Universidade Nova de Lisboa
Faculdade de Ciências e Tecnologia
Departamento de Informática

Tese de Doutoramento

Doutoramento em Informática

Every normal logic program has a 2-valued semantics: theory, extensions, applications, implementations

Alexandre Miguel dos Santos Martins Pinto

Julho de 2011

Universidade Nova de Lisboa
Faculdade de Ciências e Tecnologia
Departamento de Informática

Tese de Doutoramento

Every normal logic program has a 2-valued semantics: theory, extensions, applications, implementations

Alexandre Miguel dos Santos Martins Pinto

Orientador:
Prof. Doutor Luís Moniz Pereira

Trabalho apresentado no âmbito do Doutoramento em Informática, como requisito parcial para obtenção do grau de Doutor em Informática.

Julho de 2011

I almost wish I hadn't
gone down that rabbit-hole
—and yet—and yet—
it's rather curious, you know,
this sort of life!
Alice

Alice in Wonderland
Lewis Carrol, 1865

To Graça

In memoriam Jorge

Acknowledgements

I must start by thanking my wife, Graça, for her love, her support, all the encouragement and patience she had during these years, and for truly being my partner in our Strongly Connected Component. Also, I must thank my parents for all they have done for me, and Isaura Coelho for her support.

I especially thank Prof. Dr. Luís Moniz Pereira, my supervisor, for lots of reasons which include, but are not limited to: sharing with me his innovative thoughts and excitement, proposing me the challenge to pursue this new and exciting world of open possibilities, his availability for discussions even when he had his schedule already filled up, for introducing me to several researchers within the right timing, for teaching me the good practices of serious scientific research and allowing me to do it with him, and also for his friendship and pleasant company.

Mário Abrantes, who is now on his PhD work with my supervisor as well, is pursuing some of the avenues projected as Future Work in this thesis. I must thank him for the critical discussions we have had, the problems he identified, the examples he offered.

I acknowledge the FCT-MCTES (Fundação para a Ciência e Tecnologia do Ministério da Ciência, Tecnologia e Ensino Superior) for giving me the PhD grant (no. SFRH / BD / 28761 / 2006) that paid for my survival during these last four years.

Internationally, I thank Michael Gelfond for his comments on my previously published works and, more importantly, for sharing with me his vision of what a semantics should be: it turned out to be a priceless guideline for my writing of this thesis. I am also indebted to Terrance Swift for the countless hours he spent with me on phone calls about the insides of XSB Prolog, and all the times we worked together face-to-face. Bob Kowalski also helped me immensely with his precious expertise and advice on various semantics issues.

I acknowledge the *Departamento de Informática da Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa*, and its *Centro de Inteligência Artificial (CENTRIA)* for giving me work conditions. Anabela Duarte and Sandra Raínha are also present in my thank-you list for helping me with bureaucratic papers and arranging my travels during these last years.

Finally, a word of appreciation must go to Ana Sofia Gomes for proofreading my thesis, and to Marco Alberti, Davide d'Alimonte, Elsa Carvalho, Marco Correia, and Jorge Cruz for their friendship, and for the interesting, and sometimes a bit crazy and funny, discussions we had over so many lunches together.

Abstract

After a very brief introduction to the general subject of Knowledge Representation and Reasoning with Logic Programs we analyse the syntactic structure of a logic program and how it can influence the semantics. We outline the important properties of a 2-valued semantics for Normal Logic Programs, proceed to define the new Minimal Hypotheses semantics with those properties and explore how it can be used to benefit some knowledge representation and reasoning mechanisms.

The main original contributions of this work, whose connections will be detailed in the sequel, are:

- The Layering for generic graphs which we then apply to NLPs yielding the Rule Layering and Atom Layering — a generalization of the *stratification* notion;
- The Full shifting transformation of Disjunctive Logic Programs into (highly non-stratified) NLPs;
- The Layer Support — a generalization of the classical notion of support;
- The Brave Relevance and Brave Cautious Monotony properties of a 2-valued semantics;
- The notions of Relevant Partial Knowledge Answer to a Query and Locally Consistent Relevant Partial Knowledge Answer to a Query;
- The Layer-Decomposable Semantics family — the family of semantics that reflect the above mentioned Layerings;
- The Approved Models argumentation approach to semantics;
- The Minimal Hypotheses 2-valued semantics for NLP — a member of the Layer-Decomposable Semantics family rooted on a minimization of positive hypotheses assumption approach;
- The definition and implementation of the Answer Completion mechanism in XSB Prolog — an essential component to ensure XSB's WAM full compliance with the Well-Founded Semantics;
- The definition of the Inspection Points mechanism for Abductive Logic Programs;

- An implementation of the Inspection Points workings within the Abdual system [21]

We recommend reading the chapters in this thesis in the sequence they appear. However, if the reader is not interested in all the subjects, or is more keen on some topics rather than others, we provide alternative reading paths as shown below.

1-2-3-4-5-6-7-8-9-12	Definition of the Layer-Decomposable Semantics family and the Minimal Hypotheses semantics (1 and 2 are optional)
3-6-7-8-10-11-12	All main contributions – assumes the reader is familiarized with logic programming topics
3-4-5-10-11-12	Focus on abductive reasoning and applications

Keywords: Layering Logic Programs, Semantics, Layer-Decomposable semantics, Minimal Hypotheses semantics, Inspection Points, Argumentation

Sumário

Após uma breve introdução ao tema geral de Representação do Conhecimento e Raciocínio com Programas Lógicos, analisamos a estrutura sintáctica de um programa lógico e o modo como ela pode influenciar a semântica deste. Discutimos algumas das propriedades importantes e úteis de uma semântica a 2-valores para Programas Lógicos Normais (PLNs), prosseguimos definindo a nova semântica de Hipóteses Mínimas que goza das propriedades identificadas, e exploramos de que forma esta pode ser usada beneficiando alguns mecanismos de representação de conhecimento e raciocínio.

As principais contribuições originais deste trabalho, cujas ligações serão detalhadas mais à frente, são:

- O princípio de Distribuição em Camadas de grafos genéricos, que depois aplicamos a PLNs dando origem à Distribuição em Camadas de Regras e à Distribuição em Camadas de Átomos, sendo esta última uma generalização da conhecida noção de Estratificação;
- A aplicação da transformação de Translação Total a Programas Lógicos Disjuntivos produzindo Programas Lógicos Normais altamente não-estratificados;
- A noção de Suporte de Camada — uma generalização da noção clássica de Suporte;
- As propriedades Relevância Crédula e Monotonia Cautelosa Crédula de semânticas a 2-valores;
- As noções de Resposta Relevante com Conhecimento Parcial a Pergunta e Resposta Relevante e Localmente Consistente com Conhecimento Parcial a Pergunta;
- A família de Semânticas Decomponíveis em Camadas — a família das semânticas que reflectem as supracitadas Distribuições em Camadas;
- A semântica, baseada numa abordagem argumentativa, denominada Modelos Aprovados;
- A semântica de Hipóteses Mínimas para PLN — um membro da família das Semânticas a 2-valores Decomponíveis em Camadas radicada na abordagem de minimização de adopção de hipóteses positivas;

- A definição e implementação do mecanismo de Completação de Resposta no XSB Prolog — uma componente essencial para assegurar a completa reificação da Semântica Bem-Fundada na WAM do XSB Prolog;
- A definição do mecanismo de Pontos de Inspecção para Programas Lógicos Abdutivos;
- Uma implementação de Pontos de Inspecção dentro do sistema Abdual [21]

Recomendamos que se leiam os capítulos desta tese na sequência em que aparecem. Contudo, se o leitor não estiver interessado em todas as matérias, ou se estiver interessado em alguns tópicos mais do que noutros, apresentamos de seguida alguns percursos de leitura alternativos.

1-2-3-4-5-6-7-8-9-12	Definição da família de Semânticas Decomponíveis em Camadas e da Semântica de Hipóteses Mínimas (1 e 2 são opcionais)
3-6-7-8-10-11-12	Todas as principais contribuições – assume que o leitor está familiarizado com os tópicos da programação em lógica
3-4-5-10-11-12	Foco em raciocínio abdutivo e aplicações

Palavras-chave: Distribuição em Camadas de Programas Lógicos, Semânticas, Semânticas Decomponíveis em Camadas, Semântica de Hipóteses Mínimas, Pontos de Inspecção, Argumentação

Contents

List of Figures

Preface

Complex knowledge can be formalized with logic, and it is of great convenience for us to have computers handling Knowledge Representation and Reasoning (KRR) tasks based on logic. There are many formalisms for KRR available and used. Computational Logic (CL) is but one of those, which has been used for several decades now as a means to represent some rational aspects of human thinking, and in fact [131] argues strongly in favor of the use of CL for KRR. There are, nonetheless, several kinds of logics within CL in use for KRR. One such formalism, resorting to Non-Monotonic Logics with Default Negation, is Logic Programming. Logic Programs (LPs), provide a declarative and flexible tool for KRR, having the expressive power of a Universal Turing Machine, and therefore allowing for the expression of complex knowledge.

The complement to Knowledge Representation is Reasoning, and when dealing with Computational Logic we are especially interested in rational logical reasoning, such as deduction and abduction — the latter being akin to hypotheses assumption in order to explain observations or to construct hypothetical scenarios entailing some desired goal.

In the past few decades Logic Programming has gained an increasing important role for KRR, namely for its flexibility and applicability to knowledge-intensive tasks, and their seamless combination, as diverse as search problems, planning, constraint solving, abductive reasoning, argumentation modelling and, more recently, to Semantic Web applications, to name just a few.

The history of Logic Programming has also been tied to an ongoing effort to find a *good* semantics for logic programs with varying criteria for evaluating how *good* a semantics is. There exist several approaches to the semantics issue, ranging from 2-valued to 3-valued and even to multi-valued and to fuzzy logics but, by far, the lion's share of the efforts has been concentrated in 2- and 3-valued semantics.

Successive 2-valued semantics proposals have assigned a meaning to ever larger classes of programs, in part because one of the goals of 2-valued semantics is to provide a meaning to *every* Normal Logic Program (NLP). This goal is even more significant in a context where Knowledge Base (KB) updating and merging becomes progressively more common, e.g., among Semantic Web applications — the lack of semantics for an updated and/or merged Semantic Web KB could endanger the robustness of a whole web service. There are 3-valued semantics that ensure that security, but there are yet no 2-valued ones that achieve it while enjoying a number of useful properties.

The traditional 2-valued semantics approach fails to assign a meaning to every NLP

because of the way it handles non-well-founded negation (i.e., cyclic or with infinite descending chains); whereas the 3-valued approach has a clever solution: the *undefined* truth-value. Our new contribution is another step in the ongoing history of 2-valued semantic progress by proposing a 2-valued semantics for *all* NLPs, that successfully deals with non-well-founded negation by generalizing the classical notion of support. Assigning a semantics to non-well-founded negation increases the flexibility in expressing knowledge with NLPs, whilst simultaneously effectively separating the KRR task from the issue of representing Integrity Constraints, which are duly covered by a specific kind of rule not mistakable with a regular NLP rule.

Logic Programs can be used to represent knowledge from any subject domain and at any desired level of detail. We first identify all the structural information of a program, which we dub Layering. The Layering is induced by the interdependencies amongst rules and it is a generalization of the well-known notion of stratification. The notion of Layering is inspired by the works [22, 174, 175, 242], who argue that all fields of knowledge and inquiry — ranging from physics, chemistry to biology, and other sciences — can and should be knit together into a comprehensive worldview. The Layering notion divides knowledge into layers of modules, each with a strong intra-dependency. Modules in a layer may depend on others on layers below, but never reciprocally. We use Layering as a guideline to constraint the space of acceptable 2-valued semantics: what is determined *true* at some layer must not be contradicted by other layers. We do not advocate in favour of a reductionist stance, but certainly of an inter-layer compatibility, leaving room for intra-layer non-determinism.

We next define and propose one particular semantics fitting inside this Layered space of semantics, the Minimal Hypotheses (MH) semantics, rooted in *Ockham's razor* principle of minimality of assumed hypotheses. The semantics for logic programs that have been defined throughout history have insisted on minimality of models. With MH semantics we focus instead, not on minimality of the whole model, but rather on minimality of the hypotheses assumed to produce the model. As Albert Einstein put it

> *"The grand aim of all science is to cover the greatest number of empirical facts by logical deduction from the smallest number of hypotheses or axioms."*

We argue that minimality of hypotheses is what a semantics should strive for, with set-inclusion minimality of whole model, with respect to models of alternative sets of hypotheses, being a possible, but not necessary, consequence of the minimally assumed hypotheses. The MH semantics we propose achieves a delicate equilibrium between being 2-valued, enjoying a number of desirable theoretical properties, also useful for practical applications, and allowing for relatively simple implementations, and not unusual computational complexity.

In the past, some commonly used semantics for logic programs have been analysed from the hypotheses assumption perspective as well, which has a natural parallel in abductive reasoning; but all these approaches consider only the default negated literals as being the assumable hypotheses. This approach works fine in all situations except for non-well-founded negation. To overcome this problem, we generalized the hypotheses assumption perspective to allow the adoption of, not only negative hypotheses, but also of positive ones. Having taken this generalization step we realized that positive hypotheses assumption alone is sufficient to address all situations, i.e., there is no need for both positive and negative hypotheses assumption, and minimization of positive ones is enforced, since this naturally takes care of the usual maximization of negative ones (or default assumptions). This is the reason why we decided to embrace the only positive hypotheses assumption perspective in this thesis.

Finally we show how deductive and abductive reasoning can be interchanged and how to inspect for side-effects of hypotheses assumed in the process of finding a scenario entailing some goal.

This thesis is divided into three parts, corresponding to the three main driving forces behind its motivation: 1) to understand how the structure of a logic program can, and must, influence its semantics (we cover this in Part I and also Chapter 7); 2) to build a semantics for logic programs that complies with the structure of the knowledge represented by the program, enjoys a number of useful properties, and easily allows for both deductive and abductive reasoning mechanisms (we cover this in Part II); and 3) to provide a theoretical means to inspect consequences of assumed hypotheses during abductive reasoning as a way to lay down the foundations for future implementations of practical systems with such inspection capabilities (we cover this in Part III).

Part I, consisting of Chapters 1, 2, 3, and 4 covers the general topic of Logic Programs and their structure. In Chapter 1 we cover the basic abstract notions of Knowledge Representation and Reasoning Methods as well as more practical issues concerning the Scope of Knowledge addressed in various types of reasoning. Knowledge can be viewed as a graph where individual knowledge sentences, or clauses, may depend on others. Chapter 2 takes the general perspective of knowledge as a graph and uses the dependency relation to outline the structure of a knowledge graph. Chapter 3 takes the general graph notions from Chapter 2 and applies them to the particular case of Normal Logic Programs, and compares this approach to other classically used approaches. Chapter 4 overviews two other classes of Logic Programs commonly used for Knowledge Representation and Reasoning, and shows how these can be translated into NLPs thereby focusing on the need for a semantics just to NLPs, as we do not address topics such as paraconsistent reasoning and belief revision.

Part II, consisting of Chapters 5, 6, 7, 8, and 9 covers the topic of semantics for

Normal Logic Programs. In Chapter 5 we introduce the basic concepts for a semantics of Logic Programs; these include the notions of interpretation satisfaction, model, and two orderings of interpretations. In Chapter 6 we peruse the intuitive requirements a "good" semantics should comply with, such as the notion of support and minimality. We also examine a couple of classes of restricted logic programs and syntactic ways to determine their models, and the currently most used semantics for logic programs. Then we outline the motivation behind this thesis, including the formal properties a semantics for logic programs should enjoy (some of the properties listed therein are part and parcel of new contributions of this thesis). Chapter 7 takes the syntactical/structural information of a program, according to Chapter 3, uses it to define a family of semantics for logic programs, and argues that all "good" semantics should be members of this family. Chapter 8 takes a hypotheses assumption perspective to semantics in the spirit of Albert Einstein's quote above concerning "*cover(ing) the greatest number of empirical facts by logical deduction from the smallest number of hypotheses*", and proceeds to define the Minimal Hypotheses (MH) models semantics. We study the MH semantics' properties and show that this hypotheses assumption approach is compatible with the structural approach of Chapter 7. The MH semantics is one of the major contributions of this thesis and, in Chapter 9, we compare it to other semantics and approaches to logic programs (namely, argumentation).

Part III concerns Reasoning with Logic Programs and consists of Chapters 10, and 11. Chapter 10 covers the general topic of abductive and deductive reasoning with logic programs (showing some of their similarities and differences), relates abductive/deductive reasoning with query-answering, and discusses the issue of abductions' side-effects inspection. Chapter 11 describes the implementations of some individual theoretical aspects covered in this thesis. These implementations are prototypical in nature, but nonetheless intended to show the efforts being made towards a reification of a whole logic-based knowledge representation and reasoning system founded upon the novel concepts described herein. This thesis's contributions are mainly of a theoretical nature, yet we also wanted to show that we put som effort into reifying some of the concepts presented herein, thereby opening the way for further contributions and possibly a fully-functional implementation of an abductive/deductive existential query-answering system, with local-knowledge sufficiency as opposed to complete knowledge necessity, plus side-effect inspection capabilities.

Finally, we terminate with Chapter 12 where we draw out the conclusions of this work, overview some of the types of applications that can benefit from our approach, and point to the many branches of future work that stem from this initial stepping platform.

In order to streamline reading, we moved most of the proofs of theorems, corollaries, lemmas, and propositions to Appendix A — Proofs.

Part I

Knowledge Representation with Logic Programs

1 . Introduction

We will not know unless we begin.

HOWARD ZINN

We contemplate Artificial Intelligence as an intelligent problem solving technique which can be broken down into Knowledge Representation and Reasoning. After a very brief glance over the logic approach to Knowledge Representation we reflect on the main Reasoning methods and their scope.

According to [131]

> *"Artificial Intelligence (AI) is the attempt to program computers to behave intelligently, as judged by human standards."*

We usually refer to such intelligent computer programs as rationally thinking *software agents*. Also in [131], the author puts the rational thinking equation in the form:

> Thinking = Knowledge Representation + Problem Solving

It signifies that, in the process of endowing software agents with (artificial) intelligence, we must come up with a means to represent the agent's knowledge about itself and the world it is going to interact with, and also with a set of rational problem solving techniques that rely on that knowledge.

1.1 Knowledge Representation Formalisms

There are a number of possible alternative ways to represent knowledge. In the process of defining and understanding AI we have also learned much about our own human intelligence, the language of human thought and reasoning. In this thesis we follow the

perspective of [131] stating that

> *"Computational Logic has been developed in Artificial Intelligence over the past 50 years or so, in the attempt to program computers to display human levels of intelligence."*

Even within the Computational Logic area there are a variety of specific formalisms that can be used for Knowledge Representation (KR) including, but not limited to, classical logic ([58, 105, 225] amongst many others), description logics ([28, 29, 124]), modal logics ([141, 168]), temporal logics ([97, 240]), but also non-monotonic formalisms like default logic ([42, 213]) and logic programs ([15, 31, 32, 133]).

The last of these (logic programs) is a formalism whereby knowledge is represented in (Horn) clausal form permitting the use of default negation. In [132], the author argues more generally for this use of logic programs for Knowledge Representation and Reasoning (KRR). Logic Programming is the KR formalism we use throughout this thesis and we proffer its details in Chapter 3.

The second part of the right-hand side of the "*thinking* equation" depicted right at the beginning of this introduction is *Problem Solving*. In a KRR context within the framework of logic programs, this is more aptly phrased as *Reasoning Methods*, which we now turn to.

1.2 Reasoning Methods

In [129] the authors make a brief summary of the three different forms of reasoning identified by Peirce [172]:

- ***Deduction*** — *an analytic process based on the application of general rules to particular cases, with the inference of a result.*
- ***Induction*** — *synthetic reasoning which infers the rule from the case and the result.*
- ***Abduction*** — *another form of synthetic inference, but of the case from a rule and a result.*

Peirce further characterized abduction as the "probational adoption of a hypothesis" as explanation for observed facts (results), according to known laws.

> *"It is however a weak kind of inference, because we cannot say that we believe in the truth of the explanation, but only that it may be true".*

Induction is a reasoning method that is rather different from the other two; it lays outside the scope of this thesis and, therefore, we shall only delve into and study deduction and abduction.

1.2.1 Deduction

Deduction is the reasoning method which allows one to entail conclusions from premises. We can formalize this statement as

$$KB \models C$$

where KB is a Knowledge Base (KB), C is the conclusion we deducted from KB, and $\models$ represents the deductive entailment relation. The $\models$ relation is intimately connected to the semantics used over the knowledge base, as it is the semantics that determines what conclusions can and cannot be entailed by KB. In this sense, the $\models$ relation must be parametrized by a semantics S for the KB:

$$KB \models_S C$$

But still, neither this entailment $\models$ relation, nor the semantics S parameter say, in general, anything about the necessity or mere possibility of C given KB and $\models_S$. Since we will be considering logic programs as our KR formalism for KB, the $\models_S$ relation must, somehow, have a correspondence with the interpretations of KB accepted as models by S (informally and roughly speaking, an interpretation of KB is a set of atomic statements of KB believed to be true, and a model is an interpretation that satisfies all the rules in KB — the formal concepts of interpretation, model and their relationship to logic programs are detailed in Chapter 5). Thus, in the logic programming framework, the $KB \models_S C$ statement must be translatable into a relation between C and the model(s) of KB according to S. This poses the question of whether S accepts, in general, only one interpretation as model of KB, or possibly many — and, in parallel, if S guarantees at all the acceptance of at least one interpretation as model.

In this thesis we consider the case where S may, in general, accept several interpretations as alternative models of KB, as this supersedes the "uni-model" semantics case. Under this setting, the $KB \models_S C$ statement can be seen from either a skeptical or credulous perspective, according to the unanimity, or lack thereof, of the models of KB, according to S, concerning C's truth.

1.2.1.1 Skeptical Deduction

Under the logic programming context, skeptic deduction can thus be translated into the unanimity of the models of KB, according to S, concerning C's truth, thereby asserting the necessity of C.

Definition 1.1. Skeptical Entailment. Let KB be a Knowledge Base, C a formula in the same formalism as the one used for KB, S a semantics for KB, and $S_{KB}(M)$ denote that the interpretation M is a model of KB according to S. Then

$$KB \models_S^{Sk} C \Leftrightarrow \forall_M (S_{KB}(M) \Rightarrow M \models_S C)$$

i.e., the Knowledge Base KB skeptically entails the formula C iff C is *true* in every model of KB according to S.

In the logic programming literature, this notion of skeptical entailment is also known as "cautious reasoning". For this reason, we shall henceforth write "cautious reasoning" instead of "skeptical entailment".

1.2.1.2 Credulous Deduction

Analogously to the previous 1.2.1.1 point, under the logic programming context, credulous deduction can be translated into the existence of a model of KB, according to S, entailing C's truth, thereby asserting the possibility of C.

Definition 1.2. Credulous Entailment. Let KB be a Knowledge Base, C a formula in the same formalism as the one used for KB, S a semantics for KB and $S_{KB}(M)$ denote that the interpretation M is a model of KB according to S. Then

$$KB \models_S^{Cr} C \Leftrightarrow \exists_M (S_{KB}(M) \wedge M \models_S C)$$

i.e., the Knowledge Base KB credulously entails the formula C iff C is *true* in some model of KB according to S.

In the logic programming literature, this notion of credulous entailment is also known as "brave reasoning". For this reason, we shall henceforth write "brave reasoning" instead of "credulous entailment".

1.2.1.3 Query answering

Brave reasoning is intimately connected to existential query answering in knowledge bases represented as logic programs to the extent that it also refers to finding if there exists at

least one model of the logic program — according to the chosen semantics — entailing the truth of a user-specified logic formula, usually dubbed the user's "query".

Definition 1.3. Existential Query-answering in a Logic Program. Let P be a logic program representation of the user's Knowledge Base, Q a user-specified query (a conjunction of literals), S a semantics for P, and M an interpretation of P. Then, $S_P^Q(M)$ means that M is an existential answer, according to S, to the user's query Q over P. Formally,

$$S_P^Q(M) \Leftrightarrow \exists_M(S_P(M) \wedge M \models_S Q)$$

The negative counterpart of an answer to Q is an answer to $\neg Q$, i.e., $S_P^{\neg Q}(M)$ which means there is one model M (a possible alternative amongst several other ones) where Q is not *true*. Hence, in M, $\neg Q$ can be inferred. Formally,

$$S_P^{\neg Q}(M) \Leftrightarrow \exists_M(S_P(M) \wedge M \not\models_S Q)$$

One can now easily sketch out the dual Universal Query-answering in a Logic Program definition corresponding to cautious reasoning:

Definition 1.4. Universal Query-answering in a Logic Program. Let P be a logic program representation of the user's Knowledge Base, Q a user-specified query, S a semantics for P and M an interpretation of P. Then, S_P^Q means that Q is universally entailed by P according to semantics S iff

$$S_P^Q \Leftrightarrow \forall_M(S_P(M) \Rightarrow M \models_S Q)$$

The negative counterpart now is $S_P^{\neg Q}$ which means Q is not *true* in any model M of P (according to semantics S), hence $\neg Q$ can be unconditionally inferred. Formally,

$$S_P^{\neg Q} \Leftrightarrow \forall_M(S_P(M) \Rightarrow M \not\models_S Q)$$

With a semantics allowing for several alternative models we can see each of them as a set of hypotheses plus their corresponding necessary conclusions — a semantics allowing for only one model would not allow any hypothesizing freedom.

In [129] the authors argue in favour of viewing default negated literals in bodies of rules as abducibles, or assumable hypotheses. They defend that negation (and in logic programs in particular, default negation) must play a central role in endowing negated elements with eligibility to hypothesization. We take this hypotheses assumption perspective which has also been taken by several semantics for logic programs before, namely, the Stable Models semantics with which we compare our approach in subsequent chapters. There it will become clear how specifically we embed and semantically implement this hypotheses assumption principle.

1.2.2 Induction

Induction usually refers to a type of reasoning that involves producing a general rule from a set of specific facts. It can also be seen as a form of learning or theory-building, in which specific facts are used to create a theory that explains relationships between the facts and allows prediction of future knowledge. In the context of logic programming, Inductive Logic Programming [136, 162, 163] has been a productive research area in the past years. Inductive reasoning lies outside the scope of this thesis as it is an orthogonal issue to the one focused on here: finding a 2-valued semantics, with a specific set of properties, for all NLPs.

1.2.3 Abduction

Abductive reasoning is discussed in Chapter 10 and is commonly understood as hypothesizing plausible reasons sufficient for justifying given observations or supporting desired goals. An abductive problem can be stated in the following way: let KB be a Knowledge Base, Q a goal (also referred to as the *abductive* query), and Δ a set of adoptable (abducible) hypotheses. In this case we say $\delta \subseteq \Delta$ is an *abductive solution* to Q given KB iff $KB \cup \delta \models Q$ and $\delta \models ICs$ where ICs is the set of Integrity Constraints pertaining KB. We may, of course, be also interested in the *side-effects* of abducing δ in KB. In Chapter 10 we introduce an efficient method to check if some literal's truth-value becomes determined as a side-effect of a given abductive solution to a goal.

For centuries, a central guideline of rational scientific reasoning has been the heuristic known as *Ockham's razor*. The Encyclopedia Britannica [1] presents, amongst others, the following formulation of this reasoning principle: *"(. . .)the simplest explanation of an entity is to be preferred."* This principle which has maximal skepticism at its heart, is also known as the *law of economy*, or *law of parsimony* and has been formulated in many different ways including *"the explanation requiring the fewest assumptions is most likely to be correct."* This correlates to abduction, or hypotheses assumption, in the sense that *Ockham's razor* states that we should always strive for minimal sets of hypotheses explaining the observations, or entailing the query.

In turn, hypotheses assumption relates to credulous entailment (or existential query answering) in the sense that, as stated at the end of 1.2.1.3, with "a semantics allowing for several alternative models we can see each of them as a set of hypotheses plus their corresponding necessary conclusions". In the context of semantics for logic programs allowing for more than one model, Abductive and Deductive reasoning end up being almost equivalent as we show and further detail in Chapter 10.

The classical approach to semantics rests on demanding minimality of models, i.e., minimality of the set of hypotheses plus their conclusions; but as far as the *Ockham's razor* principle is concerned, the *law of parsimony* is only focused on the set-inclusion minimality of the hypotheses, letting the orthogonal issue of minimizing their consequences as an optional undertaking. Indeed, there may be non-minimal sets of hypotheses conducive to minimal models when the former might be incompatible amongst themselves. This issue occurs in logic programming due to default negation or integrity constraints, e.g. Example 8.3. Minimality of hypotheses, while resolving away their incompatibility, will be a major issue addressed in this thesis in the context of logic programming. In Chapter 8 we introduce a semantics focused only on minimality of the hypotheses.

1.3 Reasoning Scope

In this thesis we focus on using logic programs as the knowledge representation formalism to encode knowledge bases [131]. Under this setting, and depending on the nature of the problem at hand and the semantics considered, finding an intended solution for a query may require taking into account all the formulas in the KB or, on the other hand, it may suffice to consider only the fraction of the KB relevant to the query formula. In the first case, finding a solution to the query requires the identification of a whole model for the entire KB, whilst in the second case identifying a part of a model might suffice. When solutions to the problem at hand involve by necessity whole models, we say we perform complete knowledge reasoning when finding such solutions. Conversely, partial knowledge reasoning refers to finding parts of model being enough, it being implicit that such part models are extendible to a whole model in the program semantics used.

1.3.1 Complete Knowledge Reasoning

Complete knowledge reasoning pertains to identifying models of the whole knowledge base that conform to some user-specified requirements. From a practical standpoint, this corresponds to whole KB model computation. It is the role of the chosen semantics to determine which interpretations of a KB are accepted as models. Depending on the properties of the envisaged underlying semantics, whole model computation may be the only possibility for any kind of problem solving. In particular, such complete knowledge reasoning may be in general inevitable when the semantics lacks the relevance property. This is discussed in further detail in subsection 6.5.5.

1.3.2 Partial Knowledge Reasoning

Reasoning with large KBs represented as NLPs can be computationally very expensive, especially if the KB can be updated, whether by external agents or through self-updates. When KBs are comprised of rules concerning several fields of knowledge, we can say we have some modularization of the knowledge. This can be the case either when the KB itself is used to represent an ontology plus the set of rules to derive new knowledge, or when the KB is just such a set of rules and is associated with an ontology originating somewhere else, as in [115]. In such cases, it can be quite common for the user to want to perform some form of reasoning concerning just a fragment of the overall knowledge. Whole model computation can then be computationally overwhelming and downright unnecessary. It would also be putting too much of a burden on the user to require her/him to specify the (sub-) module(s) of knowledge to be considered when finding a solution to the query. Again, taking advantage of such a modularization naturally occurring in knowledge can only be afforded when the indicated underlying semantics enjoys the relevance property. Partial knowledge reasoning, which can be seen as partial model computation, using only the fragments of knowledge strictly necessary to find an answer, can have significant impact on real applications's efficiency.

Having introduced some of the general concepts of knowledge representation and reasoning we now take in Chapter 2 a closer look at the syntactic structure of a set of KBs induced by their interdependencies.

---

2 . The Structure of Knowledge

> An intimate Knowledge therefore of the intellectual and moral World is the sole foundation on which a stable structure of Knowledge can be erected.
>
> JOHN ADAMS
> Letter to Jonathan Sewall, 1759

In Chapter 1 we took a brief overview of the Knowledge Representation and Reasoning problem in very general terms. In this chapter we analyze the structural properties of arbitrary sets of (possibly interdependent) Knowledge Bases, assembled from a graph-theoretic perspective, thereby identifying the structure of knowledge. Resulting from such syntactical/structural analysis, we present the Layering notion for directed graphs which we then specialize to the concrete case of NLPs. The notion of Layering we present is one of the end products of our research. In [193, 194, 196, 197] we took several intermediate steps which eventually lead us to the final definition in this chapter.

2.1 Knowledge Graph

In [22], P. W. Anderson says

> *"(. . .) one may array the sciences roughly linearly in a hierarchy (. . .) The elementary entities of science X obey the laws of science Y."*

where *"science Y"* is *more fundamental* than *"science X"*. And he continues

> *"At each stage entirely new laws, concepts, and generalizations are necessary,*

requiring inspiration and creativity to just as great a degree as in the previous one."

Knowledge can be represented in a Knowledge Base as a set of formulas (or sentences). Different KBs can represent knowledge about different domains, each written in a possibly different formalism, and where some KBs may depend on the knowledge in other KBs. In this sense we can think of the whole body of Knowledge as a network, or graph, with KBs as vertices. The structure of knowledge must, therefore, be closely related to the structure of the graph of KBs, to which we now turn to analyze. For self-containment, we very briefly recap the definition of graph in particular, the directed graph variant of which is the one of specific interest here.

Definition 2.1. Directed Graph (adapted from [39]). We say $G = (V, E)$ is a directed graph iff V is the set of *vertices* of G, and $E \subseteq V \times V$ is the set of directed edges of G, where each edge is an ordered pair (v_i, v_j) standing for a directed connection from vertex v_i to vertex v_j. When the graph at hand is not unambiguously determined, we write V_G and E_G to denote, respectively, the set of vertices and the set of edges of graph G. The *vertices* and *edges* will be, henceforth, also alternatively referred to as *nodes* and *arcs*, respectively.

The directed edges in a graph are the central ingredient for the structure of the graph as they can be seen to induce dependency relationships amongst the vertices. We thus turn now to analyze these dependencies.

2.1.1 Dependencies

Definition 2.2. General dependencies in a graph. Let $G = (V, E)$ be a graph. We say vertex $v_j \in V$ *directly depends* on vertex $v_i \in V$ (abbreviated as $v_j \leftarrow v_i$) iff there is an edge (v_i, v_j) in E, and we say vertex $v_m \in V$ *depends* on vertex $v_i \in V$ (abbreviated as $v_m \twoheadleftarrow v_i$) iff there is are edges $(v_i, v_j), (v_j, v_k), \ldots (v_l, v_m)$ in E — i.e., there is a "path" in G from vertex v_i to vertex v_m.

Definition 2.3. Sub-graph. Let $G = (V, E)$ be a graph. We say $G' = (V', E')$ is a sub-graph of G iff $V' \subseteq V$ and $E' \subseteq E$ such that $\forall_{(v_i,v_j)\in E'} v_i \in V' \wedge v_j \in V'$.

Definition 2.4. Relevant sub-graph. Let $G = (V, E)$ be a graph. We say vertex $v_j \in V$ is *relevant* for vertex $v_i \in V$ iff v_i *depends* on v_j. The sub-graph of G relevant for vertex v_i is $Rel_G(v_i) = (RV, RE)$, where $RV = \{v_i\} \cup \{v_j \in V : v_i \twoheadleftarrow v_j\}$, and $RE = \{(v_k, v_l) \in E : v_k \in RV \wedge v_l \in RV\}$. We say that RV is the set of vertices of G relevant for v_i.

We also define the inverse notion, the Influence sub-graph:

Definition 2.5. Influence sub-graph. Let $G = (V, E)$ be a graph. We say vertex $v_j \in V$ *influences* vertex $v_i \in V$ iff v_i *depends* on v_j. The sub-graph of G influenced by vertex v_j is $Infl_G(v_j) = (IV, IE)$, where $IV = \{v_j\} \cup \{v_i \in V : v_i \twoheadleftarrow v_j\}$, and $IE = \{(v_k, v_l) \in E : v_k \in IV \wedge v_l \in IV\}$. We say that IV is the set of vertices of G influenced by v_j.

In general, graphs can also have circular dependencies (paths leading from a vertex back to itself); these give rise to the notions of loops and of Strongly Connected Component ([39]). In graph theory [39] a loop in a graph is usually referred to as a Strongly Connected Subgraph (SCSG). Finding the set of SCSGs of any given graph is known to be of polynomial time complexity [235].

Definition 2.6. Loop. Let $G = (V_G, E_G)$ be a graph. $L = (V_L, E_L)$ is a loop of G iff L is a subgraph of G such that for each pair of vertices $v_i, v_j \in V_L$ there is a path in L from v_i to v_j and a path in L from v_j to v_i — i.e., $v_j \twoheadleftarrow v_i$ and $v_i \twoheadleftarrow v_j$.

Definition 2.7. Strongly Connected Component (adapted from [39]). Let $G = (V_G, E_G)$ be a graph. $S = (V_S, E_S)$ is a Strongly Connected Component (SCC) of G iff S is a *maximal* subgraph of G such that for each pair of vertices $v_i, v_j \in V_S$ there is a path in S from v_i to v_j and a path in S from v_j to v_i — i.e., $v_j \twoheadleftarrow v_i$ and $v_i \twoheadleftarrow v_j$. An SCC is thus just a maximal Loop (Definition 2.6).

The concept of Strongly Connected Component from general Graph Theory [39] considers single nodes not involved in loops as SCCs too.

Definition 2.8. Modules of a Graph. Let $G = (V, E)$ be a directed graph, and let $SCC(G)$ be the set of all SCCs of G. The set $M(G)$ is said to be the set of *modules* of G iff each member of $M(G)$ is the set of vertices of an SCC of G.

$$M(G) = \{V_{scc} : scc \in SCC(G)\}$$

I.e., a module is the set of vertices of an SCC (which may include only one vertex).

Proposition 2.1. ***The set of modules is a partition of the vertices of the graph.*** *Let $G = (V_G, E_G)$ be a graph and $M(G)$ the set of modules of G. Then $M(G)$ is a partition of V_G. I.e.,*

$$V_G = \bigcup_{m \in M(G)} m$$

and

$$\forall_{m_i, m_j \in M(G)} m_i \neq m_j \Rightarrow m_i \cap m_j = \emptyset$$

Definition 2.9. Module dependencies. Let $G = (V, E)$ be a graph, and $M(G)$ the set of modules of G (cf. Definition 2.8). We say module $m_i \in M(G)$ depends on module $m_j \in M(G)$ (abbreviated as $m_i \nleftarrow m_j$) iff there are two vertices $v_i \in m_i$, and $v_j \in m_j$, such that v_i depends on v_j in G.

Proposition 2.2. ***Different modules are non-mutually-dependent.*** *Let G be a graph and $M(G)$ the set of modules of G. Then,*

$$\forall_{\substack{m_i, m_j \in M(G) \\ m_i \neq m_j}} \neg((m_i \nleftarrow m_j) \wedge (m_j \nleftarrow m_i))$$

I.e., all modules are pairwise not mutually dependent.

With the notions of modules of a graph and their respective module dependencies we can now lay down the *Modules Graph* of a graph.

Definition 2.10. Modules Graph. Let $G = (V, E)$ be a directed graph, and $M(G)$ the set of modules of G. We say $MG(G) = (M(G), ME(G))$ is the *modules graph* of G iff $ME(G) = \{(m_i, m_j) : \exists_{v_i \in m_i, v_j \in m_j}(v_i, v_j) \in E\}$.

Theorem 2.1. ***Existence and uniqueness of the modules graph.*** *Let G be a graph. Then there is exactly one* modules graph $MG(G)$ *of G.*

Proof. Trivial from Definitions 2.8, 2.9, 2.10, and Proposition 2.1. □

Proposition 2.3. ***The Modules Graph of a graph G is a Directed Acyclic Graph*[1].** *Let G be a graph, and $MG(G)$ its modules graph. Then, by construction, $MG(G)$ is a Directed Acyclic Graph.*

We now present one of our new contributions: the Graph Layering notion.

Definition 2.11. Graph Layering. Let $G = (V, E)$ be a graph with no infinitely long descending chains of dependencies of vertices. A graph *layering function* $Lf/1$ is a function mapping each vertex $v \in V$ of graph G to a non-zero ordinal such that

$$\forall_{v_1, v_2 \in V} \begin{cases} Lf(v_1) = Lf(v_2) & \Leftarrow & (v_1 \nleftarrow v_2) & \wedge & & (v_2 \nleftarrow v_1) \\ Lf(v_1) > Lf(v_2) & \Leftarrow & (v_1 \nleftarrow v_2) & \wedge & \neg & (v_2 \nleftarrow v_1) \end{cases}$$

The two cases above, which are patently mutually exclusive, leave out independent vertices, i.e., vertices that have no dependencies amongst themselves. According to this definition there is no restriction on which ordinal to assign to each independent vertex in what the other vertices' assignations are concerned.

[1] A Directed Acyclic Graph is a directed graph with no Strongly Connected Components.

A graph layering of graph G is a partition $\ldots, V^i, \ldots$ of V such that V^i contains all vertices v having $Lf(v) = i$. We write $V^{<\alpha}$ as an abbreviation of $\bigcup_{\beta<\alpha} V^\beta$, and $V^{\leq\alpha}$ as an abbreviation of $V^{<\alpha} \cup V^\alpha$, and define $V^0 = V^{\leq 0} = \emptyset$. It follows immediately that $V = \bigcup_\alpha V^\alpha = \bigcup_\alpha V^{\leq\alpha}$, and also that the $\leq$ relation between layers is a total-order in the sense that $V^i \leq V^j$ iff $i \leq j$.

Amongst the several possible graph layerings of a graph G we can always find the least one, i.e., the graph layering with least number of layers, where the ordinals of the layers are the smallest possible, and where the ordinals of $Lf(v)$, for each vertex v, are also the smallest possible, whilst respecting the graph layering function assignments. This least graph layering is easily seen to be unique.
N.B.: In the following, when referring to the graph's "layering", we mean just such least *graph layering.*

Theorem 2.2. ***Existence and uniqueness of*** **least layering** ***for a given graph*** G**.** *Let G be a graph. There is exactly one* least layering *of G.*

In graph theory there are notions and results similar to the layering but, to the best of our knowledge, none is exactly equivalent to it. There is one notion in particular — the *topological sort* — which is quite similar to the layering.

Definition 2.12. Topological sort (Section 22.4 of [62]). A topological sort of a Directed Acyclic Graph $G = (V, E)$ is a linear ordering of all its vertices such that if G contains an edge (u, v), then u appears before v in the ordering. If the graph is not acyclic, then no linear ordering is possible.

A graph's layering is akin to a topological sort albeit these notions do not coincide:

1. The topological sort is defined only for Directed Acyclic Graphs, whereas the Layering is defined for all Directed Graphs not necessarily Acyclic
2. There might be several topological sorts for a graph, whereas theres is only one least layering
3. Each instance of a topological sort assigns different order numbers to vertices that are assigned the same layer ordinal by the least layering, even for DAGs

All knowledge can be represented as a graph with Knowledge Bases as vertices (a *knowledge graph*); the corresponding *modules graph* is thus the intrinsic *structure of knowledge*. Moreover, the *least layering* of such *knowledge graph* complements the *modules graph* by providing ordering information reflecting the dependencies amongst vertices.

Layering is an ordering of modules, it is an ordering of knowledge according to syntactic dependency. Whenever the user has some other specific intended way of ordering knowledge, say, induced by some set of priorities, or ordering preferences, a different preference-based layering might emerge. In this sense, the *least layering* presented above is just the simplest, preference-free, form of layering; it is the *default* layering of knowledge. Nonetheless, we leave open the possibility of some ordering-preferences to be associated with a program, with the intent to specify how its knowledge is to be ordered, i.e., layered. We do not explore this avenue of further possibilities in this thesis, but leave it for future work.

As stated previously, in this thesis we have chosen Normal Logic Programs as the particular Knowledge Representation formalism, hence we now recap the basic definitions of NLPs, and the different kinds of knowledge graphs over NLPs, as well as how layering applies to NLPs.

---

3 . Normal Logic Programs and their Structure

> Logic is not a body of doctrine, but a mirror-image of the world.
>
> Ludwig Wittgenstein

In this chapter we review the basic technical concepts of logic programming to introduce notation and to bring the reader not familiar with these matters moderately up to speed. We cover the notion of logic program, focusing particularly on Normal Logic Programs (NLPs) as they will be the Knowledge Representation formalism we will be using throughout this thesis, and also the concepts of interpretation, classical and minimal model, and their relationship to rules.
Part of the contents of this chapter is based upon some of our previous publications, namely [193, 194, 196, 197], where we took steps towards some of the definitions currently presented here.

3.1 Normal Logic Programs

Logic programs have been used for decades for Knowledge Representation and Reasoning. Intuitively, a logic program consists of a (possibly countably infinite) set of rules, each one connecting a set of finite premises with a conclusion. Some rules may have no premises, and in such case we call them facts. An NLP is just a specific kind of logic program with some particular restrictions on what kinds of premises and conclusions are allowed in rules. We formalize these notions with the definitions below.

3.1.1 Language

Definition 3.1. Normal Logic Program. By an alphabet $\mathcal{A}$ of a language $\mathcal{L}$ we mean a (finite or countably infinite) disjoint set of constants, predicate symbols, and function symbols, with at least one constant. In addition, any alphabet is assumed to contain a countably infinite set of distinguished variable symbols. A term over $\mathcal{A}$ is defined recursively as either a variable, a constant or an expression of the form $f(t_1,\ldots,t_n)$ where f is a function symbol of $\mathcal{A}$, n its arity, and the t_i are terms. An atom over $\mathcal{A}$ is an expression of the form $P(t_1,\ldots,t_n)$ where P is a predicate symbol of $\mathcal{A}$, and the t_i are terms. A literal is either an atom A or its default negation *not* A. We dub default literals (or default negated literals — DNLs, for short) those of the form *not* A. A term (resp. atom, literal) is said ground if it does not contain variables. The set of all ground terms (resp. atoms) of $\mathcal{A}$ is called the Herbrand universe (resp. base) of $\mathcal{A}$. For short we use $\mathcal{H}$ to denote the Herbrand base of $\mathcal{A}$. A Normal Logic Program is a set of rules of the form:

$$H \leftarrow B_1,\ldots,B_n,not\ C_1,\ldots,not\ C_m, \text{ with } (m,n \geq 0 \text{ and finite})$$

where H, the B_i and the C_j are atoms. Without loss of generality, NLPs can be seen as possibly infinite (countable) sets of ground rules corresponding to all the possible instantiations of variables of the non-ground version of rules. Thus, from this point onwards, in order to simplify subsequent definitions and results, we consider only ground NLPs, ground rules, ground literals, and ground atoms. In conformity with the standard convention, we write rules of the form $H \leftarrow$ also simply as H (known as "facts"). An NLP P is called definite if none of its rules contain default literals. H is the head of the rule r, denoted by $head(r)$, and $body(r)$ denotes the set $\{B_1,\ldots,B_n,not\ C_1,\ldots,not\ C_m\}$ of all the literals in the body of r.

Definition 3.2. Constrained Normal Logic Program. Let P be an NLP and $\mathcal{C}$ a set of rules of the form

$$\bot \leftarrow B_1,\ldots,B_n,not\ C_1,\ldots,not\ C_m, \text{ with } (m,n \geq 0 \text{ and finite})$$

with a non-empty body. Such rules with head $\bot$ (or "$falsum$") are also known as a type of Integrity Constraints (ICs), specifically *denials*. We say $P \cup \mathcal{C}$ is a Constrained Normal Logic Program (CNLP). Although $\bot$ (or "$falsum$") may occur in $\mathcal{P}$ as head of IC rules, it is not part of $\mathcal{H}_P$ (the atoms of P) and therefore it cannot appear in bodies of any rules.

Example 3.1. ***Constrained Normal Logic Program.*** *Let $\mathcal{P}$ be*

$$\begin{array}{rcl} \bot & \leftarrow & a \\ a & \leftarrow & not\ b \\ b & \leftarrow & not\ a \end{array}$$

The Integrity Constraint rule $\bot \leftarrow a$ is what turns this program into a Constrained NLP. The other two rules — for a and for b — are regular NLP rules.

3.1.1.1 Useful notation

We abuse the '*not* ' default negation notation applying it to non-empty sets of literals too: we write *not* S to denote $\{not\ s : s \in S\}$, and confound *not not* $a \equiv a$. When S is an arbitrary, non-empty, set of literals — $S = \{B_1, \ldots, B_n, not\ C_1, \ldots, not\ C_m\}$ — we use the following notation

- S^+ denotes the set $\{B_1, \ldots, B_n\}$ of positive literals in S
- S^- denotes the set $\{not\ C_1, \ldots, not\ C_m\}$ of negative literals in S
- $|S| = S^+ \cup (not\ S^-)$ denotes the set $\{B_1, \ldots, B_n, C_1, \ldots, C_m\}$ — the atoms of S

As expected, we say a set of literals S is consistent iff $S^+ \cap |S^-| = \emptyset$. Also, we will apply the notation above for arbitrary sets of literals S to bodies of rules: $body(r)^+$, $body(r)^-$, and $|body(r)|$. We also write $heads(P)$ to denote the set of heads of non-IC rules of a (possibly constrained) program P, i.e., $heads(P) = \{head(r) : r \in P\} \setminus \{\bot\}$, and $facts(P)$ to denote the set of facts of P — $facts(P) = \{head(r) : r \in P \wedge body(r) = \emptyset\}$.

We assume that the alphabet $\mathcal{A}$ used to write a program P consists precisely of all the constants, predicates and function symbols that explicitly appear in P. By Herbrand universe (resp. base) of P we mean the Herbrand universe (resp. base) of $\mathcal{A}$. By grounded version of a Normal Logic Program P we mean the (possibly infinite) set of ground rules obtained from P by consistently substituting in all possible combined ways each of the variables instances in P by elements of its Herbrand universe.

These base concepts of Alphabet, Language, Rule, Herbrand Base, and Normal Logic Program are taken (with some minor changes) from [15][1].

[1] In this work we restrict ourselves to Herbrand interpretations and models. For the subject of semantics based on non-Herbrand models, and solutions to the problems resulting from always keeping to Herbrand models see e.g. [109, 133, 210]. Thus, without loss of generality (cf. [206]), we envisaged a Normal Logic Program P standing for its grounded version.

3.2 The Structure of Normal Logic Programs

"Ordnung ist das halbe Leben."
("*Order is half of life.*")

GERMAN SAYING

In Chapter 2 we assumed that the structure of knowledge is the *modules graph*. We now turn to analyze some of the different approaches to identifying the knowledge structure behind an NLP.

The traditional approach considers the atom dependency graph of an NLP.

Definition 3.3. Atom graph. Let P be an NLP. $DG(P)$ is the atom dependency (directed) graph of P where the atoms of P are the vertices of $DG(P)$, and there is a directed edge from a vertex A to a vertex B iff there is a rule in P with head B such that A appears in its body.

But as the author of [65] puts it, relating the Dependency Graph with the Answer Set semantics [113, 142]

> "*it is well-known, the traditional Dependency Graph (DG) is not able to represent programs under the Answer Set semantics: in fact, programs which are different in syntax and semantics, have the same Dependency Graph.*"

In this thesis we define a new family of semantics for NLPs, generalizing the Stable Models semantics, so the "traditional" atom Dependency Graph is also not enough for our purposes.

In the literature, we may find the rule graph, introduced in [82].

Definition 3.4. Rule graph (Definition 3.8 of [82]). Let P be a reduced negative NLP (i.e., there are only negative literals in the bodies of rules). $RG(P)$ is the rule graph of P where the rules of P are the nodes of $RG(P)$, and there is an arc from a node r_1 to a node r_2 iff the head of rule r_1 appears in the body of the rule r_2.

But, as the author of [65] says,

> "*in our opinion it would be difficult to define any practical programming methodology on the basis of the rule graph, since it does not graphically distinguish among cases which are semantically very different.*"

This sentence from [65] assumes not only that the underlying semantics is the Stable Models, but also that the arcs in the rule graph are supposed to contain all the semantic information of the program. Besides, the rule graph as defined in [82], presupposes reduced negative programs. As we shall see below, our semantic approach to rule graphs considers its structural information as a crucial necessary part in determining the semantics of the program, but not a sufficient one. Thus, we will be able to *define a practical programming methodology on the basis of the rule graph*, plus other semantic constructs, namely, hypotheses assumption, as we will see in the sequel.

The following definition extends the rule graph one (Definition 3.4) in the sense that it is applicable to all NLPs and not just to reduced negative logic programs.

Definition 3.5. Complete Rule Graph. Let $\mathcal{P} = P \cup \mathcal{C}$ be a Constrained NLP. The *complete rule graph* of $\mathcal{P}$ (denoted by $CRG(\mathcal{P})$) is the directed graph whose rules of $\mathcal{P}$ are the vertices of $CRG(\mathcal{P})$ and there is a directed edge from vertex r_1 to vertex r_2 iff the head of rule r_1 appears in the body of the rule r_2. I.e., $CRG(\mathcal{P}) = (\mathcal{P}, DP)$ where $DP = \{(r_1, r_2) : r_1, r_2 \in \mathcal{P} \wedge head(r_1) \in |body(r_2)|\}$.

Definition 3.6. Dependencies in a program. We say a rule r_2 directly depends on r_1 (written as $r_2 \leftarrow r_1$) iff there is a direct edge in $CRG(\mathcal{P})$ from r_1 to r_2. We say r_2 depends on r_1 ($r_2 \twoheadleftarrow r_1$) iff there is a directed path in $CRG(\mathcal{P})$ from r_1 to r_2.

Naturally, we again consider the other combinations of (direct) dependencies amongst atoms and rules. We also use the same graphical notation ($\leftarrow, \twoheadleftarrow$)amongst atoms, and between atoms and rules to denote (direct, indirect) dependency.

