Knowledge representation, reasoning and declarative problem solving with Answer sets[1]

Chitta Baral
Department of Computer Science and Engineering
Arizona State University
Tempe, AZ 85287
chitta@asu.edu

August 1, 2001

[1]NOTE: THIS IS A DRAFT. I PLAN TO HAVE TWO MORE ITERATIONS FROM BEGINNING TO END. AMONG THINGS THAT REMAIN TO BE DONE ARE: HAVING A PROPER NAMING CONVENTION FOR THE PROGRAMS IN THE EXAMPLES, CLEANING UP ENGLISH ERRORS, FILLING-IN MISSING SECTIONS, ETC.

Preface

This book is about the language of logic programming with answer set semantics and its application to knowledge representation, reasoning and declarative problem solving. This book will be useful to researchers in logic programming, declarative programming, artificial intelligence, knowledge representation, and autonomous agents; to knowledge engineers who would like to create and use large knowledge bases; to software practitioners who would like to use declarative programming for fast prototyping, and for developing critical programs that must be correct with respect a formal specification; to programmers of autonomous agents who would like to build intelligent components such as planners, schedulers, and diagnosis and repair systems; and to students and teachers using it as a text book in undergraduate and graduate classes.

The bulk of the book focuses on the research in logic programming since 1988, when the *stable model* semantics was proposed by Gelfond and Lifschitz. The main differences between this book and earlier books[1] on logic programming (besides the fact that it includes results since those earlier books were written) are: (i) It uses answer set semantics and covers all programs. In contrast the book by Lloyd covers only stratified programs, and the book by Lobo et al. and Alferes et al. use well-founded semantics or its variant with respect to non-stratified programs. (ii) A big part of this book is about declarative programming and knowledge representation methodology. It presents several small and big example modules, and presents the theory that describes when modules can be combined, when a module is consistent, how to incorporate an observation, etc. To the best of our knowledge no book on logic programming discusses this. (iii) Because it uses answer set semantics which allows multiple 'models' of a theory, it is able to go beyond reasoning to declarative problem solving. Thus it includes encoding of applications such as planning, diagnosis, explanation generation, scheduling, combinatorial auctions, abductive reasoning, etc. Most of these applications are related to encoding problems that are NP-complete or beyond. The well-founded semantics used in the other books is less expressive and is mostly suitable for reasoning tasks. The book does devote some attention to well-founded semantics. Since the well-founded semantics is sound with respect to answer set semantics and is easier to compute, in this book it is treated as an approximation to answer-set semantics. (iv) The book discusses the complexity and expressibility issues and identifies subsets belonging to different complexity and expressibility classes. (v) It presents algorithms to compute answer sets. Some of the algorithms it discusses use heuristics and other intelligent search ideas. (vi) Unlike the books by Lloyd and Lobo et al. most of the programs discussed in the book can be run. It uses the smodels and the dlv interpreter for this and is supplemented by a web site containing a large subset of the example programs as smodels or dlv code.

This book can be used for an undergraduate (junior-senior level) one semester course and a one semester graduate course. For the undergraduate course it is recommended to cover most of Chapter 1, Chapter 2, a very small part of Chapter 3, Chapter 4, parts of Chapter 5, parts of Chapter 7 and Chapter 8. For a follow-up graduate course it is recommended to cover the remaining material (Chapter 3, most of Chapter 5, Chapter 6, parts of Chapter 7, and Chapter 9) together with a quick overview of Chapters 1, 2, 4 and 8. For a stand alone graduate course that does not have

[1] The earlier books that we are referring to are: the books by Lloyd published in 1982 (first edition) and 1987 (second edition) titled 'Foundations of logic programming', the monograph by Lobo, Minker and Rajasekar published in 1992 titled 'Foundations of disjunctive logic programming', and the monograph by Alferes and Pereira published in 1996 titled 'Reasoning with logic programming'.

the undergraduate course as the pre-requisite it is recommended to cover Chapter 1-8, leaving out a few sections in some of the chapters.

This book came about after almost a decade of interacting with and learning from my colleague and teacher Michael Gelfond at the University of Texas at El Paso. Many ideas and perspectives behind this book that unify the different sections and chapters are due to this interaction and learning. Thomas Eiter helped the author in understanding several aspects of the complexity and expressibility results. Interactions with Vladimir Lifschitz had a significant influence in the various aspects of this book. The author would also like to acknowledge the help of his students (Nam Tran, Tran Son, Le-Chi Tuan, Viet Hung Nguyen, Cenk Uyan, and Guray Alsac), the smodels group (particularly, Niemela and Syrjanen), dlv group (particularly Eiter, Gottlob, Leone, and Pfiefer), and his colleagues Lobo, Provetti and Galindo in writing this book. Finally, he would like to acknowledge support from his start-up funds at the Arizona State University, NSF support through grants IRI-9501577 and 0070463 and NASA support through grant NCC2-1232.

About the book

Representing knowledge and reasoning with it are important components of an intelligent system, and are two important facets of Artificial Intelligence. In this the role of formal logic, in particular the non-monotonic logics is well established. Another important expectation from intelligent systems is their ability to accept high level requests – as opposed to detailed step-by-step instructions, and use their knowledge and reasoning ability to figure out the detailed steps that need to be taken. To have this ability intelligent systems must have a declarative interface – whose input language must be based on logic.

In this book Chitta Baral proposes to use the non-monotonic language of AnsProlog – logic programming with answer set semantics, for both knowledge representation and reasoning, and declarative problem solving. He presents results obtained over the years in a first ever compilation. This compilation is not only unique to logic programming, but is unique to the field of knowledge representation as a whole, as for no other language or logic a comparable body of results has been accumulated. The book is targeted towards students, researchers and practitioners and its content includes: (a) theoretical results about AnsProlog such as, properties of AnsProlog sub-classes including their complexity and expressibility, and building block results to both analyze and modularly build large AnsProlog programs; (b) illustrations of correctness proofs of AnsProlog programs; (c) algorithms to compute answer sets ('models') of AnsProlog programs; (d) demonstration of the knowledge representation and reasoning ability of AnsProlog through benchmark examples such as the frame problem, reasoning about actions, inheritance hierarchies, reasoning with priorities, and reasoning with incomplete information; (e) use of AnsProlog in problem solving activities such as planning, diagnosis, constraint satisfaction problems, logic puzzles, and combinatorial auctions; and (f) descriptions of systems that implement AnsProlog and code of various examples in the syntax of these systems.

About the author

Chitta Baral is an associate professor at the Arizona State University. He obtained his B.Tech(Hons) degree from the Indian Institute of Technology, Kharagpur and his M.S and Ph.D degrees from the University of Maryland at College Park. He has been working in the field of knowledge representation and logic programming since 1988, and his research has been supported over the years by National Science Foundation, NASA, and United Space Alliance. He received the NSF CAREER award in 1995 and led successful teams to AAAI 96 and 97 robot contests. He has published more than 65 articles in Logic Programming, Knowledge Representation, and Artificial Intelligence conferences and Journals.

Brief description of the chapters

- **Chapter 1: Declarative programming in AnsProlog: introduction and preliminaries**
  In Chapter 1 we motivate the importance of declarative languages and argue that intelligent entities must be able to comprehend and process descriptions of '*what*', rather than being told '*how*' all the time. We then make the case for AnsProlog (programming in logic with answer set semantics) and compare it with other non-monotonic languages, and with the Prolog programming language. We then present the syntax and semantics various sub-classes of AnsProlog, and consider two views of AnsProlog programs: stand alone programs, and functions. We present more than 30 examples illustrating the various definitions and results.

- **Chapter 2: Simple modules for declarative programming with answer sets**
  In this chapter we present several small AnsProlog programs/modules corresponding to several problem solving or knowledge representation modules. This chapter is like a tool box of programs that can be combined for larger applications. In a sense it gives a quick glimpse of the book, and can be thought of as introducing the usefulness and applicability of AnsProlog through examples.

- **Chapter 3: Principles and properties of declarative programming with answer sets**
  In this chapter we present several fundamental results that are useful in *analyzing* and *step-by-step building* of AnsProlog programs, viewed both as stand-alone programs and as functions. To analyze AnsProlog programs we define and describe several properties such as categoricity – presence of unique answer sets, coherence – presence of at least one answer set, computability – answer set computation being recursive, filter-abducibility – abductive assimilation of observations using filtering, language independence – independence between answer sets of a program and the language, language tolerance – preservation of the meaning of a program with respect to the original language when the language is enlarged, functional, compilable to first-order theory, amenable to removal of **or**, amenable to computation by Prolog, and restricted monotonicity – exhibition of monotonicity with respect to a select set of literals.

  We also define several subclasses of AnsProlog programs such as stratified, locally stratified, acyclic, tight, signed, head cycle free and several conditions on AnsProlog rules such as well-moded and state results about which AnsProlog programs have what properties. We present several results that relate answer sets of an AnsProlog program with its rules. We develop the notion of splitting and show how the notions of stratification, local stratification, and splitting can be used in step-by-step computation of answer sets.

  For *step by step building* of AnsProlog programs we develop the notion of conservative extension – where a program preserves its original meaning after additional rules are added to it, and present conditions for programs that exhibit this property. We present several operators such as incremental extension, interpolation, domain completion, input opening and input extension, and show how they can be used for systematically building larger programs from smaller modules.

- **Chapter 4: Declarative problem solving and reasoning in AnsProlog**
  In this chapter we formulate several knowledge representation and problem solving domains using AnsProlog. Our focus in this chapter is on program development. We start with three

well known problems from the literature of constraint satisfaction, and automated reasoning: placing queens in a chess board, determining who owns the zebra, and finding tile covering in a mutilated chess board. We present several encodings of these problems using AnsProlog and analyze them. We then discuss a general methodology for representing constraint satisfaction problems (CSPs) and show how to extend it to dynamic CSPs. We then present encodings of several combinatorial graph problems such as k-colarability, Hamiltonian circuit, and K-clique. After discussing these problem solving examples, we present a general methodology of reasoning with prioritized defaults, and show how reasoning with inheritance hierarchies is a special case of this.

- **Chapter 5: Reasoning about actions and planning in AnsProlog**
  In this chapter we consider reasoning about actions in a dynamic world and its application to plan verification, simple planning, planning with various kinds of domain constraints, observation assimilation and explanation, and diagnosis. We do a detailed and systematic formulation - in AnsProlog – of the above issues starting from the simplest reasoning about action scenarios and gradually increasing its expressiveness by adding features such as causal constraints, and parallel execution of actions. We also prove properties of our AnsProlog formulations using the results in Chapter 3.

  Our motivation behind the choice of a detailed formulation of this domain is two fold. (i) Reasoning about actions captures both major issues of this book: knowledge representation and declarative problem solving. To reason about actions we need to formulate the frame problem whose intuitive meaning is that objects in the worlds do not normally change their properties. Formalizing this has been one of the benchmark problem of knowledge representation and reasoning formalisms. We show how AnsProlog is up to this task. Reasoning about actions also form the ground work for planning with actions, an important problem solving task. We present AnsProlog encodings of planning such that the answer sets each encode a plan. (ii) Our second motivation is in regards to the demonstration of the usefulness of the results in Chapter 3. We analyze and prove properties of our AnsProlog formulations of reasoning about actions and planning by using the various results in Chapter 3, and thus illustrate their usefulness. For this we also start with simple reasoning about action scenarios and then in later sections we consider more expressive scenarios.

- **Chapter 6: Complexity, expressibility and other properties of AnsProlog programs**
  In this chapter we consider some broader properties that help answer questions such as: (a) how difficult it is to compute answer sets of various sub-classes of AnsProlog; (b) how expressive are the various sub-classes of AnsProlog; (c) how modular is AnsProlog; and (d) what is the relationship between AnsProlog and other non-monotonic formalisms.

  The answers to these questions are important in many ways. For example, if we know the complexity of a problem that we want to solve then the answer to (a) will tell us which particular subset of AnsProlog will be most efficient, and the answer to (b) will tell us the most restricted subset that we can use to represent that problem. To make this chapter self complete we start with the basic notions of complexity and expressibility, and present definitions of the polynomial, arithmetic and analytical hierarchy and their normal forms. We later use them in showing the complexity and expressibility of AnsProlog subclasses.

- **Chapter 7: Answer set computing algorithms**

In this chapter we present several answer set computing algorithms and compare them. The particular algorithms we present are the wfs-bb algorithm that uses branch and bound after computing the well-founded semantics, the assume-and-reduce algorithm of SLG, the Smodels algorithm, and the dlv algorithm.

- **Chapter 8: Query answering and answer set computing systems**
  In this chapter we explain how to program using the Smodels and dlv systems, discuss the extensions that these systems have beyond AnsProlog, and present several programs in the their syntax. We then describe when a Prolog interpreter can be used in answering queries to AnsProlog programs and under what conditions the Prolog interpreter is sound and complete with respect to AnsProlog. Finally, we briefly mention some of the other systems that can accept particular sub-classes of AnsProlog programs.

- **Chapter 9: Further extensions of and alternatives to AnsProlog**
  In this chapter we discuss further extensions to AnsProlog, such as allowing **not** in the head of rules, allowing epistemic operators, and doing abductive reasoning. We also discuss some of the alternative characterizations of programs in AnsProlog syntax.

- **Appendices**
  We have several appendices: some of them in the book, and others in a companion web site. In the appendices in the book we present some background information, such as definition of ordinals, lattices, fixpoints, and definitions of Turing machine. In the web site we have Smodels and dlv code of several programs discussed throughout the book and also a list of pointers to resources such as home pages of active scientists and researchers in this field, and web pages of implemented systems.

Contents

Chapter 1

Declarative programming in AnsProlog : introduction and preliminaries

Among other characteristics, an intelligent entity – whether an intelligent autonomous agent, or an intelligent assistant – must have the ability to go beyond just following direct instructions while in pursuit of a goal. This is necessary to be able to behave intelligently when the assumptions surrounding the direct instructions are not valid, or there are no direct instructions at all. For example even a seemingly direct instruction of 'bring me coffee' to an assistant requires the assistant to figure out what to do if the coffee pot is out of water, or if the coffee machine is broken. The assistant will definitely be referred to as lacking intelligence if he were to report to the boss that there is no water in the coffee pot and ask him what to do next. On the other hand, an assistant will be considered intelligent if he can take a high level request of "make travel arrangements for my trip to International AI conference 20XX" and figure out the lecture times of his boss, take into account his airline, hotel and car rental preferences, take into account the budget limitations, etc. and overcome hurdles such as the preferred flight being sold out and make satisfactory arrangements. This example illustrates *one benchmark of intelligence – the level of request an entity can handle.* At one end of the spectrum the request is a detailed algorithm that spells out *how* to satisfy the request, which no matter how detailed it is may not be sufficient in cases where the assumptions inherent in the algorithm are violated. In the other end of the spectrum the request spells out *what* needs to be done, and the entity has the knowledge – again in the *what* form rather than the *how* form – and the knowledge processing ability to figure out the exact steps (that will satisfy the request) and execute them, and in case of not having the necessary knowledge it either knows where to obtain the necessary knowledge, or is able to gracefully get around it through its ability to reason in presence of incomplete knowledge.

The languages for spelling out *how* are often referred to as *procedural* while the languages for spelling out *what* are referred to as *declarative* . Thus our initial thesis that intelligent entities must be able to comprehend and process descriptions of *what* leads to the necessity of inventing suitable declarative languages and developing support structures around those languages to facilitate their use. We consider the development of such languages to be fundamental to knowledge based intelligence, perhaps similar to the role of the language of calculus in mathematics and physics. *This book is about such a declarative language – the language of* **AnsProlog**. We now give a brief history behind the quest for a suitable declarative language for knowledge representation, reasoning and

declarative problem solving.

Classical logic which has been used as a specification language for procedural programming languages was an obvious initial choice to represent declarative knowledge. But it was quickly realized that classical logic embodies the monotonicity property according to which the conclusion entailed by a body of knowledge stubbornly remains valid no matter what additional knowledge is added. This disallowed human like reasoning where conclusions are made with the available (often incomplete) knowledge and may be withdrawn in presence of additional knowledge. This led to the development of the field of *non-monotonic logic*, and several non-monotonic logics such as circumscription, default logic, auto-epistemic logic, and non-monotonic modal logics were proposed. The AI journal special issue of 1980 (volume 13, number 1 and 2) contained initial articles on some of these logics. In the last twenty years there have been several studies on these languages on issues such as representation of small common-sense reasoning examples, alternative semantics of these languages, and relationship between the languages. But the dearth of efficient implementations, use in large applications – say of more than ten pages, and studies on building block support structures has diminished their applicability for the time being. Perhaps the above is due to some fundamental lacking, such as all of these languages which build on top of the classical logic syntax and allow nesting are quite complex, and all except default logic lack structure, thus making it harder to use them, analyze them and develop interpreters for them.

An alternative non-monotonic language paradigm with a different origin whose initial focus was to consider a subset of classical logic (rather than extending it) is the programming language PROLOG and the class of languages clubbed together as 'logic programming'. PROLOG and logic programming grew out of work on automated theorem proving and Robinson's resolution rule. One important landmark in this was the realization by Kowalski and Colmerauer that logic can be used as a programming language, and the term PROLOG was developed as an acronym to PROgramming in LOGic. A subset of first-order logic referred to as Horn clauses that allowed faster and simpler inferencing through resolution was chosen as the starting point. The notion of closed world assumption in databases was then imported to PROLOG and Logic programming and the operator **not** was used to refer to negative information. The evolution of PROLOG was guided by concerns to make it a full fledged programming language with efficient implementations, often at the cost of sacrificing the declarativeness of logic. Nevertheless, research also continued on logic programming languages with declarative semantics. In the late eighties and early nineties the focus was on finding the right semantics for agreed upon syntactic subclasses. One of the two most popular semantics proposed during that time is the *answer set semantics*, also referred to as the *stable model semantics*.

This book is about the language of logic programming with respect to the answer set semantics. We refer to this language as AnsProlog, as a short form of '**Pro**gramming in **log**ic with **Ans**wer sets'[1]. In the following section we compare AnsProlog with PROLOG and also with the other non-monotonic languages, and present the case for AnsProlog as the most suitable declarative language for knowledge representation, reasoning and declarative problem solving.

1.1 Motivation: Why AnsProlog?

We start with a short summary and then expand on it. The non-classical symbols $\leftarrow$, and **not** in AnsProlog give a structure to AnsProlog programs and allow us to easily define syntactic subclasses

[1] In the literature it has been sometimes referred to as A-Prolog.

and study their properties. It so happens that these various sub-classes have a range of complexity and expressibility thus allowing us to choose the appropriate subclasses for particular applications. Moreover, there exists a more tractable approximate characterization which can be used – at the possible cost of completeness – when time is a concern. Unlike the other non-monotonic logics, AnsProlog now has efficient implementations which have now been used to program large applications. In addition, the expressibility studies show AnsProlog to be as expressible as some of these logics, while syntactically it seems less intimidating as it does not allow arbitrary formulas. Finally, the most important reason to study and use AnsProlog is that there is now already a body of support structure around AnsProlog that includes the above mentioned implementations and theoretical building block results that allow systematic construction of AnsProlog programs, and assimilation of new information. We now expand on these points in greater detail.

1.1.1 AnsProlog vs Prolog

Although, Prolog grew out of programming with Horn clauses, a subset of first-order logic; several non-declarative features were included in Prolog to make it programmer friendly. We propose AnsProlog as a declarative alternative to Prolog. Following are the main differences between AnsProlog and Prolog.

- The ordering of literals in the body of a rule matters in Prolog as it processes them from left to right. Similarly, the positioning of a rule in the program matters in Prolog as it processes them from start to end. The ordering of rules and positioning of literals in the body of a rule do not matter in AnsProlog. From the perspective of AnsProlog, a program is a *set* of AnsProlog rules, and in each AnsProlog rule, the body is a *set* of literals.

- Query processing in Prolog is top-down from query to facts. In AnsProlog query-processing methodology is not part of the semantics. Most sound and complete interpreters with respect to AnsProlog do bottom-up query processing from facts to conclusions or queries.

- Because of the top-down query processing, and start to end, and left to right processing of rules and literals in the body of a rule respectively, a Prolog program may get into an infinite loop for even simple programs without negation as failure.

- The *cut* operator in Prolog is extra-logical, although there have been some recent attempts at characterizing them. This operator is not part of AnsProlog.

- There are certain problems, such as floundering and getting stuck in a loop, in the way Prolog deals with negation as failure. In general, Prolog has trouble with programs that have recursions through the negation as failure operator. AnsProlog does not have these problems, and as its name indicates it uses the *answer-set* semantics to characterize negation as failure.

In this book, besides viewing AnsProlog as a declarative alternative to Prolog, we also view Prolog as a top-down query answering system that is correct with respect to AnsProlog under certain conditions. In Section 8.3 we present these conditions and give examples that satisfy these conditions.

1.1.2 AnsProlog vs Logic programming

AnsProlog is a particular kind of logic programming. In AnsProlog we fix the semantics to *answer set semantics*, and only focus on that. On the other hand logic programming refers to a broader

agenda where different semantics are considered as alternatives. We now compare AnsProlog with the alternative semantics of programs with AnsProlog syntax.

Since the early days of logic programming there have been several proposals for semantics of programs with AnsProlog syntax. We discuss some of the popular ones in greater detail in Chapter 9. Amongst them, the most popular ones are the *stable model semantics* and the *well-founded semantics*. The stable models are same as the answer sets of AnsProlog programs, the main focus of this book. The well-founded semantics differ from the stable model semantics in that:

- Well-founded models are three-valued, while stable models are two valued.
- Each AnsProlog program has a unique well-founded model, while some AnsProlog programs have multiple models and some do not have any.

  For example, the program $\{p \leftarrow \textbf{not } p.\}$ has no stable models while it has the unique well-founded model where p is assigned the truth value *unknown*.

  The program $\{b \leftarrow \textbf{not } a., a \leftarrow \textbf{not } b., p \leftarrow a., p \leftarrow b.\}$ has two stable models $\{p, a\}$ and $\{p, b\}$ while its unique well-founded model assigns the truth value *unknown* to p, a and b.
- Computing the well-founded model or entailment with respect to it is more tractable than computing the entailment with respect to stable models. On the other hand the later increases the expressive power of the language.

As will be clear from many of the applications that will be discussed in Chapters 5 and 4.8, the non-determinism that can be expressed through multiple stable models plays an important role. In particular, they are important for enumerating choices that are used in planning and also in formalizing aggregation. On the other hand, the absence of stable models of certain programs, which was initially thought of as a drawback of the stable model semantics, is useful in formulating integrity constraints whose violation forces elimination of models.

1.1.3 AnsProlog vs Default logic

AnsProlog can be considered as a particular subclass of default logic that leads to a more efficient implementation. Recall that a default logic is a pair (W, D), where W is a first-order theory and D is a collection of defaults of the type $\frac{\alpha:\beta_1,\ldots,\beta_n}{\gamma}$, where α, β and γ are well-founded formulas. AnsProlog can be considered as a special case of a default theory where $W = \emptyset$, γ is an atom, α is a conjunction of atoms, and β_i's are literals. Moreover, it has been shown that AnsProlog^{or} and default logic have the same expressibility. In summary, AnsProlog is syntactically simpler to default logic and yet has the same expressibility, thus making it more usable.

1.1.4 AnsProlog vs Circumscription and classical logic

The connective '$\leftarrow$' and the negation as failure operator '**not** ' in AnsProlog add structure to an AnsProlog program. The AnsProlog rule $a \leftarrow b.$ is different from the classical logic formula $b \supset a$, and the connective '$\leftarrow$' divides the rule of an AnsProlog program into two parts: the head and the body.

This structure allows us to define several syntactic and semi-syntactic notions such as: *splitting, stratification, signing*, etc. Using these notions we can define several subclasses of AnsProlog

programs, and study their properties such as: *consistency, coherence, complexity, expressibility, filter-abducibility* and *compilability to classical logic.*

The subclasses and their specific properties have led to several building block results and realization theorems that help in developing large AnsProlog program in a systematic manner. For example, suppose we have a set of rules with the predicates $p_1, \ldots, p_n$ in them. Now if we add additional rules to the program such that $p_1, \ldots, p_n$ only appear in the body of the new rules, then if the overall program is consistent then the addition of the new rules does not change the meaning to the original predicates $p_1, \ldots, p_n$. Additional realization theorems deal with issues such as: *When CWA about certain predicates can be explicitly stated without changing the meaning of the modified program?* and *How to modify an AnsProlog program which assumes CWA so that it reasons appropriately when CWA is removed for certain predicates and we have incomplete information about these predicates?*

The non-classical operator $\leftarrow$ encodes a form of directionality that makes it easy to encode causality, which can not be expressed in classical logic in a straight forward way. AnsProlog is more expressive than propositional and first-order logic and can express transitive closure and aggregation that are not expressible in them.

1.1.5 AnsProlog as a knowledge representation language

There has been extensive study about the suitability of AnsProlog as a knowledge representation language. Some of the properties that have been studied are:

- When an AnsProlog program exhibits *restricted monotonicity.* I.e., it behaves monotonically with respect to addition of literals about certain predicates. This is important when developing an AnsProlog program where we do not want future information to change the meaning of a definition.
- When an AnsProlog program is *language independent*? When it is *language tolerant*? When it is *sort-ignorable*; i.e., when sorts can be ignored?
- When knew knowledge can be added through filtering?

In addition it has been shown that AnsProlog provides *compact representation* in certain knowledge representation problems; i.e., an equivalent representation in a tractable language would lead to an exponential blow-up in space. Similarly, it has been shown that certain representations in AnsProlog can not be *modularly* translated into propositional logic. On the other hand problems such as constraint satisfaction problems, dynamic constraint satisfaction problem, etc. can be modularly represented in AnsProlog. Similar to its relationship with default logic, subclasses of other non-monotonic formalisms such as auto-epistemic logic have also been shown to be equivalent to AnsProlog.

Finally, AnsProlog has a sound approximate characterization, called the well-founded semantics, which has nice properties and which is computationally more tractable.

1.1.6 AnsProlog implementations: Both a specification and a programming language

Since AnsProlog is fully declarative, representation (or programming) in AnsProlog can be considered both as a specification and a program. Thus AnsProlog representations eliminate the ubiquitous gap between specification and programming.

There are now some efficient implementations of AnsProlog, and many applications are built on top of these implementations. Although there are also some implementations of other non-monotonic logics such as default logic (DeReS at the University of Kentucky) and circumscription (at the Linkoping University), these implementations are very slow and very few applications have been developed based on these implementations.

1.1.7 Applications of AnsProlog

Following is a list of applications of AnsProlog to database query languages, knowledge representation, reasoning and planning.

- AnsProlog has a greater ability than Datalog in expressing database query features. In particular, AnsProlog can be used to give a declarative characterization of the standard *aggregate operators*, and recently it has been used to define new aggregate operators, and even data mining operators. It can be also used for querying in presence of different kinds of incomplete information, including *null values*.
- AnsProlog has been used in planning and allows easy expression of different kinds of (procedural, temporal and hierarchical) domain control knowledge, ramification and qualification constraints, conditional effects and other ADL constructs, and can be used for approximate planning in presence of incompleteness. Extension of AnsProlog with disjunction in the head can be used for conformant planning, and there are attempts to use AnsProlog for planning with sensing and diagnostic reasoning. It has been also used for assimilating observation of an agent and planning from the current situation by an agent in a dynamic world.
- AnsProlog has been used in product configuration, representing CSP and DCSP problems.
- AnsProlog has been used for scheduling, supply chain planning and in solving combinatorial auctions.
- AnsProlog has been used in formalizing deadlock and reachability in Petri nets, in characterizing monitors, and in cryptography.
- AnsProlog has been used in verification of contingency plans for shuttles, and also has been used in verifying correctness of circuits in presence of delays.
- AnsProlog has been used in benchmark knowledge representation problems such as reasoning about actions, plan verification, and the frame problem there in, in reasoning with inheritance hierarchies, and in reasoning with prioritized defaults. It has been used to formulate normative statements, exceptions, weak exceptions, and limited reasoning about what is known and what is not.
- AnsProlog is most appropriate for reasoning with incomplete information. It allows various degrees of trade-off between computing efficiency and completeness when reasoning with incomplete information.

