Dottorato di Ricerca

in

Scienze Computazionali e Informatiche

Ciclo XX

Consorzio tra Università di Catania, Università di Napoli Federico II, Seconda Università di Napoli, Università di Palermo, Università di Salerno

sede amministrativa: Università di Napoli Federico II

Sabrina Baselice

On Program Grounding in ASP

2007

Tesi di Dottorato

Il Tutor
Prof. Piero A. Bonatti

Il Coordinatore
Prof. Aldo De Luca

Acknowledgments

Many thanks to Paolo Fiorenza for his help and for his confidence in myself, he proved to be a very good and generous person. Thanks to my sister for her company and support.

Dottorato di Ricerca
in
Scienze Computazionali e Informatiche

Università degli Studi di Napoli "Federico II"

On Program Grounding in ASP

Sabrina Baselice

Abstract. Answer set programming (ASP) is a declarative problem solving framework introduced by Michael Gelfond and Vladimir Lifschitz in the late '80s. ASP has received much attention by researchers for its expressiveness and simpleness so that well-engineered and optimized implementations have been developed for it. However, state-of-the-art answer set solvers have still a strong limitation: they are not be able to reason on nonground programs and then the input program have to be instantiated before the solver can start to reason on it. Consequently, answer set solvers (i) cannot handle infinite domains and (ii) use huge amounts of memory even if domains are finite. This work wants to give some contribution for these two not trivial problems.

First, I analyze *finitary programs* as a class of programs that can effectively deal with function symbols and recursion (hence infinite domains and models). Interestingly, even if finitary programs are computationally complete, their restrictions make it possible to keep complexity under control. I study the consequences of relaxing the restrictions on finitary programs and my results enforce a kind of minimality of the properties that characterize finitary programs.

Next, I investigate what happens when we *"compose"* two programs P and Q belonging to some particular classes that imposing them some restrictions guarantee good computational properties, so obtaining a program $P \cup Q$ that, as a whole, might not be subject to the restrictions of P or Q but that again enjoys good computational properties.

Finally, I study a new approach to tackle the problem (ii) of ASP. The idea is to integrate answer set generation and constraint solving to reduce the memory requirements for a class of multi-sorted logic programs with cardinality constraints: *constrained programs*. I prove some theoretical results, introduce provably sound and complete algorithms, and report experimental results on my prototype system for evaluating constrained programs, showing that my approach can solve problem instances with significantly larger domains.

Contents

I Grounding problem in ASP **1**

1 Introduction **2**
1.1 Historical perspective . . . 2
1.2 Grounding problem . . . 4
1.2.1 Previous approaches . . . 5
1.2.2 The approach of this thesis . . . 7
1.3 Structure . . . 9

2 Logic programs **11**
2.1 Basic definitions . . . 11
2.2 Stable model semantics . . . 12
2.3 Datalog . . . 14
2.4 Stratified programs . . . 15
2.5 Acyclic programs . . . 16
2.6 Splitting set . . . 17
2.7 ω-restricted programs . . . 18
2.8 Complexity results . . . 19

3 Extended logic programs **23**
3.1 Introduction . . . 23
3.2 Weight constraint rules . . . 24
3.2.1 Stable model semantics for weight constraint programs . . . 25
3.2.2 Optimization statements . . . 26
3.2.3 Weight constraint rules with variables and functions . . . 27
3.3 CLP: Constraint Logic Programming . . . 28
3.3.1 Semantics for CLP . . . 30

4 Finitary programs 32
4.1 Introduction 32
4.2 Finitary programs 33
4.3 Properties of finitary programs 34
4.3.1 Relevant subprograms 34
4.3.2 Compactness and consistency checking 36
4.3.3 Decidability and semi-decidability of inference 38
4.4 Handling local variables 39
4.5 Recognizing finitary programs 40

5 Some implementations 41
5.1 Introduction 41
5.2 Smodels 41
5.2.1 Architecture 43
5.3 DLV 45
5.3.1 Architecture 46
5.4 A prototype recognizer for finitary programs 47

II New proposals 51

6 Finitely recursive programs 52
6.1 Introduction 52
6.2 Module sequences and a normal form for splitting sequences . . . 54
6.3 Properties of finitely recursive programs 61
6.3.1 Compactness 63
6.3.2 Reasoning on finitely recursive programs 64
6.4 Skeptical resolution and finitely recursive programs 68
6.5 Conclusions 71

7 Composing normal logic programs 72
7.1 Introduction 72
7.2 Dependency relations for logic programs 72
7.3 Composing programs 73
7.4 Structural and semantic properties 82
7.5 Conclusions 91

8 Constrained programs **92**
8.1 Introduction 92
8.2 Basic terminology 93
8.3 Constrained Programs 94
8.4 Computing strong answer sets 99
8.5 The CASP prototype 105
8.6 Experimental Results 106
8.7 Conclusions 111

List of Figures

5.1 The way of a logic program in Smodels [70]. 43
5.2 Overall architecture of Smodels [58]. 43
5.3 Overall architecture of DLV [28]. 46

8.1 Morning Problem and Car-Pool Problem 106
8.2 USA Advisor Problem 108
8.3 Scheduling Problem 108
8.4 n-Queens Problem 110
8.5 Ramsey Numbers Problem 111

Part I

Grounding problem in ASP

Chapter 1

Introduction

1.1 Historical perspective

Logic programming is a declarative formalism and a logic program specifies a problem describing its domain in terms of logical relations between entities. Data can be represented both *"extensionally"*, claiming explicitly the validity of some facts, and *"intentionally"*, describing rules that recall facts and other rules. The intentional representation makes the specification of particular worlds (or domains) very compact so proving itself to be particular suited for expressing complex ideas. The implementation of a logic programming language provides an automatic theorem prover which is used to infer a description of possible solutions to the problem. This makes logic programming languages inherently *"high-level"* because developer can concentrate on what should be computed delegating the task of exploiting how to reach the result to the inferential engine.

Logic programming has been mainly employed in the field of artificial intelligence where, since in 1960, McCarthy [51] realized many of the advantages of using logic formalisms as a basis for creating a language suitable to knowledge representation: a declarative formalism ensures (i) an high-level abstraction of problems only describing their features without giving how to solve them; (ii) a modular way of expressing information: declarative sentences *"can be true in a much wider context than specific programs can be used"*; (iii) suitability for communication among different systems: *"the supplier of a fact does not have to understand much about how the receiver functions or how or whether the receiver will use it"* [51].

However, classical logic formalisms as subclasses of first order logic with Her-

brand semantics were proved to be not suitable for representing *commonsense knowledge* and *commonsense reasoning* and new logic formalisms, *nonmonotonic logics*, was needed [52, 53, 44, 63, 55, 54, 56, 37]. Indeed, commonsense reasoning is intrinsically nonmonotonic: more information forces to evaluate again previous conclusions.

The most important feature of these nonmonotonic logics is *negation as failure* [22, 62] and much technical work has been done to develop a declarative semantics for this new logic construction [43, 22, 34, 60, 68, 32, 27, 19]. This work was started by Clark and Reiter in the late 70's but although much efforts was spent for this purpose, there is still no universally accepted semantics for logic programs with negation as failure. However, among various semantics that have been so far proposed, the *stable model semantics* is the most famous and the main works on nonmonotonic logic programming of the latest years are based on it.

The stable model semantics is a declarative semantics introduced by Michael Gelfond and Vladimir Lifschitz [35] in 1988. A stable model of a logic program P is a set of ground atoms, M, such that it is *"a possible set of beliefs that a rational agent might hold given P as his promises"* [35]. M is a stable model of a logic program P if and only if M is a minimal set of ground atoms such that $P \cup M$ is closed under immediate consequence operator. It is easy to see that a logic program can have more than one stable model (eventually infinite) or none.

For this characteristic, nonmonotonic logics under the stable model semantics were proved to be a powerful representation formalism [52, 56, 63, 20] and were used in many application areas, e.g. to model combinatorial problems. In this approach a program P defines the problem to be solved and the models of P represent the set of possible solutions for the problem.

One of the most interesting achievements in the area of logic programming and nonmonotonic reasoning is a declarative problem solving framework called *answer set programming* (ASP), introduced by Gelfond and Lifschitz in [36]. The most popular ASP languages are basically extensions of function-free possibly disjunctive logic programs where negation as failure is interpreted according to the stable model semantics [35, 36]. From the expressiveness point of view, ASP languages are able to encode efficiently and uniformly all search problems within the first two levels of the polynomial hierarchy [49, 18]. Moreover, answer set solvers are proving to be competitive with other reasoners on several benchmarks [67].

In the last years, the use of nonmonotonic logics is going further to problem solving and it is getting employment in various applications such as declarative specifications. Indeed, logic programs are proved to be a useful tool for speci-

fication, verification and enforcement of security policies [17, 9] thanks to their expressiveness power, formal foundations, flexibility and declarativeness (so that users are not required to have any programming ability). Moreover, nonmonotonic semantics allow us to reason not only on the *"presence"* of specific rules but also on the *"absence"* of rules or information and on the impossibility of inferring some facts. For example, I can authorize some actions only if some information is available or other authorizations are not inferrable. However, since a logic program can have more than one stable model, for security specifications, are used subclasses of logic programs for which the stable model semantics defines a unique model guarantying, in this way, that it is always clear if an authorization is derivable or not.

Summarizing, today nonmonotonic logics are employed in many application areas the most important of which is still knowledge representation where these logics are particularly suitable for representing commonsense knowledge, reasoning about action, planning problems, verification and configuration problems, diagnostic systems, etc., but also in areas as security for specifying and verifying negotiation policies or in areas as semantic web for representing preferences and default information.

In the following with logic programs and logic programming I mean programs over nonmonotonic logics.

1.2 Grounding problem

Although answer set programming has found employment in different applications, there are still important problems to solve in order that it can compete with classical programming methodologies.

On this subject, there is much work to do for extending logic programming and all nonmonotonic formalisms, for defining a methodology of using these languages to represent various forms of nonmonotonic reasoning and to describe knowledge in specific domains, for developing query answering systems, for investigating the relationships between logic programming and other knowledge representation methods.

ASP implementations have still some limitations that prevent ASP to be employed with success in many application areas. State-of-the-art answer set solvers, indeed, are not be able to reason on nonground programs. Consequently, the computational process of all solvers has to include two main phases: a grounding phase and a successive solving phase. The answer set solvers, in fact, have to

compute the ground instantiation of the input program before they can reason on it and this instantiation process is perhaps the most expensive computational phase of these solvers. Two strong limitations follow:

1. they currently handle only finite domains, typically finite sets of constants;
2. they use huge amounts of memory even if domains are finite.

The above limitations mean that ASP languages do not support function symbols (and hence both data constructors and infinite domains) or large domains. Then, it is not possible to reason directly on recursive data structures or temporal domains so reducing the potential expressiveness of ASP [16]. However in absence of some restrictions, reasoning with function symbols and recursion is highly undecidable [49], while simply admitting large domains, e.g. temporal domains, could make memory requirements of the grounding phase of the solvers not fulfillable.

1.2.1 Previous approaches

A line of research started in 2001 [13, 16, 15, 12], has identified a very expressive fragment of normal logic programs. These programs, called *finitary programs* [13, 16], can effectively deal with function symbols and recursion (that is infinite domains and models) keeping complexity under control. Indeed, finitary programs enjoy some interesting properties: a form of compactness hold, consistency checking and ground credulous/skeptical entailment under the stable model semantics are decidable while nonground credulous/skeptical entailment is r.e.-complete. For doing this, the class of finitary programs impose that the recursion is restricted to prevent infinite sequences of recursive calls without repeats (that is, infinite paths without cycles in the dependency graph of a program) and the number of possible sources of inconsistency (i.e. the number of odd-cycles) is required to be finite. This guarantees that a query F can be answered using a finite subset of the ground instantiation of a program P, $R(P, F)$, called *relevant subprogram*. Reasoning on a part of whole ground instance of the input program may significantly reduce memory requirements and the search space. However, in many practical applications, logic programs contain many odd-cycles and then many odd-cyclic atoms. This increases the size of relevant subprogram, so reducing the advantages of this approach. Moreover, unfortunately, the class of finitary programs is undecidable. So we cannot recognize a finitary program even if there exists a sound but incomplete prototype recognizer for deciding the class of finitary programs.

Strategies for query answering that aimed at evaluating only a part of the program instantiation had already been proposed in 1986 in [5] where Ullman introduced the concept of *magic sets*: a magic set for an argument of a predicate is the set of legal values that this argument can assume consistently with values of previous arguments occurring in that predicate.

The *Magic Sets method* is a compile-time algorithm to transform logical rules into equivalent rules that can be implemented under the fixed point semantics (note that the stable model semantics had not yet been defined) more efficiently bottom-up in a way that cuts down the facts that are irrelevant for answering a query (as a top-down evaluation). Ullman defined this algorithm only for linear logic programs with function-free Horn clauses as rules (because the least fixed point of each set of function-free Horn clauses can be computed bottom-up) and not many extensions have been introduced till now. So, not only recursion is much restricted but we have also that the efficiency of this method is much affected by the order in which the predicates occurring in the body of rules are evaluated. Then optimization techniques are needed, even if these techniques are not guaranteed to be effective.

However, both these methodologies are useful only for query answering while they do not compute a complete model for a logic program. So, they are not applicable in all those classes of problems in which a model is required as, for example, planning problems where a logic program describes the state of the "world" and the possible actions that are allowed and some constraints for these actions, and a model for this program describes a sequence of actions that can be executed to reach the goal. In fact, the above methodologies can only compute the decisional problem of deciding if a sequence of actions is a solution for a planning problem but they cannot build a plan.

Another approach that evaluates only a subset of ground instantiation of the program but that computes a model for the input program, was proposed just in the early 90's [40, 41]. This approach, called *"partial instantiation"*, combines unification with mixed integer programming and can solve a nonground program with also function symbols (as finitary programs). The general strategy alternates iteratively two phases: given a nonground logic program P, in which rules with disjunctive heads and negations in their bodies can occur, first an evaluation phase, using mixed integer programming, evaluates P and generates a set of true propositional atoms and a set of false propositional atoms. Then, a phase of partial instantiation checks if a "conflict resolution" is possible between these two sets, so computing some *conflict-set unifiers*. For each unifier θ, the process is repeated on $P \cup P\theta$. This process continues until either no more conflict-set unifier is found,

or the time taken has gone beyond a certain time limit (note that partial instantiation may be infinite in the presence of function symbols). Actually, the evaluation phase computes, for a program P, $model(sizeopt(P))$, that is it computes the models of P evaluating the program $sizeopt(P)$ that has a size smaller than size of P and such that the set of models of $sizeopt(P)$ is the same of P. Moreover, some optimizations are also proposed for computing incrementally $sizeopt(P \cup P\theta)$ by reusing $sizeopt(P)$ [57].

1.2.2 The approach of this thesis

This PhD thesis wants to give some contributions for the grounding problem of which ASP suffers. In particular, I focus on two major limitations of the state-of-the-art answer set solvers: the inability of handling infinite domains and huge requirements of memory.

In Chapter 6, I analyze the class of finitary programs as a class of programs that can effectively deal with infinite domains and models admitting function symbols and recursion, and an its extension, the class of *finitely recursive programs*. Here, I investigate the consequences of relaxing those restrictions imposed on finitary programs that guarantee their good computational properties. I prove that the restriction on recursion ensures the compactness property and brings the complexity of skeptical queries within r.e. so making logic programs more similar to classical logics, while the restriction on odd-cycles (i.e. on the number of potential inconsistency sources) makes ground queries decidable and brings the complexity of nonground credulous queries within r.e. . Moreover, I prove that by only imposing that the number of potential inconsistency sources is finite we obtain a superclass of stratified programs that is Π^1_1-complete. Then, we have no advantage with respect to the class of unrestricted normal programs, while if we only impose a restriction on recursion to prevent that a ground atom may depend on infinitely many ground atoms in the dependency graph of the program, we obtain the class of finitely recursive programs that is r.e.-complete.

In general, the class of finitely recursive programs cannot be of practical interest but further restrictions are needed, e.g. restrictions on the number of odd-cycles. These results enforce a kind of minimality of the two properties that characterize finitary programs.

Next in Chapter 7, I introduce the *"composition"* of normal logic programs. Classes of programs such as finitary programs and finitely recursive programs discussed in Chapters 4 and 6 respectively, or ω-restricted programs discussed in Chapter 2, guarantee good computational properties imposing different restric-

tions on their programs. I investigate if starting from two normal programs P and Q belonging to these classes, it is possible to obtain a program $P \cup Q$ that, as a whole, might not be subject to the restrictions of P or Q but that again enjoys good computational properties. I prove that by composing a finitary program P that depends on or is independent of a normal program Q for which tasks as consistency checking and credulous inference are decidable, we obtain a program $P \cup Q$ for which consistency checking and skeptical inference are decidable, as for finitary programs, while these tasks are semidecidable if P is finitely recursive.

Finally in Chapter 8, I study a new approach to tackle the memory requirement problem of the answer set solvers. The idea is to integrate answer set generation and constraint solving to reduce memory requirements for a class of multi-sorted logic programs with cardinality constraints [67], *constrained programs* [11, 10], whose signature can be partitioned into: (i) a set of so-called *regular predicates* over domains whose size can be handled by a standard answer set solver; (ii) a set of *constrained predicates* that can be handled by a constraint solver in a way that does not require grounding (so larger domains can be allowed here); (iii) a set of predicates—called *mixed predicates*—that create a "bridge" between the above two partitions.

The reasoning on constrained programs can be implemented by having an answer set solver interact with a constraint solver. A critical aspect is the form that the definitions of mixed predicates may take. If they were completely general, then that part of the program would be just as hard to reason with as unrestricted programs because mixed predicates may range over arbitrary domains. Accordingly, the framework introduced in Chapter 8 supports restricted definitions for mixed predicates, that can be either functions from "regular" to "large" domains (strong semantics) or slightly weaker mappings where each combination of "regular" values must be associated to at least one vector of values from "large" domains (weak semantics).

I study the relationships between strong and weak semantics, and I introduce an algorithm for computing the strong semantics efficiently under the simplifying assumption that mixed predicates do not occur in the scope of negation. Moreover, I have implemented a prototype system for evaluating the class of constrained programs and then I report some experimental results providing preliminary evidence that my approach can solve problem instances with significantly larger domains.

This approach has two important advantages: first, contrary to what happens for those approaches that reason on a subset of the ground instantiation of input program, there is no class of constrained programs for which computing the strong semantics is more expensive because all those predicates, that could make

reasoning intractable, are evaluated by a constraint solver that requires no grounding. Second, this approach allows not only to answer a query but also to compute complete strong stable models (that, as I shall prove, are only a compact form for representing sets of stable models) for constrained programs.

1.3 Structure

After this first chapter that has introduced the grounding problem as one of the major problems of which ASP suffers, the historical background and the previous approaches proposed till now for this problem, that allow to better understand the context of this work, and an overview to the approaches that I propose for investigating new ways for tackling the ground problem for the answer set programming, this PhD thesis contains the following seven chapters organized as follows.

In **Chapter 2** I report some basic definitions for introducing the reader to the ASP. In particular I give some details on some important classes of programs that enjoy particular computational properties that will be recalled in successive chapters where other classes or evaluation methodologies are explained that use these programs taking advantage of their good characteristics.

Chapter 3 presents some extensions for logic programs that can help developers to express particular domains in a more intuitively and compact way. The extensions reported do not increase the actual expressiveness power of logic programs. Formalisms as *constraint logic programming* are also less expressive than normal programs but they offers a methodology for evaluating rules that does not require any grounding process, so that in Chapter 8 a new proposal for evaluating logic programs is introduced that merges answer set solving and constraint solving techniques.

As discussed in this introductory chapter, state-of-the-art answer set solvers are not be able to reason on programs with infinite ground instances because they cannot work on nonground programs and they have to instantiate the input program before the actual reasoning activity starts. So, **Chapter 4** reports the class of *finitary programs* (introduced in [13]) as a subclass of normal logic programs that allow to reason on infinite domains and models even if, in general, it is not possible to compute a whole stable model, that might be infinite, for a program.

In **Chapter 5** some implementations to work effectively with logic programs are presented. In particular two of the most known answer set solvers, Smodels and DLV, and a recognizer for a subclass of finitary programs are described.

In **Chapter 6** I exploit the role of the restrictions that make a normal program

a finitary program, that is the restriction on recursion for avoiding that a ground atom in the dependency graph of a program depends on infinitely many ground atoms and the restriction on the number of odd-cyclic atoms for avoiding that a program contains infinitely many inconsistency sources. I will prove that the first restriction makes nonmonotonic logics more similar to classical logics while the second restriction does not decrease the complexity of logic programs.

In **Chapter 7** I propose the idea of *"composing"* those programs belonging to classes of logic programs, as those presented in Chapter 2, that enjoy good computational properties so obtaining a program that, as a whole, might not be subject to the restrictions of the original programs but that again enjoys good computational properties.

In **Chapter 8** I integrate answer set generation and constraint solving to reduce the memory requirements for a class of multi-sorted logic programs with cardinality constraints whose signature can be partitioned into: (i) a set of so-called *regular predicates* over domains whose size can be handled by a standard answer set solver, as those presented in Chapter 5; (ii) a set of *constrained* predicates that can be handled by a constraint solver in a way that, as explained in Chapter 3, does not require grounding (so larger domains can be allowed here); (iii) a set of predicates—called *mixed predicates*—that create a "bridge" between the above two partitions.

Chapter 2

Logic programs

2.1 Basic definitions

The reader is assumed to be familiar with the classical theory of logic programming [47], including all its basic syntactic and semantic notions.

Let metavariables X, Y, Z range over variables, a, b, c range over constant symbols, metavariables f, g, h over function symbols, and metavariables p, q, r over predicate symbols. Metavariables A and B range over logical atoms. A literal is a formula of the form A or $\texttt{not}\,A$, where $\texttt{not}$ is the construct of negation as failure (in the latter case the literal is *negative*). The metavariable L, possibly with subscripts, will range over literals.

Definition 2.1.1 Normal logic programs *are sets of rules*

$$A \leftarrow B_1, ..., B_m, \texttt{not}\,B_{m+1}, ..., \texttt{not}\,B_n. \, (n \geq m \geq 0)$$

such that A, B_1, ..., B_n are logical atoms.

Definition 2.1.2 Disjunctive normal logic programs *are sets of rules*

$$A_1, ..., A_k \leftarrow B_1, ..., B_m, \texttt{not}\,B_{m+1}, ..., \texttt{not}\,B_n. \, (n \geq m \geq 0)$$

such that each A_j ($j = 1, ..., k$) and each B_i ($i = 1, ..., n$) are logical atoms.

By *head*(R) and *body*(R) I denote the *head* and the *body* of a rule R and I mean $\{A_1, ..., A_k\}$ and $\{B_1, ..., B_m, \texttt{not}\,B_{m+1}, ..., \texttt{not}\,B_n\}$ respectively.

According to definition, a normal rule is a special case of a disjunctive rule where in the head there is only one atom and then $k = 1$.

Note that, for a rule as $A_1, ..., A_k \leftarrow L_1, ..., L_n$, in the body $L_1, ..., L_n$ stands for $L_1 \wedge ... \wedge L_n$, while in the head $A_1, ..., A_k$ stands for $A_1 \vee ... \vee A_k$.

In the following, the metavariable P will range over logic programs.

A *local variable* of a rule R is a variable occurring in $body(R)$ and not in $head(R)$. The ground instantiation of a program P is denoted by $\mathsf{ground}(P)$ while the set of its literals is denoted by $\mathsf{lit}(P)$ and the set of its atoms is denoted by $\mathsf{atoms}(P)$. A program is *positive* if it contains no occurrence of `not`, that is the bodies of its rules do not contain negation as failure.

The construct `not` of negation as failure denotes non derivable atoms. In this way, a new kind of logical consequence $\models_n$ is defined, such that for each ground atom A

$$P \models_n \mathtt{not}\, A \Leftrightarrow P \not\models_n A$$

and such that $\models_n$ extends the classical notion of logical consequence

$$P \models A \Rightarrow P \models_n A$$

so that if the new construct `not` does not occur in a program P then P preserves its classical semantics. Hence for any positive program P^+, $\models_n$ is equivalent to the classical construct $\models$ and then

$$P^+ \models A \Leftrightarrow P^+ \models_n A.$$

It is easy to see that $\models_n$ is nonmonotonic. For example, suppose $P = \emptyset$. Given an atom A, we have that $P \models_n \mathtt{not}\, A$ while $P \cup \{A\} \not\models_n \mathtt{not}\, A$.

2.2 Stable model semantics

The stable model semantics is the most used semantics for logic programs with negation as failure. The stable model semantics is a declarative semantics introduced in 1988 by Michael Gelfond and Vladimir Lifschitz [35] for normal logic programs and it was successively *extended* to disjunctive logic programs [36].

The *Gelfond-Lifschitz transformation* P^I of a logic program P with respect to an Herbrand interpretation I (represented as usual as a set of ground atoms) is obtained by removing from $\mathsf{ground}(P)$

- all the rules with a literal $\mathtt{not}\, B$ in their body, such that $B \in I$,

- all negative literals from the remaining rules.

Note that P^I is a set of Horn clauses. Therefore, if P^I is consistent then it has a unique minimal Herbrand model, that will be denoted by $\mathsf{lm}(P^I)$.

Definition 2.2.1 *An interpretation M is a* stable model *for a normal program P if M is the least Herbrand model for P^M.*

Note that, if P is a disjunctive program then P^M may have more than one least Herbrand model.

Definition 2.2.2 *An interpretation M is a* stable model *for a disjunctive normal program P if M is one of the least Herbrand models for P^M.*

Intuitively, stable sets are *"possible sets of beliefs that a rational agent might hold given P as his premises"*, so *"if M is the set of ground atoms that I consider true"* then *"I can simplify the premises P and replace them by P^M"* and if *"M happens to be precisely the set of atoms that logically follow from the simplified set of premises P^M, then I am rational"* [35].

An *extended (disjunctive) logic program* is a set of rules of the form

$$A_1, ..., A_k \leftarrow L_1^{\neg}, ..., L_m^{\neg}, \texttt{not}\, L_{m+1}^{\neg}, ..., \texttt{not}\, L_n^{\neg}.$$

where $k \geq 1$ and $n \geq m \geq 0$, and each $L_i^{\neg}$ is a classical literal, then $L_i^{\neg}$ is an atom B or a negative atom $\neg B$, where $\neg$ is the classical negation. Then, the extended logic programs extend normal logic programs admitting classical negation in their rules.

For extended programs the definition of stable model has been extended to the notion of *answer set* [36]. An answer set is a set of ground literals S such that S is the least model (or one of the least models) of P^S, where P^S is the extended program obtained from P by deleting (i) each rule that contains $\texttt{not}\, L^{\neg}$ in its body with $L^{\neg} \in S$, and (ii) all formulas $\texttt{not}\, L^{\neg}$ in the bodies of the remaining rules. However, in [36] it was proved that extended (disjunctive) logic programs have the same expressiveness than (disjunctive) logic programs.

Definition 2.2.3 (Dependency graph.) *The* dependency graph *of a normal program P is a labelled directed graph, denoted by $DG(P)$, whose vertexes are the ground atoms of P's language. Moreover, (i) there exists an edge labelled '+' (called positive edge) from A to B if and only if for some rule $R \in \mathsf{ground}(P)$,*

A = head(R) and $B \in$ body(R); (ii) there exists an edge labelled '-' (called negative edge) from A to B if and only if for some rule $R \in$ ground*(P), A = head(R) and* `not` *$B \in$ body(R).*

The dependency graph *of an atom A occurring in a normal program P is the subgraph of $DG(P)$ including A and all vertexes reachable from A in $DG(P)$.*

Note 2.2.4 *Note that in the following of this thesis I shall refer always to ground dependency graphs.*

An atom A *depends positively* (respectively *negatively*) on B if there is a directed path from A to B in the dependency graph with an even (respectively odd) number of negative edges. Moreover, each atom depends positively on itself. If A depends positively (respectively negatively) on B we write $A \geq_+ B$ (respectively $A \geq_- B$). We write $A \geq B$ if either $A \geq_+ B$ or $A \geq_- B$.

Definition 2.2.5 *An* odd-cycle *is a cycle in the dependency graph with an odd number of negative edges. A ground atom is* odd-cyclic *if it occurs in an odd-cycle.*

Note that there exists an odd-cycle if and only if for some ground atom A, $A \geq_- A$. Then, it is easy to see that the odd-cycles represent, for a logic program, all its possible inconsistency sources, where as *inconsistent program* we mean a program with no stable model (that is different than to have an empty stable model). However, the presence of odd-cycles is only a necessary but not sufficient condition for the inconsistency. As an immediate consequence we have that any program that has no recursion through negation has at least one stable model.

2.3 Datalog

Datalog programs are a subclass of normal logic programs where only positive literals are allowed and where function symbols cannot occur.

Then, any datalog program has exactly one least Herbrand model that coincides with its only stable model (and then the stable model semantics is monotonic for such programs) and this model is always finite.

Datalog programs have some limitations that make them useless in many practical applications. In fact, since the semantics of these programs is monotonic then datalog programs, as fragment of the class of normal logic programs, renounce all advantages and motivations that have driven to study nonmonotonic logics.

So, they was successively extended so to include negation as failure in their rules and now they are referred as normal logic programs without function symbols (and then with finite ground instances and models).

2.4 Stratified programs

A program is *order consistent* if there are no infinite chains $A_1 \geq ... \geq A_i \geq ...$. Note that odd-cycles are a special case of such chains, where each atom occurs infinitely often. This means that an order consistent program has no odd-cycle and then it is always consistent.

Theorem 2.4.1 (Fages [33].) *Every order consistent normal logic program has at least one stable model.*

Locally stratified programs are particular instances of order consistent programs that have one stable model.

Definition 2.4.2 *A program P is* locally stratified *if there exists a function $\mathcal{S}$ mapping each ground atom in* $\mathsf{ground}(P)$ *onto an ordinal in such a way that for all rules in* $\mathsf{ground}(P)$

$$A \leftarrow B_1, ..., B_m, \mathtt{not}\, C_1, ..., \mathtt{not}\, C_n.$$

1. $\mathcal{S}(B_i) \leq \mathcal{S}(A)$ *($1 \leq i \leq m$), and*
2. $\mathcal{S}(C_j) < \mathcal{S}(A)$ *($1 \leq j \leq n$).*

In the above definition, if $\mathcal{S}(B_i) < \mathcal{S}(A)$ is satisfied also by all positive body atoms B_i then the program is *acyclic*.

A subclass of locally stratified programs is the class of *stratified programs* where the stratification function is defined not over the atoms in the Herbrand base of the logic program, as for locally stratified programs, but over the predicate symbols occurring in the program.

Definition 2.4.3 *A program P is* stratified *if there exists a function $\mathcal{S}$ mapping each predicate symbol in P onto an ordinal in such a way that for all rules in P*

$$p(\vec{X}) \leftarrow q_1(\vec{Y_1}), ..., q_m(\vec{Y_m}), \mathtt{not}\, r_1(\vec{Z_1}), ..., \mathtt{not}\, r_n(\vec{Z_n}).$$

1. *$\mathcal{S}(q_i) \leq \mathcal{S}(p)$ ($1 \leq i \leq m$), and*
2. *$\mathcal{S}(r_j) < \mathcal{S}(p)$ ($1 \leq j \leq n$).*

It is easy to verify that stratified programs are also locally stratified programs and again they have one stable model. In general the contrary does not hold as the following example shows.