Rule r directly depends on atom a iff $a \in |body(r)|$; and r depends on a iff either r directly depends on atom a or r depends on some rule r' which directly depends on a. I.e.,

$$\begin{array}{lcl} r \leftarrow a & \Leftrightarrow & a \in |body(r)| \\ r \twoheadleftarrow a & \Leftrightarrow & r \leftarrow a \vee \exists_{r' \in P}(r \twoheadleftarrow r' \wedge r' \leftarrow a) \end{array}$$

We say an atom a directly depends on rule r iff $head(r) = a$; and a depends on r iff either a directly depends on r or a directly depends on some rule r' such that r' depends on r. I.e.,

$$\begin{array}{lcl} a \leftarrow r & \Leftrightarrow & head(r) = a \\ a \twoheadleftarrow r & \Leftrightarrow & a \leftarrow r \vee \exists_{r' \in P}(a \leftarrow r' \wedge r' \twoheadleftarrow r) \end{array}$$

We say an atom b directly depends on atom a iff a appears (possibly default negated) in the body of a rule with head b, and b depends on a iff either b directly depends on a, or b directly depends on some rule r which depends on a. I.e.,

$$\begin{array}{lcl} b \leftarrow a & \Leftrightarrow & \exists_{r \in P}(b \leftarrow r \wedge r \leftarrow a) \\ b \twoheadleftarrow a & \Leftrightarrow & b \leftarrow a \vee \exists_{r \in P}(b \leftarrow r \wedge r \twoheadleftarrow a) \end{array}$$

Applying the *relevant* (Definition 2.4) and *influence* (Definition 2.5) sub-graph notions to the particular case of (C)NLPs (cf. Definition 3.2) under their respective CRGs (Definition 3.5) we have:

Definition 3.7. Sub-program Relevant for Rule(s). Let $\mathcal{P} = P \cup \mathcal{C}$ be a Constrained NLP, and $CRG(\mathcal{P})$ its CRG. We say rule $r_j \in \mathcal{P}$ is *relevant* for rule $r_i \in \mathcal{P}$ iff r_i *depends* on r_j. We write $Rel_{\mathcal{P}}(r_i)$ to denote the set of such r_j, i.e.,

$$Rel_{\mathcal{P}}(r_i) = \{r_j \in \mathcal{P} : r_i \nleftarrow r_j\}$$

$Rel_{\mathcal{P}}(r_i)$ is thus the set of vertices of $CRG(\mathcal{P})$ relevant for r_i.

We abuse this notation by writing $Rel_{\mathcal{P}}(R)$, where R is a subset of rules of $\mathcal{P}$, to denote the set of rules that R depends on, i.e.,

$$Rel_{\mathcal{P}}(R) = \bigcup_{r \in R} Rel_{\mathcal{P}}(r)$$

Likewise, the inverse Influence sub-program notion:

Definition 3.8. Sub-program Influenced by Rule(s). Let $\mathcal{P} = P \cup \mathcal{C}$ be a Constrained NLP, and $CRG(\mathcal{P})$ its CRG. We say rule $r_j \in \mathcal{P}$ *influences* rule $r_i \in \mathcal{P}$ iff r_i *depends* on r_j. We write $Infl_{\mathcal{P}}(r_j)$ to denote the set of such r_i, i.e.,

$$Infl_{\mathcal{P}}(r_j) = \{r_i \in \mathcal{P} : r_i \nleftarrow r_j\}$$

$Infl_{\mathcal{P}}(r_j)$ is thus the set of vertices of $CRG(\mathcal{P})$ influenced by r_j.

We abuse this notation by writing $Infl_{\mathcal{P}}(R)$, where R is a subset of rules of $\mathcal{P}$, to denote the set of rules that R influences, i.e.,

$$Infl_{\mathcal{P}}(R) = \bigcup_{r \in R} Infl_{\mathcal{P}}(r)$$

We also apply these notions with the focus on atoms:

Definition 3.9. Sub-program Relevant for Atom. Let $\mathcal{P} = P \cup \mathcal{C}$ be a Constrained NLP, and $CRG(\mathcal{P})$ its CRG. We say rule $r \in \mathcal{P}$ is *relevant* for atom $a \in \mathcal{H}_{\mathcal{P}}$ iff a *depends* on r. We write $Rel_{\mathcal{P}}(a)$ to denote the set of such r, i.e.,

$$Rel_{\mathcal{P}}(a) = \{r \in \mathcal{P} : a \nleftarrow r\}$$

Likewise, the inverse Influence sub-graph notion:

Definition 3.10. Sub-program Influenced by Atom. Let $\mathcal{P} = P \cup \mathcal{C}$ be a Constrained NLP, and $CRG(\mathcal{P})$ its CRG. We say atom $a \in \mathcal{H}_\mathcal{P}$ *influences* rule $r \in \mathcal{P}$ iff r *depends* on a. We write $Infl_\mathcal{P}(a)$ to denote the set of such r, i.e.,

$$Infl_\mathcal{P}(a) = \{r \in \mathcal{P} : r \twoheadleftarrow a\}$$

It will also be useful to define the notion of sub-program relevant for a conjunction of literals and sub-program influenced by a conjunction of literals.

Definition 3.11. Sub-program Relevant for a Conjunction of Literals. Let $\mathcal{P} = P \cup \mathcal{C}$ be a Constrained NLP, $CRG(\mathcal{P})$ its CRG, and $Q = a_1 \wedge \ldots \wedge a_n \wedge not\ b_1 \wedge \ldots \wedge not\ b_m$ a conjunction of literals (with all the a_i and b_j in $\mathcal{H}_P$, at least one literal, and $n, m \geq 0$) which we call a *query*. We write SQ to denote the set of all the literals in Q, i.e., $SQ = \{a_1, \ldots, a_n, not\ b_1, \ldots, not\ b_m\}$.

We define $Rel_\mathcal{P}(Q)$, the Sub-program of $\mathcal{P}$ Relevant for the Conjunction of Literals (or Query) Q, as the set of all rules relevant for all the atoms corresponding to the literals in Q, i.e.,

$$Rel_\mathcal{P}(Q) = \bigcup_{q \in |SQ|} Rel_\mathcal{P}(q)$$

Definition 3.12. Sub-program Influenced by a Conjunction of Literals. Let $\mathcal{P} = P \cup \mathcal{C}$ be a Constrained NLP, $CRG(\mathcal{P})$ its CRG, and $Q = a_1 \wedge \ldots \wedge a_n \wedge not\ b_1 \wedge \ldots \wedge not\ b_m$ a conjunction of literals (with all the a_i and b_j in $\mathcal{H}_\mathcal{P}$, at least one literal, and $n, m \geq 0$) which we call a *query*. SQ denotes the set of all the literals in Q, i.e., $SQ = \{a_1, \ldots, a_n, not\ b_1, \ldots, not\ b_m\}$.

We define $Infl_\mathcal{P}(Q)$, the Sub-program of $\mathcal{P}$ Influenced by the Conjunction of Literals (or Query) Q, as the set of all rules influenced by atoms corresponding to the literals in Q, i.e.,

$$Infl_\mathcal{P}(Q) = \bigcup_{q \in |SQ|} Infl_\mathcal{P}(q)$$

Alongside with the graph perspective of logic programs is the classical stratification notion which is usually associated with the atom dependency graph.

Definition 3.13. Stratification [210]. A program P is stratified if and only if it is possible to decompose the set S of all predicates of P into disjoint sets $S_1, \ldots, S_r$, called strata, so that for every clause $A \leftarrow B_1, \ldots, B_m, not\ C_1, \ldots, not\ C_n$, in P, where A's, B's and C are atoms, we have that:

$$\forall_i stratum(B_i) \leq stratum(A)$$
$$\forall_j stratum(C_j) < stratum(A)$$

where $stratum(A) = i$, if the predicate symbol of A belongs to S_i. Any particular decomposition $\{S_1, \ldots, S_r\}$ of S satisfying the above conditions is called a stratification of P.

The stratification notion fails to capture all the structural information of a program since it focuses only on the atoms — it misses the specific dependencies for each particular rule. Moreover, there are cases of programs which have no stratification whatsoever, in particular, programs with loops.

The Layering notion we developed (Definition 2.11), when applied to the Complete Rule Graph (Definition 3.5) of a CNLP $\mathcal{P}$, covers all programs and captures all the structural information in each one.

Definition 3.14. Rule Layering. Let $\mathcal{P} = P \cup \mathcal{C}$ be a CNLP with no infinitely long descending chains of dependency and $CRG(\mathcal{P})$ its *complete rule graph* (Definition 3.5). A rule *layering function* $Lf/1$ of $\mathcal{P}$ is a function mapping each vertex of $CRG(\mathcal{P})$ (a rule r of $\mathcal{P}$) to a non-zero ordinal such that

$$\forall_{r_1, r_2 \in \mathcal{P}} \begin{cases} Lf(r_1) = Lf(r_2) & \Leftarrow & (r_1 \twoheadleftarrow r_2) & \wedge & & (r_2 \twoheadleftarrow r_1) \\ Lf(r_1) > Lf(r_2) & \Leftarrow & (r_1 \twoheadleftarrow r_2) & \wedge & \neg & (r_2 \twoheadleftarrow r_1) \end{cases}$$

A rule layering of $\mathcal{P}$ is thus a partition $\ldots, \mathcal{P}^i, \ldots$ of $\mathcal{P}$ such that $\mathcal{P}^i$ contains all rules r having $Lf(r) = i$. We write $\mathcal{P}^{<\alpha}$ as an abbreviation of $\bigcup_{\beta<\alpha} \mathcal{P}^\beta$, and $\mathcal{P}^{\leq\alpha}$ as an abbreviation of $\mathcal{P}^{<\alpha} \cup \mathcal{P}^\alpha$, and define $\mathcal{P}^0 = \mathcal{P}^{\leq 0} = \emptyset$. It follows immediately that $\mathcal{P} = \bigcup_\alpha \mathcal{P}^\alpha = \bigcup_\alpha \mathcal{P}^{\leq\alpha}$, and also that the $\leq$ relation between layers is a total-order in the sense that $\mathcal{P}^i \leq \mathcal{P}^j$ iff $i \leq j$.

Amongst the several possible rule layerings of $\mathcal{P}$ we can always find the least one, i.e., the rule layering with least number of layers, where the ordinals of the layers are the smallest possible, and where the ordinals of $Lf(r)$, for each rule r, are also the smallest possible, whilst respecting the rule layering function assignments. This least rule layering is easily seen to be unique.
N.B.: In the following, when referring to the program's "layering", we mean just such least *rule layering. Likewise, there is also a least stratification. We cover the relationship between strata and layers in the sequel.*[2]

[2]The layers notion in [123] have some similarities with the ones presented in Definition 2.11 when applied to $CRG(\mathcal{P})$, but the former (Definition 6.2 of [123]) has the limited role of providing the scaffolding of a transfinite inductive definition of the *weakly perfect model* which is a subset of the Well-Founded Model (as per Corollary 6.9 of [123]). The layering notion we present here has a standing of its own as an important syntactical ordering, besides its structuring influence inducing certain desirable characteristics of models of a semantics as we shall see in Section 7.1.

N.B.: The Rule Layering definition above states that two rules are placed in the same layer if they depend on each other. Notice this is an *if*, not an *if and only if*. I.e., according to Rule Layering, two rules *can* be placed in the same layer when, e.g, they have no dependencies amongst them. In the following example, the rules $x \leftarrow not\ x$ and $e \leftarrow e$ are placed in the same layer despite there being no dependencies whatsoever between them.

Example 3.2. ***Rule Layering example.*** *Consider the following program P, depicted along with the layer numbers for its least layering:*

$b \leftarrow not\ b$	$d \leftarrow not\ c$	$c \leftarrow not\ d, not\ y, not\ a$	P^3 — Layer 3
$b \leftarrow not\ x$	$y \leftarrow not\ x$	$z \leftarrow f$	P^2 — Layer 2
$x \leftarrow not\ x$	$e \leftarrow e$	f	P^1 — Layer 1
$\emptyset$			P^0 — Layer 0

Figure 3.1: A NLP's rules distributed along the program's layers.

Atom f has a fact rule: its body is empty (it depends on no other rule), and therefore it is placed in the lowest possible layer: P^1. The unique rule for x is also placed in Layer 1 in the least layering of P because it depends only on itself. Likewise for rule $e \leftarrow e$. Rules $b \leftarrow not\ x$ and $y \leftarrow not\ x$ are necessarily placed strictly above Layer 1 because they both depend directly on the rule for x which in turn does not depend on any of them. So, both these rules for y and for b are placed in Layer 2, P^2, in the least layering of P. For the same reason, rule $z \leftarrow f$ is placed in Layer 2, because it depends on the (fact) rule for f which is in Layer 1.

Notice this important difference between Layering and Stratification: the Layering does not distinguish between positive and negative dependencies nor does it treat such cases differently, as the Stratification does (cf. Definition 3.13). For the Layering notion the only important factor is the existence of, or lack thereof, syntactic dependency, regardless of it being through a positive or negative literal. This is the reason why the Layering puts rule $z \leftarrow f$ in a layer strictly above that of the fact f (because $z \leftarrow f$ depends on the fact f and not vice-versa), whereas the Stratification would allow $z \leftarrow f$ to be in the same stratum as the fact f (because $z \leftarrow f$ depends positively *on the fact f). I.e., the Layering and the Stratification use different criteria to assign layer/stratum ordinal indices.*

Rule $b \leftarrow not\ b$ is placed strictly above all other rules for b that do not depend on b, i.e., on Layer 3, P^3. The rule for c is placed strictly above the rule for y because it depends on not y and no rule for y depends on any rule for c. The rule for d is placed in the same Layer as the rule for c because they depend on each other. Hence, both rules for c and d are placed in Layer 3, P^3.

Building upon the (rule) layering we can now define the "atom layering" — a notion similar to that of stratification.

Definition 3.15. Atom-Layering of a Constrained Normal Logic Program $\mathcal{P}$. Let $\mathcal{P} = P \cup \mathcal{C}$ be an CNLP, and $Lf/1$ a rule layering function of $\mathcal{P}$. An atom layering function $ALf/1$ is a function defined over the atoms of $\mathcal{P}$, assigning each atom $a \in \mathcal{H}_\mathcal{P}$ an ordinal, such that

$$ALf(a) = \begin{cases} lub_{r \in \mathcal{P}: head(r)=a}(Lf(r)) & \text{if } \exists_{r \in \mathcal{P}} head(r) = a \\ 0 & \text{otherwise} \end{cases}$$

where *lub* stands for the *least upper bound* — in this case, the least upper bound of all the rule layer ordinals for layers containing a rule with the atom a as *head*.

An atom layering of program $\mathcal{P}$ is a partition $\ldots, A^i_\mathcal{P}, \ldots$ of the set of atoms of $\mathcal{P}$ — $\mathcal{H}_\mathcal{P}$ — such that $A^i_\mathcal{P}$ contains all atoms a having $ALf(a) = i$. We write $A^{\leq\alpha}_\mathcal{P}$ as an abbreviation of $\bigcup_{\beta<\alpha} A^\beta_\mathcal{P}$, and $A^{\leq\alpha}_\mathcal{P}$ as an abbreviation of $A^{\leq\alpha}_\mathcal{P} \cup A^\alpha_\mathcal{P}$, and define $A^{\leq 0}_\mathcal{P} = \emptyset$. It follows immediately that $\mathcal{H}_\mathcal{P} = \bigcup_\alpha A^\alpha_\mathcal{P} = \bigcup_\alpha A^{\leq\alpha}_\mathcal{P}$, and also that the $\leq$ relation between layers of atoms is a total-order in the sense that $A^i_\mathcal{P} \leq A^j_\mathcal{P}$ iff $i \leq j$.

Amongst the several possible atom layerings of a program $\mathcal{P}$ we can always find the least one corresponding to the definition of "atom layering function" $ALf/1$ based upon the program's least rule layering function $Lf/1$.
N.B.: In the following, when referring to the program's "atom layering", we mean just such least atom layering, and we will explicitly mention "atom", as in "atom layering" to make the distinction from (rule) layering.

This notion of atom layering is a generalization of level-mapping [120, 123] because, as explained in [123],

> *"Level mappings are mappings from Herbrand bases to ordinals, i.e. they induce orderings on the set of all ground atoms while disallowing infinite descending chains"*

and the atom layering notion does allow for infinite descending chains. Atom layerings are always defined, and in this sense they are generalizations of stratifications — partially because atom layerings are applicable also when there are loops in the program, in which cases there are no stratifications. Rule layerings are also always defined, and they capture all syntactic structure of the program. Also, due to the definition of dependency, in general, atom layerings do not coincide with stratifications [24], nor do rule layers coincide with the layers definition of [209]. When a program is not stratified there are nonetheless

atom layerings. However, when the program at hand is stratified (according to [24]) it can easily be seen that there is a relation between its atom layerings and its stratifications. Notice that a stratification, applicable to atoms, may put two atoms in the same stratum if one of them only depends through positive arcs on the other (without any reciprocal dependency), whereas, in the same conditions, an atom layering would put them in different layers — cf. Example 3.3 below concerning rule $z \leftarrow f$. So, for each stratification there is an atom layering, possibly with more layers than the strata there are in the stratification. On the other hand, assuming the program is stratified, for each atom layering there is a stratification. Moreover, there is a clear correspondence between a stratification and the least atom layering for acyclic programs — in this case the only difference relates to the atoms whose rules have only positive dependencies on some other atom.

The motivation for this difference between layering and stratification in what positive non-reciprocal dependencies are concerned is mainly a matter of uniformity and simplicity of the definition of layering. Both rule and atom layerings are particular cases of the general case of graph layering (cf. Definition 2.11), and in that general case there is no notion of "positive" or "negative" dependency: a vertex either depends on another or it does not, and if the dependency is non-reciprocal then those vertices are placed in different layers.

Example 3.3. ***Atom Layering example.*** *Consider again the program from Example 3.2, now depicted along with both its least rule layering and least atom layering:*

Rule Layer	Atom Layer	Layer Index
$P^3 = \{b \leftarrow not\ b \quad d \leftarrow not\ c \quad c \leftarrow not\ d, not\ y, not\ a\}$	$A^3_P = \{b, c, d\}$	3
$P^2 = \{b \leftarrow not\ x \quad y \leftarrow not\ x \quad z \leftarrow f\}$	$A^2_P = \{y, z\}$	2
$P^1 = \{x \leftarrow not\ x \quad e \leftarrow e \quad f\}$	$A^1_P = \{x, e, f\}$	1
$P^0 = \emptyset$	$A^0_P = \{a\}$	0

Figure 3.2: NLP's rules and atoms distributed along the program's Rule and Atom least Layerings.

Atom a has no rules, therefore it is placed in atom-layer 0: A^0_P. Atoms x, e, f have only one rule in Layer 1, therefore they are placed in atom-layer 1: A^1_P. Atoms y, z have only one rule in Layer 2, and therefore they are placed in atom-layer 2: A^2_P. Atom b has two rules: one in Layer 2 and the other in Layer 3, therefore it is placed in atom-layer 3 which is the maximum of its rules' layers: A^3_P. Atoms c, d only have rules in Layer 3: they are placed in A^3_P.

Due to the definition of (least) atom layering — based on the (least) rule layering — there is a close relation between the atom layering of an atom and the rule layering for the rules having that atom as head. This relation is captured by the following proposition:

Proposition 3.1. ***The least atom layering of an atom identifies the highest layer with rules for the atom.*** *Let $\mathcal{P}$ be an CNLP, $Lf/1$ its least rule layering function, and $ALf/1$ its least atom layering function.*

$$\forall_{a \in \mathcal{H}_\mathcal{P}} ALf(a) = \alpha \Leftrightarrow \left(\forall_{r \in \mathcal{P}: head(r)=a} r \in \mathcal{P}^{\leq \alpha} \wedge (\alpha \neq 0 \Leftrightarrow \exists_{r' \in \mathcal{P}^\alpha head(r')=a})\right)$$

Proposition 3.2. ***A rule's layer is greater than or equal to each of the body's literals' atom-layering.*** *Let $\mathcal{P}$ be an CNLP, $Lf/1$ its least rule layering function, and $ALf/1$ its least atom layering function.*

$$\forall_{\substack{r \in \mathcal{P} \\ a \in |body(r)|}} Lf(r) \geq ALf(a)$$

3.2.1 Layers and Strongly Connected Components of Rules

We now turn to analyze the relationship between SCCs of rules and layers.

Proposition 3.3. ***Rules in the same SCC are in the same layer.*** *Let $\mathcal{P}$ be a CNLP.*

$$\forall_{r,r' \in \mathcal{P}} (r \twoheadleftarrow r' \wedge r' \twoheadleftarrow r) \Rightarrow Lf(r) = Lf(r')$$

Proposition 3.4. ***Layering of SCCs.*** *Let $\mathcal{P}$ be a CNLP. If there is an edge from SCC_1 to SCC_2, with $SCC_1 \neq SCC_2$, in the $SCCG(\mathcal{P})$ then $\forall_{\substack{r_1 \in SCC_1 \\ r_2 \in SCC_2}} Lf(r_2) > Lf(r_1)$.*

3.2.1.1 Layers and bodies of rules

The (least) atom layering of a program allows us to partition the body of any given rule into atom-layer indexed subsets.

Definition 3.16. Atom-layer partition of a rule's body. Let $\mathcal{P}$ be a CNLP, r a rule of $\mathcal{P}$. $body(r)$ can be then partitioned into subsets $\ldots, body(r)^\alpha, \ldots$ such that each

$$body(r)^\alpha = \{B_i \in body(r)^+ : ALf(B_i) = \alpha\} \cup \{not\ C_j \in body(r)^- : ALf(C_j) = \alpha\}$$

Corollary 3.1. ***A rule's layer is greater than or equal to each of the body's subsets index.*** *Let $\mathcal{P}$ be a CNLP and r a rule of $\mathcal{P}$.*

$$\forall_{body(r)^\alpha \subseteq body(r)} Lf(r) \geq \alpha$$

Proposition 3.5. ***A rule's body literals in a loop have atom-layering equal to the rule's layer.***

$$\forall_{\substack{a \in \mathcal{H}_P \\ r \in \mathcal{P}}} a \in |body(r)^{Lf(r)}| \Rightarrow ALf(a) = Lf(r)$$

Proof. It follows trivially from Definition 3.16. □

Corollary 3.2. ***Loop and non-loop parts of a rule's body.*** *Let $\mathcal{P}$ be a CNLP, and r a rule of $\mathcal{P}$. $body(r)^{Lf(r)}$ is the set of literals of $body(r)$ which are in loop with r, and $body(r) \setminus body(r)^{Lf(r)}$ is the set of literals of $body(r)$ not in loop with r.*

Proof. Trivial: it follows immediately from Definition 3.16 and Proposition 3.5. □

In order to simplify notation we write $\overline{body(r)}$ as an abbreviation of $body(r) \setminus body(r)^{Lf(r)}$. By Proposition 3.2 we thus know that $\overline{body(r)}$ represents the subset of literals in the body of r whose corresponding atoms have all their rules, if any, in layers strictly below that of r. We use the $\overline{body(r)}$ notation in Definitions 6.2 and 6.7 to introduce the notion of *layer supported* interpretation and the *layered negative reduction* operation, respectively.

3.2.2 Transfinite Layering

Layering also copes with programs with a transfinite number of layers, as long as there is no infinitely long *descending* chain of dependencies. In practice, all useful programs have a finite number of layers, but for theoretical completeness we show that this layering notion also deals with the transfinite case.

Example 3.4. ***Program with transfinite number of layers.*** *Let $P =$*

$$\begin{array}{rcl} p(s(X)) & \leftarrow & p(X) \\ p(0) & & \end{array}$$

The ground (layered) version of this program, assuming there is only one constant 0 (zero) is:

$$\begin{array}{rcl} \vdots & \leftarrow & \vdots \\ p(s(s(0))) & \leftarrow & p(s(0)) \\ p(s(0)) & \leftarrow & p(0) \\ p(0) & & \end{array}$$

This program has a layering even though it has an infinite chain of dependencies. This is the case because that infinite chain is ascending *— this program has a transfinite number of layers.*

A typical case of a program with no layering (representing a whole class of programs with real theoretical interest) has an infinitely long *descending* chain of dependencies and was presented by François Fages in [100]. We repeat it here for illustration and explanation.

Example 3.5. ***Program with no layering [100].***

$$q \leftarrow not\ p(0)$$

$$p(X) \leftarrow p(s(X)) \qquad\qquad p(X) \leftarrow not\ p(s(X))$$

The ground version of this program, assuming there is only one constant 0 *(zero):*

$$q \leftarrow not\ p(0)$$

$$\begin{array}{rcl} p(0) & \leftarrow & p(s(0)) \\ p(s(0)) & \leftarrow & p(s(s(0))) \\ p(s(s(0))) & \leftarrow & p(s(s(s(0)))) \\ \vdots & \leftarrow & \vdots \end{array} \qquad\qquad \begin{array}{rcl} p(0) & \leftarrow & not\ p(s(0)) \\ p(s(0)) & \leftarrow & not\ p(s(s(0))) \\ p(s(s(0))) & \leftarrow & not\ p(s(s(s(0)))) \\ \vdots & \leftarrow & \vdots \end{array}$$

The only model of this program is $\{p(0), p(s(0)), p(s(s(0)))\ldots\}$ *or, in a non-ground form,* $\{p(X)\}$*. This program is of theoretical interest for two reasons: 1) it has no layering because is has an infinite* descending *chain of dependencies, and 2) it has no Stable Models, even though it has no loops, which shows a whole class of NLPs to which the SMs semantics provides no model. I.e., if a semantics for NLPs is to provide a model for each and every NLP it must cater for transfinite support chains, classically considered non-well-founded (cf. [100]).*

Now that we have covered the syntactic structure of NLPs we turn to consider also a few other classes of Logic Programs and see how they relate to NLPs.

---

4 . Other Classes of Logic Programs

(...) no single logic is strong enough to support the total construction of human knowledge.

JEAN PIAGET

In this chapter we discuss how other classes of Logic Programs, namely Extended Logic Programs (ELPs) and Disjunctive Logic Programs (DisjLPs), can be transformed into Normal Logic Programs. This allows us to focus the question of finding a semantics for Logic Programs in just the class of NLPs without loss of generality concerning ELPs and DisjLPs — we thus do not return to the topic of ELPs or DisjLPs outside this chapter.

4.1 Extended Logic Programs

Extended Logic Programs are different classes of Logic Programs that allow to derive negative conclusions, i.e., heads of rules may be explicitly negated literals (distinct from the default implicitly negated ones)[73].

Definition 4.1. Extended Logic Program. Extended Logic Programs are a generalization of NLPs in the sense that the rules of an ELP are of the same form as those of NLPs, but where all the non-default negated literals (both in the bodies and in the heads of rules) are objective literals — an objective literal being either an atom A or its explicit negation $\neg A$, where $\neg\neg A \equiv A$.

When considering ELPs contradictions may arise simply because heads of rules can be explicitly negated literals, for example:

Example 4.1. ***Extended Logic Programs.*** *In this ELP 'not ' is the default negation*

and '¬' is the explicit negation.

$$\begin{array}{rcl} \neg a & & \\ a & & \\ b & \leftarrow & not\ c \\ c & \leftarrow & not\ b \end{array}$$

In this case, since both a and $\neg a$ are facts, there is an outright explicit contradiction.

There are several alternative semantic approaches to deal with the possibility of explicit contradictions ELPs syntactically allow for. One can either prevent explicit contradictions or accept them. If we are to accept explicit contradictions there are two alternatives:

- Adopting classical logic's *Ex Contradictione Quodlibet* ("anything follows from contradiction") principle: in this case the unique model[1] is the set of all literals (positive, explicitly and default negated) — in example 4.1 this means accepting as unique model $\{a, \neg a, not\ a, not\ \neg a, b, \neg b, not\ b, not\ \neg b, c, \neg c, not\ c, not\ \neg c\}$.
- Taking a paraconsistent approach: in this case there might be several models, depending on the semantics being considered, and each model may be paraconsistent — in example 4.1 this means there will be the two models $\{a, \neg a, not\ a, not\ \neg a, b, not\ \neg b, not\ c\}$, and $\{a, \neg a, not\ a, not\ \neg a, c, not\ \neg c, not\ b\}$ where their paraconsistency is restricted to a, as opposed to the model according to the *Ex Contradictione Quodlibet* in the previous point, where both b and c are also paraconsistent in its model despite there being no contradiction in the program concerning b or c.

If we choose to prevent contradictions we can do so by means of a semi-normalization [11, 74] of the program, i.e., by adding $not\ \neg head(r)$ to the body of each rule r.

Definition 4.2. Semi-normalization. Let P be an Extended Logic Program. Then, $SN(P) = \{head(r) \leftarrow (body(r) \cup \{not\ \neg head(r)\}) : r \in P\}$ is the semi-normalized version of P.

Consider the again the program from example 4.1

Example 4.2. *Semi-normalization.* *After the semi-normalization the program becomes*

$$\begin{array}{rcl} \neg a & \leftarrow & not\ \neg\neg a \\ a & \leftarrow & not\ \neg a \\ b & \leftarrow & not\ c, not\ \neg b \\ c & \leftarrow & not\ b, not\ \neg c \end{array}$$

[1] We formally define the notions of interpretation, satisfaction, and model in Chapter 5.

i.e.,

$$\begin{array}{rcl} \neg a & \leftarrow & not\ a \\ a & \leftarrow & not\ \neg a \\ b & \leftarrow & not\ c, not\ \neg b \\ c & \leftarrow & not\ b, not\ \neg c \end{array}$$

Another simple way to discard inconsistent models is by adding ICs, in particular, adding one IC of the form $\bot \leftarrow a, \neg a$ for each atom $a \in \mathcal{H}_P$.

ELPs can straightforwardly be transformed into NLPs simply by mapping each explicitly negated literal $\neg A$ into a new atom $A*$ not previously present in the original program's Herbrand Base. This way, we provide a simple method to provide semantics to ELPs: given a 2-valued semantics S for NLPs, just transform an ELP P into an NLP $P*$ with recourse to $A*$ atoms as above, and considering the models of $P*$ according to S as the Extended S models of P. If desired, $P*$ can be complemented with the ICs $\bot \leftarrow a, \neg a$, i.e., $\bot \leftarrow a, a*$.

After semi-normalization, transformation and IC inclusion, the program of the example above becomes

$$\begin{array}{rcl} \bot & \leftarrow & a, a* \\ \bot & \leftarrow & b, b* \\ \bot & \leftarrow & c, c* \\ a* & \leftarrow & not\ a \\ a & \leftarrow & not\ a* \\ b & \leftarrow & not\ c, not\ b* \\ c & \leftarrow & not\ b, not\ c* \end{array}$$

4.2 Disjunctive Logic Programs

Definition 4.3. Disjunctive Logic Program. Disjunctive Logic Programs (DLPs) ([43, 44, 86, 94, 139, 140, 158]) are another generalization of NLPs in the sense that the rules of a DLP are of the same form as those of NLPs, but where the heads of rules are disjunctions of literals, i.e., rules are of the form

$$H_1 \vee \ldots \vee H_q \leftarrow B_1, \ldots, B_n, not\ C_1, \ldots, not\ C_m, \text{ with } q > 0 \text{ and } (m, n \geq 0 \text{ and finite})$$

where all the H_i, B_j, C_k are objective literals.

DisjLPs can also be easily transformed into NLPs via a simple syntactic operation known as the *Shifting Rule* [86]. For self-containment we include its definition.

Definition 4.4. *Shifting rule* (adapted from def. 3.1 of [86]). Let

$$H_1 \vee \ldots \vee H_q \leftarrow B_1, \ldots, B_n, not\ C_1, \ldots, not\ C_m$$

be a disjunctive logic rule. The *shifting rule* is a rewriting rule that transforms r into r'

$$H_1 \vee \ldots H_{i-1} \vee H_{i+1} \vee \ldots \vee H_q \leftarrow B_1, \ldots, B_n, not\ C_1, \ldots, not\ C_m, not\ H_i$$

for some $H_i \in head(r)$. A complete shift is a sequence of such individual shifts that results in a normal (i.e., non-disjunctive) rule.

In [86], the shifting rule was applied to DisjLPs, but with the special care of making sure the resulting program was stratified. This concern with stratification is related with the guarantee of existence of stable models for the resulting program (cf. lemma 5.2 of [86]) because the 2-valued semantics considered for the NLPs resulting from the "shifted" DisjLPs was the Stable Models semantics which, for some non-stratified programs cannot guarantee model existence. We overview the Stable Models semantics in detail in 6.4.1.1.

In Chapter 3 we saw that our syntactic structure of reference is the Layering, instead of the usual Stratification. Layering readily copes with loops, and SCCs in general, and thus non-stratification raises no problem to our approach. Likewise, as we shall see in the sequel, our semantics framework also deals with all kinds of non-stratification precisely because it complies with Layering. Hence, our approach to DisjLP is more general than the one in [86] because we allow for the arbitrary application of the shifting rule producing a highly non-stratified NLP:

Definition 4.5. Full shifting rule. Let

$$H_1 \vee \ldots \vee H_q \leftarrow B_1, \ldots, B_n, not\ C_1, \ldots, not\ C_m$$

be a disjunctive logic rule. The *full shifting rule* is a rewriting rule that transforms r into the set of rules:

$$
\begin{array}{lcl}
H_1 & \leftarrow & B_1,\ldots,B_n,not\ C_1,\ldots,not\ C_m,not\ H_2,\ldots,not\ H_q \\
\vdots & \leftarrow & \vdots \\
H_i & \leftarrow & B_1,\ldots,B_n,not\ C_1,\ldots,not\ C_m,not\ H_1,\ldots,not\ H_{i-1},not\ H_{i+1},\ldots,not\ H_q \\
\vdots & \leftarrow & \vdots \\
H_q & \leftarrow & B_1,\ldots,B_n,not\ C_1,\ldots,not\ C_m,not\ H_1,\ldots,not\ H_{q-1}
\end{array}
$$

Figure 4.1: Full shifting of a Disjunctive Logic Program.

4.2.1 Disjunctive LPs and the Intuitively Intended Meaning of Loops

The set of rules resulting from the *full shifting* of a disjunctive rule form a loop where each rule in the set directly depends on all the others. Taking the inverse perspective, we can say that loops over default negation can be seen as a way of writing some disjunctions, and this matches the intuition described in 6.5.4. Because the semantics for NLPs that we define in the sequel (Chapters 7 and 8) build upon the layering notions, they can assign a meaning (at least one model) to every NLP regardless of its possible resulting non-stratification, and, therefore, it can assign a semantics for DisjLPs when these are transformed into NLPs via *full shifting*. One of the advantages of our approach is that, because it copes with all kinds of non-stratification (loops), it is effectively applicable to *all* DisjLPs transforming them into NLPs which, according to the semantics we propose in the sequel, are guaranteed to have a meaning.

When we want to write the disjunction $a \vee b$, a common NLP-compatible way to do it is by means of writing the set of two rules

$$a \leftarrow not\ b \qquad b \leftarrow not\ a$$

Likewise, if we want to write, say, the three disjunctions $a \vee b$, $b \vee c$ and $c \vee a$ we would write the following set of six rules

$$
\begin{array}{llll}
a & \leftarrow & not\ b & \qquad b \leftarrow not\ a \\
b & \leftarrow & not\ c & \qquad c \leftarrow not\ b \\
c & \leftarrow & not\ a & \qquad a \leftarrow not\ c
\end{array}
$$

These rules are exactly the result of applying the Full shifting rule (Definition 4.5). Taking a "reversed" perspective of the application of the Shifting Rule, we could say a NLP with only these six rules is a representation of the three disjunctions, i.e., the intended meaning of the six rules is in fact the three disjunctions. Determining the intended meaning of a program is the task of a semantics, and we discuss this in Chapters 5 to 8, but since the intuition behind the semantics we propose in Chapter 8 is based on this approach

of considering loops (or SCCs) as a kind of disjunction, we point that intuition out here along with the discussion of Disjunctive LPs. This approach of considering the intended meaning of loops (through default negated literals) is a disjunction raises the question of, in this case, what would a program with only the three left-side rules

$$\begin{array}{rcl} a & \leftarrow & not\ b \\ b & \leftarrow & not\ c \\ c & \leftarrow & not\ a \end{array}$$

represent? Intuitively, we could interpret them a "half disjunction" since these three rules are one half of the six rules above. Each pair of the six rules above (in a separate line) stands for a single disjunction which is independent from another disjunction. E.g., the two rules $a \leftarrow not\ b$ and $b \leftarrow not\ a$ gives us the choice of believing a is true, or b is true, or both a and b are true — that is the meaning of the disjunction $a \vee b$. Any one of these choices for the first disjunction is independent from the choice taken for the second $b \vee c$. When we have only the three left-side rules these can be interpreted as representing a non-independent (or conditional) disjunction in the sense that, once we choose, say, believe a is true, the truth-values of both b and c become immediately determined, i.e., their truth-values are non-independent of (or conditional on) the truth-value chosen for a. Having no preferences nor priorities mechanism, in the three rules loop example above, any one of a or b or c could be chosen to be believed in in first place thus giving rise to three alternative solutions — we illustrate this with Example 8.1 where the literals in that example are but a renaming of the a, b, and c herein. This is the intuition behind the hypotheses assumption based semantics we define in Chapter 8.

Similarly to what was said previously about models for ELPs, models for DisjLPs can be obtained by simply transforming a DisjLP P into a NLP P^{SR} via the *shifting rule* and then consider the models of P^{SR} as the disjunctive models of P. In the case of an Extended Disjunctive Logic Program (with both explicitly negated objective literals, and disjunctions in the heads of rules) both transformations can be applied in sequence to produce an NLP: the models of an EDisjLP P are the models of $SN(P^{SR})*$.

Now that we know how to identify the structure of a (normal, or extended, or disjunctive) logic program, we are ready to take on the task of defining a semantics for them.

Part II

Semantics for Logic Programs

5 . Basic Semantics Concepts

"Multum in parvo"
("*Much in little*")

LATIN SAYING

We now cover the concepts of interpretation (both 2- and 3-valued ones), satisfaction, and classical (and minimal) models; and also include two useful orderings among interpretations and models. We conclude by defining what a semantics for (Constrained) Normal Logic Programs is.

5.1 Interpretations

The notions herein presented are those in [15]. We define 2- and 3-valued Herbrand interpretations and models of (constrained) normal logic programs. Since non-Herbrand interpretations are beyond the scope of this work, in the sequel we sometimes drop the qualification Herbrand.

Definition 5.1. 2-valued interpretation. A 2-valued interpretation I of an NLP P is a set of ground literals whose atoms exhaust the Herbrand base $\mathcal{H}_P$ of P. I.e., any 2-valued interpretation I is a set

$$I = I^+ \cup I^-$$

where $I^+ \subseteq \mathcal{H}_P$ is the set of atoms which are true in I, and $I^- = not\ (\mathcal{H}_P \setminus I^+)$ is the set of negative literals of I, corresponding to the atoms false in I. These interpretations are called 2-valued because in them each atom is either true or false, i.e. $\mathcal{H}_P = I^+ \cup |I^-|$ and $I^+ \cap |I^-| = \emptyset$.

As argued in [206], interpretations of a given program P can be thought of as "possible beliefs" representing possible states of our knowledge about the meaning of P. Since that

knowledge is likely to be incomplete, we need the ability to describe interpretations in which some atoms are neither true nor false but rather undefined, i.e. we need 3-valued interpretations:

Definition 5.2. 3-valued interpretation. By a 3-valued interpretation I of a program P we mean a set

$$I = I^+ \cup I^-$$

where I^+ and $|I^-|$ are disjoint subsets of the Herbrand base $\mathcal{H}_P$ of P. The set I^+ (the *true* or *positive* part of I) contains all ground atoms true in I, the set I^- (the *false* or *negative* part of I) contains all ground negative literals with corresponding atom false in I, and the truth value of the remaining atoms is *undefined* (or *unknown*). We sometimes refer to the *undefined* atoms of I as I^u.

It is clear that 2-valued interpretations are a special case of 3-valued ones, for which $\mathcal{H}_P = I^+ \cup |I^-|$ additionally holds, having $I^u = \emptyset$ in consequence.

Proposition 5.1. *Interpretation as function.* *Any interpretation I can equivalently be viewed as a function $I : \mathcal{H}_P \to V$ where $V = \{0, \frac{1}{2}, 1\}$ defined by:*

$$I(A) = \begin{cases} 0 & \textit{if } \textit{not } A \in I^- \\ \frac{1}{2} & \textit{if } \quad A \in I^u \\ 1 & \textit{if } \quad A \in I^+ \end{cases}$$

where $I(\textit{not } A) = 1 - I(A)$.

Of course, for 2-valued interpretations there is no atom A such that $I(A) = \frac{1}{2}$. Actually, $I(A) = \frac{1}{2}$ iff $A \in I^u$, which is only the case when I is a 3-valued interpretation which is not a 2-valued one.

5.2 Rule Satisfaction and Models

Definition 5.3. Interpretation Satisfaction. Let I be a 2-valued interpretation, and S an arbitrary set of literals. We say that I satisfies S, written $I \models S$, iff $I^+ \supseteq S^+$ and $I^- \supseteq S^-$.

Definition 5.4. Interpretation Default Satisfaction. Let I be a 3-valued interpretation, and S an arbitrary set of literals. We say that I satisfies S *by default*, written $I \models S$, iff $I^+ \supseteq S^+$ and $I^- \supseteq S^-$.

Proposition 5.2. ***Interpretation Satisfaction implies Interpretation Default Satisfaction.*** *I is a 2-valued interpretation and I satisfies S, then I also satisfies S* by default.

Proof. Trivial from Definitions 5.1, 5.2, and 5.3, 5.4. □

Since the body of a rule is an arbitrary set of literals, we can establish a useful relationship between an interpretation and the body of a rule:

Definition 5.5. Interpretation Satisfies Rule. We say a 3-valued interpretation I satisfies a rule r of a program P *by default*, written $I \models r$, iff

$$\Big(head(r) \in I^+\Big) \vee \Big(I \not\models body(r)\Big)$$

Informally, a 3-valued interpretation satisfying a rule by default means the head of the rule is true in I or the body of the rule is not satisfied by default in I (either by having one positive literal in the body which is not in I^+, or by having a negative literal in the body whose corresponding positive literal is in I^+).

Models are defined based on the rule satisfaction by default notion:

Definition 5.6. 3-valued model. A 3-valued interpretation I is called a 3-valued model of a program P (abbreviated as $I \models P$) iff for every ground instance of a program rule $r \in P$ we have $I \models r$.

The special case of 2-valued models has the following straightforward definition:

Definition 5.7. 2-valued model. A 2-valued interpretation I is called a 2-valued model of a program P (abbreviated as $I \models P$) iff for every ground instance of a program rule $r \in P$ we have $I \models r$.

We also refer to 2-valued models as Classical Models (CMs) of P, and write $CM_P(M)$ to denote M is a Classical Model of P, and $CM(P)$ to denote the set of all Classical Models of P.

When considering Constrained NLPs (cf. def. 3.2) models (either 2- or 3-valued ones) must comply with the additional requirement of not including the reserved atom $\bot$. I.e.,

Definition 5.8. Constrained model. An interpretation I is called a constrained model of a Constrained NLP $\mathcal{P} = P \cup \mathcal{C}$, where $\mathcal{C}$ is the set of Integrity Constraints, iff I is a model of $P \cup \mathcal{C}$ and $\bot \notin I$.

It follows immediately from definitions 5.5 and 5.8 that a constrained model I of $P \cup \mathcal{C}$ is such that $I \not\models body(r_\perp)$ for every Integrity Constraint rule $r_\perp \in \mathcal{C}$.

When considering a paraconsistency approach, one can also be interested in paraconsistent models.

Definition 5.9. Constrained paraconsistent model. An interpretation I is called a constrained paraconsistent model of a Constrained NLP $\mathcal{P} = P \cup \mathcal{C}$, where $\mathcal{C}$ is the set of Integrity Constraints, iff I is a model of $P \cup \mathcal{C}$ and $\perp \in I$.

These orderings among interpretations and models will be useful:

Definition 5.10. Classical (or Truth) ordering. If I and J are two interpretations then we say that $I \leq J$ if $I(A) \leq J(A)$ for any ground atom A. If $\mathcal{I}$ is a collection of interpretations, then an interpretation $I \in \mathcal{I}$ is called minimal in $\mathcal{I}$ if there is no interpretation $J \in \mathcal{I}$ such that $J \leq I$ and $I \neq J$. An interpretation I is called least in $\mathcal{I}$ if $I \leq J$ for any other interpretation $J \in \mathcal{I}$. A model M of a program P is called minimal (resp. least) if it is minimal (resp. least) among all models of P.

We write $MM_P(M)$ to denote that M is a Minimal Model (MM) of P, and $MM(P)$ to denote the set of all MMs of P.

Definition 5.11. Fitting (or Knowledge) ordering. If I and J are two 3-valued interpretations then we say that $I \leq_F J$ [101] iff $I \subseteq J$ (where $\subseteq$ is set-inclusion). If $\mathcal{I}$ is a collection of interpretations, then an interpretation $I \in \mathcal{I}$ is called F-minimal in $\mathcal{I}$ if there is no interpretation $J \in \mathcal{I}$ such that $J \leq_F I$ and $I \neq J$. An interpretation I is called F-least in $\mathcal{I}$ if $I \leq_F J$ for any interpretation $J \in \mathcal{I}$. A model M of a program P is called F-minimal (resp. F-least) if it is F-minimal (resp. F-least) among all models of P.

Note that the classical ordering is related with the number of true atoms — one could say it is a Truth Ordering, — whereas the Fitting ordering is related with the amount of information, i.e. non-undefinedness — a Knowledge Ordering.

Finally, we can formally define what a semantics for Normal Logic Programs is.

Definition 5.12. Semantics for (Constrained) Normal Logic Programs. A semantics S for (C)NLPs is a mapping, which assigns to every (C)NLP $\mathcal{P}$ a set $Models_S(\mathcal{P})$ of models of $\mathcal{P}$ such that $Models_S(\mathcal{P}) = Models_S(ground(\mathcal{P}))$, where $ground(P)$ stands for the Herbrand instantiation of $\mathcal{P}$. We also write $S_{\mathcal{P}}(M)$ to denote that M is a model of $\mathcal{P}$ according to S.

Equipped with the understanding of the structure of NLPs and the basic semantics concepts we now turn to build a general semantics framework for NLPs.

---

6 . Semantics Building Tools

> The least of things with a meaning is worth more in life than the greatest of things without it.
>
> C.G. Jung

In subsection 1.2.1 we saw that the quest for deductive reasoning with Logic Programs spins around the central axis of semantics, i.e., answering the question "which should be the intended models of a program?". *Determining which interpretations are considered models is essential to both skeptical and credulous entailment, i.e., cautious and brave deductive reasoning.*
In the context of Knowledge Representation and Reasoning using Normal Logic Programs, a semantics can be seen as a set of criteria to choose which interpretations, i.e., sets of truth values assigned to the atoms of a program, to accept as models (cf. Definition 5.12). This presupposes that there might be some degree of freedom in choosing the atoms' truth values, which immediately raises the question of what may provide or restrict such degrees of freedom.
In this chapter we first depict some general criteria which can be used as guidelines to accept or reject as models candidate interpretations — these include the notions of support*, and set inclusion minimality. Then, we review some known syntactic methods to reduce the set of candidate models by imposing restrictions on the degrees of freedom in choosing truth values for atoms. Finally, we review the current State-of-the-Art semantics (both 2- and 3-valued) and outline the motivation and need for a new 2-valued semantics for Normal Logic Programs. The contributions in this chapter stem from the research that lead to our publications [193, 194, 196, 197, 198].*

6.1 The Notion of Support

Several semantics (Minimal Models, Clark's Completion [149], Perfect Models [207], Stable Models [113], Well-Founded Semantics [109], Default Extensions [213], etc.) agree that whenever some interpretation I satisfies the body of a rule r (i.e., $I \models body(r)$, cf. Definition 5.3), then $head(r) \in I$ must hold in order for I to be considered a candidate model. This guideline mirrors the classical logic inference rule known as *modus ponens*: from a known (or believed) to be true premise, and an implication with that premise as antecedent, we must conclude (or believe in) the truth of the consequent. Formally, this is rendered by

$$\left(\exists_{r \in P}(I \models body(r) \wedge head(r) \notin I^+)\right) \Rightarrow I \notin Models_S(P)$$

which is equivalent to

$$I \in Models_S(P) \Rightarrow \left(\forall_{r \in P}(I \not\models body(r) \vee head(r) \in I^+)\right)$$

i.e. (cf. Definition 5.5),

$$I \in Models_S(P) \Rightarrow \forall_{r \in P} I \models r$$

where the right-hand side of the implication corresponds to the definition of classical model (cf. Definition 5.7). So, this guideline requires that all models of a given semantics to be classical models of the program:

$$I \in Models_S(P) \Rightarrow CM_P(I)$$

Some semantics take this guideline one step further and require all individual atoms of I^+ to have an individual support. Such requirement is known as *classical support*.

Definition 6.1. Classically Supported interpretation. An interpretation I is classically supported iff for every atom $A \in I^+$ there is some rule $r \in P$ with $head(r) = A$ such that $I \models body(r)$. When I is a 2-valued interpretation, this notion of support requires *all* the literals in the body of some rule for A (even when trivially so because there are none) to be *true* under I in order for A to be classically supported in I, because $|I|$ exhausts $\mathcal{H}_P$. In this case we also say r and A are classically supported.

There may also exist classical models (CMs) which are not classically supported as the former (Definition 5.7) does not imply the latter (Definition 6.1), while the inverse implication holds. For this reason, there might also be cases where some non-classically supported CMs may be accepted as models by a semantics (e.g., as it happens with simple Minimal Models semantics). Nonetheless, the classical notion of support has been considered by several LP semantics as a *sine qua non* condition for their models. The classical

support requirement, however, has its drawbacks as, namely, it may prevent model existence altogether. For example, the program with just the rule $a \leftarrow not\ a$ has no classically supported model, though it has a minimal one. This hints that the classical support notion might be too restrictive and that a slightly more relaxed notion of support may be necessary to allow for model existence guarantee. In fact, if a semantics *only* accepts as models classically supported interpretations it risks missing some useful theoretical properties (besides model existence) with practical implementations consequences as we shall see below, in Section 6.5. For example, model existence guarantee is an indispensable condition for, say, Relevance (cf. 6.5.5.2).

We now introduce a slightly more relaxed notion of support, that of *layered support*, which is in accordance with the Layering notions of Chapter 3 with recourse to the $\overline{body(r)}$ notation presented at the end of subsection 3.2.1[1] in page 29.

Literals in $\overline{body(r)}$ are, by definition, not in loop with r and hence the rules for those literals are necessarily placed in layers strictly lower than that of r. Notice that, e.g., in a rule $r = a \leftarrow a$, we have $\overline{body(r)} = \emptyset$, i.e., the part of the body in loop includes the atom a which, in this case, coincides with the *head* of the rule, thus leaving the set of literals of $body(r)$ not in loop — $\overline{body(r)}$ — empty.

Definition 6.2. Layer Supported interpretation. We say an interpretation I of P is layer supported iff every atom a of I is layer supported in I. Given a NLP P and some interpretation I with atom $a \in I$, we say a is layer supported in I iff there is some rule r in P with $head(r) = a$ such that $I \models \overline{body(r)}$. Likewise, we say the rule r is layer supported in I iff $I \models \overline{body(r)}$.

The notion of layered support requires that all $\overline{body(r)}$ literals be *true* under I in order for $head(r)$ to be layer supported in I. Hence, if $\overline{body(r)}$ is empty, r is *ipso facto* layer supported.

Proposition 6.1. ***Classical Support implies Layered Support.*** *Given a NLP P, an interpretation I, and an atom a such that $a \in I$, if a is classically supported in I then a is also layer supported in I.*

Proof. Knowing that, by definition, $\overline{body(r)} \subseteq body(r)$ for every rule r, it follows trivially from Definitions 6.1 (classical support) and 6.2 (layered support), that a is layer supported in I if a is classically supported in I. □

[1]The current notion of layered support is the product of a series of evolving steps previously published [193, 194, 196, 197].

6.2 Minimality

Set inclusion minimality (which refers to Definition 5.10) is also usually a guideline in restricting which interpretations can be accepted as models. Seeing models as sets of beliefs, whole model minimality can be understood as a form of skepticism in the sense that it would correspond to minimality of beliefs.

When performing abductive reasoning, for example, (cf. subsection 1.2.3 and Chapter 10), typically minimality is not strictly required of abductive solutions. Abduction can be modeled in Logic Programming, and in particular with ASP [65, 108, 142, 205, 237] systems, via NLP rules; i.e., an originally abductive LP can be syntactically transformed into a regular NLP (cf. Section 10.1) to be used under some LP system without any specific abduction mechanisms. In this sense, there is no fundamental difference between an NLP and an abductive LP, just as there is no fundamental difference between deductive and abductive reasoning (cf. Chapter 10), once suitable corresponding NLP representations are employed.

When we envisage models of a program as a set of hypotheses (akin to abduction) plus the consequences they entail via the rules of the program, the maximal skepticism principle can be seen as applicable only to the set of hypotheses, i.e., the minimality requirement aims at minimal sets of hypotheses, not necessarily minimal sets of consequences. The motivation behind minimality is the *Ockham's razor* principle mentioned in 1.2.3, which is concerned with making the "*fewest assumptions*" possible. Taking this approach one now needs to identify the set of assumable hypotheses — we do this in Chapter 8. The best known and used semantics (review in Section 6.4) require whole model minimality which makes no distinction between hypotheses and consequences because, for minimal models, minimality of hypotheses and minimality of consequences coincide. In this thesis we make the distinction clear and require minimality of hypotheses alone — in Chapter 8 we define the new Minimal Hypotheses semantics based on this approach.

The minimality requirement is usually applied to the whole model, but we take a hypotheses assumption perspective to semantics (akin to abduction), where we consider the truth-value of some literals as potentially assumable hypotheses. Under this setting, models of the semantics are the sets of assumed hypotheses plus the consequences they entail (as long as they result in a classical model), and thus the minimality (skepticism) principle should be applied to the set of assumed hypotheses and not necessarily to the whole model — whole model minimality can be imposed as an optional additional requirement if so desired, with attending cost. We fully explore the minimality of hypotheses semantics path in Chapter 8.