1.2 Answer-set theories and programs

In this section we define the syntax of an AnsProlog program (and its extensions and subclasses), and the various notations that will be used in defining the syntax and semantics of these programs and in their analysis in the rest of the book.

An *answer-set theory* consists of an alphabet, and a language $\mathcal{L}$ defined over that alphabet. The *alphabet* of an answer-set theory consists of seven classes of symbols:

1. variables,
2. object constants (also referred to as constants),
3. function symbols,
4. predicate symbols,
5. connectives,
6. punctuation symbols, and
7. the special symbol $\perp$;

where the connectives and punctuation symbols are fixed to the set $\{\neg, \mathit{or}, \leftarrow, \mathbf{not}, \&\}$ and { '(', ')', ',', '.' } respectively; while the other classes vary from alphabet to alphabet.

We now present an example to illustrate the role of the above classes of symbols. Consider a world of blocks in a table. In this world, we may have object constants such as $block1$, $block2$, ... corresponding to the particular blocks and the object constant *table* referring to the table. We may have a predicates *on_table*, and *on* that can be sued to describe the various properties that hold in a particular instance of the world. For example, $on\_table(block1)$ means that $block1$ is on the table. Similarly, $on(block2, block3)$ may mean that $block2$ is on top of $block3$. An example of a function symbol could be *on_top*, where $on\_top(block3)$ will refer to the block (if any) that is on top of block3.

Unlike the earlier prevalent view of considering logic programs as a subset of first order logic we consider answer set theories to be different from first-order theories, particularly with some different connectives. Hence, to make a clear distinction between the connectives in a first-order theory and the connectives in an answer-set theory, we use different symbols than normally used in first-order theories: *or* instead of $\vee$, & instead of $\wedge$.

We use some informal notational conventions. In general, variables are arbitrary strings of English letters and numbers that start with an upper-case letter, while constants, predicate symbols and function symbols are strings that start with a lower-case letter. Sometimes – when dealing with abstractions – we use the additional convention of using letters $p, q, \ldots$ for predicate symbols, $X, Y, Z, \ldots$ for variables, $f, g, h \ldots$ for function symbols, and $a, b, c, \ldots$ for constants.

A *term* is inductively defined as follows:

1. A variable is a term.
2. A constant is a term.
3. If f is an n-ary function symbol and $t_1, \ldots, t_n$ are terms then $f(t_1, \ldots, t_n)$ are terms.

A term is said to be *ground*, if no variable occurs in it. The Herbrand Universe of $\mathcal{L}$, denoted by $HU_{\mathcal{L}}$, is the set of all ground terms which can be formed with the functions and constants in $\mathcal{L}$.

An *atom* is a formula $p(t_1, \ldots, t_n)$, where p is a predicate symbol and each t_i is a term. If each of the t_is are ground then the atom is said to be ground. The Herbrand Base of a language $\mathcal{L}$,

denoted by $HB_{\mathcal{L}}$, is the set of all ground atoms that can be formed with predicates from $\mathcal{L}$ and terms from $HU_{\mathcal{L}}$. A *literal* is either an atom or an atom preceded by the symbol $\neg$, and is referred to as ground if the atom in it is ground. The former is referred to as a positive literal, while the later is referred to as a negative literal. A *naf-literal* is either an atom or an atom preceded by the symbol **not** . The former is referred to as a positive naf-literal, while the later is referred to as a negative naf-literal. A *gen-literal* is either a literal or a literal preceded by the symbol **not** .

Example 1 Consider an alphabet with variables X and Y, object constants a, b, function symbol f of arity 1, and predicate symbols p of arity 1. Let $\mathcal{L}_1$ be the language defined by this alphabet.

$f(X)$ and $f(f(Y))$ are examples of terms, while $f(a)$ is an example of a ground term. $p(f(X))$ and $p(Y)$ are examples of atoms, while $p(a)$a and $p(f(a))$ are examples of ground atoms.

The Herbrand Universe of $\mathcal{L}_1$ is the set $\{a, b, f(a), f(b), f(f(a)), f(f(b)), f(f(f(a))), f(f(f(b))), \ldots\}$.

The Herbrand Base of $\mathcal{L}_1$ is the set
$\{p(a), p(b), p(f(a)), p(f(b)), p(f(f(a))), p(f(f(b))), p(f(f(f(a)))), p(f(f(f(b)))), \ldots\}$. □

A *rule* is of the form:

$$L_0 \; or \; \ldots \; or \; L_k \leftarrow L_{k+1} \,\&\, \ldots \,\&\, L_m \,\&\, \mathbf{not}\; L_{m+1} \,\&\, \ldots \,\&\, \mathbf{not}\; L_n. \tag{1.2.1}$$

where L_i's are literals or when $k = 0$, L_0 may be the symbol $\bot$, and $k \geq 0$, $m \geq k$, and $n \geq m$. A rule is said to be ground if all the literals of the rule are ground. The parts on the left and on the right of "$\leftarrow$" are called the *head (or conclusion)* and the *body (or premise)* of the rule, respectively. A rule with an empty body is called a *fact*, and then if $L_0, \ldots, L_k$ are ground literals then we refer to it as a ground fact. When $k = 0$, and $L_0 = \bot$, we refer the rule as a *constraint*. Often the connective & in the body of a rule is replaced by a comma. In that case the rule (1.2.1) is written as:

$$L_0 \; or \; \ldots \; or \; L_k \leftarrow L_{k+1}, \ldots, L_m, \mathbf{not}\; L_{m+1}, \ldots, \mathbf{not}\; L_n. \tag{1.2.2}$$

Also, the $\bot$ in the head of constraints are often eliminated and simply written as rules with empty head, as in

$$\leftarrow L_1, \ldots, L_m, \mathbf{not}\; L_{m+1}, \ldots, \mathbf{not}\; L_n. \tag{1.2.3}$$

Definition 1 The *answer-set language* given by an alphabet consists of the set of all ground rules constructed from the symbols of the alphabet. □

It is easy to see that the language given by an alphabet is uniquely determined by its constants O, function symbols F, and predicate symbols P. This triple $\sigma = (O, F, P)$ is referred to as the *signature* of the answer-set theory and often we describe a language by just giving its signature.

1.2.1 AnsProlog* Programs

An AnsProlog* program is a finite set of rules of the form (1.2.2). 'AnsProlog' is a short form for *Answer set programming in logic*, and the '*' denotes that we do not place any restrictions on the rules.

With each AnsProlog* program Π, when its language is not otherwise specified, we associate the language $\mathcal{L}(\Pi)$ that is defined by the predicates, functions and constants occurring in Π. If no

constant occurs in Π, we add some constants to $\mathcal{L}(\Pi)$ for technical reasons. Unless stated otherwise, we use the simplified notation HU_Π and HB_Π instead of $HU_{\mathcal{L}(\Pi)}$ and $HB_{\mathcal{L}(\Pi)}$, respectively. When the context is clear we may just use HU and HB, without the subscripts.

Example 2 Consider the following AnsProlog* program Π:

$p(a)$.
$p(b)$.
$p(c)$.
$p(f(X)) \leftarrow p(X)$.

$\mathcal{L}(\Pi)$ is then the language defined by the predicate p, function f, and constants a, b, and c.

HU_Π is the set $\{a, b, c, f(a), f(b), f(c), f(f(a)), f(f(b)), f(f(c)), f(f(f(a))), f(f(f(b))), f(f(f(c))), \ldots\}$.

HB_Π is the set $\{p(a), p(b), p(c), p(f(a)), p(f(b)), p(f(c)), p(f(f(a))), p(f(f(b))), p(f(f(c))),$
$p(f(f(f(a)))), p(f(f(f(b)))), p(f(f(f(c)))), \ldots\}$. □

Through out this book we consider several distinct subclasses of AnsProlog* programs. The important ones are:

- AnsProlog program: A set of rules where L_i's are atoms and $k = 0$. This is the most popular subclass, and to make it easier to write and refer, it does not have a superscript.
  Such programs are syntactically[2] referred to as *general logic programs* and *normal logic programs* in the literature. The program in Example 2 is an AnsProlog program.

  Example 3 Following is an example of another AnsProlog program from which we can conclude that *tweety* flies while *skippy* is abnormal and it does not fly.

  $fly(X) \leftarrow bird(X), \textbf{not } ab(X)$.
  $ab(X) \leftarrow penguin(X)$.
  $bird(X) \leftarrow penguin(X)$.
  $bird(tweety) \leftarrow$.
  $penguin(skippy) \leftarrow$. □

- AnsProlog$^{-\textbf{not}}$ program: A set of rules where L_i's are atoms, $k = 0$, and $m = n$.
  Such programs are referred to as *definite programs* and *Horn logic programs* in the literature.

  Example 4 Following is an example of an AnsProlog$^{-\textbf{not}}$ program from which we can make conclusions about the *ancestor* relationship between the constants a, b, c, d, and e, for the particular parent relationship specified in the program:

  $anc(X, Y) \leftarrow par(X, Y)$.
  $anc(X, Y) \leftarrow par(X, Z), anc(Z, Y)$.
  $par(a, b) \leftarrow$.
  $par(b, c) \leftarrow$.
  $par(d, e) \leftarrow$.

  The first two rules of the above program can be used to define the ancestor relationship over an arbitrary set of *parent* atoms. This is an example of 'transitive closure' and in general it can not be specified using first-order logic. □

[2] AnsProlog programs also denote a particular semantics, while several different semantics may be associated with general logic programs.

- AnsProlog$^{\neg}$ program: A set of rules where $k = 0$.
  Such programs are syntactically referred to as *extended logic programs* in the literature.

  Example 5 Following is an example of an AnsProlog$^{\neg}$ program from which we can conclude that *tweety* flies while *rocky* does not.

  $fly(X) \leftarrow bird(X), \textbf{not } \neg fly(X).$
  $\neg fly(X) \leftarrow penguin(X).$
  $bird(tweety) \leftarrow.$
  $bird(rocky) \leftarrow.$
  $penguin(rocky) \leftarrow.$ □

- AnsProlog^{or} program: A set of rules where L_i's are atoms.
  Such programs are syntactically referred to as *normal disjunctive logic programs* in the literature. A subclass of it where $m = n$ is syntactically referred to as *disjunctive logic programs.*

  Example 6 Following is an example of an AnsProlog^{or} program from which we can conclude that *slinky* is either a bird or a reptile but not both.

  $bird(X)\ or\ \ reptile(X) \leftarrow lays\_egg(X).$
  $lays\_egg(slinky) \leftarrow.$ □

- In each of the above classes if we allow constraints (i.e., rules with $\bot$ in the head) then we have AnsProlog$^{\bot}$, AnsProlog$^{\neg\textbf{not},\bot}$, AnsProlog$^{\neg,\bot}$, and AnsProlog$^{or,\bot}$ programs respectively.

- AnsProlog$^{\neg, or, \bot}$ program: It is same as an AnsProlog* program.

- AnsDatalog program: An AnsProlog program, with the restriction that the underlying language does not have function symbols. The programs in Example 3 and Example 4 are also AnsDatalog programs, while the program in Example 2 is not an AnsDatalog program.

- AnsDatalog^{X} program, $X \in$ { '$\neg$**not**', '*', '$\neg$', '*or*', '$\neg$, *or*', '$\neg$**not**, $\bot$', '$\neg, \bot$', '*or*, $\bot$', '$\neg$, *or*, $\bot$' }: An AnsProlog^{X} program, with the restriction that the underlying language does not have function symbols. The programs in Example 4, Example 5 and Example 6 are examples of AnsDatalog$^{\neg\textbf{not}}$, AnsDatalog$^{\neg}$ and AnsDatalog^{or} programs, respectively.

- Propositional Y program, where Y is one of the above classes: A program from the class Y with the added restriction that all the predicates are of arity 0, i.e., all atoms are propositional ones. An example of a propositional AnsProlog program is the program $\{a \leftarrow \textbf{not } b.b \leftarrow \textbf{not } a.\}$.

- AnsProlog*(n) program: An AnsProlog*(n) program is an AnsProlog* program that has at most n literals in the body of its rules. We can make similar restrictions for other sub-classes of AnsProlog*.

AnsProlog terminology	Earlier terminologies
answer sets (of AnsProlog programs)	stable models
AnsProlog$^{\neg}$**not**	definite programs, Horn logic programs
AnsProlog	general logic programs, normal logic programs (with stable model semantics)
AnsProlog$^{\neg}$	extended logic programs (with answer set semantics)
AnsProlog^{or}	normal disjunctive logic programs (with stable model semantics)
AnsProlog$^{\neg}$**not**, or	disjunctive logic programs
AnsDatalog$^{\neg}$**not**	Datalog
AnsDatalog	Datalog**not** (with stable model semantics)
AnsDatalog^{or}	Datalog**not** , or (with stable model semantics)

1.2.2 AnsProlog* notations

Following is a list of additional notations that will be used in the rest of this book, particularly, in the rest of this chapter.

- Given a rule r of the form (1.2.2):
  - $head(r) = \{L_0, \ldots, L_k\}$,
  - $pos(r) = \{L_{k+1}, \ldots, L_m\}$,
  - $neg(r) = \{L_{m+1}, \ldots, L_n\}$ and
  - $lit(r) = head(r) \cup pos(r) \cup neg(r)$.
- For any program Π, $Head(\Pi) = \bigcup_{r \in \Pi} head(r)$.
- Various notations for sets of atoms.
  - For a predicate p, $atoms(p)$ will denote the subset of HB_Π formed with predicate p.
  - For a set of predicates A, $atoms(A)$ will denote the subset of HB_Π formed with the predicates in A.
  - For a list of predicates $p_1, \ldots, p_n$, $atoms(p_1, \ldots, p_n)$ denotes the set of atoms formed with predicates $p_1, \ldots, p_n$.
  - For a signature σ, $atoms(\sigma)$denote the set of atoms over σ.
  - Given a set of naf-literals A, $atoms(A)$ denotes the set $\{a : a \text{ is an atom, and } a \in A\} \cup \{a : a \text{ is an atom, and } \mathbf{not}\ a \in A\}$.
- Various notations for sets of literals.
  - For a program Π, $lit(\Pi) = \bigcup_{r \in \Pi} lit(r)$.
  - For a predicate p, $Lit(p)$ denotes the collection of ground literals formed by the predicate p.
  - For a language $\mathcal{L}$, $Lit(\mathcal{L})$ denotes the set of all literals in $\mathcal{L}$.
  - For a program Π, Lit_Π denotes the set of all literals in its associated language; and when the context is clear we may just use Lit.

  - For a list of predicates $p_1, \ldots, p_n$, $lit(p_1, \ldots, p_n)$ denotes the set of literals formed with predicates $p_1, \ldots, p_n$.
  - For a signature σ, $lit(\sigma)$ denote the set of literals over σ.

- Let r be a rule in a language $\mathcal{L}$. The grounding of r in $\mathcal{L}$, denoted by $ground(r, \mathcal{L})$, is the set of all rules obtained from r by all possible substitution of elements of $HU_{\mathcal{L}}$ for the variables in r. For any logic program Π, we define

$$ground(\Pi, \mathcal{L}) = \bigcup_{r \in \Pi} ground(r, \mathcal{L})$$

  and write $ground(\Pi)$ for $ground(\Pi, \mathcal{L}(\Pi))$.

  Example 7 Consider the program Π from Example 2. The program $ground(\Pi)$ consists of the following rules:

  $p(a) \leftarrow$.
  $p(b) \leftarrow$.
  $p(c) \leftarrow$.
  $p(f(a)) \leftarrow p(a)$.
  $p(f(b)) \leftarrow p(b)$.
  $p(f(c)) \leftarrow p(c)$.
  $p(f(f(a))) \leftarrow p(f(a))$.
  $p(f(f(b))) \leftarrow p(f(b))$.
  $p(f(f(c))) \leftarrow p(f(c))$.
  $\vdots$ □

- Signature $\sigma_1 = \{O_1, F_1, P_1\}$ is said to be a sub-signature of signature $\sigma_2 = \{O_2, F_2, P_2\}$ if $O_1 \subseteq O_2$, $F_1 \subseteq F_2$ and $P_1 \subseteq P_2$.

  - $\sigma_1 + \sigma_2$ denotes the signature $\{O_1 \cup O_2, F_1 \cup F_2, P_1 \cup P_2\}$.
  - The sets of all ground terms over signature σ are denoted by $terms(\sigma)$.
  - Consistent sets of ground literals over signature σ are called *states* of σ and denoted by $states(\sigma)$.

- For any literal l, the symbol $\bar{l}$denotes the literal opposite in sign to l. i.e. for an atom a, if $l = \neg a$ then $\bar{l} = a$, and if $l = a$ then $\bar{l} = \neg a$. Moreover, we say l and $\bar{l}$ are *complementary* or *contrary* literals.

  Similarly for a literal l, $not(l)$ denotes the gen-literal **not** l, while $not(\textbf{not } l)$ denotes l.

- For a set of literals S, by $\bar{S}$ we denote the set $HB \setminus S$.

- For a set of literals S, by $\neg S$ we denote the set $\{\bar{l} \ : \ l \in S\}$.

- Two sets of literals S_1 and S_2 are said to disagree if $S_1 \cap \neg S_2 \neq \emptyset$. Otherwise we say that they agree.

- Given a set of literals L and an AnsProlog* program Π, by $\Pi \cup L$ we mean the AnsProlog* program $\Pi \cup \{l \leftarrow . \ : \ l \in L\}$.

- Let Π be an AnsProlog program, A be a set of naf-literals, and B be a set of atoms. B is said to *agree* with A, if $\{a : a \text{ is an atom, and } a \in A\} \subseteq B$ and $\{a : a \text{ is an atom, and } \mathbf{not}\ a \in A\} \cap B = \emptyset$.

- A set S of literals is said to be *complete* w.r.t. a set of literals P if for any atom in P either the atom or its negation is in S. When $P = S$ or P is clear from the context, we may just say S is complete.

- A set X of literals is said to be *saturated* if every literal in X has its complement in X.

- A set X of literals is said to be *supported by* an AnsProlog$^{\neg,\bot}$ program Π, if every literal L in X there is a rule in Π with L in its head and $L_1, \ldots, L_m, \mathbf{not}\ L_{m+1}, \mathbf{not}\ L_n$ as its body such that $\{L_1 \ldots, L_m\} \subseteq X$ and $\{L_{m+1}, \ldots, L_n\} \cap X = \emptyset$.

- A rule is said to be *range restricted (or allowed)* if every variable occurring in a rule of the form 1.2.1 occurs in one of the literals $L_{k+1}, \ldots, L_m$. In presence of built-in comparative predicates such as *equal*, *greater than*, etc., the variables must occur in a non-built-in literal among $L_{k+1}, \ldots, L_m$. A program Π is range restricted (or allowed) if every rule in Π is range restricted.

  The programs in Examples 2– 6 are all range restricted. The program consisting of the following rules is not range restricted, as its first rule has the variable X in the head which does not appear in its body at all.

  $p(X) \leftarrow q.$
  $r(a) \leftarrow.$

  The program consisting of the following rules is also not range restricted, as its first rule has the variable Y, which appears in $\mathbf{not}\ r(X, Y)$ in the body, but does not appear in a positive naf-literal in the body.

  $p(X) \leftarrow q(X), \mathbf{not}\ r(X, Y).$
  $r(a, b) \leftarrow.$
  $q(c) \leftarrow.$

1.3 Semantics of AnsProlog* programs

In this section we define the semantics of AnsProlog* programs. For that, we first define the notion of answer sets for the various subclasses, and then define query languages appropriate for the various subclasses and define the entailment between programs and queries. While defining the answer sets we start with the most specific sub-class and gradually consider the more general sub-classes.

The answer sets of an AnsProlog* program Π, is defined in terms of the answer sets of the ground program $ground(\Pi)$. Hence, in the rest of the section we assume that we are only dealing with ground programs.

1.3.1 Answer sets of AnsProlog$^{-\mathbf{not}}$ and AnsProlog$^{-\mathbf{not},\perp}$ programs

AnsProlog$^{-\mathbf{not}}$ programs form the simplest class of declarative logic programs, and its semantics can be defined in several ways. We present two of them here, and refer to [Llo84, Llo87, LMR92] for other characterizations. In particular, we present a model theoretic characterization and a fixpoint characterization.

Model theoretic characterization

A *Herbrand interpretation* of an AnsProlog$^{\perp}$ program Π is any subset $I \subseteq HB_\Pi$ of its Herbrand base. Answer sets are defined as particular Herbrand interpretations that satisfy certain properties with respect to the program and are 'minimal'. We say an interpretation I is *minimal* among the set $\{I_1, \ldots, I_n\}$ if there does not exist a j, $1 \leq j \leq n$ such that I_j is a strict subset of I. We say an interpretation I is *least* among the set $\{I_1, \ldots, I_n\}$ if for all j, $1 \leq j \leq n$ $I \subseteq I_j$.

A *Herbrand interpretation* S of Π is said to *satisfy* the AnsProlog$^{\perp}$ rule $L_0 \leftarrow L_1, \ldots, L_m,$ **not** $L_{m+1}, \ldots,$ **not** L_n. if
(i) $L_0 \neq \perp$: $\{L_1, \ldots, L_m\} \subseteq S$ and $\{L_{m+1}, \ldots, L_n\} \cap S = \emptyset$ implies that $L_0 \in S$.
(ii) $L_0 = \perp$: $\{L_1, \ldots, L_m\} \not\subseteq S$ or $\{L_{m+1}, \ldots, L_n\} \cap S \neq \emptyset$.

A *Herbrand model* A of an AnsProlog$^{\perp}$ program Π is a Herbrand interpretation S of Π such that it satisfies all rules in Π. We also refer to this as A is closed under Π.

Definition 2 An answer set of an AnsProlog$^{-\mathbf{not},\perp}$ program Π is a Herbrand model of Π, which is minimal among the Herbrand models of Π. □

Example 8 Consider the following AnsProlog$^{-\mathbf{not}}$ program:

$p \leftarrow a.$
$q \leftarrow b.$
$a \leftarrow .$

The set $\{a, b, p, q\}$ is a model of this program as it satisfies all rules of this program. The sets $\{a, p, q\}$ and $\{a, p\}$ are also models of this program. But the set $\{a, b, p\}$ is not a model of this program as it does not satisfy the second rule.

Since $\{a, p\}$ is a model of this program, the sets $\{a, b, p, q\}$ and $\{a, p, q\}$ which are strict supersets of $\{a, p\}$ are not minimal models of this program. None of the sets $\{a\}$, $\{p\}$, and $\{\}$ are models of this program as each of them does not satisfy at least one of the rules of the program. Thus since all the strict subsets of $\{a, p\}$ are not models of this program, $\{a, p\}$ is a minimal model and answer set of the program. □

Example 9 The program Π in Example 4 has an answer set S_1:
$\{par(a, b), par(b, c), par(d, e), anc(a, b), anc(b, c), anc(a, c), anc(d, e)\}$, which is also its unique minimal Herbrand model. It is easy to see that S_1 satisfies all rules of $ground(\Pi)$. Hence, it is a model of Π. We now have to show that it is a minimal model. To show that S_1 is a minimal model, we will show that none of the strict subsets of S_1 are models of $ground(\Pi)$. Suppose we were to remove one of the *par* atoms of Π. In that case it will no longer be a model of $ground(\Pi)$. Now suppose we were to remove $anc(a, b)$ from S_1. The resulting interpretation is not a model of $ground(\Pi)$ as it does not satisfy one of the ground instance of the first rule of Π. The same goes for $anc(b, c)$ and $anc(d, e)$. Hence, we can not remove one of those three and still have a model. Now, if we remove

$anc(a,c)$, it will no longer be a model as it will not satisfy one of the ground instance of the second rule of Π. Hence, S_1 is a minimal model and answer set of $ground(\Pi)$ and therefore of Π.

The set $S_2 = \{par(a,b), par(b,c), par(d,e), anc(a,b), anc(b,c), anc(a,c), anc(d,e), par(d,c), anc(d,c)\}$ is a Herbrand model of $ground(\Pi)$, as it satisfies all rules in $ground(\Pi)$. S_2 is not minimal among the models of Π, as S_1 a model of Π is a strict subset of S_2. Hence, S_2 is also not an answer set of Π.

The set $\{par(a,b), par(b,c), par(d,e), anc(a,b), anc(b,c), anc(a,c), anc(d,e), par(d,c)\}$ is not a Herbrand model of as it does not satisfy the rule:

$anc(d,c) \leftarrow par(d,c)$.

which is one of the ground instance of the first rule of Π. □

The notion of model – although useful – is a relic from the semantics of first-order logic. So alternatively, answer sets can be defined without using the notion of a model in the following way:

Definition 3 An *answer set* of an AnsProlog$^{-\mathbf{not},\bot}$ program Π is a minimal subset (with respect to subset ordering) S of HB that is closed under $ground(\Pi)$. □

Proposition 1 AnsProlog$^{-\mathbf{not}}$ programs have unique answer sets. □

The above is not true in general for AnsProlog$^{-\mathbf{not},\bot}$ programs. For example, the program $\{p \leftarrow ., \bot \leftarrow p.\}$ does not have an answer set. We will denote the answer set of an AnsProlog$^{-\mathbf{not},\bot}$ program Π, if it exists, by $\mathcal{M}_0(\Pi)$. Otherwise, $\mathcal{M}_0(\Pi)$ is undefined.

Proposition 2 The intersection of the Herbrand models of an AnsProlog$^{-\mathbf{not}}$ program is its unique minimal Herbrand model. □

Iterated fixpoint characterization

From a computational viewpoint, a more useful characterization is an iterated fixpoint characterization. To give such a characterization let us assume Π to be a set (finite or infinite) of ground rules. Let 2^{HB_Π} denote the set of all Herbrand interpretations of Π. We define an operator $T^0_\Pi : 2^{HB_\Pi} \rightarrow 2^{HB_\Pi}$ as follows:

$$T^0_\Pi(I) = \{L_0 \in HB_\Pi \mid \Pi \text{ contains a rule } L_0 \leftarrow L_1, \ldots, L_m. \text{ such that } \{L_1, \ldots, L_m\} \subseteq I \text{ holds } \}. \quad (1.3.4)$$

The above operator is referred to as the *immediate consequence operator*. Intuitively, $T^0_\Pi(I)$ is the set of atoms that can be derived from a single application of Π given the atoms in I.

We will now argue that T^0_Π is monotone, i.e., $I \subseteq I' \Rightarrow T^0_\Pi(I) \subseteq T^0_\Pi(I')$. Suppose X is an arbitrary element of $T^0_\Pi(I)$. Then there must be a rule $X \leftarrow L_1, \ldots, L_m.$ in Π such that $\{L_1, \ldots, L_m\} \subseteq I$. Since $I \subseteq I'$, we have that $\{L_1, \ldots, L_m\} \subseteq I'$. Hence, X must be in $T^0_\Pi(I')$. Therefore, $T^0_\Pi(I) \subseteq T^0_\Pi(I')$.