Example 2.4.4 *Let P be the program*

$$q(a).$$
$$p(f(X)) \leftarrow \texttt{not}\ p(X).$$

and let $\mathcal{S}$ be a local stratification function such that

$$\mathcal{S}(q(a)) = 0,$$
$$\mathcal{S}(p(f^n(a))) = n,\ n \geq 0.$$

Then P is locally stratified but not stratified because the predicate symbol p depends negatively on itself.

2.5 Acyclic programs

A logic program in whose dependency graph there is no cycle is called *acyclic*.

Note that the class of acyclic programs is a subclass of locally stratified programs. Given an acyclic program P and its dependency graph $DG(P)$, we can label each vertex in $DG(P)$ with an index in $\{0, 1, ..., n\}$ (where n is the height of the graph) by visiting $DG^{-1}(P)$ (the graph obtained from $DG(P)$ by inverting the direction of its edges) in a breadth first way. In this way we label root vertexes (that there exist because $DG^{-1}(P)$ is again acyclic) with 0 and then, at each iteration, we label current vertexes with the previous label incremented by one. It is easy to see that what I have just described is a level mapping function that, for each pair of atoms A and B such that $A > B$ in P, assignes a (strictly) smaller index to B than A. This proves that the class of acyclic programs is strictly included in that of locally stratified programs. It follows that also acyclic programs have only one stable model.

2.6 Splitting set

The definition of stable models is not a constructive definition and then in order to compute a stable model of a logic program P we have to analyze any set of ground atoms M whose elements belong to the language of P and test if M is the least Herbrand model of P^M. In this section I recall the notion of "splitting" that provides us with an alternative way to compute stable models. The idea is of splitting a program into strata such that the literals in the body of a rule in any stratum either belong to that stratum or a lower stratum, and the literals in the head of a rule belong to that stratum. Note that splitting does not forbid recursion through negation (contrary to stratification), and then it does not forbid the presence of odd-cycles in the dependency graph of a logic program, that are possible inconsistency sources, but all atoms occurring in an odd-cycle must belong to same stratum in the splitted program.

Definition 2.6.1 (Splitting set [6, 46].) *A* splitting set *of a normal program P is any set U of literals such that, for any rule $R \in \mathsf{ground}(P)$, if $head(R) \in U$ then $\mathsf{lit}(R) \subseteq U$. If U is a splitting set for P, we also say that U splits P. The set of rules $R \in \mathsf{ground}(P)$ such that $\mathsf{lit}(R) \subseteq U$ is called the* bottom *of P relative to the splitting set U and is denoted by $bot_U(P)$. The subprogram $\mathsf{ground}(P) \setminus bot_U(P)$ is called the* top *of P relative to U and is denoted by $top_U(P)$.*

Definition 2.6.2 (Partial evaluation [6, 46].) *The* partial evaluation *of a normal program P with splitting set U with respect to a set of literals S is the program $e_U(\mathsf{ground}(P), S)$ defined as follows:*

$$\begin{aligned} e_U(\mathsf{ground}(P), S) = \quad & \{R' \mid \textit{there exists a rule } R \textit{ in } \mathsf{ground}(P) \textit{ such that} \\ & (body^+(R) \cap U) \subseteq S \textit{ and } (body^-(R) \cap U) \cap S = \emptyset, \\ & \textit{and } head(R') = head(R), body^+(R') = body^+(R) \setminus U, \\ & body^-(R') = body^-(R) \setminus U\} \end{aligned}$$

where $body^+(R)$ (respectively $body^-(R)$) is the set of all positive atoms A such that A occurs positively (respectively negatively) in $body(R)$.

Theorem 2.6.3 (Splitting theorem [46].) *Let U be a splitting set for a logic program P. An interpretation M is a stable model of P if and only if $M = J \cup I$, where*

1. *I is a stable model of $bot_U(P)$, and*

2. *J is a stable model of $e_U(\mathsf{ground}(P) \setminus bot_U(P), I)$.*

We can assume the stratification of a logic program as a strong form of splitting. In fact, given a locally stratified program P as in Definition 2.4.2 and any ordinal i, the set of atoms $\{A \mid A \in \mathsf{atoms}(\mathsf{ground}(P))$ *and* $\mathcal{S}(A) \leq i\}$ is a splitting set for P.

2.7 ω-restricted programs

In 2001, Syrjänen introduced the class of ω-restricted programs [71] as extension of stratified programs. As he himself says this is a *"syntactic class"* by meaning that the good properties of this class are given only by the syntactic restrictions imposed on rules.

An ω-restricted program can be divided in two part: i) a bottom, that contains a stratified program where each predicate depends positively on predicates in the same stratum or in lower strata, while it depends negatively only on predicates in strictly lower strata; ii) a top, the ω-stratum, that contains unstratifiable predicates. Moreover, each variable occurring in a rule has to occur, in that rule, in a positive body literal belonging to a strictly lower stratum. This means that each ground instantiation for the stratified part is a ground instantiation also for the ω-stratum. Then an ω-restricted program is also a *range-restricted program*.

Definition 2.7.1 *A program is* range-restricted *if in every rule each variable that occurs in the head or in a negative body literal also occurs in a positive body literal.*

Definition 2.7.2 *An ω-*stratification *for a normal program P is a function $\mathcal{S}$ that maps the predicate symbols in P onto $\mathbb{N} \cup \{\omega\}$:*

1. $\forall p \forall q(\pi^+(p, q) \implies \mathcal{S}(p) \geq \mathcal{S}(q))$,
2. $\forall p \forall q(\pi^-(p, q) \implies \mathcal{S}(p) > \mathcal{S}(q) \vee \mathcal{S}(p) = \omega)$.

where $\pi^+(p, q)$ is a path in $DG(P)$ from an atom with p as predicate symbol to an atom with q as predicate symbol without negative edges, while $\pi^-(p, q)$ is a path in $DG(P)$ from an atom with p as predicate symbol to an atom with q as predicate symbol containing negative edges.

Definition 2.7.3 *P is an ω-*restricted program *if and only if there exists an ω-stratification $\mathcal{S}$ for P such that for each rule R with p as head predicate symbol and for each variable X in R, X occurs as argument in a positive body literal with predicate symbol q and $\mathcal{S}(p) > \mathcal{S}(q)$.*

Then, an ω-stratification divides the predicate symbols in an ω-restricted program in two sets: the set of *domain predicates* containing the predicate symbols to which the ω-stratification assignes a finite value, and the set of *non-domain predicates* defined in the ω-stratum. Moreover, note that the set of domain predicates in an ω-restricted program P is a splitting set for P.

Definition 2.7.4 *Let P be a logic program. A predicate p in P is a* domain predicate *if there exists a stratification function $\mathcal{S}$ for P such that $\mathcal{S}(p)$ is finite.*

2.8 Complexity results

In this section, I report some significant results on complexity and expressiveness of logic programming paying attention to their practical consequences. Characterizing the complexity of a class of logic programs allows us to identify the class of problems that we are able to represent with that formalism and then the class of problems that we can solve. Then, the complexity for a logic formalism is a measure of its expressiveness but it shows also its limitations in describing particular problems and the computational obstacles in designing efficient programs.

I start exposing complexity results on consistency problem and entailment problem for the class of variable-free (disjunctive) normal logic programs.

Theorem 2.8.1 (Consistency [50].) *Given a ground normal logic program P, deciding whether P has a stable model is NP-complete.*

Theorem 2.8.2 (Credulous entailment [50].) *Given a ground normal logic program P, deciding whether an atom belongs to a stable model of P is NP-complete.*

Theorem 2.8.3 (Skeptical entailment [50].) *Given a ground normal logic program P, deciding whether an atom belongs to each stable model of P is coNP-complete.*

Theorem 2.8.4 (Consistency [30].) *Let P be a ground disjunctive normal logic program, deciding whether P has a stable model is Σ_2^P-complete (NP^{NP}-complete).*

Theorem 2.8.5 (Skeptical entailment [30].) *Given a ground disjunctive normal logic program P, deciding whether an atom belongs to each stable model of P is* Π_2^P*-complete (*$coNP^{NP}$*-complete).*

Note that by allowing disjunction the expressive power of stable models increases a lot. For example, with disjunction we can write a program which determines whether the maximum size of a clique in a graph is odd, which is not possible by a disjunction-free program (unless the polynomial hierarchy collapses).

The above results still hold for extended (disjunctive) logic programs. This means that admitting classical negation occurs in rules does not increase the expressiveness of logic programs.

Let us now consider programs with variables. The above results still hold for nonground logic programs if and only if their ground instantiation consists of a finite set of ground rules. In general, we can only talk about definability of relations defined by logic programs.

Definition 2.8.6 *A relation s on the set of ground terms of a language* $\mathcal{L}$ *is* definable *in logic programming under semantics* $\models_n$ *if there exists a program P and a predicate symbol p in the language* $\mathcal{L}$ *such that for each ground term t of* $\mathcal{L}$

$$s(t) \equiv P \models_n p(t) \quad or \quad s(t) \equiv P \models_n \mathtt{not}\, p(t).$$

To discuss the expressive power of logic programs, I will need to recall some important complexity classes which are beyond the polynomial hierarchy: the arithmetical and analytical hierarchies.

Arithmetical Hierarchy:

- Σ_0^0: class of *recursive* decision problems.
- Σ_1^0: class of *recursively enumerable* decision problems.
- Σ_{n+1}^0: class of relations definable by means of a *first order formula*

  $$\Psi(\vec{X}) = \exists \vec{X_0} \forall \vec{X_1} ... Q_k \vec{X_n} \psi(\vec{X_0}, ..., \vec{X_n}, \vec{Y})$$

  with free variables $\vec{Y}$, with Q_i is $\forall$ if i is odd and $\exists$ if i is even, and with ψ quantifier free and recursive.
- Π_{n+1}^0 =co-Σ_{n+1}^0: class of relations definable by means of a *first order formula*

  $$\Psi(\vec{X}) = \forall \vec{X_0} \exists \vec{X_1} ... Q_k \vec{X_n} \psi(\vec{X_0}, ..., \vec{X_n}, \vec{Y})$$

with free variables $\vec{Y}$, with Q_i is $\exists$ if i is odd and $\forall$ if i is even, and with ψ quantifier free and recursive.

Analytical Hierarchy:

- Σ^1_1: class of relations definable by means of a *second order formula*

$$\Psi(\vec{X}) = \exists \vec{P} \psi(\vec{P}; \vec{X})$$

  where $\vec{P}$ is a tuple of predicate variables and ψ is a first order formula with free variables $\vec{X}$.

- Π^1_1: class of relations definable by means of a *second order formula*

$$\Psi(\vec{X}) = \forall \vec{P} \psi(\vec{P}; \vec{X})$$

  where $\vec{P}$ is a tuple of predicate variables and ψ is a first order formula with free variables $\vec{X}$.

- Σ^1_{n+1}: class of relations definable by means of a *second order formula*

$$\Psi(\vec{X}) = \exists \vec{P}_0 \forall \vec{P}_1 ... Q_k \vec{P}_n \psi(\vec{P}_0, ..., \vec{P}_n, \vec{X})$$

  where Q_i is $\forall$ if i is odd and $\exists$ if i is even, $\vec{P}_i$ are tuples of predicate variables and ψ is a first order formula with free variables $\vec{X}$.

- Π^1_{n+1}: class of relations definable by means of a *second order formula*

$$\Psi(\vec{X}) = \forall \vec{P}_0 \exists \vec{P}_1 ... Q_k \vec{P}_n \psi(\vec{P}_0, ..., \vec{P}_n, \vec{X})$$

  where Q_i is $\exists$ if i is odd and $\forall$ if i is even, $\vec{P}_i$ are tuples of predicate variables and ψ is a first order formula with free variables $\vec{X}$.

It is well-known that general Σ^1_1 formulas are far more expressive than first order formulas, and general Π^1_2 formulas are far more expressive than Σ^1_1 formulas, and so on.

The following theorem characterizes the expressiveness of the stable model semantics for normal logic programs.

Theorem 2.8.7 ([48, 65].) *Determining if a literal L is a stable model consequence of a normal program P is Π^1_1-complete.*

Then the problem of determining if a literal L is a stable model consequence of a program P is representative of the hardest decision problem in Π^1_1.

If we restrict the class of logic programs imposing some additional constraints about the form of the rules, the expressive power decreases and consequently, as we expect, also the complexity decreases.

Theorem 2.8.8 ([50, 65].) *Datalog programs are $coNEXPTIME$-complete.*

Theorem 2.8.9 ([3].) *The class of stratified logic programs is Σ^0_{n+1}-complete (with n levels of stratification).*

ω-restricted programs enjoy good computational properties. Indeed, checking whether an ω-restricted program P has a stable model is decidable even if function symbols are admitted.

Theorem 2.8.10 ([71].) *Given an ω-restricted program P, deciding whether P has a stable model is 2-$NEXP$-complete.*

Moreover, checking whether a ground atom A belongs to the unique stable model M of the bottom of an ω-restricted program P defining the domain predicates or whether A occurs in the head of some rule in the ground instantiation of the ω-stratum with respect to M is decidable.

Chapter 3

Extended logic programs

3.1 Introduction

The language of logic programs has been often extended not only for increasing its expressiveness but also for representing problems in a more compact and simple way. It is the case of weight constraint rules that, even if I shall prove that they can be translated into equivalent disjunctive logic rules, are particularly suited for representing choices over sets of elements with different weights in a domain that imposes lower and upper bounds on total weight of selected elements. Suppose to model configuration problems where a combination of components, that respects a set of restrictions according to the type of chosen elements, has to be calculated, or to model scheduling problems where a set of activities with some times and resources requirements has to be executed satisfying global temporal and resource constraints. Weight constraint rules are very useful for representing such problems. It follows an example of a configuration problem presented in [67].

Example 3.1.1 (Simons [67].) *A simplified configuration model of a PC could include the following. There is a set of different types of IDE hard disks, software packages, and other components that can be chosen to be parts of a PC. A PC must have from one to four IDE hard disks. In addition, the software packages use and hard disks produce disk space. Different types of hard disks provide and different software packages use varied amount of disk space. The amount of disk space provided in a configuration must be larger than its use.*

Moreover, often optimal combinations of elements have to be found.

Sometimes formalisms are useful for their methodology of evaluating rules instead of their expressiveness. For example, *constraint logic programming* (CLP)

rules have the the same expressiveness of first order logic formulas but these rules are evaluated without any grounding process is required. For this characteristic, CLP has been proved to be very useful for reasoning on programs with large domains whose ground instance may not be effectively computed by modern systems. In fact in Chapter 8, I combine normal logic programs with CLP rules so to reduce memory requirements of modern answer set solvers.

3.2 Weight constraint rules

Weight constraint rules was introduced by Patrik Simons in [66] for extending normal logic programs in order to express in a compact way choices with lower and upper cardinality bounds among a set of alternatives where each element can have a different weight so that the admissible configurations may be only a subset of all possible combinations of elements. Then, weight constraint rules are particularly suited for specifying problems in many application areas such as configuration or scheduling problems.

Definition 3.2.1 *A* weight constraint *has a form as*

$$l \leq \{A_1 = w_{A_1}, \ldots, A_n = w_{A_n}, not\ B_1 = w_{B_1}, \ldots, not\ B_m = w_{B_m}\} \leq u \tag{3.1}$$

where A_i and B_i are atoms, w_{A_i} and w_{B_i} are real numbers and represent the weights associated to literals A_i and `not` *B_i, respectively, while l and u are real numbers and represent the lower and the upper bounds of the constraint, respectively.*

Note that both weights and bounds are real numbers so that also negative weights or bounds are allowed.

Intuitively, a set of atoms S satisfies a weight constraint C if and only if the sum of the weights of all literals in C satisfied by S is a value between the lower and the upper bounds of C. When the lower bound is omitted it is assumed to be $-\infty$, while when the upper bound is omitted then $+\infty$ is assumed as upper bound.

If we impose that all weights occurring in a weight constraint are to be equal to 1 then we obtain, as a special case, a *cardinality constraint*.

Definition 3.2.2 *A* cardinality constraint *has a form as*

$$l\,\{A_1, \ldots, A_n, not\ B_1, \ldots, not\ B_m\}\,u$$

where A_i and B_i are atoms and l and u are real numbers and represent the lower and the upper bounds of the constraint, respectively. The weights of literals in the constraint are assumed to be equal to 1.

A *weight constraint rule* is a rule as

$$C_0 \leftarrow C_1, \ldots, C_n.$$

where each C_i is a weight constraint.

An *integrity constraint*

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

is, instead, a particular type of weight constraint rule where the head is assumed to be an unsatisfiable weight constraint as $1 \leq \{\}$.

Then, a *weight constraint program* is a set of weight constraint rules.

The weight constraint rules do not really increase the expressiveness of logic programs. Indeed, the computational complexity is the same. This means that using cardinality and weight constraints is only a more compact way for modeling some particular domains.

3.2.1 Stable model semantics for weight constraint programs

The stable model semantics can be extended to weight constraint rules and then, to weight constraint programs extending the notion of *reduct of a program* as defined in [35], to the class of weight constraint programs.

As for normal programs, a stable model for a weight constraint rule program P is a set of ground atoms M such that M satisfies all rules in P. Then for each rule $C_0 \leftarrow C_1, \ldots, C_n$ in P, M satisfies C_0 whenever it satisfies $C_1, \ldots, C_n$ and a weight constraint C as (3.1) is satisfied by M if and only if $l \leq w(C, M) \leq u$, where

$$w(C, M) = \Sigma_{A_i \in M} w_{A_i} + \Sigma_{B_i \notin M} w_{B_i}$$

is the sum of the weights of the literals in C satisfied by M.

Definition 3.2.3 ([66].) *The* reduct C^M of a weight constraint C *as in (3.1) with respect to a set of atoms M is the constraint*

$$l' \leq \{A_1 = w_{A_1}, \ldots, A_n = w_{A_n}\}$$

where

$$l' = l - \Sigma_{B_i \notin M} w_{B_i}.$$

Then, the reduct C^M of a constraint C is obtained deleting from C all negative literals and its upper bound, and decreasing its lower bound by the weights of all its negative literals satisfied by M.

The reduct of a weight constraint rule $C_0 \leftarrow C_1, \ldots, C_n$ is $H \leftarrow C_1^M, \ldots, C_n^M$ where H is an atom and each C_i^M is the reduct of C_i with respect to M.

The reduct P^M of a weight constraint program P with respect to a set of atoms M is the set of rules

$$H \leftarrow C_1^M, \ldots, C_n^M.$$

such that, for each of them, there exists in P a rule $R : C_0 \leftarrow C_1, \ldots, C_n$ with $H \in \mathsf{lit}(C_0)$ and $H \in M$ and such that M satisfies the upper bounds of each weight constraint in R.

Definition 3.2.4 ([66].) *Let P be a weight constraint program and M be a set of atoms. The reduct P^M of P with respect to M is*

$$\begin{aligned} P^M = \quad & \{H \leftarrow C_1^M, \ldots, C_n^M \mid C_0 \leftarrow C_1, \ldots, C_n \in P \text{ and} \\ & \text{for each } C_i \text{ as in (3.1), } H \in \mathsf{lit}(C_0) \cap M \text{ and } w(C_i, M) \le u \\ & \text{where } i = 1, \ldots, n\}. \end{aligned}$$

A stable model for a weight constraint program P is a set of atoms M that satisfies all rules in P and that is the deductive closure $lm(P^M)$ of the reduct of P with respect to M.

Definition 3.2.5 *A set of atoms M is a stable model for a weight constraint program P with non-negative weights if and only if the following holds*

1. $M \models P$,
2. $M = lm(P^M)$.

Note that the reduct of a weight constraint rule is a *Horn constraint rule*. In fact, each C_i contains only positive literals and only a lower bound condition. Moreover, for a set P of Horn constraint rules, the deductive closure $lm(P)$ is the unique least set of ground atoms such that for each atom A, if $P \models A$ then $A \in lm(P)$. This set $lm(P)$ is unique because the Horn constraint rules are monotonic.

3.2.2 Optimization statements

In many application areas, in order to solve a problem, computing a solution is not enough. One should also find the optimal solution, that is a solution with minimal or maximal cost. For supporting these applications, the language of weight

constraint rules has been extended by introducing two optimization statements, the minimize statement and the maximize statement. In these statements, cost functions are expressed as linear sums of the weights of literals. A minimizing statement m has the following form

$$\mathsf{minimize}\{A_1 = w_{A_1}, ..., A_n = w_{A_n}, \texttt{not}\, B_1 = w_{B_1}, ..., \texttt{not}\, B_m = w_{B_m}\} \quad (3.2)$$

and it means that a model M with the smallest weight

$$w(m, M) = \Sigma_{A_i \in M} w_{A_i} + \Sigma_{B_i \notin M} w_{B_i}$$

is required. In a constraint rule program P many minimize statements can occur and a stable model for P has to satisfy all of them. So, let $m_1, ..., m_n$ be the sequence of statements of the form (3.2) occurring in P. The ordering $\leq_P$ for stable models is obtained by defining that $M \leq_P M'$ holds if and only if $w(m_i, M) = w(m_i, M')$ for all $i = 1, ..., n$ or there exists some $j \leq n$ such that $w(m_j, M) < w(m_j, M')$ and $w(m_i, M) = w(m_i, M')$ for all $i = 1, ..., j - 1$. A stable model M for a program P is *optimal* if for any other stable model M' for P, $M \leq_P M'$.

3.2.3 Weight constraint rules with variables and functions

For many application problems it is needed to increase the expressiveness of weight constraint programs allowing weight constraint rules with variables and functions.

However, as proved in Chapter 2, nonground logic programs without restrictions are highly undecidable. But it is possible to impose syntactic restrictions on weight constraint rules in such a way that decidability is guaranteed. These restrictions impose a particular form on rules making a weight constraint program P *domain-restricted*, that is P can be divided in two parts: P_{Do} that defines the domain predicates in P and such that it has a unique finite effectively computable stable model, call it D, and P_{Ot} that contains all other rules and such that for each its rule R, a variable in R has to occur in a domain predicate in the body of R, then P has to be an ω-restricted program. These restrictions guarantee that P has the same stable model of P_D, where P_D is the subset of $\mathsf{ground}(P)$ containing all ground rules in which the instances of domain predicates are satisfied by D. Since P_D is finite then computing the stable models for P is decidable.

A compact way to write such weight constraints is given by *conditional liter-*

als. A conditional literal is of the form $L : d$ where L is a literal and the conditional part d is a domain predicate. A ground instantiation of a conditional literal includes all ground literals L' obtained applying to L all ground substitutions for $L : d$ that result in $L' : d'$ such that d' is in the unique stable model of P_{Do}.

3.3 CLP: Constraint Logic Programming

Constraint Logic Programming (CLP) is a declarative formalism that combines constraint programming and logic programming. A constraint logic program is in fact a set of logical normal rules where constraints are allowed in the body.

A signature Σ defines the set of predefined predicates and functional symbols and their arieties, a Σ-structure $\mathcal{D}$ is the domain of computation, that is the structure over which computation is to be performed. $\mathcal{D}$ defines a set D and a mapping of functions and relations in Σ on D that respects the arieties of those symbols.

Definition 3.3.1 *If $t_1, ..., t_n$ are terms built from variables and function symbols of Σ and $p \in \Sigma$ is a predicate symbol then $p(t_1, ..., t_n)$ is a* primitive constraint.

A constraint *is a first order formula built from primitive constraints.*

The class of Σ-formulas is the class of constraints definable on Σ and is denoted by $\mathcal{L}$.

Definition 3.3.2 *A* constraint domain *is a pair $(\mathcal{D}, \mathcal{L})$ where $\mathcal{D}$ is a Σ-structure and $\mathcal{L}$ is the class of Σ-formulas on a signature Σ.*

A CLP program is a set of rules as

$$A \leftarrow C_1, ..., C_n.$$

where A is an atom and, for each $1 \leq i \leq n$, C_i is an atom or a constraint.

As in constraint programming, also in constraint logic programming a problem is specified in terms of variables and constraints on these variables and a computation is an iterative process where at each step decisions, represented as constraints, are made on admitted values for some variables and then a *"constraint propagation"* is applied to propagate the consequences of these decisions. Then, during the computation new variables and constraints may be created. Moreover at each step, constraints are tested as a whole before the execution proceeds further. For better understanding these key features of the CLP methodology, consider the following example that shows how it computes a query in a way that is different from the answer set approach.

Example 3.3.3 (Jaffar [39].) *The program below defines the relation* $sumto(n, 1 + 2 + ... + n)$ *for natural numbers* n.

$$sumto(0, 0).$$
$$sumto(N, S) \leftarrow N >= 1, N <= S, sumto(N - 1, S - N).$$

The query $S <= 3, sumto(N, S)$ *gives rise to three answers* $(N = 0, S = 0)$, $(N = 1, S = 1)$, *and* $(N = 2, S = 3)$, *and terminates. The computation sequence of states for the third answer, for example, is*

$$S \leq 3, sumto(N, S).$$

$$S \leq 3, N = N_1, S = S_1, N_1 \geq 1, N_1 \leq S_1,$$
$$sumto(N_1 - 1, S_1 - N_1).$$

$$S \leq 3, N = N_1, S = S_1, N_1 \geq 1, N_1 \leq S_1,$$
$$N_1 - 1 = N_2, S_1 - N_1 = S_2, N_2 \geq 1, N_2 \leq S_2,$$
$$sumto(N_2 - 1, S_2 - N_2).$$

$$S \leq 3, N = N_1, S = S_1, N_1 \geq 1, N_1 \leq S_1,$$
$$N_1 - 1 = N_2, S_1 - N_1 = S_2, N_2 \geq 1, N_2 \leq S_2,$$
$$N_2 - 1 = 0, S_2 - N_2 = 0.$$

The constraints in the final state imply the answer $N = 2$, $S = 3$. *Termination is reasoned as follows. Any infinite computation must use only the second program rule for state transitions. This means that its first three states must be as shown above, and its fourth state must be*

$$S \leq 3, N = N_1, S = S_1, N_1 \geq 1, N_1 \leq S_1,$$
$$N_1 - 1 = N_2, S_1 - N_1 = S_2, N_2 \geq 1, N_2 \leq S_2,$$
$$N_2 - 1 = N_3, S_2 - N_2 = S_3, N_3 \geq 1, N_3 \leq S_3,$$
$$sumto(\ldots).$$

Note that this contains an unsatisfiable set of constraints, and in CLP, no further reduction is allowed.

As in logic programming, also in CLP a program is a declarative specification of a problem and does not define how to solve the problem. This task is in fact delegated, for logic programming, to inferential engine while, for constraint logic programming, to constraint solver that involves decision-making algorithms and

constraint propagation algorithms, but these algorithms do not require any grounding process, as the Example 3.3.3 has shown. This means that a constraint solver is able to reason on nonground programs. Moreover as the following example shows, the power of CLP stands in the globally evaluation of constraints.

Example 3.3.4 (Jaffar [39].)

$$add(0, N, N).$$
$$add(s(N), M, s(K)) \leftarrow add(N, M, K).$$

where natural numbers n are represented by $s(s(...(0)...))$ *with n occurrences of s. Clearly, the meaning of the predicate* $add(N, M, K)$ *coincides with the relation* $N + M = K$. *However, the query* $add(N, M, K), add(N, M, s(K))$, *which is clearly unsatisfiable, runs forever in a conventional logic programming system. The important point here is that a global test for the satisfiability of the two add constraints is not done by the underlying logic programming machinery.*

The example above wants to underline a weak point of logic programming with respect to CLP: the *add* predicate is not evaluated considering the results of the *add* constraints tested so far, and then after the evaluation of $add(N, M, K)$, the second subgoal of the query above is not tested taking into account that $N+M = K$.

3.3.1 Semantics for CLP

A declarative semantics of CLP programs over a constraint domain $(\mathcal{D}, \mathcal{L})$ interprets a rule

$$p(\vec{X}) \leftarrow C_1, ..., C_n.$$

as the first order logic formula

$$\forall \vec{X}, \vec{Y}. p(\vec{X}) \vee \neg C_1 \vee ... \vee \neg C_n$$

where $\vec{X} \cup \vec{Y}$ is the set of all variables in the rule.

Given a constraint domain $(\mathcal{D}, \mathcal{L})$, a *valuation* v is a mapping from variables to D, and the natural extension which maps terms to D and formulas to closed $\mathcal{L}^*$-formulas. A $\mathcal{D}$*-interpretation* of a formula is an interpretation of the formula with the same domain as $\mathcal{D}$ and the same interpretation for the symbols in Σ as $\mathcal{D}$, that is a $\mathcal{D}$-interpretation $\mathcal{B}_{\mathcal{D}} = \{p(\vec{d}) \,|\, \vec{d} \in D^k\}$.

Definition 3.3.5 *A* $\mathcal{D}$-model *of a closed formula is a* $\mathcal{D}$*-interpretation which is a model of the formula.*

If we denote by $lm(P, \mathcal{D})$ the least $\mathcal{D}$-model of a constraint program P (least under the subset ordering $\subseteq$), then a *solution* to a query G with respect to P is a valuation v such that $v(G) \subseteq lm(P, \mathcal{D})$.

Chapter 4

Finitary programs

4.1 Introduction

As it is proved in Chapter 2, reasoning on not variable-free logic programs with infinite ground instantiations is undecidable. In general, we have that normal logic programs are Π^1_1-complete. Moreover, as I have underlined in the introduction to this PhD thesis, state-of-the-art answer set solvers are able to reason only on ground programs and then they cannot manage programs with infinite domains and models.

This limitations have induced many researchers to define classes of programs, such as stratified programs, order-consistent programs [33], domain-restricted programs [67], *omega*-restricted programs [71], that imposing particular forms on rules guarantee the decidability of programs even if variables and functions occur.

Another line of research [13, 12, 15, 16] has led to introduce the class of *finitary programs*, a subclass of normal logic programs that generalizes acyclic programs admitting only a finite number of ground instances of odd-cycles. For finitary programs recursion is restricted so avoiding infinite recursive calls and the number of possible inconsistency sources (that is the number of odd-cycles) is to be finite. More precisely, requiring that in the ground instance of a finitary program each atom depends on finitely many atoms and there are only finitely many odd-cyclic atoms, it is possible to keep complexity under control. In fact, for finitary programs both consistency checking and ground skeptical consequences are decidable, while nonground skeptical and credulous consequences are r.e.-complete. These results prove not only that this class of programs makes possible

to reason on recursive data structures and infinite domains, such as lists, trees, XML/HTML documents, time, and so on, but, since it is r.e.-complete and then Turing equivalent, also prove that finitary programs are a very expressive fragment of normal logic programs. Moreover, for this class of programs a form of compactness holds, that is an unusual property for logic programs under the stable model semantics that are well-known do not enjoy this property.

4.2 Finitary programs

A finitary program P is a normal program for which two important conditions hold: for each atom A in $\mathsf{ground}(P)$ the set $\{B \mid A \geq B\}$ is finite, that is P is finitely recursive, and there are finitely many inconsistency sources, and then odd-cycles, in $\mathsf{ground}(P)$.

