice Bruynooghe. Translation of aggregate programs to normal logic programs. In *Proc. Answer Set Programming*, 2003.

[Ruffolo *et al.*, 2006] Massimo Ruffolo, Marco Manna, Lorenzo Gallucci, Nicola Leone, and Domenico Saccà. A logic-based tool for semantic information extraction. In *JELIA*, pages 506–510, 2006.

[Sabuncu *et al.*, 2004] Orkunt Sabuncu, Ferda Nur Alpaslan, and Varol Akman. Using criticalities as a heuristic for answer set programming. In *Proceedings of International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, pages 234–246, 2004.

[Sakama, 2001] Chiaki Sakama. Learning by answer sets. In *Working Notes of the AAAI Spring Symposium on Answer Set Programming*, 2001.

[Simons *et al.*, 2002] Patrik Simons, Ilkka Niemelä, and Timo Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138:181–234, 2002.

[Soininen and Niemelä, 1998] Timo Soininen and Ilkka Niemelä. Developing a declarative rule language for applications in product configuration. In Gopal Gupta, editor, *Proceedings of International Symposium on Practical Aspects of Declarative Languages (PADL)*, pages 305–319. Springer-Verlag, 1998.

[Soininen *et al.*, 1999] Timo Soininen, Esthera Gelle, and Ilkka Niemelä. A fixpoint definition of dynamic constraint satisfaction. In Ronald Brachman, Hector Levesque, and Raymond Reiter, editors, *Proc. Fifth Int'l Conf. on Principles and Practice of Constraint Programming*, pages 419–433, 1999.

[Son and Lobo, 2001] Tran Cao Son and Jorge Lobo. Reasoning about policies using logic programs. In *Working Notes of the AAAI Spring Symposium on Answer Set Programming*, 2001.

[Spira, 1971] P. M. Spira. On time-hardware complexity tradeoffs for Boolean functions. In *Proceedings of the 4th Hawaii Symposium on System Sciences*, pages 525–527, North Hollywood, 1971. Western Periodicals Company.

[Subrahmanian and Zaniolo, 1995] V.S. Subrahmanian and Carlo Zaniolo. Relating stable models and AI planning domains. In *Proceedings of International Conference on Logic Programming (ICLP)*, 1995.

[Trajcevski *et al.*, 2000] Goce Trajcevski, Chitta Baral, and Jorge Lobo. Formalizing (and reasoning about) the specifications of workflows. In *Proceedings of the Fifth IFCIS International conference on Cooperative Information Systems (CoopIS'2000)*, 2000.

[Truszczynski *et al.*, 2006] Miroslaw Truszczynski, V. Wiktor Marek, and Raphael A. Finkel. Generating cellular puzzles with logic programs. In *IC-AI*, 2006.

[Turner, 2003] Hudson Turner. Strong equivalence made easy: nested expressions and weight constraints. *Theory and Practice of Logic Programming*, 3(4,5):609–622, 2003.

[Turner, 2004] Hudson Turner. Strong equivalence for causal theories. In *Proceedings of International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, pages 289–301, 2004.

[van Emden and Kowalski, 1976] Maarten van Emden and Robert Kowalski. The semantics of predicate logic as a programming language. *Journal of ACM*, 23(4):733–742, 1976.

Vita

Paolo Ferraris was born in Genova, Italy in 1972, son of Italo Ferraris and Franca Nicola. He has a brother — Enrico Ferraris — who is two years older than him. He got an high school diploma at Liceo Scientifico Cristoforo Colombo (Genova) in 1991 and a *Laurea* degree (between B.S. and M.S.) in Computer Engineering in 1999 at the Università degli Studi di Genova. After a three month visit to the United States in 2000, he worked as a software engineer at Marconi Communications, a hardware developer for telecommunication companies. In 2001 he enrolled into the PhD program in Computer Science at the University of Texas at Austin. To be mentioned are his participations at the ACM Collegiate Programming Contest world finals in 2002 and 2003 in teams representing the University of Texas at Austin.

Permanent Address: Via Padre Semeria 19/10
16131 Genova
Italy

This dissertation was typeset with LaTeX 2ε by the author.