Now, let us denote the empty set by $T^0_\Pi \uparrow 0$. Let us also denote $T^0_\Pi \uparrow (i+1)$ to be $T^0_\Pi(T^0_\Pi \uparrow i)$. Clearly, $T^0_\Pi \uparrow 0 \subseteq T^0_\Pi \uparrow 1$; and by monotonicity of T^0_Π and transitivity of $\subseteq$, we have $T^0_\Pi \uparrow i \subseteq T^0_\Pi \uparrow (i+1)$. In case of a finite Herbrand base it can be easily seen that repeated application of T^0_Π starting from the empty set will take us to a fixpoint of T^0_Π. We will now argue that this fixpoint – let us refer to

it as a – that is reached is the least fixpoint of T^0_Π. Suppose it is not the case. Then there must be a different fixpoint b. Since b is the least fixpoint and a is only a fixpoint, $b \subseteq a$. Since $\emptyset \subseteq b$, by using the monotonicity property of T^0_Π and by repeatedly applying T^0_Π to both sides we will obtain $a \subseteq b$. Thus $a = b$, contradicting our assumption that b is different from a. Hence, a must be the least fixpoint of T^0_Π.

In case of an infinite Herbrand base, the case is similar and we refer to Appendix A for the exact arguments. We can summarize the result from Appendix A as that the operator T^0_Π satisfies a property called *continuity*, and the ordering $\subseteq$ over the elements in 2^{HB_Π} is a *complete lattice*, both of which guarantee that iterative application of T^0_Π starting from the empty set will take us to the least fixpoint of T^0_Π. More formally, $lfp(T^0_\Pi) = T^0_\Pi \uparrow \omega$ = least upper bound of the set $\{T^0_\Pi \uparrow \beta \ : \ \beta < \omega\}$, where ω is the first limit ordinal.

An AnsProlog$^{\neg\mathbf{not}}$ program Π can now be characterized by its least fixpoint. Recall that we assumed Π to be a set (finite or infinite) of ground rules. When this is not the case, and Π is non-ground, we characterize Π by the least fixpoint of the program $ground(\Pi)$. It can be shown that $lfp(T^0_\Pi)$ is also the unique minimal Herbrand model of Π.

Proposition 3 For any AnsProlog$^{\neg\mathbf{not}}$ program Π, $lfp(T^0_\Pi)$ = the unique minimal Herbrand model of Π = the answer set of Π. □

We now give two examples showing how the answer set of AnsProlog$^{\neg\mathbf{not}}$ programs can be computed by the iterated fixpoint method.

Example 10 Consider the following program Π from Example 8.

$p \leftarrow a.$
$q \leftarrow b.$
$a \leftarrow .$

By definition, $T^0_\Pi \uparrow 0 = \emptyset$.
$T^0_\Pi \uparrow 1 = T^0_\Pi(T^0_\Pi \uparrow 0) = \{a\}$.
$T^0_\Pi \uparrow 2 = T^0_\Pi(T^0_\Pi \uparrow 1) = \{a, p\}$.
$T^0_\Pi \uparrow 3 = T^0_\Pi(T^0_\Pi \uparrow 2) = \{a, p\} = T^0_\Pi \uparrow 2$.

Hence $lfp(T^0_\Pi) = \{a, p\}$, and therefore $\{a, p\}$ is the answer set of Π.

Example 11 Let us now consider the program Π from Example 4.

By definition, $T^0_\Pi \uparrow 0 = \emptyset$.
$T^0_\Pi \uparrow 1 = T^0_\Pi(T^0_\Pi \uparrow 0) = \{par(a,b), par(b,c), par(d,e)\}$.
$T^0_\Pi \uparrow 2 = T^0_\Pi(T^0_\Pi \uparrow 1) = \{par(a,b), par(b,c), par(d,e), anc(a,b), anc(b,c), anc(d,e)\}$.
$T^0_\Pi \uparrow 3 = T^0_\Pi(T^0_\Pi \uparrow 2) = \{par(a,b), par(b,c), par(d,e), anc(a,b), anc(b,c), anc(d,e), anc(a,c)\}$.
$T^0_\Pi \uparrow 4 = T^0_\Pi(T^0_\Pi \uparrow 3) = T^0_\Pi \uparrow 3$.

Hence $lfp(T^0_\Pi) = \{par(a,b), par(b,c), par(d,e), anc(a,b), anc(b,c), anc(d,e), anc(a,c)\}$, is the answer set of Π.

Exercise 1 Given an AnsProlog$^{\neg\mathbf{not}}$ program Π

1. Show that 2^{HB_Π} is a complete lattice with respect to the relation $\subseteq$.
2. Show that T^0_Π is continuous.

□

1.3.2 Answer sets of AnsProlog and AnsProlog$^{\perp}$ programs

AnsProlog programs are a superclass of AnsProlog$^{\neg\mathbf{not}}$ programs in that they allow the operator **not** in the body of the rules. In the literature of logic programming there are several different semantics for programs having the same syntax has AnsProlog. In this book our focus is on one particular semantics, the *answer-set semantics*. Before defining the answer sets we show why the approach of minimal models and iterated fixpoints used in defining answer sets AnsProlog$^{\neg\mathbf{not}}$ programs can not be directly used in defining answer sets of AnsProlog programs.

The answer is that AnsProlog programs may have multiple minimal models, and intuitively not all of them may make sense. As per the iterated fixpoint approach, the direct extension of the operator T^0_Π is not monotone, and hence its repeated application starting from the empty set may not lead to a fixpoint. The following examples illustrate these two points.

Example 12 Consider the program Π consisting of the only rule:

$a \leftarrow$ **not** b.

This program has two minimal models $\{a\}$ and $\{b\}$. But the second one is not intuitive, as there is no justification for why b should be true. □

Example 13 Consider the program Π consisting of the following rules:

$a \leftarrow$ **not** b.
$b \leftarrow$ **not** a.

Let us now consider the obvious extension of the T^0_Π operator to AnsProlog programs. This extension, which we will refer to as T^1_Π is defined as follows:

$T^1_\Pi(I) = \{L_0 \in HB_\Pi \mid \Pi$ contains a rule $L_0 \leftarrow L_1, \ldots, L_m,$ **not** $L_{m+1}, \ldots,$ **not** L_n.
such that $\{L_1, \ldots, L_m\} \subseteq I$ holds, and $\{L_{m+1}, \ldots, L_n\} \cap I = \emptyset\}$.

Now, $T^1_\Pi \uparrow 0 = \emptyset$.
$T^1_\Pi \uparrow 1 = T^1_\Pi(T^1_\Pi \uparrow 0) = \{a, b\}$.
$T^1_\Pi \uparrow 2 = T^1_\Pi(T^1_\Pi \uparrow 1) = \emptyset$.
$T^1_\Pi \uparrow 3 = T^1_\Pi(T^1_\Pi \uparrow 2) = \{a, b\}$.

Thus the above sequence oscillates between $\emptyset$ and $\{a, b\}$, and never reaches a fixpoint. Moreover, while $\emptyset \subseteq \{a, b\}$, $T^1_\Pi(\emptyset) \not\subseteq T^1_\Pi(\{a, b\})$. I.e., T^1_Π is not a monotone operator. □

Answer sets of AnsProlog$^{\perp}$ programs are defined using a fixpoint definition. Given a candidate answer set S for an AnsProlog$^{\perp}$ program Π, we first transform Π with respect to S and obtain an AnsProlog$^{\neg\mathbf{not},\perp}$ program denoted by Π^S. S is defined as an answer set of Π, if S is the answer set of the transformed AnsProlog$^{\neg\mathbf{not},\perp}$ program Π^S. This transformation is referred to as the Gelfond-Lifschitz transformation, as it was originally defined by Gelfond and Lifschitz in their paper on the stable model semantics. More formally,

Definition 4 Let Π be a ground AnsProlog$^{\perp}$ program. For any set S of atoms, let Π^S be a program obtained from Π by deleting

(i) each rule that has a formula **not** L in its body with $L \in S$, and

(ii) all formulas of the form **not** L in the bodies of the remaining rules.

Clearly, Π^S does not contain **not** , so it is an AnsProlog$^{-\mathbf{not},\perp}$ program and its answer set is already defined. If this answer set coincides with S, then we say that S is an *answer set* of Π. In other words, an answer set of Π is characterized by the equation

$$S = \mathcal{M}_0(\Pi^S).$$ □

We now illustrate the above definition through several examples.

Example 14 Consider the following program Π:

$p \leftarrow a.$
$a \leftarrow$ **not** $b.$
$b \leftarrow$ **not** $a.$

We will now show that $S_1 = \{p, a\}$ and $S_2 = \{b\}$ are answer sets of Π.

$\Pi^{S_1} = \{p \leftarrow a., a \leftarrow .\}$, and the answer set of Π^{S_1} is S_1. Hence, S_1 is an answer set of Π.

$\Pi^{S_2} = \{p \leftarrow a., b \leftarrow .\}$, and the answer set of Π^{S_2} is S_2. Hence, S_2 is an answer set of Π.

To illustrate why, for example, $S = \{a, b\}$ is not an answer set of Π, let us compute Π^S. $\Pi^S = \{p \leftarrow a.\}$, and the answer set of Π^S is $\emptyset$, which is different from S. Hence, S is not an answer set of Π. □

Example 15 Consider the following program Π:

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

We will now show that $S_1 = \{d, b\}$ is an answer set of Π.

$\Pi^{S_1} = \{b \leftarrow ., d \leftarrow .\}$, and the answer set of Π^{S_1} is S_1. Hence, S_1 is an answer set of Π.

To illustrate why, for example, $S = \{a, d\}$ is not an answer set of Π, let us compute Π^S. $\Pi^S = \{a \leftarrow ., b \leftarrow ., d \leftarrow .\}$, and the answer set of Π^S is $\{a, b, d\}$, which is different from S. Hence, S is not an answer set of Π. □

Example 16 Consider the following program Π:

$p \leftarrow p.$
$q \leftarrow .$

The only answer set of this program is $\{q\}$, which is also the unique minimal model of this AnsProlog$^{-\mathbf{not}}$ program. □

Example 17 Consider the following program Π:

$p \leftarrow$ **not** $p, d.$
$r \leftarrow.$
$d \leftarrow .$

The above program does not have an answer set. Intuitively, since r and d must be any answer set, the two possible choices for answer sets of this program are $S_1 = \{r, d, p\}$ and $S_2 = \{r, d\}$. The program $\Pi^{S_1} = \{r \leftarrow ., d \leftarrow\}$ and it has the answer set $\{r, d\}$, which is different from S_1. Hence,

S_1 is not an answer set of Π. Similarly, the program $\Pi^{S_2} = \{p \leftarrow d., r \leftarrow ., d \leftarrow\}$ and it has the answer set $\{r, d, p\}$, which is different from S_2. Hence, S_2 is not an answer set of Π.

One of the early criticisms of the answer set semantics was about the characterization of the above program. But now it is realized that the above characterization is useful in expressing constraints. For example, consider the following program:

$p \leftarrow$ **not** $p, d.$
$r \leftarrow$ **not** $d.$
$d \leftarrow$ **not** $r.$

The only answer set of this program is $\{r\}$. The first rule acts like a constraint which eliminates any candidate answer set that has d true. Thus, even though the last two rules have two answer sets $\{r\}$ and $\{d\}$, the second one is eliminated by the constraint like behavior of the first rule.

An alternative approach is to replace the first rule above by the two rules in $\{p \leftarrow$ **not** $p., p \leftarrow$ **not** $d\}$. The resulting program is:

$p \leftarrow$ **not** $p.$
$p \leftarrow$ **not** $d.$
$r \leftarrow$ **not** $d.$
$d \leftarrow$ **not** $r.$

In the above program, any candidate answer set where d is $false$, forces p to be true by the second rule, and this makes the first rule ineffective in eliminating that answer set. On the other hand, any candidate answer set where d is $true$ can no longer use the second rule to force p to be true, and thus the first rule is effective in eliminating that answer set. Thus in the above program the program consisting of the last two rules has the answer sets $\{r\}$ and $\{d\}$. The first one forces p to be true by the second rule, and thus $\{r, p\}$ is an answer set of the program. The second one, on the other hand, is eliminated by the first rule, as the second rule does not come to its rescue. Thus the above program has the only answer set $\{r, p\}$. This program has some historical significance as it was used in [VG88] and later by others to argue that the answer set semantics is unintuitive. But now that we understand the role of the rule $p \leftarrow$ **not** $p.$ in constraint enforcement, the answer set characterization of the above program is quite meaningful. □

Example 18 Consider the following program Π:

$p \leftarrow$ **not** $q, r.$
$q \leftarrow$ **not** $p.$

If the above program is presented to a PROLOG interpreter and query about p or q is asked, the PROLOG interpreter will not be able to give an answer, and may get into an infinite loop. But intuitively, since there is no rule with r in its head, there is no way r can be proven to be true and hence r can be assumed to be $false$. That means, there is no way p can be proven true, as the first rule is the only one using which p can be proven true, and that rule has r in its body. Thus, p can be assumed to be $false$. That in turn means, we can infer q to be true using the second rule. The answer set semantics, captures the above reasoning, and the above program has the unique answer set $\{q\}$, which is the answer set of $\Pi^{\{q\}} = \{q \leftarrow .\}$. □

Example 19 Consider Π, the ground version of the program from Example 3 given below:

$fly(tweety) \leftarrow bird(tweety),$ **not** $ab(tweety).$
$ab(tweety) \leftarrow penguin(tweety).$

$bird(tweety) \leftarrow penguin(tweety).$
$fly(skippy) \leftarrow bird(skippy), \textbf{not}\ ab(skippy).$
$ab(skippy) \leftarrow penguin(skippy).$
$bird(skippy) \leftarrow penguin(skippy).$
$bird(tweety) \leftarrow.$
$penguin(skippy) \leftarrow.$

We will show that the above program has $S =$
$\{bird(tweety), penguin(skippy), bird(skippy), ab(skippy), fly(tweety)\}$ as an answer set.

The AnsProlog^{-}**not** program Π^S consists of the following rules:

$fly(tweety) \leftarrow bird(tweety).$
$ab(tweety) \leftarrow penguin(tweety).$
$bird(tweety) \leftarrow penguin(tweety).$
$ab(skippy) \leftarrow penguin(skippy).$
$bird(skippy) \leftarrow penguin(skippy).$
$bird(tweety) \leftarrow.$
$penguin(skippy) \leftarrow.$

Since Π^S is an AnsProlog^{-}**not** program, we will compute its answer set using the iterated fixpoint approach.

$T^0_{\Pi^S} \uparrow 0 = \emptyset.$
$T^0_{\Pi^S} \uparrow 1 = T^0_{\Pi^S}(T^0_{\Pi^S} \uparrow 0) = \{bird(tweety), penguin(skippy)\}.$
$T^0_{\Pi^S} \uparrow 2 = T^0_{\Pi^S}(T^0_{\Pi^S} \uparrow 1) = \{bird(tweety), penguin(skippy), bird(skippy), ab(skippy), fly(tweety)\}.$
$T^0_{\Pi^S} \uparrow 3 = T^0_{\Pi^S}(T^0_{\Pi^S} \uparrow 2) = T^0_{\Pi^S} \uparrow 2 = S.$

Hence $lfp(T^0_{\Pi^S}) = S$, is the answer set of Π^S. Thus, S is an answer set of Π. □

Definition 4 defines when an interpretation is an answer set. Hence, using it we can only verify if a particular interpretation is an answer set or not. To show that a particular answer set of a program is the only answer set of that program we have to rule out the other interpretations. In Chapter 3 we discuss certain conditions such as acyclicity, stratification, and local stratification, which guarantee that an AnsProlog program has a unique answer set. The program in Example 19 is locally stratified and hence has a unique answer set.

Theorem 1.3.1 Answer sets of AnsProlog programs are also minimal Herbrand models □

Proof:
Let Π be an AnsProlog program and A be an answer set of Π. To show A is a minimal model of Π, we will first show that A is a model of Π.

By definition of an answer set A is a model of Π^A. It is easy to see that the body of any rule in Π that was removed during the construction of Π^A, evaluates to false with respect to A. Hence A satisfies those rules. It is also clear that A satisfies any rule in Π that remained in Π^A with possibly some changes. Hence, A is a model of Π.

Now we will show that no subset of A is a model of Π. Suppose there is a strict subset A' of A which is a model of Π. That means A' satisfies all rules in Π. Let us now consider any rule r' in Π^A. If r' is in Π then of course A' satisfies r'. Otherwise r' must come from r in Π, where r has negative literals of the form **not** p, where $p \notin A$. Since $A' \subset A$, $p \notin A'$. Thus these negative literals

evaluate to true both with respect A and A'. Thus r is satisfied by A implies r' is satisfied by A'. Therefore A' is a model of Π^A, which contradicts with A being the minimal model of Π^A. Hence, A must be a minimal model of Π. □

The following proposition gives an alternative characterization of answer sets of AnsProlog that is often useful.

Proposition 4 M is an answer set of an AnsProlog program Π iff M is a model of Π and for all M', M' is a model of Π^M implies $M \subseteq M'$. □

Proof:
M is an answer set of Π iff
M is the answer set of Π^M iff
M is the least model of Π^M iff
M is a model of Π^M and for all M', M' is a model of Π^Mimplies $M \subseteq M'$ iff
M is a model of Π and for all M', M' is a model of Π^Mimplies $M \subseteq M'$. □

1.3.3 Answer sets of AnsProlog$^{\neg}$ and AnsProlog$^{\neg,\bot}$ programs

AnsProlog programs provide negative information implicitly, through closed-world reasoning; they do not leave the user with a choice on that matter. For example, consider the program Π from Example 3. Suppose we were to observe that *tweety* is unable to fly. *We do not have a direct way to add this information to* Π. An indirect way would be to add either, $penguin(tweety) \leftarrow$. or $ab(tweety) \leftarrow$. to Π.

Since an answer set of an AnsProlog program is a subset of the Herbrand base, an atom is either *true* with respect to it (i.e., it belongs to the answer set) or $false$ with respect to it (i.e., it does not belongs to the answer set). Sometimes we may not want to commit either way. For example, we may want to encode the information that "normally we can not make a *true*-$false$ conclusion about whether a wounded bird flies or not". We would be hard pressed to express this in AnsProlog.

For both examples above, what we need are rules that express when a bird normally flies and when it does not, and rules that can block the application of either or both. If such rules were allowed then in the first case we would just add the direct fact that "tweety does not fly". The question then is how do we directly express "tweety does not fly". We can not use the negation as failure operator "**not** " as it means "false by default or by assumption". What we need is an explicit negation, similar to the one used in classical logic. In presence of such a negation operator we would be able to express "tweety does not fly" by $\neg fly(tweety) \leftarrow$.

To further illustrate the intuitive difference between the negation as failure operator **not** and the explicit or classical negation operator $\neg$, let us try to represent the knowledge [3] that it is safe for a school bus to cross a railway intersection if there is no train coming. In the absence of the classical negation operator, we would write this as

cross $\leftarrow$ **not** *train*.

But this rule may be dangerous. Suppose because of fog the sensors can not figure out if there is a train coming or not and hence the program does not have the fact *train* $\leftarrow$. in it. The above rule would then entail that it is safe to cross the road, and this may lead to disaster if there was

[3] This example appears in [GL91] and is attributed to McCarthy.

actually a train coming. A safer alternative would be to use the classical negation operator $\neg$ and express the rule as:

$cross \leftarrow \neg train.$

In that case, the conclusion of crossing the intersection will only be made, if the program can derive $\neg train$, which in presence of a direct sensor means that the sensor has added the fact $\neg train$ to the program. In this case if there is fog, and the sensor adds neither $train \leftarrow .$ nor $\neg train \leftarrow .$, then the conclusion to *cross* will not be made.

These are some of the motivations behind the language AnsProlog$^{\neg}$ that extends AnsProlog with the operator $\neg$. In addition, since an AnsProlog$^{\neg}$ program can include explicit negative information, instead of the embedded 'closed world assumption' in AnsProlog programs, they have the 'open world assumption'. Thus they remove the bias towards negative information present in AnsProlog programs, and treat positive and negative information at par. Nevertheless, by using the negation-as-failure operator '**not** ' a user can explicitly state – by a rule of the form $\neg p \leftarrow$ **not** $p.$ – if certain negative information is to be inferred through closed world reasoning. A user can also do the opposite – by a rule of the form $p \leftarrow$ **not** $\neg p.$, and explicitly state that for certain predicates positive information will be inferred through a form of closed world reasoning.

We now define the answer sets of AnsProlog$^{\neg,\bot}$ programs. For that we first consider AnsProlog$^{\neg,}$**not**$^{,\bot}$ programs.

A *partial Herbrand interpretation* of an AnsProlog$^{\neg,\bot}$ program Π is any subset $I \subseteq Lit$. A partial Herbrand interpretation S of Π is said to *satisfy* the AnsProlog$^{\neg,\bot}$ rule
$L_0 \leftarrow L_1, \ldots, L_m,$ **not** $L_{m+1}, \ldots,$ **not** $L_n.$ if
(i) $L_0 \neq \bot$: $\{L_1, \ldots, L_m \subseteq S\}$ and $\{L_{m+1}, \ldots, L_n\} \cap S = \emptyset$ implies that $L_0 \in S$.
(ii) $L_0 = \bot : \{L_1, \ldots, L_m \not\subseteq S\}$ or $\{L_{m+1}, \ldots, L_n\} \cap S \neq \emptyset$. A *partial Herbrand model* A of Π is a partial Herbrand interpretation S of Π such that it satisfies all rules in Π, and if S contains a pair of complementary literals, then S must be Lit. We also refer to this as A is closed under Π.

Definition 5 An answer set of an AnsProlog$^{\neg,}$**not**$^{,\bot}$ program Π is a partial Herbrand model of Π, which is minimal among the partial Herbrand models of Π. □

Alternatively, an *answer set* of an AnsProlog$^{\neg,}$**not**$^{,\bot}$ program Π can be defined as a minimal (in the sense of set-theoretic inclusion) subset S of Lit such that S is closed under Π.

It can be shown that every AnsProlog$^{\neg,}$**not** program Π has a unique answer set. (Such is not the case for AnsProlog$^{\neg,}$**not**$^{,\bot}$ programs.) We denote this answer set, if it exists, by $\mathcal{M}^{\neg,\bot}(\Pi)$. Otherwise we say $\mathcal{M}^{\neg,\bot}(\Pi)$ is undefined. We now consider several simple AnsProlog$^{\neg,}$**not**$^{,\bot}$ programs and their answer sets.

Example 20 The following table lists AnsProlog$^{\neg,}$**not** programs in its left column and their answer sets in its right column.

AnsProlog$^{\neg,}$**not** programs	Their answer sets
$\{p \leftarrow q.\ \neg p \leftarrow r.\ q \leftarrow .\}$	$\{q, p\}$
$\{p \leftarrow q.\ \neg p \leftarrow r.\ r \leftarrow .\}$	$\{r, \neg p\}$
$\{p \leftarrow q.\ \neg p \leftarrow r.\}$	$\{\}$
$\{p \leftarrow q.\ \neg p \leftarrow r.\ q \leftarrow .\ r \leftarrow .\}$	Lit

The answer sets of the first two programs are quite straight forward. Notice that the answer set of the third program has neither p, nor $\neg p$. On the other hand the answer set of the fourth program has $p, \neg p, q, \neg q, r$, and $\neg r$. This is because our definition of answer sets says that if an answer set contains a single pair of complementary literals, then it consists of all literals. □

In the following example we compare the answer sets of two AnsProlog$^{\neg,-\mathbf{not}}$ programs and demonstrate that the notion of answer set is not "contrapositive" with respect to $\leftarrow$ and $\neg$.

Example 21 Consider the following two AnsProlog$^{\neg,-\mathbf{not}}$ programs:

$$\neg p \leftarrow . \qquad p \leftarrow \neg q.$$

and

$$\neg p \leftarrow . \qquad q \leftarrow \neg p.$$

Let's call them Π_2 and Π_3, respectively. Each of the programs has a single answer set, but these sets are different. The answer set of Π_2 is $\{\neg p\}$; the answer set of Π_3 is $\{\neg p, q\}$. Thus, our semantics is not "contrapositive" with respect to $\leftarrow$ and $\neg$; it assigns different meanings to the rules $p \leftarrow \neg q.$ and $q \leftarrow \neg p.$ □

We now define answer sets of AnsProlog$^{\neg}$ programs.

Definition 6 Let Π be an AnsProlog$^{\neg,\bot}$ program without variables. For any set S of literals, let Π^S be the AnsProlog$^{\neg,-\mathbf{not},\bot}$ program obtained from Π by deleting

(i) each rule that has a formula **not** L in its body with $L \in S$, and

(ii) all formulas of the form **not** L in the bodies of the remaining rules.

□

Clearly, Π^S is an AnsProlog$^{\neg,-\mathbf{not}}$ program, and hence its answer set is already defined. If this answer set coincides with S, then we say that S is an *answer set* of Π. In other words, the answer sets of Π are characterized by the equation

$$S = \mathcal{M}^{\neg,\bot}(\Pi^S). \tag{1.3.5}$$

Example 22 Consider the AnsProlog$^{\neg}$ program Π_1 consisting of just one rule:

$$\neg q \leftarrow \mathbf{not}\ \ p.$$

Intuitively, this rule means: "q is $false$ if there is no evidence that p is $true$." The only answer set of this program is $\{\neg q\}$. □

Example 23 Consider the following ground version of the program Π from Example 5.

$fly(tweety) \leftarrow bird(tweety), \mathbf{not}\ \neg fly(tweety).$
$\neg fly(tweety) \leftarrow penguin(tweety).$
$fly(rocky) \leftarrow bird(rocky), \mathbf{not}\ \neg fly(rocky).$
$\neg fly(rocky) \leftarrow penguin(rocky).$
$bird(tweety) \leftarrow.$

$bird(rocky) \leftarrow$.
$penguin(rocky) \leftarrow$.

The answer set of the above program is:
$\{bird(tweety), bird(rocky), penguin(rocky), fly(tweety), \neg fly(rocky)\}$.

The important aspect of the program Π is that if we found out that tweety does not fly, we can directly add $\neg fly(tweety)$ to the program Π, and the resulting program will remain consistent, but will now have the answer set $\{bird(tweety), bird(rocky), penguin(rocky), \neg fly(tweety), \neg fly(rocky)\}$.
□

The following proposition states that AnsProlog$^{\neg}$ programs can not have more than one answer sets if one of them is Lit.

Proposition 5 An AnsProlog$^{\neg,\bot}$ program Π has an inconsistent answer set iff Π has the unique answer set Lit. □

The proof of the above proposition is based on the following lemma.

Lemma 1.3.2 An AnsProlog$^{\neg,\bot}$ program can not have two answer sets such that one is a proper subset of the other. □

Proof: Suppose an AnsProlog$^{\neg,\bot}$ program has two answer sets A and and A' such that $A \subseteq A'$. Then $\Pi^{A'} \subseteq \Pi^{A}$. Since $\Pi^{A'}$ and Π^{A} are AnsProlog$^{\neg,\neg\mathbf{not},\bot}$ programs (i.e., they do not have the **not** operator and hence are monotonic) $\mathcal{M}^{\neg,\bot}(\Pi^{A'}) \subseteq \mathcal{M}^{\neg,\bot}(\Pi^{A})$, which means $A' \subseteq A$. Hence, A must be equal to A'.

Example 24 Consider the following program Π:

$a \leftarrow$ **not** b.
$b \leftarrow$ **not** a.
$q \leftarrow a$.
$\neg q \leftarrow a$.

We will show that this program has the unique answer set $\{b\}$, and in particular neither $\{a, q, \neg q\}$, not Lit are its answer sets.

$\Pi^{\{b\}}$ is the program $\{b \leftarrow .\}$, whose answer set is $\{b\}$. Hence, b is an answer set of Π.

Now, $\Pi^{\{a,q,\neg q\}}$ is the program $\{a \leftarrow ., q \leftarrow a., \neg q \leftarrow a.\}$. But, based on the definition of answer sets, the answer set of $\Pi^{\{a,q,\neg q\}}$ is Lit, not $\{a, q\neg q\}$. Hence, $\{a, q, \neg q\}$ is not an answer set of Π.