These two possibilities of minimality (of the assumed hypotheses set, and of the whole

model including the hypotheses and their consequences) are not exclusive and indeed they can be both required, although, in that case, with possibly an increased level of complexity in reasoning tasks.

6.3 Syntactic-based Restrictions on Models

The syntactic structure of a program can be used to restrict the truth-values of literals in the program. This is especially true for Definite and for Locally Stratified Logic Programs where the truth-values of *all* literals in the program are unambiguously determined simply by the syntactic structure of the program's rules.

6.3.1 Definite Logic Programs

The seminal paper [96] showed that the semantics of definite programs, where only positive literals can be used to write rules, should single out one model: the least Herbrand model (also known as simply the "least model"), known to be both classically supported (cf. Definition 6.1) and set inclusion minimal (cf. Definition 5.10 and Section 6.2).

One way to calculate the least model of a definite ground program is by means of the T operator by Van Emden and Kowalski [96]. For self-containment, we include the formal definition of the T operator, original from [96], but presented here according to the notation of [246] more similar to our overall notation.

Definition 6.3. T **operator [96, 246].** Let P be a definite and ground Normal Logic Program, and I an interpretation of P.

$$T_P(I) = \{head(r) : r \in P \wedge body(r) \subseteq I\}$$

The T operator returns the set of the heads of rules of P whose bodies are true in I. It follows immediately that $T_P(I)$ can be equivalentely defined as

$$T_P(I) = \{head(r) : r \in P \wedge (body(r) \setminus I) = \emptyset\}$$

or as

$$T_P(I) = facts(\{head(r) \leftarrow (body(r) \setminus I) : r \in P\})$$

i.e.,

$$T_P(I) = T_{\{head(r) \leftarrow (body(r) \setminus I) : r \in P\}}(\emptyset)$$

The T operator is called the *immediate consequences* operator because it allows us to extract the consequences of assuming I *true* in P.

The upward powers of T_P starting from an interpretation I are defined as follows:

$$\begin{aligned} T_P^0(I) &= I \\ T_P^{i+1}(I) &= T_P(T_P^i(I)), \text{ for } i \geq 0 \\ T_P^{\omega}(I) &= \bigcup_{i \geq 0} T_P^i(I) \end{aligned}$$

The least Herbrand model of a definite ground NLP P is the *Least Fixed Point of the T operator* applied to P, i.e., $lfp(T_P) = T_P^{\omega}(\emptyset)$. It follows immediately from this definition and the monotonic character of the T_P operator that the $\subseteq$ relation is a well-founded total order of the set $\{T_P^{\alpha}(\emptyset) : 0 \leq \alpha\}$.

It is worthwhile noticing that the $lfp(T_P)$, being equal to $T_P^{\omega}(\emptyset)$, is a strict superset of the empty set $\emptyset$ iff there are facts in P; i.e., if the $\leq$ dependency relation between layers of P (cf. Definition 2.11) is *well-founded*, or, equivalently, if not all rules of P have a transfinite number of layers below them.

By definition of *fixed point* it also follows that $lfp(T_P) = T_P^{\omega}(\emptyset) = T_P^{\omega+1}(\emptyset) = T_P(T_P^{\omega}(\emptyset)) = T_{\{head(r) \leftarrow (body(r) \setminus T_P^{\omega}(\emptyset)) : r \in P\}}(\emptyset) = facts(\{head(r) \leftarrow (body(r) \setminus T_P^{\omega}(\emptyset)) : r \in P\})$.

From a computational perspective, the calculus of $lfp(T_P) = T_P^{\omega}(\emptyset)$ by iterating cumulative applications of the T_P operator can be viewed as a syntactic incremental process of establishing semantic information; in this particular case, the least Herbrand model of P.

6.3.2 Locally Stratified Logic Programs

In [207] Przymusinski defined the class of *locally stratified programs* and showed it to be a superset of the class of stratified programs. Lemma 4.1 of [41] shows that a "*program P is locally stratified if, and only if, the negative dependency relation of P is well-founded.*" As it is well-known, a relation is said to be well-founded iff 1) it is acyclic; and 2) it has finitely long descending chains only. This means that locally stratified programs have no loops over default negated literals (DNLs), nor do they have infinitely long descending chains over DNLs (cf. Example 3.5). Moreover, also in [207], Przymusinski defined the *perfect model* semantics and showed that every locally stratified program has a unique perfect model, being a classically supported minimal model of P. Later (theorem 6.1 of [109]) it was shown that, for locally stratified programs, the perfect model coincides with the Well-Founded Model (we overview the Well-Founded Semantics and its Well-Founded Model in more detail in 6.4.2.1). In this context, the uniqueness of model of P is enough to say that, from a model-theoretic perspective, locally stratified logic programs are as

semantically determined (only allowing for one model) as definite programs. From this we can conclude that the only possible source of existence of several alternative models for a program rather than just one, relies on there existing loops over DNLs.

Intuitively, locally stratified programs are programs with no SCCs over DNLs in their ground version, and for this particular class of programs the following theorem holds trivially.

Theorem 6.1. ***Layered Support implies Classical Support for Acyclic Programs.*** *Let P be an acyclic logic program, I an interpretation, and $a \in I$ an atom. If a is layer supported in P, then a is also classically supported, and vice-versa.*

Analogously to what happens with definite logic programs, one can hence calculate the unique model for a locally stratified LP by means of an alternative deterministic operator: the program Remainder one (denoted by $\widehat{P}$), which was defined in [45] for calculating the Well-Founded Model and, as said above, coincides with the unique perfect model for locally stratified LPs. The Remainder can thus be seen as a generalization for NLPs of the T operator, the latter applicable only to the subclass of definite LPs.

6.3.2.1 Program Remainder operator

For self-containment, we include here the definitions of [45] upon which the Remainder operator relies, and adapt them where convenient to better match the syntactic conventions used throughout this thesis.

Definition 6.4. Program transformation (def. 4.2 of [45]). A *program transformation* is a relation $\mapsto$ between ground logic programs. A semantics S allows a transformation $\mapsto$ iff $Models_S(P_1) = Models_S(P_2)$ for all P_1 and P_2 with $P_1 \mapsto P_2$. We write $\mapsto^*$ to denote the fixed point of the $\mapsto$ operation, i.e.,

$$P \mapsto^* P' \text{ where } \nexists_{P'' \neq P'} P' \mapsto P''$$

Definition 6.5. Positive reduction (def. 4.6 of [45]). Let P_1 and P_2 be ground programs. Program P_2 results from P_1 by *positive reduction* ($P_1 \mapsto_P P_2$) iff there is a rule $r \in P_1$ and a negative literal *not* $b \in body(r)$ such that $b \notin heads(P_1)$, i.e., there is no rule for b in P_1, and $P_2 = (P_1 \setminus \{r\}) \cup \{head(r) \leftarrow (body(r) \setminus \{not\ b\})\}$.

Definition 6.6. Negative reduction (def. 4.7 of [45]). Let P_1 and P_2 be ground programs. Program P_2 results from P_1 by *negative reduction* ($P_1 \mapsto_N P_2$) iff there is a rule $r \in P_1$ and a negative literal *not* $b \in body(r)$ such that $b \in facts(P_1)$, i.e., b appears as a fact in P_1, and $P_2 = P_1 \setminus \{r\}$.

The negative reduction of [45] is consistent with the classical notion of support, but not with the layered one. Therefore, we introduce now a layered version of the negative reduction operation.

Definition 6.7. Layered negative reduction. Let P_1 and P_2 be ground programs. Program P_2 results from P_1 by *layered negative reduction* ($P_1 \mapsto_{LN} P_2$) iff there is a rule $r \in P_1$ and a negative literal $not\ b \in \overline{body(r)}$ such that $b \in facts(P_1)$, i.e., b appears as a fact in P_1, and $P_2 = P_1 \setminus \{r\}$.

This more cautious layered negative reduction has a direct translation (cf. Proposition 3.5) to deleting the rules that depend on the negation of a fact only if there are no other rules in loop with it, i.e., the negation of the fact is in the part of the body determined by rules of layers strictly below that of the rule being deleted.

Proposition 6.2. ***Layered negative reduction deletes at most the same rules as Negative reduction.*** *Let P be an NLP, and $P \mapsto_{LN} P_{LN}$, and $P \mapsto_N P_N$. Then, $P_{LN} \supseteq P_N$.*

Proof. Trivial from Definitions 6.6, and 6.7. □

Proposition 6.3. ***Layered negative reduction adds only polynomial complexity to Negative Reduction.***

Proof. The Rule Layering can be calculated in polynomial time since it is equivalent to identifying the SCCs in a graph [235], in this case, $CRG(P)$. Once having the Rule Layering, the Atom Layering can be calculated in linear time.

Once both Rule and Atom Layerings of a given P are established, the $\overline{body(r)}$ and $body(r)^{Lf(r)}$ subsets of $body(r)$, for each rule r, are identifiable in linear time — one needs to check just once for each literal in $body(r)$ if it is also in $\overline{body(r)}$ or in $body(r)^{Lf(r)}$.

Therefore, these polynomial time complexity operations are all the added complexity Layered negative reduction adds over regular Negative reduction. □

Definition 6.8. Success (def. 5.2 of [45]). Let P_1 and P_2 be ground programs. Program P_2 results from P_1 by *success* ($P_1 \mapsto_S P_2$) iff there are a rule $r \in P_1$ and a fact $b \in facts(P_1)$ such that $b \in body(r)$, and $P_2 = (P_1 \setminus \{r\}) \cup \{head(r) \leftarrow (body(r) \setminus \{b\})\}$.

It is easy to see that, when P is a definite program, the Success operation is closely related to the T operator (Definition 6.3) in the sense that

$$T_P^{\omega}(\emptyset) = facts(P^S), \text{ where } P \mapsto_S^* P^S$$

Definition 6.9. Failure (def. 5.3 of [45]). Let P_1 and P_2 be ground programs. Program P_2 results from P_1 by *failure* ($P_1 \mapsto_F P_2$) iff there are a rule $r \in P_1$ and a positive literal $b \in body(r)$ such that $b \notin heads(P_1)$, i.e., there are no rules for b in P_1, and $P_2 = P_1 \setminus \{r\}$.

Definition 6.10. Loop detection (def. 5.10 of [45]). Let P_1 and P_2 be ground programs. Program P_2 results from P_1 by *loop detection* ($P_1 \mapsto_L P_2$) iff there is a set $\mathcal{A}$ of ground atoms such that

1. for each rule $r \in P_1$, if $head(r) \in \mathcal{A}$, then $body(r) \cap \mathcal{A} \neq \emptyset$,
2. $P_2 := \{r \in P_1 | body(r) \cap \mathcal{A} = \emptyset\}$,
3. $P_1 \neq P_2$.

We are not entering here into the details of the *loop detection* step, but just taking note that 1) such a set $\mathcal{A}$ corresponds to an unfounded set (cf. [109]); 2) loop detection is computationally equivalent to finding the SCCs [235] in the $ERG(P)$ graph, as per Definition 3.5, and is known to be of polynomial time complexity; and 3) the atoms in the unfounded set $\mathcal{A}$ have all their corresponding rules involved in *loops* (cf. Definition 2.7) where all heads of rules in loop appear positive in the bodies of the rules in loop.

Example 6.1. *Loop detection and elimination.* *Let* $P =$

$$\begin{array}{lll} a & \leftarrow & b, c \\ b & \leftarrow & a \\ c & & \end{array}$$

Then, applying the loop detection operation $\mapsto_L$ *to* P *we obtain* $P' =$

$$c$$

The loop detection is finding and deleting the loop over the rules $a \leftarrow b, c$ *and* $b \leftarrow a$ *because all the heads of the rules forming the loop appear as positive literals in the bodies of the loop's rules — for this reason we call this a* positive loop. *The* unfounded set $\mathcal{A}$ *as per Definition 6.10 above is* $\mathcal{A} = \{a, b\}$.

From a philosophical or argumentation-theoretic perspective, the *loop detection* transformation corresponds, *grosso modo*, to detecting and eliminating positive self-referential arguments which are not unlike the *begging the question* fallacy. [2] defines the *Begging the Question* argumentation fallacy as

> *"A form of circular reasoning in which a conclusion is derived from premises that presuppose the conclusion."*

In essence, the *loop detection* transformation detects such positive-self-referential sets of rules and discards them as a means to support belief in the consequents (heads) of the rules involved.

Definition 6.11. Reduction (def. 5.15 of [45]).
Let $\mapsto_X$ denote the rewriting system: $\mapsto_X := \mapsto_P \cup \mapsto_N \cup \mapsto_S \cup \mapsto_F \cup \mapsto_L$.

Definition 6.12. Layered reduction.
Let $\mapsto_{LX}$ denote the rewriting system: $\mapsto_{LX} := \mapsto_P \cup \mapsto_{LN} \cup \mapsto_S \cup \mapsto_F \cup \mapsto_L$.

Lemma 6.1. ***Layered reduction deletes at most the same rules as Reduction.*** *Let P be an NLP, and $P \mapsto_{LX} P_{LX}$, and $P \mapsto_X P_X$. Then, $P_{LX} \supseteq P_X$.*

Proof. From Definitions 6.11, and 6.12 we know that the only difference between $\mapsto_X$ and $\mapsto_{LX}$ relies on the $\mapsto_N$ and $\mapsto_{LN}$ individual transformations. From Proposition 6.2 we know that, for every program P, $P \mapsto_{LN} P_{LN}$, and $P \mapsto_N P_N$, $P_{LN} \supseteq P_N$ holds. Therefore, having $P \mapsto_{LX} P_{LX}$ and $P \mapsto_X P_X$, we conclude $P_{LX} \supseteq P_X$. □

Definition 6.13. Remainder (def. 5.17 of [45]). Let P be a program. Let the program $\widehat{P}$ satisfy

1. $ground(P) \mapsto_X^* \widehat{P}$,
2. $\widehat{P}$ is irreducible w.r.t. $\mapsto_X$, i.e., there is no $P' \neq \widehat{P}$ with $\widehat{P} \mapsto_X P'$.

Then $\widehat{P}$ is called the *remainder* of P, and is guaranteed to exist and to be unique to P. Moreover, the calculus of $\mapsto_X^*$ is known to be of polynomial time complexity [45]. When convenient, we write $Rem(P)$ instead of $\widehat{P}$.

Lemma 5.18 of [45] shows that "*Every program P has a program remainder, i.e., the rewriting system $\mapsto_X$ is terminating.*" Also important is Corollary 5.22 of [45] showing that "*The rewriting system $\mapsto_X$ is confluent and the program remainder $\widehat{P}$ is the unique normalform of $ground(P)$ w.r.t. $\mapsto_X$.*"

Definition 6.14. Layered Remainder. Let P be a program. Let the program $\mathring{P}$ satisfy

1. $ground(P) \mapsto_{LX}^* \mathring{P}$,

2. $\mathring{P}$ is irreducible w.r.t. $\mapsto_{LX}$, i.e., there is no P' with $\mathring{P} \mapsto_{LX} P'$.

Then $\mathring{P}$ is called a *layered remainder* of P. Since $\mathring{P}$ is equivalent to $\widehat{P}$, apart from the difference between $\mapsto_{LN}$ and $\mapsto_N$, it is trivial that $\mathring{P}$ is also guaranteed to exist and to be unique for P. Moreover, the calculus of $\mapsto^*_{LX}$ is likewise of polynomial time complexity because $\mapsto_{LN}$ is also of polynomial time complexity. When convenient, we write $LRem(P)$ instead of $\mathring{P}$.

The Layered Remainder differs from the Remainder only the application of the Layered Negative Reduction instead of the Negative Reduction. Since the Layered Negative Reduction is but a conditional application of the Negative Reduction, by the same token, the rewriting system $\mapsto_{LX}$ is also confluent and the program layered remainder $\mathring{P}$ is the unique normalform of $ground(P)$ w.r.t. $\mapsto_{LX}$.

Example 6.2. $\mathring{P}$ ***versus*** $\widehat{P}$**.** *Let P be*

$$\begin{array}{lcl} a & \leftarrow & not\ b \\ b & \leftarrow & not\ c \\ c & \leftarrow & not\ a \\ a & & \end{array}$$

We can clearly see that the single fact rule is in layer 1 and that the remaining three rules forming the loop are in layer 2 (because they all depend on the fact rule a which, in turn, does not depend on any other rule).

$\widehat{P}$ is the fixed point of $\mapsto_X$, i.e., the fixed point of $\mapsto_P \cup \mapsto_N \cup \mapsto_S \cup \mapsto_F \cup \mapsto_L$. Since a is a fact, the $\mapsto_N$ (Negative reduction — Definition 6.6) transformation deletes the $c \leftarrow not\ a$ rule; i.e., $P \mapsto_N P'$ is such that $P' =$

$$\begin{array}{lcl} a & \leftarrow & not\ b \\ b & \leftarrow & not\ c \\ a & & \end{array}$$

Now in P' there are no rules for c and hence we can apply the $\mapsto_P$ (Positive reduction — Definition 6.5) transformation which deletes the not c from the body of b's rule; i.e, $P' \mapsto_P P''$ where $P'' =$

$$\begin{array}{lcl} a & \leftarrow & not\ b \\ b & & \\ a & & \end{array}$$

Finally, in P'' b is a fact and hence we can again apply the $\mapsto_N$ obtaining $P'' \mapsto_P P'''$ where $P''' =$

$$\begin{array}{l} b \\ a \end{array}$$

upon which no more transformations can be applied, therefore $\widehat{P} = P'''$.

On the other hand, $\mathring{P}$ *is the fixed point of* $\mapsto_{LX}$*, i.e., the fixed point of* $\mapsto_P \cup \mapsto_{LN} \cup \mapsto_S \cup \mapsto_F \cup \mapsto_L$*. To calculate* $\mathring{P}$ *the only applicable transformation is the* $\mapsto_{LN}$ *(Layered negative reduction — Definition 6.7) — which can potentially affect the* $c \leftarrow not\ a$ *rule because there is a fact rule* a*.*
However, the $c \leftarrow not\ a$ *rule also depends on the* $a \leftarrow not\ b$ *rule which depends (indirectly through the* $b \leftarrow not\ c$ *rule) on the* $c \leftarrow not\ a$*;*
therefore we have $body(c \leftarrow not\ a) = body^{Lf(c \leftarrow not\ a)}(c \leftarrow not\ a) \cup \overline{body(c \leftarrow not\ a)} = \{not\ a\} \cup \emptyset$*, i.e.,* $\overline{body(c \leftarrow not\ a)} = \emptyset$*.*
This means that the Layered negative reduction transformation, which considers only the $\overline{body(c \leftarrow not\ a)}$ *part of the rule's body, in this case, has no effect leaving the* $c \leftarrow not\ a$ *rule intact.*
Since there are no more applicable transformations of the Layered reduction, we end up with $\mathring{P} = P$*, quite different from* $\widehat{P}$*, because the Layered Remainder respects Layered Support.*

Lemma 6.2. ***The rules of the Remainder are "sub-rules" of the Layered Remainder.*** *Let* P *be an NLP. Then,* $\forall_{r \in \widehat{P}} \exists_{r' \in \mathring{P}} (head(r) = head(r') \wedge body(r) \subseteq body(r'))$.

All the individual transformations of both the Remainder and Layered Remainder operators are patently deterministic: they rely solely on the irrefutable truth of facts; the falsehood of atoms with no rules, and of those that uniquely depend positively on themselves; and propagate those deterministically known truth-values to all the rules that depend on them. As a consequence of the deterministic nature of the simplifications performed by both the Remainder and Layered Remainder, it turns out that all the heads of rules of $\widehat{P}$, respectively $\mathring{P}$, that are not simply facts correspond to literals whose truth-value is not determinable by the truth of facts and falsehood of atoms with no rules (or only positive self-referential rules) alone. For locally stratified programs all literals become determined via Remainder $\widehat{P}$, but this is not true in general for arbitrary normal logic programs.

6.3.3 Full-fledged Normal Logic Programs

The results from the previous subsections 6.3.1 and 6.3.2 show that, in what NLPs in general are concerned, the unique source of semantic model indeterminism (allowing for more than one acceptable model) must come from non-well-founded negative dependencies, i.e., when the NLP has loops over DNLs, or a transfinite number of layers with negative dependencies between rules of those layers (cf. Example 3.5). Under these circumstances DNLs may be envisioned as assumable hypotheses, or free choices, as proposed by [129].

In fact, it is precisely the non-stratification of DNLs that takes away the determinism in the truth-value assignment to literals allowing, in general, for several 2-valued models for the program.

Several attempts at finding the "right" 2-valued semantics have been made in the past decades, i.e., which 2-valued interpretations to accept as models, and one of the major concerns with those attempts was precisely how to deal with non-stratified default negation. We now review the most successful 2-valued and 3-valued approaches so far.

6.4 State of the Art Semantics

Over the past few decades the scientific community dedicated to knowledge representation and reasoning problems has developed a number of different semantics for Logic Programs. In fact, the history of the semantics for LPs has been one of successive attempts at providing a 2-valued consistent semantics for all NLPs. The process of defining a semantics has been guided by both the classical notion of support for rules and the minimality of models. However, when dealing with cases like some kinds of loops and transfinite number of Layers, the classical notion of support leads to unintuitive results. Some attempts to circumvent such undesired results resorted to the third logical truth-value, the *undefined*. In some cases a 3-valued semantics is actually desired and useful, but it is not an acceptable solution when a 2-valued semantics is what one really needs.

2-valued semantics — like the Minimal Models [38], Clark's Completion [60], Perfect Models [207], and the Stable Model [113], amongst others — strive to achieve the most complete information possible, assigning a truth-value to every atom, if possible. But in spite of their effort and insistence, none have been able to provide a semantics to every NLP.

3-valued semantics — like the Well-Founded Semantics [109] — sacrifice complete *true-or-false* information about every atom in favor of some desirable properties that some 2-valued semantics lack. The 3-valued semantics also assign a truth-value to every atom — in some cases the truth-value is *undefined*.

There are also other approaches, including multi-valued logics. Some of these multi-valued logics semantics tend to enter the realms of probabilistic reasoning, fuzzy-logic [154, 159] or other similar domains; while others consider several logical values due to the nature of the specific problem they are used to solve [27]. We leave this class of multi-valued/probabilistic/fuzzy logics outside the scope of this thesis as we are here focusing on Normal Logic Programs.

6.4.1 Two-Valued Semantics

There are several 2-valued proposals for semantics of Logic Programs: Minimal Models, Clark's Completion, Stable Models, are some of the best well-known examples.

Generally accepted by the scientific community working on 2-valued semantics for Logic Programs as the *de facto* standard, the Stable Models semantics [113] gives exactly the results one intuitively expects in most cases.

6.4.1.1 Stable Models

The semantics of Stable Models (SM) [113] is a cornerstone for the definition of some of the most important results in logic programming of the past more than two decades, providing an increase in logic programming declarativity and a new paradigm for program evaluation.

Definition 6.15. Stable Model ([113]). In their famous paper presenting Stable Models, Michael Gelfond and Vladimir Lifschitz defined a method to check if an interpretation M is a Stable Model of a Normal Logic Program P. This method is enounced in three steps:

1. Calculate a transformed NLP P/M by deleting from P all rules r having *not* $A \in body(r)$, where $A \in M^+$. Then delete from all other rules all the remaining default literals. This step is known as the *Program division* P/M. In a more compact manner,

$$P/M = \{head(r) \leftarrow body(r)^+ : r \in P \wedge (|body(r)^-| \cap M^+) = \emptyset\}$$

2. Since P/M is negation-free it has a unique Least Herbrand Model which we can calculate through the iteration of the van Emden and Kowalski's T operator [96]. The Least Model is the *Least Fixed Point of the T operator* applied to P/M, i.e., $lfp(T_{P/M}) = T^{\omega}_{P/M}(\emptyset)$.

3. Check if M^+ equals the calculated Least Model of the transformed Program.
   M is a Stable Model of P iff $M^+ = T^{\omega}_{P/M}(\emptyset)$.

In this document we also use an alternative notation for $T^{\omega}_{P/M}(\emptyset)$, which is $\Gamma_P(M)$. M is thus an SM of P iff $M^+ = \Gamma_P(M)$. For any interpretation I, we consider $\Gamma^0_P(I) = I$, and $\Gamma^{n+1}_P(I) = \Gamma_P(\Gamma^n_P(I))$.

Intuitively, the first two steps of the method described above in Definition 6.15 can be envisioned as calculating the *consequences* of the rules in P, according to the classical notion of support, assuming *true* the atoms in M and all the others false. I.e., $\Gamma_P(M)$ is the set of those classically supported consequences. It is also worth noticing that, by definition of SM, requiring $M^+ = T^{\omega}_{P/M}(\emptyset)$ implies requiring M^+ to be a *well-ordered set* in the sense pointed out after Definition 6.3 (T operator). Moreover, SMs obeys the Closed World Assumption (CWA), whereby atoms without rules will be false, thereby maximizing falsity whenever there is that particular absence of rule option. Alternatively and equivalently, one could say that it minimizes truth in such cases. This remark will become valuable as a point of view, when further on we will speak of minimizing positive hypothesis as a means to deal with loops over default negated literals.

The SM semantics enjoys a set of properties, namely: 1) every SM is a minimal model; 2) every SM is classically supported (Definition 6.1); and 3) the definition of SM is very simple: it only resorts to a fixed point of the Γ operator. On the other hand, there are quite a few useful properties the SM semantics lacks, namely: 1) guarantee of model existence for every NLP; 2) relevance; 3) cumulativity.
The formal definition of these properties can be found in subsection 6.5.5. For now we stay with their intuitive meaning and the possibilities they open for semantics who enjoy them: relevance means that the truth of a literal can be determined considering solely the rules it syntactically depends upon, and cumulativity means that the whole semantics of the program remains unchanged if atoms known to be *true* are added as facts (akin to storing lemmas). The lack of these properties somewhat reduces SMs applicability in practice[2], namely regarding local knowledge reasoning (cf. subsection 1.3.2) and abductive reasoning (mentioned in Section 1.2 and explored in detail in Chapter 10), creating practical difficulties in required computational processing. Top-down query-solving, a form of local knowledge reasoning, is not possible under SM semantics precisely because it does not enjoy the relevance property — and also because it does not guarantee the existence of a model. In this local form of reasoning, there is no need to compute whole models, like SM implementations do, but just the part of models that sustain the answer to the query. Relevance would ensure these could be extended to whole models.

Example 6.3. ***Stable Models semantics misses Relevance and Cumulativity.*** *Consider the program $P =$*

$$c \leftarrow not\ c$$
$$c \leftarrow not\ a$$
$$a \leftarrow not\ b \quad b \leftarrow not\ a$$

[2]In [68] the authors stress the importance of the cumulativity property and define an alternative more credulous version of this property (dubbing it *Extended Cumulativity* — ECM, for short). They also show that the SM semantics enjoys ECM although it does not enjoy cumulativity.

This program's unique SM is $\{b, c\}$, therefore both b and c are in the intersection of the models, i.e., the SM semantics considers both b and c as true. *However, $P \cup \{c\}$ has two SMs $\{a, c\}$, and $\{b, c\}$ rendering b no longer* true *in the SM semantics, which is the intersection of its models. This demonstrates that SM semantics lacks Cumulativity (cf. formal definition of cumulativity in Definition 6.36). Informally, Cumulativity means that any atom in the intersection of models can be added to the program as a fact without changing that intersection. Also, though b is* true *in P according to SM semantics, b is not* true *in $Rel_P(b) = \{a \leftarrow not\ b;\ \ b \leftarrow not\ a\}$, which shows SM semantics lacks Relevance (cf. formal definition of relevance in Definition 6.30). Informally, Relevance means that the truth-value of any atom in the intersection of models can be found by taking in consideration just the relevant rules (in the call-graph) for that atom.*

Moreover, a simple program like simply the rule $a \leftarrow not\ a$ has no SMs whatsoever, thus proving the SM semantics lacks guarantee of model existence. In fact, the SM semantics community uses this inability of SMs to assign a semantics in some cases where these kinds of Loops are part of the program as a means to impose Integrity Constraints. E.g., writing rules such as $a \leftarrow not\ a, X$, with no other rule for a. Here, the loop (with an odd number of default negated literals) over a prevents any interpretation considering X *true* from being stable according to (i.e., a fixed-point of) the Γ operator. Since the SM semantics cannot provide a semantics to this rule whenever X holds, this type of loop is used as IC. When writing such ICs under SMs one must be careful and make sure there are no other rules for a. But the really unintuitive thing about this kind of IC used under SM semantics is the meaning of the atom a. What does a represent? a is nothing but an artificially introduced and reserved atom in order to write an IC. It has no meaning in itself, nor does it represent any concept present in the knowledge being modeled.

A number tools like *SModels* [166], *DLV* [59], or *Clasp* [108] (amongst many others) are currently used to efficiently compute whole models according to the SM (or Answer-Set — AS) semantics. Given the lack of relevance of SM semantics, whole model computation is the only possible way to answer a query. Moreover, SM-based whole model computation can only take place on a fully grounded program, thereby requiring SM implementations to ground every rule with every possible combination of values for the variables. This grounding step is itself a major source of computational effort. In fact, there are even competitions for finding the most efficient grounder for SM-based systems. Once more, whole program grounding is necessary because the SM semantics lacks relevance, even when the user is not interested in finding a model for the whole program, but just in finding an answer to a query not involving the whole program. With some other 2-valued semantics enjoying relevance only the relevant part for a query would be ground, and this, depending on the proportion of the relevant part versus the whole program, may have a significant impact on the performance of practical implementations and tools.

Furthermore, in an abductive reasoning situation with SM semantics, finding an abductive answer to a query requires computing a whole model which entails pronouncement about every abducible whether or not it is relevant to the query at hand, and subsequently filtering the irrelevant ones. If the underlying semantics being used — other than the SMs — enjoys the relevance property, one can simply use a top-down proof-procedure (*à la* Prolog), and abduce by need. In this second case, the user does not pay the price of computing a whole model, nor the price of abducing all possible abducibles or their negations, and then filtering irrelevant ones, because the only abducibles considered will be those needed to answer the query.

The core reason SM semantics lacks all of these three properties (guarantee of model existence for every NLP; relevance; and cumulativity) is that the well-foundedness of classical support condition it imposes on models is impossible to be complied with on some programs with loops (Definition 2.6), or a transfinite number of layers (as in Example 3.5). Therefore, a 2-valued semantics guaranteeing model existence must not require classical support, but it may resort to the more relaxed and sensible version of layered support. And more sensible because it allows for solutions to all self-supporting loops over negation, as we shall see.

6.4.2 Three-Valued Semantics

3-valued semantics usually have an elegant solution for most of the problems the 2-valued semantics suffer: the *undefined* truth-value. By using a third *unknown*, or *undefined*, truth-value, rules like $a \leftarrow not\ a$ are neatly catered for. On the domain of 3-valued semantics, the Well-Founded Semantics is by far the most generally accepted and used one. In [21] it has been formally extended to embrace abduction.

6.4.2.1 Well-Founded Semantics

We can say the Well-Founded Semantics [109] (WFS for short) is to 3-valued semantics as Stable Models is to 2-valued semantics: they both consider as models the classically supported interpretations. However, the WFS has some highly desirable properties which SMs lack; namely, guarantee of model existence, Relevancy and Cumulativity. All of these properties are enjoyed by the WFS because it assigns a semantics to rules like $a \leftarrow not\ a$ and all other loops and also to Transfinite Support Chains, as in the case of Example 3.5. The WFS does so by resorting to the *undefined* truth-value. Moreover, the WFS has only one Model — the Well-Founded Model (WFM) — whereas the Stable Models can have several. However, the WFS in turn does not always produce a 2-valued model, though

for specific applications these may be desired, nor guarantees 2-valued model existence.

Definition 6.16. Well-Founded Model ([109]). The WFS was defined in [109] and has been studied and characterized in many ways. It being a 3-valued model we can denote the Well-Founded Model (WFM) of a program P as

$$WFM(P) =< WFM^{+}(P), WFM^{u}(P), WFM^{-}(P) >$$

where $WFM^{+}(P)$ denotes the set of atoms of P considered *true* in the WFM, $WFM^{u}(P)$ denotes the set of atoms of P considered *undefined* in the WFM, and $WFM^{-}(P)$ denotes the set of atoms of P considered *false* in the WFM.

We now recall two convenient and equivalent definitions of the WFM. One definition, taking advantage of the already defined Γ_P operator (Definition 6.15) used for the definition of Stable Model, is given in [37]: the $WMF^{+}(P)$ is the least fixed point of the Γ_P^2 operator (lemma 3 of [37]) — where $\Gamma_P^2(I) = \Gamma_P(\Gamma_P(I))$, — and all the other components of the WFM are defined using $WFM^{+}(P)$ as a basis. I.e.,

$$\begin{array}{rcl} WFM^{+}(P) & = & lfp(\Gamma_P^2) = (\Gamma_P^2)^{\omega}(\emptyset) \\ WFM^{+u}(P) & = & \Gamma_P(WFM^{+}(P)) \text{ — (lemma 2 of [37])} \\ WFM^{u}(P) & = & WFM^{+u}(P) \setminus WFM^{+}(P) = \Gamma_P(WFM^{+}(P)) \setminus WFM^{+}(P) \\ WFM^{-}(P) & = & \mathcal{H}_{\mathcal{P}} \setminus WFM^{+u}(P) \end{array}$$

where $\mathcal{H}_P$ is the Herbrand Base of P.

The other alternative definition of WFM, theorem 5.19 of [45], resorts to the program Remainder transformation (cf. Definition 6.13) as follows:

$$\begin{array}{rcl} WFM^{+}(P) & = & facts(\widehat{P}) \\ WFM^{+u}(P) & = & heads(\widehat{P}) \\ WFM^{u}(P) & = & WFM^{+u}(P) \setminus WFM^{+}(P) = heads(\widehat{P}) \setminus facts(\widehat{P}) \\ WFM^{-}(P) & = & \mathcal{H}_P \setminus WFM^{+u}(P) = \mathcal{H}_P \setminus heads(\widehat{P}) \end{array}$$

It follows immediately from both alternative definitions that $facts(\widehat{P}) = (\Gamma_P^2)^{\omega}(\emptyset)$ and $heads(\widehat{P}) = \Gamma_P(facts(\widehat{P}))$.

WFS enjoys the properties of relevance, cumulativity, guarantee (and uniqueness) of model existence. Moreover, [45] shows that computing the WFM has polynomial time complexity, and [232] hints that in some cases it can even be computed in linear time (in particular, when there are no SCCs in the $ERG(P)$ and, therefore, no need to perform

the "loop detection–elimination of *unfounded sets*" step which is the only step of $\widehat{P}$ with a higher-than-linear complexity) [232].

The XSB-Prolog system [219] actually implements an incremental version of the Program Remainder operator via the special predicate $get\_residual/2$. The "*get_***residual**" naming comes from the fact that, originally and for efficiency reasons, this predicate computed only the Program Residual, not the Program Remainder. In a nutshell, the difference between the Program Residual and the Program Remainder is that the former does not perform the *loop detection*, but resorts to *program unfolding* of positive literals and deletion of tautologies (rules that have $head(r) \in body^{+}(r)$). Further details on the differences between these operators can be found in [45]. In [232] and Section 11.1 we describe our contribution to the implementation of the *loop detection* mechanism in XSB-Prolog.

The extension of WFS to Extended Logic Programs (ELPs) became known as WFSX [176] — *Well-Founded Semantics with eXplicit negation*. Taking the extension process one step further, a para-consistent version of the WFSX — the WFSXp [72] — has also been defined. Under this new semantics, even if an atom and its explicit negation are both true in the model, the truth-value of all atoms that do not depend on any of those contradictory ones is not affected. Para-consistent Logic Programs [6] has been a fruitful area of research and this could be an interesting possibility for future work under the new semantics we propose in the sequel. As mentioned before, the abductive extension of WFS is covered in [21].

6.4.2.2 Layered Well-Founded Semantics

Since one definition of the Well-Founded Model (theorem 5.19 of [45]) is based upon the Program Remainder operator (Definition 6.13) we can *mutatis mutandis* define a *layered* version of the Well-Founded Semantics based upon the Layered Program Remainder operator (Definition 6.14).

Definition 6.17. Layered Well-Founded Model. The Layered WFS is a 3-valued semantics with the Layered Well-Founded Model (LWFM) of a program P being defined as

$$LWFM(P) =< LWFM^{+}(P), LWFM^{u}(P), LWFM^{-}(P) >$$

where

$$\begin{array}{rcl} LWFM^{+}(P) & = & facts(\mathring{P}) \\ LWFM^{+u}(P) & = & heads(\mathring{P}) \\ LWFM^{u}(P) & = & LWFM^{+u}(P) \setminus LWFM^{+}(P) = heads(\mathring{P}) \setminus facts(\mathring{P}) \\ LWFM^{-}(P) & = & \mathcal{H}_P \setminus LWFM^{+u}(P) = \mathcal{H}_P \setminus heads(\mathring{P}) \end{array}$$

Theorem 6.2. ***The Layered Well-Founded Model is more skeptical than the Well-Founded Model.*** *Let P be an NLP. Then*

$$\begin{array}{rcl} LWFM^{+}(P) & \subseteq & WFM^{+}(P) \wedge \\ LWFM^{u}(P) & \supseteq & WFM^{u}(P) \wedge \\ LWFM^{-}(P) & \subseteq & WFM^{-}(P) \end{array}$$

Taking a 3-valued interpretation perspective of both the WFM and LWFM, we have

$$\Big(LWFM^{+}(P) \cup LWFM^{-}(P)\Big) \leq_F \Big(WFM^{+}(P) \cup WFM^{-}(P)\Big)$$

according to Definition 5.11.

6.4.2.3 Baral and Subrahmanian's Stable Classes

In [36, 37], Baral et al. defined the Stable Classes (SCs) semantics, providing yet another approach to three-valued semantics. Let us recall here the definition of SC resorting to the definition of graph of interpretations.

Definition 6.18. *graph*(P) **(definition 2.4 of [36]).** Suppose P is a Logic Program. Then *graph*(P) is a directed graph which is defined as follows:

- The vertices of *graph*(P) are the interpretations of our language
- There is an arc from interpretation I to interpretation J iff $\Gamma_P(I) = J$

Definition 6.19. Stable Class. Theorem 3 of [36] says that, for programs with a finite Herbrand Base, $\mathcal{I} = \{I_1, \ldots, I_n\}$ is a non-empty strict stable class of P iff $\mathcal{I}$ is a cycle in *graph*(P); i.e., $I_{i+1} = \Gamma_P(I_i)$, and $I_1 = \Gamma_P(I_n)$.

An important result from [37] established the relationship between the WFS and the Stable Classes. In a nutshell, theorem 4 of [37] says the WFS is captured by a particular stable class C of P, more concretely $C = \{WFM^{+}(P), WFM^{+u}(P)\}$.

Like any other 3-valued semantics, the Stable Classes semantics resorts to the *undefined* truth-value to deal with loops over default negation. Depending on ones' goals, that might be exactly the desired outcome; but on the other hand, whenever a 2-valued complete model is always necessary, a Stable Class might be a starting point, but, in general, not the end result.

6.4.2.4 Dung's Preferred Extensions

In [91] the author introduces the Preferred Extensions, which generalize SMs to the 3-valued case. Most of the ideas and notions underlying the work we next present originate from the Argumentation field — mainly from that foundational work of Phan Minh Dung. For self-containment we provide the basic notions of argument (or set of hypotheses), attack, conflict-free set of arguments, acceptable argument, and admissible set of arguments (all originally from [91]). These notions serve as background for our argumentation approach below where we shall introduce a 2-valued conservative extension of Dung's Preferred Extensions.

The relationships between argumentation and logic programs has been covered in many other works, e.g., [15] which extends the argumentation framework for Extended Logic Programs and consider other variants w.r.t. NLPs.; [167] that takes an argumentation approach to semantics of logic programs; and [42] where the authors take a general argumentation approach to default reasoning.

Definition 6.20. Argument. In [91] the author presents an argument as

> *"an abstract entity whose role is solely determined by its relations to other arguments. No special attention is paid to the internal structure of the arguments."*

In this thesis, since we focus on Normal Logic Programs, we will pay special attention to the internal structure of an argument. One common approach to arguments, when considering NLPs are their representation system, is to consider an *argument* (or set of hypotheses) as a set H of default negated literals of P, i.e., $H \subseteq not\ \mathcal{H}_P$. Thus, a *simple argument not a* of H (or simple hypothesis) is just an element of an *argument* H.

Using these notions of *argument* and *simple argument* we can define the set of *Arguments* of P — $Arguments(P)$ — as the set of all *arguments* of P, i.e., the set of all subsets of $not\ \mathcal{H}_P$.

Definition 6.21. Attack — Argument B attacks simple argument *not a* in P [91]. In [91] Dung does not specify what the *attacks* relationship concretely is, this

way ensuring maximal generality of the argumentation framework. In our present work, since we are considering NLPs, and *arguments* as sets of default negated literals (each a *simple argument*), the *attacks* relationship corresponds to the entailment of a positive literal contradicting one *simple argument* of the attacked *argument*.

Formally, if P is a NLP, $B \in Arguments(P)$, and $not\ a \in not\ Atoms(P)$, we say B attacks $not\ a$ in P iff $P \cup B \models a$[3]. For simplicity, we just write $attacks_P(B, not\ a)$.

Abusing notation, we also write $attacks_P(B, A)$, where both A and B are *Arguments* of P, to mean that *Argument* B attacks *Argument* A in P. This means that $\exists_{not\ a \in A} attacks_P(B, not\ a)$.

Definition 6.22. Conflict-free argument A [91]. An *argument* A of P is said to be conflict-free iff there is no *simple argument* $not\ a$ in A such that $attacks_P(A, not\ a)$. I.e., A does not attack itself.

Definition 6.23. Acceptable argument [91]. An argument $A \in AR$ — where AR is a set of *arguments* — of P is said to be Acceptable with respect to a set S of arguments iff for each argument $B \in AR$: if B attacks A then B is attacked by S.

Definition 6.24. Admissible argument A of P [91]. A conflict-free *argument* A is *admissible* in P iff A is acceptable with respect to $Arguments(P)$. Intuitively, A is *admissible* if it counter-attacks every argument B in $Arguments(P)$ attacking A. Formally,

$$\forall_{B \in Arguments(P)} \forall_{not\ a \in A} attacks_P(B, not\ a) \Rightarrow \exists_{not\ b \in B} attacks_P(A, not\ b)$$

This definition corresponds to the definition of Admissible Set presented in [91]. Notice that it is not required attacking set B to be consistent with its consequences.

Definition 6.25. Preferred Extension [91]. A Preferred Extension is a maximal (with respect to set inclusion) admissible Argument of P.

In [91] the author also shows that "*Every argumentation framework possesses at least one preferred extension.*" (Corollary 12 of [91]) and "*Hence, preferred extension semantics is always defined for any argumentation framework.*"

The relation between preferred extensions and stable models is then analyzed in:

Definition 6.26. Stable Extension (Definition 13 of [91]). A conflict-free set of arguments S is called a stable extension iff S attacks each argument which does not belong to S.

[3]Where we transform all literals of the form $not\ a$ in new positive literals not_a and then, by virtue of ending up with a definite program, we can resort to the *least model* to determine entailment.

and in:

Lemma 6.3. ***Stable Extensions are Preferred Extensions(Lemma 15 of [91]).*** *Every stable extension is a preferred extension, but not vice versa.*

Finally, the author of [91] also says:

> *"Though stable semantics is not defined for every argumentation system, an often asked question is whether or not argumentation systems with no stable extensions represent meaningful systems? (...) we will provide meaningful argumentation systems without stable semantics, and thus provide a definite answer to this question."*

6.4.2.5 Our Argumentation Based Semantics

Dung's Preferred Extensions covered in 6.4.2.4 extend the Stable Models in the sense that every SM corresponds to a Stable Extension, and each Stable Extension is a Preferred Extension; moreover, every NLP has at least one Preferred Extension. However, every Preferred Extension that is not a Stable Extension (and, therefore, not a Stable Model) leaves *undefined* the literals that are not stable, i.e., those that attack themselves. For this reason, not all Preferred Extensions yield 2-valued complete models.

Assuming only negative hypotheses thus proved to be insufficient for guaranteeing 2-valued completeness of models. To respond to this lack we decided to take the next step by allowing also positive hypotheses in our argumentation approach. In order to keep consistency with the skeptical stance of maximal negative hypotheses, we had to explicitly impose its dual: minimality of positive hypotheses. Our first attempt was [189] where we also take the hypotheses assumption approach to semantics, applying it within an argumentation context and demanding *non-redundancy* (Definition 11 of [189]) and *unavoidability* (Definition 12 of [189]) of the set of *positive hypotheses* (positive literals assumed as hypotheses) as well *weak admissibility* (Definition 10 of [189]) of the set of *negative hypotheses* in order to construct a *Revision Complete Scenario* (Definition 13 of [189]). Therein we defined a Revision Complete Scenario

Definition 6.27. Revision Complete Scenario (Definition 13 of [189]). Let P be a NLP and $H = H^+ \cup H^-$ a set of positive (H^+) and negative (H^-) hypotheses. We say H is a Revision Complete Scenario iff

1. $P \cup H$ is a consistent scenario and $least(P \cup H)$ is a 2-valued complete model of P

2. H^- is weakly admissible
3. H^+ is not redundant
4. H^+ is unavoidable

The intuition for a Revision Complete Scenario goes as follows: in order to guarantee the Existence of a 2-valued total model for every NLP we allow positive hypotheses to be considered besides the usual negative hypotheses. Under this setting, the easiest way to solve the problem would be to accept every atom of a program as a positive hypotheses. However, we want our argumentation based semantics to be the most skeptical possible while ensuring compatibility among hypotheses.

To keep the semantics skeptical we want to have the maximal possible negative hypotheses and the minimum non-redundant positive hypotheses. Intuitively, a positive hypothesis L is considered redundant if, by the rules of the program and the rest of the hypotheses, L is already determined *true*; and a positive hypothesis L is considered unavoidable if, the least model of the program plus the rest of the hypotheses minus L is inconsistent; and a set H^- of negative hypotheses is weakly admissible iff for each counter-hypothesis E contradicting H^- we know that $P \cup H^- \cup E$ contradicts E. The formal details of these and other notions we appeal to can be found in [189]; we do not enter here their specific details as it would divert the reader too much from the main thread of this thesis.

Later, in [188], and building upon our work of [189], we defined the Approved Models semantics for NLPs taking again an argumentation approach that generalizes the Preferred Extensions approach discussed in 6.4.2.4 in the sense that all Approved Models are 2-valued complete containing a Preferred Extension.

Definition 6.28. Approved Models (Definition 13 of [188]). Let P be a NLP and $M = M^+ \cup M^-$ a 2-valued interpretation of P. We say M is an Approved Model of P iff:

- M is an Approvable Interpretation of P, and
- M^- contains a Preferred Extension of P

where M is an Approvable Interpretation iff M^- is maximal given $least(P \cup M) \subseteq M$.

Although the Approved Models was another significant step forward it still had a complex definition resorting to both negative and positive hypotheses. We then realized we could solve the argumentation/semantics issue by virtue of a new paradigm: that of positive hypotheses assumption alone, which is the topic of Chapter 8.

6.5 Motivation for a New Semantics

"Why the need for another 2-valued semantics for NLPs since we already have the Stable Models one?" This is the question we heard the most whilst pursuing this avenue of research. The question has its merit since the Stable Models semantics is exactly what is necessary for so many problem solving issues, but the answer to it is best understood when we ask it the other way around: "Is there any situation where the Stable Models semantics does not provide all the intended models?" and "Is there any 2-valued generalization of SMs that keeps the intended models it does provide, adds the missing intended ones, and also enjoys the useful properties of guarantee of model existence, relevance, and cumulativity?" When considering to answer these other questions there are several approaches we can take.

6.5.1 Increased Declarativity

As we have seen in Definition 3.2, an IC is a rule whose head is $\bot$, and although such syntactical definition of IC is generally accepted by the Logic Programming community, the SM semantics employs odd loops over negation, such as the $a \leftarrow not\ a, X$ as discussed after Example 6.3, to act as ICs, thereby mixing and unnecessarily confounding two distinct Knowledge Representation issues: the one of IC use, and the one of assigning semantics to loops. For the sake of declarativity, our position is that rules of the form in Definition 3.2 should be the only way to write ICs in a Logic Program: no rule, or combination of rules, with head different from $\bot$ should act as IC(s) under any given semantics. Rules with "non-$\bot$" head should only establish what can/must be true in a model given some conditions stated in the body, and rules with $\bot$ head should only establish which models, although complying with the "non-$\bot$" head rules, must be discarded anyway because of IC violation. To allow each of these kinds of rules to play the role of the other is to sacrifice and withhold a degree of freedom from declarativity. If it were really indifferent and acceptable the use of one or the other kind of rule for both KRR purposes, then there would be no need for two different kinds of rules. But, as we shall see, that is not the case.

2-valued model existence should only be "threatened" in Constrained NLPs (CNLPs), but never in other NLPs, because only "$\bot$-head" rules should play the role of rejecting candidate models.

6.5.2 Modelling Argumentation

Another insight, from an argumentation perspective, comes from [91], where the author states:

> *"Stable extensions do not capture the intuitive semantics of every meaningful argumentation system."*

where the "stable extensions" have a one-to-one correspondence to the Stable Models ([91]), and also

> *"Let P be a knowledge base represented either as a logic program, or as a nonmonotonic theory or as an argumentation framework. Then there is not necessarily a "bug" in P if P has no stable semantics.*
>
> *This theorem defeats an often held opinion in the logic programming and nonmonotonic reasoning community that if a logic program or a nonmonotonic theory has no stable semantics then there is something "wrong" in it."*

Thus, a criterium different from the *stability* one must be used in order to model argumentation more adequately.

In 6.4.2.4 and 6.4.2.5 we saw different approaches to argumentation based semantics. Stable Models are known to correspond to Stable Extensions which are a particular case of Preferred Extensions, and these stem from a negative hypotheses assumption approach. Revision Complete Scenarios and Approved Models generalize the Preferred Extensions approach to include also positive hypotheses. The semantics we propose in Chapter 8, when seen from an argumentation stance, solves the semantics problem by resorting to positive hypotheses only.

6.5.3 Allowing Arbitrary Updates and/or Merges

One of the main goals behind the conception of non-monotonic logics was the ability to deal with the changing, evolving, updating of knowledge. There are scenarios where it is possible and useful to combine several Knowledge Bases (possibly from different authors or sources) into a single one [34, 78, 76, 130, 245], and/or to update a given KB with new knowledge [12, 137]. Assuming the KBs are coded as IC-free NLPs, as well as the updates, the resulting KB is also an IC-free NLP. In such a case, the resulting (merged and/or updated) KB should always have a semantics. The lack of such guarantee when the

underlying semantics used is the Stable Models, for example, compromises the possibility of arbitrarily updating and/or merging KBs (coded as IC-free NLPs). In the case of self-updating programs, the desirable "liveness" property is put into question, even without outside intervention.

Example 6.4. ***Self-updating program with embedded odd-loop.*** *Let us consider the case where we use the Stable Models semantics as the underlying one for this EVOLP [8] program.* $P =$

$$\begin{array}{lll} a & \leftarrow & not\ b \\ assert(b \leftarrow not\ c) & \leftarrow & not\ d \\ assert(assert(c \leftarrow not\ a)) & \leftarrow & not\ e \end{array}$$

EVOLP programs have the ability to update themselves without exterior intervention, and that is precisely what happens when there is an EVOLP program rule of the form $assert(X) \leftarrow Y$. *In the next time step,* X *becomes a new rule of the program if* Y *holds in the previous time step model, whilst the assert rule (like any other) is kept by inertia unless retracted.*

Initially, the only Stable Model of this program is $\{a, assert(b \leftarrow not\ c), assert(assert(c \leftarrow not\ a))\}$, *since none of* $\{b, d, e\}$ *hold. In EVOLP, whatever assert term true in a model is then used to assert its rule argument to produce the next program state by updating it with that rule. Since* d *does not hold in the initial model, the new rule* $b \leftarrow not\ c$ *is added to the program. Also, since* e *does not hold in the initial stable model of the program, the new rule* $assert(c \leftarrow not\ a)$ *is added to it. In a nutshell, in the second time step the program becomes*

$$\begin{array}{lll} a & \leftarrow & not\ b \\ b & \leftarrow & not\ c \\ assert(b \leftarrow not\ c) & \leftarrow & not\ d \\ assert(assert(c \leftarrow not\ a)) & \leftarrow & not\ e \\ assert(c \leftarrow not\ a) & & \end{array}$$

and its single stable model is $\{b, assert(b \leftarrow not\ c), assert(assert(c \leftarrow not\ a)), assert(c \leftarrow not\ a)\}$. *A new evolution step occurs, due to the 'assert' atoms in the model, and after the third time step the program becomes*

$$\begin{array}{lll} a & \leftarrow & not\ b \\ b & \leftarrow & not\ c \\ c & \leftarrow & not\ a \quad \%\ \textit{from the last rule in the prior program state} \\ assert(b \leftarrow not\ c) & \leftarrow & not\ d \\ assert(assert(c \leftarrow not\ a)) & \leftarrow & not\ e \\ assert(c \leftarrow not\ a) & & \end{array}$$

Redundant effects of 'asserts' are discarded, i.e., asserting a rule already in the program results in not adding the duplicate. This is the time step where the problem arises. Under Stable Models semantics this EVOLP program no longer has a model due to the first three rules constituting an odd loop over negation. This odd-loop was somehow camouflaged by the 'assert' EVOLP rules in the first time step program. As the natural evolution of the program unfolded, the odd-loop reached the 'surface' and revealed itself, thereby terminating the liveliness of the programs' evolution by moving it into a state with no semantics.

The EVOLP mechanism is but one of many possible ways to (self-)update a logic program, but it is enough to illustrate the need to guarantee model existence for updatable knowledge bases.