Definition 4.2.1 (Finitely recursive programs [13].) *A normal program P is* finitely recursive *if and only if each ground atom A depends on finitely many ground atoms.*

If P is a finitely recursive program then for each atom A in $\mathsf{ground}(P)$ the dependency graph of A (cf. Definition 2.2.3), that is the subgraph of $DG(P)$ including A and all vertexes reachable from A in $DG(P)$, is finite.

Example 4.2.2 *Consider the program P as the set of following rules:*

$$p(f(X)) \leftarrow p(X).$$
$$q(X) \leftarrow \texttt{not}\, q(X).$$
$$s(X) \leftarrow q(X).$$

Note that P is a finitely recursive program because each of its ground atoms depends on finitely many ground atoms. In fact, for each ground term t, $q(t)$ depends on $q(t)$, $s(t)$ depends on $q(t)$, while $p(f(t))$ depends on $p(t)$ if t is a constant, otherwise if $t = f^i(a)$ then $p(f^{i+1}(a))$ depends on $p(f^i(a))$, $p(f^{i-1}(a))$, ..., $p(a)$. Note that, in this case, the sequence terminates because the argument $f(X)$ of the head recursive predicate of the first rule in P decreases in the body recursive predicate.

Definition 4.2.3 (Finitary programs.) *We say a program P is* finitary *if the following conditions hold:*

1. *P is finitely recursive.*

2. *There are finitely many odd-cyclic atoms in the dependency graph of P.*

Note that all finitely recursive programs with no negative cycles such as positive finitely recursive programs or locally stratified finitely recursive programs are finitary.

Example 4.2.4 *Consider a new version of the program P of the previous example:*

$$p(f(X)) \leftarrow p(X).$$
$$q(a) \leftarrow \mathtt{not}\, q(a).$$
$$s(X) \leftarrow q(X).$$

P is again a finitely recursive program but it is also a finitary program because there exists only one odd-cyclic atom, $q(a)$. Note that in the Example 4.2.2 there were infinitely many odd-cycles, one for each ground instance of rule

$$q(X) \leftarrow \mathtt{not}\, q(X).$$

4.3 Properties of finitary programs

4.3.1 Relevant subprograms

According to well-known results, the stable model semantics does not enjoy relevance property and then for answering a query F it is needed to analyze not only the rules on which F depends but also all possible inconsistency sources, and then all odd-cycles.

In this section I report some results in [13] that define the relevant subprogram for a ground formula F with respect to a program P as the set of rules needed for answering F and, mainly, these results prove that for a finitary program this relevant subprogram is finite. This is the key for providing the compactness property and decidability of consistency checking for finitary programs, besides decidability and semidecidability of query answering.

Definition 4.3.1 (Kernel atoms, relevant universe and subprogram.) *A* kernel atom *for a normal program P and a ground formula F is either an odd-cyclic atom or an atom occurring in F (note that the kernel atoms are ground by definition). The set of kernel atoms for P and F is denoted by $K(P, F)$.*

The relevant universe *for P and F, denoted by $U(P,F)$, is the set of all ground atoms B such that some kernel atom for P and F depends on B. In symbols:*

$$U(P,F) = \{B \mid for\ some\ A \in K(P,F),\ A \geq B\}.$$

The relevant subprogram *for a ground formula F (with respect to program P), denoted by $R(P,F)$, is the set of all rules in* $\mathsf{ground}(P)$ *whose head belongs to $U(P,F)$:*

$$R(P,F) = \{R \mid R \in \mathsf{ground}(P)\ and\ head(R) \in U(P,F)\}.$$

The following proposition is an immediate consequence of definition of relevant subprogram.

Proposition 4.3.2 (Bonatti [13].) *For all ground programs P and all ground formulas F, $U(P,F)$ is a splitting set for P, and $R(P,F) = bot_{U(P,F)}(P)$.*

Intuitively, if $R(P,F) = bot_{U(P,F)}(P)$ then the consistency of P as well as the skeptical inference of F from P depend on $R(P,F)$.

Example 4.3.3 *Consider the program P as in Example 4.2.4:*

$$\begin{array}{l} p(f(X)) \leftarrow p(X). \\ q(a) \leftarrow \mathtt{not}\, q(a). \\ s(X) \leftarrow q(X). \end{array}$$

and let $F = p(f(a))$. There exists one odd-cyclic atom, $q(a)$, so

$$\begin{array}{l} K(P,F) = \{q(a), p(f(a))\}, \\ U(P,F) = \{q(a), p(f(a)), p(a)\}, \\ R(P,F) = \{q(a) \leftarrow \mathtt{not}\, q(a), p(f(a)) \leftarrow p(a)\}. \end{array}$$

As the following proposition proves, all relevant subprograms for a finitary program P are finite. In fact, for any ground formula F, $K(P,F)$ must be finite because $\mathsf{ground}(P)$ has finitely many odd-cycles and since P is finitely recursive then also $U(P,F)$ is finite.

Proposition 4.3.4 (Bonatti [13].) *If P is finitary then, for all ground formulas F, $U(P,F)$ and $R(P,F)$ are finite.*

The next results are among the most important results proved in [13] and show that the finite, relevant subprogram for a finitary program suffices for query answering. This means that we have not needed of the whole ground instance of a finitary program (that could be infinite) for answering to a query but we need of only a partial instantiation of the program.

Lemma 4.3.5 (Bonatti [13].) *For all ground formulas F and all finitely recursive programs P, $R(P, F)$ has a stable model M_F if and only if P has a stable model M such that $M \cap U(P, F) = M_F$.*

Theorem 4.3.6 (Bonatti [13].) *For all finitely recursive programs P and all ground formulas F,*

1. *P credulously entails F if and only if $R(P, F)$ does.*
2. *P skeptically entails F if and only if $R(P, F)$ does.*

This theorem confirms what Proposition 4.3.2 intuitively suggested.

If P has no odd-cycle then the relevance property holds for P. In fact, $U(P, F)$ contains only the atoms on which F *syntactically* depends.

4.3.2 Compactness and consistency checking

Logic formalisms where an infinite set of formulas is inconsistent if and only if it has an inconsistent finite subset, enjoy the *compactness* property. Normal logic programs, and nonmonotonic logics in general, do not enjoy this property.

Here I report an important result for finitary programs: a finitary program is consistent if and only if it has a finite *unstable kernel*, and then the compactness property holds for this class of programs. This result proves all good computational properties that hold for finitary programs.

Definition 4.3.7 *Given a normal program P, an* unstable kernel *for P is a set $K \subseteq \mathsf{ground}(P)$ with the following properties:*

1. *K is* downward closed*, that is, for each atom A occurring in K's rules, K contains all the rules $R \in \mathsf{ground}(P)$ such that $head(R) = A$.*
2. *K has no stable model.*

Proposition 4.3.8 *Let P be a normal program. Let*

$$K_o(P) = \{B \mid B \in \mathsf{ground}(P) \textit{ and } B \textit{ is odd} - \textit{cyclic}\},$$
$$U_o(P) = \{B \mid \textit{for some } A \in K_o(P),\ A \geq B\},$$
$$R_o(P) = \{R \mid R \in \mathsf{ground}(P) \textit{ and } head(R) \in U_o(P)\}.$$

A finitely recursive program P has no stable model if and only if $R_o(P)$ *has no stable model. Moreover, if P is finitary, then* $R_o(P)$ *is finite.*

Proof. Extend the language of P with a new propositional symbol q. Note that $R(P, q) = R_o(P)$, because q does not occur in P. The proposition follows immediately from Lemma 4.3.5 by setting $F = q$. Again, note that if $R_o(P)$ is inconsistent then it is an unstable kernel for P.

The second part follows from Proposition 4.3.4. ∎

Proposition 4.3.9 *If P is a finitary program then* $K_o(P)$*,* $U_o(P)$ *and* $R_o(P)$ *are r.e. sets.*

Proof. For all programs P, the set of rules in ground(P) is an r.e. set. In fact, it is possible to enumerate all ground substitution θ of P and then to enumerate all ground rules $R\theta$ in ground(P) such that R is a rule of P.

So, for each atom A in the rules of ground(P), it is possible to check if A is odd-cyclic visiting the dependency graph of A, that is finite because P is finitely recursive, and verifying if A occurs in any odd-cycle.

If $K_o(P)$ is an r.e. set then also $U_o(P)$ is an r.e. set because, for each atom $A \in K_o(P)$, its dependency graph is finite and all atoms occurring in this graph can be included in $U_o(P)$.

Again, if $U_o(P)$ is an r.e. set then it is possible to check, for each rule R in ground(P), if *head*(R) belongs to $U_o(P)$ and then also $R_o(P)$ is an r.e. set. ∎

Theorem 4.3.10 (Compactness.) *A finitary program P has no stable model if and only if it has a finite unstable kernel.*

Proof. This theorem follows immediately from Proposition 4.3.8. In fact P is inconsistent if and only if $R_o(P)$ is inconsistent and, if P is finitary, $R_o(P)$ is finite. Moreover, as I said in the proof of Proposition 4.3.8, if $R_o(P)$ is inconsistent then it is an unstable kernel for P. ∎

Corollary 4.3.11 *Let P be a finitary program. Given the set of odd-cyclic atoms occurring in P, deciding whether P is inconsistent is decidable.*

Proof. This corollary is an immediate consequence of Theorem 4.3.10. In fact, deciding whether a finite set of normal rules is inconsistent is decidable. ∎

4.3.3 Decidability and semi-decidability of inference

In this subsection I focus on the complexity of inference within the class of finitary programs and on its upper bounds. As proved in [16] for all ground goals, both credulous and skeptical inference are decidable.

Theorem 4.3.12 (Bonatti [16].) *For all finitary programs P and ground goals F, given the set of odd-cyclic atoms in P, both the problem of deciding whether F is a credulous consequence of P and the problem of deciding whether F is a skeptical consequence of P are decidable.*

From the previous theorem it follows that existentially quantified goals $\exists F$ are semidecidable.

Theorem 4.3.13 (Bonatti [16].) *For all finitary programs P and ground goals F, given the set of odd-cyclic atoms in P, both the problem of deciding whether $\exists F$ is a credulous consequence of P and the problem of deciding whether $\exists F$ is a skeptical consequence of P are semidecidable.*

The original version of these two theorems in [16] did not assume the set of odd-cyclic atoms in P as given. Here I have corrected this mistake. Note that we cannot compute the set of all odd-cyclic atoms occurring in P since we know that they are finitely many but we do not know how many they are. Indeed, if we assume that it is possible to compute all the odd-cycles in P then the problem of recognizing finitary programs would be at least semidecidable while, as proved in [16], the problem of checking condition (2) of Definition 4.2.3 is not semidecidable.

In Chapter 6, I shall prove some results on lower bounds to the complexity of inference for the class of finitary programs by relaxing the second condition of Definition 4.2.3, while, as proved in [16], if we relax the first condition of that definition we obtain the following proposition.

Proposition 4.3.14 ([16].) *Credulous and skeptical inference are not semidecidable for the class of all programs satisfying condition (1) in Definition 4.2.3.*

4.4 Handling local variables

If P is a normal logic program, P is not finitely recursive if and only if its dependency graph $DG(P)$ contains:

1. an infinite branching, or

2. an infinite path.

When an infinite branching occurs in a dependency graph of a normal logic program P there must exist in P a rule R with infinite different ground instances that have the same ground head. Then, in the body of R there must be variables whose instances are not bound by the ground instances of head. Such variables may only be local variables, that is variables that do not occur in the head. In [16] some conditions are given since local variables are admissible.

Let R be a rule in a logic program P with local variables. If the predicates in R where local variables occur are domain predicates so that they are defined by a locally stratified program that bounds local variables, it is possible replace these predicates in P with their partial evaluated version (without local variables) obtained by evaluating the locally stratified program that define them.

For example, consider the following program proposed in [16] where the first two rules are a definition of *member* relation:

$$
\begin{array}{l}
member(X, [X \mid Y]).\\
member(X, [Y \mid Z]) \leftarrow member(X, Z).\\
p(f(X)) \leftarrow member(Y, [a, b, c]), q(X, Y).
\end{array}
$$

Note that the rule $p(f(X)) \leftarrow member(Y, [a, b, c]), q(X, Y)$ with local variable Y can be equivalently replaced by following finitely recursive rules

$$
\begin{array}{l}
p(f(X)) \leftarrow q(X, a).\\
p(f(X)) \leftarrow q(X, b).\\
p(f(X)) \leftarrow q(X, c).
\end{array}
$$

The program so obtained has the same stable models of the original program.

According to results proved in [16], each node in the dependency graph of a normal program P without local variables has finitely many outgoing edges.

Local variables can also cause infinite paths occur in the dependency graph of P. Consider the following rule

$$p(X_1) \leftarrow p(X_2).$$

This rules defines a possible path of dependences

$$p(X_1) \rightarrow p(X_2) \rightarrow p(X_3) \rightarrow p(X_4) \rightarrow ...$$

that is a path in which infinite different variables occur and these variables could be instantiated on an infinite domain and so an infinite path of dependences could be generated.

4.5 Recognizing finitary programs

In [16] it has been proved that the class of finitary programs is undecidable. Actually, neither checking if a program is finitely recursive nor checking if a program has finitely many odd-cycles is decidable because it is possible to reduce to these problems a variant of the halting problem of a Turing machine with semi-infinite tape.

Theorem 4.5.1 (Bonatti [16].) *Checking whether a program is finitely recursive is not decidable.*

Theorem 4.5.2 (Bonatti [16].) *The problem of checking condition (2) in Definition 4.2.3 is not semidecidable.*

Even if finitary programs are not decidable, there exists an its subclass that can be recognized. A sound but incomplete prototype has been, in fact, implemented and it will be dealt with more details in Chapter 5.

Chapter 5

Some implementations

5.1 Introduction

In this chapter I report some implementations for working with logic programs under the stable model semantics.

In first two sections I discuss about two of the major state-of-the-art answer set solvers: Smodels and DLV. Even if these solvers have different characteristics, implement different tasks and reason on different classes of programs, both have to compute a ground instance of the input program, even if with some optimizations, before the actual reasoning can start.

Next I describe an implementation of a recognizer for finitary programs. Since this system is sound but not complete, it proves that it is effectively possible to work with a subset of finitary programs.

5.2 Smodels

Smodels [58] is an answer set solver that implements both well-founded and stable model semantics for range-restricted function-free normal logic programs extended with weight constraint rules. The reasoner has been implemented to compute the following tasks:

1. consistency checking;

2. computation of well-founded models and stable models;

3. credulous and skeptical entailment.

Moreover, Smodels is able to compute all stable models for a logic program or only a given number of stable models.

The system is composed of two modules:

1. lparse,
2. smodels.

lparse is the grounding module and represents the user front-end. In fact, it accepts the user program, that has to be written in the Smodels language, and computes the ground instance of the input program. The ground program elaborated by lparse is not really the whole ground instance of the input program but an its optimized subset such that the original program and the program produced by lparse have the same stable models. Intuitively, lparse drops all those rule instances that cannot be applied and then that cannot contribute to derive any atoms.

The grounded program is then passed to smodels module that is the actual reasoning engine of the Smodels system. It accepts the ground programs from lparse module and computes for it the stable or well-founded models as well as its credulous or skeptical consequences.

The implementation of the stable model semantics for ground programs is based on a novel technique where a bottom-up backtracking search with a powerful pruning method is employed. One of the advantages of this technique is that it can be implemented to work in linear space so ensuring that hard instances can be solved provided that adequate amount of running time is allocated.

The Figure 5.1 shows the basic steps followed by Smodels for computing the stable model of the simple program

$$a \leftarrow \mathtt{not}\, b.$$
$$b \leftarrow \mathtt{not}\, a.$$

that implements the *OR* function.

The Smodels system is implemented in C++ and it offers an API to programmers so that it is possible to make other systems communicate with Smodels. It is the case of CASP system, a logic programming reasoner for constrained programs described with more details in Chapter 8. CASP system, in fact, computes strong answer sets by interleaving Smodels with a constraint solver.

Figure 5.1: The way of a logic program in Smodels [70].

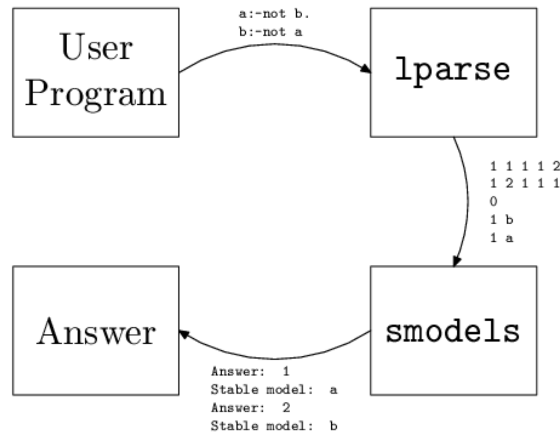
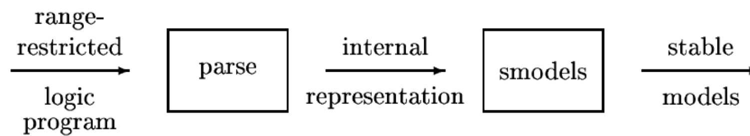


Figure 5.2: Overall architecture of Smodels [58].



5.2.1 Architecture

In Figure 5.2 a simplified scheme of the general architecture of the Smodels system is represented. As shown, the lparse module accepts the input program by user. lparse does not accept any logic program but only range-restricted function-free normal logic programs eventually extended with weight constraint rules. Obviously, these programs have to be written in the Smodels language. lparse module elaborate the input program and generate a simplified ground instance so that it has the same stable models of input program. This is done by dropping all those rules that cannot be applied because their bodies will never be true. In fact, lparse divides the predicates of input program in domain predicates, whose definition does not admit recursion through negation, and non-domain predicates. The subprogram of domain predicates is a stratified program and then it has only one

stable model. This means that all provable domain predicates are true in all stable models of input program. Since input program must be range-restricted then the provable domain predicates provide a ground instance for all variables in the input program. Then, lparse can compute a ground instance only of those rules that depend on provable domain predicates. For example, consider the following program as in [70]

$$
\begin{array}{l}
d(a). \\
e(b). \\
e(c). \\
foo(X) \leftarrow d(X), \texttt{not}\, bar(X). \\
bar(X) \leftarrow d(X), \texttt{not}\, foo(X).
\end{array}
$$

where d and e are domain predicates while foo and bar are not since they are defined using negative recursion. Its complete instantiation is:

$$
\begin{array}{l}
d(a). \\
e(b). \\
e(c). \\
foo(a) \leftarrow d(a), \texttt{not}\, bar(a). \\
foo(b) \leftarrow d(b), \texttt{not}\, bar(b). \\
foo(c) \leftarrow d(c), \texttt{not}\, bar(c). \\
bar(a) \leftarrow d(a), \texttt{not}\, foo(a). \\
bar(b) \leftarrow d(b), \texttt{not}\, foo(b). \\
bar(c) \leftarrow d(c), \texttt{not}\, foo(c).
\end{array}
$$

Since atoms $d(b)$ and $d(c)$ are not provable and any rule that depends on either of them cannot be applied, those rules can be dropped without losing any stable model. The optimized program is:

$$
\begin{array}{l}
d(a). \\
e(b). \\
e(c). \\
foo(a) \leftarrow d(a), \texttt{not}\, bar(a). \\
bar(a) \leftarrow d(a), \texttt{not}\, foo(a).
\end{array}
$$

At the last, lparse passes to smodels these ground rules translated into the language that it accepts. The smodels language includes the following four different rules:

- basic rules,

- constraint rules,
- choice rules,
- weight rules.

Basic rules are normal logic rules while constraint, choice and weight rules are as explained in Chapter 2.

smodels searches the stable models for the program received from lparse by a bottom-up algorithm. This algorithm exploits and prunes the search space approximating the admissible stable models by means of called *full sets*, that is the sets of atoms occurring negated in the program and whose positive atoms cannot occur in a stable model. In this way the search algorithm can also compute *focused model searches* and then it is able to compute those models containing or not containing a given set of atoms.

5.3 DLV

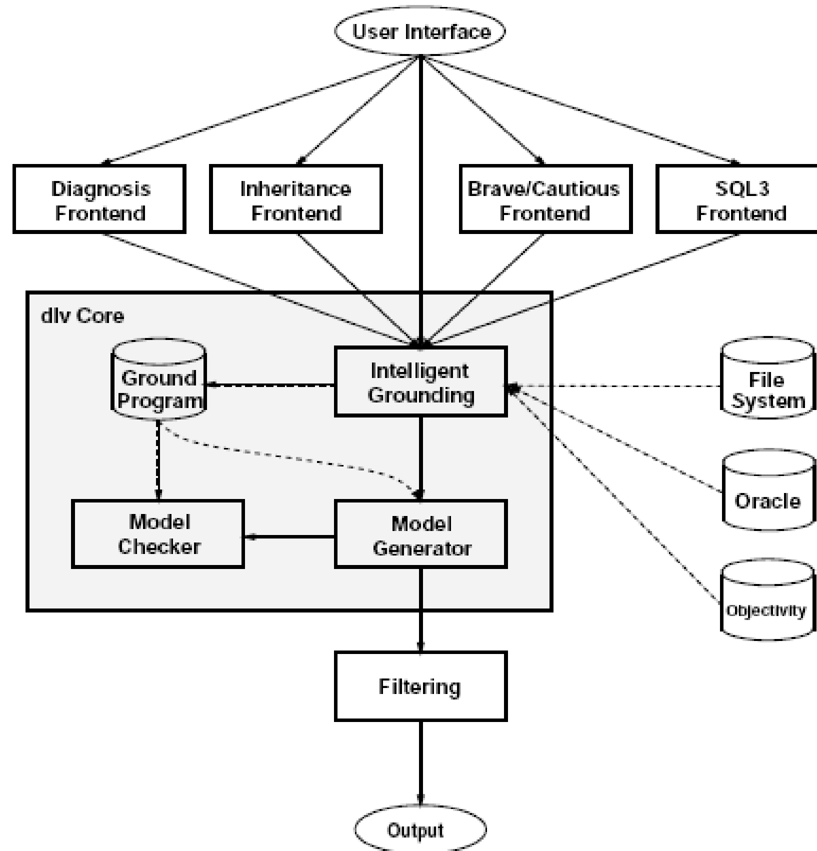
DLV [21, 28] is an answer set solver for disjunctive datalog extended with integrity constraints and classical negation under the answer set semantics [36]. Indeed, DLV support both default (or explicit) negation, denoted by $\neg$, and negation as failure, denoted by `not`. It is also possible to compose these two forms of negations and, for any atom A, `not` $\neg A$ is a legal literal while $\neg$`not` A is not.

The reasoner is able to compute the following tasks:

1. computation of answer sets,
2. credulous and skeptical entailment,
3. abductive diagnosis [61, 24, 42, 31] and consistency-based diagnosis [64],
4. SQL3 query computation,
5. planning analysis.

DLV system implements interfaces to classic relational database systems. This means that it can read input data not only from user logic programs but also from external databases as bases of ground facts.

Figure 5.3: Overall architecture of DLV [28].



5.3.1 Architecture

As shown in Figure 5.3, the architecture of DLV system involves a central *core*, that represents the central answer set solver engine, and a set of front-ends and interfaces towards user and other systems as file systems or relational databases.

The input program that the DLV core has to elaborate contains rules from a user program or a program stored on file system as well as ground facts from relational tables in a database. The DLV core reads the input program, that might have been preprocessed by a front-end, evaluates it and returns an, eventually postprocessed, answer to user.

The DLV core contains three main subsystems:

- the grounding module,
- the model generator, and
- the model checker.

First the grounding module computes an optimized subprogram of the ground instance of the input program that is equivalent to the original program in the sense that they have the same answer sets, and then the model generator and the model checker, that are the actual solver engine of DLV system, evaluate the ground program and solve the required task.

The evaluation process adopts an iterative *"Guess&Check"* methodology: at each step a model candidate is identified and then checked if it is a real solution for the input program, until no more models are to be computed or a given number of answer sets has been returned to user. More precisely, the input program is divided in two parts: a set of all disjunctive rules that define the search space of possible answer sets for the input program, and a set of rules, as integrity constraints, that cannot prove any atom but only drop those models that do not satisfy them. So, the first part is evaluated by the model generator to search a possible answer set candidate, while the second part is evaluated by the model checker for testing if the candidate model computed by the model generator is really an answer set for the input program.

5.4 A prototype recognizer for finitary programs

At the LPNMR'01 conference in Vienna [12] a sound but incomplete prototype recognizer, implemented in XSB Prolog, was presented so proving that there exists a decidable subclass of finitary programs.

For checking whether a given program is finitely recursive (condition (1) of Definition 4.2.3), the idea is to analyze the recursion patterns of the input program making sure that the *norm* of arguments (a measure of term size [26]) does not increase indefinitely. This analysis is also useful for identifying the potential cycles with an odd number of negative edges.

For all predicate symbols p, let a p-atom be an atom whose predicate is p. Similarly, let a p-literal be a literal whose predicate is p.

Definition 5.4.1 *Let t be a (possibly nonground) term. The* norm *of t, denoted by $|t|$, is the number of variables and function symbols occurrences in t (constants are regarded as* 0*-ary functions).*

Let $\vec{t} = t_1, ..., t_n$ be a term sequence. The norm of $\vec{t}$, denoted by $|\vec{t}|$, is $|t_1, ..., t_n| = |t_1| + ... + |t_n|$.

For all vectors of terms $\vec{t}$ and $\vec{u}$, following comparison relations can be defined:

- $\vec{t} \prec \vec{u}$ if and only if for all grounding substitution σ, $|\vec{t}\sigma| < |\vec{u}\sigma|$;
- $\vec{t} \preccurlyeq \vec{u}$ if and only if for all grounding substitution σ, $|\vec{t}\sigma| \leq |\vec{u}\sigma|$;
- $\vec{t} \precsim \vec{u}$ if and only if $\vec{t}$ is *almost never larger than* $\vec{u}$, that is, there exist only finitely many (possibly no) grounding substitutions σ such that $|\vec{t}\sigma| > |\vec{u}\sigma|$.

Note that $\vec{t} \prec \vec{u} \Rightarrow \vec{t} \preccurlyeq \vec{u}$ and $\vec{t} \preccurlyeq \vec{u} \Rightarrow \vec{t} \precsim \vec{u}$. Moreover, the norm over term sequences and the three comparison relations are insensitive to permutations. More precisely, for all permutations $\vec{t_1}$ of $\vec{t}$, $|\vec{t_1}| = |\vec{t}|$; therefore if $\lessdot$ is any of the relations $\preccurlyeq$, $\prec$ and $\precsim$ then for all $\vec{u}$, $\vec{t_1} \lessdot \vec{u} \Leftrightarrow \vec{t} \lessdot \vec{u}$ and $\vec{u} \lessdot \vec{t_1} \Leftrightarrow \vec{u} \lessdot \vec{t}$.

It is possible to relate the term comparison relations to restricted sets of substitutions. For all sets of grounding substitutions Σ let

- $\Sigma \Vdash \vec{t} \prec \vec{u}$ if and only if for all $\sigma \in \Sigma$, $|\vec{t}\sigma| < |\vec{u}\sigma|$;
- $\Sigma \Vdash \vec{t} \preccurlyeq \vec{u}$ if and only if for all $\sigma \in \Sigma$, $|\vec{t}\sigma| \leq |\vec{u}\sigma|$;
- $\Sigma \Vdash \vec{t} \precsim \vec{u}$ if and only if there exist only finitely many (possibly no) $\sigma \in \Sigma$ such that $|\vec{t}\sigma| > |\vec{u}\sigma|$.

This will be helpful in order to capture some semantic information, e.g. by setting Σ to the set $\mathsf{Ans}(G, P)$ of all grounding substitutions σ such that goal $G\sigma$ is true in the least model of P.

Note that $\Vdash$ is *inverse monotonic* with respect to Σ, that is, for all term comparison relations $\lessdot$, if $\Sigma \Vdash \vec{t} \lessdot \vec{u}$ and $\Sigma' \subseteq \Sigma$ then $\Sigma' \Vdash \vec{t} \lessdot \vec{u}$.

Next from the term comparison relations, the predicate arguments comparison relations can be formalized. For example, $1 \prec_q 2$ means that in the least model of P, the first argument of any q-atom is always smaller than the second argument.

Definition 5.4.2 *A* $n2k$-projection index *is a sequence of distinct integers $\vec{a} = a_1, ..., a_k$ such that $1 \leq a_i \leq n$ $(1 \leq i \leq k)$.*

A n-projection index *is any $n2k$-projection index.*

If $\vec{a}$ is a $n2k$-projection index and $\vec{t} = t_1, ..., t_n$ is a term sequence, then $\vec{t}[\vec{a}] = t_{a_1}, ..., t_{a_k}$. Similarly for all atoms $A = p(t_1, ..., t_n)$, $A[\vec{a}] = t_{a_1}, ..., t_{a_k}$.

Moreover, $-\vec{a}$ denotes the *complement* of a n-projection index $\vec{a}$ and is the ordered sequence of integers between 1 and n that do not occur in $\vec{a}$.

Let $\lessdot$ range over $\prec$, $\preccurlyeq$ and $\precsim$, and let q be a n-ary predicate of a program P:

- $P \Vdash \vec{a} \lessdot_q \vec{b}$ if and only if, for any sequence $\vec{x} = x_1, ..., x_n$ of distinct variables, $\mathsf{Ans}(q(\vec{x}), P) \Vdash \vec{x}[\vec{a}] \lessdot \vec{x}[\vec{b}]$.

Intuitively, these above notions are useful for testing if the rules in a program P generate a dependency graph $DG(P)$ where the size of arguments in the predicates does not increase along its paths or where ground substitutions for some arguments in a rule bound the admissible values for the other nonground terms. Then it is possible to use these notions for a recursion and cycle analysis and then for testing if the dependency graph of a program P might contain an infinite path or an infinite branching, so testing the first condition in Definition 4.2.3 of finitary programs, or if this dependency graph might contain cycles with an odd number of negative edges, so testing the second condition in Definition 4.2.3.

In this respect, the finitary program recognition techniques differ from the techniques for verifying termination. The latter require some arguments to decrease at each recursive call, while in our case some infinite loops are allowed.

The finitary program recognizer presented in [12] consists of four stages where the first three phases aim to test if the input program is finitely recursive while the last phase checks the second condition of finitary programs:

Interargument analysis: the relationships between the size of terms occurring as predicate arguments in a rule of the input program are analyzed. In particular, the size of head arguments with respect to the size of body arguments is checked for exploiting if the head arguments in a rule bound the local variables in the body or if the head recursive predicate can occur in the body on arguments with a bigger size (note that this could generate infinite sequences of recursive calls without repeats).

Recursion analysis: starting from the results of the interargument analysis, the recognizer finds, for each predicate symbol, the group of its arguments whose size decreases (or at least does not increase "too mach") at each recursive call and so analyzes cyclic atom dependencies classifying them in *good recursion cycles* and *false cycles*. Intuitively, a program where all cyclic dependencies are good recursion cycles or false cycles, is finitely recursive.

Recursive domain predicate identification: negative cycles classified as false cycles during the previous phase, are analyzed for identifying domain predicates that can be used to compute an optimized partial evaluation of the input program. Since the subprogram of all domain predicates has exactly one stable model which is contained in every stable model of the entire program (domain predicates constitute the bottom program of a splitting set of the given program), such model can be used to simplify the ground program instantiation by considering only rule instances whose domain subgoals are true. Depending on its recursion properties (such as the existence or lack of positive cycles) a domain predicate may be evaluated in a naive top-down fashion, in a bottom-up fashion, or in a tabled fashion.