Similarly, Π^{Lit} is the program $\{q \leftarrow a., \neg q \leftarrow a.\}$, whose answer set is $\emptyset$. Hence, Lit is not an answer set of Π. □

Under rather general conditions, evaluating a query for an extended program can be reduced to evaluating two queries for a program that does not contain classical negation. Let us now show that AnsProlog$^{\neg}$ programs can be reduced to AnsProlog programs. We will need the following notation:

For any predicate p occurring in Π, let p' be a new predicate of the same arity. The atom $p'(X_1, \ldots, X_n)$ will be called the *positive form* of the negative literal $\neg p(X_1, \ldots, X_n)$. Every positive literal is, by definition, its own positive form. The positive form of a literal L will be denoted

by L^+. Π^+ stands for the AnsProlog program obtained from Π by replacing each rule (1.2.2) (with $k = 0$) by

$$L_0^+ \leftarrow L_1^+, \ldots, L_m^+, \textbf{not } L_{m+1}^+, \ldots, \textbf{not } L_n^+$$

For any set $S \subseteq Lit$, S^+ stands for the set of the positive forms of the elements of S.

Proposition 6 *[GL90]* Let S be a consistent subset of Lit. S is an answer set of Π if and only if S^+ is an answer set of Π^+. □

Example 25 Consider the program Π from Example 24. The program Π^+ will then consist of the following rules:

$a \leftarrow \textbf{not } b.$
$b \leftarrow \textbf{not } a.$
$q \leftarrow a.$
$q' \leftarrow a.$

It is easy to see that Π^+ has two answer sets $S_1^+ = \{a, q, q'\}$ and $S_2^+ = \{b\}$. Following Proposition 6 $S_2 = \{b\}$ is an answer set of Π. But $S_1 = \{a, q, \neg q\}$ does not satisfy the assumption in Proposition 6, making it inapplicable. □

The above proposition relates Π and Π^+ only when Π is consistent. The following proposition relates them when Π is inconsistent.

Proposition 7 Lit is an answer set of Π if and only if $\Pi^{++} = \Pi^+ \cup \{\leftarrow p, p'. : p \in HB_\Pi\}$ has no answer sets. □

Example 26 Consider the following program Π:

$p \leftarrow q.$
$\neg p \leftarrow r.$
$q \leftarrow .$
$r \leftarrow .$

As mentioned earlier in Example 20 it has the unique answer set Lit. Now let us consider Π^+ given by the following rules:

$p \leftarrow q.$
$p' \leftarrow r.$
$q \leftarrow .$
$r \leftarrow .$

Π^+ has the answer set $\{p, p', q, r\}$. But Π^{++} consisting of the following additional constraints has no answer sets.

$\leftarrow p, p'.$
$\leftarrow q, q'.$
$\leftarrow r, r'.$ □

1.3.4 Answer sets of AnsProlog$^{or,\perp}$ and AnsProlog$^{\neg, or,\perp}$ programs

In this section, we will discuss a further extension of the language of AnsProlog$^{\neg,\perp}$ programs by the means necessary to represent disjunctive information about the world.

Our approach to expressing disjunctive information is based on the expansion of the language of AnsProlog$^{\neg}$ programs by a new connective *or* called *epistemic disjunction* [GL91]. Notice the use of the symbol *or* instead of classical $\vee$. The meaning of *or* is given by the semantics of AnsProlog$^{\neg, or,\perp}$ programs and differs from that of $\vee$. The meaning of a formula $A \vee B$ is "A is *true* or B is *true*" while a rule A *or* $B \leftarrow$ is interpreted epistemically and means "A is believed to be *true* or B is believed to be *true*." While for any atom A, $A \vee \neg A$ is always *true*, it is possible that A *or* $\neg A$ may not be *true*.

The definition of an answer set of a AnsProlog$^{\neg, or,\perp}$ program Π [Prz91, GL91] is almost identical to that of AnsProlog$^{\neg,\perp}$ programs. Let us first consider AnsProlog$^{\neg, or,\textbf{not},\perp}$ programs, which do not have the **not** operator.

A *partial Herbrand interpretation* of an AnsProlog$^{\neg, or,\perp}$ program Π is any subset $I \subseteq Lit$. A partial Herbrand interpretation S of Π is said to *satisfy* the AnsProlog$^{\neg, or,\perp}$ rule L_0 *or* $\ldots$ *or* $L_k \leftarrow L_{k+1}, \ldots, L_m,$ **not** $L_{m+1}, \ldots,$ **not** L_n. if
(i) $k = 0$ and $L_0 = \perp$: $\{L_{k+1}, \ldots, L_m\} \not\subseteq S$ or $\{L_{m+1}, \ldots, L_n\} \cap S \neq \emptyset$
(ii) otherwise: $\{L_{k+1}, \ldots, L_m\} \subseteq S$ and $\{L_{m+1}, \ldots, L_n\} \cap S = \emptyset$ implies that $\{L_0, \ldots, L_k\} \cap S \neq \emptyset$.

A *partial Herbrand model* A of Π is a partial Herbrand interpretation S of Π such that it satisfies all rules in Π, and if S contains a pair of complementary literals, then S must be Lit. We also refer to this as A is closed under Π.

Definition 7 An answer set of an AnsProlog$^{\neg, or,\textbf{not},\perp}$ program Π is a partial Herbrand model of Π, which is minimal among the partial Herbrand models of Π. □

Answer sets of AnsProlog$^{or,\textbf{not}}$ program are defined similarly, except that instead of partial Herbrand models, we consider the Herbrand models.

Alternatively, an *answer set* of an AnsProlog$^{\neg, or,\textbf{not},\perp}$ program Π can be defined as a smallest (in a sense of set-theoretic inclusion) subset S of Lit such that S is closed under Π.

Unlike AnsProlog$^{\neg,\textbf{not},\perp}$ programs, an AnsProlog$^{\neg, or,\textbf{not},\perp}$ program may have more than one answer sets. The following example illustrates such a program.

Example 27 The following AnsProlog$^{\neg, or,\textbf{not}}$ program

$p(a)$ *or* $\neg p(b) \leftarrow$.

has two answer sets $\{p(a)\}$ and $\{\neg p(b)\}$. □

Although, the above program can be replaced by the AnsProlog program $\{p(a) \leftarrow$ **not** $\neg p(b)., \neg p(b) \leftarrow$ **not** $p(a).\}$; in general the disjunction *or* can not be replaced by such transformations. The following two examples illustrate this.

Example 28 Consider the following program Π.

a *or* $b \leftarrow$.
$a \leftarrow b$.
$b \leftarrow a$.

It has only one answer set $S = \{a, b\}$. Now consider the program Π' obtained by transforming Π by the earlier mentioned transformation. This program will have the following rules:

$a \leftarrow$ **not** b.
$b \leftarrow$ **not** a.
$a \leftarrow b$.
$b \leftarrow a$.

The set S is not an answer set of the transformed program Π'. In fact Π' does not have any answer sets. □

Example 29 Consider the following program Π.

$p \ or \ p' \leftarrow$.
$q \ or \ q' \leftarrow$.
$not\_sat \leftarrow p, q$.
$not\_sat \leftarrow p', q'$.
$q \leftarrow not\_sat$.
$q' \leftarrow not\_sat$.

This program has two answer set $S_1 = \{p, q'\}$ and $S_2 = \{p', q\}$. This program has no answer sets containing p, q, as that would force that answer set to also have not_sat and q' making it a strict superset of S_1. Similarly, this program has no answer sets containing p', q', as that would force that answer set to also have not_sat and q making it a strict superset of S_2.

Now consider the program Π' obtained by adding the following rule to Π.

$not\_sat \leftarrow p', q$.

This program has two answer set $S'_1 = \{p, q'\}$ and $S'_2 = \{p', q', q, not\_sat\}$.

Consider the program Π'' obtained from Π' by replacing disjunctions through a transformation mentioned earlier. The program Π'' would then be:

$p \leftarrow$ **not** p'.
$p' \leftarrow$ **not** p.
$q \leftarrow$ **not** q'.
$q' \leftarrow$ **not** q.
$not\_sat \leftarrow p, q$.
$not\_sat \leftarrow p', q'$.
$not\_sat \leftarrow p', q$.
$q \leftarrow not\_sat$.
$q' \leftarrow not\_sat$.

While S'_1 is still an answer set of Π'', S'_2 is no longer an answer set of Π''. □

We denote the set of answer sets of an AnsProlog$^{\neg, \ or \ , \text{-}\mathbf{not}, \bot}$ program Π by $\mathcal{M}^{\neg, \ or \ , \bot}(\Pi)$. (Similarly, we denote the set of answer sets of an AnsProlog$^{\ or \ , \text{-}\mathbf{not}, \bot}$ program Π by $\mathcal{M}^{\ or \ , \bot}(\Pi)$.) We are now ready to define the answer set of an arbitrary AnsProlog$^{\neg, \ or \ , \bot}$ program.

A set S of literals is an answer set of an AnsProlog$^{\neg, \ or \ , \bot}$ program Π if $S \in \mathcal{M}^{\neg, \ or \ , \bot}(\Pi^S)$ where Π^S is as defined in Definition 6. Similarly, a set S of atoms is an answer set of an AnsProlog$^{\ or \ , \bot}$ program Π if $S \in \mathcal{M}^{\ or \ , \bot}(\Pi^S)$ where Π^S is as defined in Definition 6.

Example 30 Consider the following AnsProlog^{or} program Π.

p *or* $p' \leftarrow$.
q *or* $q' \leftarrow$.
$not\_sat \leftarrow p, q$.
$not\_sat \leftarrow p', q'$.
$q \leftarrow not\_sat$.
$q' \leftarrow not\_sat$.
$sat \leftarrow$ **not** not_sat.

This program has two answer set $S_1 = \{p, q', sat\}$ and $S_2 = \{p', q, sat\}$. The reason $S = \{p, q, q', not\_sat\}$ is not an answer set of Π, is because Π^S has an answer set $\{p, q'\}$ which is a strict subset of S. For similar reasons, the set $\{p', q', q, not\_sat\}$ is not an answer set of Π. □

Example 31 Consider the following AnsProlog$^{\perp, or}$ program:

a *or* $b \leftarrow$.
a *or* $c \leftarrow$.
$\lambda \leftarrow a,$ **not** $b,$ **not** c.
$\lambda \leftarrow$ **not** a, b, c.

It has no answer sets. The answer sets of the sub-program consisting of the first two rules are $\{a\}$ and $\{b, c\}$ and both violate the constraints represented by the third and fourth rules.

A wrong approach to analyze the above program would be to compute the models of the sub-program consisting of the first two rules (which are $\{a\}, \{a, c\}, \{a, b\}, \{b, c\}$ and $\{a, b, c\}$), eliminating the ones which violate the constraints ($\{a\}$ and $\{b, c\}$), and selecting the minimal ones from the rest. This will lead to the models $\{a, b\}$ and $\{a, c\}$. □

The following proposition gives an alternative characterization of answer sets of AnsProlog^{or} programs that is often useful.

Proposition 8 *M is an answer set of an AnsProlog^{or} program Π iff M is a model of Π and there does not exist M' such that M' is a model of Π^M and $M' \subset M$.* □

Proof:
M is an answer set of Π iff
M is an answer set of Π^M iff
M is a minimal model of Π^M iff
M is a model of Π^M and there does not exist M' such that M' is a model of Π^M and $M' \subset M$ iff
M is a model of Π and there does not exist M' such that M' is a model of Π^M and $M' \subset M$. □

1.3.5 Query entailment

So far we have discussed the AnsProlog* languages for representing knowledge and defined the answer sets of theories in these languages. Our ultimate goal is to be able to reason with and derive conclusions from a given AnsProlog* theory. In other words we need to define an entailment relation $\models$ between an AnsProlog* theory and a query. This means we also need a language to express queries.

The requirements of a query language is somewhat different from the requirement of a knowledge representation language. Normally the set of people who are expected to represent knowledge in

a knowledge base is a very small subset of the people who are expected to use (or query) the knowledge base. The former are often referred to as domain experts and the later are often referred to as users. Thus the query language should be simpler than the knowledge representation language, and should have constructs that are already familiar to an average user. With these in mind, we define queries as follows:

1. A ground atom is a query.
2. If q_1 and q_2 are queries, $\neg q_1$, $q_1 \vee q_2$, and $q_1 \wedge q_2$ are queries.
3. Nothing else is a query.

We will now define two different entailment relations between AnsProlog* theories and queries. We need two different entailment relations because AnsProlog^{or} programs (and AnsProlog programs) can not represent negative information directly, while AnsProlog$^{\neg, or}$ program can. Moreover an answer set of an AnsProlog^{or} program is a set of atoms while an answer set of an AnsProlog$^{\neg, or}$ program is a set of literals. So in the first case conclusions about negative literals have to be done indirectly, while in the second case they can be done directly. We first define when a query is true and when it is false with respect to an answer set for both cases, and then defined the two entailment relations: $\models$ and $\models^{*}$.

1. For AnsProlog$^{or,\perp}$ programs: Let S be an answer set of such a program.
   - A ground atom p is *true* with respect to S if $p \in S$.
   - A query $\neg p$ is true with respect to S if p is not true with respect to S. (Note that if p is an atom, this means $p \notin S$.)
   - A query $p \vee q$ is true with respect to S if p is true with respect to S or q is true with respect to S.
   - A query $p \wedge q$ is true with respect to S if p is true with respect to S and q is true with respect to S.
   - A query p is said to be $false$ with respect to S, if p is not *true* with respect to S.

   Given an AnsProlog^{or} program Π and a query q, we say $\Pi \models q$, if q is true in all answer sets of Π. Thus, $\Pi \models \neg q$, also means that q is false in all answer sets of Π. If $\Pi \models q$ then we say that the answer to query q is *yes*, and if $\Pi \models \neg q$ then we say that the answer to query q is *no*. If q true with respect to some answer sets of S and false with respect to the others then we say that the answer to query q is *unknown*.
2. For AnsProlog$^{\neg, or,\perp}$ programs: Let S be an answer set of such a program.
   - A ground atom p is *true* in S if p is in S; and is $false$ in S if $\neg p$ is in S.
   - A query $f \wedge g$ is *true* in S iff f is *true* in S and g is *true* in S.
   - A query $f \wedge g$ is $false$ in S iff f is $false$ in S or g is $false$ in S.
   - A query $f \vee g$ is *true* in S iff f is *true* in S or g is *true* in S.
   - A query $f \vee g$ is $false$ in S iff f is $false$ in S and g is $false$ in S.
   - A query $\neg f$ is $true(false)$ in S iff f is $false$ $(true)$ in S.

A query q is said to be *true* with respect to an AnsProlog$^{\neg, or}$ program Π and denoted by $\Pi \models^{*} q$ if it is *true* in all answer sets of Π; q is said to be $false$ with respect to Π and denoted by $\Pi \models^{*} \neg q$ if it is $false$ in all answer sets of Π. Otherwise it is said to be *unknown* with respect to Π. By $Cn(\Pi)$ we denote the set of ground literals that are entailed by Π with respect to the entailment relation $\models^{*}$.

Example 32 Consider the following program Π:

$p(X) \leftarrow q(X).$
$q(a) \leftarrow .$
$r(b) \leftarrow .$

The unique answer set of Π is $\{q(a), p(a), r(b)\}$. But since by looking at Π we can not be sure if it is an AnsProlog program or an AnsProlog$^{\neg}$ program, we can consider both entailment relations $\models$ and $\models^{*}$ with respect to Π.

We can reason $\Pi \models p(a)$ and $\Pi \models \neg p(b)$; but when we consider $\models^{*}$, we have $\Pi \models^{*} p(a)$ but we do not have $\Pi \models^{*} \neg p(b)$. □

Example 33 Consider the following AnsProlog$^{\neg}$ program Π:

$p(X) \leftarrow q(X).$
$q(a) \leftarrow .$
$\neg r(b) \leftarrow .$

The unique answer set of Π is $\{q(a), p(a), \neg r(b)\}$. Thus we can say $\Pi \models^{*} p(a)$. But we can not say $\Pi \models^{*} \neg p(b)$. □

The reader might wonder if it is worth sacrificing the expressiveness by having such a simple query language. Actually, we are not sacrificing the expressiveness. An unusual user who needs added expressiveness and who either knows or is willing to learn AnsProlog* can always represent a sophisticated query which may not be easily expressible by the above simple query language, by adding an appropriate set of AnsProlog* rules to the original program and asking a simple query to the resulting program. The following examples express this technique, which we further elaborate in Section 2.1.7.

Example 34 Suppose a user wants to find out if the knowledge base entails that at least one object in the Herbrand universe has the property p. This query can be asked by adding the following rules to the original program and asking the query q with respect to the new program, where q does not appear in the original program.

$q \leftarrow p(X).$

Suppose a user wants to find out if the knowledge base entails that all objects in the Herbrand universe have the property p. This query can be asked by adding the following rules to the original program and asking the query q with respect to the new program, where q and not_q do not appear in the original program.

$not\_q \leftarrow$ **not** $p(X).$
$q \leftarrow$ **not** $not\_q.$

Suppose a user wants to find out if the knowledge base entails that all objects in the Herbrand universe may have the property p with out resulting in contradiction. This query can be asked by

adding the following rules to the original program and asking the query q with respect to the new program, where q and not_q do not appear in the original program.

$not\_q \leftarrow \neg p(X)$.
$q \leftarrow$ **not** not_q. □

Proposition 6 suggests a simple way of evaluating queries in consistent AnsProlog$^\neg$ programs by just using an AnsProlog interpreter. To obtain an answer for query p with respect to a consistent AnsProlog$^\neg$ program Π we will need to run queries p and p' on the AnsProlog program Π^+. If Π^+'s answer to p is yes then Π's answer to p will be yes; if Π^+'s answer to p' is yes then Π's answer to p will be no; Otherwise Π's answer to p will be unknown.

1.3.6 Sound approximations : the well-founded semantics and Fitting's semantics

In Chapter 9 we will discuss several alternative semantics of programs with AnsProlog* syntax. Among those, the well-founded semantics of AnsProlog programs is often considered as an alternative to the answer set semantics that we prefer. We view the well-founded semantics to be as an approximation of the answer set semantics that has a lower time complexity. We now briefly expand on this.

In Section 1.3.2 we define the answer set of an AnsProlog program Π as the set S of atoms that satisfy the equation $S = \mathcal{M}_0(\Pi^S)$. Let us represent the function $\mathcal{M}_0(\Pi^S)$ as $\Gamma_\Pi(S)$. Now we can say that answer sets of a program Π are the fixpoint of Γ_Π.

In Section 9.6.4 we will show that the well-founded semantics of AnsProlog programs is given by $\{lfp(\Gamma^2_\Pi), gfp(\Gamma^2_\Pi)\}$; according to which, for an atom p we have $\Pi \models_{wf} p$ iff $p \in lfp(\Gamma^2_\Pi)$ and $\Pi \models_{wf} \neg p$ iff $p \notin gfp(\Gamma^2_\Pi)$. Since fixpoints of Γ_Π are also fixpoints of Γ^2_Π, we can easily show that the well-founded semantics is an approximation of the answer set semantics for AnsProlog programs. The following proposition formally states this.

Proposition 9 Let Π be an AnsProlog program and A be an atom.
(i) $A \in lfp(\Gamma^2_\Pi)$ implies $\Pi \models A$.
(ii) $A \notin gfp(\Gamma^2_\Pi)$ implies $\Pi \not\models A$. □

Example 35 Consider the following program Π.

$p \leftarrow a$.
$p \leftarrow b$.
$a \leftarrow$ **not** b.
$b \leftarrow$ **not** a.

The sets $\emptyset$, and $\{p, a, b\}$ are fixpoints of Γ^2_Π and since the Herbrand base is $\{p.a.b\}$, we have $lfp(\Gamma^2_\Pi) = \emptyset$ and $gfp(\Gamma^2_\Pi) = \{p, a, b\}$. Thus according to the well founded semantics a, b and p, are unknown with respect to this program. These conclusions are sound with respect to the answer set semantics, according to which a, and b are unknown and p is *true*. □

1.4 Database queries and AnsProlog* functions

Often we are not merely interested in the semantics of a stand alone AnsProlog* program, but more interested in the semantics of the programs when additional facts (rules with empty body)

are added to it. In this context AnsProlog* programs can also be thought of as functions. Viewing an AnsProlog* program as a specification of a theory about the world does not contradict with viewing it as a function; it is then a function with no inputs.

In this section we discuss several different formulations of viewing AnsProlog* programs as functions. The formulations vary by how the domain and co-domain of the function is defined, and whether the domain, co-domain, or both are extracted from the program, or given separately. Based on these criteria we will elaborate on the following notions:

1. Datalog and i-functions:
   In case of the 'i-function' or *inherent function* corresponding to a Datalog or AnsProlog* program, the domain and co-domain are extracted from the program. Normally, predicates in the left hand side of non-fact rules are referred to as IDB (intensional database) or output predicates, and the other predicates are referred to as the EDB (extensional database) or input predicates.

2. l-functions:
   An l-function or *literal function* is a three-tuple $\langle \Pi, \mathcal{P}, \mathcal{V} \rangle$, where Π is an AnsProlog* program, $\mathcal{P}$ and $\mathcal{V}$ are sets of literals referred to as parameters and values, and the domain is extracted from Π and $\mathcal{P}$.

3. s-functions:
   An s-function or *signature function* is a four-tuple $\langle \Pi, \sigma_i, \sigma_o, Dom \rangle$, where Π is an AnsProlog* program, σ_i is an input signature, σ_o is an output signature, and Dom is the domain. Following [GG99] we also refer s-functions as lp-functions, meaning logic programming functions.

4. an AnsProlog* program being functional:
   An AnsProlog* program Π is said to be functional from a set X to 2^Y, if for any $x \in X$, the answer sets of $\Pi \cup x$ agree on Y.

1.4.1 Queries and inherent functions

The queries in Section 1.3.5 are basically yes-no queries, where we want to find out if a query is true or false with respect to a knowledge base. Often, for example in databases, we need a more general notion of queries. In these queries we look for tuples of objects that satisfy certain properties. For example, we may need to query an employee database about the name, employee id, and salary of employees who have a Masters degree. Another example, is to query a genealogy database about listing all ancestors of John. While the first query can be expressed by generalizing the query language in Section 1.3.5 by allowing non-ground atoms and quantifiers (thus having the query as first-order theory), the second query involving transitive closure can not be expressed using first-order logic, but can be expressed by an AnsProlog* program where we have a new predicate of arity same as the arity of the tuples of objects that we are looking for, and rules about that predicate. For example, given a genealogy database, the query to list all ancestors of John can be expressed by the following AnsProlog program Π:

$anc\_of\_john(X) \leftarrow parent(john, X).$
$anc\_of\_john(X) \leftarrow anc\_of\_john(Y), parent(Y, X).$

If we just consider the above program by itself, it has $\emptyset$ as its unique answer set. That is not the right way to look at this program. The right way to look at the above program is to consider the

above two rules together with a genealogy database consisting of facts of the form $parent(a, b)$. Now all the anc_of_john atoms entailed by the resulting program will answer our query about who the ancestors of John are. The answer will vary depending on the genealogy database that is used. Thus the AnsProlog program Π (and the query it represents) can be thought of as a function that maps *parent* atoms to anc_of_john atoms. To make this view of AnsProlog* programs more precise, we first recall some standard relational database notions.

A *relation schema* R_i has a name N_i and a finite list of attributes $L_i = \langle A_1, \ldots, A_{l_i} \rangle$, where l_i is the arity of the relation schema R_i. It will sometimes be denoted as $R_i(A_1, \ldots, A_{l_i})$. A *database schema* [Ull88a] R is a finite set $\{R_1, \ldots, R_n\}$ of relation schemata. $\mathcal{U}$ is an arbitrarily large but finite set of objects that can be used in the relations and is referred to as the *domain*. Given a relation schema R_i, a *relation instance* is a set of tuples of the form $\langle a_1, \ldots, a_{l_i} \rangle$, where $\{a_1, \ldots, a_{l_i}\} \subset \mathcal{U}$. This tuple may also be denoted by the atom $R_i(a_1, \ldots, a_{l_i})$. A *database instance* W is a set of relation instances. A *query* from a database schema R (called an input database schema) to a database schema S (called the output database schema) is a partial mapping from instances of R to (incomplete) instances of S.

In the context of AnsProlog* programs, relations in the previous paragraph correspond to a predicate, and relation instances correspond to ground facts. Given an AnsProlog* program, such as Π above, it can be viewed as a query (or i-function or inherent function) with the output database schema as the set of predicates that appear in the head of the rules of the program and the input database schema as the remaining set of predicates in the program. Thus the program Π above is a query from the input database schema $\{parent\}$ to the output database schema $\{anc\_of\_john\}$.

1.4.2 Parameters, Values and literal functions

Traditionally, the intuitive meaning of a database instance W is based on the closed world assumption (CWA). I.e. if $R_i(a_1, \ldots, a_{l_i}) \in W$ then we say that $R_i(a_1, \ldots, a_{l_i})$ is $true$ w.r.t. W, otherwise we say that $R_i(a_1, \ldots, a_{l_i})$ is $false$ w.r.t. W. In presence of incompleteness the notion of relation instance can be extended to *incomplete relation instances* which consist of positive literals (or atoms) of the form $R_i(a_1, \ldots, a_{l_i})$ and negative literals of the form $\neg R_j(a_1, \ldots, a_{l_j})$. An *incomplete database instance* is a set of incomplete relation instances. When dealing with incomplete database instance CWA is no longer assumed. Given a relation schema R_k and an incomplete database instance W, we say $R_k(a_1, \ldots, a_{l_k})$ is $true$ w.r.t. W if $R_k(a_1, \ldots, a_{l_k}) \in W$, we say $R_k(a_1, \ldots, a_{l_k})$ is $false$ w.r.t. W if $\neg R_k(a_1, \ldots, a_{l_k}) \in W$; otherwise we say $R_k(a_1, \ldots, a_{l_k})$ is *unknown* w.r.t. W. We can now define the notion of an *extended query* from a database schema R to a database schema S as a partial mapping from incomplete instances of R to incomplete instances of S.

An AnsProlog$^\neg$ program Π can be viewed as an extended query (or i-function or inherent function) with the output database schema as the set of predicates that appear in the head of the rules in Π and the input database schema as the remaining set of predicates from Π.

Example 36 Consider an instance of a genealogy database of dinosaurs obtained from a particular archaeological site. Its very likely that we will have an incomplete instance where we have facts such as $parent(a, b)$ and also facts such as $\neg parent(c, d)$. In that case the following AnsProlog$^\neg$ program expresses the extended query from the input database schema $\{parent\}$ to the output database schema $\{anc\_of\_john\}$.

$maybe\_parent(Y, X) \leftarrow$ **not** $\neg parent(Y, X)$.
$maybe\_anc\_of\_john(X) \leftarrow maybe\_parent(john, X)$.

$maybe\_anc\_of\_john(X) \leftarrow maybe\_anc\_of\_john(Y), maybe\_parent(Y, X).$
$anc\_of\_john(X) \leftarrow parent(john, X).$
$anc\_of\_john(X) \leftarrow anc\_of\_john(Y), parent(Y, X).$
$\neg anc\_of\_john(X) \leftarrow$ **not** $maybe\_anc\_of\_john(X).$ □

Often, instead of viewing an AnsProlog* program as an i-function where the input and output is derived from the program, we may want to explicitly specify them. In addition we may want to relax the criteria that the input and output be disjoint and that input and output be defined in terms of predicates.

Thus an AnsProlog* program Π and two sets of literals $\mathcal{P}_\Pi$ and $\mathcal{V}_\Pi$, define an *l-function (literal function)* whose domain consists of subsets of $\mathcal{P}_\Pi$ with certain restrictions that depend on the particular subclass of AnsProlog* program we are interested in, and co-domain consists of subsets of $\mathcal{V}_\Pi$. Given a valid input X, $\Pi(X)$ is defined as the set $\{l : l \in \mathcal{V}_\Pi \text{ and } \Pi \cup X \models l\}$. Since it is not required that $\mathcal{P}_\Pi$ and $\mathcal{V}_\Pi$ be disjoint they are referred to as *parameter* and *values* instead of input and output. In Section 3.7 we use l-functions to represent queries and extended queries, and define the notion of expansion of queries and the corresponding interpolation of l-functions.