Example 6.5. ***A Joint Vacation Problem — Merging Logic Programs.*** *Three friends are planning a joint vacation. First friend says "If we don't go to the mountains, then we should go to the beach". The second friend says "If we don't go travelling, then we should go to the mountains". The third friend says "If we don't go to the beach, then we should go travelling". We code this information as the following NLP:*

$$\begin{aligned} beach &\leftarrow not\ mountain \\ mountain &\leftarrow not\ travel \\ travel &\leftarrow not\ beach \end{aligned}$$

Each of these individual consistent rules come from a different friend. According to the Stable Models semantics, each friend had a "solution" (a stable model) for its own rule, but when we put the three rules together, because they form an odd loop over negation, the resulting merged logic program has no stable model. In this case our intuition tells us this program should have at least one (or possibly several alternative) model(s) corresponding to joint vacation solution(s). This example too shows the importance of having a 2-valued semantics guaranteeing model existence, in this case for the sake of arbitrary merging of logic programs (and for the sake of existence of a joint vacation for these three friends).

The examples just shown are quite simple, but they make evident that there a need for a new 2-valued semantics guaranteeing model existence. Such a semantics will enable "KB as NLP"-based systems to be safely used no matter how complex the series of merges and/or updates it receives, and it will also contribute to guaranteeing the system's liveness. This is especially important for the Semantic Web [10] applications which may come to rely on updatable (possibly self-updatable) logic-based web pages.

These three (6.5.1, 6.5.2, and 6.5.3) motivational issues raise the questions "Which should be the 2-valued models of an NLP when it has no Stable Models?", "How do

these relate to SMs?", "Is there a uniform approach to characterize both such models and the SMs?", and "Is there any 2-valued generalization of the SMs that encompasses the intuitive semantics of *every* logic program?". To answer these questions is a paramount motivation and thrust behind this thesis.

After some previous attempts [187, 193, 204] which aimed at answering the first two of these questions, we have gotten a better understanding of what the answers to the third and fourth ones should be, and of how to put them across. This new knowledge lead us to an intuitive notion of how a 2-valued semantics should behave for all NLPs, what properties it should enjoy, and how its meaning assignment should take place. We analyze these in turn now.

6.5.4 Intuitively Desired Semantics

It is commonly accepted that the non-stratification of the default *not* is the fundamental ingredient which allows for the possibility of existence of several models for a program, as we have seen in subsections 6.3.1 and 6.3.2. The non-stratified DNLs (i.e., in a loop — Definition 2.6) of a program can thus be seen as non-deterministically assumable choices. The rules in the program, as well as the particular semantics we wish to assign them, is what constrains which sets of those choices we take as acceptable.

Usually, both the Closed World Assumption (CWA) imposed on default negation *not* and the $\leftarrow$ in rules of Logic Programs reflect some intended ordering in the truth-value assignment to literals. E.g., in a program with just the rule $a \leftarrow not\ b$ we first assign the truth-value *false* to b because it has no rules (that is what the CWA does), and then, as a consequence of b's assumed truth-value, we are forced, by virtue of the rule $a \leftarrow not\ b$, to conclude a's truth-value must be *true*; hence, the only intended model is $\{a\}$. This is afforded by the syntactic asymmetry of the rule, reflected in the one-way direction of the $\leftarrow$, coupled with the intended semantics of CWA applied to default negation.

If we were to re-write the rule $a \leftarrow not\ b$ in classical logic form we would get $a \Leftarrow \neg b$, which, as we know, is equivalent to $a \vee \neg\neg b$, i.e., $a \vee b$. And $a \vee b$ can be (classically) satisfied by any of the models $\{a\}$, $\{b\}$, and $\{a, b\}$. Clearly, the last of these is not minimal considering the first two alternative models; these are thus the only candidate models. But non-monotonic logic, the formalism used in Normal Logic Programs, is not the same as classical logic, in part precisely because default negation is not the same as classical negation (nor is the $\leftarrow$ the same as $\Leftarrow$). In NLPs we adopt the CWA principle, a fair guideline underlying the rationale of a reasonable semantics for NLPs, which states we always should assume as false atoms with no rules. This principle rejects $\{b\}$ as a model of program $a \leftarrow not\ b$.

When rules form loops, the syntactic asymmetry disappears and, as far as the loop alone is concerned, the truth value of the negative literals in the loop are equally assumable choices, and that is why they become *undefined* in the Well-Founded Model of the program. If we had the program $P = \{a \leftarrow not\ b \quad , \quad b \leftarrow not\ a\}$, there would be no specific order in which to proceed in the truth-value assignment to literals, as far as a and b are concerned — we could legitimately start by assuming a *true* and propagate the consequences of that assumption to constraint b's truth-value to *false*, or vice-versa. In this case P would have two alternative models: $\{a\}$ and $\{b\}$. If we were to re-write this program in classical logic form we would get $P = \{a \Leftarrow \neg b \quad , \quad b \Leftarrow \neg a\}$, which we know would be equivalent to $P = \{a \vee \neg\neg b \quad , \quad b \vee \neg\neg a\}$, i.e., $P = \{a \vee b \quad , \quad b \vee a\}$, or simply $P = \{a \vee b\}$. We already know that $a \vee b$ has two minimal models: $\{a\}$ and $\{b\}$. It is no coincidence that both the original program with a loop $P = \{a \leftarrow not\ b \quad , \quad b \leftarrow not\ a\}$ and its classical logic representation $P = \{a \vee b\}$ have exactly the same minimal models: $\{a\}$ and $\{b\}$. The syntactic symmetry of the loop thus induces a semantic symmetry on the truth values of the literals involved, whether positive or negative.

However, the unidirectional way of the $\leftarrow$s of all the loop's rules must still be respected as soon as the atom of one given DNL is assumed *true*. Intuitively, assuming *true* the atom of one DNL of the loop should have the practical effect of "cutting" one of the loop's arcs thereby rendering the set of rules an asymmetric chain and, consequently, constraining the truth values of the remaining literals (including the DNLs) in the (former) loop. The semantic symmetry of the loop comes from the initial freedom in choosing any one DNL to "break" the loop. However, loops may also depend on other literals with which they form no loop. Those asymmetric dependencies should have the same semantics as the single $a \leftarrow not\ b$ rule case described previously. Of course, because of symmetry, it does not matter whether an atom or its default literal is chosen as true in order to break the loop. However, because heads of rules are atoms only, choosing a DNL true may lead to inconsistency, as in $a \leftarrow not\ a$, so pragmatically it is better to concentrate on choosing atoms true. But the symmetry lets us see the parallel with the view of DNLs as adoptable assumptions, or as abducibles, whose overall minimality must be required and tested.

So, on the one side, asymmetric dependencies should have the semantics of a single $a \leftarrow not\ b$ rule; and on the other, the symmetric dependencies (of *any* loop over negation, whether an odd or even one) should subscribe to the semantic symmetry principle of assuming *true* the atom of any one of the loop's DNLs and extracting the consequences. Intuitively, a good semantics should cater for both the symmetric and asymmetric dependencies as described. Asymmetry in favor of DNLs is desirable in the absence of (remaining) rules for their atoms, and symmetry desirable otherwise.

As seen above, ICs are commonly used as a simple and declarative way of imposing the truthfulness (or falsehood, for that matter) of chosen conjunctions of literals. The

correct way of writing ICs is, as recalled in Definition 3.2, by means of rules with $\bot$ for head. A "good" semantics should allow this kind of IC and no other. I.e., no other kind of "non-$\bot$" rule, or combination of such rules, should be diverted to function as IC lest it undermine the declarativity of the knowledge representing rules with a "non-$\bot$" head, and their use for KR; e.g. loops like $a \leftarrow not\ a$ must solely be usable as means to engender solutions, not as a means to prune them. E.g., a program with the single rule $a \leftarrow not\ a$ should have the model $\{a\}$.

6.5.5 Desirable Formal Properties

Depending on the particular problem solving task at hand, there are several formal properties of the engaged underlying semantics that can be quite useful. For example, one might require a 2-valued answer to a query, or, on the contrary, be satisfied with a 3-valued one. One might need the guarantee that no matter what the knowledge encoded in the NLP it will have a semantics, at least one model; or, to the contrary, it might be acceptable that a certain KB simply has no models. One might want to be able to solve queries in a top-down fashion (*à la* Prolog) and be content with model subsets supporting the query; or one might want to compute whole models for the entire NLP — a dichotomy not unlike the local/global knowledge reasoning scope requirements identified in 1.3.1 and 1.3.2. We might wish to have the possibility of storing lemmas (results from previous computations) in, say, a table, to speed-up later computations.

The primary focus of this thesis is to provide a new 2-valued semantics for NLPs which, by virtue of the properties it enjoys, lays the theoretical ground for practical and computationally efficient Knowledge Representation and Reasoning with NLPs and their extensions including Integrity Constraints (as in Definition 3.2), abduction, argumentation, explicit negation and disjunction.

The new semantics should abide by the intuitive caveats described in the previous subsection 6.5.4, and also enjoy several practical properties such as guarantee of model existence, allow for the development of top-down proof-procedures to answer queries, allow for the storage of lemmas for speeding up computations, and being a generalization of stable models by taking all of them, if any, as models also.

6.5.5.1 Model Existence

Guarantee of model existence ensures all programs without ICs have a semantics, i.e., at least one model. Of course, programs with ICs may indeed have no model simply because the combination of ICs therein may utterly prevent them, unless a paraconsistent

semantics is being considered. This property, required of ICs-free programs, is especially important when building a system or a service grounded on (self- and/or externaly-) updatable knowledge bases. Any system built upon a reasoning engine using a semantics not guaranteeing model existence may simply not work at all if the underlying knowledge base has no model according to the semantics employed. Even if the initial KB has a model according to the semantics, that may no longer be the case after an unpredictable sequence of updates. Putting the issue on security matters terms, using a semantics not enjoying guarantee of model existence is allowing a major security hole: an ill-intended agent (or even a well-intended one making a mistake) can update the KB in such a way that it would no longer have a model, thereby breaking down the whole system. Therefore, a semantics guaranteeing model existence will be more theoretically robust, providing an additional level of security.

Intersection of models There is a set of formal properties (discussed below) of a 2-valued semantics that are defined over the intersection of all models of the semantics. Before we proceed introducing such properties we must, therefore, formalize the notion of intersection of 2-valued models. In general, 2-valued semantics accept several models for a given program. From these models one can extract a unique skeptic 3-valued model resulting from the intersection.

Definition 6.29. Skeptic 3-valued model of a 2-valued semantics. Let P be an NLP, Sem a 2-valued semantics for NLPs, and $Models_{Sem}(P)$ the set of 2-valued models of P according to Sem (cf. Definition 5.12).

The unique 3-valued model of P according to Sem is defined as

$$Sem_{3v}(P) =< Sem^{+}_{3v}(P), Sem^{u}_{3v}(P), Sem^{-}_{3v}(P) >$$

where

$$\begin{array}{lcll} Sem^{+}_{3v}(P) & = & \bigcap\limits_{M \in Models_{Sem}(P)} M^{+} & \text{the } \textit{true} \text{ atoms according to } Sem \\ Sem^{-}_{3v}(P) & = & |\bigcap\limits_{M \in Models_{Sem}(P)} M^{-}| & \text{the } \textit{false} \text{ atoms according to } Sem \\ Sem^{u}_{3v}(P) & = & \mathcal{H}_P \setminus (Sem^{+}_{3v}(P) \cup Sem^{-}_{3v}(P)) & \text{the } \textit{undefined} \text{ atoms according to } Sem \end{array}$$

Naturally, the existence of such a 3-valued model is conditioned upon 2-valued model existence. The relevance (and cumulativity, amongst others) properties of a semantics, as defined in [85], pertain to such 3-valued model induced by the 2-valued models of the semantics. The definitions of these properties, being focused on the 3-valued model, turns out to be weak for our purposes: we are more interested in these properties at the level of each individual model. Therefore, besides accounting for the properties in [85], we also define and cater for their corresponding individual 2-valued model versions.

6.5.5.2 Relevance

Relevance ([85]) concerns simple (object-level) top-down querying about truth of a query in the program (like in Prolog) without requiring production of the whole 3-valued model, just the part of it supporting the call-graph rooted on the query. Formally:

Definition 6.30. Relevance. A semantics Sem for logic programs is said Relevant iff for every program P

$$\forall_{a \in \mathcal{H}_P} a \in Sem(P) \Leftrightarrow a \in Sem(Rel_P(a))$$

Since this definition (from [85]) had 3-valued models in mind, using the notation in accordance to Definition 6.29, we can re-write it as

$$\forall_{a \in \mathcal{H}_P} a \in Sem^+_{3v}(P) \Leftrightarrow a \in Sem^+_{3v}(Rel_P(a))$$

which is also equivalent to

$$\forall_{a \in \mathcal{H}_P} (\forall_{M \in Models_{Sem}(P)} a \in M) \Leftrightarrow (\forall_{M_a \in Models_{Sem}(Rel_P(a))} a \in M_a)$$

Relevance, because it applies to the intersection of all 2-valued models, ensures Cautious Reasoning (as discussed in Chapter 1) can take place considering only the relevant part for the query and not, in general, the whole program, unless, of course, they coincide. I.e., with a Relevant semantics, an atom is *true* in *all* the models of the whole program iff it is *true* in *all* the sub-models of the part of the program *relevant* to the atom.

We now present the Brave counterpart of Relevance — the Brave Relevance — which, *mutatis mutandis*, ensures Brave (or Credulous) Reasoning can take place considering only the relevant part for the query.

Definition 6.31. Brave Relevance. Let P be an NLP, a an atom of P, M a model of P according to semantics Sem, and M_a a model of $Rel_P(a)$ according to semantics Sem. A semantics Sem for logic programs is said Brave Relevant iff

$$\forall_{a \in \mathcal{H}_P} \Big(\forall_{M \in Models_{Sem}(P)} a \in M \Rightarrow (\exists_{M_a \in Models_{Sem}(Rel_P(a))} M_a \subseteq M \wedge a \in M_a) \Big)$$

$$\wedge$$

$$\Big(\forall_{M_a \in Models_{Sem}(Rel_P(a))} a \in M_a \Rightarrow \exists_{M \in Models_{Sem}(P)} M_a \subseteq M \Big)$$

I.e., with a Brave Relevant semantics, an atom is *true* in *some* model of the whole program iff it is *true* in *some* sub-model of the part of the program *relevant* to the atom,

where that sub-model is a subset of a model for the whole program where the atom is *true*.

Brave Relevance ensures both 1) that if a query is true in some complete model, then it is also true in some submodel for the subset of the program relevant to the query and 2) that any submodel supporting the query's truth can always be extended to a complete model. This way, Brave Relevance guarantees that finding solutions to queries can be done resorting only to the relevant call-graph and that any solution found is part of a complete model. Moreover, Brave Relevance lays the cornerstone for abduction by need, in that only abducibles in the call-graph need be considered for abduction, but we discuss abductive reasoning further in Chapter 10, and the concomitent implementation issues in Chapter 11.

Proposition 6.4. *Brave Relevance implies Relevance.* *If a semantics Sem enjoys Brave Relevance, then it also enjoys Relevance.*

This means that whenever a semantics enjoys Brave Relevance, i.e., it allows the truth-value of a literal in a model to be determined using only the relevant part for that literal, then the semantics also enjoys Relevance, i.e., it allows the truth-value of a literal which is common to *all* models to be determined using only the same relevant part.

6.5.5.2.1 Relevant Answer to Query The Relevance and Brave Relevance properties are useful for a semantics in the sense that they allow the possibility of specification and development of top-down proof-procedures that can be used to answer queries resorting only to the partial knowledge relevant for the query. By virtue of requiring only that partial knowledge, such query-answering methods are potentially more efficient than computing complete knowledge models for the whole program which might need also to check whether the computed whole model entails the query or not.

Definition 6.32. Relevant Partial Knowledge Answer to a Query. Let P be an NLP, $Q = a_1 \wedge \ldots \wedge a_n \wedge not\ b_1 \wedge \ldots \wedge not\ b_m$ a conjunction of literals (with $n, m \geq 0$ and at least one literal) which we call a *query*, S a semantics for NLPs enjoying Relevance and Brave Relevance. We write SQ to denote the set of all the literals in Q, i.e., $SQ = \{a_1, \ldots, a_n, not\ b_1, \ldots, not\ b_m\}$. Then, M_Q is said to be a *relevant partial knowledge answer to* Q iff

- M_Q is a model of $Rel_P(Q)$ (cf. Definition 3.11) according to S;
- $M_Q \models Q$ in the sense that $M_Q \supseteq SQ$;
- There is some model M of P according to S such that $M \supseteq M_Q$

In this sense, a relevant partial knowledge answer to Q is a sub-model of P — indeed, a model of $Rel_P(Q)$, which is a subset of P — that entails the query.

For the general case of Constrained NLPs, model existence is not guaranteed and, therefore, no non-paraconsistent semantics can enjoy Relevance nor Brave Relevance. However, for paraconsistent semantics[4], model existence can be guaranteed as well as Relevance and Brave Relevance, depending, of course, on the definition of the semantics. For paraconsistent semantics we additionally define:

Definition 6.33. Locally Consistent Relevant Partial Knowledge Answer to a Query. Let $\mathcal{P} = P \cup \mathcal{C}$ be a CNLP, $Q = a_1 \wedge \ldots \wedge a_n \wedge not\ b_1 \wedge \ldots \wedge not\ b_m$ a conjunction of literals (with $n, m \geq 0$ and at least one literal) which we call a *query*, S a paraconsistent semantics for CNLPs enjoying Relevance and Brave Relevance. We write SQ to denote the set of all the literals in Q, i.e., $SQ = \{a_1, \ldots, a_n, not\ b_1, \ldots, not\ b_m\}$. Then, M_Q is said to be a *locally consistent relevant partial knowledge answer to* Q (abbreviated as *locally consistent answer* to Q) iff

- M_Q is a model of $Rel_{\mathcal{P}}(Q) \cup Rel_{\mathcal{P}}(Infl_{\mathcal{P}}(Q) \cap \mathcal{C})$ according to S such that $\bot \notin M_Q$;
- $M_Q \models Q$ in the sense that $M_Q \supseteq SQ$;
- There is some model M of $\mathcal{P}$ according to S such that $M \supseteq M_Q$

The first condition for M_Q to be a locally consistent answer to Q demands M_Q to be consistent given that M_Q is a model, according to S, of the part of $\mathcal{P}$ relevant for Q, plus all the rules that are relevant for the Integrity Constraints influenced by Q. With this definition, even in the case where all the models of a program are paraconsistent, there still might exist locally consistent sub-models entailing some queries.

In this sense, a locally consistent relevant partial knowledge answer to Q is a sub-model of P — indeed, a model of $Rel_{\mathcal{P}}(Q) \cup Rel_{\mathcal{P}}(Infl_{\mathcal{P}}(Q) \cap \mathcal{C})$, which is a subset of P — that entails the query and still is consistent, i.e., does not include $\bot$. Notice that we do not require the whole program model M, such that $M \supseteq M_Q$, to be consistent.

6.5.5.3 Cumulativity

Cumulativity [84] signifies atoms true in the semantics can be added as facts without thereby changing it; thus, lemmas can be stored. According to [84], Cumulativity equals Cut plus Cautious Monotony. For self-containment we recap their respective definitions:

[4]We assume a paraconsistent semantics is such that it allows its models to contain $\bot$.

Definition 6.34. Cautious Monotony [84]. A semantics is said to enjoy the Cautious Monotony property iff

$$\forall_{a,b \in \mathcal{H}_P}(a \in Sem(P) \wedge b \in Sem(P)) \Rightarrow b \in Sem(P \cup \{a\})$$

Definition 6.35. Cut [84]. A semantics is said to enjoy the Cut property iff

$$\forall_{a,b \in \mathcal{H}_P}(a \in Sem(P) \wedge b \in Sem(P \cup \{a\})) \Rightarrow b \in Sem(P)$$

Notice that $P \cup \{a\}$ denotes the program P to which a is added as a fact.

Formally, cumulativity is defined as follows:

Definition 6.36. Cumulativity [84]. Let P be an NLP, and a, b two atoms of $\mathcal{H}_P$. A semantics Sem is Cumulative iff the semantics of P (i.e., the intersection of models of P) remains unchanged when any atom *true* in the semantics is added to P as a fact:

$$\forall_{a,b \in \mathcal{H}_P} a \in Sem(P) \Rightarrow (b \in Sem(P) \Leftrightarrow b \in Sem(P \cup \{a\}))$$

Similarly to what happens with Relevance (from [85]), this definition had 3-valued models in mind, and using the notation in accordance to Definition 6.29, we can re-write it as

$$\forall_{a,b \in \mathcal{H}_P} a \in Sem^+_{3v}(P) \Rightarrow (b \in Sem^+_{3v}(P) \Leftrightarrow b \in Sem^+_{3v}(P \cup \{a\}))$$

which is also equivalent to

$$\forall_{a,b \in \mathcal{H}_P} \Big((\forall_{M \in Models_{Sem}(P)} a \in M) \Rightarrow (\forall_{M \in Models_{Sem}(P)} b \in M \Leftrightarrow \forall_{M_a \in Models_{Sem}(P \cup \{a\})} b \in M_a) \Big)$$

Cautious Monotony pertains to the possibility of adding as facts atoms true in the semantics without altering the semantics. We now introduce the Brave reasoning counterpart of Cautious Monotony — Brave Cautious Monotony — which expresses the same possibility but for atoms true inside each individual model.

Definition 6.37. Brave Cautious Monotony. Let P be an NLP, and a an atom of $\mathcal{H}_P$. A semantics Sem for logic programs is said to enjoy the Brave Cautious Monotony property iff

$$\forall_{\substack{a \in \mathcal{H}_P \\ M \in Models_{Sem}(P)}} a \in M \Rightarrow M \in Models_{Sem}(P \cup \{a\})$$

Brave Cautious Monotony ensures that any atom found *true* in some model M can be added as a fact to the program, and this addition preserves M as a model of the resulting program. This Brave Cautious Monotony is useful to speed-up computations because as soon as some atom a is found *true* in a partial solution to some query, we can store a in a table of lemmas and use this table to immediately solve other parts of the query that rely on a. Cautious Monotony, on the other hand, only allows this tabling of lemmas to take place for atoms *true* in all models. Brave Cautious Monotony allows it even for individual models. This property is all-important for enabling top-down tabled execution of queries in semantics enjoying Brave Relevance.

6.5.5.4 Stable Models generalization

Definition 6.38. Model conservative generalization semantics. Let P be an NLP, S_1 and S_2 two semantics for NLPs, and $Models_{S_1}(P)$ and $Models_{S_2}(P)$ the sets of models of P according to S_1 and S_2, respectively. We say S_1 is a model conservative generalization of S_2 iff $Models_{S_1}(P) \supseteq Models_{S_2}(P)$, for every P.

The literature on semantics for NLPs usually follows the formalizations of [85] when saying that a semantics S of an NLP P is defined as being the intersection of its models, along the lines of Definition 6.29. When we have two semantics S_1 and S_2, where S_1 is a model conservative generalization of S_2, we necessarily have $S^+_{1_{3v}}(P) \subseteq S^+_{2_{3v}}(P)$. This may lead to misunderstandings on what we mean by the "model conservative generalization" expression, as the reader may think that we mean $S^+_{1_{3v}}(P) \supseteq S^+_{2_{3v}}(P)$, when in fact we mean $Models_{S_1}(P) \supseteq Models_{S_2}(P)$ as per Definition 6.38. Informally, we say a semantics S_1 is a model conservative generalization of another S_2 when it provides at least the same models as the latter. In the particular case where S_2 is not defined for all kinds of NLPs, it may be that S_1 is defined for some of the programs uncovered by S_2. It is in this sense that we say S_1 is more general than S_2.

Every SM complies with the intuitive requirements described in 6.5.4. For this reason, one desirable property of any 2-valued semantics should be to be a model conservative generalization of the Stable Models semantics.

Now that we know which formal properties we seek in a semantics for NLPs, we turn to understand how the structure of an NLP (in accordance to Chapter 3) can, and should, restrict any semantics for NLPs — this is the topic of the next Chapter 7.

---

7 . The Layer-Decomposable Semantics Family

(. . .) at each level of complexity
entirely new properties appear (. . .)

P.W. Anderson

As pointed out in the end of Chapter 2, the modules graph *(Definition 2.10) of a* knowledge graph *and corresponding* least layering *(Definition 2.11), capture both the* structure *and* ordering *of the knowledge. In this chapter we define a family of semantics for Normal Logic Programs — the Layer-Decomposable Semantics (LDS) family — which is compliant with these structuring concepts, and we argue that all semantics for NLPs should be part of this family. The LDS family not only includes the Layer Decomposable Models and the Minimal Hypotheses semantics (defined in Chapter 8), but also the Stable Models semantics.The new 2-valued Minimal Hypotheses semantics has some of the most important and convenient formal properties of the Well-Founded Semantics. With a clear intent to bridge together the Stable Models (and Answer-Set Programming) and Well-Founded Semantics communities, this new semantics offer a new way to handle the truth-values of atoms which would be* undefined *in a 3-valued Well-Founded Semantics.*
The contributions in this chapter are rooted in our publications [193, 194, 196, 197, 198].

7.1 Semantically Reflecting the Layerings

In Chapter 2 we presented the notion of (least) layering, and in Chapter 3 we applied this general notion to NLPs yielding both rule and atom (least) layerings of a NLP; these were shown to exist and to be unique for each NLP. Since in Chapter 2 we showed that the (least) layering notion captures the intrinsic ordering of a knowledge graph, we conclude that every reasonable semantics for NLPs should accept as models only interpretations

that reflect the structural information of the layerings[1]. We dub the family of all such semantics the Layer-Decomposable Semantics (LDS) family.

Intuitively, we say a semantics for NLPs is Layer-Decomposable iff all its models are decomposable into a partition of subsets, each of which is a model for an individual layer, containing all the atoms determined necessarily *true* in that layer, and the default negation of all atoms necessarily *false*, and, what is more, also compliant with all the models for the other layers where "compliance" will be characterized in the sequel. The unique model for layer 0 is the set of default negated literals corresponding to the atoms of P with no rules. The literals in the bodies of rules of a given layer that have no rules for them in the same layer are to be considered as assumable hypotheses by each of the individual layer models. Enforcing inter-layers compliance can be achieved by requiring consistency of the union of individual layers' models.

Before we introduce the formal definition of Layer Decomposable model (Definition 7.3) — where we also formalize which sets of literals are eligible for being *considered assumable hypotheses* within a given layer — we need to formalize the intuitive notion of actually *assuming* those literals as hypotheses (we do this by addition of the positive literals in the set of *assumed hypotheses* as facts to the layer, and then calculating the consequences in the layer, given those new facts).

As we mentioned before, the hypotheses assumption approach we use focuses exclusively on the adoption of positive literals. In this sense, assuming hypotheses corresponds promptly to adding as facts to the program the adopted positive literals. We consider, however, two ways of calculating the consequences of the newly assumed premises: one is by means of the Remainder operator, the other by means of the Layered Remainder one (both previously defined in 6.3.2).

Definition 7.1. Classical Division. Let P be an NLP and let I be a 3-valued interpretation of P. The classical division of P by I, denoted by $P :: I$, is the Remainder (Definition 6.13) reduction of P when we consider I^+ true, i.e., $P :: I = (\widehat{P \cup I^+})$.

Definition 7.2. Layer Division. Let P be an NLP and let I be a 3-valued interpretation of P. The layer division of P by I, denoted by $P : I$, is the Layered Remainder (Definition 6.14) reduction of P when we consider I^+ true, i.e., $P : I = (P \overset{\circ}{\cup} I^+)$. *Note the operator $^\circ$ is applied to the resulting union.*

In both Definitions 7.1 and 7.2, the interpretation I plays the role of the set of assumed hypotheses.

[1]Other works have been done concerning the syntactic structure of a program, with result similar, but not quite equivalent to our own approach, namely [65, 66, 119, 120, 222, 223].

We need the Layer Division because we want to define a family of semantics that is fully compatible with the notion of Layered Support (Definition 6.2), and the Classical Division is clearly not. Notwithstanding, for locally stratified programs the Classical and the Layer Divisions coincide simply because $\overline{body(r)} = \emptyset$ for every rule r in a program. Layer Division is nothing more than a layered support (cf. Section 6.1) compatible version of Classical Division[2]. As a consequence of the definitions of Layer and Classical Divisions, one can be sure that Classical Division deletes more rules than the Layer Division, and the former also simplifies the bodies of rules more than the latter. Recall Example 6.2 that can also be used to illustrate the difference between $\widehat{P \cup \{a\}}$ and $P \overset{\circ}{\cup} \{a\}$ where $P = \{a \leftarrow not\ b\ \ ,\ \ b \leftarrow not\ c\ \ ,\ \ c \leftarrow not\ a\ \ ,\ \ a\}$.

Proposition 7.1. ***Rules of*** $P :: I$ ***are "sub-rules" of*** $P : I$**.** *Let P be an NLP, and I a 3-valued interpretation of P. Then,*

$$\forall_{r \in P::I} \exists_{r' \in P:I} head(r) = head(r') \wedge body(r) \subseteq body(r')$$

Proposition 7.2. ***Models of*** $P :: I$ ***are Models of*** $P : I$**.** *Let P be an NLP, and I a set of literals of P. If some M is a model of $P :: I$ then M is also a model of $P : I$.*

We can now use the syntactic scaffolding of layers, along with the corresponding Layer Division, to define the Layer-Decomposable semantics family. The intuitive idea behind a Layer-Decomposable model M is that it can be decomposed into a set of sub-models $\{M_{\leq 0}, \ldots, M_{\leq \alpha}, \ldots, M_{\leq \omega}\}$, each of which referring to the set of layers $\leq \alpha$ of P, i.e., to $P^{\leq \alpha}$. Each layer-specific sub-model takes as assumed hypotheses truth values for all atoms whose rules appear only in strictly lower layers. Having assumed truth values for such atoms that appear in $P^{<\alpha}$, we need to propagate, in a Layer-support-consistent fashion, those truth values through the rules of P^{α} — we do this *assumption+propagation of consequences* via Layer Division. Finally, we need to select models of the resulting rules that are consistent with assumptions already adopted before (cf. Example 7.1 below).

Definition 7.3. Layer Decomposable Model. Let P be an NLP, and M a 2-valued model of P. M is Layer Decomposable in P — denoted by $LDM_P(M)$ — iff there is a *Layer Decomposition* $LD_P(M) = \{M_{\leq 0}, \ldots, M_{\leq \alpha}, \ldots, M_{\leq \omega}\}$ of M in P, i.e., $M = \bigcup_{\alpha \geq 0} M_{\leq \alpha}$ and

$$\forall_{\alpha \geq 0} M_{\leq \alpha} \text{ is a 3-valued model of } P^{\alpha} : M_{<\alpha} \qquad \text{with} \qquad M^{-}_{\leq \alpha} = not\ (A_P^{\leq \alpha} \setminus M^{+}_{\leq \alpha})$$

where $M_{<0} = M^{+}_{\leq 0} = \emptyset$, and $A_P^{\leq \alpha}$ is the set of atoms whose rules are all placed in layers up to α as per Definition 3.15. We also write M_{α} as a shorthand notation for $M_{\leq \alpha} \setminus M_{<\alpha}$

[2]Since Layer Division and Classical Division are quite similar, it might be possible to implement Layer Division via a syntactic program transformation. I.e., there might be some transformation $LT/1$ such that, for every program P and interpretation I, $P : I \Leftrightarrow LT(P) :: I$. In this thesis we do not explore this path but only mention here its possibility and consider it for future work.

and say M_α is an *individual layer model* of P^α. $M_{<\alpha}$ is the set of hypotheses assumed for layer α and it pertains to atoms of layers strictly below P^α.

Each $M_{\leq\alpha}$ sub-model is a 3-valued model of $P^{\leq\alpha}$, and thus a 3-valued interpretation of P. The positive part of $M_{\leq\alpha}$ states which atoms are believed to be *true* considering only the rules in $P^{\leq\alpha}$. The negative part of $M_{\leq\alpha}$, which is *not* $(A_P^{\leq\alpha} \setminus M^+_{\leq\alpha})$, states that all the atoms that were not determined *true* in $M^{\leq\alpha}$ and that have no more rules in layers above P^α are necessarily determined *false*. It follows directly from Definition 7.2 that $\forall_{\alpha\leq\beta}\ M_{\leq\alpha} \leq_F M_{\leq\beta}$ according to the $\leq_F$ Fitting (knowledge) Ordering notation — Definition 5.11 — where M is a Layer Decomposable model and $M_{\leq\alpha}, M_{\leq\beta} \in LD_P(M)$. As a consequence of this, $(\{M_{\leq\alpha} : \alpha \geq 0\}, \leq_F)$ is a total order with $M_{\leq 0}$ its lower bound and $M = \bigcup_{\alpha\geq 0} M_{\leq\alpha}$ its upper bound.

Along with Corollary 3.2 we introduced the notation $body(r)^{Lf(r)}$ and $\overline{body(r)}$ to make the distinction between the parts of $body(r)$ corresponding to the literals that have rules in the same layer as r, and the literals whose rules, if any, are all placed in layers strictly below that of r, respectively. Let us see the relationship between $\overline{body(r)}$, with $r \in P^\alpha$, and $M_{<\alpha}$. For any given layer P^α, all the literals in all the $\overline{body(r)}$, where $r \in P^\alpha$, are necessarily determined in $M_{<\alpha}$. This is the case because, by definition of $\overline{body(r)}$, *all* the rules for the atoms corresponding to literals in $\overline{body(r)}$ are placed in $P^{<\alpha}$, and because $M_{<\alpha}$ is such that $M^-_{<\alpha} = not\ (A_P^{\leq\alpha} \setminus M^+_{<\alpha})$ we immediately conclude that for every atom $a \in A_P^{\leq\alpha}$ either $a \in M^+_{<\alpha}$ or $not\ a \in M^-_{<\alpha}$ simply because of the definition of $M^-_{<\alpha}$. I.e., $|M_{<\alpha}| \supseteq A_P^{\leq\alpha} \supseteq |\overline{body(r)}|$ for every rule $r \in P^\alpha$.

In Definition 6.2 we relaxed the classical notion of support by introducing the distinction between $body(r)^{Lf(r)}$ and $\overline{body(r)}$ when we introduced the layered support notion which only requires $I \models \overline{body(r)}$ in order for r to be layer supported in I. Thus, we opened the way for an atom to be supported by rules in layers below, even if not classically supported by rules in its own layer. However, this still leaves open the question of how to assign truth values to the literals in $body(r)^{Lf(r)}$. The Layer-Decomposable criterion only demands rule satisfaction — by demanding $M_{\leq\alpha}$ being a model of $P^\alpha : M_{<\alpha}$ — as the requirement for $M_{\leq\alpha}$ to be accepted as LD model of $P^\alpha : M_{<\alpha}$. In Chapter 8 we introduce a more strict criterion for the acceptance of candidate models: that of minimality of positive hypotheses assumed to satisfy the rules of $P^\alpha : M_{<\alpha}$.

Example 7.1. ***The Joint Vacation Problem Revisited.*** *Recall the program in Example 6.5 modeling the vacation related statements of three friends. We now add another level of knowledge representation by making sure that travelling is possible only if the passports of the three friends are OK, i.e., they are not expired; and they are expired if they are not OK. We code this information as the following NLP:*

$$\begin{aligned} beach &\leftarrow not\ mountain \\ mountain &\leftarrow not\ travel \\ travel &\leftarrow not\ beach, not\ expired\_passport \end{aligned}$$

$$\begin{aligned} passport\_ok &\leftarrow not\ expired\_passport \\ expired\_passport &\leftarrow not\ passport\_ok \end{aligned}$$

The rules for $passport\_ok$ *and* $expired\_passport$ *are in layer 1, and the other three rules for* $beach$, $mountain$ *and* $travel$ *are in layer 2. We can constructively identify the Layer Decomposable models of this program the following way:*

1. *Begin at layer 0 with the 3-valued model* $M_0 = not\ A_P^0 = not\ \emptyset = \emptyset$

2. *Make the Layer Division of* P^1 *by* $M_{<1} = M_{\leq 0} = M_0$ *thus obtaining* $P^1 : M_{<1} = P^1 : \emptyset = \mathring{P}^1 = P^1 =$

$$\begin{aligned} passport\_ok &\leftarrow not\ expired\_passport \\ expired\_passport &\leftarrow not\ passport\_ok \end{aligned}$$

3. *Non-deterministically select a 3-valued model* $M_{\leq 1}$ *of* $P^1 : M_{<1}$ *such that* $M^-_{\leq 1} = not\ (A_1^{\leq 1} \setminus M^+_{\leq 1})$ *— there are three such models:* $M_{\leq 1_1} = \{passport\_ok, not\ expired\_passport\}$, $M_{\leq 1_2} = \{not\ passport\_ok, expired\_passport\}$, $M_{\leq 1_3} = \{passport\_ok, expired\_passport\}$. *For the example's illustration purposes let us pick, say,* $M_{\leq 1_1}$

4. *Make the Layer Division of* P^2 *by* $M_{<2_1} = M_{\leq 1_1}$ *thus obtaining* $P^2 : M_{<2_1} = P^2 : \{passport\_ok, not\ expired\_passport\} =$

$$\begin{aligned} beach &\leftarrow not\ mountain \\ mountain &\leftarrow not\ travel \\ travel &\leftarrow not\ beach \\ passport\_ok & \end{aligned}$$

5. *Non-deterministically select a 3-valued model of* $P^2 : M_{<2_1}$ *such that* $M^-_{\leq 2} = not\ (A_2^{\leq 2} \setminus M^+_{\leq 2})$ *— there are three such models:* $M_{\leq 2_{1_1}} = \{beach, travel, not\ mountain, passport\_ok, not\ expired\_passport\}$, $M_{\leq 2_{2_1}} = \{beach, not\ travel, mountain, passport\_ok, not\ expired\_passport\}$, $M_{\leq 2_{3_1}} = \{not\ beach, travel, mountain, passport\_ok, not\ expired\_passport\}$ *and the process terminates because all layers have been covered*

When we pick, say, $M_{\leq 1_2} = \{not\ passport\_ok, expired\_passport\}$ *as the 3-valued model for layer 1, the corresponding Layer Division of* P^2 *by* $M_{<2_2} = M_{\leq 1_2}$ *is*
$P^2 : M_{<2_2} = P^2 : \{not\ passport\_ok, expired\_passport\}$ *which results in the set of facts*

$$mountain$$
$$expired\_passport$$

thus yielding the unique model
$M_{\leq 2_{1_2}} = \{not\ beach, not\ travel, mountain, not\ passport\_ok, expired\_passport\}$.

Not all classical models are Layer Decomposable. For example, the classical model $\{not\ passport\_ok, expired\_passport, travel, not\ mountain, beach\}$ *is not Layer Decomposable because it includes* $M_{\leq 1_2} = \{not\ passport\_ok, expired\_passport\}$ *and when we do a Layer Division of* P^2 *by* $M_{\leq 1_2}$, *mountain becomes a fact and thus necessarily* true *in any Layer Decomposable model.*

Likewise, not all minimal models are Layer Decomposable. For example, a program consisting of only the rule $a \leftarrow not\ b$ has as one of its minimal models $M = \{not\ a, b\}$ which is not Layer Decomposable: the unique rule is placed in Layer 1; a's atom layering is also 1, but since b has no rules its atom layering is 0. Since any LD model must include $not\ A_P^{\leq 0}$, and in this case $A_P^{\leq 0} = \{b\}$, any LD model must include $\{not\ b\}$ which is not the case with $M = \{not\ a, b\}$ above.

Definition 7.4. Layer-Decomposable Semantics. Let P be an NLP. A semantics *Sem* for NLPs is Layer-Decomposable iff every model M of P according to semantics *Sem* is Layer Decomposable.

The first and more obvious Layer-Decomposable semantics we can define is the Layer Decomposable Models semantics which accepts all, and only, Layer Decomposable Models.

Definition 7.5. Layer Decomposable Models semantics. Let P be an NLP. The Layer Decomposable Models semantics (LDMS) accepts as models all, and only, the Layer Decomposable Models of P. I.e., $\forall_M LDMS_P(M) \Leftrightarrow LDM_P(M)$.

As an immediate consequence of this definition 7.3 it follows that

Proposition 7.3. ***The Layer Decomposable Models semantics is a model conservative generalization of every Layer-Decomposable Semantics.***

Proof. Trivial from Definitions 6.38, 7.4, and 7.5. □

One can define many semantics fitting the Layer Decomposability criterion and thus surely we are not here exploring all such possibilities. However there is one other Layer-Decomposable semantics we can easily define:

Definition 7.6. Layer Decomposable Minimal Models semantics. Let P be an NLP. The Layer Decomposable Minimal Models semantics (LDMS) accepts as individual layer models all, and only, the minimal models of the layer after the respective Layer Division. Layer Decomposable Models of P.

Let P be an NLP, and M a 2-valued model of P. M is a Layer Decomposable Minimal Model of P iff there is a *Layer Decomposition* $LD_P(M) = \{M_{\leq 0}, \ldots, M_{\leq \alpha}, \ldots, M_{\leq \omega}\}$ of M in P, i.e., $M = \bigcup_{\alpha \geq 0} M_{\leq \alpha}$ and

$$\forall_{\alpha \geq 0} M_{\leq \alpha} \text{ is a minimal 3-valued model of } P^{\alpha} : M_{<\alpha} \qquad \text{with} \qquad M^{-}_{\leq \alpha} = not\ (A_P^{\leq \alpha} \setminus M^{+}_{\leq \alpha})$$

where $M_{<0} = M^{+}_{\leq 0} = \emptyset$, and $A_P^{\leq \alpha}$ is the set of atoms whose rules are all placed in layers up to α as per Definition 3.15.

We also define Classically Decomposable Model the same way as the Layer Decomposable Model but with Classical Division $P^{\alpha} :: M_{<\alpha}$ instead of Layer Division $P^{\alpha} : M_{<\alpha}$, and, accordingly, the notion of Classically-Decomposable Semantics. It follows immediately that a Classically Decomposable Model is also a Layer Decomposable Model, and that a Classically Decomposable Semantics is also a Layer-Decomposable Semantics. As a consequence, since the Layer-Decomposable Semantics family is a superset of the Classically Decomposable Semantics one, we focus solely on the former without loss of generality.

Moreover, our focus on Layer-Decomposable Semantics stems also from the importance of Layer Division (and, naturally, Layer Decomposability) versus Classical Division (and Classical Decomposability) which is tied to the Cumulativity property (Definition 6.36). A 2-valued semantics for NLP can only enjoy Cumulativity if all its models are compatible with Layer Division.

Example 7.2. ***Layer Division is necessary for Cumulativity.*** *Let P be*

$$\begin{array}{lcl} b & \leftarrow & a \\ a & \leftarrow & not\ b, c \\ c & \leftarrow & not\ a \end{array}$$

which has no stable models. All the rules depend on each other, so they are all in the same layer 1. This program has three classical models: $M_1 = \{a, b, not\ c\}$, $M_2 = \{not\ a, b, c\}$, and $M_3 = \{a, b, c\}$. b is true *in all models. If a semantics enjoys Cumulativity then we*

can add b as a fact to P and the resulting semantics will remain unchanged. $P \cup \{b\}$ is

$$\begin{array}{lcl} b & \leftarrow & a \\ a & \leftarrow & not\ b, c \\ c & \leftarrow & not\ a \\ b & & \end{array}$$

where the fact b is in layer 1 of $P \cup \{b\}$ while the other three original rules are now in layer 2 of $P \cup \{b\}$. The unique model for layers up to 0 is $M_{\leq 0} = \emptyset$, and the unique model for layers up to 1 is $M_{\leq 1} = \{b\}$.

If we take a Classical Division then $P^2 :: M_{\leq 1}$ is just the set of facts $\{b, c\}$. Let us see why: we have $P^2 :: M_{\leq 1} = \widehat{P^2 \cup \{b\}}$ which results in

1. *adding b as a fact to P^2*
2. *deleting the rule $a \leftarrow not\ b, c$ because b is a fact — cf. Definition 6.6 $\mapsto_N$*
3. *deleting the rule $b \leftarrow a$ because a has no rules — cf. Definition 6.9 $\mapsto_F$*
4. *deleting the $not\ a$ from the body of the rule $c \leftarrow not\ a$ because a has no rules — cf. Definition 6.5 $\mapsto_P$*

hence, $P^2 :: M_{\leq 1} = \{b, c\}$ and the unique model of $P^2 :: M_{\leq 1}$ is $M_{\leq 2} = \{not\ a, b, c\}$ — recall that we must explicitly add the negation of all the atoms that were not assigned the truth-value true *and that have no more rules, if any, in layers above, i.e., $M^-_{\leq 2} = not\ (A^{\leq 2}_P \setminus M^+_{\leq 2}) = not\ (\{a, b, c\} \setminus \{b, c\}) = not\ \{a\} = \{not\ a\}$. But now, after adding b as a fact to the program, c becomes also* true *in every (just one) model — the semantics has changed by the addition of an atom that was* true *in the semantics, i.e., the semantics is not Cumulative.*

If instead we take the Layer Division, then $P^2 : M_{\leq 1} = P^2 \mathbin{\mathring{\cup}} \{b\}$ which results in just adding b as a fact to P^2. Let us see why: in this Layer Division case the rule $a \leftarrow not\ b, c$ is not deleted because, although b is a fact, there is also another rule $b \leftarrow a$ that depends on $a \leftarrow not\ b, c$, i.e., $\overline{body(a \leftarrow not\ b, c)} = \emptyset$ and the Layered negative reduction $\mapsto_{LN}$ operation can only use facts b to delete rules if not $b \in \overline{body(r)}$, not if not $b \in body(r)$ as the (non-Layered) Negative reduction does. Since, in this case, the Layer Division does not affect P^2 besides adding the fact b, all the M_1, M_2, and M_3 previously seen remain models of $P \cup \{b\}$ thus keeping the intersection of models — the semantics — unchanged, i.e., the semantics can enjoy Cumulativity.

The Layer Division is a crucial ingredient for Cumulativity exactly because the Layered Negative reduction (the only difference between Layer Division and Classical Division) prevents facts (that are always placed in layer 1) from deleting rules, with the negation of the fact in their bodies, which are in loop with other rules whose head is the same as the fact. Hence, with the Layer Division, atoms that are common to models of loops can be safely added as facts.

7.2 Procedural Methods for Layer Decomposable Models

From Definition 7.3 we can think of different ways to computationally address the calculus of Layer Decomposable Models for a given ground NLP with a finite number of layers.

From Example 7.1 and the Layer Decomposable models definition we can see that, for programs with a finite number of layers, we can define a sound and complete constructive method, which is guaranteed to terminate, for obtaining all the LD models of a program. Such a "*bottom-up*" constructive method might indeed be useful for building models of the whole program.

Definition 7.7. Constructive Method for Layer Decomposable models. Let P be an NLP with a finite number n of layers. Then all the Layer Decomposable models of P can be constructed in the following manner

Algorithm Bottom-Up Construct an LDM($Program\ P$)

$M_{\leq 0} = M_0 = not\ A_P^0$ is the unique *individual layer* model of layer 0, i.e., $P^0 = P^{\leq 0}$;
for each layer index $0 \leq i < n$
    Make the Layer Division of P^{i+1} by $M_{<i+1}$ where $M_{<i+1} = M_{\leq i}$;
    Non-deterministically select a 3-valued model $M_{\leq i+1}$ of $P^{i+1} : M_{<i+1}$ such that $M_{\leq i+1} \supseteq M_{<i+1}$ and $M^-_{\leq i+1} = not\ (A_P^{\leq i+1} \setminus M^+_{\leq i+1})$;
$M_{\leq n}$ is a Layer Decomposable model of P

Figure 7.1: Algorithm Bottom-Up Construct an LDM.

Given that the definition of Layer Decomposable Model relies on the consistent union of *individual layer models* for each layer P^α, we can also contemplate the possibility of distributing the calculus of a LD model by several parallel tasks, one per layer. Each task calculating an individual layer model $M_{\leq \alpha}$ for a given layer P^α would

1. Assume a set $M_{<\alpha}$ of truth values for the all literals in $\overline{body(r)}$ all rules $r \in P^\alpha$;

2. Make the Layer Division of P^α by $M_{<\alpha}$;
3. Non-deterministically calculate a 3-valued model $M_{\leq\alpha}$ of $P^\alpha : M_{<\alpha}$ with $M^-_{\leq\alpha} = not\ (A^{\leq\alpha}_P \setminus M^+_{\leq\alpha})$

The union $M = \bigcup_{\alpha\geq 0} M_{\leq\alpha}$ of the individual layer models $M_{\leq\alpha}$ is a LD model of P iff M is consistent.

However, one of the main motivations for this thesis is not so much the development of constructive methods for Layer-Decomposable Semantics, but the development of a 2-valued semantics for NLPs that allows for top-down query-driven proof-procedures to be developed and used. In this sense, the aim is to have a method for finding submodels, relevant to the query, entailing the truth of the user's query, and known to be extendible to complete models covering the whole program. The Layer Decomposability (induced by the syntactic structure of a program) is the general criterion we adopt for framing 2-valued semantics for NLPs. This is, however, a very general criterion and possibly not all Layer-Decomposable semantics are of interest to us because we are focused on semantics that also enjoy a number of properties, as described in 6.5.5. In Chapter 8 we define and study one such semantics which, we will show, also fits in the Layer-Decomposable Semantics family and fulfils all the requirements we outlined before for a 2-valued semantics for NLPs. For this reason we do not dwell any more upon methods for general Layer-Decomposable semantics.

7.3 Bounding the Layer-Decomposable Semantics Family

Stable Models are often [21, 113, 143, 156] interpreted as sets of beliefs a rational agent can have in order to satisfy a given set of rules. The requirement that every 2-valued semantics should be a generalization of Stable Models, as described in 6.5.4, leads to the intuitive perspective that the models of every 2-valued semantics should, somehow, have this characteristic of embodying tenable sets of beliefs. Thus, together, the Stable Models semantics and the Layer Decomposable Models semantics establish "lower" and "upper" bounds on the set of models a Layer-Decomposable Semantics can accept. Formally, this means that for every NLP P and 2-valued Layer-Decomposable Semantics Sem we have

$$Models_{LDMS}(P) \supseteq Models_{Sem}(P) \supseteq Models_{SM}(P)$$

And hence, by definition 6.29,

$$\begin{array}{lcl} LDMS^+_{3v}(P) & \subseteq & Sem^+_{3v}(P) \subseteq SM^+_{3v}(P) \\ LDMS^u_{3v}(P) & \supseteq & Sem^u_{3v}(P) \supseteq SM^u_{3v}(P) \\ LDMS^-_{3v}(P) & \subseteq & Sem^-_{3v}(P) \subseteq SM^-_{3v}(P) \end{array}$$

The LDS family requires the models for an individual layer to be consistent with the other layers' individual models. Moreover, a model for an individual layer assumes as hypotheses certain truth values for all the literals having their rules in layers strictly below the layer at hand, and possibly also literals with rules in the current layer. The overall consistency requirement of the union of individual layers models ensures that all the hypothesized truth values are compatible with the truth values assigned to the same literals in models of other individual layers. Still, there might be some Layer-Decomposable semantics that do not accept all Layer Decomposable Models, e.g., because they impose stricter conditions on individual layer models.

The belief set self-corroboration of Stable Models can be seen as a notion of support. In the case of the SMs, this is the classical one (Definition 6.1) which requires *all* the literals in the body of some rule for an atom a to be *true* by default under interpretation I in order for a to be classically supported in I. This classical notion of support makes no distinction between literals whose atoms have rules in the same layer, and literals whose atoms have all their rules (if any) in layers strictly lower than that of the rule at hand.

In Definition 6.2 we relaxed the classical notion of support by introducing the layered support notion which only requires $I \models \overline{body(r)}$ in order for r to be layer supported in I. In so doing, we opened the way for an atom to be supported by rules in layers below, even if not classically supported by rules in its own layer. However, this still leaves open the question of which notion of support should be required on $body(r)^{Lf(r)}$. The next Chapter 8 indirectly addresses this issue by taking a novel and general approach to semantics of NLPs.

---

8 . Minimal Hypotheses semantics

> The grand aim of all science is to cover the greatest number of empirical facts by logical deduction from the smallest number of hypotheses or axioms.
>
> Albert Einstein

In Chapter 3 we identified the syntactic structure of a program which is captured by the (rule and atom) Layering notions. In Chapter 6 we took an overview of the current state-of-the-art semantics, the constructs they use, and some "good" properties a semantics should enjoy to allow efficient (possibly abductive) existential query-answering replied to with enough partial knowledge only. In Chapter 7 we outlined the necessary semantic skeleton drawn out of the syntactic scaffolding of the Layerings, which lead to the definition of the Layer-Decomposable family of semantics.
In this chapter we take a different approach at semantics but still following the guidelines of 6.5.4. Fully taking the requirements outlined therein, we devised the Minimal Hypotheses (MH) semantics, which we present in this chapter, it being also a Layer-Decomposable semantics. MH semantics takes a hypotheses assumption approach as a means to provide support to the body$(r)^{Lf(r)}$ part of a rule (cf. last paragraph of Chapter 7).
We begin by introducing the fundamental semantic concept lying at the core of the MHS, that of minimality of assumed hypotheses, then go about defining the MH semantics based on that concept, and proceed to analyse the semantics' properties.

8.1 Minimality of Hypotheses

The abductive perspective of [129] depicts the atoms of default negated literals (DNLs) as abducibles, i.e., assumable hypotheses. We explore this relation to abductive reasoning in further detail in Chapter 10, but for now let us simply consider this hypotheses-assumption

perspective informally.

Atoms of DNLs can be considered as abducibles, i.e., assumable hypotheses, but not all of them. When we have a locally stratified program (viz. 6.3.2) we cannot really say there is any degree of freedom in assuming truth values for the atoms of the program's DNLs. In this sense, we realize that only the atoms of DNLs involved in non-well-founded negation[1] are eligible to be considered further assumable hypotheses.