A similar optimized grounding procedure is implemented by the lparse module of Smodels system (see Section 5.2), but the finitary recognizer evaluates all locally stratified recursive domain predicates, while lparse evaluates only stratified domain predicates.

Cycle analysis: during this phase all potential cycles through an odd number of negations are analyzed. If any potential odd-cycle is ground or is a false cycle then the number of odd-cyclic atoms is finite. For example, program

$$p(a) \leftarrow \texttt{not}\, p(a).$$

is accepted because this odd-cycle is ground, while

$$p(f(X)) \leftarrow \texttt{not}\, p(f(X)).$$

is rejected.

Also the program [12]

$$\begin{array}{l} even(0). \\ even(s(X)) \leftarrow \texttt{not}\, even(X). \end{array}$$

is accepted. In fact in this program, the argument in the body recursive predicate of the second rule is strictly smaller than the head argument and then these rules define a false cycle, that is actually they do not generate any cycle in the dependency graph of the program. Moreover, potential odd-cycles belonging to ω-restricted programs again are accepted since any their ground instance contains a finite number of odd-cycles (cf. Section 2.7).

Part II

New proposals

Chapter 6

Finitely recursive programs

6.1 Introduction

The class of *finitary programs* has been introduced in Chapter 4 as a subclass of logic programs interpreted according to the stable model semantics, and admitting function symbols (therefore data constructors and infinite domains). In order to make the main reasoning tasks decidable or at least semidecidable, finitary programs are required to satisfy two restrictions: the dependency graph of each ground atom and the number of odd-cyclic atoms must be finite.

In particular, in [16] it was proved that for finitary programs, consistency checking is decidable as well as credulous reasoning and skeptical reasoning on ground queries, while nonground queries were proved to be r.e.-complete. Moreover, a form of compactness holds: an inconsistent finitary program has always a finite *unstable kernel*, i.e. a finite subset of the program's ground instantiation with no stable models. All of these properties are quite unusual for a nonmonotonic logic.

In [8] we extended these good properties to larger program classes. Two obvious candidate classes are obtained by dropping one of the two restrictions defining finitary programs. It has already been noted in [16] that by dropping the first condition on recursion, one obtains a superclass of locally stratified programs whose complexity is then far beyond computability. If the second condition on odd-cyclic atoms is dropped then queries are not decidable anymore [16], but we gave a precise characterization of their complexity and a detailed analysis of the effects that dropping the second restriction has on the other properties of finitary programs.

The programs that satisfy only the first restriction are called *finitely recursive*

programs. They cover a wide range of practically interesting programs. For example, most standard list manipulation programs (member, append, remove etc.) are finitely recursive. The reader can find numerous examples of finitely recursive programs in [16]. Many interesting programs are finitely recursive but not finitary, due to integrity constraints that apply to infinitely many individuals.

Example 6.1.1 *Figure 4 of [16] illustrates a finitary program for reasoning about actions, defining—among others—two predicates* $\mathtt{holds}(fluent, time)$ *and* $\mathtt{do}(action, time)$*. The simplest way to add a constraint that forbids any parallel execution of two incompatible actions* a_1 *and* a_2 *is including a rule* $f \leftarrow \mathtt{not}\, f$, $\mathtt{do}(a_1, T)$, $\mathtt{do}(a_2, T)$ *in that program, where* f *is a fresh propositional symbol (often such rules are equivalently expressed as* denials $\leftarrow \mathtt{do}(a_1, T)$, $\mathtt{do}(a_2, T)$*). This program is not finitary (because* f *depends on infinitely many atoms since* T *has an infinite range of values) but it can be reformulated as a finitely recursive program by replacing the above rule with*

$$f(T) \leftarrow \mathtt{not}\, f(T), \mathtt{do}(a_1, T), \mathtt{do}(a_2, T)\,.$$

Note that the new program is finitely recursive but not finitary, because the new rule introduces infinitely many odd cycles (one for each instance of $f(T)$*).*

If P is finitely recursive then, for each atom A occurring in $\mathsf{ground}(P)$, the dependency graph of A is a finite graph. Moreover, if A is an odd-cyclic atom then its dependency graph contains all the odd-cycles in which A occurs. So, the set of atoms on which A depends and the set of odd-cycles in which A occurs are finite.

We proved that for finitely recursive programs the compactness property still holds, and inconsistency checking and skeptical reasoning are semidecidable. Then, the restriction on recursion makes logic programs more similar to classical logics, while the restriction on odd-cycles (i.e. on the number of potential inconsistency sources) makes ground queries decidable and brings the complexity of nonground credulous queries within r.e. . Moreover, we extended the completeness of skeptical resolution [14, 16] from finitary programs to all finitely recursive programs. These results clarified the role that each of the two restrictions defining finitary programs has in ensuring their properties.

In order to prove these results program splittings [45] was used, but the focus was shifted from splitting sequences (whose elements are sublanguages) to the corresponding sequences of subprograms. For this purpose the notion of *module sequence* was introduced. It turns out that finitely recursive programs are exactly those programs whose module sequences are made of finite elements. Moreover

a finitely recursive program P has a stable model if and only if each element P_i of the sequence has a stable model, a condition which is not valid in general for normal programs.

6.2 Module sequences and a normal form for splitting sequences

The stable models [35, 36] of a normal program can be obtained by splitting a program into two modules—of which one is self-contained, while the other module depends on the former—and then combining the stable models of the two modules.

The splitting theorem has been extended to transfinite sequences in [45].

Definition 6.2.1 *A* (transfinite) sequence *is a family whose index set is an initial segment of ordinals,* $\{\alpha \ : \ \alpha < \mu\}$*. The ordinal* μ *is the* length *of the sequence.*

A sequence $\langle U_\alpha \rangle_{\alpha<\mu}$ *of sets is* monotone *if* $U_\alpha \subseteq U_\beta$ *whenever* $\alpha < \beta$*, and* continuous *if, for each limit ordinal* $\alpha < \mu$*,* $U_\alpha = \bigcup_{\nu<\alpha} U_\nu$*.*

Definition 6.2.2 (Lifschitz-Turner, [45]) *A* splitting sequence *for a program* P *is a monotone, continuous sequence* $\langle U_\alpha \rangle_{\alpha<\mu}$ *of splitting sets for* P *such that* $\bigcup_{\alpha<\mu} U_\alpha = \mathsf{atoms}(P)$.

In [45] Lifschitz and Turner generalized the splitting theorem to splitting sequences. They proved that each stable model M of P equals the infinite union of a sequence of models $\langle M_\alpha \rangle_{\alpha<\mu}$ such that (i) M_0 is a stable model of $bot_{U_0}(P)$, (ii) for all successor ordinals $\alpha < \mu$, M_α is a stable model of $e_{U_{\alpha-1}}(bot_{U_\alpha}(P) \setminus bot_{U_{\alpha-1}}(P), \bigcup_{\beta<\alpha} M_\beta)$, and (iii) for all limit ordinals $\lambda < \mu$, $M_\lambda = \emptyset$. They proved also that each sequence of models with these properties yields a stable model of P.

In this study on the computational properties of finitely recursive programs, the notion of splitting sequence will replaced with suitable sequences of bottom programs whose length is bounded by ω.

Definition 6.2.3 (GH, Module sequence) *Let* P *be a normal program and let the set of its* ground heads *be*

$$GH = \{ A \ | \ A = head(R), \ R \in \mathsf{ground}(P) \}.$$

The module sequence $P_1, P_2, P_3, ..., P_n, ...$ induced by an enumeration $A_1, A_2, A_3, ..., A_n, ...$ of *GH is defined as follows:*

$$\begin{array}{rcl} P_1 & = & \{ R \in \mathsf{ground}(P) \mid A_1 \textit{ depends on } \textit{head}(R) \} \\ P_{i+1} & = & P_i \cup \{ R \in \mathsf{ground}(P) \mid A_{i+1} \textit{ depends on } \textit{head}(R) \} \quad (i \geq 1). \end{array}$$

Example 6.2.4 *Consider the finitely recursive program P [16]:*

$$\begin{array}{l} p(f(X)) \leftarrow p(X), q(X). \\ q(X) \leftarrow s(X). \\ u(X) \leftarrow \mathtt{not}\, u(X). \\ z(X) \leftarrow p(X). \end{array}$$

The Herbrand Universe of P is $\{a, f(a), f(f(a)), ...\}$. *One enumeration of the GH is*

$$\begin{array}{rl} e = & \{p(f(a)), q(a), u(a), z(a), \\ & p(f(f(a))), q(f(a)), u(f(a)), z(f(a)), \\ & p(f(f(f(a)))), ...\}. \end{array}$$

A module sequence for *P* induced by the enumeration *e* of *GH is*

$$\begin{array}{l} P_1 = \{p(f(a)) \leftarrow p(a), q(a); q(a) \leftarrow s(a)\}; \\ P_2 = P_1; \\ P_3 = P_2 \cup \{u(a) \leftarrow \mathtt{not}\, u(a)\}; \\ P_4 = P_3 \cup \{z(a) \leftarrow p(a)\}; \\ P_5 = P_4 \cup \{p(f(f(a))) \leftarrow p(f(a)), q(f(a)); q(f(a)) \leftarrow s(f(a))\}; \\ P_6 = P_5; \\ P_7 = P_6 \cup \{u(f(a)) \leftarrow \mathtt{not}\, u(f(a))\}; \\ P_8 = P_7 \cup \{z(f(a)) \leftarrow p(f(a))\}; \\ \vdots \end{array}$$

Of course, those properties of module sequences which are independent of the enumeration of GH are interesting.

The following proposition follows easily from the definitions. A ground subprogram $P' \subseteq \mathsf{ground}(P)$ is *downward closed*, if for each atom A occurring in P', P' contains all the rules $R \in \mathsf{ground}(P)$ such that $A = \textit{head}(R)$.

Proposition 6.2.5 *Let P be a normal program. For all module sequences* $P_1, P_2, ...,$ *for P:*

1. $\bigcup_{i \geq 1} P_i = \mathsf{ground}(P)$,

2. *for each* $i \geq 1$ *and* $j \geq i$, $\mathsf{atoms}(P_i)$ *is a splitting set of* P_j *and* $P_i = bot_{\mathsf{atoms}(P_i)}(P_j)$,

3. *for each* $i \geq 1$, $\mathsf{atoms}(P_i)$ *is a splitting set of* P *and* $P_i = bot_{\mathsf{atoms}(P_i)}(P)$,

4. *for each* $i \geq 1$, P_i *is downward closed.*

Therefore, we immediately see that each module sequence for P consists of the bottom programs corresponding to a particular splitting sequence $\langle \mathsf{atoms}(P_i) \rangle_{i<\omega}$ that depends on the underlying enumeration of GH. Roughly speaking, such sequences constitute a *normal form* for splitting sequences. If P is finitely recursive, then "normal form" sequences can be required to satisfy an additional property:

Definition 6.2.6 (Smoothness) *A transfinite sequence of sets* $\langle X_\alpha \rangle_{\alpha<\mu}$ *is* smooth *if and only if* X_0 *is finite and for each non-limit ordinal* $\alpha + 1 < \mu$, *the difference* $X_{\alpha+1} \setminus X_\alpha$ *is finite.*

Note that when $\mu = \omega$ (as in module sequences), smoothness implies that each X_α in the sequence is finite. Finitely recursive programs are completely characterized by smooth module sequences:

Theorem 6.2.7 *The following are equivalent:*

1. *P is finitely recursive;*

2. *P has a smooth module sequence (where each* P_i *is finite);*

3. *all module sequences for P are smooth.*

Proof. Take any module sequence S of P. The fact that each P_i is finite is equivalent to say that P_1 is finite and, for each $i > 0$, $P_{i+1} \setminus P_i$ is finite. This is equivalent to say that each A_k depends on a finite number of ground atoms of P, i.e. that P is finitely recursive. ∎

Since smooth module sequences clearly correspond to smooth splitting sequences, the above theorem implies that by working with module sequences we are implicitly restricting our attention to *smooth splitting sequences of length* ω. Then the characterization of finitely recursive programs can be completed as follows, using standard splitting sequences:

Corollary 6.2.8 *For all programs P, the following are equivalent:*

1. *P is finitely recursive;*
2. *P has a smooth splitting sequence with length ω.*

Proof. If P is finitely recursive then each module sequence $P_1, P_2, ...$ for P is such that each $\mathsf{atoms}(P_i)$ is finite and then $\langle \mathsf{atoms}(P_\alpha) \rangle_{\alpha<\omega}$ is a smooth splitting sequence for P.

Now, suppose that $\langle U_\alpha \rangle_{\alpha<\mu}$ is a smooth splitting sequence for P but P is not finitely recursive. Then there exists a ground atom A_k such that there is an infinite chain of ground atoms $A_k \leq B_1 \leq B_2 \leq \dots$. Let α_1 be the least ordinal such that A_k belongs to U_{α_1}. Since $A_k \in U_{\alpha_1}$ then the infinite set $\{B_1, B_2, ...\}$ is included in U_{α_1} and U_{α_1} is infinite. Note that, since α_1 is the least such that A_k belongs to U_{α_1}, then by definition of splitting sequence, α_1 is not a limit ordinal. So, the difference $U_{\alpha_1} \setminus U_{\alpha_1-1}$ is finite and then U_{α_1-1} must be infinite. Moreover, $A_k \in U_{\alpha_1} \setminus U_{\alpha_1-1}$. Let B_i be the first atom such that $B_i \notin U_{\alpha_1} \setminus U_{\alpha_1-1}$ and let α_2 be the least ordinal such that $B_i \in U_{\alpha_2}$. Note that $\alpha_2 < \alpha_1$. Then, again, α_2 is not a limit ordinal and the difference $U_{\alpha_2} \setminus U_{\alpha_2-1}$ is finite while U_{α_2-1} is infinite because B_i depends on infinitely many ground atoms. If we iterate this procedure, we obtain a strictly descendant sequence of not limit ordinal, $\alpha_1, \alpha_2, ...$, such that U_{α_i} is infinite while $U_{\alpha_i} \setminus U_{\alpha_i-1}$ is finite and then U_{α_i-1} must be infinite. Since 0 is the lower bound of this sequence, then also U_0 must be infinite and this is a contradiction. ∎

Next I illustrate how module sequences provide an incremental characterization of the stable models of normal logic programs.

Note 6.2.9 *By Proposition 6.2.5 and the splitting theorem, if P is a normal program and $P_1, P_2, ..., P_n, ...$ is a module sequence for P, then for all $j \geq i \geq 1$ and for all stable models M_j of P_j, the set $M_i = M_j \cap \mathsf{atoms}(P_i)$ is a stable model of P_i. Similarly, for each stable model M of P, the set $M_i = M \cap \mathsf{atoms}(P_i)$ is a stable model of P_i.*

Roughly speaking, the following theorem rephrases the splitting sequence theorem of [45] in terms of module sequences. The original splitting sequence theorem applies to sequences of disjoint program "slices", while our theorem applies to monotonically increasing program sequences. Since no direct proof of the splitting sequence theorem was ever published (only the proof of a more general result for default logic was published [72]), here I report a direct proof that we gave of our result in [8].

Theorem 6.2.10 (Module sequence theorem) *Let P be a normal program and $P_1, P_2, ...$ be a module sequence for P. Then M is a stable model of P if and only if there exists a sequence $M_1, M_2, ...$ such that :*

1. *for each $i \geq 1$, M_i is a stable model of P_i,*
2. *for each $i \geq 1$, $M_i = M_{i+1} \cap \mathsf{atoms}(P_i)$,*
3. *$M = \bigcup_{i\geq 1} M_i$.*

Proof. Let M be a stable model of P. Since $P_1, P_2, ...$ is a module sequence for P then for each $i \geq 1$, $\mathsf{atoms}(P_i)$ is a splitting set of P and $P_i = bot_{\mathsf{atoms}(P_i)}(P)$. So, we consider the sequence $M_1 = M \cap \mathsf{atoms}(P_1), M_2 = M \cap \mathsf{atoms}(P_2), ...$ where, by the splitting theorem [46], for each $i \geq 1$, M_i is a stable model of P_i, $M_{i+1} \cap \mathsf{atoms}(P_i) = M_i$ and, by definition of $M_1, M_2, ...$ and by property 1 of Proposition 6.2.5, $\bigcup_i M_i = M$. Then for each stable model M of P there exists a sequence of finite sets of ground atoms that satisfies the properties 1, 2 and 3.

Now, suppose P ground and suppose that there exists a sequence $M_1, M_2, ...$ that satisfies the properties 1, 2 and 3. It is only to prove that the set $M = \bigcup_{i\geq 1} M_i$ is a stable model of P; equivalently,

$$\bigcup_{i\geq 1} M_i = \mathsf{lm}(P^M).$$

First $\bigcup_{i\geq 1} M_i \subseteq \mathsf{lm}(P^M)$ will be proved. Property 2 implies that for all $i \geq 1$, $(M \cap \mathsf{atoms}(P_i)) = M_i$; consequently $P_i^M = P_i^{M_i}$. Moreover, since $P_i \subseteq P$, then $P_i^M \subseteq P^M$, and hence

$$P_i^{M_i} = P_i^M \subseteq P^M.$$

By the monotonicity of $\mathsf{lm}(.)$ [2, 73], $\forall i \geq 1. \mathsf{lm}(P_i^{M_i}) \subseteq \mathsf{lm}(P^M)$. Moreover, M_i is a stable model of P_i and then $M_i = \mathsf{lm}(P_i^{M_i})$, so

$$\forall i \geq 1. M_i \subseteq \mathsf{lm}(P^M).$$

Now, it is left to prove the opposite inclusion, that is $\mathsf{lm}(P^M) \subseteq \bigcup_{i\geq 1} M_i$. Suppose $A \in \mathsf{lm}(P^M)$. From $P_i^M = P_i^{M_i}$ and $\bigcup_i P_i = P$ it follows that $\bigcup_i P_i^{M_i} = \bigcup_i P_i^M = P^M$. Moreover, by definition, for each $i \geq 1$, P_i is downward closed, then there must be a k such that $A \in \mathsf{lm}(P_k^{M_k})$. Hence, $A \in M_k$ and then $A \in \bigcup_i M_i$. ∎

The module sequence theorem suggests a relationship between the consistency of a program P and the consistency of each step in P's module sequences.

Definition 6.2.11 *A module sequence $P_1, P_2, ...$ for a normal program P is* inconsistent *if there exists an $i < \omega$ such that P_i has no stable model,* consistent *otherwise.*

Proposition 6.2.12 *If a normal program P has an inconsistent module sequence then P is inconsistent.*

Proof. Suppose that P has an inconsistent module sequence $P_1, P_2, ...$. Then there exists a P_i that has no stable models. Hence, P has an inconsistent bottom set and then P is inconsistent by the splitting theorem. ∎

Next the inconsistency of a module sequence S will be proved to be invariant with respect to the enumeration of GH inducing S.

Theorem 6.2.13 *Let $S = P_1, P_2, ...$ be a module sequence for a normal program P. If S is inconsistent then each module sequence for P is inconsistent.*

Proof. Let $S = P_1, P_2, ...$ be an inconsistent module sequence for P induced by the enumeration $A_1, A_2, ...$ of GH and let i be the least index such that P_i is inconsistent. Let $S' = P'_1, P'_2, ...$ be any module sequence for P induced by the enumeration $A'_1, A'_2, ...$ of GH. Since i is finite, there exists a finite k such that $\{A_1, A_2, ..., A_i\} \subseteq \{A'_1, A'_2, ..., A'_k\}$. So, by construction, $P_i \subseteq P'_k$ and then $\mathsf{atoms}(P_i) \subseteq \mathsf{atoms}(P'_k)$. Moreover, by definition, P_i is downward closed and then $P_i = bot_{\mathsf{atoms}(P_i)}(P'_k)$. Since P_i is inconsistent then P'_k is inconsistent (by the splitting theorem) and then also S' is inconsistent. ∎

In other words, for a given program P, either all module sequences are inconsistent, or they are all consistent. In particular, if P is consistent, then every member P_i of any module sequence for P must be consistent.

It may be tempting to assume that the converse holds, that is, if a module sequence for P is consistent, then P is consistent, too. Unfortunately, this statement is not valid, in general, as the following example shows.

Example 6.2.14 *Consider the following program P_f (due to Fages [33]):*

$$
\begin{array}{l}
q(X) \leftarrow q(f(X)). \\
q(X) \leftarrow \texttt{not}\, q(f(X)). \\
r(0).
\end{array}
$$

Note that P_f is not finitely recursive because, for each grounding substitution σ, $q(X)\sigma$ depends on the infinite set of ground atoms $\{ q(f(X))\sigma,\ q(f(f(X)))\sigma,\ \ldots\}$.

The first two rules in P_f are classically equivalent to

$$q(X) \leftarrow [q(f(X)) \vee \mathtt{not}\, q(f(X))]\,.$$

Since the body is a tautology and the stable models of a program are also classical models of the program, a stable model of P_f should satisfy all ground instances of $q(X)$. However, the Gelfond-Lifschitz transformation with respect to such a model would contain only the first and the third rules of the program, and hence the least model of the transformation would contain no instance of $q(X)$. It follows that P_f is inconsistent (it has no stable models). Now consider the following extension P of P_f:

1. $q(X) \leftarrow q(f(X)), p(X).$
2. $q(X) \leftarrow \mathtt{not}\, q(f(X)), p(X).$
3. $r(0).$
4. $p(X) \leftarrow \mathtt{not}\, p'(X).$
5. $p'(X) \leftarrow \mathtt{not}\, p(X).$
6. $c(X) \leftarrow \mathtt{not}\, c(X), \mathtt{not}\, p(X).$

To see that P is inconsistent, suppose M is a stable model of P. By rules 4 and 5, each ground instance of $p(X)$ can be either true or false. But each ground instance of $c(X)$ is odd-cyclic, therefore if $p(X)$ is false then rule 6 produces an inconsistency. It follows that all ground instances of $p(X)$ must be true in M. But, in this case, the rules 1, 2 and 3 are equivalent to program P_f and prevent M from being a stable model, as explained above. So P is inconsistent.

Next, consider the enumeration $e = \{r(0),\ q(0),\ p(0),\ p'(0),\ c(0),\ q(f(0)),\ p(f(0)),\ p'(f(0)),\ c(f(0)),\ \ldots\}$ of the set GH. This enumeration induces the following module sequence for P.

$$\begin{aligned}
P_0 &= \{r(0)\} \\
P_1 &= P_0 \cup \textstyle\bigcup_{k<\omega} \{ \quad q(X) \leftarrow q(f(X)), p(X), \\
&\qquad\qquad\qquad q(X) \leftarrow \mathtt{not}\, q(f(X)), p(X), \\
&\qquad\qquad\qquad p(X) \leftarrow \mathtt{not}\, p'(X), \\
&\qquad\qquad\qquad p'(X) \leftarrow \mathtt{not}\, p(X)\,\}\,[X/f^k(0)] \\
P_{i+1} &= P_i \cup \qquad \{c(X) \leftarrow \mathtt{not}\, c(X), \mathtt{not}\, p(X)\}\,[X/f^{i-1}(0)] \quad (i \geq 1)
\end{aligned}$$

Note that $M_0 = \{r(0)\}$ *is a stable model of* P_0 *and for each* $i \geq 1$ *and* $k \geq i - 2$

$$M_i^k = \{ \quad \begin{array}{l} r(0), p(f^0(0)), p(f^1(0)), p(f^2(0)), \ldots, p(f^k(0)), \\ p'(f^{k+1}(0)), p'(f^{k+2}(0)), \ldots, p'(f^{k+j}(0)), \ldots \\ q(f^0(0)), q(f^1(0)), q(f^2(0)), \ldots, q(f^k(0))\} \end{array}$$

is a stable model of P_i*. Therefore, each* P_i *is consistent even if* $\bigcup_i P_i = \mathsf{ground}(P)$ *is inconsistent. This happens because for each stable model* M *of* P_1 *there exists a* P_j $(j > 1)$ *such that* M *is not the bottom part of any stable model of* P_j*. Intuitively,* M *has been "eliminated" at step* j*. In this example* P_1 *has infinitely many stable models, and it turns out that no finite step eliminates all of them. Consequently, each* P_i *in the module sequence is consistent, but the entire program is not.*

6.3 Properties of finitely recursive programs

The smoothness of finitely recursive programs overcomes the problem illustrated by the above example. Since every module P_i is finite, no step in the sequence has infinitely many stable models. Therefore if every P_i is consistent, the entire program P must be consistent, too, and the following theorem holds:

Theorem 6.3.1 *For all finitely recursive programs P:*

1. *if P is consistent then every module sequence for P is consistent;*

2. *if some module sequence for P is consistent, then P is consistent.*

Proof. The point 1 will be proved by contraposition. Suppose that P has an inconsistent module sequence $P_1, P_2, \ldots$. Then, there exists a P_i with no stable models. Therefore, P has an inconsistent bottom set and hence P is inconsistent by the splitting theorem.

To prove point 2, consider a module sequence S for P. If S is consistent then each P_i has a nonempty set of stable models. It suffices to prove that there exists a sequence $M_1, M_2, \ldots$ of stable models of $P_1, P_2, \ldots$, respectively, that satisfies the properties of Theorem 6.2.10, because this implies that $M = \bigcup_i M_i$ is a stable model of P.

A stable model M_i of P_i is *"bad"* if there exists a $k > i$ such that no model M_k of P_k extends M_i, *"good"* otherwise. M_k extends M_i if $M_k \cap \mathsf{atoms}(P_i) = M_i$ (cf. Note 6.2.9). It is possible to claim that each P_i must have at least a *"good"* model.

To prove the claim, suppose that all models of P_i are *"bad"*. Since P_i is a finite program it has a finite number $M_{i_1}, ..., M_{i_r}$ of models. By assumption, for each M_{i_j} there is a program $P_{k_{i_j}}$ none of whose models extends M_{i_j}. Let $k = max\{k_{i_1}, ..., k_{i_r}\}$; then no model of P_k extends a model of P_i, and this is a contradiction because P_k, by hypotheses, has at least a stable model M_k and by the splitting theorem M_k extends a stable model of P_i. This proves the claim.

Now, let M_1 be a *"good"* stable model of P_1; then there must exist a *"good"* stable model of P_2 that extends M_1, exactly for the same reasons, and so on. Therefore there exists an infinite sequence $M_1, M_2, ...$ that satisfies both properties 1 and 2 of Theorem 6.2.10 and hence $M = \bigcup_i M_i$ is a stable model of P. ∎

Note that in Example 6.2.14 there is an infinite P_1 with infinitely many stable models, but all these models are *"bad"* and each of them is dropped by the stable models, respectively, of programs $P_2, P_3, ...$. There is no P_k that eliminates all of them, becoming inconsistent, as it happens when the program is finitely recursive and inconsistent. Indeed the program P of Example 6.2.14 is not finitely recursive.

This result can be extended to all smooth splitting sequences, including sequences with length $\mu > \omega$.

Theorem 6.3.2 *Let $\langle U_\alpha \rangle_{\alpha<\mu}$ be a smooth splitting sequence for a normal program P. P is consistent if and only if for each $\alpha < \mu$, $bot_{U_\alpha}(P)$ is consistent.*

Proof.[Sketch] If there exists an inconsistent $bot_{U_\alpha}(P)$ then P is inconsistent because it has an inconsistent bottom set.

Now, let $\langle U_\alpha \rangle_{\alpha<\omega}$ be an enumerable splitting sequence for P and suppose that, for each $n \in \mathbb{N}$, $P_n = bot_{U_n}(P)$ is consistent. It will be proved that P is consistent.

Since the splitting sequence is smooth then P_0 is finite and it is also consistent by hypotheses. Hence, P_0 must have at least a *"good"* model (cf. the proof of Theorem 6.3.1), call it M'_0. By definition of splitting sequence, for each n, $U_n \subseteq U_{n+1}$ and then $P_n \subseteq P_{n+1}$ and $P_n = bot_{U_n}(P_{n+1})$. In particular, $P_1 = bot_{U_1}(P) = bot_{U_0}(P) \cup P_1 \setminus P_0$. Again, P_1 is consistent and then, by splitting theorem, each model of P_1 must have the form $M_0 \cup M_e$, where M_0 is a stable model of P_0 and M_e is a stable model of $e_{U_0}(P_1 \setminus P_0, M_0)$. But $P_1 \setminus P_0$ is finite (by definition of smooth splitting sequence), hence P_1 can have a finite number of stable models and then, because M'_0 is *"good"*, there must exist an M'_e such that $M'_1 = M'_0 \cup M'_e$ is a *"good"* model of P_1.

In general, for each $n \in \mathbb{N}$, P_{n+1} must have a *"good"* model M_{n+1} which contains all the chosen *"good"* models M'_i, $i \leq n$. Because all such models $M'_0, M'_1, ...,$

satisfy the conditions 1 and 2 of Theorem 6.2.10, then $M'_\omega = \bigcup_{i\in\mathbb{N}} M'_i$ is a stable model of $\mathsf{ground}(P) = \bigcup_{n<\omega} P_n$.

This proof can be generalized to any smooth splitting sequence $\langle U_\alpha\rangle_{\alpha<\mu}$ in the following way. For each ordinal α let $P_\alpha = bot_{U_\alpha}(P)$ and suppose that for each not limit ordinal α, P_α is consistent. P is consistent if each P_α is consistent. It will be proved by induction up to μ that for each ordinal $\alpha < \mu$ holds the following property:

1. P_α has a *"good"* model M'_α that contains all the *"good"* models M'_0, M'_1, ..., M'_μ, ..., M'_β so far chosen for each P_β, with $\beta < \alpha$.

As proved, this is the case for each $n < \omega$ and for $P_\omega = \bigcup_{n<\omega} P_n$.

Now, if α is not a limit ordinal, being $P_\alpha = P_{\alpha-1} \cup P_\alpha \setminus P_{\alpha-1} = bot_{U_{\alpha-1}}(P) \cup P_\alpha \setminus P_{\alpha-1}$, then again each model M_α of P_α must have the form $M_{\alpha-1} \cup M_e$ where $M_{\alpha-1}$ is a stable model of $P_{\alpha-1}$ and M_e is a stable model of $e_{U_{\alpha-1}}(P_\alpha \setminus P_{\alpha-1}, M_{\alpha-1})$, which is finite because $\langle U_\alpha\rangle_{\alpha<\mu}$ is smooth. If $P_{\alpha-1}$ satisfies the property 1 and $M'_{\alpha-1}$ is the *"good"* model of $P_{\alpha-1}$, then there must exist at least an M'_e such that $M'_\alpha = M'_{\alpha-1} \cup M'_e$ is a *"good"* model of P_α and that contains all the *"good"* models chosen so far.

If, instead, α is a limit ordinal then $P_\alpha = \bigcup_{\beta<\alpha} P_\beta$ and $M'_\alpha = \bigcup_{\beta<\alpha} M'_\beta$ is necessarily the *"good"* model of P_α, being $\langle U_\beta\rangle_{\beta<\alpha}$ a splitting sequence for P_α and of course it satisfies the property 1. Therefore, $\mathsf{ground}(P) = \bigcup_{\alpha<\mu} P_\alpha$ is consistent. ∎

Note that if the enumeration of GH is effective, then for each i the corresponding subprogram P_i can be effectively constructed. In this case $\langle \mathsf{atoms}(P_i)\rangle_{i<\omega}$ is an effective enumerable splitting sequence for P. This observation is the basis for the complexity results proved in the rest of this chapter.