1.4.3 The signature functions

Recall that the language associated with an AnsProlog* program is determined by its signature consisting of the constants, function symbols and predicate symbols. In signature functions (or s-functions) the input and output is specified through signatures. Thus an s-function has four parts: an AnsProlog* program, an input signature, an output signature and a domain. An s-function differs from an l-function in two main aspects:

- The domain in s-functions is specified, instead of being derived as in l-functions.
- Unlike the parameters and values of l-functions which are sets of literals, input and output in s-functions are specified through input and output signatures. In both cases, they are directly specified, and are not derived from the AnsProlog* program.

The s-functions are used in formulating building block results whereby we can formally discuss composition and other operations on AnsProlog* programs viewed as s-functions.

1.4.4 An AnsProlog* program being functional

In i-functions, l-functions and s-functions we consider the entailment relation of AnsProlog* programs and associate functions with them. A different perspective is to consider the various answer sets and analyze the mapping between input (literals or predicates) and the output (literals or predicates) in each of the answer sets. This leads to the following definition of when an AnsProlog* program encodes a function.

Definition 8 An AnsProlog* program T is said to encode a function (or is functional) from a set of literals, called *input*, to a set of literals called *output* if for any complete subset E of input such that $T \cup E$ is consistent, all answer sets of $T \cup E$ agree on the literals from output. □

The above definition of an AnsProlog* program being functional is used in Section 3.9 as one of the sufficiency conditions for determining when an observation can be directly added to a program, and when it needs to be assimilated through abduction or conditioning.

1.5 Notes and references

Logic programming and the programming language PROLOG started off as programming with Horn clauses, a subset of first order logic formulas. The first book on logic programming and PROLOG was by Kowalski [Kow79]. Lloyd's books [Llo84, Llo87] have various formal characterizations of AnsProlog$^{-\mathbf{not}}$ programs (called definite programs) and their equivalence results. A similar book on AnsProlog$^{-\mathbf{not},\,or}$ (called disjunctive logic programs) is [LMR92] which is partly based on Rajasekar's thesis. Minker's workshop and his edited book [Min88a] presents several papers on characterizing various subclasses – such as stratification and local stratification – of AnsProlog programs. The stable model semantics of AnsProlog programs was first proposed in [GL88] and then extended to AnsProlog$^{\neg}$ programs in [GL90, GL91]. (The notion of answer sets was first introduced in [GL90].) They were further extended to AnsProlog$^{\neg,\,or}$ programs and programs with epistemic operators in [Gel91b, Gel91a, Gel94]. The survey paper [BG94] discussed and cataloged various knowledge representation possibilities using the answer set semantics of AnsProlog* programs and its extensions. The survey paper [AB94] focussed on semantics of negation in logic programs. Two other more recent surveys on logic programming that have significant material on the answer set semantics are [Lif96] and [DEGV97]. The former has a good discussion on query answering methods for AnsProlog programs, while the later has a good summary on the complexity and expressibility of AnsProlog* languages.

Chapter 2

Simple modules for declarative programming with answer sets

In this chapter we present several small AnsProlog* programs corresponding to several declarative problem solving modules or knowledge representation and reasoning aspects. Although in general we may have intermingling of the declarative problem solving, and the knowledge representation and reasoning aspects, they can be differentiated as follows.

Normally a problem solving task is to find solutions of a problem. A declarative way to do that is to declaratively enumerate the possible solutions, and the tests such that the answer sets of the resulting program correspond to the solutions of the problem. The declarative problem solving modules that we consider in this chapter include modules that enforce simple constraints, modules that enumerate interpretations with respect to a set of atoms, modules that uniquely choose from a set of possibilities, modules that encode propositional satisfiability, modules that represent closed first-order queries, modules that check satisfiability of quantified boolean formulas with up to two quantifiers, modules that assign a linear ordering between a set of objects, modules that can represent various aggregation of facts, such as minimization, maximization, count and average, module that encode classical disjunction conclusions, modules that encode exclusive-or conclusions, and modules that encode cardinality and weight constraints.

By knowledge representation and reasoning aspects we mean representing particular bench mark aspects of non-monotonic and common-sense reasoning that we want to be encoded by AnsProlog* programs. The knowledge representation and reasoning modules that we consider in this chapter include, modules for representing normative statements, exceptions, weak exceptions, and direct contradictions, modules for representing the frame problem, and reasoning with incomplete information, transformations necessary for removing closed world assumption, modules for representing null values, and modules for reasoning about what is known to an agent and what is not known, as opposed to what is true or false in the world.

Although we individually examine and analyze each of these modules in this Chapter, we do not discuss *general properties and principles* of these modules. We explore the later in Chapter 3 and discuss how modules can be systematically analyzed to identify their properties, how modules can be combined or put on top of others to develop larger and more involved programs, and how answer sets of a program can be constructed by decomposing the program into smaller programs, and composing the answer sets of the smaller programs.

2.1 Declarative problem solving modules

2.1.1 Integrity Constraints

Integrity constraints are written as rules with an empty head. Intuitively, an integrity constraint r written as $\leftarrow l_1, \ldots, l_m, \textbf{not } l_{m+1}, \ldots, \textbf{not } l_n$, where l_is are literals, *forbids* answer sets which contain the literals $l_1, \ldots, l_m$ and do not contain the literals $l_{m+1}, \ldots, l_n$. Sets of literals that are forbidden by r are said to violate r.

Since many of the results through out the book are about AnsProlog* programs that do not allow constraints, we show here how constraints can be alternatively represented using rules with non-empty heads by introducing a new atom, which we will refer to as *inconsistent* and adding the following rule $c_1(r)$ to the program.

$inconsistent \leftarrow \textbf{not } inconsistent, l_1, \ldots, l_m, \textbf{not } l_{m+1}, \ldots, \textbf{not } l_n.$

Proposition 10 Let Π be an AnsProlog* program that does not contain the atom *inconsistent*. Let $r_1, \ldots, r_n$ be a set of integrity constraints that do not contain the atom *inconsistent*. A is an answer set of Π that does not violate the constraints $r_1, \ldots, r_n$ iff A is an answer set of $\Pi \cup \{c_1(r_1), \ldots, c_1(r_n)\}$. □

Proof: Exercise.

Example 37 Let Π be an AnsProlog program consisting of the rules:

$a \leftarrow \textbf{not } b.$
$b \leftarrow \textbf{not } a.$

It is easy to see that Π has two answer sets $\{a\}$ and $\{b\}$. Now suppose we would like to incorporate the constraint r :

$\leftarrow a.$

to prune any answer set where a is true. To achieve this we will add the following rule $c_1(r)$.

$p \leftarrow \textbf{not } p, a.$

to Π.

It is easy to see that $\Pi \cup \{c_1(r)\}$ has only one answer set, $\{b\}$ and the set $\{a\}$ which was an answer set of Π is no longer an answer set of $\Pi \cup \{c_1(r)\}$. □

An alternative way to encode the integrity constraint r is by adding the following – which we will refer to as $c_2(r)$ – to a program Π with the stipulation that p and q are not in the language of Π.

$p \leftarrow l_1, \ldots, l_m, \textbf{not } l_{m+1}, \ldots, \textbf{not } l_n.$

$q \leftarrow \textbf{not } p.$
$q \leftarrow \textbf{not } q.$

Proposition 11 Let Π be an AnsProlog* program that does not contain the atoms p, and q. Let $r_1, \ldots, r_n$ be a set of integrity constraints that do not contain the atoms p and q. $A \setminus \{q\}$ is an answer set of Π that does not violate $r_1, \ldots, r_n$ iff A is an answer set of $\Pi \cup c_2(r_1) \cup \ldots \cup c_2(r_n)$. □

Proof: Exercise.

2.1.2 Finite enumeration

Suppose we have propositions $p_1, \ldots p_n$ and we would like to construct a program where for each interpretation of the propositions we have an answer set, and each answer set encodes a particular interpretation. An AnsProlog program that achieves this is as follows:

$p_1 \leftarrow$ **not** n_p_1.
$n\_p_1 \leftarrow$ **not** p_1.
$\vdots$

$p_n \leftarrow$ **not** n_p_n.
$n\_p_n \leftarrow$ **not** p_n.

An AnsProlog$^{\neg, or}$ program that achieves this is as follows:

p_1 *or* $\neg p_1 \leftarrow$.
$\vdots$

p_n *or* $\neg p_n \leftarrow$.

It should be noted that if we add the above programs to another program, then the resulting program will not necessarily preserve the property that we started with. Nevertheless, the encodings above are often useful in enumerating the interpretations and allowing other part of the resulting program to prune out interpretations that do not satisfy certain properties.

For example, the p_is may denote action occurrences and we may want to enforce that at each time point at least one action occur. This can be achieved by the following encoding.

p_1 *or* $\neg p_1 \leftarrow$.
$\vdots$

p_n *or* $\neg p_n \leftarrow$.
$none \leftarrow \neg p_1, \neg p_2, \ldots, \neg p_n$.
$inconsistent \leftarrow$ **not** $inconsistent, none$.

In section 2.1.6 we show how enumeration is used in encoding propositional satisfiability in AnsProlog.

2.1.3 General enumeration but at least one

In the previous section we enumerated among a set of propositions. In presence of variables, and predicate symbols, we may like to enumerate a set of terms that satisfy some particular criteria. Let us assume that rules with $possible(X)$ encode when X is possible, and our goal is to enumerate the various terms that are possible, with the added stipulation that in each answer set at least one term should be chosen. The the following AnsProlog program achieves our purpose.

$chosen(X) \leftarrow possible(X),$ **not** $not\_chosen(X)$.
$not\_chosen(X) \leftarrow possible(X),$ **not** $chosen(X)$.
$some \leftarrow chosen(X)$.
$inconsistent \leftarrow$ **not** $inconsistent,$ **not** $some$.

Example 38 The following AnsProlog$^{\neg, or}$ program also achieves the purpose of general enumeration but at least one.

$chosen(X)$ *or* $\neg chosen(X) \leftarrow possible(X)$.
$some \leftarrow chosen(X)$.
$inconsistent \leftarrow$ **not** $inconsistent$, **not** $some$.

Let us consider the program obtained by adding the following set of facts to the above AnsProlog$^{\neg, or}$ program.

$possible(a) \leftarrow$.
$possible(b) \leftarrow$.
$possible(c) \leftarrow$.

The resulting program has the following answer sets:

$\{possible(a), possible(b), possible(c), chosen(a), \neg chosen(b), \neg chosen(c), some\}$
$\{possible(a), possible(b), possible(c), chosen(b), \neg chosen(c), \neg chosen(a), some\}$
$\{possible(a), possible(b), possible(c), chosen(c), \neg chosen(a), \neg chosen(b), some\}$
$\{possible(a), possible(b), possible(c), chosen(a), chosen(b), \neg chosen(c), some\}$
$\{possible(a), possible(b), possible(c), chosen(a), chosen(c), \neg chosen(b), some\}$
$\{possible(a), possible(b), possible(c), chosen(b), chosen(c), \neg chosen(a), some\}$
$\{possible(a), possible(b), possible(c), chosen(a), chosen(b), chosen(c), some\}$

Note that neither $\{possible(a), possible(b), possible(c), \neg chosen(a), \neg chosen(b), \neg chosen(c)\}$ nor $\{possible(a), possible(b), possible(c), \neg chosen(a), \neg chosen(b), \neg chosen(c), inconsistent\}$ are answer sets of the above program. □

2.1.4 Choice: general enumeration with exactly one

Let us continue with the assumption in the previous subsection that we have rules with $possible(X)$ in their head that encode when X is possible. Now we would like to write a program whose answer sets are such that in each answer set only one of the possible X is chosen, and there is at least one answer set for each X that is possible. As an example, consider that we have $P = \{possible(a) \leftarrow ., possible(b) \leftarrow ., possible(c) \leftarrow .\}$. Then our goal is to have a program which has answer sets with the following as subsets.

$S_1 = \{chosen(a), \neg chosen(b), \neg chosen(c)\}$

$S_2 = \{\neg chosen(a), chosen(b), \neg chosen(c)\}$

$S_3 = \{\neg chosen(a), \neg chosen(b), chosen(c)\}$

An AnsProlog$^{\neg}$ program Π with such answer sets can be written as follows:

$\neg chosen(X) \leftarrow chosen(Y), X \neq Y$.
$chosen(X) \leftarrow possible(X),$ **not** $\neg chosen(X)$.

It is easy to see that $\Pi \cup P$ has three answer sets, corresponding to S_1, S_2 and S_3, respectively.

Recall that by using the Proposition 6 we can replace the above AnsProlog$^{\neg}$ program by an AnsProlog program. Following is such an AnsProlog program

$diff\_chosen\_than(X) \leftarrow chosen(Y), X \neq Y$.
$chosen(X) \leftarrow possible(X),$ **not** $diff\_chosen\_than(X)$.

The above constructs are widely used in applications such as: encoding the linear planning condition that only one action occurs at each time point, and labeling each tuple of database with a unique

number during aggregate computation. In the Smodels logic programming system the above can be achieved by the following rules that uses cardinality constraints with variables.

$1\{chosen(X) : possible(X)\}1.$

2.1.5 Constrained enumeration

The module in the previous section can be thought of as a specific case of the more general notion of constrained enumeration where while enumerating we need to obey certain constraints. For example let us consider placing objects in a rectangular board so that no two objects are in the same row or in the same column, and at least one object is in each row and each column. Although, the problem solving task may involve additional constraints, we can encode the above enumeration by the following AnsProlog program:

$not\_chosen(X, Y) \leftarrow chosen(X', Y), X' \neq X.$
$not\_chosen(X, Y) \leftarrow chosen(X, Y'), Y' \neq Y.$
$chosen(X, Y) \leftarrow row(X), column(Y), \textbf{not}\ not\_chosen(X, Y).$

We can also achieve the above enumeration using the general enumeration of Section 2.1.3 together with additional constraints. For example, the following will achieve our goal.

$chosen(X, Y) \leftarrow row(X), column(Y), \textbf{not}\ not\_chosen(X, Y).$
$not\_chosen(X, Y) \leftarrow row(X), column(Y), \textbf{not}\ chosen(X, Y).$
$filled\_row(X) \leftarrow chosen(X, Y).$
$filled\_column(Y) \leftarrow chosen(X, Y).$
$missing\_row \leftarrow row(X), \textbf{not}\ filled\_row(X).$
$missing\_column \leftarrow column(X), \textbf{not}\ filled\_column(X).$
$inconsistent \leftarrow \textbf{not}\ inconsistent, missing\_row.$
$inconsistent \leftarrow \textbf{not}\ inconsistent, missing\_column.$
$inconsistent \leftarrow \textbf{not}\ inconsistent, chosen(X, Y), chosen(X, Z), Y \neq Z.$
$inconsistent \leftarrow \textbf{not}\ inconsistent, chosen(X, Y), chosen(Z, Y), X \neq Z.$

In the above program the first two rules enumerate that for any pair of row X, and column Y either it is chosen or it is not; the next six rules enforce that at least one object is in each row and each column; and the last two rules enforce that no more than one object is in each row and no more than one object is in each column.

As evident from the above two programs the constrained enumeration approach results in a smaller program than using general enumeration together with constraints. Constrained enumeration is a key component of declarative problem solving tasks. The particular constrained enumeration that we discussed in this section is useful in tasks such as the Nqueens problem; tournament scheduling problem, where rows could correspond to teams, and columns to dates; and seat assignment problems.

2.1.6 Propositional satisfiability

Propositional logic was one of the first languages used for declarative problem solving, and [KS92] reported one of the early successes of doing the problem solving task of planning by mapping a planning problem to a propositional theory, and extracting plans from the models of the propositional theory. In this subsection we show how we can map a propositional theory to an AnsProlog program so that there is a one-to-one correspondence between the models of the propositional theory and the answer sets of the AnsProlog program.

Given a set S of propositional clauses (where each clause is a disjunction of literals), we construct the AnsProlog program $\Pi(S)$, such that there is a one-to-one correspondence between models of S and answer sets of $\Pi(S)$, as follows:

- For each proposition p in S we introduce a new atom n_p and have the following two rules in $\Pi(S)$.

  $p \leftarrow$ **not** n_p.
  $n\_p \leftarrow$ **not** p.

- For each clause in S, we introduce a new atom c and include one rule for each literal l in S in the following way:

  - If l is a positive atom then we have the rule $c \leftarrow l$.
  - If l is a negation of an atom a then we have the rule $c \leftarrow n\_a$.

  Then we include the constraint $\leftarrow$ **not** c

Example 39 Let $S = \{p_1 \vee p_2 \vee p_3, p_1 \vee \neg p_3, \neg p_2 \vee \neg p_4\}$. The AnsProlog program $\Pi(S)$ consists of the following:

$p_1 \leftarrow$ **not** n_p_1.
$n\_p_1 \leftarrow$ **not** p_1.
$p_2 \leftarrow$ **not** n_p_2.
$n\_p_2 \leftarrow$ **not** p_2.
$p_3 \leftarrow$ **not** n_p_3.
$n\_p_3 \leftarrow$ **not** p_3.
$p_4 \leftarrow$ **not** n_p_4.
$n\_p_4 \leftarrow$ **not** p_4

$c_1 \leftarrow p_1$.
$c_1 \leftarrow p_2$.
$c_1 \leftarrow p_3$.
$\leftarrow$ **not** c_1

$c_2 \leftarrow p_1$.
$c_2 \leftarrow n\_p_3$.
$\leftarrow$ **not** c_2

$c_3 \leftarrow n\_p_2$.
$c_3 \leftarrow n\_p_4$.
$\leftarrow$ **not** c_3

The models of S are $\{\{p_1, p_2, p_3\}, \{p_1, p_2\}, \{p_1, p_3, p_4\}, \{p_1, p_3\}, \{p_1, p_4\}, \{p_1\}, \{p_2\}\}$ and the answer sets of $\Pi(S)$ are $\{\{p_1, p_2, p_3, n\_p_4, c_1, c_2, c_3\}, \{p_1, p_2, n\_p_3, n\_p_4, c_1, c_2, c_3\}, \{p_1, n\_p_2, p_3, p_4, c_1, c_2, c_3\}, \{p_1, n\_p_2, p_3, n\_p_4, c_1, c_2, c_3\}, \{p_1, n\_p_2, n\_p_3, p_4, c_1, c_2, c_3\}, \{p_1, n\_p_2, n\_p_3, n\_p_4, c_1, c_2, c_3\}, \{n\_p_1, p_2, n\_p_3, n\_p_4, c_1, c_2, c_3\}\}$. □

Proposition 12 A set of propositional clauses S is satisfiable iff $\Pi(S)$ has an answer set. □

Exercise 2 Formulate and prove the one-to-one correspondence between models of S and answer sets of $\Pi(S)$. □

2.1.7 Closed first-order queries in AnsProlog and AnsProlog$^\neg$

In Section 1.3.5 we discussed a simple query language for querying AnsProlog* programs and hinted in Example 34 how some more complex queries can be expressed by AnsProlog* programs. In this subsection we generalize Example 34 and give a methodology to express queries that can be otherwise expressed as a closed first-order theory.

Let Π be an AnsProlog program and F be a closed first-order query, to compute if $\Pi \models F$, we can systematically break down F to a set of AnsProlog rules $c(F)$ and ask if $\Pi \cup c(F) \models p\_F$, where p_F is the atom corresponding to the whole formula F. In the following we show how $c(F)$ is constructed bottom-up.

Let F_1 and F_2 be closed formulas and p_F_1 and p_F_2 be the atoms corresponding to F_1 and F_2 respectively. The following describes the rules we need to add when F is $F_1 \wedge F_2$, $F_1 \vee F_2$, $\neg F_1$, $\forall X.F_1(X)$, $\forall X.[in\_class(X) \rightarrow F_1(X)]$, $\exists X.F_1(X)$, and $\exists X.[in\_class(X) \rightarrow F_1(X)]$ respectively.

1. **And**: The formula $F_1 \wedge F_2$ is translated to the following rule:

   $p\_F \leftarrow p\_F_1, p\_F_2$.

2. **Or**: The formula $F_1 \vee F_2$ is translated to the following rules:

   $p\_F \leftarrow p\_F_1$.

   $p\_F \leftarrow p\_F_2$.

3. **Not**: The formula $\neg F_1$ is translated to the following rule:

   $p\_F \leftarrow$ **not** p_F_1.

4. **Existential quantifier**: The formula $\exists X.F_1(X)$ is translated to the following rule:

   $p\_F \leftarrow p\_F_1(X)$.

5. **Bounded existential quantifier**: The formula $\exists X.(in\_class(X) \rightarrow F_1(X))$ is translated to the following rule:

   $p\_F \leftarrow in\_class(X), p\_F_1(X)$.

6. **Universal quantifier**: The formula $\forall X.F_1(X)$ is translated to the following rules:

   $n\_p\_F \leftarrow$ **not** $p\_F_1(X)$.

   $p\_F \leftarrow$ **not** n_p_F.

7. **Bounded universal quantifier**: The formula $\forall X.(in\_class(X) \rightarrow F_1(X))$ is translated to the following rules:

   $n\_p\_F \leftarrow in\_class(X),$ **not** $p\_F_1(X)$.

   $p\_F \leftarrow$ **not** n_p_F.

If instead of AnsProlog we use AnsProlog$^\neg$ then the encodings of $F_1 \wedge F_2$, $F_1 \vee F_2$, $\exists X.F_1(X)$, and $\exists X.[in\_class(X) \rightarrow F_1(X)]$ remain unchanged while the other formulas are encoded as below:

- **Not**: The formula $\neg F_1$ is translated to the following rule:

  $p\_F \leftarrow \neg p\_F_1$.

- **Universal quantifier**: The formula $\forall X.F_1(X)$ is translated to the following rules:

  $\neg p\_F \leftarrow$ **not** $p\_F_1(X)$.

  $p\_F \leftarrow$ **not** $\neg p\_F$.

- **Bounded universal quantifier**: The formula $\forall X.(in\_class(X) \rightarrow F_1(X))$ is translated to the following rules:

  $\neg p\_F \leftarrow in\_class(X),$ **not** $p\_F_1(X)$.

  $p\_F \leftarrow$ **not** $\neg p\_F$.

Example 40 Consider an admission process in a college where admission is refused to an applicant if he has not taken any honors classes. The following encoding of this information is erroneous.

$refuse\_admission(X) \leftarrow applicant(X),$ **not** $taken\_honors\_class(X, Y)$.

Suppose we have an applicant John who has taken the honors class in Physics and has not taken the honors class in Chemistry. Since he has taken some honors classes he should not be refused admission. But the above rule will have one instantiation where Y will be instantiated to 'chemistry' and X to 'john', and due to that instantiation, it will entail $refuse\_admission(john)$. An alternative explanation of the inappropriateness of the above rule is that it encodes the classical formula:

$refuse\_admission(X) \subset \exists X \exists Y.[applicant(X) \wedge \neg taken\_honors\_class(X, Y)]$

which is equivalent to

$refuse\_admission(X) \subset \exists X.[applicant(X) \wedge \exists Y.[\neg taken\_honors\_class(X, Y)]]$

which is equivalent to

$refuse\_admission(X) \subset \exists X.[applicant(X) \wedge \neg \forall Y.[taken\_honors\_class(X, Y)]]$.

This is different from the information that was supposed to be encoded, which expressed in classical logic is:

$refuse\_admission(X) \subset \exists X.[applicant(X) \wedge \neg \exists Y.[taken\_honors\_class(X, Y)]]$.

A correct encoding of the above information in AnsProlog is as follows:

$has\_taken\_a\_honors\_class(X) \leftarrow taken\_honors\_class(X, Y)$.
$refuse\_admission(X) \leftarrow applicant(X),$ **not** $has\_taken\_a\_honors\_class(X)$. □

2.1.8 Checking satisfiability of universal quantified boolean formulas (QBFs)

Quantified boolean formulas (QBFs) are propositional formulas with quantifiers that range over the propositions. For example, if $F(p_1, p_2)$ is a positional formula then the satisfiability of the QBF $\exists p_1 \forall p_2 F(p_1, p_2)$ means that there exists a truth value of the proposition p_1, such that for all truth values of p_2, the formula $F(p_1, p_2)$ evaluates to true. The importance of QBFs come from the fact that they are often used as canonical examples of various complexity classes in

the polynomial hierarchy. We use them in Chapter 6 when showing the complexity of various AnsProlog* classes. In this and the subsequent several subsections we give examples of how QBF formulas are encoded in AnsProlog*. We start with encoding universal QBFs, which are of the form $\forall q_1, \ldots, q_l.\ F(q_1, \ldots, q_l)$, where $\{q_1, \ldots, q_l\}$ is a set of propositions, and $F(q_1, \ldots, q_l)$ is a propositional formula of the form $\theta_1 \vee \ldots \vee \theta_n$ with each θ_i being a conjunction of propositional literals (a proposition, or a proposition preceded by $\neg$). We first construct an AnsProlog$^{\neg, or}$ program which allows us to verify the satisfiability of the universal QBF $\forall q_1, \ldots, q_l.\ F(q_1, \ldots, q_l)$.

Proposition 13 The QBF $\forall q_1, \ldots, q_l.\ F(q_1, \ldots, q_l)$ is satisfiable iff $exist\_satisfied$ is true in all answer sets of the following program.

$q_1\ or\ \neg q_1 \leftarrow .$
$\vdots$

$q_l\ or\ \neg q_l \leftarrow .$

$exist\_satisfied \leftarrow \theta_1.$
$\vdots$

$exist\_satisfied \leftarrow \theta_n.$ □

Proof: Exercise.

The use of or in the above program is not essential. The following proposition gives us another alternative.

Proposition 14 The QBF $\forall q_1, \ldots, q_l.\ F(q_1, \ldots, q_l)$ is satisfiable iff $exist\_satisfied$ is true in all answer sets of the following program.

$q_1 \leftarrow$ **not** $\neg q_1.$
$\neg q_1 \leftarrow$ **not** $q_1.$
$\vdots$

$q_l \leftarrow$ **not** $\neg q_l.$
$\neg q_l \leftarrow$ **not** $q_l.$

$exist\_satisfied \leftarrow \theta_1.$
$\vdots$

$exist\_satisfied \leftarrow \theta_n.$ □

Proof: Exercise.

The use of $\neg$ in the above two programs is also not essential. They can be replaced by replacing each $\neg p$ by p' in all of the rules. Let θ_i' denote the transformation of θ_i, where all negative propositional literals of the form $\neg p$ are replaced by a proposition of the form p'. For example, if $\theta_i = p_1 \wedge \neg p_2 \wedge \neg p_3 \wedge p_4$ then $\theta_i' = p_1 \wedge p_2' \wedge p_3' \wedge p_4$. Besides replacing the $\neg p$ by p' in the enumeration rules, we will need to replace θ_i by θ_i' in the other rules. We use the above methodology in the following alternative encoding of a universal QBF in AnsProlog^{or} .

Proposition 15 The QBF $\forall q_1, \ldots, q_l.\ F(q_1, \ldots, q_l)$ is satisfiable iff $exist\_satisfied$ is true in the unique answer set of the following program:

$q_1\ or\ q_1' \leftarrow .$

$\vdots$

$q_l\ or\ q_l' \leftarrow .$

$exist\_satisfied \leftarrow \theta_1'.$

$\vdots$

$exist\_satisfied \leftarrow \theta_n'.$

$q_1 \leftarrow exist\_satisfied.$

$q_1' \leftarrow exist\_satisfied.$

$\vdots$

$q_l \leftarrow exist\_satisfied.$

$q_l' \leftarrow exist\_satisfied.$ □

Proof: Exercise.