Both the Stable Models and the approach of [129], when taking the abductive perspective, adopt negative hypotheses only. E.g., in a program like $a \leftarrow not\ b \quad b \leftarrow not\ a$, if we assume the hypothesis *not b* we conclude a is *true* which undermines b's single rule, thereby corroborating the initial *not b* assumption. This approach works fine for some instances of non-well-founded negation such as loops (in particular, for even loops over negation like this one), but not for odd loops over negation like, e.g. $a \leftarrow not\ a$: assuming *not a* would lead to the conclusion that a is *true* which contradicts the initial assumption. To overcome this problem, we generalized the hypotheses assumption perspective to allow the adoption, not only of negative hypotheses, but also of positive ones. Having taken this generalization step we realized that positive hypotheses assumption alone is sufficient to address all situations, i.e., there is no need for both positive and negative hypotheses assumption. Indeed, because we minimize the positive hypotheses we are in one stroke maximizing the negative ones, which has been the traditional way of dealing with the CWA, and also with stable model because the latter's requirement of classical support minimizes models. This is the reason why we decided to embrace the only positive hypotheses assumption perspective in this thesis.

In subsection 6.3.2 we recalled the Remainder ($\widehat{P}$) and Layered Remainder ($\mathring{P}$) operators, each consisting of a series of straightforward deterministic linear syntactic transformations (except for the *loop detection* $\mapsto_L$ and the *layered negative reduction* $\mapsto_{LN}$ which are polynomial). Both the Remainder and the Layered Remainder can be used to simplify a program down to the subset of its original rules (some of which may have their bodies likewise reduced) which then becomes (classically for $\widehat{P}$, and layer-wise for $\mathring{P}$) independent from those literals having determined truth-values. Since we are striving for a hypotheses-assumption based approach to a Layer-Decomposable semantics, we must select $\mathring{P}$ instead of $\widehat{P}$ as the means to simplify away the layer-wise deterministic part of P in order to leave in $\mathring{P}$ the layered support compatible assumable hypotheses. Thus, all the literals of P that are not determined *false* in $\mathring{P}$ are candidates for the role of hypotheses we may consider to assume as *true*. Merging this perspective with the abductive perspective

[1]Recall that by non-well-founded negation we mean either Strongly Connected Components of rules with at least one head of a rule appearing as a DNL in some body of a rule of the SCC (cf., e.g., Examples 6.5,7.1,7.2); or an infinitely long descending chain of rules with negative dependencies amongst them (cf. Example 3.5).

of [129] (where the DNLs are the abducibles) we come to the following definition of the Hypotheses set of a program.

Definition 8.1. Hypotheses set of a program. Let P be an NLP. We write $Hyps(P)$ to denote the set of assumable hypotheses of P: the atoms that appear as default negated literals in the bodies of rules of P and which are not determined *false* in $\mathring{P}$. Formally,

$$Hyps(P) = heads(\mathring{P}) \cap \{a : \exists_{r \in P} not\ a \in body(r)\}$$

or equivalently

$$Hyps(P) = \{a : \exists_{r \in \mathring{P}} not\ a \in body(r)\}$$

One can define a classical support compatible version of the Hypotheses set of a program, only needing to that effect to use the Remainder instead of the Layered Remainder. I.e.,

Definition 8.2. Classical Hypotheses set of a program. Let P be an NLP. We write $CHyps(P)$ to denote the set of assumable hypotheses of P consistent with the classical notion of support: the atoms that appear as default negated literals in the bodies of rules of P and which are not determined *false* in $\widehat{P}$. Formally,

$$CHyps(P) = heads(\widehat{P}) \cap \{a : \exists_{r \in P} not\ a \in body(r)\}$$

or equivalently

$$CHyps(P) = \{a : \exists_{r \in \widehat{P}} not\ a \in body(r)\}$$

In this thesis we are taking the layered support compatible approach and, therefore, we will use the Hypotheses set as in Definition 8.1. However, since $CHyps(P) \subseteq Hyps(P)$ for every NLP P, there is no generality loss in using $Hyps(P)$ instead of $CHyps(P)$, while at the same time using $Hyps(P)$ allows for some useful semantics properties as we shall examine in the sequel.

A 2-valued model of a program, by definition, assigns a 2-valued truth-value for every literal in the program. Assuming as *true* the facts of $\mathring{P}$, and as *false* all the atoms of P with no rules in $\mathring{P}$, is the first and most basic requirement any Layer-Decomposable 2-valued model must comply with. But a full 2-valued model must assign a truth-value to all, if any, other atoms — i.e., the Hypotheses and the atoms that depend on them. Once having simplified away the layer-deterministic part of P — obtaining $\mathring{P}$ — we must now start making assumptions about the Hypotheses of the program, bearing in mind that each such hypothesis assumed *true* may immediately constraint the possible truth-values of other candidate hypotheses (and other literals depending on the hypothesis) via the consequences it entails, in order to find a 2-valued model. We resort to the Remainder

$\widehat{P}$ as a means to fully propagate the truth-values of the assumed hypotheses and thus calculate their consequences.

Intuitively, a Minimal Hypotheses model of a program is derived from a set of such hypotheses which is sufficiently large to determine the truth-value of all literals via Remainder, and simultaneously set-inclusion minimal, i.e., the hypotheses it assumes are no more than those necessary to determine the truth-values of all literals, in the set inclusion sense rather than in the cardinality sense.

> Je n'avais pas besoin de cette hypothése-lá.
>
> Pierre-Simon Laplace

Definition 8.3. Minimal Hypotheses model. Let P be an NLP. Let $Hyps(P)$ be the set of assumable hypotheses of P (cf. Definition 8.1), and H some subset of $Hyps(P)$.

A 2-valued model M of P is a Minimal Hypotheses model of P iff

$$M^+ = facts(\widehat{P \cup H}) = heads(\widehat{P \cup H})$$

where $H = \emptyset$ or H is non-empty set-inclusion minimal (the set-inclusion minimality is considered only for non-empty Hs). I.e., the hypotheses set H is minimal but sufficient to determine (via Remainder) the truth-value of all literals in the program.

Notice we do *not* resort to an incremental way of constructing a set of hypotheses by, e.g., iteratively adding hypotheses as facts and computing the Remainder in order to check if more hypotheses are needed or not — nor do we recompute new Layerings as hypotheses would be added. Such a process cannot guarantee set-inclusion minimality of the assumed hypotheses. Instead, finding a Minimal Hypotheses model can be done by selecting some *set* of hypotheses, checking via Remainder that they are enough to propagate 2-valuedness to all literals in the program, and then checking the initially assumed set of hypotheses is minimal w.r.t guaranteeing 2-valued completeness.

By Definition 6.16 we know that $WFM^+(P) = facts(\widehat{P})$ and that $WFM^{+u}(P) = heads(\widehat{P})$. Thus, whenever $facts(\widehat{P}) = heads(\widehat{P})$ we have $WFM^+(P) = WFM^{+u}(P)$ which means $WFM^u(P) = \emptyset$. Moreover, whenever $WFM^u(P) = \emptyset$ we know, by Corollary 5.6 of [109], that the 2-valued model M such that $M^+ = facts(\widehat{P})$ is the unique stable model of P. Thus, we conclude that, as an alternative equivalent definition, M is a Minimal Hypotheses model of P iff M is a stable model of $P \cup H$ where H is either empty or a non-empty set-inclusion minimal subset of $Hyps(P)$. This provides an alternative

way to understand Minimal Hypotheses models: as Stable Models of a program that has been augmented with new facts — the hypotheses. The crucial point remains in identifying the set of possible hypotheses $Hyps(P)$ which, as seen in Definition 8.1, must be compatible with the Layered Remainder. Were it not for this central new different contribution — the Layered Remainder (the difference coming from the new Layered Negative Reduction) — all MH models would coalesce to the SMs. It is the use of the Layered Remainder that allows for more hypotheses, thus allowing for more MH models, some of them not being SMs.

The $H = \emptyset$ case is necessary to cover the cases where the program is locally stratified. For these cases it is known the Well-Founded Model is 2-valued complete, i.e., $WFM(P)^u = \emptyset$ which means that $facts(\widehat{P}) = heads(\widehat{P})$, or equivalently $facts(\widehat{P \cup \emptyset}) = heads(\widehat{P \cup \emptyset})$. In such cases we want to accept $M_\emptyset = WFM^+(P) \cup not\ WFM^-(P)$ as a MH model of P simply because $M^+_\emptyset = facts(\widehat{P \cup \emptyset}) = WFM^+(P)$.

Example 8.1. ***Minimal Hypotheses models for the vacation problem.*** *Recall again the program from the vacation Example 6.5 (not the one from Example 7.1 with the passport rules).* $P =$

$$\begin{array}{rcl} beach & \leftarrow & not\ mountain \\ mountain & \leftarrow & not\ travel \\ travel & \leftarrow & not\ beach \end{array}$$

All the rules depend on each other, and so they are all in layer 1. What is more $\overline{body(r)} = \emptyset$ *for each and every rule in this program — all the literals in the bodies of these rules are in loop with the rule whose body they are part of. In this case we thus have* $P = \widehat{P} = \mathring{P}$, *and also* $Hyps(P) = \{beach, mountain, travel\}$. *Let us see what are the MH models of this program.*

There is no point in trying out $H = \emptyset$ *to check if* $M_\emptyset$ *is a MH model of* P *because we already know that* $facts(\widehat{P \cup \emptyset}) = facts(\widehat{P}) \neq heads(\widehat{P}) = heads(\widehat{P \cup \emptyset})$.

Assuming $H = \{beach\}$ *we have* $P \cup H = P \cup \{beach\} =$

$$\begin{array}{rcl} beach & \leftarrow & not\ mountain \\ mountain & \leftarrow & not\ travel \\ travel & \leftarrow & not\ beach \\ beach & & \end{array}$$

and thus $\widehat{P \cup H} = P \cup \widehat{\{beach\}}$ *which is just the set of facts* $\{beach, mountain\}$. *This means that* $M^+_{beach} = facts(P \cup \widehat{\{beach\}}) = heads(P \cup \widehat{\{beach\}}) = \{beach, mountain\}$ *and thus* $M_{beach} = \{beach, mountain, not\ travel\}$ *is a Minimal Hypotheses model of* P.

Likewise, assuming $H = \{mountain\}$ *we have* $M^+_{mountain} = facts(P \cup \{\widehat{mountain}\}) = heads(P \cup \{\widehat{mountain}\}) = \{mountain, travel\}$ *rendering* $M_{mountain} = \{not\ beach, mountain, travel\}$ *also a MH model of* P. *And finally, assuming* $H = \{travel\}$ *we have* $M^+_{travel} = facts(P \cup \widehat{\{travel\}}) = heads(P \cup \widehat{\{travel\}}) = \{beach, travel\}$ *rendering* $M_{travel} = \{beach, not\ mountain, travel\}$ *also a MH model of* P. *There are no other MH models of* P *as any other non-empty subset of* H *is not minimal w.r.t. to at least one of the* $H = \{beach\}$, $H = \{mountain\}$ *and* $H = \{travel\}$ *considered for the three models we have just seen.*

Consider now the same example, but this time with an additional fourth stubborn friend that insists on going to the beach for the vacation, no matter what the other friends decide. The program is now[2] $P_{stubborn} =$

$$\begin{array}{rcl} beach & \leftarrow & not\ mountain \\ mountain & \leftarrow & not\ travel \\ travel & \leftarrow & not\ beach \\ beach & & \end{array}$$

Again we have $\overset{\circ}{P}_{stubborn} = P_{stubborn}$ *and thus the set of hypotheses of* $P_{stubborn}$ *remains* $\{beach, mountain, travel\}$. *Notice that the use of* $\overset{\circ}{P}_{stubborn}$ *instead of* $\widehat{P_{stubborn}}$ *allows* $Hyps(P_{stubborn})$ *to be the same as for* P *(without the stubborn friend) above. The MH models in this case are:*
assuming $H = \emptyset$, *we have* $P_{stubborn} \cup H = P_{stubborn} \cup \emptyset = P_{stubborn}$ *and* $M^+_{stubborn_\emptyset} = facts(\widehat{P_{stubborn}} \cup \emptyset) = heads(\widehat{P_{stubborn}} \cup \emptyset) = \{beach, mountain\}$, *thus* $M_{stuborn_\emptyset} = \{beach, mountain, not\ travel\}$ *is a Minimal Hypotheses model of* $P_{stubborn}$;
assuming $H = \{beach\}$, *we have* $P_{stubborn} \cup H = P_{stubborn} \cup \{beach\} = P_{stubborn}$ *because beach was already a fact in* $P_{stubborn}$

$$\begin{array}{rcl} beach & \leftarrow & not\ mountain \\ mountain & \leftarrow & not\ travel \\ travel & \leftarrow & not\ beach \\ beach & & \end{array}$$

and $M^+_{stubborn_{beach}} = facts(\widehat{P_{stubborn} \cup \{beach\}}) = heads(\widehat{P_{stubborn} \cup \{beach\}}) = \{beach, mountain\}$, *thus* $M_{stuborn_{beach}} = \{beach, mountain, not\ travel\}$ *is a Minimal Hypotheses model of* $P_{stubborn}$, *exactly the same as* $M_{stubborn_\emptyset}$ *above;*

[2]This variation of the vacation example is but a renaming of the literals in Example 6.2.

assuming $H = \{mountain\}$, *we have* $P_{stubborn} \cup H = P_{stubborn} \cup \{mountain\} =$

$$\begin{array}{rcl} beach & \leftarrow & not\ mountain \\ mountain & \leftarrow & not\ travel \\ travel & \leftarrow & not\ beach \\ beach & & \\ mountain & & \end{array}$$

and $M^+_{stubborn_{mountain}} = facts(P_{stubborn}\widehat{\cup\{mountain\}}) = heads(P_{stubborn}\widehat{\cup\{mountain\}}) = \{beach, mountain\}$, *thus* $M_{stubborn_{mountain}} = \{beach, mountain, not\ travel\}$ *which is the same MH model of* $P_{stubborn}$ *as both* $M_{stubborn_\emptyset}$ *and* $M_{stubborn_{beach}}$ *above;*
assuming $H = \{travel\}$, *we have* $P_{stubborn} \cup H = P_{stubborn} \cup \{travel\} =$

$$\begin{array}{rcl} beach & \leftarrow & not\ mountain \\ mountain & \leftarrow & not\ travel \\ travel & \leftarrow & not\ beach \\ beach & & \\ travel & & \end{array}$$

and $M^+_{stubborn_{travel}} = facts(P_{stubborn}\widehat{\cup\{travel\}}) = heads(P_{stubborn}\widehat{\cup\{travel\}}) = \{beach, travel\}$, *thus* $M_{stubborn_{travel}} = \{beach, not\ mountain, travel\}$ *is also an MH model of* $P_{stubborn}$, *and there are no other.*

We can see that, when we add the fourth stubborn friend insisting on going to the beach, we no longer have the $M_{mountain} = \{not\ beach, mountain, travel\}$ *model we had in the original example. Recall that in the original program (without the beach fact) there were three models* $\{beach, mountain, not\ travel\}$, $\{beach, not\ mountain, travel\}$, *and* $\{not\ beach, mountain, travel\}$. *Of these, only the third model* $\{not\ beach, mountain, travel\}$ *is inconsistent with adding the fact beach, i.e., adding beach as a fact (insisting on going to the beach) should not invalidate the* $\{beach, not\ mountain, travel\}$ *model as this was already a possibility considered by the three initial friends.*

The minimality of H is not sufficient to ensure minimality of $M^+ = facts(\widehat{P \cup H})$ making its checking explicitly necessary if that is so desired. Minimality of hypotheses is indeed the common practice in science, not the minimality of their inevitable consequences. To the contrary, the more of these the better because it signifies a greater predictive power.

In Logic Programming whole model minimality is a consequence of resorting to least fixed point definitions: the T operator in definite programs is conducive to defining a least fixed point, a unique minimal model semantics; in SM, though there may be more

than one model, minimality turns out to be a property because of the stability (and its attendant classical support) requirement; in the WFS, again the existence of a least fixed point operator affords a minimal (information) model. In abduction too, minimality of consequences is not a caveat, but rather minimality of hypotheses is, if that even. Hence our approach to LP semantics via MHS is novel indeed, and by insisting instead on positive hypotheses establishes an improved and more general link to abduction and to argumentation [188, 189].

Example 8.2. ***Minimality of Hypotheses does not guarantee minimality of model.*** *Let P, affording a single layer and with no SMs, be*

$$\begin{array}{rcl} a & \leftarrow & not\ b, c \\ b & \leftarrow & not\ c, not\ a \\ c & \leftarrow & not\ a, b \end{array}$$

In this case $P = \widehat{P} = \mathring{P}$, which makes $Hyps(P) = \{a, b, c\}$.

$H = \emptyset$ does not determine all literals of P because

$$\begin{array}{rcl} facts(\widehat{P \cup \emptyset}) & = & facts(\widehat{P}) = \emptyset \text{ and} \\ heads(\widehat{P \cup \emptyset}) & = & heads(\widehat{P}) = \{a, b, c\} \end{array}$$

$H = \{a\}$ does determine all literals of P because

$$\begin{array}{rcl} facts(\widehat{P \cup \{a\}}) & = & \{a\} \text{ and} \\ heads(\widehat{P \cup \{a\}}) & = & \{a\} \end{array}$$

thus yielding the MH model M_a such that $M_a^+ = facts(\widehat{P \cup \{a\}}) = \{a\}$, i.e., $M_a = \{a, not\ b, not\ c\}$.

$H = \{c\}$ is also a minimal set of hypotheses determining all literals

$$\begin{array}{rcl} facts(\widehat{P \cup \{c\}}) & = & \{a, c\} \text{ and} \\ heads(\widehat{P \cup \{c\}}) & = & \{a, c\} \end{array}$$

thus yielding the MH model M_c of P such that $M_c^+ = facts(\widehat{P \cup \{c\}}) = \{a, c\}$, i.e., $M_c = \{a, not\ b, c\}$. However, M_c is not a minimal model of P because $M_c^+ = \{a, c\}$ is a strict superset of $M_a^+ = \{a\}$. M_c is indeed an MH model of P, but just not a minimal one thereby being a clear example of how minimality of hypotheses does not entail minimality of consequences.

Just to make this example complete, we show that $H = \{b\}$ *also determines all literals of* P *because*

$$\begin{array}{rcl} facts(\widehat{P \cup \{b\}}) & = & \{b, c\} \text{ and} \\ heads(\widehat{P \cup \{b\}}) & = & \{b, c\} \end{array}$$

thus yielding the MH model M_b *such that* $M_b^+ = facts(\widehat{P \cup \{b\}}) = \{b, c\}$*, i.e.,* $M_b = \{not\ a, b, c\}$*. Any other hypotheses set is necessarily a strict superset of either* $H = \{a\}$*,* $H = \{b\}$*, or* $H = \{c\}$ *and, therefore, not set-inclusion minimal.*

Minimal Hypotheses models are Layer Decomposable models (as we show below), but not all LD models are MH models, as the following example shows.

Example 8.3. ***Some Layer Decomposable models are not Minimal Hypotheses models.*** *Let single layer with no SMs* P *be*

$$\begin{array}{rcl} a & \leftarrow & k \\ k & \leftarrow & not\ t \\ t & \leftarrow & a, b \\ a & \leftarrow & not\ b \\ b & \leftarrow & not\ a \end{array}$$

In this case $P = \widehat{P} = \mathring{P}$ *and therefore* $Hyps(P) = \{a, b, t\}$*. Since* $facts(\widehat{P}) \neq heads(\widehat{P})$*, the hypotheses set* $H = \emptyset$ *does not yield a MH model.*

Assuming $H = \{a\}$ *we have* $\widehat{P \cup H} = \widehat{P \cup \{a\}} =$

$$\begin{array}{l} a \\ k \end{array}$$

so, $\widehat{P \cup H}$ *is the set of facts* $\{a, k\}$ *and, therefore,* M_a *such that* $M_a^+ = facts(\widehat{P \cup H}) = facts(\widehat{P \cup \{a\}}) = \{a, k\}$*, is a MH model of* P*.*

Assuming $H = \{b\}$ *we have* $\widehat{P \cup \{b\}} =$

$$\begin{array}{rcl} a & \leftarrow & k \\ k & \leftarrow & not\ t \\ t & \leftarrow & a \\ b & \leftarrow & not\ a \\ b & & \end{array}$$

thus $facts(\widehat{P \cup \{b\}}) = \{b\} \neq heads(\widehat{P \cup \{b\}}) = \{a, b, t, k\}$*, which means the set of hypotheses* $H = \{b\}$ *does not yield a MH model of* P*.*

Assuming $H = \{t\}$ *we have* $\widehat{P \cup \{t\}} =$

$$\begin{array}{lcl} t & \leftarrow & a, b \\ b & \leftarrow & not\ a \\ a & \leftarrow & not\ b \\ t & & \end{array}$$

thus $facts(\widehat{P \cup \{t\}}) = \{t\} \neq heads(\widehat{P \cup \{t\}}) = \{a, b, t\}$*, which means the set of hypotheses* $H = \{t\}$ *does not yield a MH model of* P*.*

Since we already know that $H = \{a\}$ *yields an MH model* M_a *with* $M_a^+ = \{a, k\}$*, there is no point in trying out any subset* H' *of* $Hyps(P) = \{a, b, t\}$ *such that* $a \in H'$ *because any such subset would not be minimal w.r.t.* $H = \{a\}$*. Let us, therefore, move on to the unique subset left:* $H = \{b, t\}$*. Assuming* $H = \{b, t\}$ *we have* $\widehat{P \cup \{b, t\}} =$

$$\begin{array}{l} t \\ b \end{array}$$

thus $facts(\widehat{P \cup \{b, t\}}) = \{b, t\} = heads(\widehat{P \cup \{b, t\}})$*, which means* $M_{b,t}$ *such that* $M_{b,t}^+ = facts(\widehat{P \cup H}) = facts(\widehat{P \cup \{b, t\}}) = \{b, t\}$*, is a MH model of* P*.*

It is important to remark that this program has other classical models, e.g, $\{a, k\}, \{b, t\}$*, and* $\{a, t\}$*, all of which are Layer Decomposable Models of* P*, but only the first two are Minimal Hypotheses models —* $\{a, t\}$ *is obtainable only via the set of hypotheses* $\{a, t\}$ *which is non-minimal w.r.t.* $H = \{a\}$ *that yields the MH model* $\{a, k\}$*.*

8.2 Properties of the Minimal Hypotheses Semantics

As explained in 6.5.5.4, every SM complies with the intuitive requirements described in 6.5.4 and hence the MH semantics (and any other 2-valued semantics for NLPs for that matter) should be model conservative generalization of the Stable Models semantics.

Theorem 8.1. ***Every Stable Model is a Minimal Hypotheses model.*** *Let* P *be an NLP, and* M *a stable model of* P*. Then,* M *is also a Minimal Hypotheses model of* P*. The Minimal Hypotheses semantics is thus a model conservative generalization of the Stable Models semantics — cf. Definition 6.38.*

It is also convenient to understand how MH models relate to the Well-Founded Model (and to the Layered Well-Founded Model) of a program.

In the variation with the stubborn friend of Example 8.1 we saw that there are two distinct MH models: $\{beach, mountain, not\ travel\}$ and $\{beach, not\ mountain, travel\}$. We can also see easily that the Well-Founded Model of that variation with the stubborn friend is such that $WFM(P)^+ = \{beach, mountain\}$, $WFM(P)^u = \emptyset$, and $WFM(P)^- = \{travel\}$. Knowing this one can wonder why, then, is there a MH model like $\{beach,$ $not\ mountain,\ travel\}$ which clearly contradicts the WFM, both in taking *mountain* as *false* when it is *true* in the WFM, and by taking *travel* as *true* when it is *false* in the WFM. To answer this question we must recall that, according to Definition 6.29, the 3-valued model of a 2-valued semantics for a given program results from the intersection of all the 2-valued models of the semantics for that program. In this sense, the 3-valued model of Example 8.1 according to MH semantics is $MH_{3v}(P_{stubborn})^+ = \{beach\}$, $MH_{3v}(P_{stubborn})^u = \{mountain, travel\}$, $MH_{3v}(P_{stubborn})^- = \emptyset$ which complies (and even coincides), not with the WFM, but with the Layered WFM (Definition 6.17). Since the MH semantics is a Layer-Decomposable semantics, it only makes sense that its 3-valued model complies with the LWFM but not necessarily with the WFM, the latter resting upon the classical support notion, instead of the layered support of the former. The fundamental reason for there existing the MH model $\{beach, not\ mountain, travel\}$ which contradicts the WFM is that we want the MH semantics to enjoy Cumulativity, amongst other properties, and, as seen on Example 7.2, a 2-valued semantics can only enjoy Cumulativity if it is compatible with Layer Division. In that Example 7.2 we can clearly see that, if the MH definition would allow the set-inclusion minimality required of H to range over possible empty H then, in that case, there would be a single MH model for a program which would coincide with its 2-valued Well-Founded Model, and this, as illustrated in Example 7.2, would undermine Cumulativity.

In this current particular vacation example we can see an instance of the Theorem 6.2 where
$MH_{3v}(P_{stubborn})^+ = LWFM(P)^+ = \{beach\} \subseteq WFM(P)^+ = \{beach, mountain\}$,
$MH_{3v}(P_{stubborn})^u = LWFM(P)^u = \{mountain, travel\} \supseteq WFM(P)^u = \emptyset$, and
$MH_{3v}(P_{stubborn})^- = LWFM(P)^- = \emptyset \subseteq WFM(P)^- = \{travel\}$.

Nonetheless, $P_{stubborn}$ has a MH model $\{beach, mountain, not\ travel\}$ that completely complies with and corroborates the WFM. And this is also true in general, i.e., for every program P there is a MH model M such that $M^+ \supseteq WFM^+(P)$ and $M^- \supseteq WFM^-(P)$.

Theorem 8.2. ***At least one Minimal Hypotheses model of*** P ***complies with the Well-Founded Model.*** *Let P be an NLP. Then, there is at least one Minimal Hypotheses model M of P such that $M^+ \supseteq WFM^+(P)$ and $M^- \supseteq not\ WFM^-(P)$.*

Theorem 8.3. ***All Minimal Hypotheses models of*** P ***comply with the Layered Well-Founded Model.*** *Let P be an NLP, and M a Minimal Hypotheses model M of P. Then, M is such that $M^+ \supseteq LWFM^+(P)$ and $M^- \supseteq not\ LWFM^-(P)$.*

The minimality requirement imposed on H for MH models is sufficient to ensure Layer-Decomposablity of MH models.

Theorem 8.4. ***Minimal Hypotheses models are Layer Decomposable models.*** *Let P be an NLP, and M a Minimal Hypotheses model of P. Then, M is also a Layer Decomposable model of P.*

Example 8.4. ***Minimal Hypotheses models for the vacation with passport variation.*** *Consider again the vacation problem with the variation from Example 7.1*
$P =$

$$\begin{aligned} beach &\leftarrow not\ mountain \\ mountain &\leftarrow not\ travel \\ travel &\leftarrow not\ beach, not\ expired\_passport \\ \\ passport\_ok &\leftarrow not\ expired\_passport \\ expired\_passport &\leftarrow not\ passport\_ok \end{aligned}$$

We have $P = \mathring{P} = \widehat{P}$ and thus $Hyps(P) = \{beach, mountain, travel, passport\_ok, expired\_passport\}$. Let us see which are the MH models for this program and how they relate to the Layer Decomposable models in Example 7.1.

$H = \emptyset$ does not yield a MH model.
Assuming $H = \{beach\}$ we have $P \cup H = P \cup \{beach\} =$

$$\begin{aligned} beach &\leftarrow not\ mountain \\ mountain &\leftarrow not\ travel \\ travel &\leftarrow not\ beach, not\ expired\_passport \\ beach & \\ passport\_ok &\leftarrow not\ expired\_passport \\ expired\_passport &\leftarrow not\ passport\_ok \end{aligned}$$

and $\widehat{P \cup H} =$

$$\begin{aligned} mountain & \\ beach & \\ passport\_ok &\leftarrow not\ expired\_passport \\ expired\_passport &\leftarrow not\ passport\_ok \end{aligned}$$

which means $H = \{beach\}$ is not sufficient to determine the truth values of all literals of P. One can easily see that the same happens for $H = \{mountain\}$ and for $H = \{travel\}$: in either case the literals $passport\_ok$ and $expired\_passport$ remain non-determined.

If we assume $H = \{expired\_passport\}$ *then* $P \cup H$ *is*

$$\begin{array}{rcl} beach & \leftarrow & not\ mountain \\ mountain & \leftarrow & not\ travel \\ travel & \leftarrow & not\ beach, not\ expired\_passport \\ \\ passport\_ok & \leftarrow & not\ expired\_passport \\ expired\_passport & \leftarrow & not\ passport\_ok \\ expired\_passport & & \end{array}$$

and $\widehat{P \cup H} =$

$$\begin{array}{r} mountain \\ expired\_passport \end{array}$$

which means $M^{+}_{expired\_passport} = facts(\widehat{P \cup H}) = heads(\widehat{P \cup H}) = \{mountain, expired\_passport\}$, *i.e.,*
$M_{expired\_passport} = \{not\ beach, mountain, not\ travel, not\ passport\_ok, expired\_passport\}$, *is a MH model of* P *— this corresponds to* $M_{\leq 2_{1_2}}$ *of Example 7.1. Since assuming* $H = \{expired\_passport\}$ *alone is sufficient to determine all literals, there is no other set of hypotheses* H' *of* P *such that* $H' \supset \{expired\_passport\}$ *(notice the* strict $\supset$*, not* $\supseteq$*), yielding a MH model of* P*. E.g.,* $H' = \{travel, expired\_passport\}$ *does not lead to a MH model of* P *simply because* H' *is not minimal w.r.t.* $H = \{expired\_passport\}$.

If we assume $H = \{passport\_ok\}$ *then* $P \cup H$ *is*

$$\begin{array}{rcl} beach & \leftarrow & not\ mountain \\ mountain & \leftarrow & not\ travel \\ travel & \leftarrow & not\ beach, not\ expired\_passport \\ \\ passport\_ok & \leftarrow & not\ expired\_passport \\ expired\_passport & \leftarrow & not\ passport\_ok \\ passport\_ok & & \end{array}$$

and $\widehat{P \cup H} =$

$$\begin{array}{rcl} beach & \leftarrow & not\ mountain \\ mountain & \leftarrow & not\ travel \\ travel & \leftarrow & not\ beach \end{array}$$

which corresponds to the original version of this example and still leaves literals with non-determined truth-values. I.e., assuming the passports are OK allows for the three possibilities of Example 6.5 but it is not enough to entirely "solve" the vacation problem: we need some hypotheses set containing one of beach, mountain, *or* travel *if (in this case,* and only if*) it also contains* passport_ok.

Corollary 8.1. ***Stable Models are Layer Decomposable models.*** *Let P be an NLP, and M a stable model of P. Then, M is also a Layer Decomposable Model of P.*

Proof. Trivial from Theorems 8.1 and 8.4. □

The MH semantics trivially guarantees model existence.

Theorem 8.5. ***Minimal Hypotheses semantics guarantees model existence.*** *Let P be an NLP. There is always, at least, one Minimal Hypotheses model of P.*

8.2.1 Relevance

The MH semantics enjoys both Brave Relevance and Relevance, thus allowing for top-down query-driven proof-procedures to be developed.

Theorem 8.6. ***Minimal Hypotheses semantics enjoys Brave Relevance.*** *Let P be an NLP. Then, according to Definition 6.31,*

$$\Big(\forall_{\substack{a\in\mathcal{H}_P \\ M\in Models_{MH}(P)}} a \in M^+ \Rightarrow (\exists_{M_a\in Models_{MH}(Rel_P(a))} M_a \subseteq M \wedge a \in M_a^+)\Big) \wedge \Big(\forall_{\substack{a\in\mathcal{H}_P \\ M_a\in Models_{MH}(Rel_P(a))}} a \in M_a^+ \Rightarrow \exists_{M\in Models_{MH}(P)} M_a \subseteq M\Big)$$

holds.

Theorem 8.7. ***Minimal Hypotheses semantics enjoys Relevance.*** *Let P be an NLP. Then, by Definition 6.30,*

$$(\forall_{M\in Models_{MH}(P)} a \in M^+) \Leftrightarrow (\forall_{M_a\in Models_{MH}(Rel_P(a))} a \in M_a^+)$$

holds.

Proof. It follows trivially from Proposition 6.4 and Theorem 8.6. □

As pointed out before, one of the main goals of this thesis was to come up with a 2-valued semantics allowing for sound and complete top-down proof-procedures to be developed and used for query-solving. The formal definition of a sound and complete proof procedure regarding the MH semantics, along with corresponding proofs and implementation, is itself a task almost substantial enough for another PhD work. We do not

undertake such endeavour here, but only point some directions and hints on the specific issues of MH semantics one such proof-procedure must address: in particular, during top-down querying the solver must identify the SCC a queried literal might be involved in before trying out different minimal sets of hypotheses determining all literals in the SCC and simultaneously not contradicting (and if possible, at least partially satisfying) the top queried literals. Notice SCC detection can be done efficiently (i.e., in polynomial time). SCC detection is potentially simpler than detecting all call-graph loops a literal may be involved in, as this is an NP-complete task. SCC detection is a computationally efficient and deterministic task (there is only one graph of SCCs for any given program), and it can even be performed only once, before any query takes place, as a pre-processing task. In this case, query solving would only need to consult which SCCs a literal is involved in and not to compute them at query-solving-time for optimized efficiency.

8.2.2 Cumulativity

MH semantics enjoys both Cumulativity and Brave Cautious Monotony, thus allowing for lemma storing techniques to be used during computation of answers to queries.

Theorem 8.8. ***Minimal Hypotheses semantics enjoys Cumulativity.*** *Let P be an NLP. Then*

$$\forall_{a,b\in\mathcal{H}_\mathcal{P}}\Big((\forall_{M\in Models_{MH}(\mathcal{P})}a\in M^+)\Rightarrow (\forall_{M\in Models_{MH}(\mathcal{P})}b\in M^+\Leftrightarrow\forall_{M_a\in Models_{MH}(\mathcal{P}\cup\{a\})}b\in M_a^+)\Big)$$

Theorem 8.9. ***Minimal Hypotheses semantics enjoys Brave Cautious Monotony.*** *Let P be an NLP. Then*

$$\forall_{\substack{a\in\mathcal{H}_P\\ M\in Models_{MH}(P)}} a\in M\Rightarrow M\in Models_{MH}(P\cup\{a\})$$

8.2.3 Complexity

The complexity issues usually relate to a particular set of tasks, namely: 1) knowing if the program has a model; 2) if it has any model entailing some set of ground literals (a query); 3) if all models entail a set of literals. In the case of MH semantics, the answer to the first question is an immediate "yes" because MH semantics guarantees model existence for NLPs; the second and third questions correspond (respectively) to Brave and Cautious Reasoning, which we now analyse.

8.2.3.1 Brave Reasoning

The complexity of the Brave Reasoning task with MH semantics, i.e., finding an MH model satisfying some particular set of literals is Σ_2^P-complete.

Theorem 8.10. ***Brave Reasoning with MH semantics is*** Σ_2^P***-complete.*** *Let P be an NLP, and Q a set of literals, or* query*. Finding an MH model such that $M \supseteq Q$ is a Σ_2^P-complete task.*

8.2.3.2 Cautious Reasoning

Conversely, the Cautious Reasoning task with MH semantics, i.e., guaranteeing that every MH model satisfies some particular set of literals is Π_2^P-complete.

Theorem 8.11. ***Cautious Reasoning with MH semantics is*** Π_2^P***-complete.*** *Let P be an NLP, and Q a set of literals, or* query*. Guaranteeing that all MH models are such that $M \supseteq Q$ is a Π_2^P-complete task.*

Proof. Guaranteeing that all MH models are such that $M \supseteq Q$ is equivalent to ensuring there is no $M \not\supseteq Q$. I.e., ensuring there is no $M \supseteq \neg Q$. Cautious Reasoning is thus the complement of Brave Reasoning, and since the latter is Σ_2^P-complete (Theorem 8.10), the former must necessarily be Π_2^P-complete. □

The set of hypotheses $Hyps(P)$ is obtained from $\mathring{P}$ which identifies rules that depend on themselves. The hypotheses are the atoms of DNLs of $\mathring{P}$, i.e., the "atoms of *not*s in loop". A Minimal Hypotheses model is then obtained from a minimal set of these hypotheses sufficient to determine the 2-valued truth-value of every literal in the program. The MH semantics imposes no ordering or preference between hypotheses — only their set-inclusion minimality. For this reason, we can think of the choosing of a set of hypotheses yielding a MH model as finding a minimal solution to a disjunction problem, where the disjuncts are the hypotheses. In this sense, it is therefore understandable that the complexity of the reasoning tasks with MH semantics is in line with that of, e.g., reasoning tasks with SM semantics with Disjunctive Logic Programs, i.e, Σ_2^P-complete and Π_2^P-complete.

8.3 Procedural Methods for Minimal Hypotheses semantics

By Theorem 8.4 we know we can use the constructive method in Definition 7.7 for building the Minimal Hypotheses models of programs with a finite number of layers. In order to make this method sound and complete according to MH semantics, we need to include the Minimal Hypotheses assumption step.

Definition 8.4. Layer-wise Constructive Method for Minimal Hypotheses models. Let P be an NLP with a finite number n of layers. Then all the Minimal Hypotheses models of P can be constructed in the following manner

Algorithm Bottom-Up Construct an MH model($Program\ P$)

$M_{\leq 0} = M_0 = not\ A_P^0$ is the unique *individual layer* model of layer 0, i.e., $P^0 = P^{\leq 0}$;
for each layer index $0 \leq i < n$
  Make the Layer Division of P^{i+1} by $M_{<i+1}$ where $M_{<i+1} = M_{\leq i}$;
  Non-deterministically select a set $H_{i+1} \subseteq Hyps(P^{i+1} : M_{<i+1})$ with
    $heads((P^{i+1} : M_{<i+1}) :: H_{i+1}) = facts((P^{i+1} : M_{<i+1}) :: H_{i+1})$ such that
    H_{i+1} is set-inclusion minimal
  $M^+_{\leq i+1} = facts((P^{i+1} : M_{<i+1}) :: H_{i+1})$;
  $M^-_{\leq i+1} = not\ (A_P^{\leq i+1} \setminus M^+_{\leq i+1})$;
  $M_{\leq i+1} = M^+_{\leq i+1} \cup M^-_{\leq i+1}$;
$M_{\leq n}$ is a Minimal Hypotheses model of P

Figure 8.1: Algorithm Bottom-Up Construct an MH model.

Recall that $P :: I = \widehat{P \cup I}$ as per Definition 7.1 and so, in the current definition we use the $P :: I$ notation instead of $\widehat{P \cup I}$ just for convenience. Although such a constructive method might be indeed useful, our main focus is on top-down querying using only the part of the program relevant to the query, and under this setting a bottom-up constructive method is not primarily important. We return to the issue of top-down query-solving with MH semantics when we address the Relevance property below.

In this chapter we presented the Minimal Hypotheses semantics, a member of the Layer-Decomposable semantics family complying with the Ockham's razor *principle of minimality of hypotheses, and enjoying a number of properties useful for future practical implementations. We now turn to compare this semantics to others and to different approaches to*

logic programming.

9 . Comparisons

> The secret of what anything means to us depends on how we have connected it to all other things we know.
>
> Marvin Minsky

In Chapter 7 we defined the Layer-Decomposable semantics family and showed how the Layer Decomposable models mirror the syntactic structure of layering. In Chapter 8 we defined the Minimal Hypotheses semantics and showed it to be a member of the Layer-Decomposable semantics family (cf. Theorem 8.4). We have also shown that, because all stable models are Minimal Hypotheses models (cf. Theorem 8.1), they are also Layer-Decomposable (cf. Corollary 8.1).
The Minimal Hypotheses semantics is a special case of a Layer-Decomposable semantics, but there might be several other Layer-Decomposable semantics. We are not defining and studying here every possible Layer-Decomposable semantics; instead, we opt to focus on MH semantics as this corresponds to the intuition described in Section 6.5 and enjoys the properties detailed in 6.5.5, and consequently we choose it for comparison with other semantics for NLPs and other approaches to logic.

9.1 Other Semantics for NLPs

As we have seen in Chapter 8, and in particular in Theorem 8.1, all stable models are MH models. Since MH models are always guaranteed to exist for every NLP (cf. Theorem 8.5) and SMs are not, it follows immediately that the Minimal Hypotheses semantics is a strict model conservative generalization (cf. Definition 6.38) of the Stable Models semantics. For Normal Logic Programs, the Stable Models semantics coincides with the Answer-Set semantics (which is a generalization of SMs to Extended Logic Programs), where the latter is known (cf. [110]) to correspond to Reiter's default logic. Hence, all Reiter's default extensions have a corresponding Minimal Hypotheses model. Also, since

Moore's expansions of an autoepistemic theory [160] are known to have a one-to-one correspondence with the stable models of the NLP version of the theory, we conclude that for every such expansion there is a matching Minimal Hypotheses model for the same NLP.

As shown in Theorem 8.2, at least one MH model of a program complies with its well-founded model, although not necessarily all MH models do. E.g., the program from Example 6.2 has the two MH models $\{a, b, not\ c\}$ and $\{a, not\ b, c\}$, whereas the $WFM(P)$ imposes $WFM^+(P) = \{a, b\}$, $WFM^u(P) = \emptyset$, and $WFM^-(P) = \{c\}$. This, as we already know, is due to the set of Hypotheses $Hyps(P)$ of P being taken from $\mathring{P}$ (which is based on the layered support notion) instead of being taken from $\widehat{P}$ (which is based upon the classical notion of support).

Not all Minimal Hypotheses models are Minimal Models of a program. As we have already noted in Chapter 8, the rationale behind MH semantics is minimality of hypotheses, but not necessarily minimality of consequences, the latter being enforceable, if so desired, as an additional requirement, although at the expense of increased complexity.

In [187, 204] (the latter being our M.Sc. thesis) we defined and studied the Revised Stable Models (RSMs) semantics for NLPs. This semantics had the same motivational drives as the ones for this thesis, but the definition of the RSM semantics turned out to be quite hard to grasp and to explain. The RSM semantics built upon the notion of *Reductio ad Absurdum* (RAA): intuitively, if some atom is assumed *false* in an interpretation and the same atom comes out *true* as a consequence of that assumption plus the rest of the interpretation, then, by *reductio ad absurdum*, that interpretation is rejected as a candidate model.

Although we have not yet undergone a thorough comparative analysis between the MH and the RSM semantics, in every example program we tried we noticed that all RSMs were also MH models, although the converse was not true.

Example 9.1. ***Revised Stable Models versus Minimal Hypotheses models.*** *Let P be*

$$\begin{aligned} c &\leftarrow not\ m \\ m &\leftarrow not\ b \\ b &\leftarrow m, not\ c \end{aligned}$$

This program has three MH models: $M_1 = \{b, c, not\ m\}$, $M_2 = \{not\ b, c, m\}$, and $M_3 = \{b, not\ c, m\}$. Of these, only M_1 and M_2 are RSMs.

In a very informal way, it seems that the RAA approach of RSMs can be seen as a restricted version of the hypotheses assumption of MH semantics, but the rigorous comparison of these two semantics is still to be done and, as such, is a topic for future work. Moreover, it turned out that the RSM semantics does not enjoy Cumulativity as

we would desire, and this was one of the additional factors that motivated the definition of the MH semantics. E.g., in the Example 9.1 above, according to the RSM semantics, c is *true* in all RSMs, but if we add c as a fact to the program then $P \cup \{c\}$ admits only $M_2 = \{not\ b, c, m\}$ as an RSM thus rendering m also *true* in all its RSMs. Since m was not *true* in *all* RSMs of P, the RSM semantics fails Cumulativity.

9.2 Argumentation

The relation between logic programs and argumentation systems has been considered for a long time now ([14, 42, 91, 173] amongst many others) and we have also taken steps to understand and further that relationship [188, 189, 190].

In 6.4.2.4 and 6.4.2.5 we saw different approaches to the argumentation perspective of semantics for NLPs.

We are not making a comprehensive comparative analysis of the MH semantics with the Revision Complete Scenarios but just noticing the philosophical similarities between them, namely concerning the minimal assumption of hypotheses producing, as a consequence, a 2-valued complete model. The MH semantics explicitly demands minimality of positive hypotheses, whereas the Revision Complete Scenario argumentation approach makes the *non-redundant* and *unavoidable* requirements on a set of positive hypotheses. These are not the same as explicit minimality, but the rationale behind them is similar. The Approved Models semantics, on the other hand, strives for maximality (in the sense of Definition 5.10) of negative hypotheses M^-. One feature of the Approved Models semantics is that it generalizes Dung's Preferred Extensions to 2-valued complete models. It remains for future work to analyse the Minimal Hypotheses semantics from an argumentation perspective, thoroughly comparing it to the Revision Complete Scenarios and the Approved Models semantics.

In [167], the author (with whom we shared our goals and vision of semantics for NLPs a few years ago) also aims at semantics for LPs, guaranteeing model existence, enjoying relevance, and seamlessly encompassing the argumentation approach. He also compares his own approach to our own [187] thereby showing that other researchers in our scientific community have recognized the importance and non-triviality of the issues and concerns we have been addressing.

9.3 Other Aspects

We can summarize a comparison of some complexity results of the main 2-valued and 3-valued semantics (the SM and the WF semantics, respectively) against the MH semantics's own results in a table:

	Model Existence	Brave Reasoning	Cautious Reasoning
WFS	$\mathcal{O}(1)$	P-complete	P-complete
SMs	NP-complete	NP-complete	coNP-complete
MHs	$\mathcal{O}(1)$	Σ_2^P-complete	Π_2^P-complete

By assigning a meaning, a model, to every NLP, the MH semantics completely lifts any restriction on the use of NLP rules for Knowledge Representation and Reasoning and, as pointed out in 6.5.1, it increases the declarativity of NLPs by unequivocally separating the roles of Integrity Constraints from rules with "non-$\bot$" head. From a "generate-and-test" perspective, with MH semantics, rules with "non-$\bot$" head can be arbitrarily used for the "generate" role, whereas Integrity Constraints (rules with "$\bot$" as head) are used for the "test" role. Most importantly, with MH semantics, *only* ICs can play the "test" role, and *only* "non-$\bot$" head rules can play the "generate" role.

Overall, the MH is a 2-valued semantics (like the SMs one), it guarantees model existence, relevance and cumulativity (like the WFS), and also Brave Relevance and Brave Cautious Monotony. By stemming from a hypotheses assumption approach, the MH semantics is naturally fit for effectively embodying abductive knowledge representation and reasoning, as well as argumentation settings as these can be described by argumentation hypotheses assumption. Much remains to be explored.

The MH semantics builds upon a hypotheses assumption approach. This brings the MH semantics very close to the abduction reasoning mechanism as previously discussed (in 1.2.3, Section 6.2, Section 7.1, and Chapter 8). The MH semantics strives to achieve a pragmatic balance between being conceptually simple stand (like the Stable Models semantics), enjoying a number of useful properties (like the Well-Founded Semantics), providing a natural bridge between deductive and abductive reasoning, and allowing for relatively simple implementations. Now that we have a slightly better understanding of the relations between the MH and other semantics and approaches to logic programs, we can consider Reasoning with Logic Programs, including abductive reasoning, with the MH semantics.

Part III

Reasoning with Logic Programs

10 . Abductive and Deductive Reasoning

> Logic: The art of thinking and reasoning in strict accordance with the limitations and incapacities of the human misunderstanding.
>
> Ambrose Bierce

In the previous parts of this thesis we studied desirable properties of semantics and of program structure, with a view to represent knowledge with Normal Logic Programs, and also how to extend that representation formalism to deal with explicit negation and disjunction. We identified the intrinsic structure of knowledge as written as a set of sentences, or logic rules, using such formalisms. We outlined the desirable properties a semantics for such logic programs should enjoy, defined the family of semantics that conforms to the structure of knowledge. Then, we defined the Minimal Hypotheses models semantics, showed how if fits in that family, enjoys those properties, and compared it with other semantics and approaches to logic programs. We now address the problem of reasoning with logic programs under MH semantics.
We begin this chapter by establishing a quasi-*equivalence relation between abduction and deduction by showing that deductive reasoning can be seen as a particular case of abductive reasoning, and that, on the other hand, abductive reasoning can be emulated by purely deductive reasoning. Then we show how general abductive and deductive reasoning boil down to local knowledge existential query-answering. After doing so we turn to the issue of checking for side-effect consequences of such query answers, for they might be of unavoidable interest in practical applications.*
The contributions in this chapter have been previously published in [182, 192, 195].

10.1 Abduction and Deduction

As we have seen in Chapter 1, reasoning with Knowledge Bases represented as Logic Programs can be split into three possible methods: deduction, induction, and abduction. Induction, as said before, is out of the scope of this thesis, as it amounts to a sort of learning which clearly is not a topic to be broached here.

In 1.2.3 we outlined the abductive problem-solving general idea as *"hypothesizing plausible reasons sufficient for justifying given observations or supporting desired goals"*. The simple common formalization of the abductive problem solving framework was stated as: $\delta \subseteq \Delta$ is an *abductive* solution to goal (or *abductive* query) Q given a Knowledge Base KB iff $KB \cup \delta \models Q$ and $KB \cup \delta \not\models \bot$, where Δ is the set of abducible hypotheses — and usually abducibles are considered to have no rules, i.e., $\Delta \cap heads(KB) = \emptyset$. Without loss of generality, we consider such abducibles to consist just of literals, as reasons in the form of rules can simply have their bodies labeled by inclusion of hypothetical literals that can switch a rule on or off.

Deductive reasoning can be easily perceived as a special case of abductive reasoning: the one where there are no abducible hypotheses, and thus conclusions can only stem from the (abducible-free) rules of the program.

On the other hand, 2-valued abductive logic programs can be modeled by "non-abductive" normal logic programs: because they do not depend on any other literal in the program, abducibles can be modeled in a Logic Program system without specific abduction mechanisms by including, for each abducible, a pair of rules generating the two alternative abductions, e.g.,

$$\begin{aligned} abducible &\leftarrow not\ neg\_abducible \\ neg\_abducible &\leftarrow not\ abducible \end{aligned}$$

where both *abducible* and *neg_abducible* are new reserved atoms, (i.e., not previously in $\mathcal{H}_P$) representing the abducible and its negation, respectively. This approach is also followed by [238].

According to Definition 6.31, a semantics enjoying Brave Relevance allows for query answering with partial knowledge only (i.e., resorting exclusively to the part of the program relevant for the query). This guarantees that a sub-model entailing the query found during relevant query-solving is extendible to a complete model. With this NLP representation of abducibles, an abductive answer to a query Q using only partial knowledge will include just the abducibles $\delta \subseteq \Delta$ considered during the query-solving; i.e., the abductive sub-model M_δ found entailing the query ($M_\delta \models Q$) is a 3-valued abductive model guaranteed to be a part of a 2-valued abductive model M for the complete program ($M \models KB$).

This means that all abducibles that are not explicitly assumed *true* or *false* in M_δ remain *undefined* in this sub-model, whereas they will have a 2-valued truth-value in each whole model $M \models KB$ such that $M \supseteq M_\delta$.

With this we see how in fact abduction and deduction are just two different ways of dealing with the same sort of logical reasoning, which 1) buttresses even more the hypotheses assumption approach as a way to define a semantics for (not necessarily abductive) logic programs; and 2) is in accordance with [61].

We now turn to the general issue of (abductive/deductive) logical reasoning as query answering.

10.2 Reasoning as Query-Answering

The semantic entailment $\models$ used in the formulas above leaves open the issue of universal versus existential (abductive/deductive) goal/query entailment as explained in 1.2.1; and yet another orthogonal axis of problem framing covers the scope of reasoning — complete versus partial knowledge reasoning, as outlined in 1.3.

One of the main motivations behind this work is to provide a formal scaffolding upon which practical reasoning tools can be developed allowing for abductive/deductive existential query answering resorting only to partial knowledge. This is important for the subsequent development of efficient practical tools and also because, when being guaranteed, further generalizations can be developed to encompass all the other scenarios, namely:

- *universal* (abductive/deductive) query answering (either using *partial* or *complete* knowledge) can be achieved by systematically checking each and every individual existential query answer obtained with the same scope of knowledge
- *existential* (abductive/deductive) query answering using *complete* knowledge is trivially achievable if existential query answering is attainable using only *partial* knowledge — the "non-local" part of the *global* knowledge can simply be ignored

As seen before, partial knowledge reasoning requires the Relevance property — in its several variants presented in 6.5.5.2. Since the Minimal Hypotheses semantics enjoys all the Relevance properties (Relevance, and Brave Relevance) we can safely use it as the underlying semantics for abductive/deductive universal/existential query answering resorting to complete/partial knowledge.

Definition 10.1. Local Knowledge Existential Abductive Query-answering in a Logic Program. Let P be a NLP representation of the user's Knowledge Base, Δ a set of abducible hypotheses such that $\Delta \cap heads(KB) = \emptyset$, S a semantics for NLPs enjoying Brave Relevance, and Q a user-specified (abductive) query.

$M_Q \cup \delta$ is a local-knowledge abductive existential answer to query Q according to semantics S iff

- $\delta \subseteq \Delta$
- $M_Q \cup \delta$ is a model of $Rel_P(Q) \cup \delta$ according to S
- $M_Q \cup \delta \supseteq Q$

Since S enjoys Brave Relevance, we know that there is some model M of $\mathcal{P} \cup \delta$ such that $M \supseteq M_Q$, i.e., Brave Relevance guarantees existence of a complete knowledge answer given a partial knowledge one. Also, notice that whenever $\Delta = \emptyset$, or $\delta = \emptyset$ the definition above coalesces into the simple deduction scenario (abduction-free).