6.3.1 Compactness

Here it will be proved that the compactness theorem for finitary programs actually holds for all finitely recursive programs.

Theorem 6.3.3 *Let P be a finitely recursive program and S be any module sequence for P. P is consistent if and only if S is consistent.*

Proof. It follows from Theorems 6.3.1 and 6.2.7. ∎

For understanding the following theorem, see Definition 4.3.7 of unstable kernel.

Theorem 6.3.4 (Compactness) *A finitely recursive program P has no stable model if and only if it has a finite unstable kernel.*

Proof. By Proposition 6.2.12 and Theorem 6.3.1, P has no stable model if and only if it has an inconsistent module sequence. So, let $P_1, P_2, ..., P_n, ...$ be an inconsistent module sequence for P and let $i \geq 1$ such that P_i is inconsistent. By Proposition 6.2.5, $P_i \subseteq \mathsf{ground}(P)$ and P_i is also downward closed. So it is an unstable kernel for P. Moreover, by Theorem 6.2.7, P_i is finite. ∎

The compactness theorem for finitary programs [16] can now be regarded as a corollary of the above theorem.

6.3.2 Reasoning on finitely recursive programs

By taking an effective enumeration of the set GH, one can effectively compute each element of the corresponding module sequence. Let us call CONSTRUCT (P, i) an effective procedure that, given the finitely recursive program P and the index i, returns the ground program P_i, and let $SM(P_i)$ be an algorithm that computes the finite set of the finite stable models of P_i:

Theorem 6.3.5 *Let P be a finitely recursive program. Deciding whether P is inconsistent is at most semidecidable.*

Proof. Given a module sequence $P_1, P_2, ..., P_n, ...$ for P, consider the algorithm CONSISTENT (P).

By Proposition 6.2.12 and Theorem 6.3.1, P is inconsistent if and only if there exists an $i \geq 1$ such that P_i is inconsistent (note that the consistency of P_i can be always checked because P_i is finite). Then, the algorithm returns $FALSE$ if and only if P is inconsistent.

Note that if $\mathsf{ground}(P)$ is infinite then any module sequence for P is infinite and the algorithm CONSISTENT (P) terminates if and only if P is not consistent. ∎

Corollary 6.3.6 *Let P be a finitary program and $K_o(P)$ be the set of all odd-cyclic atoms of* $\mathsf{ground}(P)$*. Deciding whether P is inconsistent is decidable.*

Algorithm CONSISTENT (P)

1: $i = 0$;
2: $answer$ = TRUE;
3: **repeat**
4: $i = i + 1$;
5: P_i =CONSTRUCT (P, i);
6: **if** $SM(P_i) = \emptyset$ **then**
7: $answer$ = FALSE;
8: **until** $\neg answer$ OR $P_i = \mathsf{ground}(P)$
9: **return** $answer$;

Proof. Consider the set

$$R_o(P) = \{R \mid R \in \mathsf{ground}(P) \text{ and } head(R) \in U_o(P)\}$$

where

$$U_o(P) = \{B \mid \text{for some } A \in K_o(P),\ A \geq B\}.$$

A finitary program P has no stable model if and only if $R_o(P)$ has no stable model [16]. Moreover, $R_o(P)$ is finite. Therefore CONSISTENT $(R_o(P))$ always terminates. ∎

Next I deal with skeptical inference. Recall that a closed formula F is a skeptical consequence of P if and only if F is satisfied (according to classical semantics) by all the stable models of P.

Theorem 6.3.7 *Let P be a finitely recursive program and $P_1, P_2, \ldots$ be a module sequence for P. A ground formula F is a skeptical consequence of P if and only if there exists a finite $k \geq 1$ such that F is a skeptical consequence of P_k and* $\mathsf{atoms}(F) \subseteq \mathsf{atoms}(P_k)$.

Proof. Let h be the least integer such that $\mathsf{atoms}(F) \subseteq \mathsf{atoms}(P_h)$ (note that there always exists such a h because $\mathsf{atoms}(F)$ is finite). Suppose that there exists a $k \geq h$ such that F is a skeptical consequence of P_k. Since P_k is a bottom for P, then each stable model of P contains a model of P_k and then satisfies F. So, F is a skeptical consequence of P. This proves the "if" part.

Now suppose that, for each $k \geq h$, F is not a skeptical consequence of P_k. This implies that each P_k is consistent (hence P is consistent) and, moreover, the set S of all the stable models of P_k that falsify F is not empty.

Note that S is finite because P_k is finite (as P is finitely recursive). So, if all the models in S are *"bad"* (cf. the proof of Theorem 6.3.1), then there exists a finite integer $j > k$ such that no model of P_j contains any model of S. Consequently, F is a skeptical consequence of P_j — a contradiction.

Therefore at least one of these model must be *"good"*. Then there must be a model M of P that contains this *"good"* model of P_k, and hence F is not a skeptical consequence of P. ∎

The next theorem follows easily.

Theorem 6.3.8 *Let P be a finitely recursive program. For all ground formulas F, the problem of deciding whether F is a skeptical consequence of P is at most semidecidable.*

Proof. Given a module sequence $P_1, P_2, ..., P_n, ...$ for P, consider the algorithm SKEPTICAL (P, F) where P is supposed to be ground.

Algorithm SKEPTICAL (P, F)

```
 1: answer = FALSE;
 2: i = 0;
 3: repeat
 4:    i = i + 1;
 5:    P_i = CONSTRUCT (P, i);
 6: until atoms(F) ⊆ atoms(P_i)
 7: repeat
 8:    if SM(P_i) = ∅ OR P_i skeptically entails F then
 9:       answer = TRUE;
10:    else
11:       i = i + 1;
12:       P_i = CONSTRUCT (P, i);
13: until answer OR P_i = P
14: return answer;
```

For each P_i such that $\mathsf{atoms}(F) \subseteq \mathsf{atoms}(P_i)$, the algorithm SKEPTICAL (P, F) checks if F is a skeptical consequence of P_i. Since P_i is finite, we can always decide if F is a skeptical consequence of P_i. So, by Theorem 6.3.7, the algorithm returns $TRUE$ if and only if F is a skeptical consequence of P.

Note that if ground(P) is infinite then any module sequence for P is infinite and the algorithm SKEPTICAL (P, F) terminates if and only if F is a skeptical consequence of P. ∎

For a complete characterization of the complexity of ground queries and inconsistency checking, it is only left to prove that the above upper bounds are tight.

Theorem 6.3.9 *Deciding whether a finitely recursive program P is inconsistent is r.e.-complete.*

Proof. By Theorem 6.3.5 the inconsistency checking over the class of finitely recursive programs is at most semidecidable.

Now it is proved to be also r.e.-hard by reducing the problem of skeptical inference of a quantified formula over a finitary program (that is an r.e.-complete problem [16, Corollary 23]) to the problem of inconsistency checking over a finitely recursive program.

Let P be a finitary program and $\exists F$ be a closed quantified formula. Let $((L_{11} \vee L_{12} \vee ...) \wedge (L_{21} \vee L_{22} \vee ...) \wedge ...)$ be the conjunctive normal form of $\neg F$. Then $\exists F$ is a skeptical consequence of P if and only if the program $P \cup C$ is inconsistent, where

$$C = \left\{ \begin{array}{l} p_1(\vec{X_1}) \leftarrow \texttt{not}\, L_{11}, \texttt{not}\, L_{12}, ..., \texttt{not}\, p_1(\vec{X_1}) \\ p_2(\vec{X_2}) \leftarrow \texttt{not}\, L_{21}, \texttt{not}\, L_{22}, ..., \texttt{not}\, p_2(\vec{X_2}) \\ \quad\quad\quad \vdots \end{array} \right\},$$

p_1, p_2, ... are new atom symbols not occurring in P or F, and $\vec{X_i}$ is the vector of all variables occurring in $(L_{i1} \vee L_{i2} \vee ...)$. Note that $P \cup C$ is a finitely recursive program.

The constraints in C add no model to P, but they only discard those models of P that satisfy $F\theta$ (for some substitution θ). So, let $SM(P)$ be the set of stable models of P. Then each model in $SM(P \cup C)$ satisfies $\forall \neg F$. $SM(P \cup C) = \emptyset$ (that is $P \cup C$ is inconsistent) if and only if either $SM(P) = \emptyset$ or all stable models of P satisfy $\exists F$. Then $SM(P \cup C) = \emptyset$ if and only if $\exists F$ is a skeptical consequence of P. ∎

Theorem 6.3.10 *Deciding whether a finitely recursive program P skeptically entails a ground formula F is r.e.-complete.*

Proof. As proved in Theorem 6.3.8 deciding whether a finitely recursive program P skeptically entails a ground formula F is at most semidecidable.

Now it is proved to be also r.e.-hard by reducing the problem of inconsistency checking over a finitely recursive program to the problem of skeptical inference of a ground formula over a finitely recursive program.

Let P be a finitely recursive program and A be a new ground atom that does not occur in P. Then, P is inconsistent if and only if A is a skeptical consequence of P. Since A occurs in the head of no rule of P, A cannot occur in a model of P. So, P skeptically entails A if and only if P has no model. ∎

Corollary 6.3.11 *Deciding whether a finitely recursive program P does not credulously entail a ground formula F is co-r.e. complete.*

Proof. The proof follows immediately from Theorem 6.3.10 and from the fact that a ground formula F is not a credulous consequence of P if and only if $\neg F$ is a skeptical consequence of P. ∎

6.4 Skeptical resolution and finitely recursive programs

In this section the work in [14, 16] is extended by proving that skeptical resolution (a top-down calculus which is known to be complete for datalog and finitary programs under the skeptical stable model semantics) is complete also for the class of finitely recursive programs. Skeptical resolution has several interesting properties. For example, it does not require the input program P to be instantiated before reasoning, and it can produce nonground (i.e., universally quantified) answer substitutions.

For a complete description of the five inference rules of skeptical resolution, the reader is referred to [14]. Here, I only recall that a crucial rule called *failure rule* is expressed in terms of an abstract negation-as-failure mechanism derived from the notion of *support*. Recall that a support for a ground atom A is a set of negative literals obtained by unfolding the goal A with respect to the given program P until no positive literal is left.

Definition 6.4.1 ([14]) *Let A be a ground atom. A* ground counter-support *for A in a program P is a set of atoms K with the following properties:*

1. *For each support S for A, there exists* $\texttt{not}\, B \in S$ *such that* $B \in K$.
2. *For each $B \in K$, there exists a support S for A such that* $\texttt{not}\, B \in S$.

In other words, the first property says that K contradicts all possible ways of proving A, while the second property is a sort of relevance property. Informally speaking, the failure rule of skeptical resolution says that if all atoms in a counter-support are true, then all attempts to prove A fail, and hence `not` A can be concluded.

Of course, in general, counter-supports are not computable and may be infinite (while skeptical derivations and their goals should be finite). In [14] the notion of counter-support is generalized to nonground atoms in the following way:

Definition 6.4.2 *A (generalized)* counter-support *for A is a pair $\langle K, \theta \rangle$ where K is a set of atoms and θ a substitution, such that for all grounding substitutions σ, $K\sigma$ is a ground counter-support for $A\theta\sigma$.*

The actual mechanism for computing counter-supports can be abstracted by means of a suitable function CounterSupp, mapping each (possibly nonground) atom A onto a set of *finite* generalized counter-supports for A. The underlying intuition is that function CounterSupp captures all the negative inferences that can actually be computed by the chosen implementation. To achieve completeness for the nonground skeptical resolution calculus, the negation-as-failure mechanism is needed to be complete in the following sense.

Definition 6.4.3 *The function* CounterSupp *is* complete *if and only if for each atom A, for all of its ground instances $A\gamma$, and for all ground counter-supports K for $A\gamma$, there exist $\langle K', \theta \rangle \in \text{CounterSupp}(A)$ and a substitution σ such that $A\theta\sigma = A\gamma$ and $K'\sigma = K$.*

Skeptical resolution is based on *goals with hypotheses* (*h-goals* for short) which are pairs $(G \mid H)$ where H and G are finite sequences of literals. Roughly speaking, the answer to a query $(G \mid H)$ should be *yes* if G holds in all the stable models that satisfy H. Hence $(G \mid H)$ has the same meaning in answer set semantics as the formula $(\bigwedge G \leftarrow \bigwedge H)$. Finally, a *skeptical goal* (*s-goal* for short) is a finite sequence of h-goals, and a *skeptical derivation from P and* CounterSupp *with restart goal G_0* is a (possibly infinite) sequence of s-goals g_0,g_1,..., where each g_{i+1} is obtained from g_i through one of the five rewrite rules of the calculus, as explained in [14]. This calculus is sound for *all* normal programs and counter-support calculation mechanisms, as stated in the following theorem.

Theorem 6.4.4 (Soundness, [14]) *Suppose that an* s-goal $(G \mid H)$ *has a successful skeptical derivation from P and* CounterSupp *with restart goal G and answer*

substitution θ. Then, for all grounding substitution σ, all the stable models of P satisfy $(\bigwedge G\theta \leftarrow \bigwedge H\theta)\sigma$ (equivalently, $\forall(\bigwedge G\theta \leftarrow \bigwedge H\theta)$ is skeptically entailed by P).

However, skeptical resolution is not always complete. Completeness analysis is founded on ground skeptical derivations, that require a ground version of $\mathsf{CounterSupp}$.

Definition 6.4.5 *For all ground atoms A, let $\mathsf{CounterSupp}^g(A)$ be the least set such that if $\langle K, \theta\rangle \in \mathsf{CounterSupp}(A')$ and for some grounding σ, $A = A'\theta\sigma$, then $\langle K\sigma, \epsilon\rangle \in \mathsf{CounterSupp}^g(A)$, where ϵ is the empty substitution.*

Theorem 6.4.6 (Finite Ground Completeness [14]) *If some ground implication $\bigwedge G \leftarrow \bigwedge H$ is skeptically entailed by a* finite *ground program P and the function $\mathsf{CounterSupp}$ is complete with respect to P, then $(G \mid H)$ has a successful skeptical derivation from P and $\mathsf{CounterSupp}^g$ with restart goal G. In particular, if G is skeptically entailed by P, then $(G \mid \emptyset)$ has such a derivation.*

This basic theorem and the following standard lifting lemma allow to prove completeness for all finitely recursive programs.

Lemma 6.4.7 (Lifting [14]) *Let $\mathsf{CounterSupp}$ be complete. For all skeptical derivations $\mathcal{D}$ from $\mathsf{ground}(P)$ and $\mathsf{CounterSupp}^g$ with restart goal G_0, there exists a substitution σ and a skeptical derivation $\mathcal{D}'$ from P and $\mathsf{CounterSupp}$ with restart goal G'_0 and answer substitution θ, such that $\mathcal{D} = \mathcal{D}'\theta\sigma$ and $G_0 = G'_0\theta\sigma$.*

Theorem 6.4.8 (Completeness for finitely recursive programs) *Let P be a finitely recursive program. Suppose $\mathsf{CounterSupp}$ is complete with respect to P and that for some grounding substitution γ, $(\bigwedge G \leftarrow \bigwedge H)\gamma$ holds in all the stable models of P. Then $(G \mid H)$ has a successful skeptical derivation from P and $\mathsf{CounterSupp}$ with restart goal G and some answer substitution θ more general than γ.*

Proof. By Theorems 6.2.7 and 6.3.7, there exists a smooth module sequence for P with finite elements $P_1, P_2, ...,$ and a finite k such that $(\bigwedge G \leftarrow \bigwedge H)\gamma$ holds in all the stable models of P_k. Since each P_i is downward closed, the ground supports of any given $A \in \mathsf{atoms}(P_k)$ with respect to program P_k coincide with the ground supports of A with respect to the entire program P. Consequently, also ground counter-supports and (generalized) counter-supports, respectively, coincide in P_k

and P. Therefore, CounterSupp is complete with respect to P_k, too. As a consequence, since P_k is a ground, finite program, the ground completeness theorem can be applied to conclude that $(G \mid H)\gamma$ has a successful skeptical derivation from P_k and $\mathsf{CounterSupp}^g$ with restart goal $G\gamma$. The same derivation is also a derivation from P (as $P_k \subset \mathsf{ground}(P)$) and $\mathsf{CounterSupp}^g$. Then, by the Lifting lemma, $(G \mid H)$ has a successful skeptical derivation from P and CounterSupp, with restart goal G and some answer substitution θ, such that $(G \mid H)\gamma$ is an instance of $(G \mid H)\theta$. It follows that θ is more general than γ. ∎

6.5 Conclusions

In this chapter I have shown some important properties of the class of finitely recursive programs, a very expressive fragment of logic programs under the stable model semantics. Finitely recursive programs extend the class of finitary programs by dropping the restrictions on odd-cycles. Many of the nice properties of finitary programs are extended to finitely recursive programs: (i) a compactness property (Theorem 6.3.4); (ii) the r.e.-completeness of inconsistency checking and skeptical inference (Theorem 6.3.9); (iii) the completeness of skeptical resolution (Theorem 6.4.8).

Unfortunately, some of the nice properties of finitary programs do *not* carry over to finitely recursive programs: (i) ground queries are not decidable (Theorem 6.3.10 and Corollary 6.3.11); (ii) nonground credulous queries are not semi-decidable (as ground queries are co-r.e.).

The results reported in this chapter clarify precisely the role of each of the two conditions defining finitary programs: the restriction on recursion ensures compactness and brings the complexity of skeptical queries within r.e.; then, roughly speaking, this restriction makes logic programs more similar to classical logic. The restriction on odd-cycles (i.e., on the number of potential inconsistency sources) is necessary to make ground queries decidable and reduce the complexity of nonground credulous queries to r.e. .

As a side benefit, a normal form for splitting and module sequences is introduced, where sequence length is limited to ω and—if the program is finitely recursive—the sequence is smooth (i.e., the "delta" between each non-limit element and its predecessor is finite). Such properties constitute an alternative characterization of finitely recursive programs.

Chapter 7

Composing normal logic programs

7.1 Introduction

In previous chapters I discussed about classes of logic programs such as finitary programs, finitely recursive programs, ω-restricted programs, etc. . All of these classes guarantee good computational properties imposing different restrictions on their programs. If P and Q are two normal programs belonging to these classes, is it possible to reason on the program $P \cup Q$ by taking advantage of properties of P and Q? The idea is to *"compose"* P and Q so obtaining a program $P \cup Q$ that, as a whole, might not be subject to the restrictions of P or Q (in particular this happens when P and Q belong to different classes) but that again enjoys good computational properties.

To do this it is necessary to distinguish what relation there exists between predicates defined in P and predicates defined in Q, that is if P can call predicates defined in Q without redefine them, in that case P *depends* on Q, or if predicates defined in Q cannot occur in P and vice versa, in that case P and Q are *independent*.

7.2 Dependency relations for logic programs

In this chapter, metavariables P and Q are supposed to range over normal programs. $\mathsf{Def}(P)$ denotes the set of predicates *defined in* P, that is, the set of all predicate symbols occurring in the head of some rule in P, while $\mathsf{Called}(P)$ is the set of predicates *called by* P, that is, the set of all predicate symbols occurring in the body of some rule in P.

Now, dependency relations for normal logic programs can be defined as follows:

P depends on Q, in symbols $P \rhd Q$, if and only if

$$\begin{aligned} \mathsf{Def}(P) \cap \mathsf{Def}(Q) &= \emptyset, & (7.1)\\ \mathsf{Def}(P) \cap \mathsf{Called}(Q) &= \emptyset, & (7.2)\\ \mathsf{Called}(P) \cap \mathsf{Def}(Q) &\neq \emptyset. & (7.3) \end{aligned}$$

Conversely, *P and Q are independent* (equivalently, P is independent of Q), in symbols $P \| Q$, if and only if

$$\begin{aligned} \mathsf{Def}(P) \cap \mathsf{Def}(Q) &= \emptyset, & (7.4)\\ \mathsf{Def}(P) \cap \mathsf{Called}(Q) &= \emptyset, & (7.5)\\ \mathsf{Called}(P) \cap \mathsf{Def}(Q) &= \emptyset. & (7.6) \end{aligned}$$

Note that $P \| Q$ if and only if $Q \| P$.

7.3 Composing programs

In this section I propose some important observations for better understanding what resulting program we obtain by composing dependent or independent programs. In particular, given two normal programs P and Q such that $P \rhd Q$ or $P \| Q$, I shall focus on two questions:

- What is the set of odd-cyclic atoms occurring in $P \cup Q$? Recall that the odd-cycles represent for a program its possible inconsistency sources.
- What relation there exists between the ground instance of $P \cup Q$ and the ground instances of P and Q? Since $\mathsf{ground}(P) \cup \mathsf{ground}(Q) \subseteq \mathsf{ground}(P \cup Q)$, does the projection of $\mathsf{ground}(P \cup Q)$ over the rules in P (resp. Q) still enjoy good properties of $\mathsf{ground}(P)$ (resp. $\mathsf{ground}(Q)$)?

Let $\mathsf{OC}(P)$ be the set of odd-cyclic atoms occurring in the dependency graph $DG(P)$ of P.

Proposition 7.3.1 *If $P \rhd Q$ or $P \| Q$, then* $\mathsf{OC}(Q) \cap \mathsf{OC}(P) = \emptyset$.

Proof. Let p be a predicate symbol of a ground atom A belonging to an odd-cycle c in $DG(Q)$ (resp. $DG(P)$). Since c is a cycle, A must have in c at least an outgoing

edge. Then $p \in \mathsf{Def}(Q)$ (resp. $p \in \mathsf{Def}(P)$). By definition of dependency relation $\rhd$ and independency relation $\|$, $\mathsf{Def}(Q) \cap \mathsf{Def}(P) = \emptyset$ and then any ground atom A with predicate symbol p cannot belong both to $\mathsf{OC}(Q)$ and to $\mathsf{OC}(P)$. ∎

Note that, even if P and Q cannot have odd-cyclic atoms in common, they can have in common the predicates on which their odd-cyclic atoms depend.

Proposition 7.3.2 *Let P and Q be normal logic programs with the same Herbrand universe. Then* $\mathsf{ground}(P \cup Q) = \mathsf{ground}(P) \cup \mathsf{ground}(Q)$.

Proof. The proof follows immediately from the definition of ground instantiation. ∎

The above proposition does not hold for programs P and Q with different languages, in particular with different function symbols and constants, and then with different Herbrand universes. For this reason, even if P (and hence $\mathsf{ground}(P)$) belongs to a class of programs (e.g., the class of finitary programs) it might be the case that the projection of $\mathsf{ground}(P \cup Q)$ over the rules in P is not in that class (e.g., it is not finitary).

Theorem 7.3.3 *Let P and Q be finitely recursive and $P \rhd Q$ or $P\|Q$. $P \cup Q$ might not be finitely recursive.*

Proof. Consider Q as

$$R: \; p(a) \leftarrow q(X).$$

Note that the Herbrand universe for Q is $\{a\}$ and then $\mathsf{ground}(Q) = \{p(a) \leftarrow q(a)\}$. However we choose a finitely recursive program P such that $P \rhd Q$ or $P\|Q$, and such that P contains a function symbol f, we have that in $\mathsf{ground}(P \cup Q)$ the rule R has infinitely many ground instances

$$\begin{array}{l} p(a) \leftarrow q(a). \\ p(a) \leftarrow q(f(a)). \\ p(a) \leftarrow q(f(f(a))). \\ \vdots \end{array}$$

and then $p(a)$ depends on infinitely many ground atoms. ∎

Corollary 7.3.4 *Let P and Q be finitary and $P \rhd Q$ or $P\|Q$. $P \cup Q$ might not be finitary.*

Proof. If P and Q are finitary, by definition of finitary programs, they have to be finitely recursive. Hence, by Theorem 7.3.3, $P \cup Q$ might not be finitely recursive and then neither finitary.

Moreover, consider Q as

$$R: \; q(X) \leftarrow \texttt{not}\, q(X).$$

Note that the Herbrand universe for Q is $\{a\}$ and then $\mathsf{ground}(Q) = \{q(a) \leftarrow \texttt{not}\, q(a)\}$. However we choose a finitary program P such that $P \rhd Q$ or $P \| Q$, and such that P contains a function symbol f, the rule R has infinitely many ground instances in $\mathsf{ground}(P \cup Q)$

$$\begin{array}{l} q(a) \leftarrow \texttt{not}\, q(a). \\ q(f(a)) \leftarrow \texttt{not}\, q(f(a)). \\ q(f(f(a))) \leftarrow \texttt{not}\, q(f(f(a))). \\ \vdots \end{array}$$

and then there are infinitely many ground odd-cyclic atoms. ∎

Note that in Theorem 7.3.3 and Corollary 7.3.4, Q is finitary or finitely recursive only because its Herbrand universe is finite, while for any infinite Herbrand universe this property does not hold. This means that Q is finitary or finitely recursive not for a particular form of its rules (that have no syntactic restriction) but only because the language of Q does not contain function symbols.

Next, I give sufficient (even if not necessary) conditions in order that a normal program P is finitely recursive. These conditions impose only syntactic restrictions on the rules of P and are independent from the universe with respect to which P is grounded. Since these conditions are merely syntactic, for any program P satisfying them and for any universe U[1] of ground terms, $\mathsf{ground}(P, U)$ is finitely recursive, where $\mathsf{ground}(P, U)$ is the ground instance of P with respect to the universe U.

Definition 7.3.5 *Let $p(\vec{t})$ be an atom that occurs in the head of some rules of P. A* dependency sequence *for $p(\vec{t})$ and a (possibly nonground) substitution θ over the language of P is a sequence as*

$$(p_1(\vec{t_1}) \leftarrow p_2(\vec{t_2'}))\theta_1, (p_2(\vec{t_2}) \leftarrow p_3(\vec{t_3'}))\theta_2, \ldots$$

[1] I assume that the number of function symbols and constants in U is finite.

where $p(\vec{t}) = p_1(\vec{t_1})$ *and* $\theta = \theta_1$ *and where for each* $i = 1, 2, ...$

1. θ_i *is a possibly nonground substitution over the language of P,*
2. $p_{i+1}(\vec{t'_{i+1}})\theta_i = p_{i+1}(\vec{t_{i+1}})\theta_{i+1}$,
3. *there exists a rule* R_i *belonging to P and such that* $p_i(\vec{t_i})\theta_i$ *is unifiable with* $head(R_i)$ *and* $p_{i+1}(\vec{t'_{i+1}})\theta_i$ *is unifiable with some atom that occurs (possibly negated) in* $body(R_i)$*.*

Note that a dependency sequence may be also a nonground sequence.

Note 7.3.6 *Note that we can write equivalently a dependency sequence as*

$$p_1(\vec{t_1})\theta_1 \leftarrow p_2(\vec{t'_2})\theta_1, p_2(\vec{t_2})\theta_2 \leftarrow p_3(\vec{t'_3})\theta_2, ...$$

In the following I assume a substitution is *trivial* if it is the empty substitution or if it is only a variable renaming. Then a trivial substitution is a substitution equivalent to the empty substitution. In general, two substitutions θ and σ are equivalent if and only if θ is more general than σ and σ is more general than θ. Moreover, θ is strictly more general than σ if θ is more general than σ but θ and σ are not equivalent.

Theorem 7.3.7 *Let P be a normal logic program. For each pair of atoms* $p(\vec{t})$ *and* $p(\vec{t'})$*, such that* $p(\vec{t})$ *occurs in the head of some rules of P and* $p(\vec{t'})$ *occurs (possibly negated) in the body of some rules of P, and for each substitution* σ *over the language of P, if the following holds*

- *whenever* $p(\vec{t})$ *and* $p(\vec{t'})$ *are in different rules of P then*

$$\sigma \textit{ is non trivial} \implies t\sigma \neq t';$$

- *whenever* $p(\vec{t})$ *and* $p(\vec{t'})$ *are in the same rule of P then*

$$\sigma \textit{ is non empty} \implies t\sigma \neq t';$$

then each dependency sequence for $p(\vec{t})$ *and some substitution* θ *(possibly nonground) contains only finitely many (possibly nonground) distinct instances for* $p(\vec{t})$*.*

Proof. Suppose that standardization apart has been applied to P. Let

$$S = p_1(\vec{t_1})\theta_1 \leftarrow p_2(\vec{t_2'})\theta_1, p_2(\vec{t_2})\theta_2 \leftarrow p_3(\vec{t_3'})\theta_2, ...$$

be a dependency sequence for an atom $p_1(\vec{t_1})$ belonging to P and a substitution θ_1.

Note that, for each i, θ_i is not strictly more general than θ_{i+1}. Indeed, if θ_i is strictly more general than θ_{i+1} then there exists a substitution σ such that $\theta_i\sigma = \theta_{i+1}$ (with σ non trivial) and then $t'_{i+1}\theta_i = t_{i+1}\theta_i\sigma$ implies that there exists a non trivial substitution ψ such that $t'_{i+1} = t_{i+1}\psi$, and this is a contradiction.

Suppose $p(\vec{t})$ a recursive atom in P (that is an atom that occurs both in the head of some rule in P and in the body of some rule in P) and let S be

$$\begin{array}{l} p(\vec{t})\theta_{11} \leftarrow q_{12}(\vec{s'_{12}})\theta_{11}, q_{12}(\vec{s_{12}})\theta_{12} \leftarrow q_{13}(\vec{s'_{13}})\theta_{12}, ..., q_{1k_1}(\vec{s_{1k_1}})\theta_{1k_1} \leftarrow p(\vec{t^1})\theta_{1k_1}, \\ p(\vec{t})\theta_{21} \leftarrow q_{22}(\vec{s'_{22}})\theta_{21}, q_{22}(\vec{s_{22}})\theta_{22} \leftarrow q_{23}(\vec{s'_{23}})\theta_{22}, ..., q_{2k_2}(\vec{s_{2k_2}})\theta_{2k_2} \leftarrow p(\vec{t^2})\theta_{2k_2}, \\ ... \end{array}$$

a dependency sequence for $p(\vec{t})\theta_{11}$. Since standardization apart has been applied on P, the fact that in the sequence an atom $q(\vec{s})$ occurs more than once means that a rule is applied more than once.

I will prove that if $p(\vec{t})$ does not occur finitely many times in S then there exists only a finite set $\Theta = \{\theta_1, \theta_2, ..., \theta_m\}$ of distinct substitutions such that $p(\vec{t})\theta_i$ $(i \leq m)$ occurs in S.

Consider the sequence θ_{11}, θ_{21}, θ_{31}, ..., of substitutions applied to $p(\vec{t})$ in S. By what has been proved, for each i, either $\theta_{(i+1)1}$ is strictly more general than θ_{i1} or $\theta_{(i+1)1}$ and θ_{i1} are equivalent.

If for each i, $\theta_{(i+1)1}$ is strictly more general than θ_{i1} then, since the variables in $\vec{t}$ are finitely many and the terms in $\vec{t}$ are finite, then there exists a finite integer h such that θ_{h1} is equivalent to the empty substitution.