Exercise 3 Explain why if we replace the rules of the form

$q_i\ or\ q_i' \leftarrow .$

by the rules

$q_i \leftarrow$ **not** $q_i'.$
$q_i' \leftarrow$ **not** $q_i.$

in the above program the answer sets will not be the same. (Hint: Consider the QBF $\forall p, q.(p \vee \neg p)$. The program with *or* will have an answer set, but the other program will have no answer sets.) □

2.1.9 Checking satisfiability of existential QBFs

In this subsection we construct AnsProlog^{or} programs which allows us to verify the satisfiability of the existential QBFs of the form $\exists q_1, \ldots, q_l.\ F(q_1, \ldots, q_l)$.

Proposition 16 The QBF $\exists q_1, \ldots, q_l.\ (\theta_1(q_1, \ldots, q_l) \vee \ldots \vee \theta_n(q_1, \ldots, q_l))$ is satisfiable iff $exist\_satisfied$ is true in at least one answer set of the following program.

$q_1\ or\ q_1' \leftarrow.$

$\vdots$

$q_l\ or\ q_l' \leftarrow.$

$exist\_satisfied \leftarrow \theta_1'.$

$\vdots$

$exist\_satisfied \leftarrow \theta_n'.$ □

Proof: Exercise.

In the above encoding, we can replace each of the rules $q_i \; or \; q_i'$ by the two AnsProlog rules, $q_i \leftarrow$ **not** q_1'. and $q_i' \leftarrow$ **not** q_1. This is not the case in the following encoding. In this alternative encoding, let ϕ_i denote a disjunction of propositional literals and let $\hat{\phi}_i$ denote the negation of ϕ_i with literals of the form $\neg p$ replaced by propositions of the form p'. For example, if ϕ is the disjunction $p \vee q \vee \neg r$, then $\hat{\phi}$ is the conjunction $p' \wedge q' \wedge r$.

Proposition 17 The QBF $\exists q_1, \ldots, q_l. \; (\phi_1(q_1, \ldots, q_l) \wedge \ldots \wedge \phi_n(q_1, \ldots, q_l))$, where ϕ_i's are disjunction of propositional literals, is satisfiable iff $not\_satisfied$ is false in all answer sets of the following program:

$q_1 \; or \; q_1' \leftarrow.$

$\vdots$

$q_l \; or \; q_l' \leftarrow.$

$not\_satisfied \leftarrow \hat{\phi_1}.$

$\vdots$

$not\_satisfied \leftarrow \hat{\phi_n}.$

$q_1 \leftarrow not\_satisfied.$

$q_1' \leftarrow not\_satisfied.$

$\vdots$

$q_l \leftarrow not\_satisfied.$

$q_l' \leftarrow not\_satisfied.$ □

Proof: Exercise.

2.1.10 Checking satisfiability of Universal-existential QBFs

We now consider encoding the satisfiability of Universal-existential QBFs using AnsProlog^{or} programs. The use of or in these encodings are essential and can not be eliminated away, and this explains the added expressibility of AnsProlog^{or} programs over AnsProlog programs.

Proposition 18 The QBF $\forall p_1, \ldots p_k. \; \exists q_1, \ldots, q_l. \; (\phi_1(p_1, \ldots, p_k, q_1, \ldots, q_l) \wedge \ldots \wedge \phi_n(p_1, \ldots, p_k, q_1, \ldots, q_l))$, where ϕ_i's are disjunction of propositional literals, is satisfiable iff $not\_satisfied$ is false in all answer sets of the following program:

$p_1 \; or \; p_1' \leftarrow.$

$\vdots$

$p_k \; or \; p_k' \leftarrow.$

$q_1 \; or \; q_1' \leftarrow.$

$\vdots$

$q_l \; or \; q_l' \leftarrow.$

$not\_satisfied \leftarrow \hat{\phi_1}.$

$\vdots$

$not\_satisfied \leftarrow \hat{\phi_n}.$

$q_1 \leftarrow not\_satisfied.$

$q_1' \leftarrow not\_satisfied.$

$\vdots$

$q_l \leftarrow not\_satisfied.$

$q_l' \leftarrow not\_satisfied.$ □

Proof: Exercise.

Exercise 4 Prove that the QBF $\forall p_1, \ldots p_k.\ \exists q_1, \ldots, q_l.\ (\phi_1(p_1, \ldots, p_k, q_1, \ldots, q_l) \wedge \ldots \wedge \phi_n(p_1, \ldots, p_k, q_1, \ldots, q_l))$, where ϕ_i's are disjunction of propositional literals, is satisfiable iff sat is true in all answer sets of the following program:

$p_1\ or\ p_1' \leftarrow.$

$\vdots$

$p_k\ or\ p_k' \leftarrow.$

$q_1\ or\ q_1' \leftarrow.$

$\vdots$

$q_l\ or\ q_l' \leftarrow.$

$not\_satisfied \leftarrow \hat{\phi_1}.$

$\vdots$

$not\_satisfied \leftarrow \hat{\phi_n}.$

$q_1 \leftarrow not\_satisfied.$

$q_1' \leftarrow not\_satisfied.$

$\vdots$

$q_l \leftarrow not\_satisfied.$

$q_l' \leftarrow not\_satisfied.$

$sat \leftarrow$ **not** $not\_satisfied.$

□

Example 41 Let $F(p_1, p_2, q_1, q_2)$ be the formula $(p_1 \vee p_2 \vee q_1) \wedge (q_1 \vee q_2)$. Consider the QBF $\forall p_1, p_2.\ \exists q_1, q_2.F(p_1, p_2, q_1, q_2)$. It is easy to see that this QBF is satisfiable as $F(p_1, p_2, q_1, q_2)$ evaluates to *true* for the following interpretations:
$\{p_1, p_2, q_1, q_2\}$, $\{p_1, p_2, q_1, \neg q_2\}$, $\{p_1, p_2, \neg q_1, q_2\}$,
$\{p_1, \neg p_2, q_1, q_2\}$, $\{p_1, \neg p_2, q_1, \neg q_2\}$, $\{p_1, \neg p_2, \neg q_1, q_2\}$,
$\{\neg p_1, p_2, q_1, q_2\}$, $\{\neg p_1, p_2, q_1, \neg q_2\}$, $\{\neg p_1, p_2, \neg q_1, q_2\}$,
$\{\neg p_1, \neg p_2, q_1, q_2\}$, and $\{\neg p_1, \neg p_2, q_1, \neg q_2\}$.

Now let us consider the following program based on the construction in Proposition 18.

p_1 *or* $p_1' \leftarrow$.

p_2 *or* $p_2' \leftarrow$.

q_1 *or* $q_1' \leftarrow$.

q_2 *or* $q_2' \leftarrow$.

$not\_satisfied \leftarrow p_1', p_2', q_1'$.

$not\_satisfied \leftarrow q_1', q_2'$.

$q_1 \leftarrow not\_satisfied$.

$q_1' \leftarrow not\_satisfied$.

$q_2 \leftarrow not\_satisfied$.

$q_2' \leftarrow not\_satisfied$.

We will now argue that the only answer sets of the above programs are as follows:

$\{p_1, p_2, q_1, q_2\}$, $\{p_1, p_2, q_1, q_2'\}$, $\{p_1, p_2, q_1', q_2\}$,
$\{p_1, p_2', q_1, q_2\}$, $\{p_1, p_2', q_1, q_2'\}$, $\{p_1, p_2', q_1', q_2\}$,
$\{p_1', p_2, q_1, q_2\}$, $\{p_1', p_2, q_1, q_2'\}$, $\{p_1', p_2, q_1', q_2\}$,
$\{p_1', p_2', q_1, q_2\}$, and $\{p_1', p_2', q_1, q_2'\}$.

It is easy to check that each of the above are closed under the above rules, and each of them is minimal because removing any element from any of the above will not make it closed with respect to one of the first four rules of the program. Thus they are all answer sets of the above programs. Now we have to argue why there are no other answer sets.

First, it is easy to show that we can not have an answer set with both p_1 and p_1' as a subset of it will be always closed under the program rules. Similarly, we can not have an answer set with both p_2 and p_2'.

Now let us consider the possible answer sets that contain p_1 and p_2. In the above list of answer sets we do not have an answer set that contains $\{p_1, p_2, q_1', q_2'\}$. We will now argue that no answer set can contain $\{p_1, p_2, q_1', q_2'\}$. By Proposition 22 (from Chapter 3) any answer set that contains $\{p_1, p_2, q_1', q_2'\}$ must also contain $\{not\_satisfied, q_1, q_1', q_2, q_2'\}$. But then there are proper subsets of this set which are answer sets. Hence, no answer set can contain $\{p_1, p_2, q_1', q_2'\}$.

Now let us enumerate the 16 possible combinations of q_1, q_2, q_1' and q_2' and there interactions with p_1 and p_2. The sixteen combinations and whether they are part of an answer set is are listed below with explanations:

(i) $\{p_1, p_2\}$ (No, not closed under 3rd and 4th rule of the program)
(ii) $\{p_1, p_2, q_1\}$ (No, not closed under 3rd and 4th rule of the program)
(iii) $\{p_1, p_2, q_2\}$ (No, not closed under 3rd and 4th rule of the program)
(iv) $\{p_1, p_2, q_1'\}$ (No, not closed under 3rd and 4th rule of the program)
(v) $\{p_1, p_2, q_2'\}$ (No, not closed under 3rd and 4th rule of the program)
(vi) $\{p_1, p_2, q_1, q_2\}$ (Yes, is an answer set)
(vii) $\{p_1, p_2, q_1, q_1'\}$ (No, not closed under 3rd and 4th rule of the program)
(viii) $\{p_1, p_2, q_1, q_2'\}$ (Yes, is an answer set)
(ix) $\{p_1, p_2, q_2, q_1'\}$ (Yes, is an answer set)
(x) $\{p_1, p_2, q_2, q_2'\}$ (No, not closed under 3rd and 4th rule of the program)

(xi) $\{p_1, p_2, q_1', q_2'\}$ (No, no answer set can contain $\{p_1, p_2, q_1', q_2'\}$)
(xii) $\{p_1, p_2, q_1, q_2, q_1'\}$ (No, its proper subset is an answer set)
(xiii) $\{p_1, p_2, q_1, q_2, q_2'\}$ (No, its proper subset is an answer set)
(xiv) $\{p_1, p_2, q_1, q_1', q_2'\}$ (No, no answer set can contain $\{p_1, p_2, q_1', q_2'\}$)
(xv) $\{p_1, p_2, q_2, q_1', q_2'\}$ (No, no answer set can contain $\{p_1, p_2, q_1', q_2'\}$)
(xvi) $\{p_1, p_2, q_1, q_2, q_1', q_2'\}$ (No, no answer set can contain $\{p_1, p_2, q_1', q_2'\}$)

Similarly, we can argue about the other combinations of p_1, p_2, p_1' and p_2' in terms of why there are no answer sets other than the ones we listed in the beginning. □

Example 42 Let $F(p_1, p_2, q_1, q_2)$ be the formula $(p_1 \vee p_2) \wedge (q_1 \vee q_2)$. Consider the QBF $\forall p_1, p_2.\ \exists q_1, q_2.\ F(p_1, p_2, q_1, q_2)$. It is easy to see that this QBF is not satisfiable as there is no assignment to q_1 and q_2 that will make $F(p_1, p_2, q_1, q_2)$ satisfiable when p_1 and p_2 are both assigned $false$.

Now let us consider the following program based on the construction in Proposition 18.

p_1 *or* $p_1' \leftarrow$.

p_2 *or* $p_2' \leftarrow$.

q_1 *or* $q_1' \leftarrow$.

q_2 *or* $q_2' \leftarrow$.

$not\_satisfied \leftarrow p_1', p_2'$.

$not\_satisfied \leftarrow q_1', q_2'$.

$q_1 \leftarrow not\_satisfied$.

$q_1' \leftarrow not\_satisfied$.

$q_2 \leftarrow not\_satisfied$.

$q_2' \leftarrow not\_satisfied$.

The above logic program has $S = \{p_1', p_2', q_1, q_2, q_1', q_2', not\_satisfied\}$ as one of the answer sets. It is easy to see that S is a model of the above program. We now argue why no subset of S is an answer set of the program. We can not remove either p_1' or p_2' from S as it will no longer be a model by virtue of the first and the second rule. By virtue of the fifth rule, we can not remove $not\_satisfied$ from S. Then by virtue of the last four rules we can not remove any of q_1, q_2, q_1' and q_2' from S. Hence, S is an answer set of the above program.

□

2.1.11 Checking satisfiability of Existential-universal QBFs

We now consider encoding the satisfiability of Existential-universal QBFs using AnsProlog^{or} programs. The use of *or* in these encodings are also essential and can not be eliminated away. This also explains the added expressibility of AnsProlog^{or} programs over AnsProlog programs.

Proposition 19 The QBF $\exists p_1, \ldots p_k.\ \forall q_1, \ldots, q_l.\ (\theta_1(p_1, \ldots, p_k, q_1, \ldots, q_l) \vee \ldots \vee \theta_n(p_1, \ldots, p_k, q_1, \ldots, q_l))$ is satisfiable iff $exist\_satisfied$ is true in at least one answer set of the following program.

p_1 *or* $p_1' \leftarrow$.

$\vdots$

p_k *or* $p'_k \leftarrow$.

q_1 *or* $q'_1 \leftarrow$.

$\vdots$

q_l *or* $q'_l \leftarrow$.

$exist\_satisfied \leftarrow \theta'_1$.

$\vdots$

$exist\_satisfied \leftarrow \theta'_n$.

$q_1 \leftarrow exist\_satisfied$.

$q'_1 \leftarrow exist\_satisfied$.

$\vdots$

$q_k \leftarrow exist\_satisfied$.

$q'_k \leftarrow exist\_satisfied$. □

Proof: Exercise.

Example 43 Let $F(p_1, p_2, q_1, q_2)$ be the formula $(p_1 \wedge q_1) \vee (p_2 \wedge q_2) \vee (\neg q_1 \wedge \neg q_2)$. Consider the QBF $\exists p_1, p_2.\ \forall q_1, q_2. F(p_1, p_2, q_1, q_2)$. It is easy to see that this QBF is satisfiable as $F(p_1, p_2, q_1, q_2)$ evaluates to *true* for the following interpretations:
$\{p_1, p_2, q_1, q_2\}$, $\{p_1, p_2, q_1, \neg q_2\}$, $\{p_1, p_2, \neg q_1, q_2\}$, and $\{p_1, p_2, \neg q_1, \neg q_2\}$

Consider the following program:

p_1 *or* $p'_1 \leftarrow$.

p_2 *or* $p'_2 \leftarrow$.

q_1 *or* $q'_1 \leftarrow$.

q_2 *or* $q'_2 \leftarrow$.

$exist\_satisfied \leftarrow p_1, q_1$.

$exist\_satisfied \leftarrow p_2, q_2$.

$exist\_satisfied \leftarrow q'_1, q'_2$.

$q_1 \leftarrow exist\_satisfied$.

$q'_1 \leftarrow exist\_satisfied$.

$q_2 \leftarrow exist\_satisfied$.

$q'_2 \leftarrow exist\_satisfied$.

The above program has one of the answer sets as $S = \{p_1, p_2, q_1, q_2, q'_1, q'_2, exist\_satisfied\}$.

We will now argue that the only other answer set of the above program are:
$S_1 = \{p_1, p'_2, q'_1, q_2\}$,
$S_2 = \{p'_1, p_2, q_1, q'_2\}$,
$S_3 = \{p'_1, p'_2, q_1, q_2\}$, $S_4 = \{p'_1, p'_2, q_1, q'_2\}$ and $S_5 = \{p'_1, p'_2, q'_1, q_2\}$.

First we will argue why no answer set of the above program can contain $S_6 = \{p_1, p_2', q_1, q_2\}$. Any answer set S' that contains $\{p_1, p_2', q_1, q_2\}$ must also contain $\{exist\_satisfied, q_1, q_2, q_1', q_2'\}$. But then this set will be a proper super set of the answer set $\{p_1, p_2', q_1', q_2\}$, and hence S' can not be an answer set. We can similarly argue that no answer set of the above program can contain any of the following sets.

$S_7 = \{p_1, p_2', q_1, q_2'\}$, $S_8 = \{p_1, p_2', q_1', q_2'\}$,
$S_9 = \{p_1', p_2, q_1, q_2\}$, $S_{10} = \{p_1', p_2, q_1', q_2\}$, $S_{11} = \{p_1', p_2, q_1', q_2'\}$,
and $S_{12} = \{p_1', p_2', q_1', q_2'\}$.

Any other set containing any other combinations of $p_1, p_1', p_2, p_2', q_1, q_1', q_2$ and q_2' which is closed under the rules of the program will be a super set of one of the sets S or $S_1 \ldots S_{12}$, and hence will not be an answer set.

Thus it is interestingly to note that S is the only answer set of the above program that contains $exist\_satisfied$. □

Example 44 Let $F(p_1, p_2, q_1, q_2)$ be the formula $(p_1 \wedge q_1) \vee (p_2 \wedge q_2)$. Consider the QBF $\exists p_1, p_2. \forall q_1, q_2. F(p_1, p_2, q_1, q_2)$. This QBF is not satisfiable as for each interpretation of p_1 and p_2 there is an interpretation of $q_1$and q_2 where $F(p_1, p_2, q_1, q_2)$ evaluates to $false$. Some of these interpretations are as follows:

$\{p_1, p_2, \neg q_1, \neg q_2\}$, $\{p_1, \neg p_2, \neg q_1, q_2\}$, $\{\neg p_1, p_2, q_1, \neg q_2\}$, and $\{\neg p_1, \neg p_2, q_1, q_2\}$.

Consider the following program:

p_1 *or* $p_1' \leftarrow$.

p_2 *or* $p_2' \leftarrow$.

q_1 *or* $q_1' \leftarrow$.

q_2 *or* $q_2' \leftarrow$.

$exist\_satisfied \leftarrow p_1, q_1$.

$exist\_satisfied \leftarrow p_2, q_2$.

$q_1 \leftarrow exist\_satisfied$.

$q_1' \leftarrow exist\_satisfied$.

$q_2 \leftarrow exist\_satisfied$.

$q_2' \leftarrow exist\_satisfied$.

It is easy to show that the following are answer sets of the above program:

$S_1 = \{p_1, p_2, q_1', q_2'\}$,
$S_2 = \{p_1, p_2', q_1', q_2\}$, $S_3 = \{p_1, p_2', q_1', q_2'\}$,
$S_4 = \{p_1', p_2, q_1, q_2'\}$, $S_5 = \{p_1', p_2, q_1', q_2'\}$,
$S_6 = \{p_1', p_2', q_1, q_2\}$, $S_7 = \{p_1', p_2', q_1, q_2'\}$, $S_8 = \{p_1', p_2', q_1', q_2\}$ and $S_9 = \{p_1', p_2', q_1', q_2'\}$.

It can now be argued (similar to the arguments in Example 41) that there are no other answer sets of the above program. □

2.1.12 Smallest, largest, and next in a linear ordering

Suppose we are given a set of distinct objects and a linear ordering among the objects. We now write an AnsProlog program which defines the smallest and the largest object (with respect to the given ordering) in that set and also the *next* object (if any) for each of the objects in the set. The first two rules define *smallest*, the next two rules define *largest*, and the last four rules define *next*.

$not\_smallest(X) \leftarrow object(X), object(Y), less\_than(Y, X).$
$smallest(X) \leftarrow object(X), \textbf{not}\ not\_smallest(X).$

$not\_largest(X) \leftarrow object(X), object(Y), less\_than(X, Y).$
$largest(X) \leftarrow object(X), \textbf{not}\ not\_largest(X).$

$not\_next(X, Y) \leftarrow X = Y.$
$not\_next(X, Y) \leftarrow less\_than(Y, X).$
$not\_next(X, Y) \leftarrow object(X), object(Y), object(Z), less\_than(X, Z), less\_than(Z, Y).$
$next(X, Y) \leftarrow object(X), object(Y), \textbf{not}\ not\_next(X, Y).$

2.1.13 Establishing linear ordering among a set of objects

In the previous section we were given a linear ordering and we only needed to define *smallest*, *largest* and*next*. Now suppose we are only given a set of objects and our goal is to establish a linear ordering among them. Since we do not have reasons to prefer one linear ordering over the other, the following program generates answer sets such that each answer set corresponds to a particular linear ordering among the objects and each linear ordering is captured by exactly one answer set.

The program is divided into two parts. The first part enumerates a ordering *prec* between each pairs of objects. The second part tests if *prec* is a linear ordering or not by trying to define the *smallest*, *largest* and *next* with respect to *prec* and checking if the largest element can be reached from the smallest through the next operator. If it is then *prec* is a linear ordering; otherwise if either of the three conditions is not satisfied (no largest element, or no smallest element or not reachable) it is not a linear ordering. Following is the program where we use the predicates $first$ and *last* instead of *smallest* and *largest*, and *succ* instead of *less_than*.

1. Defining *prec*: The first rules says that between any two objects one must precede the other. The second makes *prec* a transitive relation.

   $prec(X, Y) \leftarrow \textbf{not}\ prec(Y, X), X \neq Y.$
   $prec(X, Z) \leftarrow prec(X, Y), prec(Y, Z).$

2. Defining *succ*: The following rules define when an object is a successor of another object based on the precedence ordering.

   $not\_succ(X, Z) \leftarrow prec(Z, X).$
   $not\_succ(X, Z) \leftarrow prec(X, Y), prec(Y, Z).$
   $not\_succ(X, X) \leftarrow.$
   $succ(X, Y) \leftarrow \textbf{not}\ not\_succ(X, Y).$

3. Defining $first$: The following rules define when an object is first with respect to the precedence ordering.

$not\_first(X) \leftarrow prec(Y, X).$
$first(X) \leftarrow$ **not** $not\_first(X).$

4. Defining *last*: The following rules define when an object is last with respect to the precedence ordering.

   $not\_last(X) \leftarrow prec(X, Y).$
   $last(X) \leftarrow$ **not** $not\_last(X).$

5. Defining reachability: The following rules define which objects are reachable from the first object using the successor relationship.

   $reachable(X) \leftarrow first(X).$
   $reachable(Y) \leftarrow reachable(X), succ(X, Y).$

6. Defining linear orderings and eliminating models that do not have a linear ordering.

   An ordering is tested to be linear by checking if the last element (as defined before) is reachable from the first element through the successor relationship.

   $linear \leftarrow reachable(X), last(X).$
   $inconsistent \leftarrow$ **not** $inconsistent,$ **not** $linear.$

2.1.14 Representing Aggregates

In this subsection we show how aggregate computation can be done using AnsProlog. We demonstrate our program with respect to a small example. Consider the following database, where $sold(a, 10, Jan1)$ means that 10 units of item a was sold on Jan 1.

$sold(a, 10, Jan1) \leftarrow.$
$sold(a, 21, Jan5) \leftarrow.$
$sold(a, 15, Jan16) \leftarrow.$
$sold(b, 16, Jan4) \leftarrow.$
$sold(b, 31, Jan21) \leftarrow.$
$sold(b, 15, Jan26) \leftarrow.$
$sold(c, 24, Jan8) \leftarrow.$

We would like to answer queries such as: "List all items of which more than 50 units (total) were sold, and the total quantity sold for each." Our goal is to develop an AnsProlog program for this. For simplicity, we assume that the same item has not been sold for the same units.

1. The first step is to assign numbers to each tuple of *sold* while grouping them based on their item. In other words we would like the answer sets to contain the following facts (or similar ones with a different numbering).

   $assigned(a, 10, 1) \leftarrow.$
   $assigned(a, 21, 2) \leftarrow.$
   $assigned(a, 15, 3) \leftarrow.$
   $assigned(b, 16, 1) \leftarrow.$

$assigned(b, 31, 2) \leftarrow$.
$assigned(b, 15, 3) \leftarrow$.
$assigned(c, 24, 1) \leftarrow$.

The AnsProlog program with such answer sets has the following three groups of rules:

(a) The following three rules make sure that numbers are uniquely assigned to each pair of item, and units.

$assigned(X, Y, J) \leftarrow sold(X, Y, D), \textbf{not}\ \neg assigned(X, Y, J)$.
$\neg assigned(X, Y, J) \leftarrow assigned(X, Y', J), Y \neq Y'$.
$\neg assigned(X, Y, J) \leftarrow assigned(X, Y, J'), J \neq J'$

(b) The following rules ensure that among the tuples corresponding to each item there is no gap in the number assignment.

$numbered(X, J) \leftarrow assigned(X, Y, J)$.
$\leftarrow numbered(X, J + 1), \textbf{not}\ numbered(X, J), J \geq 1$.

(c) The following rules ensure that for each item, there is a tuple that is assigned the number 1.

$one\_is\_assigned(X) \leftarrow assigned(X, Y, 1)$.
$\leftarrow sold(X, Y, D), \textbf{not}\ one\_is\_assigned(X)$.

An alternative set of rules that can achieve the same purpose is as follows:

$assign\_one(X, Y, 1) \leftarrow sold(X, Y, D), \textbf{not}\ \neg assign\_one(X, Y, 1)$.
$\neg assign\_one(X, Y, 1) \leftarrow assign\_one(X, Y', 1), Y \neq Y'$.
$assigned(X, Y, 1) \leftarrow assign\_one(X, Y, 1)$.

2. Initializing and updating the aggregate operations. Depending on the aggregate operators the *initialize* and *update* facts describe how to start the aggregate process when the first tuple in each grouping is encountered, and how the aggregate valued is updated when additional tuples are encountered. We now write several such facts for a few different aggregate operators:

(a) Sum:

$initialize(sum, Y, Y) \leftarrow$.
$update(sum, W, Y, W + Y) \leftarrow$.
Intuitively, $initialize(sum, Y, Y)$ means that for the aggregate sum, during the aggregation process when the tuple $(\_, Y)$ assigned the initial number (which is 1) is considered, then Y is the value from which the aggregation starts. The aggregation starts from the tuple assigned 1, and runs though the other tuples in the linear order of their assignment. The fact, $update(sum, W, Y, W + Y)$ is used in that process and intuitively means that while doing the aggregation sum, if the next tuple (based on the ordering of its assignment) is $(\_, Y)$, and the current accumulated value is W, then after considering this tuple the accumulated valued is to be updated to $W + Y$.

(b) Count:

$initialize(count, Y, 1)$.
$update(count, W, Y, W + 1) \leftarrow$.

(c) Min:

$initialize(min, Y, Y) \leftarrow.$
$update(min, W, Y, W) \leftarrow W \leq Y.$
$update(min, W, Y, Y) \leftarrow Y \leq W.$

Using the predicates *initialize* and *update* we can define other aggregate operators of our choice. Thus, AnsProlog allows us to express user-defined aggregates.

3. The following three rules describe how the *initialize* and *update* predicates are used in computing the aggregation. The first rule uses *initialize* to account for the tuple that is assigned the number 1, and the second rule encodes the aggregate computation when we already have computed the aggregate up to the Jth tuple, and we encounter the J+1th tuple.

$aggr(Aggr\_name, 1, X, Z) \leftarrow assigned(X, Y, 1), initialize(aggr\_name, Y, Z).$
$aggr(Aggr\_name, J+1, X, Z) \leftarrow J > 0, aggr(Aggr\_name, J, X, W), assigned(X, Y, J+1),$
$update(Aggr\_name, W, Y, Z).$

4. Computing new aggregate predicates: Once the aggregation is done, we can define new predicates for the particular aggregation that we need. Following are some examples of the encoding of such predicates.