As we have just seen, both deduction and abduction are fully covered (in both universal and existential variants) by partial knowledge query-answering. In practical applications, however, this may not be enough as it may be useful, sometimes indeed necessary, to check for side-effects of an answer to a query, because only partial – not complete – models are obtained in answer to queries. We now turn to further explore this aspect of logic reasoning, so that side-effects can be just checked for, without conducing to abductions just on their behalf.

10.3 Inspecting Side-Effects of Abductions

One application of abductive reasoning is that of finding which actions to perform, their names being coded as abducibles. Under this setting, besides needing to abductively discover which hypotheses to assume in order to satisfy some query/goal condition, we may also want to know some of the side-effects of those assumptions; in fact, this is a rather rational thing to do. But, most of the time, we do not wish to know *all* possible side-effects of our assumptions, as some of them will be irrelevant to our concern. Likewise, the side-effects inherent in abductive explanations might not all be of interest.

It is also important to point out that such a conceptual inspection mechanism is independent of the underlying semantics used for the logic programs at hand: the side-effect

inspection construct is useful in itself independently of whether the Minimal Hypotheses models, the Stable Models, the Well-Founded, or any other semantics is being used for the abductive logic program.

Example 10.1. ***Relevant and irrelevant side-effects.***

Consider this logic program where 'drink_water' and 'drink_beer' are the abducibles.

wet_glass	$\leftarrow$	use_glass						$unsafe\_drive$	$\leftarrow$	$drunk$	
use_glass	$\leftarrow$	$drink$		$\bot$	$\leftarrow$	$not\ drink, thirsty$				$drunk$	$\leftarrow$	$drink\_beer$
$drink$	$\leftarrow$	$drink\_water$	$drink$	$\leftarrow$	$drink\_beer$		$thirsty$				

Suppose we want to satisfy the Integrity Constraint, and also to check if we get drunk or not. However, we do not care about the glass becoming wet — that being completely irrelevant to our current concern. In this case, computation of whole models is a waste of time, because we are interested only in a subset of the program's literals. Moreover, in this example, we may simply want to know the side-effects of the possible actions in order to decide (to drive or not to drive) ***after*** *we know which side-effects are true. In such a case, we do not want to simply introduce an IC expressed as* $\bot \leftarrow unsafe\_drive$ *because that would always impose abducing* $not\ drink\_beer$*. We want to allow all possible solutions for the single IC* $\bot \leftarrow not\ drink, thirsty$ *and then check for the side-effects of each abductive solution.*

Checking for consequences of abductions amounts to inspecting the side-effects literals of interest whose truth-value may be affected by the abductions. We now turn to define both the declarative and procedural semantics of such an *inspection* mechanism.

10.3.1 Backward and Forward chaining

Backward and Forward chaining are but different procedural methods to implement a deductive mechanism [55]. If abductive reasoning is desired, any deductive method plus a hypothesizing mechanism will suffice [21, 61]. Procedurally, abductive query-answering can be seen as a backward-chaining process, a top-down dependency-graph oriented proof-procedure. On the other hand, finding the side-effects of a set of abductive assumptions may be conceptually envisaged as forward-chaining, as it consists of progressively deriving conclusions from the assumptions until the truth value of the chosen side-effect literals is determined.

The problem with full-fledged forward-chaining is that too many (often irrelevant) conclusions of the adopted assumptions are derived. Wasting time and resources deriving them only to be discarded afterwards is a flagrant setback. Worse, there may be many alternative models satisfying an abductive query (and the ICs) whose differences just repose

on irrelevant conclusions. This way, unnecessary computation of irrelevant conclusions can be compounded by the need to discard irrelevant alternative complete models too.

A more intelligent solution would be afforded by selective forward-chaining, where the user would be allowed to specify those conclusions (s)he is focused on, and only those would be computed. Combining backward-chaining with selective forward-chaining would allow for a greater precision in specifying what we wish to know, and improve efficiency altogether. In the sequel we show how such a selective forward chaining from a set of abductive hypotheses can be replaced by backward chaining from the focused on conclusions — the inspection points — by virtue of a controlled form of abduction which, never performing extra abductions, just checks for abducibles assumed elsewhere.

10.3.2 Operational Intuition of Inspection Points

In order to endow abductive logic programs with this side-effect inspection mechanism we need to extend the abductive logic programming language with the special reserved construct *inspect*/1. The operational intuition of inspection points goes as follows: the user wraps with the reserved construct *inspect*(.) the literal whose truth-value (s)he intends to check being a consequence of a solution to the query; then, when the user launches an abductive query (which itself may include *inspect*ed literals), abducible hypothesis are adopted, as usual, to satisfy the query except when the abduction step is to be performed inside a sub-tree with an *inspect*ed literal as root.

Example 10.2. ***Police and Tear Gas Issue.*** *Consider this NLP, where 'tear_gas', 'fire', and 'water_cannon' are the only abducibles. Notice that inspect is applied to calls.*

$$\begin{array}{rcl} \bot & \leftarrow & police, riot, not\ contain \\ contain & \leftarrow & tear\_gas \\ smoke & \leftarrow & fire \\ police & & \end{array} \qquad \begin{array}{rcl} contain & \leftarrow & water\_cannon \\ smoke & \leftarrow & inspect(tear\_gas) \\ riot & & \end{array}$$

Notice the two rules for 'smoke'. The first states that one explanation for 'smoke' is fire, when assuming the hypothesis 'fire'. The second states 'tear_gas' is also a possible explanation for smoke. However, the presence of tear gas is a much more unlikely situation than the presence of fire; after all, tear gas is only used by police to contain riots and that is truly an exceptional situation. Fires are much more common and spontaneous than riots. For this reason, 'fire' is a much more plausible explanation for 'smoke' and, therefore, in order to let the explanation for 'smoke' be 'tear_gas', there must be a plausible reason — imposed by some other likely phenomenon. This is represented

by inspect(tear_gas) instead of simply 'tear_gas'. Declaratively, the 'inspect' Operationally, the 'inspect' construct disallows regular abduction to be performed whilst trying to find an abductive answer to 'tear_gas' in a top-down abductive proof-procedure. I.e., if we take 'tear_gas' as an abductive solution for smoke, this rule imposes that the step where we abduce 'tear_gas' must be performed elsewhere, not under the derivation tree for 'smoke'. Thus, 'tear_gas' is an inspection point. *The IC, because there is 'police' and a 'riot', forces 'contain' to be* true, *and hence, 'tear_gas' or 'water_cannon' or both, must be abduced. 'smoke' is only explained if, at the end of the day, 'tear_gas' is abduced to enact containment. Abductive solutions should be plausible, and 'smoke' is plausibly explained by 'tear_gas' if there is a reason, a best explanation, that makes the presence of tear gas plausible; in this case the riot and the police. Plausibility is an important concept in science, for lending credibility to hypotheses. Assigning plausibility measures to situations is an orthogonal issue.*

If we were to remove the '$\bot \leftarrow police, riot, not\ contain$' IC and launch the abductive query 'smoke', the unique abductive answer would be '$fire$'. However, if we were to remove both the IC and the inspect around 'tear_gas' in the '$smoke \leftarrow inspect(tear\_gas)$' rule — thus rendering it '$smoke \leftarrow tear\_gas$' — then 'tear_gas' would become an alternative abductive answer to the query 'smoke'. This clearly shows how the inspect construct enforces the passive role of merely checking if a literal's truth-value follows as a consequence of abductions instead of allowing extra abduction to take place solely in order to satisfy the query.

In this example, another way of viewing the need for the new mechanism embodied by the inspect predicate is to consider we have 2 agents: one is a police officer and has the possibility of abducing (corresponding to actually throwing) 'tear_gas'; the other agent is a civilian who, obviously, does not have the possibility of abducing (throwing) 'tear_gas'. For the police officer agent, having the $smoke \leftarrow inspect(tear\_gas)$ rule, with the inspect is unnecessary: the agent knows that 'tear_gas' is the explanation for 'smoke' because it was himself who abduced (threw) 'tear_gas'; but for the civilian agent the inspect in the $smoke \leftarrow inspect(tear\_gas)$ rule is absolutely indispensable, since he cannot abduce 'tear_gas' and therefore cannot know, without inspecting, if that is the real explanation for 'smoke'.

10.3.2.1 Meta-abduction

An intuitive way to model the *inspect*ion mechanism is by viewing it as meta-abduction. Intuitively, when an abducible is considered under mere *inspect*ion, meta-abduction abduces only the intention to *a posteriori* check for its abduction elsewhere, i.e. it abduces the intention of verifying that the abducible is indeed adopted, but elsewhere, not

under *inspect*ion. A practical operational way to implement such meta-abduction is, when we want to meta-abduce some abducible 'x', we abduce a literal '$consume(x)$' (or '$abduced(x)$') instead, which represents the intention that 'x' is eventually abduced elsewhere in the process of finding an abductive solution to the top query/goal. The check is performed after a complete abductive answer to the top query is found. Operationally, 'x' will already have been, or will later be, abduced as part of the ongoing solution to the top goal.

Example 10.3. ***Nuclear Power Plant Decision Problem.*** *This example was extracted from [215] and adapted to our current designs, and its abducibles do not represent actions.*

In a nuclear power plant there is decision problem: cleaning staff will dust the power plant on cleaning days, but only if there is no alarm sounding. The alarm sounds when the temperature in the main reactor rises above a certain threshold, or if the alarm itself is faulty. When the alarm sounds everybody must evacuate the power plant immediately! Abducible literals are $cleaning\_day$, $temperature\_rise$ *and* $faulty\_alarm$.

$$\begin{aligned} dust &\leftarrow cleaning\_day, inspect(not\ sound\_alarm) \\ \bot &\leftarrow not\ cleaning\_day \\ evacuate &\leftarrow sound\_alarm \\ sound\_alarm &\leftarrow temperature\_rise \\ sound\_alarm &\leftarrow faulty\_alarm \end{aligned}$$

Satisfying the unique IC imposes $cleaning\_day$ true *and gives us three minimal abductive solutions:*

$$\begin{aligned} S_1 &= \{dust, cleaning\_day\} \\ S_2 &= \{cleaning\_day, sound\_alarm, temperature\_rise, evacuate\} \\ S_3 &= \{cleaning\_day, sound\_alarm, faulty\_alarm, evacuate\} \end{aligned}$$

If we pose the query $?-not\ dust$ *we want to know what could justify the cleaners dusting not to occur given that it is a cleaning day (enforced by the IC). However, we do not want to abduce the rise in temperature of the reactor nor to abduce the alarm to be faulty in order to prove not dust. Any of these justifying two abductions must result as a side-effect of the need to explain something else, for instance the observation of the sounding of the alarm, expressible by adding the IC* $\bot \leftarrow not\ sound\_alarm$, *which would then force the abduction of one or both of those two abducibles as plausible explanations. The* $inspect/1$ *in the body of the rule for dust prevents any abduction below* $sound\_alarm$ *to be made just to make not dust true. One other possibility would be for two observations, coded by ICs* $\bot \leftarrow not\ temperature\_rise$ *or* $\bot \leftarrow not\ faulty\_alarm$, *to be present in order for not dust to be true as a side-effect. A similar argument can be made about evacuating: one thing is to explain why evacuation takes place, another altogether is to justify it as*

necessary side-effect of root explanations for the alarm to go off. These two pragmatic uses correspond to different queries: $?-evacuate$ *and* $?-inspect(evacuate)$*, respectively.*

10.3.3 Declarative Semantics of Inspection Points

A simple transformation maps programs with inspection points into programs without them. Mark that the Minimal Hypotheses models of the transformed program where each $abducible(X)$ is matched by the abducible X (X being a literal a or its default negation $not\ a$) clearly correspond to the intended procedural meanings ascribed to the inspection points of the original program.

The intuition for the program transformation we present next is as follows: we transform a program P into another $\Pi(P)$ which is a duplication of P where the duplicate rules are used for *inspect*ing, i.e., under these duplicate rules all references to abducibles L are replaced by meta-abductions ($abduced(L)$). This way, under an *inspect* rule there are no abducibles, only eventually meta-abductions.

Definition 10.2. Transforming Inspection Points. Let P be a program containing rules whose body possibly contains inspection points. The program $\Pi(P)$ consists of:

1. all the rules obtained by the rules in P by systematically replacing:
   - $inspect(not\ L)$ with $not\ inspect(L)$;
   - $inspect(a)$ or $inspect(abduced(a))$ with $abduced(a)$ if a is an abducible, and keeping $inspect(a)$ otherwise.
2. for every rule $A \leftarrow L_1, \ldots, L_t$ in P, the additional rule: $inspect(A) \leftarrow L_1^{'}, \ldots, L_t^{'}$ where for every $1 \leq i \leq t$:
$$L_i^{'} = \begin{cases} abduced(L_i) & \text{if } L_i \text{ is an abducible} \\ inspect(X) & \text{if } L_i \text{ is } inspect(X) \\ inspect(L_i) & \text{otherwise} \end{cases}$$
3. for every abducible A, the additional rules:
$$\begin{aligned} A &\leftarrow not\ neg\_A \\ neg\_A &\leftarrow not\ A \\ abduced(A) &\leftarrow not\ abduced(neg\_A) \\ abduced(neg\_A) &\leftarrow not\ abduced(A) \\ \bot &\leftarrow abduced(A), not\ A \\ \bot &\leftarrow abduced(neg\_A), not\ neg\_A \end{aligned}$$

The semantics of the *inspect* predicate is exclusively given by the generated rules for *inspect*.

Example 10.4. ***Transforming a Program P with Nested Inspection Levels.*** *Consider the following program where the abducibles are* a, b, c, d

$$\begin{array}{lcllcl} x & \leftarrow & a, inspect(y), b, c, not\ d & y & \leftarrow & inspect(not\ a) \\ z & \leftarrow & d & y & \leftarrow & b, inspect(not\ z), c \end{array}$$

Then, $\Pi(P)$ *is:*

$$\begin{array}{rcl} x & \leftarrow & a, inspect(y), b, c, not\ d \\ inspect(x) & \leftarrow & abduced(a), inspect(y), abduced(b), abduced(c), not\ abduced(d) \\ y & \leftarrow & not\ abduced(a) \\ inspect(y) & \leftarrow & not\ abduced(a) \\ y & \leftarrow & b, not\ inspect(z), c \\ inspect(y) & \leftarrow & abduced(b), not\ inspect(z), abduced(c) \\ z & \leftarrow & d \\ inspect(z) & \leftarrow & abduced(d) \end{array}$$

plus the rules for the abducibles a, b, c, d *produced by step 3 of the program transformation in def. 10.2. If, say, we want* x *to be* true*, we can simply add the IC* $\bot \leftarrow not\ x$*, in which case the unique abductive layer decomposable model of* $\Pi(P) \cup \{\bot \leftarrow not\ x\}$ *respecting the inspection points is:*
$\{x, a, b, c, abduced(a), abduced(b), abduced(c), inspect(y), abduced(neg\_d)\}$*. Note that for each* $abduced(a)$ *the corresponding* a *is in the model.*

10.3.4 Inspection Points in Other Abductive Systems

In the context of abductive logic programs, we have presented a new mechanism of inspecting literals that can be used to check for side-effects, by relying on conditional meta-abduction. We have implemented the inspection mechanism within the Abdual [21] meta-interpreter (cf. 11.3). We have further checked that our approach can easily be adopted, in part, by other systems [57] with the help of these cited authors.

HyProlog [57] is an abduction/assumption system which allows for the user to specify if an abducible is to be consumed only once or many times. In HyProlog, as the query solving proceeds, when abducible/assumption consumptions take place, they are executed by storing the corresponding consumption intention in a store. After an abductive solution for a query is found, the actual abductions/assumptions are matched against the consumption intentions. Overall, there is not such a big gap between the operational semantics of

HyProlog and the inspection points implementation we present; however, there is a major functional difference: in HyProlog we can only specify consumption directly on abducibles, whereas in our more general inspection points approach we can declare inspection of any literal (not just abducibles) — meaning any abducible found below an inspect-wrapped literal call is automatically just inspected.

In [215], the authors detect a problem with the IFF abductive proof procedure [104] of Fung and Kowalski, in what concerns the treatment of negated abducibles in integrity constraints (e.g. in their examples 2 and 3). They then specialize IFF to avoid such problems, which arise only in ICs, and prove correctness of the new procedure. The detected problem refers to the active use of an IC comprising in its body some *not* A, where A is an abducible, whereas the intended use should be a passive one, simply checking whether some A is proved in the abductive solution found. To that effect, by means of an inference rule used during query evaluation, it is as if they replaced such occurrences of *not* A by *not* $provable(A)$, before moving each as a disjunct $provable(A)$ to the IC head along with other disjuncts, so as to ensure that no new abductions are allowed during IC checking, by virtue of $provable/1$. For a detailed exposition the reader is referred to their section 4.2. Our own work generalizes the scope of the problem they solved, and solves the problems arising in this wider scope. For one, we abduce both positive and negative literals, and the latter are not true by default. Moreover, we allow for passive checking not just of negated abducibles but also of positive ones, as well as passive checking of any literal, whether or not abducible and whether in ICs or other rules. Furthermore, we allow to single out which specific occurrences are passive or active. Thus, we can cater for both passive and active ICs, depending on the desired usage. Our solution uses abduction itself to solve the problem, making it general for deployment in other abductive frameworks and procedures.

We have covered some aspects of abductive and deductive reasoning with logic programs, and also presented a practical method for inspecting side-effects of abductions which can be used for different purposes. We now turn to describe our implementation efforts of some of the mechanisms we use in this thesis.

---

11 . Implementations

> To build may have to be the slow and laborious task of years. To destroy can be the thoughtless act of a single day.
>
> Winston Churchill

It is one of our ultimate goals to build a software application that uses the Minimal Hypotheses semantics as the underlying platform for Knowledge Representation and Reasoning, and allows for top-down query-solving with abduction and respective side-effect inspection. Once the theoretical fundaments for this are pinned down, however, such a task is more of a software engineering challenge than a knowledge representation and reasoning within computational logic one, the latter being the scope of our present work. In this thesis's work we focus on the theoretical issues as they must be dealt with before any practical implementation efforts can be undertaken. Nonetheless, we wanted to take the initial implementation steps for some of the core components of the theoretical scaffolding we presented in earlier chapters.

We firstly describe these prototypical proof-of-concept implementations now: first, our contribution to the innards implementation of the loop detection *mechanism of the Well-Founded Semantics in XSB-Prolog, which is the engine of our choice; then, our envisioned approach for solving queries in a top-down fashion resorting to XSB-Prolog and its XASP interface to Smodels; and secondly and orthogonally, our implementation of abductions' side-effect inspection described in Chapter 10. The motivation for the latter's implementation arises from the need to be able to check for specific side-effect consequences of abductive solutions to a query, but without in the process making further abductions whilst doing the checking.*

The reader not interested in implementations may skip this chapter without loss of any other non-implementation-related content. Most of the contents of this chapter has been previously published in our contributions [182, 192, 194, 195, 232].

11.1 Enforcing the WFS in XSB-Prolog via Answer Completion

The Prolog language has been for quite some time one of the most accepted means to codify and execute logic programs, and as such has become a useful tool for research and application development in logic programming. Several stable/production stage implementations have been developed and refined over the years, with plenty of working solutions to pragmatic issues, ranging from efficiency and portability to explorations of language extensions. The XSB Prolog system[1] is a particular instance of a Prolog system with a special focus on implementing program evaluation following the WFS for NLPs.

XSB-Prolog intends to implement the WFS but, up to now, for efficiency reasons, it had not yet included the *loop detection* mechanism (cf. Definition 6.10). Without this *loop detection* the XSB-Prolog does not fully correctly implement the Well-Founded Semantics and, equivalently, it does not fully correctly implement the Remainder operator (cf. Definition 6.13). Herein we discuss our practical efforts towards implementing that *loop detection* component in the SLG-WAM of XSB-Prolog. The vast majority of the progress and results reported in this section are to be credited to our co-author Terrance Swift [232], but since we did give a hands-on persistent contribution to them, and they are closely related to the rest of this thesis, we include them here as well.

Answer Completion amounts to a reification of the *positive loop detection and elimination* necessary for a fully correct implementation of the WFS.

11.1.1 Motivation — Unfounded Sets detection

Designers of logic programming engines must weigh the usefulness of operations against the burden of complexity they require. Perhaps the best known example is the *occurs check* in unification. Prologs derived from the WAM do not usually perform occurs check between two terms. Rather, the occurs check, if needed, must be explicitly invoked through the ISO predicate
`unify_with_occurs_check/2` or a similar mechanism. For evaluating normal programs using tabling, checking for certain positive loops (cf. Definition 6.10) involves similar considerations. While most positive loops can be efficiently checked, positive subloops within larger negative loops are more difficult to detect, and account for the over-linear complexity of evaluating a program P according to WFS, which is $atoms(P) \times size(P)$, where $atoms(P)$ is the number of atoms of P and $size(P)$ is the number of rules of P. As implemented in XSB, the SLG-WAM detects positive loops between tabled subgoals

[1]Both the XSB Logic Programming system and Smodels are freely available at: `http://xsb.sourceforge.net` and `http://www.tcs.hut.fi/Software/smodels`.

so that answers are not added to a table unless they are true, or if they are involved in a loop through negation and thus being undefined at the time of their addition (termed *conditional answers*). This sort of evaluation can be done in time linear in $size(P)$ [2]. However, a situation can arise where certain conditional answers are later determined to be true or false. This determination might break a negative loop, which then uncovers a positive loop and makes the answers false. Within SLG, this situation is addressed by the Answer Completion operation, which is not implemented within the currently available version of the SLG-WAM. So far, the lack of Answer Completion has not proven a problem for most programs. However, the SLG-WAM is increasingly being used to produce well-founded residues for highly non-stratified programs for applications involving intelligent agents (e.g. [183]), where the need for Answer Completion is greater.

In this section we examine issues involved in adding Answer Completion to the SLG-WAM. We illustrate the situation of a positive loop being uncovered when a negative loop is resolved through a concrete example, and then we provide a formal result on the contribution Answer Completion makes to the complexity of computing WFS. We introduce an algorithm for efficiently performing Answer Completion (subject to its complexity), and discuss performance results obtained by implementing it in the SLG-WAM. We must assume knowledge of tabled evaluation of WFS through SLG resolution [56] as well as certain data structures of the SLG-WAM [217].

Example 11.1. ***Need for positive loop detection.*** *The following program is soundly, but not completely, evaluated by the SLG-WAM, where tnot/1 indicates that tabled negation is used:*

$$\begin{array}{rcl} & :- & table\ p/1, r/0, s/0. \\ p(X) & :- & tnot(s). \\ p(X) & :- & p(X). \\ s & :- & tnot(r). \\ s & :- & p(X). \\ r & :- & tnot(s), r. \end{array}$$

The well-founded model for this program has true atoms $\{s\}$ *and false atoms* $\{r, p(X)\}$. *Recall that literals that do not have a proof and that are involved in loops over default negation are considered* undefined *in WFS. Unproved literals involved only in positive loops, i.e., without negations, are* unsupported *and, hence,* false *in WFS. Accordingly,* $p(X)$*, whose second clause fails, is* false*; however, a query to* $p(X)$ *in XSB indicates that* $p(X)$ *is* undefined. *The reason is that during evaluation the engine detects a strongly connected component (SCC) of mutually dependent goals containing* $p(X)$*,* r *and* s*, along*

[2] This result corresponds to Theorem 1 of [232] whose proof and related definitions can be found in that paper's online full version with proofs at `http://www.cs.sunysb.edu/~tswift/webpapers/iclp-09-iac.pdf`

with negative dependencies, and no answers for any of these goals. In such a situation, the SLG-WAM delays negative literals and continues execution. Here, the literal $tnot(s)$ *in the rule* $p(X) :- tnot(s)$ *is delayed, producing an answer* $p(X) :- tnot(s)|$*, indicating that* $p(X)$ *is conditional on a* delay list*, here* $tnot(s)$*. That answer is returned to the goal* $p(X)$ *in the clause* $p(X) :- p(X)$ *and a conditional answer* $p(X) :- p(X)|$ *is derived. Later, a positive loop is detected for* r*, causing its truth value to become* false*. The failure of* r *causes* s *to become* true*, and* Simplification *removes the answer* $p(X) :- tnot(s)|$*. At this stage, however, no further simplification is possible for* $p(X) :- p(X)|$*, which is now unsupported.*

The Answer Completion operation addresses such cases by detecting positive loops in dependencies among conditional answers. More precisely, Answer Completion marks *false* sets of answers that are not *supported*: i.e. conditional answers for completed subgoals that contain only positive, and no negative dependencies in their delay lists. The creation of unsupported answers are uncommon in the SLG-WAM because its evaluation is *delay minimal* – that is, the engine performs no unnecessary Delaying operations [218]. Delay minimality reduces the need for simplification of dependencies among answers, and thereby the chances of uncovering positive loops among answers, as with the answer $p(X) :- p(X)|$ above.

11.1.2 Implementation of Answer Completion

Within an SLG evaluation, a tabled subgoal can be marked as *complete*, which indicates that all possible answers have been produced for the subgoal, although Simplification and Answer Completion operations may remain to simplify or delete conditional answers. Completed subgoals do not require execution stack space, but only table space, so that completing subgoals as early as possible is a critical step for engine efficiency. Accordingly the SLG-WAM performs *incremental completion* via a completion instruction, which maintains information about mutually dependent sets of subgoals (SCCs), and completes each SCC when all applicable operations have been performed. In addition to marking each subgoal S in an SCC as complete, if S failed (has no answers) the completion instruction may initiate Simplification for conditional answers that depend negatively on S. In terms of Answer Completion, observe that any positive loop among the delayed literals of conditional answers must be contained within the SCC being completed, as each delayed literal was a selected literal before it was delayed. This incremental approach has several benefits. Performing Answer Completion operation within the completion instruction restricts the space that any such operation needs to search. In addition, performing Answer Completion after all other Simplification operations have been

performed on answers within the SCC similarly reduces search space. As a final optimization, Answer Completion is not required unless delaying has been performed within the SCC, a fact that is easily noted using data structures in the SLG-WAM's *Completion Stack*, which maintains information about SCCs.

11.1.2.1 Iterate Answer Completion

The pseudo code for Iterate Answer Completion, which traverses all subgoals in the SCC using the *Completion Stack*, and checks each answer for support, deleting unsupported answers from the table and invoking Simplification operations, is presented in Figure 11.1. Simplification may remove further negative loops, and uncover new unsupported other answers as a side-effect. In such case, the Answer Completion procedure should be executed once more, and this is guaranteed by the use of the $reached\_fixed\_point$ flag. A fixed-point is reached when all answers within the scope of the SCC are known to be supported.

Algorithm Iterate Answer Completion ($CompletionStack$)

```
reached_fixed_point = FALSE;
while not reached_fixed_point
    reached_fixed_point = TRUE;
    for each subgoal S in the Completion Stack
        for each answer A for subgoal S
            if not Check_Supported_Answer(A)          /* A is unsupported */
                reached_fixed_point = FALSE;
                delete A;
                propagate A's deletion's simplifications;
```

Figure 11.1: Algorithm Iterate Answer Completion.

11.1.2.2 Check Supported Answer

This procedure (Figure 11.2) does the actual check of whether a (positive) answer is unsupported. It detects positive loops whenever it encounters an answer that has already been visited and which is in the SCC. In this case, the algorithm terminates returning $FALSE$ to indicate the answer is unsupported. On the other hand, if the answer has been visited but is not part of the SCC, it means such answer has been produced during some other branch of query-solving and was therefore, rightfully supported and stored in the table: the algorithm terminates returning $TRUE$.

Checking a non-visited answer consists of 1) marking it as visited; 2) adding it to the state of the search (stored in the *Completion Stack*); and then 3) traversing all the Delay Elements (literals) of the Delay Lists for the answer recursively checking each in turn for supportedness. Whenever an answer is determined to be unsupported, all Delay Lists containing (Delay Elements that reference) it are deleted, which may cause further simplification and iterations of ANSWER COMPLETION.

Algorithm Check Supported Answer($Answer$)

```
if Answer has already been visited
    if Answer is in the SupportCheckStack return FALSE;
    else return TRUE;
else
    mark Answer as visited;
    push Answer onto the SupportCheckStack;
    mark Answer as supported_unknown;
    for each Delay List DL for Answer
        if Answer is supported_true exit loop;
        mark DL as supported_true;
        for each Delay Element DE in the Delay List DL
            if DL is not supported_true exit loop;
            if DE is positive and it is in the SupportCheckStack
                recursively call Check Supported Answer(Answer of DE)
                if Answer of DE is not supported_true
                    mark DL as supported_false;
    if DL is supported_false
        remove DL from Answer's DLs list
        if Answer's DLs list is now empty
            delete Answer node;
            simplify away unsupported positives of Answer;
        else mark Answer as supported_true;
    if the Answer node was deleted return TRUE;
    else return FALSE;
```

Figure 11.2: Algorithm CHECK SUPPORTED ANSWER.

Example 11.1 is actually representative of the typical situation where ANSWER COMPLETION is needed. This is so because it contains (at least) two rules for some literal (in this case $p(X)$) where the first one depends on a loop through negation (rendering $p(X)$ *undefined*) and the second one depend on a positive loop, which is unsupported. The "undefinedness" coming from the first clause is passed on to the $p(X)$ in the body of

the second one. Only then must ANSWER COMPLETION be used to clean away the delay list with $p(X)$ from the answer coming from the second clause for $p(X)$. The "pathological" nature of this example follows from the, until now, XSB's SLG-WAM inability to rightfully detect and simplify away unsupported literals such as $p(X)$.

11.2 Top-down Query-Solving Approach with XSB-Prolog

In order to find a solution to a query, a Minimal Hypotheses semantics based top-down query-solver will need to find a Minimal Hypotheses sub-model entailing the query's literals. The Relevance (and Brave Relevance) of the MH semantics ensures such a sub-model is extendable to a complete model. Moreover, these properties also ensure that considering only the relevant part of the program is sufficient (and necessary) to find such an answer. To accomplish this task a MH-based query solver will need to

1. Collect rules in the program relevant for the query
2. Assign the truth-value *false* to atoms with no rules, and to atoms in *positive loops* (cf. Section 11.1)
3. Assign the truth-value *true* to facts
4. Detect and "solve" strongly connected components of rules according to the Minimal Hypotheses principle

In previous works [194] we resorted to XSB-Prolog's `get_residual/2` predicate as a means to get the relevant rules for the query. According to the XSB Prolog's manual "the predicate `get_residual/2` unifies its first argument with a tabled subgoal and its second argument with the (possibly empty) delay list of that subgoal. The truth of the subgoal is taken to be conditional on the truth of the elements in the delay list." The delay list is the list of literals whose truth value could not be determined to be *true* nor *false* in the WFM, i.e., their truth value is *undefined* in the WFM of the program. It is possible to obtain the *residual* clause of a solution for a query literal, and in turn the *residual* clauses for the literals in its body, and so on. This way we can reconstruct the complete relevant residual part of the KB for the literal — we call this a *residual program* or *reduct* for that solution to the query. This way, not only do we get just the relevant part of the KB for the literal, we also get precisely the part of those rules bodies still *undefined*, i.e., that are involved in Loops Over Negation. The use of the `get_residual/2` predicate proved to be a useful strategy, but since not all MH models comply with the Well-Founded Model of the program (cf. Example 6.2 on page 55), we cannot use it for a MH semantics based

system implementation. Instead we must collect the relevant rules by directly accessing Prolog's `clause/2` ISO predicate. Alternatively, when a WAM-level implementation of the Layered Remainder is available in XSB by a corresponding new system predicate, e.g., `get_layered_remainder/2`, we could use it to construct the layered remainder part relevant for the query. We leave such task for future work.

A prototypical implementation of a MH-based query solver could resort to a Stable Models solver — e.g., Smodels [166] — to "solve" the SCCs of rules, according to the Minimal Hypotheses principle, to that effect needing to incorporate the additional functionality of minimal hypotheses assumption. The Smodels system [166] allows disjunctive rules, i.e., with disjunctions in the heads of rules (cf. Section 4.2), which can be added to a NLP. Such disjunctive rules could be used to allow the disjuncts to play the role of hypotheses to be assumed. Moreover, the disjuncts in Disjunctive Stable Models generated by Smodels are automatically minimized (cf. Example 4.12 of [233]) thereby reifying the *minimality* principle of assumed hypotheses. We can use XSB-Prolog's capabilities to accomplish items 1, 2, and 3 above; and surely a cleverly developed meta-interpreter in XSB could also be implemented to fulfil step 4, but using Smodels with its handling of Disjunctive rules could save a lot of work and debugging for this 4th step. Fortunately, XSB-Prolog has an inbuilt interface to Smodels which allows the programmer to accumulate rules in a clause-store that is sent to Smodels to solve.

The XASP interface [50, 52] (standing for XSB Answer Set Programming), is included in XSB-Prolog as a practical programming interface to Smodels [166], one of the best known implementations of the SMs over generalized LPs. The XASP system allows one not only to compute the models of a given NLP, but also to effectively combine 3-valued with 2-valued reasoning. Such integration permits to make use of relevance for queries. In SMs it is necessary to compute all complete models for the whole program. In the implementation framework we propose, we sidestep this issue by collecting and transforming the rules relevant for the query, and then by using XASP to send those rules to Smodels for computation of stable models of the relevant sub-program. The top-down computation, to boot, helps in partly or totally grounding the transformed relevant sub-program.

XSB's XASP communication with Smodels enables the programmer to use a "Smodels clause store" to which several rules can be added. After adding all the original residual relevant rules, and also the newly created disjunctive rules, we add two extra rules: 1) `userGoal :- Query`, where `Query` is the query conjunct posed by the user (and `userGoal` is a reserved atom); and 2) `:- not userGoal` which prevents Smodels from producing as an answer any model where the `userGoal` does not hold. This clause store is then sent to Smodels which will consider only those rules when computing the models — these are obtained by asking Smodels to compute one model (and on backtracking to compute

others, if we so wish). The SMs obtained are Minimal Hypotheses sub-models of the original program containing only the literals relevant for the query.

As we have seen before, with MH semantics loops/SCCs are seen as kinds of disjunctions. In this sense, a NLP might be transformable into a Disjunctive LPs where the disjuncts are the assumable hypotheses. With this approach, in order to guarantee sound and complete MH semantics based top-down querying implementation we still have to answer two questions:

1. Which atoms to be considered hypotheses and, therefore, to include in heads of disjunctive rules
2. Which literals should be in the bodies of the disjunctive rules

We do not address these questions for now, but leave them for future work along with the implementation of a fully functional MH-based KRR system.

11.3 Inspection Points

Although the Inspection Points theoretical construct is independent of the underlying semantics used, in order to actually implement and test it, we needed some solid abductive platform. We based our Inspection Points implementation on a formally defined, XSB-implemented, true and tried abduction system — ABDUAL [21]. ABDUAL lays the foundations for efficiently computing queries over ground three-valued abductive frameworks for extended logic programs with integrity constraints, on the well-founded semantics and its partial stable models. We are aware of the fact that, by complying with the well-founded semantics, the ABDUAL system is not fully suitable for inspecting side-effects of abductive logic programs under the MH semantics. Nonetheless, this first implementation of the IPs mechanism, simple as it is, stands on its own as a reification of a new reasoning mechanism complementary to the abduction one.

The syntax of ABDUAL programs is roughly the same as that of Extended Logic Programs, i.e., ABDUAL programs allow for explicitly negated literals to occur both in the heads and bodies of rules. Moreover, ABDUAL allows for declaration of *abducibles* via the reserved predicate *abds*/1, where its argument is a list of elements of the form *abducible_name*/*artity*. The reader can find the full details of ABDUAL in [21].

The query processing technique in ABDUAL relies on a mixture of program transformation and tabled evaluation. A transformation removes default negated literals (by

making them positive) from both the program and the integrity rules. Specifically, a dual transformation is used, that defines for each objective literal O [3] and its set of rules R, a dual set of rules whose conclusions *not* (O) are true if and only if O is false in R. Tabled evaluation of the resulting program turns out to be much simpler than for the original program, whenever abduction over negation is needed. At the same time, termination and complexity properties of tabled evaluation of extended programs are preserved by the transformation, when abduction is not needed. Regarding tabled evaluation, ABDUAL is in line with SLG [51, 102, 103, 212, 216, 229, 230, 231] evaluation, which computes queries to normal programs according to the well-founded semantics. To it, ABDUAL tabled evaluation adds mechanisms to handle abduction and deal with the dual programs.

ABDUAL is composed of two modules: the preprocessor which transforms the original program by adding its dual rules, plus specific abduction-enabling rules; and a meta-interpreter allowing for top-down abductive query solving. When solving a query, abducibles are dealt with by means of extra rules the preprocessor added to that effect. These rules just add the name of the abducible to an ongoing list of current abductions, unless the negation of the abducible was added before to the lists failing in order to ensure abduction consistency. The inspection point mechanism is implemented adroitly by means of the reserved predicate, '*inspect*/1' taking some literal L as argument, which engages the ABDUAL's abduction mechanism to try and discharge any meta-abductions performed under L by matching with the corresponding abducibles, adopted elsewhere outside any '*inspect*/1' call. The approach taken can easily be adopted by other abductive systems, as we had the occasion to check, e.g., with system [57]. We have also enacted an alternative implementation, relying on XSB-XASP and the declarative semantics transformation in 10.3.3, which is reported below.

Procedurally, in the ABDUAL implementation, the checking of an inspection point corresponds to performing a top-down query-proof for the inspected literal, but with the specific proviso of disabling new abductions during that proof. The proof for the inspected literal will succeed only if the abducibles needed for it were already adopted, or will be adopted, in the present ongoing solution search for the top query. Consequently, this check is performed after a solution for the query has been found. At inspection-point-top-down-proof-mode, whenever an abducible is encountered, instead of adopting it, we simply adopt the intention to *a posteriori* check if the abducible is part of the answer to the query (unless of course the negation of the abducible has already been adopted by then, allowing for immediate failure at that search node.) That is, one (meta-) abduces the checking of some abducible A, and the check consists in confirming that A is part of the abductive solution by matching it with the object of the check. According to our method, the side-effects of interest are explicitly indicated by the user by wrapping the corresponding goals subject to inspection mode, with the reserved construct '*inspect*/1'.

[3] An objective literal is either an atom or its explicit negation.

11.3.1 ABDUAL with Inspection Points

Inspection Points in ABDUAL function mainly by means of controlling the general abduction step, which involves very few changes, both in the pre-processor and the meta-interpreter. Whenever an '$inspect(X)$' literal is found in the body of a rule, where 'X' is a goal, a meta-abduction-specific counter — the '*inspect_counter*' — is increased by one, in order to keep track of the allowed character, active or passive, of performed abductions. The top-down evaluation of the query for 'X' then proceeds normally. Actual abductions are only allowed if the counter is set to zero, otherwise only meta-abductions are allowed. After finding an abductive solution for the query 'X' the counter is decreased by one. Backtracking over counter assignations is duly accounted for. Of course, this way of implementing the inspection points (with one '*inspect_counter*') presupposes the abductive query answering process is carried out "depth-first", guaranteeing the order of the literals in the bodies of rules actually corresponds to the order they are processed. We assume such a "depth-first" discipline in the implementation of inspection points, described in detail below. We lift this restriction at the end of the subsection.

11.3.1.1 Changes to the ABDUAL's pre-processor:

1. A new dynamic predicate was added: the '*inspect_counter*/1'. This is initialized to zero ('*inspect_counter*(0)') via an assert, before a top-level query is launched.

2. The original rules for the normal abduction step are now preceded by an additional condition checking that the '*inspect_counter*' is indeed set to zero.

3. Extra rules for the "inspection" abduction step are added, preceded by a condition checking the '*inspect_counter*' is set to greater than zero. When these rules are called, the corresponding abducible 'A' is not abduced as it would happen in the original rules; instead, '$consume(A)$' is abduced. This corresponds to the meta-abduction: we abduce the need to abduce 'A', the need to 'consume' the abduction of 'A', which is finally checked when derivation for the very top goal is finished.

The semantics of inspection points in ABDUAL is such that if a meta-abduction on 'X' (resulting from abducing '$consume(X)$') is not matched by an actual abduction on 'X' when we reach the end of solving the top query, the candidate abductive answer is considered invalid and the query solving fails. On backtracking, another alternative abductive solution (possibly with other meta-abductions) will be sought. The changes to the meta-interpreter include all the remaining processing needed to correctly implement inspection points, namely matching the meta-abduction of '$consume(X)$' against the abduction of 'X'.

11.3.1.2 Changes to the meta-interpreter:

1. Two 'quick-kill' rules for improved efficiency that detect and immediately solve trivial cases for meta-abduction:
   - When literal 'X' about to be meta-abduced ('$consume(X)$' about to be added to the abductions list) has actually been abduced already ('X' is in the abductions list) the meta-abduction succeeds immediately and '$consume(X)$' is not added to the abductions list;
   - When the situation in the previous point occurs, but with '$not\ X$' already abduced instead, the meta-abduction immediately fails.
2. Two new rules for the general case of meta-abduction, that now specifically treat the '$inspect(not\ X)$' and '$inspect(X)$' literals. In either rule, first we increase the '*inspect_counter*' mentioned before, then proceed with the usual meta-interpretation for '$not\ X$' ('X', respectively), and, when this evaluation succeeds, we then decrease '*inspect_counter*'.
3. After an abductive solution is found to the top query, check (impose) that every meta-abduction, i.e., every '$consume(X)$' literal abduced, is matched by a respective and consistent abduction, i.e., is matched by the abducible 'X' in the abductions list; otherwise the tentative solution found fails.

A counter — '*inspect_counter*' — is used instead of a toggle because several '$inspect(X)$' literals may appear at different graph-depth levels under each other, and reseting a toggle after solving a lower-level meta-abduction would allow actual abductions under the higher-level meta-abduction. An example clarifies this.

Example 11.2. ***Nested Inspection Points.*** *Consider again the program of the previous example 10.4, where the abducibles are* a, b, c, d*:*

$$\begin{array}{lcl@{\qquad\qquad}lcl} x & \leftarrow & a, inspect(y), b, c, not\ d & y & \leftarrow & inspect(not\ a) \\ z & \leftarrow & d & y & \leftarrow & b, inspect(not\ z), c \end{array}$$

When we want to find an abductive solution for the query $?-x$*, skipping over the low-level technical details, we proceed as follows:*

1. *The inspect_counter is initially set to* 0*, and the running abductions list is initialized to the* empty list [ ]*;*
2. *We pick a rule for* x *(in this case, there is only one) and try to find an abductive solution to its body '*$a, inspect(y), b, c, not\ d$*';*

3. *a is an abducible and since the 'inspect_counter' is still set initially to 0 we can abduce a by adding it to the running abductions list which now becomes* $[a]$*;*

4. *Next we need to find an abductive solution to 'inspect*(y)*' — since y is not an abducible we cannot use any 'quick kill' rule on it. We increase the 'inspect_counter' — which now takes the value* 1 *— and proceed to find an abductive solution for y;*

5. *Since the 'inspect_counter' is now different from 0, only meta-abductions are allowed;*

6. *Picking the first rule for y —* $'y \leftarrow inspect(not\ a)'$ *— we need to satisfy the 'inspect*$(not\ a)$*' in its body, but since we have already abduced a — the running abductions list currently '*$[a]$*' includes 'a' — a 'quick-kill' is applicable here: we already know that this 'inspect*$(not\ a)$*' will fail. The value of the 'inspect_counter' will remain 1;*

7. *On backtracking, we pick the second rule for y —* $'y \leftarrow b, inspect(not\ z), c'$ *— and now we need to find a solution to its body* $'b, inspect(not\ z), c'$*. b is an abducible and to satisfy it we could simply abduce b by adding it to the running abductions list, but since the inspect_counter is not* 0 *only meta-abductions are allowed, hence we meta-abduce b instead, by adding 'consume*(b)*' to the ongoing abductions list which now becomes* $[a, consume(b)]$*;*

8. *Proceeding to satisfy the rest of the body 'inspect*$(not\ z), c$*' we go on to 'inspect*$(not\ z)$*';*

9. *We increase 'inspect_counter' again, making it take the value 2, and continue on, searching for an abductive solution to not z;*

10. *The unique rule for* z *—* $z \leftarrow d$ *— imposes that the only solution for not z is by abducing not d, but since the 'inspect_counter' is greater than 0, we can only meta-abduce not d, and thus 'consume*$(not\ d)$*' is added to the running abductions list which now becomes* $[a, consume(b), consume(not\ d)]$*;*

11. *Now that we are done with the rule for z we return to y's rule: the meta-interpretation of 'inspect*$(not\ z)$*' succeeds (because we were able to meta-abduce not d) and so we decrease the 'inspect_counter' by one, it taking the value 1 again. Now we proceed and try to solve c;*

12. *c is an abducible, but since the inspect_counter is not zero we can only meta-abduce c by adding 'consume*(c)*' to the running abductions list which now becomes* $[a, consume(b), consume(not\ d), consume(c)]$*;*

13. *Now we are done with the rule for y and we thus return to x's rule: the meta-interpretation of 'inspect*(y)*' succeeded and so we decrease the 'inspect_counter' once more, and it now takes the value 0. From this point onwards regular abductions will take place instead of meta-abductions;*

14. *The rest of the body of the rule for x we were using is $b, c, not\ d$, and since all these literals correspond to abducibles we abduce b, c, and $not\ d$ by adding them to the abductions list which now becomes $[a, consume(b), consume(not\ d), consume(c), b, c, not\ d]$;*

15. *A tentative abductive solution is found to the initial query and it consists of the abductions in the running abductions list above;*

16. *The abductive solution is now checked for matches between meta-abductions and actual abductions. In this case, for every '$consume(A)$' in the abduction list there is an A also in the abduction list, i.e., every intention of abduction '$consume(A)$' is satisfied by the actual abduction of A — e.g, $consume(b)$ is matched by b, and $consume(not\ d)$ is matched by $not\ d$ in the abductive solution found. Because this final checking step succeeds, the whole answer is actually accepted. Note it is irrelevant which order a '$consume(A)$' and the corresponding A appear and were placed in the abductions list. The A in $consume(A)$ is just any abducible literal a or its default negation $not\ a$.*

In this example, we can see clearly that the *inspect* predicate can be used on any arbitrary literal (as in the case of $inspect(y)$), and not just on abducibles.

The correctness of this implementation against the declarative semantics provided before in Definition 10.2 can be sketched by noticing that whenever the *inspect_counter* is set to 0 the meta-interpreter performs actual abduction which corresponds to the use of the original program rules; whenever the *inspect_counter* is set to some value greater than 0 the meta-interpreter just abduces $consume(A)$ (where A is the abducible being checked for its abduction being produced elsewhere), and this corresponds to the use of the program transformation rules for the *inspect* predicate.

11.3.1.3 More general query solving

In case the "depth-first" discipline is not followed, either because goal delaying is taking place, or multi-threading, or co-routining, or any other form of parallelism is being exploited, then each queried literal will need to carry its own list of ancestors with their individual '*inspect_counters*'. This is necessary so as to have a means, in each literal, to know which and how many *inspect*s there are between the root node and the currently being processed literal, and which *inspect_counter* to update; otherwise there would be no way to know if abductions or meta-abductions should be performed. We did not implement this more general method but only outline it here and envisage it for future work, viz. 12.3.

11.3.2 Alternative Implementation Method for Inspection Points

The method presented here is an avenue for implementing the inspection points mechanism through the simple syntactic program transformation of Definition 10.2 which can be readily used by any logic programming system that implements some Layer-Decomposable semantics. Our ultimate goal is to develop a Minimal Hypotheses semantics based abductive reasoning system with inspection points, but since the time requirements for building such a system would surpass our time constraints for writing this thesis, we considered, for now, an SM based implementation. Since the Stable Models semantics is a particular case of a Layer-Decomposable semantics and there exist already several Stable Models systems, like SModels [166], DLV [59], or Clasp [108], we can use this method to implement an abductive SM-based inspection mechanism. First, we need a way to implement abduction in an SM based system. One can model SM-based abducibles of some program P by transforming it into a P' where P' is obtained from P by adding an even loop over negation for each abducible (like the one depicted in Section 10.1), by means of a newly introduced reserved atom for that purpose. Secondly, in order to obtain the inspection mechanism we just need to further transform P' into a CNLP without inspection points. The program transformation we presented for the declarative semantics of the inspection points (Definition 10.2) achieves both goals. Finally, we just need an implementation of SMs, like SModels, DLV, Clasp, or any other. This way, under the SM semantics, a program may have models where some *abducible* is *true* and another where it is *false*, i.e. the corresponding reserved atom *neg_abducible* is *true*. If there are n abducibles in the program, there will be 2^n models corresponding to all the possible combinations of *true* and *false* for each. Under the WFS without a specific abduction mechanism, e.g. the one available in ABDUAL, both *abducible* and *neg_abducible* remain *undefined* in the Well-Founded Model (WFM), but may hold (as alternatives) in some Partial Stable Models.

When finding an abductive solution to a set of observations, a 2-valued abduction step can be viewed as committing oneself to the just found possible abductive hypotheses — e.g., assuming *abducibles* as *true* or as *false*. Using the SM semantics abduction is done by guessing the truth-value of each abducible and providing the whole model and testing it for stability; whereas using the MH semantics, which enjoys the several Relevance properties, added now with abduction, the latter can be performed *by need*, induced by the top-down query solving procedure, solely for the relevant abducibles — i.e., irrelevant abducibles are left unconsidered. Thus, top-down abductive query answering is a means of finding those abducible values one might commit to in order to satisfy a query. In particular, this is one of the main problems which abduction over stable models has been facing, in that it always has to consider all the abducibles in a program and then progressively dismiss all those that are irrelevant for the problem at hand. This is not so in our system framework, since we can usually begin evaluation by a top-down derivation of a query,

which immediately constrains the set of abducibles that are relevant to the satisfaction and proof of that particular query.

An important consideration when computing consequences of abductive solutions, is that we could end up having to compute the models of the whole program in order to obtain just a particular relevant subset that will be used to enact on it *a posteriori* preferences. This can be easily avoided by performing preliminary computation of the relevant sub-program, given the consequences that we expect to observe. This means that the consequences believed significant for model preference can be computed on the XSB side, and their additional relevant sub-program sent to Smodels as well. For a MH semantics based implementation of Inspection Points using XSB Prolog we would need, as said before, a `get_layered_remainder/2` predicate to compute the Layered relevant sub-program. In this phase, we do not allow for additional abduction of literals, but merely enforce that rules for consequences are consumers of considered abducibles which have already been produced. In this way, we combine a declarative methodology to describe the abductive process, with an efficient and viable implementation of reasoning by complementing a 3-valued well-founded derivation with the computation of the stable models of the relevant sub-program, in a natural way to obtain all the possible 2-valued models from the well-founded one. Using XSB-XASP, the process would be the same as for using an SMs implementation alone, but instead of sending the whole transformed program to the SMs engine, only the sub-program relevant for the query at hand would be sent. This way, abductive reasoning can benefit from the relevance property enjoyed by the Well-Founded Semantics implemented by the XSB-Prolog's SLG-WAM.

Given the top-down proof procedure for abduction, implementing inspection points for program P becomes just a matter of adapting the evaluation of derivation subtrees falling under '$inspect/1$' literals, at meta-interpreter level, subsequent to performing the transformation $\Pi(P)$ presented before, which defines the declarative semantics. Basically, any considered abducibles evaluated under '$inspect/1$' subtrees, say A, are codified as '$abduced(A)$', where:

$$\begin{aligned} abduced(A) &\leftarrow not\ abduced\_not(A) \\ abduced\_not(A) &\leftarrow not\ abduced(A) \end{aligned}$$

All $abduced/1$ literals collected during computation of the relevant sub-program are later checked against the stable models themselves. Every '$abduced(a)$' must pair with a corresponding abducible a for the model to be accepted. Thus, we combine and complement the best of both worlds, the 2- and the 3-valued ones.

In the next and final Chapter we summarize the contributions in this thesis, make a brief overview of some applications the Minimal Hypotheses semantics can be useful for, and point to future work directions.

12 . Conclusions, Applications and Future Work

"Ex terminus novus orsa"
("*From ends new beginnings*")

LATIN SAYING

We summarize here the present thesis, underscore the most important subjects and issues, highlight our novel contributions, overview some of the applications where our results have been used in the past few years, and point to future work directions. The applications overviewed here are discussed in [182, 190, 192, 195].

12.1 Conclusions

One of the goals of the general Artificial Intelligence field is to automate logic based Knowledge Representation and Reasoning by means of computers. This aim has been growing in importance, e.g. to provide a solid foundation for Semantic Web applications. We addressed both Knowledge Representation and Reasoning issues in the context of logic programming, adopting a novel approach both to the syntactic structure of a program and to its semantics.

The standard syntactic notion of (acyclic) stratification is applied to atoms of a program, and not all programs are stratifiable. With the intention of modelling relations between Knowledge Bases we considered a generic (possibly cyclic) graph perspective, with KBs at the vertices and the edges representing dependency relations. For such graphs we defined, instead of stratification, the general notion of Layering, which attaches an ordering to the Strongly Connected Components of the graph and to the remaining vertices not part of any SCC. The generic graph notion of Layering is not unlike, but not the same as, a topological sorting.

Casting this new notion to the specific case of logic programs, we devised two syntactic notions, dubbed Rule Layering and Atom Layering, which are applied, respectively, to rules and to atoms of the program — the latter being more general than the stratification and depending on the former, the more fundamental one. These layerings fully capture all the syntactic information of the program and thus provide a sound basis for the definition of a semantics. Moreover, and from the practical implementations perspective, the syntactical information of the layerings can itself be useful in developing modular and efficient reasoning engines, as they provide an easy and natural means of partitioning and composing the knowledge represented in a logic program. Extended Logic Programs and Disjunctive Logic Programs can be syntactically transformed into Normal Logic Programs, and by showing this much, we can restrict the job of building a semantics for logic programs only to NLPs without loss of generality concerning ELPs and DisjLPs[1].