Otherwise, suppose that there is an i such that θ_{i1}, $\theta_{(i+1)1}$, $\theta_{(i+2)1}$, ..., is a (possibly infinite) sequence of equivalent substitutions. I will prove that this sequence contains only finitely many distinct substitutions. Indeed, if θ_{i1} is equivalent to $\theta_{(i+1)1}$ then all substitutions θ_{i1}, θ_{i2}, ..., θ_{ik_i} are equivalent and, since in P no local variable occurs in a recursive predicate, the variables occurring in $p(\vec{t^i})\theta_{ik_i}$ (that is the variables occurring in the atom $p(\vec{t^i})$ after that the substitution θ_{ik_i} has been applied to it) occur also in $p(\vec{t})\theta_{i1}$ and, since $p(\vec{t^i})\theta_{ik_i} = p(\vec{t})\theta_{(i+1)1}$, also variables occurring in $p(\vec{t})\theta_{(i+1)1}$ occur in $p(\vec{t})\theta_{i1}$. Moreover, both θ_{i1} and $\theta_{(i+1)1}$ are substitutions over $\vec{t}$. Then the set of terms and variables of $\vec{t}$ occurring in $\theta_{(i+1)1}$ is a

subset of those in θ_{i1}. For the same reasons, terms and variables of $\vec{t}$ occurring in $\theta_{(i+2)1}$ are also in $\theta_{(i+1)1}$, and so on. Since $\vec{t}$ and $p(\vec{t})\theta_{i1}$ are finite, then the number of substitutions in θ_{i1}, $\theta_{(i+1)1}$, $\theta_{(i+2)1}$, ..., that are distinct with respect to terms and variables in $\vec{t}$, is finite. ∎

Corollary 7.3.8 *Let P be a normal logic program without local variables. Under the conditions of Theorem 7.3.7, P is finitely recursive.*

Proof. If P has no local variable then its dependency graph cannot contain an infinite branching. Moreover, if P satisfies the conditions of Theorem 7.3.7 then any path in its dependency graph contains finitely many distinct atoms. ∎

Definition 7.3.9 *A finitely recursive program P is* domain independent *if:*

1. *P has no local variable;*
2. *for each pair of atoms $p(\vec{t})$ and $p(\vec{t'})$, such that $p(\vec{t})$ occurs in the head of some rules of P and $p(\vec{t'})$ occurs (possibly negated) in the body of some rules of P, however we choose a non trivial substitution σ over the language of P, $t\sigma \neq t'$.*

The following example shows how a domain independent finitely recursive program allows us to encode a bounded simulation of a given Turing machine while it cannot simulate a general Turing machine.

Example 7.3.10 *Let $\mathcal{M}$ be a deterministic Turing machine with semi-infinite tape and with S as set of states and V as tape alphabet. An instruction for $\mathcal{M}$ is a 5-tuples $\langle s, v, v', s', m\rangle \in S \times V \times V \times S \times \{\mathsf{left}, \mathsf{right}\}$, where s and v are the current state and symbol respectively, v' is the symbol to be written in the current cell of tape, s' is the next state and m is the $\mathcal{M}$'s head movement.*

Let $t(s, L, v, R)$ be the predicate that encodes a configuration of $\mathcal{M}$ where s is the current state, L is the list of symbols (in reverse order) on the left of $\mathcal{M}$'s head, v is the current symbol and R is the list of symbols on the right of $\mathcal{M}$'s head (R might have a tail of blank symbols) and let the following program $P_{\mathcal{M}}$ (due to Bonatti [16]) be an encoding of all bounded simulations of $\mathcal{M}$:

$$
\begin{array}{lll}
R1: & t(s, L, v, [V|R]) \leftarrow t(s', [v'|L], V, R). & for\ all\ instr.\ \langle s, v, v', s', \mathsf{right}\rangle \\
R2: & t(s, [V|L], v, R) \leftarrow t(s', L, V, [v'|R]). & for\ all\ instr.\ \langle s, v, v', s', \mathsf{left}\rangle \\
R3: & t(s, L, v, R). & for\ all\ final\ states\ s \\
R4: & blank_list([]). & \\
R5: & blank_list([b|L]) \leftarrow blank_list(L). &
\end{array}
$$

Note that $P_{\mathcal{M}}$ is a domain independent finitely recursive program (and also a finitary program because it is positive). Indeed, $P_{\mathcal{M}}$ contains no local variable. Moreover, for any substitution σ, $t(s, L, v, [V|R])\sigma \neq t(s', [v'|L], V, R)$ and $t(s, L, v, [V|R])\sigma \neq t(s', L, V, [v'|R])$ because there is no substitution σ' such that $[V|R]\sigma' = R$ or $v\sigma' = V$. The same holds for $t(s, [V|L], v, R)$ and $t(s, L, v, R)$ with respect to $t(s', [v'|L], V, R)$ and $t(s', L, V, [v'|R])$, and for $blank_list([])$ and $blank_list([b|L])$ with respect to $blank_list(L)$.

As proved in [16], for any ground substitution θ the goal

$$G = (blank_list(R), t(s, [], v_0, [v_1, ..., v_n|R]))\theta$$

can be derived from $P_{\mathcal{M}}$ if and only if $\mathcal{M}$ terminates on $\langle v_0, v_1, ..., v_n\rangle$ using only k cells, where k is the tape length for the encoding of $([], v_0, [v_1, ..., v_n|R\theta])$.

If we add to $P_{\mathcal{M}}$ the rule

$$R6: \ u(R) \leftarrow blank_list(R), t(s, [], v_0, [v_1, ..., v_n|R]), \texttt{not}\, u(R).$$

then we obtain a finitely recursive (but not finitary) program $P'_{\mathcal{M}}$ that is inconsistent if and only if $\mathcal{M}$ terminates on $\langle v_0, v_1, ..., v_n\rangle$.

Note that $P'_{\mathcal{M}}$ is not finitary because the rule $R6$ generates infinitely many odd-cycles and it is neither a domain independent finitely recursive program because we can unify the head of the instance of rule $R1$ with v_0 as third argument, with $t(s, [], v_0, [v_1, ..., v_n|R])$ in the body of rule $R6$:

$$t(s, L, v_0, [V|R])\{L/[], V/[v_1, ..., v_n]\} = t(s, [], v_0, [v_1, ..., v_n|R]).$$

Definition 7.3.11 *A finitary program is* domain independent *if it is a domain independent finitely recursive program and its odd-cycles are ground.*

Domain independent finitary (and then also domain independent finitely recursive) programs can codify most of the standard predicates on lists [69] as shown by the following example.

Example 7.3.12 *The programs here illustrated*

$member(X, [X|Y]).$
$member(X, [Y|Z]) \leftarrow member(X, Z).$

$append([], L, L).$
$append([X|X_s], L, [X|Y_s]) \leftarrow append(X_s, L, Y_s).$

$reverse(L, R) \leftarrow reverse(L, [], R).$
$reverse([], R, R).$
$reverse([X|X_s], A, R) \leftarrow reverse(X_s, [X|A], R).$

implement operations, such as member*,* append *and* reverse*, on lists.*

All of these programs are domain independent finitary programs. Indeed they are all positive programs without local variables. Moreover, consider the recursive predicate member*. Then, for any substitution* σ*,* $\mathsf{member}(X, [Y|Z])\sigma \neq \mathsf{member}(X, Z)$ *and* $\mathsf{member}(X, [X|Y])\sigma \neq \mathsf{member}(X, Z)$ *because* $[Y|Z]$ *and* $[X|Y]$ *are not unifiable with* Z *without applying any substitution to* Z*.*

For the same reasons, however we choose a substitution σ*,* $\mathsf{append}([X|X_s], L, [X|Y_s])\sigma \neq \mathsf{append}(X_s, L, Y_s)$ *and* $\mathsf{append}([], L, L)\sigma \neq \mathsf{append}(X_s, L, Y_s)$ *and again, this holds for* reverse *predicate.*

Domain independent finitely recursive programs and domain independent finitary programs satisfy the conditions of Theorem 7.3.7 and Corollary 7.3.8, then following theorems hold.

Theorem 7.3.13 *Let* P *be a domain independent finitely recursive program. For each ground universe* U*,* $\mathsf{ground}(P, U)$ *is finitely recursive.*

Proof. The theorem follows from Theorem 7.3.7 and Corollary 7.3.8. ∎

Note that, under the conditions of Theorem 7.3.7, the rules in P can contain local variables only if body recursive predicates in which local variables occur cannot be unified with the same recursive predicates in the head of some rule. This, in fact, prevents infinite paths to be in the dependency graph of P while does not prevent infinite branching. Intuitively, this is because under the conditions of Theorem 7.3.7, we cannot simulate recursion through increasing terms, as in the following example.

Example 7.3.14 *Consider the rule:*

$$p(X_1) \leftarrow p(X_2).$$

This rule allows us to simulate the rule

$$p(X) \leftarrow p(f(X)).$$

whose dependency graph has an infinite path

$$p(a) \rightarrow p(f(a)) \rightarrow p(f(f(a))) \rightarrow ....$$

Theorem 7.3.15 *Let P be a domain independent finitary program. For each ground universe U,* $\mathsf{ground}(P, U)$ *is finitary.*

Proof. By Definition 7.3.11, P is a domain independent finitely recursive program and, by Theorem 7.3.13, $\mathsf{ground}(P, U)$ is still finitely recursive. Moreover, since in P the odd-cycles are ground, and then are finitely many, it follows that $\mathsf{ground}(P, U)$ has again finitely many odd-cycles. So, $\mathsf{ground}(P, U)$ is finitary. ∎

For achieving program composition, I suppose a very general class of programs that can be composed with finitely recursive programs or finitary programs preserving good computational properties.

I define *ELP* as a class of normal logic programs for which two of the main tasks, consistency checking and credulous inference, are effectively computable.

Definition 7.3.16 *Let $\mathcal{U}$ be the set of all universes U of ground terms and $\mathcal{S}$ be the set of all sets S of ground literals. A normal logic programs P is an ELP program if there exist effective functions $\mathcal{F}_m : \{P\} \times \mathcal{U} \rightarrow \{\mathtt{yes}, \mathtt{no}\}$ and $\mathcal{F}_c : \{P\} \times \mathcal{U} \times \mathcal{S} \rightarrow \{\mathtt{yes}, \mathtt{no}\}$ such that*

1. *$\mathcal{F}_m(P, U)$ returns* yes *if* $\mathsf{ground}(P, U)$ *has at least one stable model,* no *otherwise;*

2. *$\mathcal{F}_m(P, U, S)$ returns* yes *if* $\mathsf{ground}(P, U)$ *has at least one stable model that satisfies S,* no *otherwise.*

From the above definition follows that if P is an *ELP* program then, for any universe of ground terms U, $\mathsf{ground}(P, U) \in ELP$.

Note that any ω-restricted program P belongs to *ELP*. Indeed, if we consider U as the Herbrand universe of P then $\mathsf{ground}(P, U)$ is the ground instance of P

and it is always possible to check if P is consistent. Moreover, given a ground atom A, $P \cup \{A\}$ is still an ω-restricted program and also $P \cup \{f \leftarrow \texttt{not}\, f, A\}$, where f is a ground atom not occurring in P. So, we can always decide whether P has a stable model containing A or not containing A. Adding to P similar rules for each literal in a set of ground literals S it is possible to check if S is a credulous consequence of P. If U is any ground universe then $\mathsf{ground}(P, U)$ is again an ω-restricted program (the definition of ω-restricted program is based only on the predicate symbols occurring in P and not on their instances [71]). Then an ω-restricted program satisfies the properties 1 and 2 of the *ELP* class.

Let $P_{P\cup Q}$ be the projection of $\mathsf{ground}(P \cup Q)$ over the rules in P, then $P_{P\cup Q}$ is ground and, by Theorems 7.3.13 and 7.3.15, $P_{P\cup Q}$ is finitely recursive if P is finitely recursive and finitary if P is finitary while, by definition of class *ELP*, it belongs to *ELP* if P does. Moreover, $P_{P\cup Q} \cup Q_{P\cup Q} = \mathsf{ground}(P \cup Q)$ and if P and Q has the same Herbrand universe then, by Proposition 7.3.2, $P_{P\cup Q} = \mathsf{ground}(P)$ and $Q_{P\cup Q} = \mathsf{ground}(Q)$.

Theorem 7.3.17 *If $P \triangleright Q$ or $P \| Q$, then $P_{P\cup Q} \triangleright Q_{P\cup Q}$ or $P_{P\cup Q} \| Q_{P\cup Q}$, respectively.*

Proof. Let U be the Herbrand universe of $P \cup Q$. Since $P_{P\cup Q} = \mathsf{ground}(P, U)$ then $\mathsf{Def}(P) = \mathsf{Def}(P_{P\cup Q})$ and $\mathsf{Called}(P) = \mathsf{Called}(P_{P\cup Q})$ (resp. $\mathsf{Def}(Q) = \mathsf{Def}(Q_{P\cup Q})$ and $\mathsf{Called}(Q) = \mathsf{Called}(Q_{P\cup Q})$). Then the relations that hold among $\mathsf{Def}(P_{P\cup Q})$, $\mathsf{Called}(P_{P\cup Q})$, $\mathsf{Def}(Q_{P\cup Q})$ and $\mathsf{Called}(Q_{P\cup Q})$ are the same than for P and Q. ∎

7.4 Structural and semantic properties

In the following I analyze the characteristics of programs resulting from composition. First, I prove the relationships between the class which the resulting program belongs to and the classes of programs involved in the composition, and then the relationships between the structural properties of the composed program and the structural properties of programs that compose it. Then, I investigate the complexity of some of the main reasoning tasks on the resulting program with respect to the complexity of the same tasks on the starting programs.

Theorem 7.4.1 *Let $P \triangleright Q$, then:*

1. *if P and Q are domain independent finitely recursive programs then $P \cup Q$ is finitely recursive;*

2. *if $P \cup Q$ is finitely recursive then P and Q are finitely recursive.*

Proof. Suppose P and Q domain independent finitely recursive programs, then by Theorem 7.3.13 $P_{P \cup Q}$ and $Q_{P \cup Q}$ are finitely recursive. For each atom A in $\mathsf{ground}(P \cup Q)$, if A depends on some atoms then the predicate symbol in A either belongs to $\mathsf{Def}(Q)$ or to $\mathsf{Def}(P)$. If the predicate symbol in A belongs to $\mathsf{Def}(Q)$ then A depends on finitely many atoms because $Q_{P \cup Q}$ is finitely recursive and, by Theorem 7.3.17, A cannot depend on atoms in $P_{P \cup Q}$ not occurring in $Q_{P \cup Q}$.

If the predicate symbol in A belongs to $\mathsf{Def}(P)$ then A depends, in $P_{P \cup Q}$, on a finite set of atoms $\mathsf{DS}(P_{P \cup Q}, A) = \{B_1, B_2, ..., B_k\}$ and, in $Q_{P \cup Q}$, on the set of atoms $\bigcup_{1 \leq i \leq k} \mathsf{DS}(Q_{P \cup Q}, B_i)$ (note that the predicate symbol in A does not belong to $\mathsf{Def}(Q)$). So, in $\mathsf{ground}(P \cup Q)$, A depends on $\mathsf{DS}(P \cup Q, A) = \mathsf{DS}(P_{P \cup Q}, A) \cup \bigcup_{1 \leq i \leq k} \mathsf{DS}(Q_{P \cup Q}, B_i)$. The set $\mathsf{DS}(P_{P \cup Q}, A)$ is finite because $P_{P \cup Q}$ is finitely recursive. Moreover, since $Q_{P \cup Q}$ is finitely recursive, each $\mathsf{DS}(Q_{P \cup Q}, B_i)$ is finite.

Now, suppose that P (resp. Q) is not finitely recursive. Then, in P (resp. in Q) there exists an atom A that depends on infinitely many atoms $\mathsf{DS}(P, A) = \{B_1, B_2, ...\}$ (resp. $\mathsf{DS}(Q, A) = \{B_1, B_2, ...\}$). Again in $P \cup Q$, A depends on $\{B_1, B_2, ...\}$ (and maybe on also other adding atoms) and then $P \cup Q$ is not finitely recursive. ∎

Theorem 7.4.2 *Let $P \triangleright Q$, then:*

1. *if P and Q are domain independent finitary programs then $P \cup Q$ is finitary;*

2. *if $P \cup Q$ is finitary then P and Q are finitary.*

Proof. Suppose P and Q domain independent finitary programs. Then it must to be proved that

1. $P \cup Q$ is finitely recursive,

2. there are finitely many odd-cyclic atoms in the dependency graph of $P \cup Q$.

By Theorem 7.4.1 the condition 1 holds.

The condition 2 is satisfied because, by Proposition 7.3.1, the number of odd-cyclic atoms in $P \cup Q$ is the sum of odd-cyclic atoms in P and of odd-cyclic atoms in Q, and both are finite.

Now, suppose that P (resp. Q) is not finitary. Then, P (resp. Q) does not satisfy the condition 1 or the condition 2. If P (resp. Q) is not finitely recursive then, by Theorem 7.4.1, $P \cup Q$ is not finitely recursive.

If P (resp. Q) contains infinitely many odd-cyclic atoms these atoms again, by Proposition 7.3.1, occur in $P \cup Q$ as odd-cyclic atoms. ∎

Theorem 7.4.3 *Let $P \| Q$, then:*

1. *if P and Q are domain independent finitely recursive programs then $P \cup Q$ is finitely recursive;*

2. *if $P \cup Q$ is finitely recursive then P and Q are finitely recursive.*

Proof. Suppose P and Q domain independent finitely recursive programs. By Theorem 7.3.13, $P_{P \cup Q}$ and $Q_{P \cup Q}$ are finitely recursive and then, for each atom A in $\mathsf{ground}(P \cup Q)$, if A depends on some atoms then the predicate symbol of A either belongs to $\mathsf{Def}(Q)$ or belongs to $\mathsf{Def}(P)$. If the predicate symbol of A belongs to $\mathsf{Def}(P)$ (resp. $\mathsf{Def}(Q)$) then A depends on finitely many atoms because $P_{P \cup Q}$ (resp. $Q_{P \cup Q}$) is finitely recursive and A cannot depend on atoms in $Q_{P \cup Q}$ (resp. $P_{P \cup Q}$) not occurring in $P_{P \cup Q}$ (resp. $Q_{P \cup Q}$) by Theorem 7.3.17.

Now, suppose that P (resp. Q) is not finitely recursive. Then, in $P_{P \cup Q}$ (resp. in $Q_{P \cup Q}$) there exists an atom A that depends on infinitely many atoms $\mathsf{DS}(P_{P \cup Q}, A) = \{B_1, B_2, ...\}$ (resp. $\mathsf{DS}(Q_{P \cup Q}, A) = \{B_1, B_2, ...\}$). Again in $P \cup Q$, A depends on $\{B_1, B_2, ...\}$ and then $P \cup Q$ is not finitely recursive. ∎

Theorem 7.4.4 *If $P \| Q$, then:*

1. *if P and Q are domain independent finitary programs then $P \cup Q$ is finitary;*

2. *if $P \cup Q$ is finitary then P and Q are finitary.*

Proof. Suppose P and Q domain independent finitary programs. Then it must to be proved that

1. $P \cup Q$ is finitely recursive,

2. there are finitely many odd-cyclic atoms in the dependency graph of $P \cup Q$.

By Theorem 7.4.3 the condition 1 holds.

The condition 2 is satisfied by $P \cup Q$ because, by Proposition 7.3.1, its number of odd-cyclic atoms is the sum of odd-cyclic atoms in P and of odd-cyclic atoms in Q, and both are finite.

Now, suppose that P (resp. Q) is not finitary. Then, P (resp. Q) does not satisfy the condition 1 or the condition 2. If P (resp. Q) is not finitely recursive then, by Theorem 7.4.3, $P \cup Q$ is not finitely recursive.

If P (resp. Q) contains infinitely many odd-cyclic atoms, these atoms again, by Proposition 7.3.1, occur in $P \cup Q$ as odd-cyclic atoms. ∎

The definition of *ELP* does not impose syntactic restrictions and so programs belonging to different classes with different syntactic characteristics could be *ELP* programs while their union not, because by merging their different rules we might obtain a program for which the consistency checking or credulous inference is not decidable. So, if we consider two normal programs P and Q belonging to *ELP* and such that a dependency relation holds, the program $P \cup Q$ might not belong to *ELP*.

Example 7.4.5 *Let Q' be any logic program not belonging to ELP and let Q be the program $Q' \cup \{p \leftarrow \texttt{not}\, q, \texttt{not}\, p\}$ where the predicate symbols p and q are not in Q'. The program $Q \in ELP$ because it has a finite inconsistent bottom $\{p \leftarrow \texttt{not}\, q, \texttt{not}\, p\}$. Let $P = \{q\}$ be a ground program (note that $P \triangleright Q$). Then $P \in ELP$ but clearly $P \cup Q$ is not an ELP program.*

However, if P and Q are independent then, by properties of the independency relation $\|$, again $P \cup Q$ is an *ELP* program as the following theorem proves.

Theorem 7.4.6 *Suppose $P\|Q$. If both P and Q belongs to ELP then $P \cup Q$ belongs to ELP.*

Proof. If $P\|Q$ then each stable model for $P \cup Q$ is the union of a stable model for $P_{P \cup Q}$ and a stable model for $Q_{P \cup Q}$, and vice versa. So, $P \cup Q$ is consistent if and only if $P_{P \cup Q}$ and $Q_{P \cup Q}$ are consistent, and this check is decidable.

Moreover, for the same reasons, each ground atom is a credulous consequence of $P \cup Q$ if and only if it is a credulous consequence of $P_{P \cup Q}$ or of $Q_{P \cup Q}$ and $P \cup Q$ is consistent, and again this check is decidable. ∎

In general, when $P \triangleright Q$ or $P\|Q$ and $P \cup Q \in ELP$, P and Q are not necessarily *ELP* programs. In this regard, consider the following example.

Example 7.4.7 *Let Q be the following ELP program*

$$\{p \leftarrow \texttt{not}\, p\}.$$

Since Q is inconsistent however we choose a program P such that $P \triangleright Q$ or $P\|Q$, $P \cup Q$ is inconsistent and no ground atom is a credulous consequence of $P \cup Q$. Then, $P \cup Q$ belongs to ELP, while P might not be an ELP program.

Now, let analyze some semantic properties for composed programs.

Theorem 7.4.8 *Suppose that either $P \triangleright Q$ or $P \| Q$. If P and Q are domain independent finitary programs and the set $\mathsf{OC}(P \cup Q)$ of all odd-cyclic atoms in $\mathsf{ground}(P \cup Q)$ is given, then*

1. *Skeptical and credulous ground queries, G, are decidable in $P \cup Q$.*
2. *Skeptical and credulous nonground queries, $\exists G$, are semidecidable in $P \cup Q$.*

Proof. By Theorems 7.4.2 and 7.4.4, $P \cup Q$ is finitary and, as proved in [16, 8] the assertions 1 and 2 hold for finitary programs. ∎

Theorem 7.4.9 *Suppose that $P \| Q$. If P and Q belong to ELP, then skeptical and credulous ground queries for $P \cup Q$ are decidable.*

Proof. By Theorem 7.4.6, $P \cup Q$ belongs to *ELP*. Credulous ground queries are, then, decidable for $P \cup Q$. A ground query G is skeptical inferred from $P \cup Q$ if and only if $P \cup Q$ is inconsistent or $\texttt{not}\, G$ is not a credulous consequence of $P \cup Q$ (that are decidable tasks). ∎

Theorem 7.4.10 *Let P be a domain independent finitely recursive program and $Q \in ELP$ such that $P \triangleright Q$. If $Q_{P \cup Q}$ has a finite number of stable models then deciding whether $P \cup Q$ is inconsistent is at most semidecidable.*

Proof. Let $P_1, P_2, ...$ be a module sequence (cf. Definition 6.2.3) for $P_{P \cup Q}$ (note that $P_{P \cup Q}$ is still finitely recursive by Theorem 7.3.13) and let $PQ_0, PQ_1, ...$, be the module sequence for $P_{P \cup Q} \cup Q_{P \cup Q}$ such that

- $PQ_0 = Q_{P \cup Q}$
- $PQ_1 = P_1 \cup Q_{P \cup Q}$
- $PQ_2 = P_2 \cup Q_{P \cup Q}$
- ... .

For each P_i, let S_i be the set of all atoms in P_i whose predicate symbols belong to $\mathsf{Def}(Q)$. S_i is finite and is a set of ground atoms because P_i is finite and ground. If there not exists a subset SS_i of S_i such that $P_i \cup SS_i$ has a model that does not satisfy $S_i \backslash SS_i$ and $Q_{P \cup Q}$ has a model that satisfies SS_i and that does not satisfy $S_i \backslash SS_i$, then $PQ_i = P_i \cup Q_{P \cup Q}$ is inconsistent. Since, for each i, $PQ_i = bot_{\mathsf{atoms}(PQ_i)}(P \cup Q)$ then if PQ_i is inconsistent also $P \cup Q$ is inconsistent. It follows that if there

exists a finite i such that PQ_i is inconsistent then also $P \cup Q$ is inconsistent, and deciding whether there exists such an i is semidecidable.

Now it will be proved that if, for each $i \geq 1$, PQ_i is consistent then also $P \cup Q$ is consistent. This means that deciding whether $P \cup Q$ (and then $P_{P \cup Q} \cup Q_{P \cup Q}$) is inconsistent, is semidecidable.

By hypotheses, for each $i \geq 1$, PQ_i is consistent and then $Q_{P \cup Q}$ is consistent.

Now, it is proved that there exists a sequence M_0, M_1, $M_2, ...$ of stable models for PQ_0, PQ_1, $PQ_2, ...$, respectively, that satisfies the conditions of Theorem 6.2.10. So it is proved that $P_{P \cup Q} \cup Q_{P \cup Q}$ has a stable model $M = \bigcup_{i \geq 0} M_i$, that is $P_{P \cup Q} \cup Q_{P \cup Q}$ is consistent.

A stable model M_i for PQ_i is *"bad"* if there exists a k such that no model M_k for PQ_k extends M_i, *"good"* otherwise. M_k extends M_i if $M_k \cap \mathsf{atoms}(PQ_i) = M_i$ (note that by construction $PQ_i = bot_{\mathsf{atoms}(PQ_i)}(PQ_k)$).

If PQ_i is consistent then it must have at least a *"good"* model. Indeed, suppose that all models of PQ_i are *"bad"*. Since $PQ_i = P_i \cup Q_{P \cup Q}$, P_i is finite, $Q_{P \cup Q}$ has finitely many stable models and $Q_{P \cup Q} = bot_{\mathsf{atoms}(Q_{P \cup Q})}(PQ_i)$ because $P_{P \cup Q} \rhd Q_{P \cup Q}$ and $P_i \subseteq P_{P \cup Q}$, then PQ_i has a finite number of stable models $M_{i_1}, ..., M_{i_r}$. By assumption, for each M_{i_j} there is a program PQ_{k_j} none of whose models extends M_{i_j}. Let $k = max\{k_{i_1}, ..., k_{i_r}\}$. Then no model of PQ_k extends a model of PQ_i and this is a contradiction because, by hypotheses, PQ_k has at least a stable model M_k and, by splitting theorem, M_k extends a model of PQ_i. Now let M_0 be a *"good"* stable model for PQ_0; then there must exist a *"good"* stable model M_1 for PQ_1 that extends M_0, exactly for the same reasons, and so on. Therefore, there exists a sequence $M_0, M_1, M_2, ...$ of stable models for $PQ_0, PQ_1, PQ_2, ...$, respectively, where each model extends the previous model in the sequence. Then, by Theorem 6.2.10, $M = \bigcup_{i \geq 0} M_i$ is a stable model for $P_{P \cup Q} \cup Q_{P \cup Q}$. ∎

The proof of previous theorem proves that if P is a domain independent finitely recursive program and $Q \in ELP$ then there is a module sequence S for $P \cup Q$ such that $P \cup Q$ is consistent if and only if S is consistent. Theorem 6.3.1 proves that this holds for finitely recursive programs because all of their module sequences are smooth and then each element P_i of such a module sequence is a finite ground program and then it has finitely many stable models. Since a module sequence for $Q_{P \cup Q}$ may be not smooth it is needed to impose that $Q_{P \cup Q}$ has a finite number of stable models.

Proposition 7.4.11 *If $P \| Q$, then a set of ground atoms M is a stable model of $P \cup Q$ if and only if $M = M_Q \cup M_P$ where M_Q is a stable model of $Q_{P \cup Q}$ and M_P is a stable model of $P_{P \cup Q}$.*

Proof. By definition of independency relation $\|$ and by Theorem 7.3.17, we have that $Q_{P\cup Q} = bot_{\mathsf{atoms}(Q_{P\cup Q})}(P \cup Q)$. So, by splitting theorem [45], M is a stable model of $P \cup Q$ if and only if $M = M_Q \cup M_P$ where M_Q is a stable model of $Q_{P\cup Q}$ and M_P is a stable model of $e_{\mathsf{atoms}(Q_{P\cup Q})}(P_{P\cup Q}, M_Q)$. Since $e_{\mathsf{atoms}(Q_{P\cup Q})}(P_{P\cup Q}, M_Q) = P_{P\cup Q}$ because, by definition of independency relation M_Q cannot contain atoms that occur in the body of some rule of $P_{P\cup Q}$, then M_P is also a stable model of $P_{P\cup Q}$. ∎

Theorem 7.4.12 *Let P be a domain independent finitely recursive program and* $Q \in ELP$ *such that* $P\|Q$. *Deciding whether* $P \cup Q$ *is inconsistent is at most semidecidable.*

Proof. If $P\|Q$ then $P \cup Q$ is consistent if and only if, by Proposition 7.4.11, both $P_{P\cup Q}$ and $Q_{P\cup Q}$ are so. Deciding whether $Q_{P\cup Q}$ is consistent is decidable by definition of ELP class, and deciding whether $P_{P\cup Q}$ is consistent is semidecidable because $P_{P\cup Q}$, by Theorem 7.3.13, is finitely recursive. Then deciding whether $P \cup Q$ is consistent is semidecidable. ∎

Theorem 7.4.13 *Let P be a domain independent finitary program and* $Q \in ELP$ *such that* $P \rhd Q$ *or* $P\|Q$. *Deciding whether* $P \cup Q$ *is inconsistent is decidable.*

Proof. Let P_{odd} be the ground subprogram of $P_{P\cup Q}$ containing all its odd-cycles (note that P is domain independent finitary program and then its odd-cyclic atoms are ground) and the ground instances of rules on which they depend (that are finitely many because $P_{P\cup Q}$ is finitary by Theorem 7.3.15). Let S be the set of all atoms in P_{odd} whose predicate symbols are in $\mathsf{Def}(Q)$. S is finite and ground because P_{odd} is finite and ground. If $P \rhd Q$ or $P\|Q$ then $P \cup Q$ is consistent if and only if there exists a subset SS of S such that $P_{odd} \cup SS$ has a model that does not satisfy $S \setminus SS$ and such that $Q_{P\cup Q}$ has a model that satisfy SS and not $S \setminus SS$. Since it is always possible to decide the above task then deciding whether $P \cup Q$ is consistent is decidable. ∎

Theorem 7.4.14 *Let* $Q \in ELP$, *P be a domain independent finitely recursive program,* $P \rhd Q$ *and G be a ground atom. If* $Q_{P\cup Q}$ *has a finite number of stable models then deciding whether G is a skeptical consequence of* $P \cup Q$ *is at most semidecidable.*

Proof. Suppose that the predicate symbol in G belongs to $\mathsf{Def}(P)$. G is a skeptical consequence of $P \cup Q$ if and only if $P \cup \{f \leftarrow \texttt{not}\, f, G\} \cup Q$ (with f predicate

symbol not occurring in $P \cup Q$) is inconsistent. Note that the same dependency relation between P and Q again holds between $P \cup \{f \leftarrow \texttt{not}\, f, G\}$ and Q. Moreover, $P \cup \{f \leftarrow \texttt{not}\, f, G\}$ is still a domain independent finitely recursive program. Then, by Theorem 7.4.10, proving that $P \cup \{f \leftarrow \texttt{not}\, f, G\} \cup Q$ is inconsistent is at most semidecidable.