(a) Total sold per item:

$total\_sold\_per\_item(X, Q) \leftarrow aggr(sum, J, X, Q),$
$\textbf{not } aggr(sum, J+1, X, Y).$

(b) Number of transactions per item:

$number\_of\_transactions\_per\_item(X, Q) \leftarrow aggr(count, J, X, Q),$
$\textbf{not } aggr(count, J+1, X, Y).$

Here we will need an additional rule if we want to display that for some items the number of transactions is zero. One such rule could be:

$number\_of\_transactions\_per\_item(X, 0) \leftarrow \textbf{not } has\_sold(X).$
$has\_sold(X) \leftarrow sold(X, Y, Z).$

(c) Minimum amount (other than zero) sold per item:

$min\_sold\_per\_item(X, Q) \leftarrow aggr(min, J, X, Q),$
$\textbf{not } aggr(min, J+1, X, Y).$

Using the above program the answer sets we will obtain will contain the following:

$total\_sold\_per\_item(a, 46)$
$total\_sold\_per\_item(b, 62)$
$total\_sold\_per\_item(c, 24)$

2.1.15 Representing classical disjunction conclusions using AnsProlog

Suppose we would like to represent classical disjunction in the head of the rules. I.e, suppose we would like to express rules of the following form which are not part of the AnsProlog* syntax.

$a_1 \vee \ldots \vee a_l \leftarrow a_{l+1}, \ldots, a_m, \textbf{not } a_{m+1}, \ldots, \textbf{not } a_n.$

Here $\vee$ is the classical disjunction. Intuitively, the above rule means that in every answer set where the right hand side is true we would like the left hand side to be true, and unlike when we use *or* , we do not minimize truth. i.e., the program:

a *or* $b \leftarrow p.$
$p \leftarrow.$

will have three answer sets, $\{a, p\}$, $\{b, p\}$, and $\{a, b, p\}$.

This can be achieved by the following translation to AnsProlog.

1. $f' \leftarrow f, \textbf{not } f'.$
2. $f \leftarrow a'_1, \ldots, a'_l, a_{l+1}, \ldots, a_m, \textbf{not } a_{m+1}, \ldots, \textbf{not } a_n.$
3. For $i = 1 \ldots l$ we have
   (a) $a_i \leftarrow \textbf{not } a'_i, a_{l+1}, \ldots, a_m, \textbf{not } a_{m+1}, \ldots, \textbf{not } a_n.$
   (b) $a'_i \leftarrow \textbf{not } a_i, a_{l+1}, \ldots, a_m, \textbf{not } a_{m+1}, \ldots, \textbf{not } a_n.$

Example 45 Consider the following program with classical disjunction in its head.

$a \vee b \leftarrow q$
$q \leftarrow \textbf{not } r.$
$r \leftarrow \textbf{not } q.$

Our intention is that the AnsProlog program obtained by translating the above program should have the answer sets $\{q, a\}, \{q, b\}, \{q, a, b\}$, and $\{r\}$.

The translated AnsProlog program is:

$q \leftarrow \textbf{not } r.$
$r \leftarrow \textbf{not } q.$
$f' \leftarrow f, \textbf{not } f'.$
$f \leftarrow q, a', b'.$
$a \leftarrow \textbf{not } a', q.$
$a' \leftarrow \textbf{not } a, q.$
$b \leftarrow \textbf{not } b', q.$
$b' \leftarrow \textbf{not } b, q.$

and it indeed has the desired answer sets. □

One implication of the existence of the above encoding is that adding classical disjunction to the head of the rules with out any minimization requirement does not increase the expressibility of the language. In fact the same is also true if we allow exclusive-or in the head of rules. The following subsection gives an AnsProlog encoding for such rules.

2.1.16 Representing exclusive-or conclusions using AnsProlog

Suppose we would like to represent

$a_1 \oplus \ldots \oplus a_l \leftarrow a_{l+1}, \ldots, a_m, \textbf{not } a_{m+1}, \ldots, \textbf{not } a_n.$

where $\oplus$ is exclusive-or. Here we require that if the right hand side is true then exactly one atom in the head should be true.

This can be achieved by the following translation to AnsProlog:

1. $f' \leftarrow f, \textbf{not } f'.$
2. $f \leftarrow a'_1, \ldots, a'_l, a_{l+1}, \ldots, a_m, \textbf{not } a_{m+1}, \ldots, \textbf{not } a_n.$
3. For $i = 1 \ldots l$ we have
   (a) $a_i \leftarrow \textbf{not } a'_i, a_{l+1}, \ldots, a_m, \textbf{not } a_{m+1}, \ldots, \textbf{not } a_n.$
   (b) $a'_i \leftarrow \textbf{not } a_i.$
4. For each i, j such that $1 \leq i < j \leq l$ we have the rule:

   $f \leftarrow a_{l+1}, \ldots, a_m, \textbf{not } a_{m+1}, \ldots, \textbf{not } a_n, a_i, a_j.$

2.1.17 Cardinality Constraints

Cardinality and weight constraints are introduced in the logic programming implementation Smodels as an extension of AnsProlog and are given a new semantics and the complexity of the new language is analyzed to be the same as the complexity of AnsProlog. In this section and the next we show how such constraints can be expressed in AnsProlog, without any extensions.

Consider the following cardinality constraint:

$1 \leq \{colored(V, C) : color(C)\} \leq 1 \leftarrow vertex(V).$

The intuitive meaning of the above cardinality constraint is that, for each vertex v, exactly one instance of $colored(v, c)$ should be chosen such that $color(c)$ holds.

The above constraint can be expressed through the following AnsProlog rules:

1. Enumerating the possibilities:

   $colored(V, C) \leftarrow vertex(V), color(C), \textbf{not } not\_colored(V, C).$

   $not\_colored(V, C) \leftarrow vertex(V), color(C), \textbf{not } colored(V, C).$

2. Rules that define $count\_color(V, N)$, where for any vertex v, $count\_color(v, n)$ is true in an answer set A if there are n different facts of the form $colored(v, c)$ with distinct c's in A. These aggregate rules are given in Section 2.1.14.

3. The constraint:

   $\leftarrow vertex(V), number(N), count\_color(V, N), n \neq 1.$

Now consider the cardinality constraint

$l_1 \leq \{colored(V, C) : color(C)\} \leq l_2 \leftarrow vertex(V).$

Intuitively, it means that for every vertex v, $colored(v, c)$ should be true for k distinct cs, where $l_1 \leq k \leq l_2$

To encode such a cardinality constraint we proceed as above except that we have the following constraints instead on the one we had before.

1. $\leftarrow color(V), number(N), count\_color(V, N), n < l_1.$
2. $\leftarrow color(V), number(N), count\_color(V, N), n > l_2.$

Cardinality constraints in the body can be encoded by using the aggregate and then converting the constraints in the previous section into rules. For example if we want define a predicate $three\_six(V)$ on vertices which are assigned between 3 to 6 colors we have the following rule:

1. $three\_six(V) \leftarrow color(V), number(N), count\_color(V, N), 3 \leq N \leq 6.$

together with rules that define $count\_color(V, N)$.

2.1.18 Weight Constraints

Cardinality constraints can be generalized to weight constraints. When cardinality constraints in the head are generalized to weight constraints, instead of count, we can use other aggregates such as *'sum'*. For example, consider the following weight constraint:

$w \leq \{l_1 : w_1, \ldots, l_m : w_m, \textbf{not } l_{m+1} : w_{m+1}, \ldots, \textbf{not } l_n : w_n\} \leq w' \leftarrow p.$

We can encode it in AnsProlog as follows:

1. We give a name to the above rule, say r.
2. We represent the weights of the literals as follows:

   $weight(r, l_1, w_1) \leftarrow.$

   $\vdots$

   $weight(r, l_m, w_m) \leftarrow.$

   $weight(r, l'_{m+1}, w_{m+1}) \leftarrow.$

   $\vdots$

   $weight(r, l'_n, w_n) \leftarrow.$

3. We enumerate the literals in the head of the rule by having the following: We use the predicate *holds* so as to be able to compute the sum of the weights of the literals in the head of r, that hold.

   $holds(R, L) \leftarrow holds(p), weight(R, L, X), contrary(L, L'), \textbf{not } holds(R, L').$

$holds(R, L) \leftarrow holds(p), weight(R, L', X), contrary(L, L'), \textbf{not}\ holds(R, L')$.

In addition we have the definition of *contrary*.

$contrary(l_i, l_i') \leftarrow$.
$contrary(l_i', l_i) \leftarrow$.

4. We define $sum\_holds(R, Wt)$ which sums the weight of all literals in R that hold, following the approach in Section 2.1.14.

5. To eliminate candidate answer sets whose aggregate weight do not satisfy the weight conditions in the head of r we have the following constraints:

   $\leftarrow sum\_holds(r, Wt), Wt < w$.
   $\leftarrow sum\_holds(r, Wt), Wt > w'$.

6. Finally we define which literals hold.

   $holds(L) \leftarrow atom(L), holds(R, L)$.

2.2 Knowledge representation and reasoning modules

In this section we show how we can use AnsProlog* programs to represent various knowledge representation and reasoning modules. We start with the representation of normative statements, and exceptions, weak exceptions and direct contradictions to those statements.

2.2.1 Normative statements, exceptions, weak exceptions and direct contradictions: the tweety flies story

Normative statements are statements of the form "normally elements belonging to a class c have the property p." A good representation of normative statements should at least allow us to easily 'incorporate' information about exceptional elements of c with respect to the property c. A good hallmark of incorporation of such additional information is the property of elaboration tolerance. The measure of elaboration tolerance of a representation is determined by the classes of new information that can be incorporated through local changes to the original representation. We now discuss this issue with respect to one of the oldest examples in non-monotonic reasoning – "Normally birds fly. Penguins are exceptional birds that do not fly."

1. We start with a representation in AnsProlog where the 'Closed World Assumption' about all predicates is part of the semantics. In this representation the AnsProlog program defines when positive literals are true, and negative literals are assumed (by the CWA) to hold if the corresponding positive literal are not forced to be true by the definitions.

   A representation of the knowledge that "Normally birds Fly. Tweety is a bird" that is elaboration tolerant to new knowledge of the kind "Penguins are exceptional birds that do not fly." is as follows:

   The original representation is :

$flies(X) \leftarrow bird(X), \textbf{not}\ ab(X).$
$bird(tweety) \leftarrow.$
$bird(sam) \leftarrow.$

From the above we can conclude that *tweety* and *sam* fly. Next when we are told that *sam* is a penguin, and penguins are exceptional birds that do not fly, we can incorporate this additional knowledge by just adding the following to our original representation.

$bird(X) \leftarrow penguin(X).$
$ab(X) \leftarrow penguin(X).$
$penguin(sam) \leftarrow.$

From the resulting program we can conclude that *tweety* flies, but now we change our earlier conclusion about *sam* and conclude that *sam* does not fly.

Now suppose we get additional knowledge of another class of birds, the ostriches, that are also exceptions. Again, this can be incorporated by simply adding the following rules:

$bird(X) \leftarrow ostritch(X).$
$ab(X) \leftarrow ostritch(X).$

2. Now suppose we want to represent the above information in AnsProlog$^\neg$, where CWA is not hard coded in the semantics. In that case we can write explicit CWA rules about each predicate. The overall representation will then be as follows.

   $flies(X) \leftarrow bird(X), \textbf{not}\ ab(X).$
   $bird(X) \leftarrow penguin(X).$
   $ab(X) \leftarrow penguin(X).$
   $bird(tweety) \leftarrow.$
   $penguin(sam) \leftarrow.$

   $\neg bird(X) \leftarrow \textbf{not}\ bird(X).$
   $\neg penguin(X) \leftarrow \textbf{not}\ penguin(X).$
   $\neg ab(X) \leftarrow \textbf{not}\ ab(X).$
   $\neg flies(X) \leftarrow \textbf{not}\ flies(X).$

   The main advantage of having explicit CWA rules is that if for a particular predicate we do not want to have CWA, then we can simply remove the corresponding explicit CWA rule. This can not be done in an AnsProlog representation.

3. Now let us consider a different kind of elaboration. We would like to add information about *john* who is a wounded bird and wounded birds are subset of birds and are *weakly exceptional* with respect to the property of flying. By this we mean that for wounded birds we can not make a definite conclusion about whether they can fly or not.

   To represent the above elaboration the first step is to remove the CWA rule about *flies* so that we are not forced to conclude one way or other about the flying ability of wounded birds. But we do need to make conclusions about the flying ability of penguins and non-birds. For this we need to add explicit rules that state when certain objects do not fly. The following program does these to changes to our previous formulation.

$flies(X) \leftarrow bird(X), \textbf{not } ab(X).$
$bird(X) \leftarrow penguin(X).$
$ab(X) \leftarrow penguin(X).$
$bird(tweety) \leftarrow.$
$penguin(sam) \leftarrow.$

$\neg bird(X) \leftarrow \textbf{not } bird(X).$
$\neg penguin(X) \leftarrow \textbf{not } penguin(X).$
$\neg ab(X) \leftarrow \textbf{not } ab(X).$

$\neg flies(X) \leftarrow penguin(X).$
$\neg flies(X) \leftarrow \neg bird(X).$

Now we can incorporate the new knowledge about the wounded birds by adding the following rules.

$wounded\_bird(john) \leftarrow.$
$\neg wounded\_bird(X) \leftarrow \textbf{not } wounded\_bird(X).$
$bird(X) \leftarrow wounded\_bird(X).$
$ab(X) \leftarrow wounded\_bird(X).$

It is easy to see that we still conclude *tweety* flies and *sam* does not fly from the resulting program, and about *john* our program does not entail *john* flies, and nor does it entail *john* does not fly.

4. In the previous three representations we had closed world assumption – whether explicit or implicit – about *birds* and *penguins*, and had explicit CWA with respect to *wounded_bird* in the last representation. Now consider the case when our information about *birds*, *penguins* and *wounded_birds* is *incomplete*. By this we mean that we may know that *tweety* is a bird, and know that $swa-21$ is not a bird (despite Southwest airlines claim that it is the state bird of Texas), and for some other objects we may not know whether it is a bird or not. By having CWA we will be forced to conclude that these other objects are not birds. Since we do not want that we will remove the explicit CWA rule about *birds*, *penguins* and *wounded_birds*. The resulting program is as follows:

$flies(X) \leftarrow bird(X), \textbf{not } ab(X).$
$bird(X) \leftarrow penguin(X).$
$ab(X) \leftarrow penguin(X).$
$bird(tweety) \leftarrow.$
$penguin(sam) \leftarrow.$

$\neg flies(X) \leftarrow penguin(X).$
$\neg flies(X) \leftarrow \neg bird(X).$

$wounded\_bird(john) \leftarrow.$
$bird(X) \leftarrow wounded\_bird(X).$
$ab(X) \leftarrow wounded\_bird(X).$

Now suppose we want to reason about the object *et*, which we know to be a bird. Since *et* is not known to be a penguin or an wounded bird, the above program will not entail $ab(et)$,

and hence will entail $flies(et)$. In the absence of CWA about penguins and wounded birds, it is possible that et is a penguin, and in that case our conclusion would be wrong. To avoid such possibly wrong conclusions, we can make some changes to our program so that it is *conservative* in making conclusion about what flies and what does not. The main change we make is in the rules that define ab; while before one of our rule was that *penguins* are abnormal, now we change the rule to that any object that can possibly be a penguin (i.e., we do not know for sure that it is not a penguin) is abnormal. Similarly, we change the ab rule about wounded birds to: any object that can possibly be an wounded bird (i.e., we do not know for sure that it is not an wounded bird) is abnormal. In addition, since we remove the explicit CWA rules about penguins and wounded birds, we must add rules that define when an object is not a penguin and when an object is not an wounded bird. The resulting program with these changes is as follows, with the changed rules underlined.

$flies(X) \leftarrow bird(X), \mathbf{not}\ ab(X).$
$bird(X) \leftarrow penguin(X).$
$\underline{ab(X) \leftarrow \mathbf{not}\ \neg penguin(X).}$
$bird(tweety) \leftarrow.$
$penguin(sam) \leftarrow.$

$\underline{\neg penguin(X) \leftarrow \neg bird(X).}$

$\neg flies(X) \leftarrow penguin(X).$
$\neg flies(X) \leftarrow \neg bird(X).$

$wounded\_bird(john) \leftarrow.$
$\underline{\neg wounded\_bird(X) \leftarrow \neg bird(X).}$
$bird(X) \leftarrow wounded\_bird(X).$
$\underline{ab(X) \leftarrow \mathbf{not}\ \neg wounded\_bird(X).}$

5. Let us now consider the case where we may have explicit information about the non-flying ability of certain birds. We would like our representation to gracefully allow such additions. The last program is not adequate to this task because if we were to add new facts $\{\neg fly(tweety), \neg penguin(tweety), \neg wounded\_bird(tweety)\}$ to it the resulting program will not have a consistent answer set, while intuitively no such break down in reasoning is warranted.

   Such a break down can be avoided by replacing the rule

   $flies(X) \leftarrow bird(X), \mathbf{not}\ ab(X).$ by the rule:

   $flies(X) \leftarrow bird(X), \mathbf{not}\ ab(X), \mathbf{not}\ \neg flies(X).$

   In addition, for *exceptions* we only need to have the rule:

   $\neg flies(X) \leftarrow exceptional\_bird(X).$

   and no longer need a rule of the form:

   $ab(X) \leftarrow \mathbf{not}\ \neg exceptional\_bird(X).$

   For *weak exceptions* we will need the rule:

   $ab(X) \leftarrow \mathbf{not}\ \neg weakly\_exceptional\_bird(X).$

2.2.2 The frame problem and the Yale Turkey shoot

Another important benchmark in the history of knowledge representation and non-monotonic reasoning is the 'frame problem'. the original frame problem is to be able to succinctly represent and reason about what does not change in a world due to an action. The problem arises in avoiding writing explicit axioms for each object that does not change its value due to a particular axiom.

To demonstrate how AnsProlog and its extensions represent the frame problem we consider the Yale Turkey shoot from the literature. In this example, there are two actions: *load* and *shoot*. There are also two fluents – objects that may change their value due to an action: *alive* and *loaded*. In our representation we follow the notation of situation calculus from the literature.

In situation calculus the initial situation is represented by a constant s_0, and the situation arising after executing an action A in a situation S is denoted by $res(A, S)$. The truth value of fluents in a situation is described using the predicate *holds*, where $holds(F, S)$ means that the fluent F holds in the situation S.

1. With CWA about each situation:

   In our first representation we want to represent the information that initially the turkey is alive and the gun is not loaded, and the effect of the actions *load* and *shoot*. Our goal is that our representation should allow us to make conclusion about situations that may arise (i.e., about hypothetical future worlds) if we perform a particular sequence of actions. Such a reasoning mechanism can be used for verifying plans and planning.

   Since *loaded* and *alive* are the only two fluents, we have complete information about these fluents with respect to the initial situation, and our actions *shoot* and *load* are deterministic in our first representation using AnsProlog we will assume CWA and only focus on the truth, with the intention that conclusions about falsity can be reasoned using CWA.

   We start with representing what is known about the initial situation. This can be accomplished by the following rule:

   $holds(alive, s_0) \leftarrow$.

   Since we are using CWA, by not having any explicit information about the fluent *load* in s_0 we can conclude using CWA that $\neg holds(loaded, s_0)$.

   To represent that the fluent *loaded* will be true after executing the action *load* in any arbitrary situation S, we have the following rule, which we refer to as an *effect rule*.

   $holds(loaded, res(load, S)) \leftarrow$.

   To represent the frame axiom (which is a normative statement) saying that fluents which are true normally preserve their value after an action, we have the following rule, which we call the *frame rule* or *inertia rule*.

   $holds(F, res(A, S)) \leftarrow holds(F, S), \mathbf{not}\ \ ab(F, A, S)$.

   The atom $ab(F, A, S)$ in the above rule encodes when a fluent F is abnormal with respect to an action A and S. The intention is to use it in encoding exceptions to the normative statement. We now define particular instances of when $ab(F, A, S)$ is true.

One instance is when the action *shoot* is executed in a situation where *loaded* is true. In this case the fluent *alive* will not remain true after the action. Thus it gives rise to an exception to the inertia rule. This exception is encoded by the following:

$ab(alive, shoot, S) \leftarrow holds(loaded, S).$

Note that in the above formulation we do not have an effect rule for the action *shoot*. This is because among the two fluents *loaded* and *alive*, shoot does not affect the first one, and its effect on the second one is indirectly encoded by the *exception* to the *inertia rule*. This is different from the encoding of the effect of the action *load* on the fluent *loaded* using *effect rules*. This difference is due to our use of CWA to infer $\neg holds(alive, res(shoot, S))$ by not being able to infer $holds(alive, res(shoot, S))$; thus the use of an exception instead of an effect rule.

2. To allow incomplete information about the initial situation:

   Let us now consider the case when we may have incomplete information about the initial situation. In that case we would have to explicitly represent both what we know to be true in the initial situation and what we know to be false in the initial situation. Thus we can no longer use CWA, and use AnsProlog$^\neg$ as our representation language.

   Let us consider the scenario when we know that the turkey is initially alive, but have no idea whether the gun is loaded or not. In this case the initial situation can be represented by the following rule:

   $holds(alive, s_0) \leftarrow.$

   Note that since we are using AnsProlog$^\neg$ we can no longer infer $\neg holds(loaded, s_0)$ using CWA. That is of course what we want. For the same reason we now need two explicit inertia rules, one about the inertia of truth of fluent and another about the inertia of falsity of fluents. These two rules are:

   $holds(F, res(A, S)) \leftarrow holds(F, S), \textbf{not}\ \ ab(F, A, S).$
   $\neg holds(F, res(A, S)) \leftarrow \neg holds(F, S), \textbf{not}\ \ ab(F, A, S).$

   The effect rules due to the action *load* and *shoot* respectively are encoded by the following rules:

   $holds(loaded, res(load, S)) \leftarrow.$
   $\neg holds(alive, res(shoot, S)) \leftarrow holds(loaded, S).$

   Since we have explicit effect rules for both truth and falsity we will also need exceptions for the blocking of the opposite. In other words if A makes F true then we have to block the inference of $\neg holds(F, res(A, S))$. (This was not necessary when we used the CWA in our AnsProlog encoding.) The two exception rules corresponding to the above two effect rules are as follows:

   $ab(loaded, load, S) \leftarrow.$
   $ab(alive, shoot, S) \leftarrow \textbf{not}\ \ \neg holds(loaded, S).$

The use of **not** $\neg holds(loaded, S)$ instead of the simpler $holds(loaded, S)$ in the body of the last rule is because of the possibility that we may have incompleteness about the situations. In that case we want to reason *conservatively.* This can be explained in terms of our particular scenario. Recall that we know that initially *alive* is true an we have no idea if *loaded* is true or not. Now suppose we want to reason about the situation $res(shoot, s_0)$. Since we have $holds(alive, s_0)$ we will conclude $holds(alive, res(shoot, s_0))$ using the first inertia rule unless it is blocked by deriving $ab(alive, shoot, s_0)$. If we had only $holds(loaded, S)$ in the body of the last exception rule, we will not be able to derive $ab(alive, shoot, s_0)$ and hence conclude $holds(alive, res(shoot, s_0))$. Our conclusion would not be correct if in the real world *loaded* is true in s_0. The use of **not** $\neg holds(loaded, S)$ instead of the simpler $holds(loaded, S)$ prevents us from making this possibly wrong conclusion. With the above formulation we neither conclude $holds(alive, res(shoot, s_0))$, nor we conclude $\neg holds(alive, res(shoot, s_0))$, as either would be wrong in one of the possible scenarios: the first when *loaded* is $true$ in s_0, and the second when *loaded* is $false$ in s_0.

Finally, the above encoding also works fine when there is complete information about the initial situation.

3. Allowing backward reasoning:

In the previous two formulations our goal was to hypothetically reason about future situations. Now consider the case when in the beginning we do not know whether *loaded* is true in s_0 or not. Then we are given the oracle $holds(alive, res(shoot, s_0))$, and from this added information we would like to conclude that *loaded* must be false in s_0.

One way to be able to do such reasoning is to enumerate the possible worlds in the initial situation using AnsProlog$^\neg$. In our example we can add the following two rules to our previous formulation.

$holds(alive, s_0) \leftarrow$ **not** $\neg holds(alive, s_0)$.
$\neg holds(alive, s_0) \leftarrow$ **not** $holds(alive, s_0)$.
$holds(loaded, s_0) \leftarrow$ **not** $\neg holds(loaded, s_0)$.
$\neg holds(loaded, s_0) \leftarrow$ **not** $holds(loaded, s_0)$.

The above two rules lead to multiple answer sets each corresponding to one of the possible world about the initial situation. Now we can prune out the worlds that do not lead to the oracle by having the oracle as the following integrity constraint.

$\leftarrow$ **not** $holds(alive, res(shoot, s_0))$.

The above integrity constraint will eliminate the answer sets (of the rest of the program) that do not entail the oracle $holds(alive, res(shoot, s_0))$. Since reducing the number of answer sets means we can make more conclusions, the oracle leads us to additional conclusions. In our particular scenario the integrity constraint will eliminate answer sets where $holds(loaded, s_0)$ is true and in the remaining answer sets we will have $\neg holds(loaded, s_0)$. Thus the additional information provided by the oracle will lead us to the conclusion that $\neg holds(loaded, s_0)$.

2.2.3 Systematic removal of Close World Assumption: an example

Consider the following program that defines the transitive closure notion of ancestor given facts about parents.

$anc(X,Y) \leftarrow parent(X,Y)$.
$anc(X,Y) \leftarrow parent(X,Z), anc(Z,Y)$.

The above program assumes complete information about parent and gives a complete definition of ancestor. I.e., using the above program together with a set of facts about *parent*, we can completely determine about any arbitrary pair (a,b), whether $anc(a,b)$ is true or false.

Now consider the case that where the objects are fossils of dinosaurs dug at an archaeological site, and for pairs of objects (a,b) through tests we can sometimes determine $par(a,b)$, sometimes determine $\neg par(a,b)$, and sometimes neither. This means our knowledge about *par* is not complete. Now the question is how do we define when *anc* is true and when it is false.

To define when *anc* is true we keep the old rules.

$anc(X,Y) \leftarrow parent(X,Y)$.
$anc(X,Y) \leftarrow parent(X,Z), anc(Z,Y)$.

Next we define a predicate $m\_par(X,Y)$ which encodes when X may be a parent of Y. We do this through the following rule.

$m\_par(X,Y) \leftarrow$ **not** $\neg par(X,Y)$.

Using m_par we now define $m\_anc(X,Y)$ which encodes when X may be an ancestor of Y. We do this through the following rule.

$m\_anc(X,Y) \leftarrow m\_par(X,Y)$.
$m\_anc(X,Y) \leftarrow m\_par(X,Z), m\_anc(Z,Y)$.

Now we use m_anc to define when $\neg anc(X,Y)$ is true through the following rule.

$\neg anc(X,Y) \leftarrow$ **not** $m\_anc(X,Y)$.

2.2.4 Reasoning about what is known and what is not

Recall that an AnsProlog$^\neg$ program may have an answer set with respect to which we can neither conclude an atom to be true, nor conclude it to be false. This happens if for atom f, and answer set S, neither f not $\neg f$ is in S. In this case we can say that the truth value of f is unknown in the answer set S. We can write the following rule to make such a conclusion and reason with it further.

$unknown(f) \leftarrow$ **not** f, **not** $\neg f$.

The following program π_{gpa} encodes the eligibility condition of a particular fellowship and the rule (4) below deals with the case when the reasoner is not sure whether a particular applicant is eligible or not. In that case rule (4) forces the reason to conduct an interview for the applicant whose eligibility is unknown.

(1) $eligible(X) \leftarrow highGPA(X)$.
(2) $eligible(X) \leftarrow special(X), fairGPA(X)$.
(3) $\neg eligible(X) \leftarrow \neg special(X), \neg highGPA(X)$.
(4) $interview(X) \leftarrow$ **not** $eligible(X)$, **not** $\neg eligible(X)$.
(5) $fairGPA(john) \leftarrow$.
(6) $\neg highGPA(john) \leftarrow$.