We analysed some of the fundamental principles behind the construction of a semantics including the notion of support. We realized the classical semantic notion of support is not fully compatible with the new layering generalization of stratification. This lead us to the define the new notion of layered support, a generalization of the classical support that is in harmony with the layerings.

We learned, from reviewing the currently most used 2- and 3-valued semantics for NLPs, some of the features that make each of them interesting, useful, and intuitive, but also their drawbacks and contexts where they cannot be applied with as much success. From this encompassing glance we summarized, both from an intuitive and formal approaches, the desirable properties of 2-valued semantics for NLPs: guarantee of model existence, relevance, cumulativity, exhibiting a model conservative generalization of the stable models (i.e., all SMs being also models of the new semantics). Model existence guarantee is especially important to lend robustness to the whole system, thereby enabling arbitrary KB merging and/or updating. Relevance allows the construction of top-down query-solving proof-procedures *à la*, Prolog sound and complete according to the semantics, and is thus quite convenient especially when doing query-answering in very large KBs which possibly contain knowledge about sparsely connected domains. We leave a construction for such a top-down proof-procedure for future work. Cumulativity allows tabling/lemma storing techniques to be used to speed-up computations, and one knows that there is no such thing as "too much efficiency".

Based on the Layerings we defined a family of semantics, the Layer-Decomposable Semantics family, whose members ensure their models "respect" the Layerings. We argue that every "good" semantics for logic programs should be a member of this family. Naturally, not all semantics that are members of this family enjoy the same properties —

[1]We need impose no restrictions on the use of the shifting rule for disjunctions when we can assign semantics to any loops.

e.g., the Stable Models semantics is a member of the Layer-Decomposable Semantics and it does not enjoy the same properties as, say, the Minimal Hypotheses semantics.

There have been in the literature approaches to semantics that consider default negated literals as assumable hypotheses. Under this perspective, the truth values of other literals become determined by those actually assumed negative hypotheses. This approach is usually associated with requiring maximality of negative hypotheses safeguarding overall consistency and the results stemming from it correspond to Stable Models in the 2-valued case, and to the Well-Founded Semantics in the 3-valued one. With the intent of coming up with a 2-valued semantics for NLPs with the useful properties we identified before, we took the hypotheses assumption approach one step further to allow for positive hypotheses too. Soon enough we found out there is no need for assuming both positive and negative hypotheses to define a model for an NLP according to some semantics: assuming positive hypotheses is enough *per se* to guarantee model existence (one of the properties for a semantics we consider highly desirable) and it generalizes negative hypotheses assumption. The dual version of the maximal negative hypotheses assumption principle is minimal positive hypotheses assumption, and it is based on this new strategy that we defined the Minimal Hypotheses semantics. We show that MH semantics is a member of the Layer-Decomposable Semantics family, and that it enjoys all the properties we listed as desirable. MH semantics is a model conservative extension of the SM semantics, and as a secondary gain of this thesis's work we show the Stable Models semantics is also a member if the Layer-Decomposable Semantics family. Furthermore, since MH semantics guarantees model existence for every NLP, it successfully separates the knowledge representation role of NLP rules from the integrity constraint role of rules with $\bot$ as head that are typical of Constrained NLPs, as explained in 6.5.1. This way, MH semantics not only guarantees that all stable model solutions to a search problem are preserved as MH models, but also that only ICs are allowed to prune candidate solutions[2]. Also, by growing from a hypotheses assumption stance, the MH semantics seamlessly encompasses abductive reasoning.

Throughout these last few years we have defined several tentative semantics, always with the intent to come up with one enjoying the properties we identified as desirable. These semantics have worked as stepping stones that finally lead us to the Minimal Hypotheses semantics. To accomplish that, one key feature of such a semantics would have to be guarantee of model existence for every NLP. This immediately raises the issue that inevitably, some of the models of the new semantics would lack classical support. This is also, and unavoidably so, the case of some MH models. In fact, all MH models that are not also SMs lack classical support, and this has been one of the main criticism for the semantics we have defined, including the Minimal Hypotheses semantics. It is, however,

[2]We have reused the SM programs in the literature that use odd loops over default negation to implement ICs, and ascertained that they can all be readily rewritten as ICs instead.

an inescapable feature if one wants to guarantee model existence, amongst other useful properties. On the other hand, all MH models (and, naturally, all SMs too) enjoy the new generalized layered support which, we believe and propose, should be the standard notion of support. Nevertheless, any MH model which is also a SM exhibits classical support. On the other hand, the extension of support beyond the classical one is a feature arising from the possibility of breaking non-well-founded negation (odd loops over negation)by means of "self-supporting" minimal positive hypotheses.

For query answering, the MH semantics provides mainly three advantages over the SMs: 1) by enjoying Relevance (and Brave Relevance) top-down query-solving is possible, thereby circumventing whole model computation (and grounding) which is unavoidable with SMs; 2) by considering only the relevant sub-part of the program when answering a query it is possible to enact grounding of only those rules, if grounding is really desired, whereas with SM semantics whole program grounding is, once again, inevitable — grounding is known to be a major source of computational time consumption; MH semantics, by enjoying Relevance, permits curbing this task to the minimum sufficient to answer a query; 3) by enjoying Cumulativity (and Brave Cautious Monotony), as soon as the truth-value of a literal is determined in a branch for the top query it can be stored in a table and its value used to speed up the computations of other branches within the same top query.

Goal-driven abductive reasoning is elegantly modelled by top-down abductive-query-solving and by taking a hypotheses assumption approach and, by enjoying Relevance, MH semantics caters nicely for this convenient problem representation and reasoning category. Abductive logic programming is also used to model planning problems with abducibles coding actions. In this context, inspecting for side-effects of abduced actions can be quite useful in evaluating alternative abductive scenarios. We introduced a new reasoning mechanism corresponding to this side-effect inspection, dubbing it Inspection Points, which avoids having to produce whole models for examining just the side-effects of interest, and also avoids producing irrelevant abductions.

This thesis focuses mainly on establishing new theoretical foundational principles for logic program based knowledge representation and reasoning. Notwithstanding, we took some proof-of-concept implementation steps with which we learned valuable lessons for the future building of a Minimal Hypotheses based Knowledge Representation and Reasoning abductive system with Inspection Points.

12.2 Applications

Many applications have been developed using the Stable Model/Answer-set semantics as the underlying theoretical platform. These generally tend to be focused on solving problems that require complete knowledge, such as search problems where all the knowledge represented is relevant to the solutions. However, as Knowledge Bases increase in size and complexity, and as merging and updating of KBs becomes more and more common, e.g. for Semantic Web applications [115], partial knowledge problem solving importance grows, as well as the need to ensure overall consistency of the merged/updated KBs.

During these last years that we have been developing different approaches to semantics of NLPs, one of the questions we heard most frequently was "What is the application for your semantics?". We want to answer here clearly to that question: the Minimal Hypotheses semantics is intended to, and can be used in *all* the applications where the Stable Models/Answer-Sets semantics are used to model KRR and search problems, *plus* all applications where query answering (both under a credulous mode of reasoning as well as under a skeptical one) is intended, *plus* all applications where abductive reasoning is needed. The MH semantics presumes in its aims to be a sound theoretical platform for 2-valued (possibly abductive) reasoning with logic programs.

Although we have not yet implemented a fully functional and integrated MH-based KRR system, we have already done prototypical implementations of some components as described in Chapter 11. Some of these components have already been integrated into other systems. For example, in Section 11.2 we described our XSB-Prolog/XASP/Smodels interaction mechanism which we use to collect relevant rules for a query (on the XSB-Prolog side), and then transform and send them to Smodels via XSB's XASP interface for relevant sub-model computation. This mechanism was embedded at the core of the application system ACORDA [152, 153, 203, 178, 179, 184, 185, 199, 200, 201, 202] used to build prospective logic agents. Also, the inspection points mechanism, formally defined in Section 10.3, and some of its implementation alternatives in Section 11.3, has been adopted in [181, 195] and also as a way to implement *a posteriori* preferences for filtering models, e.g, subsections 1.2.1.3 "*Conditional Abduction*", and 1.3.1 "*Constraining Abduction*" of [182][3].

Yet another application scenario that can benefit from our current work is collaborative learning. In [190] we address several issues that could benefit from MH semantics:

1. Inductive concept learning in a 3-valued setting where we learn a definition for both the target concept and its opposite, considering positive and negative examples as

[3]Subsection 1.3.4 "*Modelling Inspection Points*" of [182] makes explicit reference to our work.

instances of two disjoint classes. Explicit negation is used to represent the opposite concept, while default negation is used to ensure consistency and to handle exceptions to general rules. Under this 3-valued setting we resort to the WFSX (Well-Founded Semantics with eXplicit negation, cf. [7, 15]); but if the setting were to be a 2-valued one, and needed to guarantee model existence, then one could benefit from Minimal Hypotheses semantics.

2. Collaborative KB construction: after obtaining the knowledge resulting from a learning process, an agent can then interact with the environment by perceiving it and acting upon it. One single agent exploring an environment may gather only so much information about it and that may not suffice to solve some problem. In such case, a collaborative multi-agent strategy, where each agent explores a part of the environment and shares with the others its findings, might provide better results. A semantics that ensures overall model existence, like the MH semantics, would lend robustness to such a collaboratively built Knowledge Base.

12.3 Future Work

Much work still remains to be done that can be rooted in this platform contribution.

In Chapter 2 we assumed the edges of a graph with KBs for vertices as the fundamental means to identify the least layering of the graph, but we also mentioned the possibility of having the user specifying his own ordering preferences thereby gaining explicit control over the layering. We consider this for future work, with its relation to other works about preferences, and also to Multi-Dimensional updates [138].

The (non-preference based) least layerings are a syntactical notion. When considering a static KB (with no updates nor merges) one can contemplate the practical usefulness of compiling the layers of a program into an efficient representation, perhaps resorting to lower abstraction level programming languages like C/C++. A particular possible road of development would include implementing the Layered Negative reduction and the Layered Remainder in XSB-Prolog's WAM. This could be done by taking advantage of the efforts already made to implement the Loop detection operation, in the context of the Answer Completion implementation, since both operations resort to the identification of Strongly Connected Components. Although one of the aims of MH semantics is to be usable for top-down query answering, it can also be used for modelling search problems where the answers are models for the whole program. In this case those low-level representations can then be used for improving efficiency of whole model computation.

The general topics of using logic programs for Belief Revision([18, 77, 135]), Updates

([13, 12, 16, 30, 138, 224, 239, 247, 248]), Preferences ([79, 80, 180]), etc., are *per se* orthogonal to the semantics issue, and therefore, all these subjects can now be addressed with Minimal Hypotheses semantics as the underlying platform, and some of them might even benefit from the Inspection Points mechanism, e.g., for enacting some kinds of preferences (cf. [182]). Importantly, MH can guarantee the liveness of updated and self-updating LP programs such as those of EVOLP and related applications [10, 8, 137, 178, 179, 184, 190].

The Minimal Hypotheses semantics still has not been thoroughly compared with Revised Stable Models [187], PStable Models [170], and other related semantics.

In 11.2 we mention the possibility of an implementation of the Minimal Hypotheses semantics resorting to Stable Models and Disjunctive Logic Programs. This can only be the case if there is some syntactic transformation of an NLP P into another P' where the MH models of P exactly coincide with the disjunctive SMs of P'. We consider this line of research for future work as it may help in providing an alternative way to explain the intuition of the MH semantics to the logic programming community.

As mentioned before (in 11.3.1.3), a more general and flexible implementation of the Inspection Points mechanism is needed, namely for allowing parallel processing of branches of the same query. Associating the list of ancestors to each node in the query tree might be a possible way to facilitate the parallelization.

A future application of Inspection Points is planning in a multi-agent setting. An agent may have abduced a plan (a sequence of abducible individual actions) and, in the course of carrying out its abduced actions, it may find that another agent has undone some of its already executed actions. So, before executing an action, the agent should check all necessary preconditions hold. Note it should only *check*, thereby avoiding abducing again a plan for them: this way, if the preconditions hold the agent can continue and execute the planned action. The agent should only take measures to enforce the preconditions again whenever the check fails. Clearly, an *inspect*ion of the preconditions is what we need here. On the other hand, the "other" agent can actually be cooperating with our agent; and when our agent goes on to execute some planned action, it might find that the companion agent has already done some of the work for him. In this case, our agent also wants only to check, not perform any actions in order to ensure the conditions. Merely checking if an intermediate goal has already been achieved — because some other agent helped, for example — is what we need to take advantage of this collaborative scenario. Again, an *inspect*ion is the way to implement such check.

More generally, inspection points afford us with the ability to avoid having to generate complete abductive models in order to glean the consequences of interest of abductive solutions. The developed techniques can be employed too for permitting passive ICs, which are not allowed to actively abduce but only to verify their satisfaction with regard

to given abductions, in contrast to active ICs that can further abduce in order to be satisfied. Plus, of course, to enable ICs which contain a combination of both active and passive literals.

Another future use concerns the computation of inspected consequences of partially defined 2-valued models, obtained by top-down querying of NLPs, wherein the abducibles are the default nots themselves, plus appropriate ICs to enforce consistency. Once again, the computation of complete models can thus be avoided. A 2-valued semantics which enjoys relevance must then be used, or otherwise a guarantee that the NLP is stratified or does not contain non-well-founded negation.

In summary, we have provided a fresh platform on which to re-examine some semantic issues present in Logic Programming and its uses, which purports to provide a natural continuation and improvement of LP development.

Bibliography

[1] Encyclopedia britannica — http://www.britannica.com/.

[2] Internet encyclopedia of philosophy — http://www.iep.utm.edu/.

[3] *Proceedings of the Eighth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, March 29-31, 1989, Philadelphia, Pennsylvania*. ACM Press, 1989.

[4] F. Toni A. Kakas, R. Kowalski. *The role of abduction in logic programming*. Oxford, 1995.

[5] Marco Alberti, Federico Chesani, Marco Gavanelli, Evelina Lamma, Paola Mello, and Paolo Torroni. Verifiable agent interaction in abductive logic programming: The sciff framework. *ACM Trans. Comput. Logic*, 9(4):1–43, 2008.

[6] João Alcântara, Carlos Viegas Damásio, and Luís Moniz Pereira. Paraconsistent logic programs. In In S. Flesca and G. Ianni, editors, *Proceedings of the 8th European Conference of Logics In Artificial Intelligence, JELIA 2002*, volume 2424, pages 345–356. Springer, September 2002.

[7] José Júlio Alferes. *Semantics of Logic Programs with Explicit Negation*. PhD thesis, Universidade Noval de Lisboa, October 1993.

[8] José Júlio Alferes, Antonio Brogi, João Alexandre Leite, and Luís Moniz Pereira. Evolving logic programs. In S. Flesca, S. Greco, N. Leone, and G. Ianni, editors, *Procs. of the 8th European Conf. on Logics in Artificial Intelligence (JELIA'02)*, number 2424 in LNCS, pages 50–61. Springer, September 2002.

[9] José Júlio Alferes, Carlos Viegas Damásio, and Luís Moniz Pereira. Slx - a top-down derivation procedure for programs with explicit negation. In *SLP*, pages 424–438, 1994.

[10] José Júlio Alferes, M. Eckert, and Wolfgang May. *Semantic Techniques for the Web, The REWERSE Perspective*, volume 5500 of *LNCS*, chapter Evolution and Reactivity in the Semantic Web, pages 161–200. Springer Verlag, 2009.

[11] José Júlio Alferes, Heinrich Herre, and Luís Moniz Pereira. Partial models of extended generalized logic programs. In Lloyd et al. [150], pages 149–163.

[12] José Júlio Alferes, João Alexandre Leite, Luís Moniz Pereira, H. Przymusinska, and T. C. Przymusinski. Dynamic updates of non-monotonic knowledge bases. *The Journal of Logic Programming*, 45(1–3):43–70, September/October 2000.

[13] José Júlio Alferes, João Alexandre Leite, Luís Moniz Pereira, and Paulo Quaresma. Planning as abductive updating. In D. Kitchin, editor, *Procs. of the AISB'00 Symposium on AI Planning and Intelligent Agents*, pages 1–8. AISB, 2000.

[14] José Júlio Alferes and Luís Moniz Pereira. An argumentation theoretic semantics based on non-refutable falsity. In Dix et al. [88], pages 3–22.

[15] José Júlio Alferes and Luís Moniz Pereira. *Reasoning with Logic Programming*, volume 1111 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 1996.

[16] José Júlio Alferes and Luís Moniz Pereira. Update-programs can update programs. In *Non-Monotonic Extensions of Logic Programming*, pages 110–131, 1996.

[17] José Júlio Alferes, Luís Moniz Pereira, H. Przymusinska, and T. C. Przymusinski. Lups - a language for udating logic programs. *Artificial Intelligence*, 138(1–2), 2002.

[18] José Júlio Alferes, Luís Moniz Pereira, and Teodor C. Przymusinski. Belief revision in non-monotonic reasoning and logic programming. *Fundam. Inform.*, 28(1-2):1–22, 1996.

[19] José Júlio Alferes, Luís Moniz Pereira, and Teodor C. Przymusinski. 'classical' negation in nonmonotonic reasoning and logic programming. *J. Autom. Reasoning*, 20(1):107–142, 1998.

[20] José Júlio Alferes, Luís Moniz Pereira, and Terrance Swift. Well-founded abduction via tabled dual programs. In *International Conference on Logic Programming*, pages 426–440, 1999.

[21] José Júlio Alferes, Luís Moniz Pereira, and Terrance Swift. Abduction in well-founded semantics and generalized stable models via tabled dual programs. *Theory and Practice of Logic Programming*, 4(4):383–428, July 2004.

[22] P. W. Anderson. More is different. *Science*, 177(4047):393–396, August 1972.

[23] Christian Anger, Kathrin Konczak, and Thomas Linke. Nomore: Non-monotonic reasoning with logic programs. In *Proceedings of the Eighth European Conference on Logics in Artificial Intelligence (JELIA'02*, pages 521–524. Springer, 2002.

[24] Krzysztof R. Apt and Howard A. Blair. Arithmetic classification of perfect models of stratified programs. *Fundam. Inform.*, 14(3):339–343, 1991.

[25] Krzysztof R. Apt, Howard A. Blair, and Adrian Walker. Towards a theory of declarative knowledge. In *Foundations of Deductive Databases and Logic Programming.*, pages 89–148. Morgan Kaufmann, 1988.

[26] Krzysztof R. Apt and M. H. van Emden. Contributions to the theory of logic programming. *J. ACM*, 29:841–862, July 1982.

[27] Francisco Azevedo. *Constraint Solving over Multi-valued Logics - Application to Digital Circuits*, volume 91 of *Frontiers of Artificial Intelligence and Applications*. IOS Press, 2003.

[28] Franz Baader, Sebastian Brandt, and Carsten Lutz. Pushing the el envelope. In Kaelbling and Saffiotti [127], pages 364–369.

[29] Franz Baader, Diego Calvanese, Deborah L. Mcguinness, Daniele Nardi, and Peter F. Patel-Schneider. *The description logic handbook: theory, implementation, and applications*. Cambridge University Press, New York, NY, USA, 2003.

[30] Federico Banti, José Júlio Alferes, and Antonio Brogi. Well founded semantics for logic program updates. In J. A. González C. Lemaître, C. A. Reyes, editor, *Advances in Artificial Intelligence - IBERAMIA 2004, 9th Ibero-American Conference on AI*, volume 3315 of *Lecture Notes in Computer Science*, pages 397–407. Springer-Verlag, 2004.

[31] Chitta Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 2003.

[32] Chitta Baral and Michael Gelfond. Logic programming and knowledge representation. *J. Log. Program.*, 19/20:73–148, 1994.

[33] Chitta Baral, Michael Gelfond, and J. Nelson Rushton. Probabilistic reasoning with answer sets. *TPLP*, 9(1):57–144, 2009.

[34] Chitta Baral, Sarit Kraus, Jack Minker, and V. S. Subrahmanian. Combining knowledge bases consisting of first order theories. In *ISMIS '91: Proceedings of the 6th International Symposium on Methodologies for Intelligent Systems*, pages 92–101, London, UK, 1991. Springer-Verlag.

[35] Chitta Baral and V. S. Subrahmanian. Stable and extension class theory for logic programs and default logics - technical report cs-tr-2402. Technical report, University of Maryland, 1990.

[36] Chitta Baral and V. S. Subrahmanian. Stable and extension class theory for logic programs and default logics. *Journal of Automated Reasoning*, 8:345–366, 1992.

[37] Chitta Baral and V. S. Subrahmanian. Dualities between alternative semantics for logic programming and nonmonotonic reasoning. *J. Autom. Reasoning*, 10(3):399–420, 1993.

[38] Rachel Ben-eliyahu and Rina Dechter. On computing minimal models. *Annals of Mathematics and Artificial Intelligence*, 18:2–8, 1993.

[39] Claude Berge. *Graphes et hypergraphes*. Dunod, Paris, 1970.

[40] Bharat K. Bhargava, Timothy W. Finin, and Yelena Yesha, editors. *CIKM 93, Proceedings of the Second International Conference on Information and Knowledge Management, Washington, DC, USA, November 1-5, 1993*. ACM, 1993.

[41] Howard A. Blair, Wiktor Marek, and John S. Schlipf. The expressiveness of locally stratified programs. *Annals of Mathematics and Artificial Intelligence*, 15:209–229, 1995.

[42] Andrei Bondarenko, Phan Minh Dung, Robert A. Kowalski, and Francesca Toni. An abstract, argumentation-theoretic approach to default reasoning. *Artif. Intell.*, 93:63–101, 1997.

[43] Stefan Brass and Jürgen Dix. Characterizations of the disjunctive well-founded semantics: Confluent calculi and iterated gcwa. In *Journal of Automated Reasoning*, pages 268–283. Springer, LNCS, 1997.

[44] Stefan Brass and Jürgen Dix. Semantics of (disjunctive) logic programs based on partial evaluation. *J. Log. Program.*, 40(1):1–46, 1999.

[45] Stefan Brass, Burkhard Freitag, and Ulrich Zukowski. Transformation-based bottom-up computation of the well-founded model. Technical Report MIP-9620, Universität Passau, 2001.

[46] Gerhard Brewka and Jérôme Lang, editors. *Principles of Knowledge Representation and Reasoning: Proceedings of the Eleventh International Conference, KR 2008, Sydney, Australia, September 16-19, 2008*. AAAI Press, 2008.

[47] G. Brignoli, S. Costantini, O. D'Antona, and A.Provetti. Characterizing and computing stable models of logic programs: the non-stratified case. In C. Baral and H. Mohanty, editors, *Proceedings of the 1999 Conference on Information Technology*. AAAI Press — 2000, Decemebr 1999.

[48] Marco Cadoli. The complexity of model checking for circumscriptive formulae. *Inf. Process. Lett.*, 44(3):113–118, 1992.

[49] Marco Cadoli and Andrea Schaerf. Compiling problem specifications into sat. *Artificial Intelligence*, 162:89–120, 2005.

[50] L. Castro, T. Swift, and D. S. Warren. *XASP: Answer Set Programming with XSB and Smodels*. `http://xsb.sourceforge.net/packages/xasp.pdf`.

[51] L.F. Castro, T. Swift, and D.S. Warren. Suspending and resuming computations in engines for SLG evaluation. In *Practical Applications of Declarative Languages*, volume 2257 of *LNCS*, pages 332–346. Springer-Verlag, 2002.

[52] L.F. Castro and D.S. Warren. An environment for the exploration of non monotonic logic programs. In A. Kusalik, editor, *Proc. of the 11th Intl. Workshop on Logic Programming Environments (WLPE'01)*, 2001.

[53] Philip K. Chan and Salvatore J. Stolfo. Experiments on multi-strategy learning by meta-learning. In Bhargava et al. [40], pages 314–323.

[54] Philip K. Chan and Salvatore J. Stolfo. Toward multi-strategy parallel & distributed learning in sequence analysis. In Hunter et al. [125], pages 65–73.

[55] Kaustuv Chaudhuri, Frank Pfenning, and Greg Price. A logical characterization of forward and backward chaining in the inverse. In *In Uli Furbach and Natrajan Shankar, editors, Proceedings of the 3rd International Joint Conference on Automated Reasoning (IJCAR'06*, pages 97–111. Springer, 2006.

[56] W. Chen and D. S. Warren. Tabled Evaluation with Delaying for General Logic Programs. *Journal of the ACM*, 43(1):20–74, January 1996.

[57] Henning Christiansen and Verónica Dahl. Hyprolog: A new logic programming language with assumptions and abduction. In Gabbrielli and Gupta [106], pages 159–173.

[58] Alonzo Church. *Introduction to Mathematical Logic*. Princeton, Princeton University Press, 1956.

[59] 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 *Workshop Logische Programmierung*, 1997.

[60] K. Clark. Negation as failure. In H.Gallaire and J.Minker, editors, *Logic and Databases*, pages 293–322. Plenum Press, 1978.

[61] Luca Console, Daniele Theseider Dupre, and Pietro Torasso. On the relationship between abduction and deduction. *Journal of Logic and Computation*, 1(5):661–690, 1991.

[62] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Introduction to algorithms, second edition, 2001.

[63] Agostino Cortesi and Gilberto Filé. Graph properties for normal logic programs. *Theor. Comput. Sci.*, 107(2):277–303, 1993.

[64] Stefania Costantini. Contributions to the stable models semantics of logic programs with negation. *Theoretical Computer Science*, 149(2):231–255, 1995.

[65] Stefania Costantini. Comparing different graph representations of logic programs under the answer set semantics. In Provetti and Son [205].

[66] Stefania Costantini. Component-based answer set programming. In Osorio and Provetti [171].

[67] Stefania Costantini. On the existence of stable models of non-stratified logic programs. *TPLP*, 6(1-2):169–212, 2006.

[68] Stefania Costantini, Gaetano Aurelio Lanzarone, and Giuseppe Magliocco. Asserting lemmas in the stable model semantics. In *JICSLP*, pages 438–452, 1996.

[69] Carlos Viegas Damásio, Wolfgang Nejdl, and Luís Moniz Pereira. Revise: An extended logic programming system for revising knowledge bases. In *Knowledge Representation and Reasoning*. Morgan Kaufmann, 1994.

[70] Carlos Viegas Damásio and Luís Moniz Pereira. Abduction over 3-valued extended logic programs. In Marek and Nerode [155], pages 29–42.

[71] Carlos Viegas Damásio and Luís Moniz Pereira. Default negated conclusions: Why not? In Dyckhoff et al. [92], pages 103–117.

[72] Carlos Viegas Damásio and Luís Moniz Pereira. A paraconsistent semantics detecting contradiction support. In J. Dix, U. Furbach, and A. Nerode, editors, *Logic Programming and NonMonotonic Reasoning, 4th Int. Conf.*, number 1265 in LNAI, pages 224–243. Springer, July 1997.

[73] Carlos Viegas Damásio and Luís Moniz Pereira. A survey of paraconsistent semantics for logic programs. In D.M. Gabbay and Ph. Smets, editors, *Handbook of Defeasible Reasoning and Uncertainty Management Systems*, volume 2, pages 241–320. Kluwer Academic Publishers, 1998.

[74] Carlos Viegas Damásio, Luís Moniz Pereira, and Terrance Swift. Coherent well-founded annotated logic programs. In Gelfond et al. [112], pages 262–276.

[75] Terrance Deacon. *Evolution and Learning: The Baldwin Effect Reconsidered*, chapter The Hierarchic Logic of Emergence: Untangling the Interdependence of Evolution and Self-Organization. MIT Press, 2003.

[76] James P. Delgrande and Torsten Schaub. Consistency-based approaches to merging knowledge bases: Preliminary report. In *Leite (Eds.), Proceedings of the Ninth European Conference on Logics in Artificial Intelligence (JELIA'04), Lecture Notes in Artificial Intelligence*, pages 126–133, 2004.

[77] James P. Delgrande, Torsten Schaub, Hans Tompits, and Stefan Woltran. Belief revision of logic programs under answer set semantics. In Brewka and Lang [46], pages 411–421.

[78] James P. Delgrande, Torsten Schaub, Hans Tompits, and Stefan Woltran. Merging logic programs under answer set semantics. In Hill and Warren [118], pages 160–174.

[79] Pierangelo Dell'Acqua and Luís Moniz Pereira. Preferring and updating in logic-based agents. In Bartenstein, Geske, Hannebauer, and Yoshie, editors, *Web-Knowledge Management and Decision Support*, volume 2543 of *LNAI*, pages 69–71. Springer, 2003.

[80] Pierangelo Dell'Acqua and Luís Moniz Pereira. Preferential theory revision. *J. Applied Logic*, 5(4):586–601, 2007.

[81] Marc Denecker and Daniel De Schreye. SLDNFA: An abductive procedure for normal abductive programs. In Apt, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 686–700, Washington, USA, 1992. The MIT Press.

[82] Yannis Dimopoulos and Alberto Torres. Graph theoretical structures in logic programs and default theories. *Theoretical Computer Science*, 170(1-2):209 – 244, 1996.

[83] Jürgen Dix. A Framework for Representing and Characterizing Semantics of Logic Programs. In B. Nebel, C. Rich, and W. Swartout, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the Third International Conference (KR '92)*, pages 591–602. San Mateo, CA, Morgan Kaufmann, 1992.

[84] Jürgen Dix. A Classification Theory of Semantics of Normal Logic Programs: I. Strong Properties. *Fundamenta Informaticae*, 22(3):227–255, 1995.

[85] Jürgen Dix. A Classification Theory of Semantics of Normal Logic Programs: II. Weak Properties. *Fundamenta Informaticae*, 22(3):257–288, 1995.

[86] Jürgen Dix, Georg Gottlob, Wiktor Marek, and Cecylia Rauszer. Reducing disjunctive to non-disjunctive semantics by shift-operations. *Fundamenta Informaticae*, 28:87–100, 1996.

[87] Jürgen Dix and Martin Müller. Partial evaluation and relevance for approximations of stable semantics. In *ISMIS '94: Proceedings of the 8th International Symposium on Methodologies for Intelligent Systems*, pages 511–520, London, UK, 1994. Springer-Verlag.

[88] Jürgen Dix, Luís Moniz Pereira, and Teodor C. Przymusinski, editors. *Non-Monotonic Extensions of Logic Programming (NMELP'94), ICLP '94 Workshop, Santa Margherita Ligure, Italy, June 17, 1994, Selected Papers*, volume 927. Springer, 1995.

[89] Patrick Doherty, John Mylopoulos, and Christopher A. Welty, editors. *Proceedings, Tenth International Conference on Principles of Knowledge Representation and Reasoning, Lake District of the United Kingdom, June 2-5, 2006*. AAAI Press, 2006.

[90] Phan Minh Dung. On the relations between stable and well-founded semantics of logic programs. *Theoretical Computer Science*, 105:7–25, 1992.

[91] Phan Minh Dung. On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games. *Artificial Intelligence*, 77(2):321–358, 1995.

[92] Roy Dyckhoff, Heinrich Herre, and Peter Schroeder-Heister, editors. *Extensions of Logic Programming, 5th International Workshop, ELP'96, Leipzig, Germany, March 28-30, 1996, Proceedings*, volume 1050. Springer, 1996.

[93] Markus Eiglsperger, Martin Siebenhaller, and Michael Kaufmann. An Efficient Implementation of Sugiyama's Algorithm for Layered Graph Drawing. *Journal of Graph Algorithms and Applications*, 9(3):305–325, 2005.

[94] Thomas Eiter and Georg Gottlob. Complexity results for disjunctive logic programming and application to nonmonotonic logics. In *Proceedings of the International Logic Programming Symposium (ILPS*, pages 266–278. MIT Press, 1993.

[95] Thomas Eiter, Georg Gottlob, and Nicola Leone. Abduction from logic programs: semantics and complexity. *Theoretical Computer Science*, 189(1–2):129–177, 1997.

[96] M. H. Van Emden and R. A. Kowalski. The semantics of predicate logic as a programming language. *J. ACM*, 23(4):733–742, October 1976.

[97] E. A. Emerson. Temporal and modal logic. *Handbook of Theoretical Computer Science*, pages 997–1072, 1990.

[98] Esra Erdem and Vladimir Lifschitz. Tight logic programs. *Theory and Practice of Logic Programming*, 3(4):499–518, 2003.

[99] Esra Erdem, Fangzhen Lin, and Torsten Schaub, editors. *Logic Programming and Nonmonotonic Reasoning, 10th International Conference, LPNMR 2009, Potsdam, Germany, September 14-18, 2009. Proceedings*, volume 5753 of *Lecture Notes in Computer Science*. Springer, 2009.

[100] François Fages. Consistency of Clark's Completion and Existence of Stable Models. *Methods of Logic in Computer Science*, 1:51–60, 1994.

[101] Melvin Fitting. A Kripke-Kleene Semantics for Logic Programs. *J. Log. Program.*, 2(4):295–312, 1985.

[102] J. Freire, T. Swift, and D. S. Warren. An alternative scheduling strategy for the slg-wam. Technical report, SUNY at Stony Brook, 1995.

[103] J. Freire, T. Swift, and D. S. Warren. A framework for scheduling strategies in slg. Technical report, SUNY Stony Brook, 1998.

[104] T. H. Fung and Robert A. Kowalski. The IFF logic programming. *J. Log. Prog.*, 33(2):151 – 165, 1997.

[105] Dov M. Gabbay. Classical vs non-classical logic. In D.M. Gabbay, C.J. Hogger, and J.A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 2, chapter 2.6. Oxford University Press, 1994.

[106] Maurizio Gabbrielli and Gopal Gupta, editors. *Logic Programming, 21st International Conference, ICLP 2005, Sitges, Spain, October 2-5, 2005, Proceedings*, volume 3668. Springer, 2005.

[107] M. Osorio Galindo, J. R. Arrazola Ramírez, and J. L. Carballido. Logical weak completions of paraconsistent logics. *Journal of Logic and Computation*, 18(6):913–940, 2008.

[108] Martin Gebser, Benjamin Kaufmann, and Torsten Schaub. The conflict-driven answer set solver clasp: Progress report. In Erdem et al. [99], pages 509–514.

[109] Allen Van Gelder, Kenneth A. Ross, and John S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650, July 1991.

[110] M. Gelfond and V. Lifschitz. Logic programs with classical negation. In Warren and Szeredi, editors, *ICLP*, pages 579–597. MIT Press, 1990.

[111] Michael Gelfond. On stratified autoepistemic theories. In *AAAI*, pages 207–211, 1987.

[112] Michael Gelfond, Nicola Leone, and Gerald Pfeifer, editors. *Logic Programming and Nonmonotonic Reasoning, 5th International Conference, LPNMR'99, El Paso, Texas, USA, December 2-4, 1999, Proceedings*, volume 1730 of *Lecture Notes in Computer Science*. Springer, 1999.

[113] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In *In Procs. of ICLP-88*, pages 1070–1080. International Conference on Logic Programming 88, 1988.

[114] Matthew Ginsberg. Bilattices and modal operators. *Journal of Logic and Computation*, 1:1–41, 1990.

[115] Ana Sofia Gomes, José Júlio Alferes, and Terrance Swift. Implementing query answering for hybrid mknf knowledge bases. In Manuel Carro and Ricardo Peña, editors, *Practical Aspects of Declarative Languages, 12th International Symposium, PADL 2010*, volume 5937 of *Lecture Notes in Computer Science*, pages 25–39. Springer, January 2010.

[116] Patrick Healy and Nikola S. Nikolov. How to layer a directed acyclic graph. In Mutzel et al. [164], pages 16–30.

[117] Patrick Healy and Nikola S. Nikolov. A branch-and-cut approach to the directed acyclic graph layering problem. In *Revised Papers from the 10th International Symposium on Graph Drawing*, GD '02, pages 98–109, London, UK, 2002. Springer-Verlag.

[118] P. Hill and D. Warren, editors. *Proceedings of the Twenty-fifth International Conference on Logic Programming (ICLP'09)*, volume 5649 of *Lecture Notes in Computer Science*. Springer-Verlag, 2009.

[119] Pascal Hitzler. Topology and logic programming semantics. Diplomarbeit in mathematik, Universität Tübingen, 1998.

[120] Pascal Hitzler and Sibylle Schwarz. Level mapping characterizations of selector generated models for logic programs. In Wolf et al. [243], pages 65–75.

[121] Pascal Hitzler and Anthony Karel Seda. A note on the relationships between logic programs and neural networks. In *Proceedings of the Fourth Irish Workshop on Formal Methods, IWFM'00, Electronic Workshops in Computing (eWiC). British Computer Society*, pages 1–9, 2000.

[122] Pascal Hitzler and Anthony Karel Seda. Unique supported-model classes of logic programs. *Information*, 4:4–3, 2001.

[123] Pascal Hitzler and Matthias Wendt. A uniform approach to logic programming semantics. *TPLP*, 5(1-2):93–121, 2005.

[124] Ian Horrocks, Oliver Kutz, and Ulrike Sattler. The even more irresistible sroiq. In Doherty et al. [89], pages 57–67.

[125] Lawrence Hunter, David B. Searls, and Jude W. Shavlik, editors. *Proceedings of the 1st International Conference on Intelligent Systems for Molecular Biology, Bethesda, MD, USA, July 1993*. AAAI, 1993.

[126] Katsumi Inoue and Chiaki Sakama. A fixpoint characterization of abductive logic programs. *Journal of Logic Programming*, 27(2):107–136, 1996.

[127] Leslie Pack Kaelbling and Alessandro Saffiotti, editors. *IJCAI-05, Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence, Edinburgh, Scotland, UK, July 30-August 5, 2005*. Professional Book Center, 2005.

[128] A. C. Kakas and F. Riguzzi. Learning with abduction. In S. Džeroski and N. Lavrač, editors, *Proceedings of the 7th International Workshop on Inductive Logic Programming*, volume 1297, pages 181–188. Springer-Verlag, 1997.

[129] Antonis C. Kakas, Robert A. Kowalski, and Francesca Toni. Abductive logic programming. *J. Log. Comput.*, 2(6):719–770, 1992.

[130] Sébastien Konieczny. On the difference between merging knowledge bases and combining them, 2000.

[131] Robert A. Kowalski. Computational logic and human thinking: How to be artificially intelligent — http://www.doc.ic.ac.uk/∼rak/papers/newbook.pdf.

[132] Robert A. Kowalski. *Logic for problem solving*. Elsevier North Holland, New York, 1979.

[133] Kenneth Kunen. Negation in logic programming. *J. Log. Program.*, 4(4):289–308, 1987.

[134] Evelina Lamma and Paola Mello, editors. *AI*IA 99:Advances in Artificial Intelligence, 6th Congress of the Italian Association for Artificial Intelligence, Bologna, Italy, September 14-17, 1999, Proceedings*, volume 1792. Springer, 2000.

[135] Evelina Lamma, Luís Moniz Pereira, and Fabrizio Riguzzi. Belief revision via lamarckian evolution. *New Generation Computing*, 21(3):247–275, August 2003.

[136] N. Lavrac and S. Dzeroski. *Inductive Logic Programming: Techniques and Applications*. Ellis Horwood, New York, 1994.

[137] João Alexandre Leite. *Evolving Knowledge Bases - Specification and Semantics*. IOS Press, 2003.

[138] João Alexandre Leite, José Júlio Alferes, Luís Moniz Pereira, H. Przymusinska, and T. C. Przymusinski. A language for multi-dimensional updates. *Electronic Notes in Theoretical Computer Science*, 70(5), 2002.

[139] Nicola Leone, Pasquale Rullo, and Francesco Scarcello. Stable model checking for disjunctive logic programs. In *LID '96: Proceedings of the International Workshop on Logic in Databases*, pages 265–278, London, UK, 1996. Springer-Verlag.

[140] Nicola Leone, Francesco Scarcello, and V. S. Subrahmanian. Optimal models of disjunctive logic programs: Semantics, complexity, and computation. *IEEE Trans. Knowl. Data Eng.*, 16(4):487–503, 2004.

[141] Clarence Lewis. *A Survey of Symbolic Logic*. University of California Press, 1918. Republished by Dover, 1960.

[142] Vladimir Lifschitz. Answer set planning. In *Proceedings of the International Conference on Logic Programming*, pages 23–37, 1999.

[143] Vladimir Lifschitz. Twelve definitions of a stable model. In *ICLP '08: Proceedings of the 24th International Conference on Logic Programming*, pages 37–51, Berlin, Heidelberg, 2008. Springer-Verlag.

[144] Vladimir Lifschitz and Thomas Y. C. Woo. Answer sets in general nonmonotonic reasoning (preliminary report). In *KR*, pages 603–614, 1992.

[145] Fangzhen Lin and Xishun Zhao. On odd and even cycles in normal logic programs. In *Proceedings of the 19th National Conference on Artificial Intelligence (AAAI–2004)*, pages 80–85. The AAAI Press, 2004.

[146] Fangzhen Lin and Yuting Zhao. ASSAT: Computing Answer Sets of a Logic Program by SAT solvers. In *Artificial Intelligence*, pages 112–117, 2002.

[147] Thomas Linke, Hans Tompits, and Stefan Woltran. On acyclic and head-cycle free nested logic programs. In *Proceedings of 19th International Conference on Logic Programming (ICLP04), volume 3132 of Lecture Notes in Computer Science*, pages 225–239. Springer-Verlag, 2004.

[148] Julie Yuchih Liu, Leroy Adams, and Weidong Chen. Constructive negation under the well-founded semantics. *Journal of Logic Programming*, 38(3):295–330, 1999.

[149] J. W. Lloyd. *Foundations of logic programming; (2nd extended ed.)*. Springer-Verlag New York, Inc., New York, NY, USA, 1987.

[150] John W. Lloyd, Verónica Dahl, Ulrich Furbach, Manfred Kerber, Kung-Kiu Lau, Catuscia Palamidessi, Luís Moniz Pereira, Yehoshua Sagiv, and Peter J. Stuckey, editors. *Computational Logic - CL 2000, First International Conference, London, UK, 24-28 July, 2000, Proceedings*, volume 1861 of *Lecture Notes in Computer Science*. Springer, 2000.

[151] Zbigniew Lonc and Miroslaw Truszczynski. Computing minimal models, stable models and answer sets. *CoRR*, abs/cs/0506104, 2005.

[152] Gonçalo Lopes and Luís Moniz Pereira. Prospective programming with acorda. In *Empirically Successful Computerized Reasoning (ESCoR'06) workshop at The 3rd International Joint Conference on Automated Reasoning (IJCAR'06)*, Seattle, USA, August 2006.

[153] Gonçalo Lopes and Luís Moniz Pereira. Prospective storytelling agents. In Manuel Carro and R. Peña, editors, *Procs. 12th Intl. Symp. Practical Aspects of Declarative Languages (PADL'10)*, volume 5937 of *LNCS*, pages 294–296. Springer, January 2010. http://centria.di.fct.unl.pt/ lmp/publications/online-papers/storytelling.pdf.

[154] Thomas Lukasiewicz. Probabilistic and truth-functional many-valued logic programming. Technical report, Institute für Informatik - Justus-Liebig-Universität Giessen, December 1998.

[155] V. Wiktor Marek and Anil Nerode, editors. *Logic Programming and Nonmonotonic Reasoning, Third International Conference, LPNMR'95, Lexington, KY, USA, June 26-28, 1995, Proceedings*, volume 928. Springer, 1995.

[156] V. Wiktor Marek, Anil Nerode, and Jeffrey B. Remmel. The stable models of a predicate logic program. *Journal of Logic Programming*, 21:446–460, 1992.

[157] George F. McNulty. Fragments of first order logic, i: Universal horn logic. *The Journal of Symbolic Logic*, 42(2):pp. 221–237, 1977.

[158] Jack Minker and Carolina Ruiz. Semantics for disjunctive logic programs with explicit and default negation. *Fundam. Inf.*, 20(1-3):145–192, 1994.

[159] Bamshad Mobasher, Don Pigozzi, and Giora Slutzki. Multi-valued logic programming semantics: An algebraic approach. *Theoretical Computer Science*, 171(1–2):77–109, 1997.

[160] Robert C. Moore. Semantical considerations on nonmonotonic logic. *Artif. Intell.*, 25(1):75–94, 1985.

[161] Joan Moschovakis. Intuitionistic logic. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Spring 2004.

[162] Stephen Muggleton. Inductive logic programming. *New Generation Comput.*, 8(4):295–, 1991.

[163] Stephen Muggleton and Luc De Raedt. Inductive logic programming: Theory and methods. *J. Log. Program.*, 19/20:629–679, 1994.

[164] Petra Mutzel, Michael Jünger, and Sebastian Leipert, editors. *Graph Drawing, 9th International Symposium, GD 2001 Vienna, Austria, September 23-26, 2001, Revised Papers*, volume 2265 of *Lecture Notes in Computer Science*. Springer, 2002.

[165] José Neves, Manuel Filipe Santos, and José Machado, editors. *Progress in Artificial Intelligence, 13th Portuguese Conference on Aritficial Intelligence, EPIA 2007, Workshops: GAIW, AIASTS, ALEA, AMITA, BAOSW, BI, CMBSB, IROBOT, MASTA, STCS, and TEMA, Guimarães, Portugal, December 3-7, 2007, Proceedings*, volume 4874. Springer, 2007.

[166] Ilkka Niemelä and P. Simons. Smodels - an implementation of the stable model and well-founded semantics for normal logic programs. In *Proceedings of the 4th International Conference on Logic Programming and Nonmonotonic Reasoning*, volume 1265 of *Lecture Notes in Artificial Intelligence*, pages 420–429, July 1997.

[167] Juan Carlos Nieves. *Modeling arguments and uncertain information — A nonmonotonic reasoning approach.* PhD thesis, Technical University of Catalonia, 2008.

[168] Stanford Encyclopedia of Philosophy http://plato.stanford.edu/entries/logic modal/. Modal logic, October 2009.

[169] Artificial Intelligence Centre of the Computer Science Department at Universidade Nova de Lisboa. CENTRIA — Artificial Intelligence Centre Web Site — http://centria.fct.unl.pt/.

[170] Mauricio Osorio and Juan Carlos Nieves. Pstable semantics for possibilistic logic programs. In *MICAI*, pages 294–304, 2007.

[171] Mauricio Osorio and Alessandro Provetti, editors. *Latin-American Workshop on Non-Monotonic Reasoning, Proceedings of the 1st Intl. LA-NMR04 Workshop, Antiguo Colegio de San Ildefonso, Mexico City, D.F , Mexico, April 26th 2004*, volume 92. CEUR-WS.org, 2004.

[172] Charles Sanders Peirce. *Collected Papers of Charles Sanders Peirce*. Harvard University Press, 1931.

[173] Luís Moniz Pereira. *Handbook of the Logic of Argument and Inference*, volume 1 of *Studies in Logic and Practical Reasoning*, chapter Philosophical Incidences of Logic Programming, pages 425–448. Elsevier Science, 2002.

[174] Luís Moniz Pereira. Evolving towards an evolutionary epistemology. *International Journal of Reasoning-based Intelligent Systems (IJRIS)*, 1(1/2):68–76, July 2009.

[175] Luís Moniz Pereira. Evolutionary psychology and the unity of sciences - towards an evolutionary epistemology. In O. Pombo, J. Symons, and J. M. Torres, editors, *New approaches to the Unity of Science - vol II: Special Sciences and the Unity of Science*, Logic, Epistemology, and the Unity of Science. Springer, http://www.springer.com/series/6936, 2011.

[176] Luís Moniz Pereira and José Júlio Alferes. Well founded semantics for logic programs with explicit negation. In B. Neumann, editor, *European Conf. on Artificial Intelligence*, pages 102–106. John Wiley & Sons, 1992.

[177] Luís Moniz Pereira, José Júlio Alferes, and J. N. Aparício. Adding closed world assumptions to well founded semantics (extended improved version). *Theoretical Computer Science (Special issue on selected papers from FGCS'92)*, 122:49–68, 1993.

[178] Luís Moniz Pereira and Han The Anh. Evolution prospection. In K. Nakamatsu, editor, *Procs. First KES Intl. Symp. on Intelligent Decision Technologies - KES-IDT'09*, Engineering Series, Himeji, Japan, April 2009. Springer.

[179] Luís Moniz Pereira and Han The Anh. Evolution prospection in decision making. *Intelligent Decision Technologies (IDT)*, 3(3):157–171, October 2009.

[180] Luís Moniz Pereira, Pierangelo Dell'Acqua, and Gonçalo Lopes. Prospective updating of theories with preferences. In O. Pombo and A. Gerner, editors, *Abduction and the Process of Scientific Discovery*, Colecção Documenta, pages 65–96. Publidisa, September 2007.

[181] Luís Moniz Pereira, Pierangelo Dell'Aqua, and Gonçalo Lopes. On preferring and inspecting abductive models. In *Procs. 11th Intl. Symp. Practical Aspects of Declarative Languages (PADL'09)*, LNCS. Springer, January 2009.

[182] Luís Moniz Pereira, Pierangelo Dell'Aqua, Alexandre Miguel Pinto, and Gonçalo Lopes. Inspecting and preferring abductive models. In K. Nakamatsu and L. Jain, editors, *Handbook on Reasoning-based Intelligent Systems*. World Scientific, 2011.

[183] Luís Moniz Pereira and Gonçalo Lopes. Prospective logic agents. In Neves et al. [165], pages 73–86.

[184] Luís Moniz Pereira and Gonçalo Lopes. Prospective logic agents. *International Journal of Reasoning-based Intelligent Systems (IJRIS)*, 1(3/4):200–208, October 2009.

[185] Luís Moniz Pereira, Gonçalo Lopes, and Pierangelo Dell'Acqua. Pre and post preferences over abductive models. In James P. Delgrande and W. Kießling, editors, *Multidisciplinary Workshop on Advances in Preference Handling (M-Pref'07) at 33rd Intl. Conf. on Very Large Data Bases (VLDB'07)*. VLDB, 2007.

[186] Luís Moniz Pereira and Alexandre Miguel Pinto. Revised stable models - a new semantics for logic programs. In *In Procs. Convegno Italiano di Logica Computazionale (CILC'04)*. Convegno Italiano di Logica Computazionale (CILC'04), July 2004.

[187] Luís Moniz Pereira and Alexandre Miguel Pinto. Revised stable models - a semantics for logic programs. In Carlos Bento, A. Cardoso, and G. Dias, editors, *Procs. 12th Portuguese Intl.Conf. on Artificial Intelligence (EPIA'05)*, LNAI, pages 29–42, Covilhã, Portugal, December 2005. Springer.

[188] Luís Moniz Pereira and Alexandre Miguel Pinto. Approved models for normal logic programs. In Nachum Dershowitz and Andrei Voronkov, editors, *Procs. 14th Intl. Conf. on Logic for Programming Artificial Intelligence and Reasoning*, LPAR - LNAI, Yerevan, Armenia, October 2007. Springer.

[189] Luís Moniz Pereira and Alexandre Miguel Pinto. Reductio ad absurdum argumentation in normal logic programs. In Chitta Baral, G. Brewka, and John S. Schlipf, editors, *Ninth International Conference on Logic Programming and Nonmonotonic Reasoning*, LPNMR, pages 96–113, Tempe, AZ, USA, May 2007. Springer.

[190] Luís Moniz Pereira and Alexandre Miguel Pinto. *Oppositional Concepts in Computational Intelligence*, chapter Collaborative vs. Conflicting Learning, Evolution and Argumentation. Studies in Computational Intelligence. Springer, 2008.

[191] Luís Moniz Pereira and Alexandre Miguel Pinto. Adaptive reasoning for cooperative agents. In Salvador Abreu and Dietmar Siepel, editors, *18th Intl. Conf. on Applications of Declarative Programming and Knowledge Management (INAP'09)*, Évora, Portugal, November 2009.

[192] Luís Moniz Pereira and Alexandre Miguel Pinto. Inspection points and meta-abduction in logic programs. In Salvador Abreu and Dietmar Siepel, editors, *18th Intl. Conf. on Applications of Declarative Programming and Knowledge Management (INAP'09)*, November 2009.

[193] Luís Moniz Pereira and Alexandre Miguel Pinto. Layer supported models of logic programs. In *Procs. 10th LPNMR*, LNCS. Springer, September 2009.

[194] Luís Moniz Pereira and Alexandre Miguel Pinto. Layered models top-down querying of normal logic programs. In *Proceedings of the Practical Aspects of Declarative Languages (PADL'09)*, volume 5418 of *LNCS*, pages 254–268. Springer, January 2009.

[195] Luís Moniz Pereira and Alexandre Miguel Pinto. Side-effect inspection for decision making. In K. Nakamatsu, editor, *Procs. First KES Intl. Symp. on Intelligent Decision Technologies - KES-IDT'09*, volume 199 of *Engineering Series*, pages 139–150, Himeji, Japan, April 2009. Springer.

[196] Luís Moniz Pereira and Alexandre Miguel Pinto. Stable model implementation of layer supported models by program transformation. In Salvador Abreu and Dietmar Siepel, editors, *18th Intl. Conf. on Applications of Declarative Programming and Knowledge Management (INAP'09)*, Évora, Portugal, November 2009.

[197] Luís Moniz Pereira and Alexandre Miguel Pinto. Stable versus layered logic program semantics. In *Fifth Latin American Workshop on Non-Monotonic Reasoning 2009*, Apizaco, Tlaxcala, México, November 2009. Proceedings at http://ftp.informatik.rwth-aachen.de/Publications/CEUR-WS/.

[198] Luís Moniz Pereira and Alexandre Miguel Pinto. Inductive tight semantics for logic programs. In *Liber Amicorum in honour of Maurice Bruynooghe*, pages 17–31. K.U.Leuven, July 2010. invited paper.