Suppose that the predicate symbol in G belongs to $\mathsf{Def}(Q)$. If $\texttt{not}\, G$ is not a credulous consequence of $Q_{P \cup Q}$ (that is decidable) then G is a skeptical consequence of $P \cup Q$.

If the predicate symbol in G does not belong either to $\mathsf{Def}(Q)$ or to $\mathsf{Def}(P)$ then G may be a skeptical consequence of $P \cup Q$ if and only if $P \cup Q$ is inconsistent, that is at most semidecidable by Theorem 7.4.10. ∎

Theorem 7.4.15 *Let $Q \in ELP$, P be a domain independent finitely recursive program, $P \| Q$ and G be a ground atom. Deciding whether G is a skeptical consequence of $P \cup Q$ is at most semidecidable.*

Proof. If $P \| Q$ then G is a skeptical consequence of $P \cup Q$ if and only if G is a skeptical consequence of $Q_{P \cup Q}$ or G is a skeptical consequence of $P_{P \cup Q}$.

Suppose that the predicate symbol of G belongs to $\mathsf{Def}(Q)$. If $\texttt{not}\, G$ is not a credulous consequence of $Q_{P \cup Q}$ (that is decidable) then G is a skeptical consequence of $P \cup Q$.

Suppose that the predicate symbol in G belongs to $\mathsf{Def}(P)$. Since by Theorem 7.3.13 $P_{P \cup Q}$ is finitely recursive then checking whether G is an its skeptical consequence is semidecidable.

If the predicate symbol in G does not belong either to $\mathsf{Def}(Q)$ or to $\mathsf{Def}(P)$ then G may be a skeptical consequence of $P \cup Q$ if and only if $P \cup Q$ is inconsistent, that is at most semidecidable by Theorem 7.4.10. ∎

Theorem 7.4.16 *Let $Q \in ELP$, P be a domain independent finitary program, $P \triangleright Q$ or $P \| Q$, and G be a ground atom. Deciding whether G is a skeptical consequence of $P \cup Q$ is decidable.*

Proof. Suppose that the predicate symbol in G belongs to $\mathsf{Def}(P)$. G is a skeptical consequence of $P \cup Q$ if and only if $P \cup \{f \leftarrow \texttt{not}\, f, G\} \cup Q$ (with f predicate symbol not occurring in $P \cup Q$) is inconsistent. Note that the same dependency relation between P and Q again holds between $P \cup \{f \leftarrow \texttt{not}\, f, G\}$ and Q. Moreover, $P \cup \{f \leftarrow \texttt{not}\, f, G\}$ is still a domain independent finitary program. Then, by Theorem 7.4.13, proving that $P \cup \{f \leftarrow \texttt{not}\, f, G\} \cup Q$ is inconsistent is decidable.

Suppose that the predicate symbol G belongs to $\mathsf{Def}(Q)$. If $\mathtt{not}\ G$ is not a credulous consequence of $Q_{P\cup Q}$ (that is decidable) then G is a skeptical consequence of $P \cup Q$.

If the predicate symbol in G does not belong either to $\mathsf{Def}(Q)$ or $\mathsf{Def}(P)$ then G may be a skeptical consequence of $P \cup Q$ if and only if $P \cup Q$ is inconsistent, that is decidable by Theorem 7.4.13. ∎

Theorem 7.4.17 *There exist P and Q such that*

1. $P \rhd Q$,
2. *P belongs to ELP and Q is a domain independent finitary program,*
3. *Skeptical and credulous ground queries are* not *semidecidable.*

Proof. Consider the positive finitary program Q, as defined in Example 7.3.10, that encodes all bounded simulations of a deterministic Turing machine $\mathcal{M}$ with semi-infinite tape:

$$
\begin{array}{ll}
t(s, L, v, [V \mid R]) \leftarrow t(s', [v' \mid L], V, R)\,. & \text{for all instr. } \langle s, v, v', s', \mathsf{right}\rangle \\
t(s, [V \mid L], v, R) \leftarrow t(s', L, V, [v' \mid R])\,. & \text{for all instr. } \langle s, v, v', s', \mathsf{left}\rangle \\
t(s, L, v, R)\,. & \text{for all final states } s.
\end{array}
$$

Note that Q is a domain independent finitary program.

Let P be

$$
\begin{array}{l}
p(X) \leftarrow p([b \mid X])\,. \\
p(X) \leftarrow t(s_0, [\,], v_0, [v_1, \ldots, v_n \mid X])\,.
\end{array}
$$

$P \in ELP$ because it is a positive logic program without facts and then its only stable model is $\emptyset$.

Note that $p(x)$ holds if x is a list of blank symbols such that $\mathcal{M}$ terminates on $[v_0, v_1, \ldots, v_n \mid x]$ starting from the initial state s_0.

Consider the ground query $p([\,])$. For checking if $p([\,])$ is a skeptical/credulous consequence of $P \cup Q$ we have to check if $\mathcal{M}$ terminates on $[v_0, v_1, \ldots, v_n]$, then if $\mathcal{M}$ terminates on $[v_0, v_1, \ldots, v_n, b]$, then on $[v_0, v_1, \ldots, v_n, b, b]$, and so on (note that $\mathsf{OC}(P \cup Q) = \emptyset$). That is, we have to do possibly infinitely many semidecidable tests. ∎

The previous proof shows that the composition of an *ELP* program P and a domain independent finitary program Q, such that $P \rhd Q$, does not necessarily yield a finitary program.

7.5 Conclusions

What I have presented in this chapter is only a preliminary study and much work has still to be done for better investigating the potential of composition.

The classes of domain independent finitely recursive and domain independent finitary programs allow us to reason on infinite domains and models as well as finitely recursive and finitary programs do. But the former classes are closed under the grounding operation $\mathsf{ground}(.,.)$, that is the ground instance of a domain independent finitely recursive (resp. finitary) program over any universe of ground terms again belongs to that class, and then these classes are more suitable to be composed with programs with different Herbrand universes. So, a complete characterization of domain independent finitely recursive and domain independent finitary programs would be useful to define their complexity and what problems they can really express.

Moreover, I would extend the results on composition by exploiting the properties of programs obtained by composing *ELP* programs with other programs not belonging to the class of domain independent finitely recursive or domain independent finitary programs.

Chapter 8

Constrained programs

8.1 Introduction

In this chapter, as we first proposed in [11], I integrate answer set generation and constraint solving to reduce the memory requirements for a class of multi-sorted *logic programs with cardinality constraints* [67] whose signature can be partitioned into: (i) a set of so-called *regular predicates* over domains whose size can be handled by a standard answer set solver; (ii) a set of *constrained* predicates that can be handled by a constraint solver in a way that does not require grounding (so larger domains can be allowed here); (iii) a set of predicates—called *mixed predicates*—that create a "bridge" between the above two partitions.

Then reasoning can be implemented by having an answer set solver interact with a constraint solver. A critical aspect is the form that the definitions of mixed predicates may take. If they were completely general, then that part of the program would be just as hard to reason with as unrestricted programs because mixed predicates may range over arbitrary domains. Accordingly, the framework presented in [11] and reproposed in this chapter supports restricted definitions for mixed predicates, that can be either functions from "regular" to "large" domains (strong semantics) or slightly weaker mappings where each combination of "regular" values must be associated to at least one vector of values from "large" domains (weak semantics).

In [11] we studied the relationships between strong and weak semantics, and introduced an algorithm for computing the strong semantics efficiently under the simplifying assumption that mixed predicates do not occur in the scope of negation. Here, I report new experimental results providing evidence that this approach

can solve problem instances with significantly larger domains. In this chapter I focus only on the comparison with a standard answer set programming approach.

8.2 Basic terminology

Here a sorted first-order language based on a given signature Σ is adopted. Let $\mathcal{S}$ be a finite set of *sorts*. Assume a *sort specification* is given, that is, a function sort mapping:

- each constant c onto a set $\mathsf{sort}(c) \subseteq \mathcal{S}$;
- each variable X onto a (single) sort $\mathsf{sort}(X) \in \mathcal{S}$;
- each n-ary function symbol f onto a tuple $\mathsf{sort}(f) = \langle S_1, \ldots, S_{n+1} \rangle \in \mathcal{S}^{n+1}$;
- each n-ary predicate symbol p onto a tuple $\mathsf{sort}(p) = \langle S_1, \ldots, S_n \rangle \in \mathcal{S}^n$.

Note that sorts may overlap because constants may be associated to two or more sorts.

Example 8.2.1 ([11].) *A sort* steps*, modeling plan steps, may contain the integer constants in the interval* $[0, 10]$*, while a sort* time*, modeling time points, may contain the integer constants in* $[0, 600000]$.

All the other terms have a unique sort. Intuitively, in $\mathsf{sort}(f)$, S_i is the sort of the i-th argument of f $(1 \leq i \leq n)$ and S_{n+1} is the sort of the output. Similarly, in $\mathsf{sort}(p)$, S_i is the sort of the i-th argument of predicate p $(1 \leq i \leq n)$.

Terms and atoms are defined accordingly. Each variable X with $\mathsf{sort}(X) = S$ and each constant c such that $S \in \mathsf{sort}(c)$ are terms of sort S. Each expression $f(t_1, \ldots, t_n)$ such that $\mathsf{sort}(f) = \langle S_1, \ldots, S_n, S \rangle$ and each t_i is a term of sort S_i is a term of sort S. Nothing else is a term. I write $t : s$ to state that term t belongs to sort s.

All expressions $p(t_1, \ldots, t_n)$ such that $\mathsf{sort}(p) = \langle S_1, \ldots, S_n \rangle$ and each t_i is a term of sort S_i are atoms. As usual, literals are either atoms (positive literals) or expressions of the form `not` A where A is an atom (negative literals).

A variable substitution over $\{X_1, \ldots, X_n\}$ is a function mapping each variable X_i onto a term of $\mathsf{sort}(X_i)$. The notions of instance and ground instantiation are defined as usual from the above notion of (typed) substitution. The ground instantiation of a set of expressions E will be denoted by $\mathsf{ground}(E)$.

8.3 Constrained Programs

The sorts of constrained programs are partitioned into *regular* and *constrained* sorts. Intuitively, regular sorts are small enough to be handled by standard answer set solvers, while constrained sorts are large enough to require reasoners that do not instantiate the corresponding variables.

Variables and constants are called *regular* or *constrained* according to their sorts. A function f is regular (resp. constrained) if all the sorts in $\mathsf{sort}(f)$ are regular (resp. constrained). Function f is *mixed* if $\mathsf{sort}(f)$ comprises both regular and constrained sorts. Predicate symbols are classified in a similar way.

I assume that the output sort of all functions is a constrained sort. The reason is that most answer set solvers do not (yet) support function symbols, while constraint solvers do (functions are typically standard arithmetic functions).

According to the above classification, signature Σ is partitioned into Σ_r, Σ_c and Σ_m, where r, c and m stand for *regular*, *constrained* and *mixed* respectively.

The atoms over Σ_r, Σ_c, and Σ_m are referred to as r-atoms, c-atoms, and m-atoms respectively. Similarly for literals. The parameters of an m-atom whose sorts are constrained (regular) will be often referred to as c-parameters (r-parameters).

I assume that c-predicates have a predefined interpretation, and that the equality predicate is a c-predicate. The intended interpretation of c-predicates will be represented by a set of ground atoms M_c (the set of all true ground c-atoms).

Regular predicates can be defined with normal programs, as in standard ASP. The definitions of mixed predicates are restricted, instead. Let an atom be *free* if its arguments are all pairwise distinct variables. For all free atoms A I write $A(\vec{X}_r, \vec{X}_c)$ to state that the r-variables (resp. c-variables) of A are those in $\vec{X}_r$ (resp. $\vec{X}_c$). I denote with $A(\vec{a}, \vec{b})$ the instance of A such that $\vec{X}_r$ is replaced by $\vec{a}$ and $\vec{X}_c$ with $\vec{b}$.

I deal with two possible semantics of mixed predicates.[1] Under the *weak semantics*, for all free mixed atoms $A(\vec{X}_r, \vec{X}_c)$ there is an implicit axiom

$$\forall \vec{X}_r \exists \vec{X}_c . A(\vec{X}_r, \vec{X}_c)\,, \tag{8.1}$$

that can be expressed by including into the program a cardinality constraint $1\{A(\vec{a}, \vec{X}_c)\}$ (cf. Chapter 3) for each sequence of ground arguments $\vec{a}$ of the appropriate type and length.[2]

[1] A more general approach is described in the final discussion.

[2] In Smodels this can be done with a single rule having a cardinality constraint in the head. A similar remark applies to the encoding of (8.2). I refer the reader to [67] for more details.

Under the *strong semantics*, for all free mixed atoms $A(\vec{X}_r, \vec{X}_c)$ there is an implicit axiom

$$\forall \vec{X}_r \exists! \vec{X}_c . A(\vec{X}_r, \vec{X}_c)\,, \tag{8.2}$$

that can be encoded in a similar way with a suitable set of cardinality constraints like $1\{A(\vec{a}, \vec{X}_c)\}1$.

Moreover, constrained programs may contain constraints that relate all kinds of predicates (regular, constrained, and mixed).

Definition 8.3.1

1. *A* regular rule *(r-rule) is a rule of the form $A \leftarrow B$ or $\leftarrow B$ where A is an r-atom and B is a collection of r-literals.*

2. *A (proper)* constraint *is a rule of the form $\leftarrow B$ where B is a collection of arbitrary literals, including at least one nonregular literal.*

3. *A* constrained program*, P, is the union of a set of regular rules, $R(P)$, and a set of constraints, $C(P)$.*

Example 8.3.2 ([11].) *In the running example (a planning and scheduling problem) there are two regular sorts: step (representing plan steps) and action. I write step* $: 0..\,10$ *to state that the constants c with step $\in \mathsf{sort}(c)$ are those in the integer interval* $[0,\ 10]$. *Analogously, I may write action : $a_1, \ldots, a_n$ to enumerate all possible actions.*

The regular signature Σ_r contains only one relation o over action $\times$ step. Intuitively, $o(A, S)$ means that action A occurs at step S.

The regular part $R(P)$ contains n rules that force at least one action to be executed at each step. For $i = 1, \ldots, n$:

$$o(a_i, S) \leftarrow \mathtt{not}\, o(a_1, S), \ldots, \mathtt{not}\, o(a_{i-1}, S), \mathtt{not}\, o(a_{i+1}, S), \ldots, \mathtt{not}\, o(a_n, S).$$

Moreover, $R(P)$ contains a denial that forbids concurrent actions:

$$\leftarrow o(A, S1), o(A, S2), \mathtt{not}\, eq(S1, S2).$$
$$eq(X, X).$$

The constraint signature Σ_c comprises the sort time $: 0..600000$ *with the standard arithmetic functions: $+, -, |\,|$ etc., and relations: $>, \geq$, etc.*

The mixed signature Σ_m comprises a relation $time(S, T)$ associating each plan step S to at least one time point T under the weak semantics (exactly one under the strong semantics).

The following constraints $C(P)$ ensure that time is assigned to steps monotonically and that each step is associated to exactly one time point (the latter is needed only under the weak semantics);

$$\leftarrow time(S1, T1), time(S2, T2), S1 < S2, T1 \geq T2.$$
$$\leftarrow time(S, T1), time(S, T2), T1 \neq T2.$$

Moreover, one can specify a minimal duration for each action, e.g., 3 time units for a_1

$$\leftarrow o(a_1, S1), time(S1, T1), o(A2, S2), time(S2, T2), |T2 - T1| < 3\,. \qquad (8.3)$$

Formally, the semantics of constrained programs is a specialization of the stable model semantics for logic programs with weight constraints, taking into account the intended interpretation M_c of Σ_c and the implicit semantics of mixed predicates.

First a generalization of the program *reduct* P^I (cf. Chapter 2) is needed, where P is now a constrained program and I a set of ground atoms. The reduct P^I is obtained from $\mathsf{ground}(P)$ by removing:

- all the rules and constraints with a literal $\texttt{not}\, B$ in their body, such that $B \in I \cup M_c$;
- all rules and constraints with a c-atom A in their body, such that $A \notin M_c$;
- all negative literals and c-atoms from the remaining rules and constraints.

Note that P^I is a set of Horn clauses also under the generalized definition (cf. Chapter 2). Therefore, if P^I is consistent, then it has a unique minimal Herbrand model $\mathsf{lm}(P^I)$. Like the standard notion of reduct, P^I results from the evaluation of negative literals against I. Moreover, the generalized notion evaluates all the constrained literals with respect to their intended semantics M_c.

Definition 8.3.3 *A* weak answer set *of a constrained program P is a set of ground atoms $M = M_r \cup M_m$ satisfying the following conditions:*

AS1 *M_r is a set of r-atoms and M_m is a set of m-atoms;*

AS2 $R(P)^{M_r}$ *is consistent and* $M_r = \mathsf{lm}(R(P)^{M_r})$*;*

AS3 *each constraint* $(\leftarrow \vec{L}) \in \mathsf{ground}(C(P))$ *contains a literal* L_i *false in* M*;*

AS4 *for each free m-atom* $A(\vec{X}_r, \vec{X}_c)$*, and for each vector of r-constants* $\vec{a}$ *of the appropriate length,* M_m *contains at least one instance of* $A(\vec{a}, \vec{X}_c)$*.*

A strong answer set *of a constrained program* P *is a weak answer set* $M = M_r \cup M_m$ *satisfying the following additional condition:*

AS5 *for each free m-atom* $A(\vec{X}_r, \vec{X}_c)$*, and for each vector of r-constants* $\vec{a}$ *of the appropriate length,* M_m *contains* at most *one instance of* $A(\vec{a}, \vec{X}_c)$*.*

Note that AS2 basically states that M_r is a stable model of the regular part of P.

The semantics of a constrained program P can be alternatively specified as the stable models of the program obtained by extending P with M_c and with the cardinality constraints that encode (8.1) and (8.2). Then AS1-AS5 might have been proved as theorems. This requires an extension of the *splitting set theorem* [46].

Theorem 8.3.4 (Extended Splitting Theorem [7]) *Let* U *be a splitting set for a normal logic program* P *with cardinality constraints. A set* M *of ground atoms is a stable model for* P *if and only if* $M = I \cup J$*, where*

1. *I is a stable model of* $bot_U(P)$*, and*

2. *J is a stable model of* $e_U(\mathsf{ground}(P) \setminus bot_U(P), I)$*.*

Proof. Suppose P ground. P can be always translated into an equivalent logic program P' without cardinality constraints replacing each cardinality constraint in P

$$l\{A_1, \ldots, A_n, not\ B_1, \ldots, not\ B_m\}u$$

with the following three sets of rules:

$$\begin{array}{l}\{\leftarrow A_{i_1}, \ldots, A_{i_k}, not\ A_{i_{k+1}}, \ldots, not\ A_{i_n}, not\ B_{j_1}, \ldots, not\ B_{j_k}, B_{j_{k+1}}, \ldots, B_{j_m} : \\ \quad 1 \leq i_1 < \ldots < i_k \leq n,\ 1 \leq i_{k+1} < \ldots < i_n \leq n, \\ \quad 1 \leq j_1 < \ldots < j_k \leq m,\ 1 \leq j_{k+1} < \ldots < j_n \leq m, \\ \quad |\{A_{i_1}, \ldots, A_{i_k}, B_{j_1}, \ldots, B_{j_k}\}| < l\},\end{array}$$

$$\begin{array}{l}\{\leftarrow A_{i_1}, \ldots, A_{i_k}, not\ B_{j_1}, \ldots, not\ B_{j_k} : \\ \quad 1 \leq i_1 < \ldots < i_k \leq n, \\ \quad 1 \leq j_1 < \ldots < j_k \leq m, \\ \quad |\{A_{i_1}, \ldots, A_{i_k}, B_{j_1}, \ldots, B_{j_k}\}| > u\}\end{array}$$

and

$$\{A_{i_1}, \ldots, A_{i_k}, B_{j_{k+1}}, \ldots, B_{j_m} \leftarrow not\ A_{i_{k+1}}, \ldots, not\ A_{i_n}, not\ B_{j_1}, \ldots, not\ B_{j_k} :$$
$$1 \leq i_1 < \ldots < i_k \leq n,\ 1 \leq i_{k+1} < \ldots < i_n \leq n$$
$$1 \leq j_1 < \ldots < j_k \leq m,\ 1 \leq j_{k+1} < \ldots < j_n \leq m,$$
$$l \leq |\{A_{i_1}, \ldots, A_{i_k}, B_{j_1}, \ldots, B_{j_k}\}| \leq u\},$$

such that $\{i_1, \ldots, i_k\} \cap \{i_{k+1}, \ldots, i_n\} = \emptyset$, $\{j_1, \ldots, j_k\} \cap \{j_{k+1}, \ldots, j_n\} = \emptyset$.

According to this translation, P' is the set of all normal rules in P and of rules that translate cardinality constraints occurring in P.

Follows that if U is a splitting set for P then it is a splitting set also for P', $bot_U(P')$ is the set of all normal rules in $bot_U(P)$ and of rules that translate cardinality constraints occurring in $bot_U(P)$, and $top_U(P')$ is the set of all normal rules in $top_U(P)$ and of rules that translate cardinality constraints occurring in $top_U(P)$ (note that the translation does not introduce new predicate symbols). So, $bot_U(P) \equiv bot_U(P')$ and $top_U(P) \equiv top_U(P')$.

Since $P \equiv P'$, a set M is a stable model for P if and only if it is a stable model for P'. By splitting theorem, a set M is a stable model for P' if and only if $M = I \cup J$ where I is a stable model for $bot_U(P')$ and J is a stable model for $e_U(P' \setminus bot_U(P'), I)$. Since $bot_U(P) \equiv bot_U(P')$ and $top_U(P) \equiv top_U(P')$, then I (resp. J) is a stable model for $bot_U(P')$ (resp. for $e_U(P' \setminus bot_U(P'), I)$) if and only if it is a stable model for $bot_U(P)$ (resp. for $e_U(P \setminus bot_U(P), I)$). ∎

Theorem 8.3.5 *(Strong vs. Weak semantics) Let P be a constrained program in which m-atoms never occur in the scope of negation. For each weak answer set M of P, there exists a strong answer set M′ of P such that M′ ⊆ M and M \ M′ is a set of m-atoms.*

Proof. Let M be a weak answer set of P. Then $M = M_r \cup M_m$ is a set of ground atoms and M satisfies the properties AS1, AS2, AS3, AS4.

Let $K_{11}(M), \ldots, K_{1m_1}(M), \ldots, K_{n1}(M), \ldots, K_{nm_n}(M)$ be the subsets of M such that, for each $1 \leq i \leq n$ and $1 \leq j \leq m_i$, $K_{ij}(M) = \{A_i(\vec{a}_i^j, \vec{b}) : A_i(\vec{a}_i^j, \vec{b}) \in M_m$ is a ground instance of $A_i(\vec{a}_i^j, \vec{X}_c)\}$. Note that no $K_{ij}(M)$ is empty because M satisfies the property AS4.

If there exists at least one couple (i, j) ($1 \leq i \leq n$ and $1 \leq j \leq m_i$) such that the set $K_{ij}(M)$ has cardinality greater than one, then let $ma \in M$ be a ground m-atom belonging to $K_{ij}(M)$. Note that, by construction, ma must belong to only one of the sets $K_{11}(M), \ldots, K_{1m_1}(M), \ldots, K_{n1}(M), \ldots, K_{nm_n}(M)$. Because ma is not the unique element of $K_{ij}(M)$, then $M' = M \setminus \{ma\}$ must satisfy the property AS4.

Moreover M' satisfies the property AS3. In fact, for each constraint $(\leftarrow \vec{L}) \in ground(C(P))$, either ma does not occur in $\vec{L}$, and then the value of each L_i is the same in M' than M, or ma occurs in $\vec{L}$ and so $(\leftarrow \vec{L})$ contains one more literal false in M' than M because negative m-literals do not occur in $\vec{L}$.

Moreover M' and M have the same r-literals and then M' satisfies also the properties AS1 and AS2. Then M' is a weak answer set of P as M is.

By iterating the same process starting from M', a set M^* can be obtained such that all sets $K_{11}(M^*), \ldots, K_{1m_1}(M^*), \ldots, K_{n1}(M^*), \ldots, K_{nm_n}(M^*)$ contain only one element. For the same reasons of M', M^* is still a weak answer set of P and $M \setminus M^*$ is a set of m-atoms by construction. Note that M^* is a strong answer set of P because it satisfies also the property AS5. ∎

Note that the assumption on negative m-atoms is satisfied by running example.

Corollary 8.3.6 *Under the hypothesis of Theorem 8.3.5, the strong answer sets of P are the minimal weak answer sets of P.*

Proof. Corollary follows from Theorem 8.3.5 and from the fact that a strong answer set is also a weak answer set. ∎

Corollary 8.3.7 *Under the hypothesis of Theorem 8.3.5, the strong and weak skeptical semantics of P (i.e., the intersection of the strong, resp. weak answer sets) coincide.*

Proof. Let M_w be the skeptical weak answer set for P and M_s be the skeptical strong answer set, then $M_w = \bigcap_j M_{w_j}$ (where each M_{w_j} is a weak answer set for P) and $M_s = \bigcap_i M_{s_i}$ (where each M_{s_i} is a strong answer set for P). By Theorem 8.3.5, for each M_{w_j} there exists a M_{s_i} s.t. $M_{s_i} \subseteq M_{w_j}$. This means that each M_{w_j} in $\bigcap_j M_{w_j}$ contains a M_{s_i} occurring in $\bigcap_i M_{s_i}$. Moreover, each M_{s_i} occurs in $\bigcap_j M_{w_j}$ because a strong answer set is also a weak answer set. Then, by set theory, $\bigcap_j M_{w_j} = \bigcap_i M_{s_i}$. ∎

In the light of the above corollaries, we can focus on the strong semantics, which is a way of computing a "representative" class of answer sets.

8.4 Computing strong answer sets

In this section I present a nondeterministic algorithm, that we introduced in [11], for computing strong answer sets. The actual implementation used in the experiments is derived from the nondeterministic algorithm by adding backtracking.

This algorithm can be applied to constrained programs where mixed predicates have only positive occurrences. More general approaches require further work (cf. Section 8.7).

This algorithm computes *strong kernels*, that is, compact representations of a (potentially large) set of strong answer sets.

Definition 8.4.1 *1. A* strong completion *of a set of ground atoms I is a set $I \cup J$ such that:*

- *J is a set of ground m-atoms;*
- *for each free m-atom $A(\vec{X}_r, \vec{X}_c)$ and each vector of r-constants $\vec{a}$ of the appropriate length, $I \cup J$ contains exactly one instance of $A(\vec{a}, \vec{X}_c)$.*

2. A strong kernel *of a constrained program P is a set of ground atoms K which has at least one strong completion, and such that all the strong completions of K are strong answer sets of P.*

In general, K is the intersection of exponentially many strong answer sets of P. Since *all* strong completions of K are strong answer sets, it is trivial to generate any particular answer set including K, given K itself.

The algorithm that integrates answer set solving and constraint solving is formulated in terms of a generic answer set solver and a generic constraint solver. The former, called ASGen, takes as input a regular program P and a set of ground literals S. Intuitively, ASGen is an incremental solver, and S is the previous partial attempt at constructing an answer set for P. The solver may either fail to further extend S to an answer set of P, or it may return a refined attempt S'. So ASGen is assumed to enjoy following formal properties:

1. ASGen(P, S) returns either NULL or a set S' of ground literals consistent with P.

2. If ASGen(P, S) returns a set S' then $S \subset S'$.

3. If ASGen(P, S) returns a complete set S' then S' is an answer set of P; here, by *complete* I mean that each ground literal occurs in S', either positively or negatively.

4. ASGen is nondeterministically complete, that is for each answer set S of P there exists an integer $n \geq 0$ such that at least one computation of ASGen$^n(P, \emptyset)$ returns S.

As usually, when I write $\text{ASG}\textsc{en}^n(P, \emptyset)$ I mean:

$$\begin{aligned}\text{ASG}\textsc{en}^0(P, \emptyset) &= \emptyset \\ \text{ASG}\textsc{en}^n(P, \emptyset) &= \text{ASG}\textsc{en}(P, \text{ASG}\textsc{en}^{n-1}(P, \emptyset)).\end{aligned}$$

Note that this formulation is compatible with virtually any strategy for interleaving the answer set construction and constraint solving. Note also that as a special case, ASGEN may immediately return complete sets (upon success) like Smodels.

The only requirements on the constraint solver are that it should be sound and nondeterministically complete for each set of c-clauses χ. In other words, all substitutions σ returned by the constraint solver should be solutions of χ (i.e., $\chi\sigma$ should be satisfiable), and for each solution σ of χ, there should be a computation that returns σ.

The constraint solver is applied to a partially evaluated version of the constraints. To specify the partial evaluation procedure some auxiliary notation is needed.

For each constraint $c = \leftarrow B$, I denote by $\mathsf{reg}(c)$, $\mathsf{con}(c)$, and $\mathsf{mix}(c)$, respectively, the collections of regular, constrained and mixed literals belonging to B.

I say that a substitution γ is *r-grounding* if and only if γ replaces each r-variable with a ground r-term and leaves the other variables unchanged.

Definition 8.4.2 *The* partial r-evaluation *of a set of constraints C with respect to a set of ground literals S, denoted by $\mathsf{PE}(C, S)$, is defined by*

$$\mathsf{PE}(C, S) = \{(\leftarrow \mathsf{mix}(c), \mathsf{con}(c))\gamma \mid c \in C,\ \gamma \textit{ r-grounding, and } \mathsf{reg}(c)\gamma \subseteq S\}.$$

Note that the members of $\mathsf{PE}(C, S)$ contain no r-atoms and no r-variables, because the former have been simplified away and the latter have been replaced with r-constants. Note also that in this process some constraints may disappear, as $\mathsf{reg}(c)$ may match no literals in S. Intuitively, S is to be provided by the answer set solver.

The constraint processing algorithm applies to a *normalized* version of $\mathsf{PE}(C, S)$, denoted by $\mathsf{PE}^n(C, S)$, satisfying the following properties:

N1 No m-literal occurring in $\mathsf{PE}^n(C, S)$ contains two or more occurrences of the same variable;

Moreover, for all free m-atoms $A(\vec{X}_r, \vec{X}_c)$,

N2 If both $A(\vec{a}, \vec{y}_c)$ and $A(\vec{a}, \vec{z}_c)$ occur in $\mathsf{PE}^n(C, S)$, then $\vec{y}_c = \vec{z}_c$.

N3 If both $A(\vec{a}, \vec{y}_c)$ and $A(\vec{b}, \vec{z}_c)$ occur in $\mathsf{PE}^n(C, S)$ and $\vec{a} \neq \vec{b}$, then $\vec{y}_c$ and $\vec{z}_c$ have no variables in common.

Note that condition N2 is the opposite of the classic standardization apart approach. N2 and N3 together require the vectors of c-variables to be in one-to-one correspondence with the vectors of regular arguments. Condition N1 can be fulfilled by introducing equations $X_i = X_j$ in $\mathsf{con}(c)$ when needed. Condition N2 and N3 can be fulfilled by variable renaming.

Example 8.4.3 ([11].) *In the running example, whenever the set S contains the pair $o(a_1, 1)$, $o(a_i, 2)$, constraint (8.3) yields the partially evaluated constraint*

$$\leftarrow time(1, T1), time(2, T2), |T2 - T1| < 3.$$

After normalization, and assuming this particular constraint has not been modified, for all the atoms $time(1, X)$ occurring in $\mathsf{PE}^n(C(P), S)$, I have $X = T1$. In this way—roughly speaking—any solution to the constraints is forced to fulfill the property (8.2) of strong semantics.

Now soundness and completeness for Algorithm 1 can be proved.

Theorem 8.4.4 *If a non-failed run of Algorithm 1 returns a set of literals K, then K is a strong kernel of P.*

Proof. Let K be a set returned by a non-failed run of Algorithm 1.

In order to prove that K is a strong kernel of P, it has to be proved that for each set of m-atoms J, if $K \cup J$ is a strong completion of K then $K \cup J$ is a strong answer set of P. That is, it is needed to prove that $K \cup J$ satisfies the properties AS1, AS2, AS3, AS4, AS5, when $K \cup J$ satisfies the properties of the Definition 8.4.1 of strong completion.

If a run r of the algorithm returns a set K then $K = S \cup M(C)\sigma$ where S is a stable model of $R(P)$ and $M(C)\sigma$ is a set of ground m-atoms. Then $K \cup J = S \cup M(C)\sigma \cup J$ satisfies the properties AS1 (because $M(C)\sigma \cup J$ is still a set of ground m-atoms) and AS2.

Suppose that $K \cup J$ does not satisfy the property AS3. Then there exists a constraint $c = (\leftarrow \vec{L}) \in ground(C(P))$ such that all literals L_i in $\vec{L}$ are true in $K \cup J$. If $c = (\leftarrow \vec{L}) \in ground(C(P))$ then there exists a constraint $c' \in C(P)$ and a ground substitution $\gamma = \gamma_r \gamma_c$ of c' such that $c = c'\gamma$ and γ_r is r-grounding. If $\vec{L}$ is true in $K \cup J$ then $reg(c) \subseteq S$ and then $\leftarrow (mix(c'))\gamma_r, (con(c'))\gamma_r \in PE(C(P), S)$. Because

Algorithm 1

CASPSOLVER (P)

```
1:  Inputs: P = R(P) ∪ C(P): a constrained program with no negative m-literals.
2:  Outputs: either a strong kernel of P or FAIL
3:  begin
4:  S := ∅;
5:  loop
6:     S := ASGEN(R(P), S);
7:     if S = NULL then
8:        FAIL;
9:     else
10:       C := PE^n(C(P), S);
11:       if ⋀_{c∈C} ¬con(c) has no solution then
12:          FAIL;
13:       else if S is complete then
14:          choose a solution σ of ⋀_{c∈C} ¬con(c);
15:          Let M(C) be the set of mixed literals in C;
16:          return S ∪ M(C)σ;
17: end
```

$con(c) = (con(c'))\gamma = (con(c'))\gamma_r\gamma_c$ and $mix(c) = (mix(c'))\gamma = (mix(c'))\gamma_r\gamma_c$ are true in $K \cup J$, then γ_c is not a solution of $\neg(con(c'))\gamma_r$. Then the solution σ of $\bigcup_{c\in C} \neg \mathsf{con}(c)$ chosen at the step 14 of the algorithm cannot be factorized in $\sigma = \sigma_1\gamma_c\sigma_2$ (where σ_1 and σ_2 are substitution possibly empty). Consequently, $mix(c) = (mix(c'))\gamma_r\gamma_c$ cannot be added to K at the step 16, while $(mix(c'))\gamma_r\sigma$ is added to K. Because, by hypotheses, $K \cup J$ is a strong completion of K then $(mix(c'))\gamma_r\gamma_c$ cannot belong neither to J. So $mix(c)$ is false in $K \cup J$ and this is a contradiction. Then $K \cup J$ satisfies the property AS3.

By the definition of strong completion, $K \cup J$ satisfies also the properties AS4 and AS5. Consequently $K \cup J$ is a strong answer set of P. ∎

Theorem 8.4.5 *For each strong answer set M of P there exists a run of Algorithm 1 that returns a strong kernel $K \subseteq M$.*

Proof. By the Definition 8.3.3, if M is a strong answer set of P then $M = M_r \cup M_m$ and M satisfies the properties AS1, AS2, AS3, AS4, AS5. According to properties AS1 and AS2, M_r is a stable model of $R(P)$. Then, by the properties of $ASGen$, there exists a set of runs, RUN, of the algorithm that execute with success the test at the step 13 on the set M_r. For each $r \in RUN$, if r does not return FAIL, then r returns a set $K = M_r \cup M(C)\sigma$ that, by the soundness of the algorithm, is a strong kernel of P.

Now, it is only to prove that there always exists an $r \in RUN$ that at the step 14 chooses a solution σ of $\bigcup_{c\in C} \neg \mathsf{con}(c)$ such that $K \subseteq M$. From $M = M_r \cup M_m$ and $K = M_r \cup M(C)\sigma$ follows that $K \subseteq M$ if and only if $M(C)\sigma \subseteq M_m$.

So it is needed to prove that there must always exist a solution σ such that $M(C)\sigma \subseteq M_m$. If such a substitution σ exists then σ can be nondeterministically chosen by a run $r \in RUN$.

Let M_{cfree} be a set of all r-grounded m-atoms of $C(P)$. Then $M_m = M_{cfree}\gamma$ where γ is a ground substitution of M_{cfree} such that for each $A(\vec{a}, \vec{X'})$ and $A(\vec{a}, \vec{X''})$, $A(\vec{a}, \vec{X'})\gamma = A(\vec{a}, \vec{X''})\gamma$. Immediately follows that $M(C) \subseteq M_{cfree}$. γ can be always factorized in $\gamma = \sigma\rho$ where σ is a ground substitution of $M(C)$. Then $M(C)\sigma \subseteq M_{cfree}\sigma\rho$, but it is also need that σ is a solution of $\bigcup_{c\in C} \neg \mathsf{con}(c)$.

Suppose that σ is not a solution of $\bigcup_{c\in C} \neg \mathsf{con}(c)$. Then there exists a constraint $c \in C$ such that $(con(c))\sigma$ is true in M. Then $mix(c) \in M(C)$, because $c \in C$, and $mix(c)\sigma \in M_m$, because $M(C) \subseteq M_{cfree}$. By construction of C, there exists a constraint $c' \in ground(C(P))$ such that $reg(c') \subseteq S$ and $mix(c') = (mix(c))\sigma$ and $con(c') = (con(c))\sigma$. This implies that the constraint c' is not true in M because

its body is true in M, but this is a contradiction because M is a strong answer set of P.

Then there exists a solution σ of $\bigcup_{c \in C} \neg \mathsf{con}(c)$ such that $M(C)\sigma \subseteq M_m$. ∎

8.5 The CASP prototype

The CASP prototype is an implementation of Algorithm 1, based on the answer set solver Smodels [58], that computes a strong kernel for any input constrained program.

CASP is meant to be an exploratory prototype that allow us to exploit the potential interleaving of answer set solving and constraint solving, supported by Algorithm 1, optimizing the process of incrementally computing strong kernels made by the answer set solver and the constraint solver: at each iteration, as the answer set solver returns a partial attempt S' at constructing an answer set for $R(P)$, the constraint processor computes a solution γ' for constraints in $\mathsf{PE}^n(C(P), S')$ starting from the solution γ of $\mathsf{PE}^n(C(P), S)$ computed at previous iteration with respect to previous attempt S of the answer set solver, and backtracking if no solution γ' can be built extending γ.

CASP consists of a script CaspScript that first computes a grounding of $R(P)$ by lparse and then runs a C++ program that implements the interleaving between Smodels and a GNU Prolog constraint logic program with finite domains, that implements steps 10-16 of Algorithm 1. In case of failure (step 12), CaspScript does not always fail; if $R(P)$ has more (partial) stable models, the C++ program feeds the next one to the Prolog module.

The finite domain (FD) constraint solver of GNU Prolog is an instance of the Constraint Logic Programming scheme introduced by Jaffar and Lassez in 1987 [39] and is based on the CLP(FD) framework [23]. Constraints are defined on FD variables and solved by means of arc-consistency (AC) techniques [38]. Arc consistency is not a complete inference mechanism; it ensures only that all solutions (if any) are in the current variable domains. In general, some variable assignments over the current domains are not solutions. Therefore, a final solution generation and checking phase is needed. In many cases, though, the domains produced by arc consistency are tight enough to speed up significantly the computation of solutions.

Figure 8.1: Morning Problem and Car-Pool Problem

Morning Problem	
Smodels (lparse)	**CASP**
> 11m57.341s	0m0.333s

Car-Pool Problem	
Smodels (lparse)	**CASP**
> 11m4.211s	0m0.466s

8.6 Experimental Results

I experimented CASP system over three common types of problems: planning, scheduling and combinatorial problems. The tests have been run on a Pentium(R) M processor 1.5GHz, with 1MB cache and 1.5GB core memory. I started from two very simple planning problems, the *morning problem* and the *car-pool problem* [25].

Morning Problem: Coffee and toast are to be prepared and they are to be ready within two minutes of each other. Coffee is to be brewed for 3-5 minutes and toast is to be toasted for 2-4 minutes. If you take shower in 5-8 minutes and get dressed in 5 minutes, how can you be ready to go by 8:20?

Car-Pool Problem: John goes to work either by car (30-40 minutes), or by bus (at least 60 minutes). Fred goes to work either by car (20-30 minutes), or in a car-pool (40-50 minutes). Today John left home between 7:10 and 7:20, and Fred arrived between 8:00 and 8:10. Moreover, John arrived at work about 10-20 minutes after Fred left home. Is the information in the story consistent? Is it possible that John took the bus, and Fred used the car-pool? What are the possible times at which Fred left home?

Even if these two problems might seem apparently harmless in both cases, as shown in Figure 8.1, after more than ten minutes the front-end lparse of Smodels had not yet terminated the grounding phase (the main reasoning process was not yet reached) while the CASP system returns a solution in less than half a second. Moreover, note that in these two problems there are some temporal domains but

they do not cover more than eight hours and twenty minutes and these domains have only minute granularity.

Then I tested CASP system on a planning problem of significant interest. Programs similar to this example have been used in the USA Advisor project, related to NASA missions [4, 59], and for protocol verification [1]. In both cases memory requirements happened to cause problems.

USA Advisor Problem: A given number of steps are to be done executing at each step one of the two possible actions, an action can be executed more than once. Is it possible to fix a starting time for each step so that the following conditions hold?

- one starting time point has to be assigned to each step,
- successive steps have to start in successive time points,
- the total time needed to execute all steps has an upper bound,
- the first step cannot starts before a fixed time point,
- each of the two actions takes a given time to be terminated,
- actions cannot overlap one each other.

The results of this test are shown in Figure 8.2, where fails that are reported give us how many times the Smodels module in the CASP system has done backtrack. It can be noted that for any instance of the problem, CASP system takes less time than Smodels even if Smodels has been run over a temporal domain of 60 time units, that is the equivalent of a minute, while CASP has been run over a temporal domain of 6.000.000 time units, that is the equivalent of about 70 days.

A simple scheduling problem has then tested.

Scheduling Problem: A set of machines is available to execute a given set of jobs. Each job is computable only on some machines and for each of them a computation time is defined. As usual, on each machine one job at a time can run and a nonpreemptive scheduling algorithm has to be implemented, that is when a job starts it frees the machine only if it has terminated its task. Is it possible to assign to each job a start time and a machine on which it is executed such that the following conditions hold?

- on a machine cannot be executed three jobs such that the execution time of one of them is smaller than the sum of the execution times needed for the other two jobs (intuitively this prevents that smaller jobs have to wait for bigger jobs before they can start),

Figure 8.2: USA Advisor Problem

USA Advisor Problem		
Instance	**Smodels**	**CASP**
	time={0..60}	time={0..6000000}
step={0..1}	0m2.728s	0m0.567s (no fail)
step={0..2}	0m11.272s	0m0.472s (no fail)
step={0..3}	0m23.193s	0m0.868s (no fail)
step={0..4}	1m4.565s	0m1.120s (no fail)
step={0..5}	1m40.183s	0m1.182s (no fail)
step={0..6}	2m34.899s	0m1.405s (no fail)
step={0..7}	5m4.367s	0m38.437s (1 fail)
step={0..8}	7m10.597s	1m20.540s (2 fail)
step={0..9}	9m37.138s	2m15.227s (3 fail)
step={0..10}	12m58.458s	3m16.754s (5 fail)

Figure 8.3: Scheduling Problem

Scheduling Problem		
Instance	**Smodels**	**CASP**
	time={1..60}	time={1..864000}
5 jobs	0m4.791s	0m0.847s
9 jobs	0m23.490s	0m2.908s
17 jobs	> 29m28.875s	1m37.581s

- the total time needed to execute all jobs has an upper bound.

The results of this test are in Figure 8.3. Only for small instances of the problem, Smodels takes less time than CASP while for bigger sizes of the input CASP terminates in a time rather smaller than which Smodels takes.

At the last I have tested the CASP system over two well-known combinatorial problems: n-Queens and Ramsey numbers problems.

n-Queens Problem: How can you place n queens on an $n \times n$ chess board so that no two queens attack each other? Two queens are said to attack each other if they are in the same row or in the same column or in the same diagonal.

Ramsey Numbers Problems: The Ramsey number $R(k,m)$ is the least integer

such that however we color the edges of the complete graph (clique) with n vertexes using two colors red and blue, there is a red clique with k vertexes (a red k-clique) or a blue clique with m vertexes (a blue m-clique). Ramsey numbers exist for all pairs of positive integers k and m. Given a graph with n vertexes and two integer k and m, is it $n < R(k, m)$? That is, is there a coloring such that neither a red k-clique nor a blue m-clique exists?

The results of these tests reported in Figures 8.4 and 8.5 are very different one each other.

The computational times of CASP system over the n-queens problem are very good. In fact, if the number of queens is small (5-10 queens) Smodels terminates before than CASP does, while when the number of queens increases then, as the graphic in Figure 8.4 shows, the time that Smodels takes increases more than the computational time of CASP does.

Instead, over the second combinatorial problem, the Ramsey numbers problem, the performance of CASP is not very good. In fact it takes always more time than Smodels to return the result and, as shown in Figure 8.5, both in the test over the Ramsey number $R(4, 4)$ and in the test over the Ramsey number $R(4, 6)$ there is a point after which the performance of CASP suddenly degrades.

From the tests reported in this section it can be deduced that the CASP system and constrained programs are very suited for representing planning and scheduling problems and for reasoning on them where often big domains, as well as temporal domains, are to be represented so that answer set solvers as Smodels might have some difficulties to reason on them. Note that in all planning problems exploited in this section, there are big temporal domains but only few elements of these domains have a role in the solution. This means that of all ground rules in the ground instance of logic programs that implement these problems only few rules are applicable and the constraint solving module of the CASP system drops all of them without any grounding process is needed.

Over combinatorial problems, instead, the performance of CASP is not always very good above all when the most of search space of solutions has to be visited.

Another important limitation of CASP system is in expressing minimization and maximization statements. These statements are, in fact, not supported by the constrained part of a constrained program. Again, even if Smodels supports these statements, it is not possible to put them in the regular part of a constrained program because during the interleaving of Smodels with the constraint solver, Smodels is not be able to make backtrack for computing an alternative stable model for regular rules when these statements occur.

Figure 8.4: n-Queens Problem

n-Queens Problem		
Instance	**Smodels**	**CASP**
5 queens	0m0.134s	0m0.313s
10 queens	0m0.223s	0m0.347s
15 queens	0m0.496s	0m0.411s
20 queens	0m5.110s	0m0.972s
25 queens	8m29.227s	0m0.973s
30 queens	> 46m17.707s	1m14.999s
35 queens	—	5m59.358s

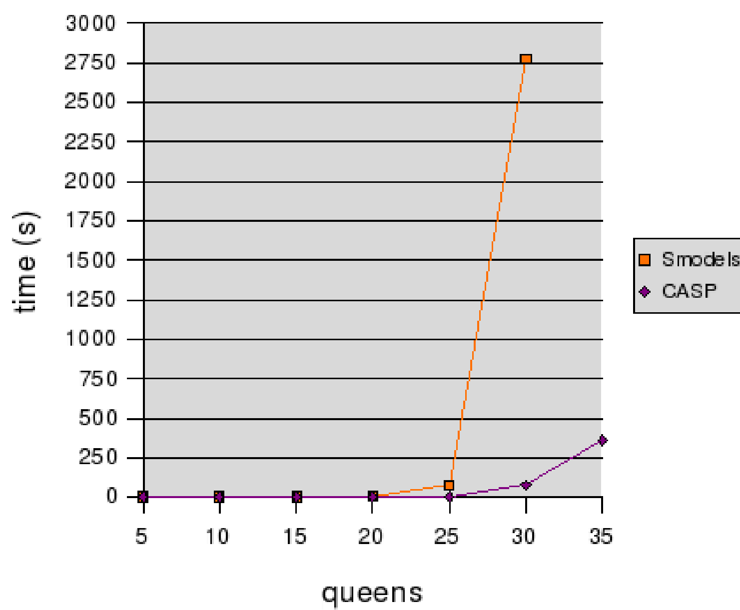


Figure 8.5: Ramsey Numbers Problem

Ramsey Numbers Problem		
Instance	**Smodels**	**CASP**
4—4		
5 vertexes	0m0.084s	0m0.229s
6 vertexes	0m0.076s	0m0.306s
7 vetexes	0m0.089s	0m0.449s
8 vertexes	0m0.109s	> 41m50.618s
9 vertexes	0m0.143s	—
10 vertexes	0m1.743s	—
4—6		
5 vertexes	0m0.060s	0m0.394s
6 vertexes	0m0.080s	0m0.310s
7 vetexes	0m0.094s	0m0.559s
8 vertexes	0m0.117s	0m0.815s
9 vertexes	0m0.164s	> 40m37.431s

8.7 Conclusions

The experimental results show that the integration of answer set programming and constraint solving techniques may significantly enhance the applicability range of ASP. A simple planning and scheduling problem can be naturally formulated and solved, while one of the most powerful state-of-the-art answer set solvers cannot even reach the main reasoning phase. The method that I have presented in this chapter shares with constraint logic programming frameworks the ability of returning answers that may be compact representations of exponentially many distinct problem solutions, each of which can be easily extracted from the answer.

This work can be extended along several directions.

I mentioned that constrained programs are basically a subclass of weight constraint programs. It may be possible to extend the class of weight constraints supported by this approach, e.g., by using different bounds (e.g., mixing weak and strong semantics), and by dropping the requirement that for all free *m*-atoms A and all vector of *r*-constants $\vec{a}$, answer sets must contain at least one instance of $A(\vec{a}, \vec{X}_c)$. Many of the results presented in this chapter can be adapted under the assumption that for all distinct weight constraints $l_1\{A_1\}u_1$ and $l_2\{A_2\}u_2$ in a program, A_1 and A_2 are not unifiable.

Moreover, it would be nice to support negative mixed literals. Unfortunately, this approach cannot be easily adapted; the solutions I have explored so far require blind grounding over constrained domains, which is exactly what should be avoided.

Bibliography

[1] L. C. Aiello and F. Massacci. Verifying security protocols as planning in logic programming. *ACM Trans. Comput. Log.*, 2(4):542–580, 2001.

[2] K. R. Apt. Introduction to Logic Programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Model and Semantics, pages 495–574. Elsevier, Amsterdam and The MIT Press, Cambridge, 1990.

[3] K. R. Apt and H. A. Blair. Arithmetic classification of perfect models of stratified programs. In *ICLP/SLP*, pages 765–779, 1988.

[4] M. Balduccini, M. Gelfond, R. Watson, and M. Nogueira. The usa-advisor: A case study in answer set planning. In Eiter et al. [29], pages 439–442.

[5] F. Bancilhon, D. Maier, Y. Sagiv, and J. D. Ullman. Magic sets and other strange ways to implement logic programs. In *PODS*, pages 1–15. ACM, 1986.

[6] C. Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, Cambridge, 2003.

[7] S. Baselice. Integrazione di tecniche di Answer Set Programming e Constraint Solving. Tesi di laurea, Università degli studi di Napoli Federico II, Naples, Italy, October 2004.

[8] S. Baselice, P. A. Bonatti, and G. Criscuolo. On finitely recursive programs. In V. Dahl and I. Niemelä, editors, *ICLP*, volume 4670 of *Lecture Notes in Computer Science*, pages 89–103. Springer, 2007.

[9] S. Baselice, P. A. Bonatti, and M. Faella. On interoperable trust negotiation strategies. In *POLICY*, pages 39–50. IEEE Computer Society, 2007.

[10] S. Baselice, P. A. Bonatti, and M. Gelfond. A preliminary report on integrating of answer set and constraint solving. In M. De Vos and A. Provetti, editors, *Answer Set Programming*, volume 142 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2005.

[11] S. Baselice, P. A. Bonatti, and M. Gelfond. Towards an integration of answer set and constraint solving. In M. Gabbrielli and G. Gupta, editors, *ICLP*, volume 3668 of *Lecture Notes in Computer Science*, pages 52–66. Springer, 2005.

[12] P. A. Bonatti. Prototypes for reasoning with infinite stable models and function symbols. In Eiter et al. [29], pages 416–419.

[13] P. A. Bonatti. Reasoning with infinite stable models. In B. Nebel, editor, *IJCAI*, pages 603–610. Morgan Kaufmann, 2001.

[14] P. A. Bonatti. Resolution for skeptical stable model semantics. *J. Autom. Reasoning*, 27(4):391–421, 2001.

[15] P. A. Bonatti. Reasoning with infinite stable models ii: Disjunctive programs. In P. J. Stuckey, editor, *ICLP*, volume 2401 of *Lecture Notes in Computer Science*, pages 333–346. Springer, 2002.

[16] P. A. Bonatti. Reasoning with infinite stable models. *Artif. Intell.*, 156(1):75–111, 2004.

[17] P. A. Bonatti and P. Samarati. Logics for authorization and security. In *Logics for Emerging Applications of Databases*, pages 277–323, 2003.

[18] M. Cadoli, F. M. Donini, and M. Schaerf. Is intractability of nonmonotonic reasoning a real drawback? *Artif. Intell.*, 88(1-2):215–251, 1996.

[19] D. Chan. Constructive negation based on the completed database. In *ICLP/SLP*, pages 111–125, 1988.

[20] P. Cholewinski, V. W. Marek, A. Mikitiuk, and M. Truszczynski. Experimenting with nonmonotonic reasoning. In *ICLP*, pages 267–281, 1995.

[21] S. Citrigno, T. Eiter, W. Faber, G. Gottlob, C. Koch, N. Leone, C. Mateis, G. Pfeifer, and F. Scarcello. The dlv system: Model generator and advanced frontends (system description). In *WLP*, pages 0–, 1997.

[22] K. L. Clark. Negation as failure. In *Logic and Data Bases*, pages 293–322, 1977.

[23] P. Codognet and D. Diaz. Compiling constraints in clp(fd). *J. Log. Program.*, 27(3):185–226, 1996.

[24] L. Console, D. T. Dupré, and P. Torasso. On the relationship between abduction and deduction. *J. Log. Comput.*, 1(5):661–690, 1991.

[25] R. Dechter, I. Meiri, and J. Pearl. Temporal constraint networks. *Artif. Intell.*, 49(1-3):61–95, 1991.

[26] S. Decorte, D. De Schreye, and M. Fabris. Exploiting the power of typed norms in automatic inference of interargument relations, 1994.

[27] P. M. Dung. Negations as hypotheses: An abductive foundation for logic programming. In *ICLP*, pages 3–17, 1991.

[28] T. Eiter, W. Faber, C. Koch, N. Leone, and G. Pfeifer. Dlv - a system for declarative problem solving. *CoRR*, cs.AI/0003036, 2000.

[29] T. Eiter, W. Faber, and M. Truszczynski, editors. *Logic Programming and Nonmonotonic Reasoning, 6th International Conference, LPNMR 2001, Vienna, Austria, September 17-19, 2001, Proceedings*, volume 2173 of *Lecture Notes in Computer Science*. Springer, 2001.

[30] T. Eiter and G. Gottlob. On the computational cost of disjunctive logic programming: Propositional case. *Ann. Math. Artif. Intell.*, 15(3-4):289–323, 1995.

[31] T. Eiter, G. Gottlob, and N. Leone. Abduction from logic programs: Semantics and complexity. *Theor. Comput. Sci.*, 189(1-2):129–177, 1997.

[32] K. Eshghi and R. A. Kowalski. Abduction compared with negation by failure. In *ICLP*, pages 234–254, 1989.

[33] F. Fages. Consistency of Clark's completion and existence of stable models. *Methods of Logic in Computer Science*, 1:51–60, 1994.

[34] M. Fitting. A kripke-kleene semantics for logic programs. *J. Log. Program.*, 2(4):295–312, 1985.

[35] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *ICLP/SLP*, pages 1070–1080, 1988.

[36] M. Gelfond and V. Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Comput.*, 9(3/4):365–386, 1991.

[37] M. L. Ginsberg and D. E. Smith. Possible worlds and the qualification problem. In *AAAI*, pages 212–217, 1987.

[38] P. Van Hentenryck, Y. Deville, and C. Teng. A generic arc-consistency algorithm and its specializations. *Artif. Intell.*, 57(2-3):291–321, 1992.

[39] J. Jaffar and M. J. Maher. Constraint logic programming: A survey. *J. Log. Program.*, 19/20:503–581, 1994.

[40] V. Kagan, A. Nerode, and V. S. Subrahmanian. Computing definite logic programs by partial instantiation. *Ann. Pure Appl. Logic*, 67(1-3):161–182, 1994.

[41] V. Kagan, A. Nerode, and V. S. Subrahmanian. Computing minimal models by partial instantiation. *Theor. Comput. Sci.*, 155(1):157–177, 1996.

[42] A. C. Kakas, R. A. Kowalski, and F. Toni. Abductive logic programming. *J. Log. Comput.*, 2(6):719–770, 1992.

[43] K. Kunen. Negation in logic programming. *J. Log. Program.*, 4(4):289–308, 1987.

[44] V. Lifschitz. Computing circumscription. In *IJCAI*, pages 121–127, 1985.

[45] V. Lifschitz and H. Turner. Splitting a logic program. In *ICLP*, pages 23–37, 1994.

[46] V. Lifschitz and H. Turner. Splitting a Logic Program. In *Proceedings of the 12th International Conference on Logic Programming, Kanagawa 1995*, MIT Press Series Logic Program, pages 581–595. MIT Press, 1995.

[47] J. W. Lloyd. *Foundations of Logic Programming, 1st Edition*. Springer, 1984.

[48] V. W. Marek, A. Nerode, and J. B. Remmel. The stable models of a predicate logic program. *J. Log. Program.*, 21(3):129–153, 1994.

[49] V. W. Marek and J. B. Remmel. On the expressibility of stable logic programming. In Eiter et al. [29], pages 107–120.

[50] V. W. Marek and M. Truszczynski. Autoepistemic logic. *J. ACM*, 38(3):588–619, 1991.

[51] J. McCarthy. Programs with common sense. In *Proceedings of the Teddington Conference on the Mechanization of Thought Processes*, pages 75–91, London, 1959. Her Majesty's Stationary Office.

[52] J. McCarthy. Circumscription - a form of non-monotonic reasoning. *Artif. Intell.*, 13(1-2):27–39, 1980.

[53] J. McCarthy. Applications of circumscription to formalizing common-sense knowledge. *Artif. Intell.*, 28(1):89–116, 1986.

[54] D. V. McDermott. Nonmonotonic logic ii: Nonmonotonic modal theories. *J. ACM*, 29(1):33–57, 1982.

[55] D. V. McDermott and J. Doyle. Non-monotonic logic i. *Artif. Intell.*, 13(1-2):41–72, 1980.

[56] R. C. Moore. Semantical considerations on nonmonotonic logic. *Artif. Intell.*, 25(1):75–94, 1985.

[57] R. T. Ng and X. Tian. Incremental algorithms for optimizing model computation based on partial instantiation. Technical report, University of British Columbia, Vancouver, BC, Canada, Canada, 1994.

[58] I. Niemelä and P. Simons. Smodels - an implementation of the stable model and well-founded semantics for normal lp. In J. Dix, U. Furbach, and A. Nerode, editors, *LPNMR*, volume 1265 of *Lecture Notes in Computer Science*, pages 421–430. Springer, 1997.

[59] M. Nogueira, M. Balduccini, M. Gelfond, R. Watson, and M. Barry. An a-prolog decision support system for the space shuttle. In I. V. Ramakrishnan, editor, *PADL*, volume 1990 of *Lecture Notes in Computer Science*, pages 169–183. Springer, 2001.

[60] A. Di Pierro, M. Martelli, and C. Palamidessi. Negation as instantitation: A new rule for the treatment of negation in logic programming. In *ICLP*, pages 32–45, 1991.

[61] D. Poole. Explanation, prediction: an architecture for default, abductive reasoning. *Computational Intelligence*, 5:97–110, 1989.

[62] R. Reiter. On closed world data bases. In *Logic and Data Bases*, pages 55–76, 1977.

[63] R. Reiter. A logic for default reasoning. *Artif. Intell.*, 13(1-2):81–132, 1980.

[64] R. Reiter. A theory of diagnosis from first principles. *Artif. Intell.*, 32(1):57–95, 1987.

[65] J. S. Schlipf. The expressive powers of the logic programming semantics. *J. Comput. Syst. Sci.*, 51(1):64–86, 1995.

[66] P. Simons. Extending the stable model semantics with more expressive rules. In M. Gelfond, N. Leone, and G. Pfeifer, editors, *LPNMR*, volume 1730 of *Lecture Notes in Computer Science*, pages 305–316. Springer, 1999.

[67] P. Simons, I. Niemelä, and T. Soininen. Extending and implementing the stable model semantics. *Artif. Intell.*, 138(1-2):181–234, 2002.

[68] R. F. Stärk. A complete axiomatization of the three-valued completion of logic programs. *J. Log. Comput.*, 1(6):811–834, 1991.

[69] L. Sterling and E. Y. Shapiro. *The Art of Prolog - Advanced Programming Techniques, 2nd Ed.* MIT Press, 1994.

[70] T. Syrjanen. Lparse 1.0 user's manual, February 06 2001.

[71] T. Syrjänen. Omega-restricted logic programs. In Eiter et al. [29], pages 267–279.

[72] H. Turner. Splitting a default theory. In *AAAI/IAAI, Vol. 1*, pages 645–651, 1996.

[73] M. H. van Emden and R. A. Kowalski. The semantics of predicate logic as a programming language. *J. ACM*, 23(4):733–742, 1976.