[199] Luís Moniz Pereira and Carroline D. P. Kencana Ramli. Modelling probabilistic causation in decision making. In K. Nakamatsu, editor, *Procs. First KES Intl. Symp. on Intelligent Decision Technologies - KES-IDT'09*, Engineering Series, Himeji, Japan, April 2009. Springer.

[200] Luís Moniz Pereira and Carroline D. P. Kencana Ramli. Modelling decision making with probabilistic causation. *Intelligent Decision Technologies*, 4(2):133–148, February 2010. http://centria.di.fct.unl.pt/∼lmp/publications/online-papers/IDT-Probabilistic-Causation.pdf.

[201] Luís Moniz Pereira and Ari Saptawijaya. Modelling morality with prospective logic. In J. Maia Neves, M. F. Santos, and J. M. Machado, editors, *Progress in Artificial Intelligence, Procs. 13th Portuguese Intl.Conf. on Artificial Intelligence (EPIA'07)*, LNAI 4874, pages 99–111, Guimarães, December 2007. Springer.

[202] Luís Moniz Pereira and Ari Saptawijaya. Modelling morality with prospective logic. *International Journal of Reasoning-based Intelligent Systems (IJRIS)*, 1(3/4):209–221, October 2009.

[203] Luís Moniz Pereira and Ari Saptawijaya. Modelling morality with prospective logic. *Machine Ethics*, (ISBN: 978-0521112352), July 2011.

[204] Alexandre Miguel Pinto. Explorations in revised stable models - a new semantics for logic programs. Master's thesis, Faculdade de Ciências e Tecnologia - Universidade Nova de Lisboa, June 2005. Luís Moniz Pereira (superv.).

[205] Alessandro Provetti and Tran Cao Son, editors. *Answer Set Programming, Towards Efficient and Scalable Knowledge Representation and Reasoning, Proceedings of the 1st Intl. ASP'01 Workshop, Stanford, March 26-28, 2001*, 2001.

[206] H. Przymusinska and T. Przymusinski. Semantic issues in deductive databases and logic programs. In R. Banerji, editor, *Formal Techniques in Artificial Intelligence, a Sourcebook*, pages 321–367. North Holland, 1990.

[207] T. C. Przymusinski. *On the declarative semantics of deductive databases and logic programs*, pages 193–216. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1988.

[208] T.C. Przymusinski. Perfect model semantics. In *ICLP/SLP*, pages 1081–1096, 1988.

[209] Teodor C. Przymusinski. Every logic program has a natural stratification and an iterated least fixed point model. In *PODS* [3], pages 11–21.

[210] Teodor C. Przymusinski. On the declarative and procedural semantics of logic programs. *J. Autom. Reasoning*, 5(2):167–205, 1989.

[211] Teodor C. Przymusinski. Well-founded and stationary models of logic programs. *ANNALS OF MATHEMATICS AND ARTIFICIAL INTELLIGENCE*, 12:141–187, 1994.

[212] R. Ramesh and W. Chen. A portable method of integrating SLG resolution into Prolog systems. In *Proc. of the Symposium on Logic Programming*, 1994.

[213] Raymond Reiter. A logic for default reasoning. *Artificial Intelligence*, 13:81–132, 1980.

[214] Abhik Roychoudhury, K. Narayan Kumar, C.R.Ramakrishnan, and I.V. Ramakrishnan. A parametrized unfold/fold transformation framework for definite logic programs. In *Proceedings of PPDP'99*, volume 1702 of *LNCS*. Springer-Verlag, 1999.

[215] Fariba Sadri and Francesca Toni. Abduction with negation as failure for active and reactive rules. In Lamma and Mello [134], pages 49–60.

[216] K. Sagonas. *The SLG-WAM: A Search-Efficient Engine for Well-Founded Evaluation of Normal Logic Programs*. PhD thesis, SUNY at Stony Brook, 1996.

[217] K. Sagonas and T. Swift. An abstract machine for tabled execution of fixed-order stratified logic programs. *ACM Transactions on Programming Languages and Systems*, 20(3):586 – 635, May 1998.

[218] K. Sagonas, T. Swift, and D. S. Warren. The limits of fixed-order computation. *Theoretical Computer Science*, 254(1-2):465–499, 2000.

[219] Konstantinos F. Sagonas, Terrance Swift, and David Scott Warren. The xsb programming system. In *Workshop on Programming with Logic Databases (Informal Proceedings), ILPS*, page 164, 1993.

[220] T. Sato. On consistency of first-order logic programs. Technical report, Electrotechnical Laboratory, University of Ibaraki, 1987.

[221] M. Schirn, editor. *Philosophy of Mathematics Today: Proceedings of an International Conference in Munich*. Oxford University Press, Oxford, 1998.

[222] Anthony Karel Seda and Pascal Hitzler. Topology and iterates in computational logic. In *Proceedings of the 12th Summer Conference on Topology and its Applications: Special Session on Topology in Computer Science*, pages 427–469, 1997.

[223] Anthony Karel Seda and Pascal Hitzler. Strictly level-decreasing logic programs. In *Proceedings of the Second Irish Workshop on Formal Methods (IWFM'98*, pages 1–18, 1998.

[224] Ján Sefránek. Updates of logic programs. *Computing and Informatics*, 26(3):225–238, 2007.

[225] Stewart Shapiro. Logical consequence: Models and modality. In Schirn [221], pages 131–156.

[226] Yi-Dong Shen, Li yan Yuan, and Jia huai You. Slt-resolution for the well-founded semantics. *Journal of Automated Reasoning*, 28:53–97, 2002.

[227] Kozo Sugiyama, Shojiro Tagawa, and Mitsuhiko Toda. Methods for visual understanding of hierarchical system structures. *Systems, Man and Cybernetics, IEEE Transactions on*, 11(2):109 –125, 1981.

[228] T. Swift. Tabling for non-monotonic programming. *Annals of Mathematics and Artificial Intelligence*, 25(3-4):201–240, 1999.

[229] T. Swift and D. S. Warren. Performance of sequential SLG evaluation. Technical report, SUNY at Stony Brook, 1993.

[230] T. Swift and D. S. Warren. An abstract machine for SLG resolution: definite programs. In *Proceedings of the Symposium on Logic Programming*, pages 633–654, 1994.

[231] T. Swift and D. S. Warren. Analysis of sequential SLG evaluation. In *Proceedings of the Symposium on Logic Programming*, pages 219–238, 1994.

[232] Terrance Swift, Alexandre Miguel Pinto, and Luís Moniz Pereira. Incremental answer completion in xsb-prolog. In *Procs. 25th ICLP*, LNCS. Springer-Verlag, July 2009.

[233] Tommi Syrjänen. Lparse 1.0 user's manual.

[234] Takushi Tanaka, Setsuo Ohsuga, and Moonis Ali, editors. *Industrial and Engineering Applications of Artificial Intelligence and Expert Systems, Proceedings of the Ninth International Conference, Fukuoka, Japan, June 4-7, 1996*. Gordon and Breach Science Publishers, 1996.

[235] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[236] The REWERSE Team. Rewerse - reasoning on the web with rules and semantics. WWW.

[237] WASP Team. Working group on answer set programming. WWW.

[238] Francesca Toni and Robert A. Kowalski. Reduction of abductive logic programs to normal logic programs. In *Proc. 12th ICLP*, pages 367–381. MIT Press, 1994.

[239] Manuela M. Veloso and Subbarao Kambhampati, editors. *Proceedings, The Twentieth National Conference on Artificial Intelligence and the Seventeenth Innovative Applications of Artificial Intelligence Conference, July 9-13, 2005, Pittsburgh, Pennsylvania, USA*. AAAI Press / The MIT Press, 2005.

[240] Yde Venema, Lou Goble (ed, Blackwell Guide, Philosophical Logic, and Blackwell Publishers. Temporal logic. In *The Blackwell Guide to Philosophical Logic. Blackwell Philosophy Guides (2001.* Basil Blackwell Publishers, 1998.

[241] W3C. World wide web consortium. WWW.

[242] Edward O. Wilson. *Consilience : the unity of knowledge*. Knopf : Distributed by Random House, New York, 1998.

[243] Armin Wolf, Thom W. Frühwirth, and Marc Meister, editors. *19th Workshop on (Constraint) Logic Programming, Ulm, Germany, February 21-23, 2005*, volume 2005-01 of *Ulmer Informatik-Berichte*. Universität Ulm, Germany, 2005.

[244] Jia-Huai You, , Jia huai You, and Li Yan Yuan. On the equivalence of semantics for normal logic programs. *Journal of Logic Programming*, 22:211–222, 1995.

[245] Anbu Yue, Weiru Liu, and Anthony Hunter. Approaches to constructing a stratified merged knowledge base.

[246] Carlo Zaniolo. Key constraints and monotonic aggregates in deductive databases. In *Computational Logic: Logic Programming and Beyond, Essays in Honour of Robert A. Kowalski, Part II*. Springer, 2001.

[247] Yan Zhang. Logic program-based updates. *ACM Trans. Comput. Log.*, 7(3):421–472, 2006.

[248] Yan Zhang and Norman Y. Foo. A unified framework for representing logic program updates. In Veloso and Kambhampati [239], pages 707–713.

[249] Neng-Fa Zhou, Isao Nagasawa, Masanobu Umeda, Keiichi Katamine, and Toyohiko Hirota. B-prolog: A high performance prolog compiler. In Tanaka et al. [234], page 790.

A . Proofs

A.1 Proofs from Chapter 2

Proposition 2.1. ***The set of modules is a partition of the vertices of the graph.*** *Let $G = (V_G, E_G)$ be a graph and $M(G)$ the set of modules of G. Then, $M(G)$ is a partition of V_G. I.e.,*

$$V_G = \bigcup_{m \in M(G)} m$$

and

$$\forall_{m_i, m_j \in M(G)} m_i \neq m_j \Rightarrow m_i \cap m_j = \emptyset$$

Proof. First, by Definition 2.8, for each vertex $v \in V$ there is a module $m \in M(G)$ such that $v \in m$; thus proving $V_G = \bigcup_{m \in M(G)} m$. Assume now there are two different modules $m_i \neq m_j$ in $M(G)$ such that $m_i \cap m_j \neq \emptyset$. If m_i is the set of vertices of an SCC of G then m_j must be the same as m_i because, by definition of SCC, every vertex in m_i depends on every other vertex in m_i and, at least one vertex of m_j is one of the vertices of m_i. This contradicts the initial assumption that $m_i \neq m_j$, thus proving that no two different modules intersect: $\forall_{m_i, m_j \in M(G)} m_i \neq m_j \Rightarrow m_i \cap m_j = \emptyset$, which concludes the proof that $M(G)$ is a partition of V_G. □

Proposition 2.2. ***Different modules are non-mutually-dependent.*** *Let G be a graph and $M(G)$ the set of modules of G. Then,*

$$\forall_{\substack{m_i, m_j \in M(G) \\ m_i \neq m_j}} \neg((m_i \nleftarrow m_j) \wedge (m_j \nleftarrow m_i))$$

I.e., all modules are pairwise not mutually dependent.

Proof. Assume there are two different modules $m_i \neq m_j$ in $M(G)$ such that m_i and m_j are mutually dependent. Since m_i and m_j are mutually dependent, by definition 2.7, they are the vertices of an SCC and, thus, $m_i = m_j$ which contradicts the initial assumption that $m_i \neq m_j$. It follows immediately two different modules are necessarily non-mutually-dependent. □

Proposition 2.3. ***The Modules Graph of a graph*** G ***is a Directed Acyclic Graph.*** *Let* G *be a graph, and* $MG(G)$ *its modules graph. Then, by construction,* $MG(G)$ *is a Directed Acyclic Graph.*

Proof. From Proposition 2.2 we know that every two modules m_i, m_j of G, with $i \neq j$, are not mutually dependent. Therefore, if there is an edge from m_i to m_j in $MG(G)$ then there must not be an inverse edge, otherwise m_i and m_j would be mutually dependent. Hence, there are no cycles in $MG(G)$; i.e., $MG(G)$ is Acyclic. □

Theorem 2.2. ***Existence and uniqueness of*** **least layering** ***for a given graph*** G**.** *Let* G *be a graph. There is exactly one* least layering *of* G*.*

Proof. It follows directly from the definition of graph layering (Definition 2.11) that it is always possible to find a layering function for any given graph because the two cases of a layering function are mutually exclusive (thereby ensuring each vertex is assigned, at most, one single layer index), and they also cover all vertices, even those that do not depend on any other vertex (thereby ensuring each vertex is assigned, at least, one layer index). Moreover, amongst the several possible layerings there is always one with the least number of layers, and which assigns the lowest possibles ordinals to the indices of layers — that is the *least* layering. □

A.2 Proofs from Chapter 3

Proposition 3.1. ***The least atom layering of an atom identifies the highest layer with rules for the atom.*** *Let* $\mathcal{P}$ *be an CNLP,* $Lf/1$ *its least rule layering function, and* $ALf/1$ *its least atom layering function.*

$$\forall_{a \in \mathcal{H}_\mathcal{P}} ALf(a) = \alpha \Leftrightarrow \left(\forall_{r \in \mathcal{P}: head(r)=a} r \in \mathcal{P}^{\leq \alpha} \wedge (\alpha \neq 0 \Leftrightarrow \exists_{r' \in \mathcal{P}^{\alpha} head(r')=a})\right)$$

Proof. $\Rightarrow$:

Assume $a \in \mathcal{H}_\mathcal{P}$ and $ALf(a) = \alpha$. If a has rules then, by definition of least atom layering function, we have $ALf(a) = max_{r \in \mathcal{P}: head(r)=a}(Lf(r))$, i.e., $\alpha = max_{r \in \mathcal{P}: head(r)=a}(Lf(r))$. This means that all rules $r \in \mathcal{P}$ having $head(r) = a$ have $Lf(r) \leq \alpha$, and there is at least one rule r' such that $Lf(r') = \alpha \neq 0$. I.e. $\forall_{r \in \mathcal{P}: head(r)=a} r \in \mathcal{P}^{\leq \alpha} \wedge (\alpha \neq 0 \Leftrightarrow \exists_{r' \in \mathcal{P}^{\alpha} head(r')=a})$.

On the other hand, if a has no rules then, by definition of least atom layering function, we have $ALf(a) = 0$, i.e., $\alpha = 0$. Thus, $\forall_{r \in \mathcal{P}:head(r)=a} r \in \mathcal{P}^{\leq \alpha}$ becomes vacuously true because, by hypothesis, a has no rules. By knowing $\alpha = 0$ and that a has no rules ($\alpha \neq 0 \Leftrightarrow \exists_{r' \in \mathcal{P}^{\alpha} head(r')=a}$) immediately follows.

$\Leftarrow$:

Assume $a \in \mathcal{H}_{\mathcal{P}}$ and $\forall_{r \in \mathcal{P}:head(r)=a} r \in \mathcal{P}^{\leq \alpha}$. If a has rules ($\exists_{r' \in \mathcal{P}^{\alpha} head(r')=a}$) then they are all in layers $\leq \alpha$ for some α. The layers ordinals' maximum is thus α which is, by Definition 3.15, non-zero, i.e., $\alpha \neq 0$. In this case, $max_{r \in \mathcal{P}:head(r)=a}(Lf(r)) = \alpha = ALf(a)$.

If a has no rules then, $\forall_{r \in \mathcal{P}:head(r)=a} r \in \mathcal{P}^{\leq \alpha}$ vacuously holds for whichever ordinal. In particular, $\forall_{r \in \mathcal{P}:head(r)=a} r \in \mathcal{P}^{\leq 0}$ holds, i.e., $\alpha = 0$. Since a has no rules, by definition of atom least layering $ALf(a) = \alpha = 0$ also holds. □

Proposition 3.2. ***A rule's layer is greater than or equal to each of the body's literals' atom-layering.*** *Let $\mathcal{P}$ be an CNLP, $Lf/1$ its least rule layering function, and $ALf/1$ its least atom layering function.*

$$\forall_{\substack{r \in \mathcal{P} \\ a \in |body(r)|}} Lf(r) \geq ALf(a)$$

Proof. Assume $\mathcal{P}$ a CNLP, r a rule of $\mathcal{P}$ and a an atom of $\mathcal{H}_{\mathcal{P}}$ such that $a \in |body(r)|$. If a has no rules then, by definition of atom-layering function $ALf(a) = 0$, and since by definition of rule-layering function $\forall_{r \in \mathcal{P}} Lf(r) \geq 0$ we conclude $Lf(r) \geq ALf(a)$.

If a has rules then, because r depends on a we know that r depends on every rule r_a such that $head(r_a) = a$. By definition of rule-layering it must be either the case that r_a also depends on r — in which case $Lf(r) = Lf(r_a)$ — or that r_a does not depend on r — in which case $Lf(r) > Lf(r_a)$. Either way, $Lf(r) \geq Lf(r_a)$ always holds for every rule r_a. In particular, $Lf(r) \geq max_{r_a \in \mathcal{P}:head(r_a)=a}(Lf(r_a))$, i.e., $Lf(r) \geq= ALf(a)$. □

Proposition 3.3. ***Rules in the same SCC are in the same layer.*** *Let $\mathcal{P}$ be a CNLP.*

$$\forall_{r,r' \in \mathcal{P}}(r \twoheadleftarrow r' \wedge r' \twoheadleftarrow r) \Rightarrow Lf(r) = Lf(r')$$

Proof. By Definition 2.7, two rules r and r' are in the same SCC iff $r \twoheadleftarrow r'$ and $r' \twoheadleftarrow r$ hold; and by Definition 2.11, in that case, $Lf(r) = Lf(r')$ holds. Two rules in the same

SCC must also necessarily depend on each other, and, hence, be placed in the same layer. □

Proposition 3.4. ***Layering of SCCs.*** *Let $\mathcal{P}$ be a CNLP. If there is an edge from SCC_1 to SCC_2, with $SCC_1 \neq SCC_2$, in the $SCCG(\mathcal{P})$ then $\forall_{\substack{r_1 \in SCC_1 \\ r_2 \in SCC_2}} Lf(r_2) > Lf(r_1)$.*

Proof. From Proposition 3.3 we know that all rules in SCC_1 are in the same layer. Likewise, all rules in SCC_2 are in the same layer. By Definition 2.10, there is an arc from SCC_1 to SCC_2 in $SCCG(\mathcal{P})$ iff SCC_2 depends on SCC_1. Since all rules of SCC_2 depend on each other, they all also depend on SCC_1, i.e., all the rules of SCC_2 depend on all the rules of SCC_1. Since SCC_1 and SCC_2 are non-mutually-dependent (cf. Proposition 2.2) modules (cf. Definition 2.8) of $\mathcal{P}$, and SCC_2 depends on SCC_1, it must be the case, by Definition 2.11, that

$$\forall_{\substack{r_1 \in SCC_1 \\ r_2 \in SCC_2}} Lf(r_2) > Lf(r_1)$$

□

Corollary 3.1. ***A rule's layer is greater than or equal to each of the body's subsets index.*** *Let $\mathcal{P}$ be a CNLP and r a rule of $\mathcal{P}$.*

$$\forall_{body(r)^\alpha \subseteq body(r)} Lf(r) \geq \alpha$$

Proof. From Proposition 3.2 we know that $\forall_{\substack{r \in \mathcal{P} \\ a \in |body(r)|}} Lf(r) \geq ALf(a)$. It follows immediately that, for whichever subset S of $body(r)$, $\forall_{a \in S} Lf(r) \geq ALf(a)$. In particular, this holds for sets S of the form $body(r)^\alpha \subseteq body(r)$, as per Definition 3.16. □

A.3 Proofs from Chapter 6

Theorem 6.1. ***Layered Support implies Classical Support for Locally Stratified Programs.*** *Let P be a locally stratified logic program, I an interpretation, and $a \in I$ an atom. If a is layer supported in P, then a is also classically supported.*

Proof. Since a is layer supported in P, by definition, there must be some rule $r \in P$ such that $head(r) = a$ and $I \models \overline{body(r)}$. Since P is locally stratified there are no SCCs in P

and, by definition, $\overline{body(r)} = body(r)$ for every rule $r \in P$. Hence, $I \models body(r)$, i.e., a is classically supported. □

Lemma 6.2. ***The rules of the Remainder are "sub-rules" of the Layered Remainder.*** *Let P be an NLP. Then,*

$$\forall_{r \in \widehat{P}} \exists_{r' \in \mathring{P}} (head(r) = head(r') \wedge body(r) \subseteq body(r'))$$

Proof. By Definitions 6.13 and 6.14 we know that the Remainder $\widehat{P}$ and the Layered Remainder $\mathring{P}$ are fixed points of, respectively, the Reduction $\mapsto_X$ and the Layered reduction $\mapsto_{LX}$. From Lemma 6.1 we know that if $P \mapsto_{LX} P_{LX}$ and $P \mapsto_X P_X$ then $P_{LX} \supseteq P_X$, i.e., $\forall_{r \in P_X} r \in P_{LX}$. Whenever $P_X \subset P_{LX}$ it may be the case that there is some atom a with rules in P_{LX} but with no rules in P_X. In such case there might be some other individual transformations $\mapsto_P, \mapsto_N, \mapsto_S, \mapsto_F$, or $\mapsto_L$ which are still applicable in P_X but not in P_{LX}. These will further delete literals from bodies of rules of P_X but not from P_{LX} since they are not applicable to the latter. Hence, some rules $r \in P_X$ may have a counterpart $r' \in P_{LX}$ such that $body(r) \subseteq body(r')$. Further applications of $\mapsto_X$ to P_X might even reduce some rules to facts (by eventually deleting all literals in the body) which may then allow even more $\mapsto_N$ transformations to be applicable. Since $\widehat{P}$ is the fixed point of $\mapsto_X$ and $\mathring{P}$ is the fixed point of $\mapsto_{LX}$ it follows that $\forall_{r \in \widehat{P}} \exists_{r' \in \mathring{P}} (head(r) = head(r') \wedge body(r) \subseteq body(r'))$. □

Theorem 6.2. ***The Layered Well-Founded Model is more skeptical than the Well-Founded Model.*** *Let P be an NLP. Then*

$$\begin{array}{rcl} LWFM^+(P) & \subseteq & WFM^+(P) \wedge \\ LWFM^u(P) & \supseteq & WFM^u(P) \wedge \\ LWFM^-(P) & \subseteq & WFM^-(P) \end{array}$$

Proof. By Definition 6.16 we know that $WFM^+(P) = facts(\widehat{P})$, and by Definition 6.17 we know that $LWFM^+(P) = facts(\mathring{P})$. Thus, $LWFM^+(P) \subseteq WFM^+(P)$ iff $facts(\widehat{P}) \subseteq facts(\mathring{P})$.

By Lemma 6.2 we know that $\forall_{r \in \widehat{P}} \exists_{r' \in \mathring{P}} (head(r) = head(r') \wedge body(r) \subseteq body(r'))$. Since $a \in facts(\widehat{P})$ iff $\exists_{r \in \widehat{P}} head(r) = a \wedge body(r) = \emptyset$, and we know that $\exists_{r' \in \mathring{P}} head(r') = a \wedge body(r) \subseteq body(r')$, we conclude $facts(\widehat{P}) \subseteq facts(\mathring{P})$.

Again, by Lemma 6.2 it follows that $heads(\mathring{P}) \supseteq heads(\widehat{P})$. Since we already know that $facts(\mathring{P}) \subseteq facts(\widehat{P})$ it follows trivially that $(heads(\mathring{P}) \setminus facts(\mathring{P})) \supseteq (heads(\widehat{P}) \setminus facts(\widehat{P}))$, i.e., $LWFM^u(P) \supseteq WFM^u(P)$.

Since $LWFM^-(P) = \mathcal{H}_P \setminus LWFM^u(P)$ and $WFM^-(P) = \mathcal{H}_P \setminus WFM^u(P)$ it follows immediately that $LWFM^-(P) \subseteq WFM^-(P)$. □

Proposition 6.4. ***Model Relevance implies Relevance.*** *If a semantics Sem enjoys Model Relevance, then it also enjoys Relevance.*

Proof. Assume *Sem* enjoys Model Relevance, i.e.,

$$\forall_{a \in \mathcal{H}_P} \Big(\forall_{M \in Models_{Sem}(P)} a \in M \Rightarrow (\exists_{M_a \in Models_{Sem}(Rel_P(a))} M_a \subseteq M \wedge a \in M_a) \Big)$$
$$\wedge$$
$$\Big(\forall_{M_a \in Models_{Sem}(Rel_P(a))} a \in M_a \Rightarrow \exists_{M \in Models_{Sem}(P)} M_a \subseteq M \Big)$$

holds. Now we prove *Sem* also enjoys Relevance, i.e.,

$$\forall_{a \in \mathcal{H}_P} (\forall_{M \in Models_{Sem}(P)} a \in M) \Leftrightarrow (\forall_{M_a \in Models_{Sem}(Rel_P(a))} a \in M_a)$$

holds.

$\Rightarrow$:

Assuming *Sem* enjoys Brave Relevance and also that $\forall_{a \in \mathcal{H}_P} (\forall_{M \in Models_{Sem}(P)} a \in M)$ we conclude that for each such M, $\exists_{M_a \in Models_{Sem}(Rel_P(a))} M_a \subseteq M \wedge a \in M_a$. Because these are 2-valued complete models we know that there cannot be two distinct such M_a, thus we conclude more specifically that $\exists^1_{M_a \in Models_{Sem}(Rel_P(a))} M_a \subseteq M \wedge a \in M_a$, i.e., there is exactly one such model M_a of $Rel_P(a)$ contained in M with $a \in M_a$. Since all models of P contain a and there is at most (and at least) one such model M_a of $Rel_P(a)$, we conclude that $\forall_{M_a \in Models_{Sem}(Rel_P(a))} a \in M_a$.

$\Leftarrow$:

Assuming *Sem* enjoys Brave Relevance and also that $\forall_{M_a \in Models_{Sem}(Rel_P(a))} a \in M_a$. Every model M of P is necessarily a superset of some model M_a of $Rel_P(a)$ because $Rel_P(a)$ is itself a subset of P. The only requirement left to check is to guarantee the existence of such an M for every such M_a. This is assured by

$$\forall_{a \in \mathcal{H}_P} \forall_{M_a \in Models_{Sem}(Rel_P(a))} a \in M_a \Rightarrow \exists_{M \in Models_{Sem}(P)} M_a \subseteq M$$

because *Sem* is Brave Relevant. Since we assumed $\forall_{M_a \in Models_{Sem}(Rel_P(a))} a \in M_a$, we conclude $\forall_{M \in Models_{Sem}(P)} a \in M$ holds. □

A.4 Proofs from Chapter 7

Proposition 7.1. ***Rules of*** $P :: I$ ***are "sub-rules" of*** $P : I$**.** *Let P be an NLP, and I a set of literals of P. Then,*

$$\forall_{r \in P::I} \exists_{r' \in P:I} head(r) = head(r') \wedge body(r) \subseteq body(r')$$

Proof. The first step of both the Classical and the non-Classical Layer Division operations are exactly the same — adding as facts to P the atoms in I^+.

By Definitions 7.1 and 7.2 we know that

$P :: I = (\widehat{P \cup I^+})$ and

$P : I = (P \overset{\circ}{\cup} I^+)$, and by Lemma 6.2 we conclude that

$\forall_{r \in \widehat{P \cup I^+}} \exists_{r' \in P \overset{\circ}{\cup} I^+} (head(r) = head(r') \wedge body(r) \subseteq body(r'))$, i.e.,

$\forall_{r \in P::I} \exists_{r' \in P:I} head(r) = head(r') \wedge body(r) \subseteq body(r')$. □

Proposition 7.2. ***Models of a*** $P :: I$ ***are Models of*** $P : I$**.** *Let P be an NLP, and I a set of literals of P. If some M is a model of $P :: I$ then M is also a model of $P : I$.*

Proof. Since by Definition 7.1 we know $P :: I = \widehat{P \cup I^+}$, and by Definition 7.2 we know $P : I = P \overset{\circ}{\cup} I^+$, we show that models of $\widehat{P \cup I^+}$ are models of $P \overset{\circ}{\cup} I+$.

Any model M of $\widehat{P \cup I^+}$, by definition, satisfies all rules of $\widehat{P \cup I^+}$. By Proposition 7.1 we thus conclude that all rules $r' \in P \overset{\circ}{\cup} I^+$ for which there is a rule $r \in \widehat{P \cup I^+}$ such that $head(r) = head(r') \wedge body(r) \subseteq body(r')$ are already satisfied by M. Finally, all remaining rules $r'' \in P \overset{\circ}{\cup} I^+$ — i.e., there is no rule $r \in \widehat{P \cup I^+}$ such that $head(r) = head(r') \wedge body(r) \subseteq body(r')$ — are also satisfied by M because those are the rules that have been deleted by $\mapsto_N$ but not by $\mapsto_{LN}$, or by subsequent applications of the other individual transformations of the Remainder and Layered Remainder operators. This is the case because a rule is deleted by $\mapsto_N$ iff there is a fact a such that *not* a is in the body of the deleted rule, i.e., the rule has a false body and, therefore, is satisfied by the fact a which is necessarily in every model of $\widehat{P \cup I^+}$. That rule may not be deleted by $\mapsto_{LN}$ but it is, nonetheless, satisfied by a and consequently by M. The other rules deleted by

subsequent applications of the individual transformations $\mapsto_P, \mapsto_S, \mapsto_F, \mapsto_L$ in the Remainder, that were not deleted by the same transformations in the Layered Remainder, are nonetheless satisfied by M because their bodies became false in $\widehat{P \cup I^+}$ due to still having one positive literal in the body in $P \mathring{\cup} I^+$ that has no rules in $\widehat{P \cup I^+}$ and therefore is not in M. □

A.5 Proofs from Chapter 8

Theorem 8.1. ***Every Stable Model is a Minimal Hypotheses model.*** *Let P be an NLP, and M a stable model of P. Then M is also a Minimal Hypotheses model of P.*

Proof. Since M is a stable model of P we know that $M^+ = T^{\omega}_{P/M}(\emptyset)$. It follows from this that
$M^+ = T^{\omega}_{P/M}(\emptyset) = T^{\omega}_{P/M}(\emptyset) \cup M^+ = T^{\omega}_{(P/M)\cup M^+}(\emptyset) = T^{\omega}_{(P/M^+)\cup M^+}(\emptyset) = T^{\omega}_{(P\cup M^+/M^+)}(\emptyset)$
From Definitions 6.5 (Positive Reduction), 6.6 (Negative Reduction) and 6.15 (Stable Model) it follows that $P \cup M^+/M^+ = P_{M^+}$ where the latter is the result of $(P \cup M^+)(\mapsto_P \cup \mapsto_N)^{\omega} P_{M^+}$. Moreover, from Definition 6.8 (Success) we know that $T^{\omega}_{(P\cup M^+/M^+)}(\emptyset) = T^{\omega}_{P_{M^+}}(\emptyset) = facts(P^S_{M^+})$ where $P_{M^+} \mapsto^{\omega}_S P^S_{M^+}$.
I.e., $(P \cup M^+)(\mapsto_P \cup \mapsto_N \cup \mapsto_S)^{\omega} P^S_{M^+}$ and thus $T^{\omega}_{(P\cup M^+/M^+)}(\emptyset) = facts(P^S_{M^+}) = M^+$.
We know that $P^S_{M^+}$ is necessarily a *definite* logic program, hence, every rule of $P^S_{M^+}$ which does not have all of the atoms in its body dependent on facts, either depends on atoms with no rules, or all its other dependencies are circular with other likewise circular dependent rules. In this case, such rules will not be used by the T operator to produce consequences. I.e., $facts(P^S_{M^+}) = facts(P^X_{M^+})$, where $P^S_{M^+}(\mapsto_F \cup \mapsto_L)^{\omega} P^X_{M^+}$. This immediately leads to the conclusion that since
$(P \cup M^+)(\mapsto_P \cup \mapsto_N \cup \mapsto_S)^{\omega} P^S_{M^+}(\mapsto_F \cup \mapsto_L)^{\omega} P^X_{M^+}$ then
$(P \cup M^+)(\mapsto_P \cup \mapsto_N \cup \mapsto_S \cup \mapsto_F \cup \mapsto_L)^{\omega} P^X_{M^+}$, i.e, $(P \cup M^+) \mapsto^{\omega}_X P^X_{M^+}$, which, by Definition 6.13 (Remainder), allows us to conclude $P^X_{M^+} = \widehat{P \cup M^+}$. Hence, $M^+ = facts(P^S_{M^+}) = facts(P^X_{M^+}) = facts(\widehat{P \cup M^+}) = heads(\widehat{P \cup M^+})$. Now let us take the set-inclusion minimal subset H of M^+ such that $M^+ = facts(\widehat{P \cup H}) = heads(\widehat{P \cup H})$ — clearly there is such a set H since, *in extremis*, $H = M^+$.

It follows from Definition 6.13 that all rules of $\widehat{P}$ that are not simple facts have a non-well-founded negative dependency, i.e., they either depend circularly on themselves through some literal in their bodies (where at least one literal in the loop is a default negated literal), or they have an infinite dependency chain (as in, e.g., Example 3.5).

Since $M^+ = facts(\widehat{P \cup H}) = heads(\widehat{P \cup M^+})$ and H is minimal, it follows that for any strict subset $H' \subset H$ we have $facts(\widehat{P \cup H'}) \subset heads(\widehat{P \cup H'})$, and this means that there is at least one rule $r' \in \widehat{P \cup H'}$ with some *not* a literal in its body. This a atom is, by definition, an element of $Hyps(P)$ — cf. Definition 8.1. Since H is set-inclusion minimal, it follows that all the atoms in H must be elements of $Hyps(P)$ — otherwise, if there were some atom $b \in H$ such that $b \notin Hyps(P)$, then b would either be a fact or have no rules in $P \cup \widehat{(H \setminus \{b\})}$, in which case H would not be minimal, but $H \setminus \{b\}$ could possibly be. Hence, we conclude that for each stable model M of P there is some set-inclusion minimal $H \subseteq Hyps(P)$ such that $M^+ = facts(\widehat{P \cup H}) = heads(\widehat{P \cup M^+})$, i.e., M is a Minimal Hypotheses model. □

Theorem 8.2. ***At least one Minimal Hypotheses model of*** P ***complies with the Well-Founded Model.*** *Let P be an NLP. Then, there is at least one Minimal Hypotheses model M of P such that $M^+ \supseteq WFM^+(P)$ and $M^- \supseteq WFM^-(P)$.*

Proof. If $facts(\widehat{P}) = heads(\widehat{P})$, or equivalently, $WFM^u(P) = \emptyset$, then M_H is a MH model of P given that $H = \emptyset$ because $M_H^+ = facts(\widehat{P \cup H}) = heads(\widehat{P \cup H}) = facts(\widehat{P \cup \emptyset}) = heads(\widehat{P \cup \emptyset}) = facts(\widehat{P}) = heads(\widehat{P})$.

On the other hand, if $facts(\widehat{P}) \neq heads(\widehat{P})$, then there is at least one non-empty set-inclusion minimal set of hypotheses $H \subseteq Hyps(P)$ such that $H \supseteq facts(P)$. The corresponding M_H is, by definition, a MH model of P which is guaranteed to comply with $M_H^+ \supseteq WFM^+(P) = facts(\widehat{P})$ and $M_H^- \supseteq not\ WFM^-(P) = not\ (\mathcal{H}_P \setminus M_H^+)$. □

Theorem 8.3. ***All Minimal Hypotheses models of*** P ***comply with the Layered Well-Founded Model.*** *Let P be an NLP, and M a Minimal Hypotheses model M of P. Then, M is such that $M^+ \supseteq LWFM^+(P)$ and $M^- \supseteq not\ LWFM^-(P)$.*

Proof. By Definition 6.17 we know that $LWFM^+(P) = facts(\mathring{P})$ and that $LWFM^-(P) = \mathcal{H}_P \setminus heads(\mathring{P})$. Since M is a MH model of P we know there is some $H \subseteq Hyps(P)$ for which it holds that either $H = \emptyset$ or H is non-empty set-inclusion minimal such that $M^+ = facts(\widehat{P \cup H}) = heads(\widehat{P \cup H})$ and $M^- = not\ (\mathcal{H}_P \setminus M^+)$. We know that H is a subset of $Hyps(P) = \{a : \exists_{r \in \mathring{P}} not\ a \in body(r)\}$ and that M is yielded by H. This also means that M^+, being equal to $facts(\widehat{P \cup H})$ is necessarily a superset of $facts(P \mathring{\cup} H)$ and, therefore, of $facts(\mathring{P})$, i.e., $M^+ \supseteq facts(\mathring{P}) = LWFM^+(P)$. Again, since H is a subset of $Hyps(P)$ we know that for each $h \in H$ there is at least one rule r_h in $\mathring{P}$ such that $head(r_h) = h$, i.e., $h \notin LWFM^-(P)$. This means that no atoms of $LWFM^-(P)$ can ever be true in the MH model M, i.e., $M^- \supseteq not\ LWFM^-(P)$. □

Theorem 8.4. ***Minimal Hypotheses models are Layer Decomposable models.*** *Let P be an NLP, and M a Minimal Hypotheses model of P. Then, M is also a Layer Decomposable model of P, i.e., there is a* Layer Decomposition $LD_P(M) = \{M_{\leq 0}, \ldots, M_{\leq \alpha}, \ldots\}$ *of M in P, such that $M = \bigcup_{\alpha \geq 0} M_{\leq \alpha}$ and*

$$\forall_{\alpha \geq 0} M_{\leq \alpha} \text{ is a 3-valued model of } P^{\alpha} : M_{<\alpha} \qquad \text{with} \qquad M^{-}_{\leq \alpha} = not\ (A_P^{\leq \alpha} \setminus M^{+}_{\leq \alpha})$$

where $M_{<0} = M^{+}_{\leq 0} = \emptyset$.

Proof. We make this proof by induction. First we prove that $M_{\leq 0} = not\ A^0_P$ is a 3-valued model of $P^0 : M_{<0} = \emptyset : \emptyset = \emptyset$ with $M^{-}_{\leq 0} = not\ (A_P^{\leq 0} \setminus M^{+}_{\leq 0}) = not\ A^0_P$ and $M_{\leq 0} \subseteq M$. Then we prove that $M_{\leq \beta}$ is a 3-valued model of $P^{\beta} : M_{<\beta}$ with $M^{-}_{\leq \beta} = not\ (A_P^{\leq \beta} \setminus M^{+}_{\leq \beta})$, and $M_{\leq \beta} \subseteq M$ for every $\beta > 0$. Finally we prove that $M = \bigcup_{\alpha \geq 0} M_{\leq \alpha}$.

By Definition 8.3, since M is an MH model of P, we know that $M^+ = facts(\widehat{P \cup H}) = heads(\widehat{P \cup H})$ where $H \subseteq Hyps(P)$ is either $H = \emptyset$ or H is non-empty set-inclusion minimal. By Definition 8.1 we know that $H \subseteq Hyps(P) \subseteq heads(P)$. Hence, $heads(P \cup H) = heads(P)$. Moreover, since $M^+ = facts(\widehat{P \cup H}) = heads(\widehat{P \cup H})$ we conclude $M^+ \subseteq heads(P \cup H) = heads(P)$, i.e., $M^+ \subseteq heads(P)$. By Definition 3.15 we know that $A^0_P = \mathcal{H}_P \setminus heads(P)$ which, together with $M^+ \subseteq heads(P)$ leads to the conclusion that $M^- \supseteq not\ A^0_P$. Since $P^0 = \emptyset$, by Definition 3.14, we can make $M_0 = M_{\leq 0} = M^{+}_{\leq 0} \cup M^{-}_{\leq 0} = \emptyset \cup not\ A^0_P$, i.e., $M_0 = M_{\leq 0} = not\ A^0_P$. Since $M^- \supseteq not\ A^0_P$ and $M_{\leq 0} = not\ A^0_P$ we conclude $M \supseteq M_{\leq 0}$, i.e., $M_{\leq 0} = not\ A^0_P$ is indeed a 3-valued model of $P^0 : M_{<0} = \emptyset : \emptyset = \emptyset$ with $M^{-}_{\leq 0} = not\ (A_P^{\leq 0} \setminus M^{+}_{\leq 0}) = not\ (A^0_P \setminus \emptyset) = not\ A^0_P$, which makes it an element of the Layer Decomposition $LD_P(M) = \{M_{\leq 0}, \ldots, M_{\leq \alpha}, \ldots\}$ of M in P.

Now we take any $\beta > 0$, and assume $M_{\leq \beta - 1}$ is a 3-valued model of $P^{\beta-1} : M_{<\beta-1}$ with $M^{-}_{\leq \beta-1} = not\ (A_P^{\leq \beta-1} \setminus M^{+}_{\leq \beta-1})$ i.e., $M_{\leq \beta-1}$ is an element of the Layer Decomposition $LD_P(M) = \{M_{\leq 0}, \ldots, M_{\leq \beta}, \ldots\}$ of M in P and $M_{\leq \beta-1} \subseteq M$. Since $P = \bigcup_{\alpha \geq 0} P^{\alpha}$ we can take $P^{\beta} : M_{\leq \beta-1} = P^{\beta} : M_{<\beta}$ and find a 3-valued model $M_{\leq \beta}$ of it such that $M_{\leq \beta} \subseteq M$. Let us see why we can guarantee this. Take $P^{\beta} : M_{<\beta}$. By Proposition 7.2 we know that, taking $M_{<\beta-1} = M_{<\beta} \subseteq M$, a model of $P^{\beta} :: M_{<\beta}$ is a model of $P^{\beta} : M_{<\beta}$. Since M is a MH model of P we know that $M^+ = facts(\widehat{P \cup H})$, i.e., $M^+ = facts(P :: H)$ and, therefore, $M_{\leq \beta}$ is a 3-valued model of $P^{\beta} :: M_{<\beta}$, i.e., $M_{\leq \beta}$ is also a 3-valued model of $P^{\beta} : M_{<\beta}$ where we can safely make $M^{-}_{\leq \beta} = not\ (A_P^{\leq \beta} \setminus M^{+}_{\leq \beta})$.

Finally, since M is such that $M^+ = facts(\widehat{P \cup H})$ we know that $M^+ = T^{\omega}_{(P \cup H)/M^+}(\emptyset)$ because M is necessarily a Stable Model of $P \cup H$. I.e., $M^+ = \bigcup_{\alpha \geq 0} T^{\alpha}_{(P \cup H)/M^+}(\emptyset)$ where, necessarily, $T^{\alpha}_{(P \cup H)/M^+}(\emptyset) = M^{+}_{\leq \alpha}$, which means $M^+ = \bigcup_{\alpha \geq 0} M^{+}_{\leq \alpha}$ and therefore $M^- = \bigcup_{\alpha \geq 0} M^{-}_{\leq \alpha}$. □

Theorem 8.5. ***Minimal Hypotheses semantics guarantees model existence.*** *Let P be an NLP. There is always, at least, one Minimal Hypotheses model of P.*

Proof. The program Remainder $\widehat{P}$ of P is always guaranteed to exist and to be unique (cf. [45]), therefore both $\mathring{P}$ and, hence, $Hyps(P)$ (Definition 8.1) are always guaranteed to exist and to be unique as well. One can always non-deterministically select some subset H of $Hyps(P)$ and add its elements as facts to P producing $P \cup H$, of which the Remainder $\widehat{P \cup H}$ is also always guaranteed to exist and to be unique. For some such subsets H, the $\widehat{P \cup H}$ will be just a set of facts — in the extreme case where $H = Hyps(P)$, $\widehat{P \cup H}$ is necessarily guaranteed to be a set of facts. So, it is always possible to find all such subsets H of $Hyps(P)$ that yield $\widehat{P \cup H}$ as a set of facts, and to select only the empty H and the non-empty set inclusion minimal amongst those Hs. Any 2-valued model M, such that $M^+ = facts(\widehat{P \cup H})$, with such a guaranteed to exist minimal H, is a Minimal Hypotheses model of P, consequently also guaranteed to exist. □

Theorem 8.6. ***Minimal Hypotheses semantics enjoys Brave Relevance.*** *Let P be an NLP. Then, according to Definition 6.31,*

$$\Big(\forall_{\substack{a \in \mathcal{H}_P \\ M \in Models_{MH}(P)}} a \in M \Rightarrow (\exists_{M_a \in Models_{MH}(Rel_P(a))} M_a \subseteq M \wedge a \in M_a)\Big)$$

$$\wedge$$

$$\Big(\forall_{\substack{a \in \mathcal{H}_P \\ M_a \in Models_{MH}(Rel_P(a))}} a \in M_a \Rightarrow \exists_{M \in Models_{MH}(P)} M_a \subseteq M\Big)$$

holds.

Proof. Let us first prove

$$\Big(\forall_{\substack{a \in \mathcal{H}_P \\ M \in Models_{MH}(P)}} a \in M \Rightarrow (\exists_{M_a \in Models_{MH}(Rel_P(a))} M_a \subseteq M \wedge a \in M_a)\Big)$$

and then we will prove

$$\Big(\forall_{\substack{a \in \mathcal{H}_P \\ M_a \in Models_{MH}(Rel_P(a))}} a \in M_a \Rightarrow \exists_{M \in Models_{MH}(P)} M_a \subseteq M\Big)$$

Assume $a \in \mathcal{H}_P \wedge M \in Models_{MH}(P) \wedge a \in M$.
Now we need to prove $\exists_{M_a \in Models_{MH}(Rel_P(a))} M_a \subseteq M \wedge a \in M_a$. Since $P \supseteq Rel_P(a)$ and $M \in Models_{MH}(P)$ there is necessarily some $M_a \subseteq M$ such that $M_a \in Models_{MH}(Rel_P(a))$. What we need to prove in this case is that $a \in M_a$. Assume $a \notin M_a$. Since M_a is a 2-valued complete model of $Rel_P(a)$ we know that $|M_a| = \mathcal{H}_{Rel_P(a)}$ and, hence, if $a \notin M_a$,

then necessarily *not* $a \in M_a^-$. Since $M \supseteq M_a$ is also a 2-valued complete model of P it cannot be the case that $a \in M \wedge not\ a \in M^-$, otherwise M would not even be a model (cf. Definitions 5.1,5.7). Hence, *not* $a \in M^-$, i.e., *not* $a \in M$ and therefore $a \notin M$ which contradicts our initial assumption $a \in M$. We conclude $a \notin M_a$ cannot hold, i.e., $a \in M_a$ must hold.

Assume $a \in \mathcal{H}_P \wedge M_a \in Models_{MH}(Rel_P(a)) \wedge a \in M$.
Now we need to prove $\exists_{M \in Models_{MH}(P)} M_a \subseteq M$. Let us write $P_{)a(}$ as an abbreviation of $P \setminus Rel_P(a)$. We have therefore $P = P_{)a(} \cup Rel_P(a)$. Let us now take $P_{)a(} \cup M_a$. By Theorem 8.5 we know that every NLP as an MH model, hence every MH model M of $P_{)a(} \cup M_a$ is such that $M \supseteq M_a$. Let H_{M_a} denote the Hypotheses set of M_a — i.e., $M_a^+ = facts(\widehat{Rel_P(a) \cup H_{M_a}}) = heads(\widehat{Rel_P(a) \cup H_{M_a}})$, with $H_{M_a} = \emptyset$ or non-empty set-inclusion minimal, as per Definition 8.3. If $facts(\widehat{P \cup H_{M_a}}) = heads(\widehat{P \cup H_{M_a}})$ then $M^+ = facts(\widehat{P \cup H_M}) = heads(\widehat{P \cup H_M})$ is an MH model of P with $H_M = H_{M_a}$ and, necessarily, $M \supseteq M_a$.

If $facts(\widehat{P \cup H_{M_a}}) \neq heads(\widehat{P \cup H_{M_a}})$ then, by Theorem 8.5, we can always find an MH model M of $P_{)a(} \cup M_a$, with $H' \subseteq Hyps(P_{)a(} \cup M_a)$, where $M^+ = facts(\widehat{P \cup H'}) = heads(\widehat{P \cup H'})$. Such M is thus $M^+ = facts(\widehat{P \cup H_M}) = heads(\widehat{P \cup H_M})$ where $H_M = H_{M_a} \cup H'$, which means M is a MH model of P with $M \supseteq M_a$. □

Theorem 8.8. ***Minimal Hypotheses semantics enjoys Cumulativity.*** *Let P be an NLP. Then*

$$\forall_{a,b \in \mathcal{H}_\mathcal{P}} \Big((\forall_{M \in Models_{MH}(\mathcal{P})} a \in M^+) \Rightarrow$$

$$(\forall_{M \in Models_{MH}(\mathcal{P})} b \in M^+ \Leftrightarrow \forall_{M_a \in Models_{MH}(\mathcal{P} \cup \{a\})} b \in M_a^+) \Big)$$

Proof. Assume $\forall_{\substack{a \in \mathcal{H}_\mathcal{P} \\ M \in Models_{MH}(\mathcal{P})}} a \in M^+$.

$\Rightarrow$:

Assume $\forall_{M \in Models_{MH}(\mathcal{P})} b \in M^+$. Since every MH model M contains a, from Theorem 8.9 we know that all such M are also MH models of $P \cup \{a\}$. Since we assumed $b \in M$ as well, we know that b and M is a MH model of $P \cup \{a\}$ we know b is also in those MH model M of $P \cup \{a\}$. By adding a as a fact we have necessarily $Hyps(P \cup \{a\}) \subseteq Hyps(P)$ which means that there cannot be more MH models for $P \cup \{a\}$ than for P. Since we already know that for every MH model M of P, M is also a MH model of $P \cup \{a\}$

we must conclude that $\forall_{M \in Models_{MH}(P)} \exists^{1}_{M' \in Models_{MH}(P)}$ such that $M'^{+} \supseteq M^{+}$. Since $\forall_{M \in Models_{MH}(\mathcal{P})} b \in M^{+}$ we necessarily conclude $\forall_{M_a \in Models_{MH}(\mathcal{P} \cup \{a\})} b \in M_a^{+}$.

$\Leftarrow$:

Assume $\forall_{M_a \in Models_{MH}(\mathcal{P} \cup \{a\})} b \in M_a^{+}$. Since the MH semantics is relevant Theorem 8.7 if b does not depend on a then adding a as a fact to P or not has no impact on b's truth-value, and if $b \in M_a^{+}$ then $b \in M^{+}$ as well.

If b does depend on a, which is true in every MH model M of P, then either 1) b depends positively on a, and in this case since $a \in M$ then $b \in M$ as well; or 2) b depends negatively on a, and in this case the lack of a as a fact in P can only contribute, if at all, to make b true in M as well.

Then we conclude $\forall_{M \in Models_{MH}(\mathcal{P})} b \in M^{+}$. □

Theorem 8.9. ***Minimal Hypotheses semantics enjoys Brave Cautious Monotony.*** *Let P be an NLP. Then*

$$\forall_{\substack{a \in \mathcal{H}_P \\ M \in Models_{MH}(P)}} a \in M \Rightarrow M \in Models_{MH}(P \cup \{a\})$$

Proof. Assume $a \in \mathcal{H}_P$, $M \in Models_{MH}(P)$ and $a \in M$. Since M is a MH model of P, by Definition 8.3 (Minimal Hypotheses model), we know there is an $H \subseteq Hyps(P)$ which is either empty or non-empty set-inclusion minimal such that $M^{+} = facts(\widehat{P \cup H}) = heads(\widehat{P \cup H})$. Since all MH models comply with the LWFM (Theorem 8.3 and $a \in M$ it must be the case that either $a \in facts(\mathring{P})$ or $a \in (heads(\mathring{P}) \setminus facts(\mathring{P}))$. If $a \in facts(\mathring{P})$ holds then $\mathring{P} = P \mathring{\cup} \{a\}$ and therefore $Hyps(P) = Hyps(P \cup \{a\})$ which implies M is a MH model of $P \cup \{a\}$. If $a \in (heads(\mathring{P}) \setminus facts(\mathring{P}))$ then it follows immediately that $a \in LWFM^{u}(P)$ which means one of two possibilities:

1) If there are any rules in $\mathring{P}$ with *not* a in the body that are part of some SCC, then by Definition 6.7 the $\mapsto_{LN}$ operation of the Layered Remainder does not delete any of those rules, i.e., $\mathring{P}$ is also equal to $P \mathring{\cup} \{a\}$ from which M is a MH model of $P \cup \{a\}$ follows.

2) If all the rules $\mathring{P}$ with *not* a in the body are not part of any SCC then they are necessarily part of an infinite descending chain of negative dependencies along with another infinite descending chain of positive dependencies. In such case all the atoms of M that are part of the infinite chains and that depend on a become immediately determined

and necessarily take the same truth value as in M; the atoms in the infinite chains that do not depend on a are still undetermined in $P \mathring{\cup} \{a\}$, but the $Hyps(P \cup \{a\})$ still allow for the same H to be chosen, i.e., M is still a MH model of $P \cup \{a\}$. □

Theorem 8.10. ***Brave Reasoning with MH semantics is Σ_2^P-complete.*** *Let P be an NLP, and Q a set of literals, or* query. *Finding an MH model such that $M \supseteq Q$ is an Σ_2^P-complete task.*

Proof. To show that finding a MH model $M \supseteq Q$ is Σ_2^P-complete, note that a nondeterministic Turing machine with access to an NP-complete oracle can solve the problem as follows: nondeterministically guess a set H of hypotheses (i.e., a subset of $Hyps(P)$). It remains to check if H is empty or non-empty minimal such that $M^+ = facts(\widehat{P \cup H}) = heads(\widehat{P \cup H})$ and $M \supseteq Q$. Checking that $M^+ = facts(\widehat{P \cup H}) = heads(\widehat{P \cup H})$ can be done in polynomial time (because computing $\widehat{P \cup H}$ can be done in polynomial time [45] for whichever $P \cup H$), and checking H is empty or non-empty minimal requires a nondeterministic guess of a strict subset H' of H and then a polynomial check if $facts(\widehat{P \cup H'}) = heads(\widehat{P \cup H'})$. □