2.3 Notes and references

Many of the encoding techniques discussed in this chapter are folklore in logic programming. We point out here some of the sources that we are aware of. The notion of 'choice' was first presented by Sacca and Zaniolo in [SZ90]. The encodings of quantified boolean formulas with two quantifiers and also the encoding of the linear ordering are due to Eiter and Gottlob [EG93a, EGM94] where they use such encodings to prove the complexity and expressibility of AnsProlog^{or} programs. The encoding for aggregates are based on work by Zaniolo and his colleagues [ZAO93, GSZ93, Sac93]. The use of the "$p \leftarrow$ **not** p." construct to encode integrity constraints, the representation of ex-or and classical disjunction in AnsProlog and the expression of first-order queries are based on the papers by Niemela, Simmons and Soininen [Nie99, NSS99, Sim99]. Most of the knowledge representation and reasoning modules are from Gelfond's papers [GL90, GL91, Gel94] on AnsProlog and AnsProlog$^{\neg}$, and also appear in the survey paper [BG94]. The recent survey paper [MT99] also has many small AnsProlog* modules. In Chapter 5 we consider many more examples and larger AnsProlog* programs.

Chapter 3

Principles and properties of declarative programming with answer sets

3.1 Introduction

In this chapter we present several fundamental results that are useful in *analyzing* and *step-by-step building* of AnsProlog* programs, viewed both as a stand-alone programs and as functions. To analyze AnsProlog* programs we define and describe several properties such as categoricity – presence of unique answer sets, coherence – presence of at least one answer set, computability – answer set computation being recursive, filter-abducibility – abductive assimilation of observations using filtering, language independence – independence between answer sets of a program and the language, language tolerance – preservation of the meaning of a program with respect to the original language when the language is enlarged, functional, compilable to first-order theory, amenable to removal of *or* , amenable to computation by Prolog, and restricted monotonicity – exhibition of monotonicity with respect to a select set of literals.

We also define several subclasses of AnsProlog* programs such as stratified, locally stratified, acyclic, tight, signed, head cycle free and several conditions on AnsProlog* rules such as well-moded and state results about which AnsProlog* programs have what properties. We present several results that relate answer sets of an AnsProlog* program with its rules. We develop the notion of splitting and show how the notions of stratification, local stratification, and splitting can be used in step-by-step computation of answer sets.

For *step by step building* of AnsProlog* programs we develop the notion of conservative extension – where a program preserves its original meaning after additional rules are added to it, and present conditions for programs that exhibit this property. We present several operators such as incremental extension, interpolation, domain completion, input opening and input extension, and show how they can be used for systematically building larger programs from smaller modules.

The rest of the chapter is organized as follows. We first define some of the basic notions and properties and then enumerate many of the sub-classes and their properties. We then consider the more involved properties one-by-one and discuss conditions under which they hold.

3.2 Basic notions and basic properties

3.2.1 Categorical and Coherence

Uniqueness of answer sets is an important property of a program. Programs which have a unique answer-set are called *categorical* . Not all programs are categorical. There are programs with multiple answer sets and with no answer sets at all. The latter will be called *incoherent* . Programs with at least one answer set are called *coherent* . A program Π is said to be *consistent* if $Cn(\Pi)$ – the set of literals entailed by Π – is consistent; Otherwise Π is said to be *inconsistent*. Since programs that are not coherent do not have any answer sets, they entail *Lit*; hence those programs are also inconsistent. Coherence, categoricity, and consistency are important properties of logic programs. In Section 3.3 we consider several subclasses of AnsProlog* programs and categorize them in terms of which ones exhibit the properties of coherence, categoricity, and consistency.

Example 46 The AnsProlog program $\{p \leftarrow \neg p.\}$ is incoherent as it does not have an answer set.

The AnsProlog program $\{a \leftarrow \textbf{not } b., b \leftarrow \textbf{not } a.\}$, although coherent has two answer sets $\{a\}$ and $\{b\}$ and hence is not categorical.

The AnsProlog$^\neg$ program $\Pi = \{p \leftarrow \textbf{not } b., \neg p \leftarrow \textbf{not } b.\}$ is categorical and coherent as it has *Lit* as its unique answer set; but it is not consistent as $Cn(\Pi)$ – which contains both p and $\neg p$ – is not consistent. □

3.2.2 Relating answer sets and the program rules

Suppose we are given an AnsProlog program Π and are told that an atom A belongs to one of its answer set S. What can we infer about the relation between A and Π? An intuitive answer is that there must be at least one rule in Π with A in its head such that its body evaluates to *true* with respect to S. We refer to such a rule as *a support for the atom A, with respect to S in Π*.

Similarly, given a rule r of an AnsProlog$^{\neg, or}$ Π if we are told that its body evaluates to *true* with respect to an answer set S of Π, what can we say about the head of r? An intuitive answer in this case is that, one of the literals in the head must be in S. We now formalize these intuitions as propositions. Theses propositions are very useful when trying to show that a given set of literals is an answer set of a program Π.

Proposition 20 *[MS89]* (a) **Forced atom proposition** : Let S be an answer set of an AnsProlog program Π. For any ground instance – of a rule – of the type $A_0 \leftarrow A_1, \ldots, A_m, \textbf{not } A_{m+1}, \ldots, \textbf{not } A_n$ from Π, if $\{A_1, \ldots, A_m\} \subseteq S$ and $\{A_{m+1}, \ldots, A_n\} \cap S = \emptyset$ then $A_0 \in S$.

(b) **Supporting rule proposition** : If S is an answer-set of an AnsProlog program Π then S is supported by Π. I.e., if $A_0 \in S$, then there exists a ground instance – of a rule – of the type $A_0 \leftarrow A_1, \ldots, A_m, \textbf{not } A_{m+1}, \ldots, \textbf{not } A_n$ in Π such that $\{A_1, \ldots, A_m\} \subseteq S$ and $\{A_{m+1}, \ldots, A_n\} \cap S = \emptyset$. □

Proof: Exercise.

We now enlarge the notion of support for an atom to support for a whole set of atoms and introduce the notion of well-supportedness and relate well-supported models with answer sets.

Definition 9 A Herbrand interpretation S is said to be *well-supported* in an AnsProlog program Π iff there exists a strict well-founded partial order $\prec$ on S such that for any atom $A \in S$, there

exists a rule $A \leftarrow A_1, \ldots, A_m, \textbf{not } A_{m+1}, \ldots, \textbf{not } A_n$ in $ground(\Pi)$, such that $\{A_1, \ldots, A_m\} \subseteq S$ and $\{A_{m+1}, \ldots, A_n\} \cap S = \emptyset$ and for $1 \leq i \leq m$, $A_i \prec A$. □

Proposition 21 *[Fag90]* For any AnsProlog program Π, the well-supported models of Π are exactly the answer sets of Π. □

We now consider AnsProlog$^{\neg, or}$ programs and expand the forced atom proposition and supporting rule proposition for such programs.

Proposition 22 (a) **Forced disjunct proposition** : Let S be an answer set of an AnsProlog$^{\neg, or}$ program Π. For any ground instance – of a rule – of the type (1.2.2) from Π, if $\{L_{k+1} \ldots L_m\} \subseteq S$ and $\{L_{m+1} \ldots L_n\} \cap S = \emptyset$ then there exists an i, $0 \leq i \leq k$ such that $L_i \in S$.

(b) **Exclusive supporting rule proposition** : If S is a consistent answer set of an AnsProlog$^{\neg, or}$ program Π and $L \in S$ then there exists a ground instance – of a rule – of the type (1.2.2) from Π such that $\{L_{k+1} \ldots L_m\} \subseteq S$, and $\{L_{m+1} \ldots L_n\} \cap S = \emptyset$, and $\{L_0 \ldots L_k\} \cap S = \{L\}$. □

Proof: Exercise.

Example 47 Consider the following AnsProlog$^{\neg, or}$ program:

a *or* $b \leftarrow$
b *or* $c \leftarrow$
c *or* $a \leftarrow$

The above program has three answer sets $S_1 = \{a, b\}$, $S_2 = \{a, c\}$, and $S_3 = \{b, c\}$. For the atom a in answer set S_1, the third rule, but not the first rule, of the above program satisfies the conditions of the exclusive supporting rule proposition. For atom b in S_1, it is the second rule. Similarly, it is easy to verify the exclusive supporting rule proposition for the atoms in the other answer sets. □

3.2.3 Conservative extension

When programming in AnsProlog, often we would like to enhance a program Π with additional rules. An important question is under what conditions the new program *preserves* the meaning of the original program. Answers to this and similar questions is very important and useful in systematically developing a large program. We now formally define the notion of conservative extension and present certain conditions under which it holds. We discuss additional aspects of systematically developing large programs in Section 3.8.

Definition 10 Let Π and Π' be ground AnsProlog* programs such that $\Pi \subseteq \Pi'$. We say that Π' is a *conservative extension* for Π if the following condition holds: A is a consistent answer set for Π iff there is a consistent answer set A' for Π' such that $A = A' \cap Lit(\mathcal{L}_\Pi)$. □

The following proposition directly follows from the above definition.

Proposition 23 If a program Π' is a conservative extension of a program Π, then $Cn(\Pi) = Cn(\Pi') \cap Lit(\mathcal{L}_\Pi)$. □

We now present syntactic conditions on AnsProlog$^{\neg}$ programs Π and D such that $\Pi \cup D$ is a conservative extension of Π.

Theorem 3.2.1 *[GP91]* Let $\mathcal{L}_0$ be a language and $\mathcal{L}_1$ be its extension by a set of new predicates. Let Π and D be AnsProlog$^{\neg}$ programs in $\mathcal{L}_0$ and $\mathcal{L}_1$ respectively. If for any rule of the type $L_0 \leftarrow L_1, \ldots, L_m, \textbf{not } L_{m+1}, \ldots, \textbf{not } L_n$ from D, $L_0 \in \mathcal{L}_1 - \mathcal{L}_0$ and $\forall i, i > 0 \;\; L_i \in \mathcal{L}_0$ then if no answer set of Π satisfies the premises of contrary rules (i.e. rules with contrary literals in the head) from D then for all $L \in \mathcal{L}_0$, $\Pi \models L$ iff $\Pi \cup D \models L$ □

The syntactic conditions in the above theorem are very restrictive. The following proposition has broader but semantic conditions and allows Π and D to be AnsProlog$^{\neg, or}$ programs.

Theorem 3.2.2 *[GP96]* Let D and Π be AnsProlog$^{\neg, or}$ programs such that $head(D) \cap lit(\Pi) = \emptyset$ and for any consistent answer set A of Π the program $D \cup (A \cap lit(D))$ is consistent. Then $D \cup \Pi$ is a conservative extension of the program Π. □

Since semantic conditions are difficult to check, in the following proposition we present syntactic conditions but restrict D to be an AnsProlog$^{\neg}$ program.

Proposition 24 *[LT94]* If Π is an AnsProlog$^{\neg, or}$ program, C is a consistent set of literals that does not occur in Π, and D is an AnsProlog$^{\neg}$ program such that for every rule $r \in D$, $head(r) \subseteq C$, and $neg(r) \subseteq lit(\Pi)$ then $D \cup \Pi$ is a conservative extension of Π. □

Example 48 Let Π be the following program:

$p \leftarrow \textbf{not } q.$
$q \leftarrow \textbf{not } p.$

and D be the following program.

$r \leftarrow \textbf{not } r, q.$
$r \leftarrow p.$

The program Π has two answer sets $\{p\}$ and $\{q\}$ and $\Pi \not\models p$. On the other hand $\Pi \cup D$ has a single answer set $\{p, r\}$ and entails p. Hence, $\Pi \cup D$ is not a conservative extension of Π. We now show that none of the conditions of Theorems 3.2.1 and 3.2.2 and Proposition 24 are satisfied with respect to this Π and D.

The first rule of D has r in its body which belongs to the language $\mathcal{L}_1$ of D but is not in $\mathcal{L}_0$, the language of Π. Hence, the conditions of Theorem 3.2.1 are not satisfied.

Although $\{q\}$ is an answer set of Π, the program $D \cup (\{q\} \cap lit(D)) = D \cup \{q\}$ is inconsistent as it does not have any answer set. Hence, the conditions of Theorem 3.2.2 are not satisfied.

With respect to the conditions of Proposition 24 we have $C = \{r\}$. But for the first rule r_1 of D, we have $neg(r_1) = \{r\} \not\subseteq lit(\Pi)$. Hence, the conditions of Proposition 24 are not satisfied. □

3.2.4 I/O Specification of a program

In this subsection we define the notion of a mode (or I/O specification) of a predicate and use it to define the notion of well-moded programs and stable programs. The notion of well-moded programs is useful in identifying for which programs the top-down query answering approach of Prolog is correct, while the notion of stable programs is one of the conditions for language tolerance in Theorem 3.6.3.

By a *mode* for an n-ary predicate symbol p in an AnsProlog we mean a function Σ_p from $\{1, \ldots, n\}$ to the set $\{+, -\}$. In an AnsProlog$^{\neg, or}$ program the mode for p also includes a function $\Sigma_{\neg p}$. If

$\Sigma_p(i) =$ '+' the i is called an *input* position of p and if $\Sigma_p(i) =$ '-' the i is called an *output* position of p. We write Σ_p in the form $p(\Sigma_p(1), \ldots, \Sigma_p(n))$. Intuitively, queries formed by p will be expected to have input positions occupied by ground terms. To simplify the notation, when writing an atom as $p(u, v)$, we assume that u is the sequence of terms filling in the input positions of p and that v is the sequence of terms filling in the output positions. By $l(u, v)$ we denote expressions of the form $p(u, v)$ or **not** $p(u, v)$; $var(s)$ denotes the set of all variables occurring in s. Assignment of modes to the predicate symbols of a program Π is called *input-output specification*.

Definition 11 An AnsProlog rule $p_0(t_0, s_{m+1}) \leftarrow l_1(s_1, t_1), \ldots, l_m(s_m, t_m)$ is called *well-moded* w.r.t. an input output specification if for $i \in [1, m+1]$, $var(s_i) \subseteq \bigcup_{j=0}^{i-1} var(t_j)$.

An AnsProlog program is called well-moded w.r.t. an input-output specification if all its rules are. □

In other words, an AnsProlog rule is well-moded with respect to an input-output specification if

(i) every variable occurring in an input position of a body goal occurs either in an input position of the head or in an output position of an earlier body goal;

(ii) every variable occurring in an output position of the head occurs in an input position of the head, or in an output position of a body goal.

Example 49 Consider the following program Π_1:

$anc(X, Y) \leftarrow par(X, Y)$.
$anc(X, Y) \leftarrow par(X, Z), anc(Z, Y)$.

with the input-output specification $\Sigma_{anc} = (+, +)$ and $\Sigma_{par} = (-, -)$.

We now verify that Π_1 is well-moded with respect to Σ. We first check condition (i) above, and find that Z and Y in the second rule of Π_1 occur in an input position of the body goal $anc(Z, Y)$. The variable Y occurs in an input position in the head, and the variable Z occurs in an output position of the earlier body goal $par(X, Z)$. Thus condition (i) holds. Condition (ii) holds vacuously as no variable occurs in an output position of the head of either rule of Π_1.

Now let us consider the following program Π_2 which is obtained by Π_1 by switching the order of the literals in the body of the second rule.

$anc(X, Y) \leftarrow par(X, Y)$.
$anc(X, Y) \leftarrow anc(X, Z), par(Z, Y)$.

We will show that Π_2 is not well-moded with respect to Σ. We check condition (i) above, and find that Z and Y in the second rule of Π_1 occur in an input position of the body goal $anc(X, Z)$. The variable X occurs in an input position in the head, but the variable Z neither occurs in an input position in the head nor occurs in an output position of an earlier body goal, as there is no earlier body goal. Thus condition (i) is violated.

From the answer set semantics point of view Π_1 and Π_2 are equivalent. The syntactic difference between them comes to the forefront when we view them as Prolog programs, in which case the ordering of the literals in the body of rules matter and a Prolog interpreter differentiates between Π_1 and Π_2. □

We need the following notations to define stable programs. For any term E, by $FreeVar(E)$ we designate the set of free variables that occur in E. Given a mode Σ, for any literal L, $FreeVar^+(L)$ and $FreeVar^-(L)$ are the sets of free variables in the various terms in L, that are moded as $+$ and $-$ respectively.

Definition 12 *(Stable programs)* Let Π be an AnsProlog$^{\neg, or}$ program. A rule $R \in \Pi$ is *stable* with respect to a mode Σ if there exists an ordering $L_1, \ldots, L_k$ of $pos(R)$ such that at least one of the following conditions is satisfied for every variable X that occurs in R.

1. $\forall L \in head(R).X \in FreeVar^+(L)$. (i.e., X occurs in the input position of one of the literals in the head.)
2. $\exists i \in \{1, \ldots, k\}$ such that $X \in FreeVar^-(L_i)$ and $\forall j \in \{1, \ldots, i\}.X \notin FreeVar^+(L_j)$. (i.e., X occurs in the output position of some positive subgoal L_i in the body, and does not occur in the input position of any positive subgoal in the body that comes before L_i.

Program Π is stable with respect to Σ if every rule in Π is stable with respect to Σ, and Π is *stable* if for some Σ, Π is stable with respect to Σ. □

Example 50 Consider the program Π_1 and input-output specification Σ from Example 49. We now argue that Π_1 is stable with respect to Σ.

Let us consider the first rule. The variables X and Y occur in it. Both occur in the input position of the literal $anc(X, Y)$ in the head of the first rule, thus satisfying condition (1). Hence, the first rule is stable with respect to Σ.

Now let us consider the second rule. The variables X, Y and Z occur in it. The variables X and Y occur in the input position of the literal $anc(X, Y)$ in the head of the first rule, thus satisfying condition (1). The variable Z does not occur in the head of the second rule, thus violating condition (1). But it occurs in the output position of the literal $par(X, Z)$ in the body of the second rule and does not occur in the input position of any positive subgoal in the body that comes before $par(X, Z)$, as no literal comes before $par(X, Z)$ in the body. Hence, Z satisfies condition (2). Since the three variables X, Y and Z occurring in the second rule satisfy at least one of the two conditions we have that the second rule is stable with respect to Σ. Therefore, Π is a stable. □

3.2.5 Compiling AnsProlog programs to classical logic: Clark's completion

In this section we present methods to compile AnsProlog programs to theories in classical logic. In Section 3.3 we present conditions when the answer sets of AnsProlog programs have a 1-1 correspondence with the models of the compiled theory and thus can be obtained using classical model generator systems. We first present a simple compilation of propositional AnsProlog programs to propositional theories.

Definition 13 Given a propositional AnsProlog program Π consisting of rules of the form:

$Q \leftarrow P_1, \ldots, P_n, \textbf{not } R_1, \ldots, \textbf{not } R_m.$

its completion $Comp(\Pi)$ is obtained in two steps:

- Step 1: Replace each rule of the above mentioned form with the rule

  $Q \Leftarrow P_1 \wedge \ldots \wedge P_n \wedge \neg R_1 \wedge \ldots \neg R_m$

- Step 2: For each symbol Q, let $Support(Q)$ denote the set of all clauses with Q in the head. Suppose $Support(Q)$ is the set:

  $Q \Leftarrow Body_1$

  $\vdots$

  $Q \Leftarrow Body_k$

  Replace this set with the single formula,

  $Q \Leftrightarrow Body_1 \vee \ldots \vee Body_k$.

  If $Support(Q) = \emptyset$ then replace it by $\neg Q$. □

Example 51 Consider the following program Π:

$p \leftarrow a.$
$p \leftarrow b.$
$a \leftarrow$ **not** $b.$
$b \leftarrow$ **not** $a.$

$Comp(\Pi) = \{p \Leftrightarrow a \vee b, a \Leftrightarrow \neg b, b \Leftrightarrow \neg a\}$, and its models are $\{a, p\}$ and $\{b, p\}$. □

The above definition of completion of propositional AnsProlog programs can also be used for AnsProlog programs – such as programs Π without function symbols – whose grounding $ground(\Pi)$ is finite. In that case we compile $ground(\Pi)$. In presence of function symbols we need the following more elaborate definition of Clark, which can also be used in the absence of function symbols.

Definition 14 Given an AnsProlog program Π consisting of rules of the form:

$Q(Z_1, \ldots, Z_n) \leftarrow P_1, \ldots, P_n,$ **not** $R_1, \ldots,$ **not** $R_m.$

with variables $Y_1, \ldots, Y_d$, where $\{Z_1, \ldots, Z_n) \subseteq \{Y_1, \ldots, Y_d\}$,

its completion $Comp(\Pi)$ is obtained in three steps:

- Step 1: Replace each rule of the above mentioned form with the rule

  $Q(X_1, \ldots, X_n) \Leftarrow \exists Y_1, \ldots, Y_d((X_1 = Z_1) \wedge \ldots \wedge (X_n = Z_n)) \wedge P_1 \wedge \ldots \wedge P_n \wedge \neg R_1 \wedge \ldots \neg R_m.$

- Step 2: For each predicate Q, let $Support(Q)$ denote the set of all clauses with Q in the head. Suppose $Support(Q)$ is the set:

  $Q(X_1, \ldots, X_n) \Leftarrow Body_1$

  $\vdots$

  $Q(X_1, \ldots, X_n) \Leftarrow Body_k$

  Replace this set with the single formula,

  $\forall X_1, \ldots X_n.(Q(X_1, \ldots, X_n) \Leftrightarrow Body_1 \vee \ldots \vee Body_k).$

  If $Support(Q) = \emptyset$ then replace it by $\forall X_1, \ldots X_n.\neg Q(X_1, \ldots, X_n)$. □

- Step 3: Add the following equality theory[1].

  1. $a \neq b$, for all pairs a, b of distinct constants in the language.
  2. $\forall. f(X_1, \ldots, X_n) \neq g(Y_1, \ldots Y_m)$, for all pairs f, g of distinct function symbols.
  3. $\forall. f(X_1, \ldots, X_n) \neq a$, for each constant a, and function symbol f.
  4. $\forall. t[X] \neq X$, for each term $t[X]$ containing X and different from X.
  5. $\forall.((X_1 \neq Y_1) \vee \ldots \vee (X_n \neq Y_n)) \Rightarrow f(X_1, \ldots, X_n) \neq f(Y_1, \ldots, Y_n)$, for each function symbol f.
  6. $\forall.(X = X)$.
  7. $\forall.((X_1 = Y_1) \wedge \ldots \wedge (X_n = Y_n)) \Rightarrow f(X_1, \ldots, X_n) = f(Y_1, \ldots, Y_n)$, for each function symbol f.
  8. $\forall.((X_1 = Y_1) \wedge \ldots \wedge (X_n = Y_n)) \Rightarrow (p(X_1, \ldots, X_n) \Rightarrow p(Y_1, \ldots, Y_n))$, for each predicate symbol p, including $=$.

Given an AnsProlog program Π, if its completion has an Herbrand model then we say that its completion is consistent. If the Herbrand models of $Comp(\Pi)$ coincide with the answer sets of Π then we say that Π is equivalent to its completion. We use these terminology in the summary table in Section 3.3.8. We now illustrate the above definition using an example.

Example 52 Consider the following program Π, a slight modification of the program in Example 3.

$fly(X) \leftarrow bird(X), \textbf{not}\ ab(X).$
$ab(X) \leftarrow penguin(X).$
$ab(X) \leftarrow ostritch(X).$
$bird(X) \leftarrow penguin(X).$
$bird(X) \leftarrow ostritch(X).$
$bird(tweety) \leftarrow.$
$penguin(skippy) \leftarrow.$

$Comp(\Pi)$ consists of the equality theory plus the following:

$\forall X. fly(X) \Leftrightarrow (bird(X) \wedge \neg ab(X))$
$\forall X. ab(X) \Leftrightarrow (penguin(X) \vee ostritch(X))$
$\forall X. bird(X) \Leftrightarrow (penguin(X) \vee ostritch(X) \vee X = tweety)$
$\forall X. penguin(X) \Leftrightarrow (X = skippy)$
$\forall X. \neg ostritch(X)$ □

The following example shows that for some AnsProlog$^{\neg\textbf{not}}$ programs the models of their completion are not necessarily the answer sets of those programs.

Example 53 Suppose that we are given a graph specified as follows:

$edge(a, b) \leftarrow$

$edge(c, d) \leftarrow$

$edge(d, c) \leftarrow$

[1] Often in first order logic with equality '=' is interpreted as the identity relation. In that case we need not axiomatize '=' as a congruent relation and only need the axiom schemata 2, 4, and 7.

and want to describe which vertices of the graph are reachable from a given vertex a. The following rules seems to be a natural candidate for such description:

$reachable(a) \leftarrow$

$reachable(X) \leftarrow edge(Y, X), reachable(Y)$

We clearly expect vertices c and d not to be reachable and this is manifested by the unique answer set $S = \{edge(a, b), edge(c, d), edge(d, c), reachable(a), reachable(b)\}$ of the AnsProlog$\neg$**not** program Π_1 consisting of the above five rules.

Its Clark's completion $Comp(\Pi_1)$ consists of the equality theory plus the following:

$\forall X, Y.\ edge(X, Y) \equiv ((X = a \wedge Y = b) \vee (X = c \wedge Y = d) \vee (X = d \wedge Y = c))$

$\forall X.\ reachable(X) \equiv (X = a \vee \exists Y (reachable(Y) \wedge edge(Y, X)))$

It is easy to see that while $S = \{edge(a, b), edge(c, d), edge(d, c), reachable(a), reachable(b)\}$ is a model of $Comp(\Pi_1)$, $S' = \{edge(a, b), edge(c, d), edge(d, c), reachable(a), reachable(b), reachable(c), reachable(d)\}$ is also a model of $Comp(\Pi_1)$.

But S' is not an answer set of Π_1. Thus while $\Pi_1 \models \neg reachable(c)$ and $\Pi_1 \models \neg reachable(d)$, $Comp(\Pi_1) \not\models \neg reachable(c)$ and $Comp(\Pi_1) \not\models \neg reachable(d)$. □

We have the following general result about completion of AnsProlog programs:

Proposition 25 *[GL88]* Let Π be an AnsProlog program. If M is an answer set of P then M is a minimal model of $Comp(P)$. □

3.3 Some AnsProlog* subclasses and their basic properties

In this section we exploit the structure of AnsProlog* programs to define several subclasses and analyze their basic properties, in particular, coherence, categoricity, computability of answer set computation, relationship with completion, and computability of determining if a program belongs to that sub-class. We focus mostly on AnsProlog programs, as they have been analyzed in more detail in the literature.

3.3.1 Stratification of AnsProlog Programs

In the quest for characterizing the **not** operator of AnsProlog programs, and identifying a sub-class for which the semantics was non-controversial, and computable through an iteration process, one of the early notion was the notion of stratification. It was realized early that recursion through the **not** operator was troublesome in Prolog and, in logic programming it often led to programs without a unique 2-valued semantics. Thus stratification was defined as a notion which forced stratified programs to not have recursion through negation. There are two equivalent definitions of stratification. We start with the first one.

Definition 15 A partition $\pi_0, \ldots, \pi_k$ of the set of all predicate symbols of a AnsProlog program Π is *a stratification* of Π, if for any rule of the type $A_0 \leftarrow A_1, \ldots, A_m, \textbf{not}\ A_{m+1}, \ldots, \textbf{not}\ A_n$, and for any $p \in \pi_s$, $0 \leq s \leq k$
if $A_0 \in atoms(p)$, then:
(a) for every $1 \leq i \leq m$ there is q and $j \leq s$ such that $q \in \pi_j$ and $A_i \in atoms(q)$
(b) for every $m + 1 \leq i \leq n$ there is q and $j < s$ such that $q \in \pi_j$ and $A_i \in atoms(q)$.

A program is called *stratified* if it has a stratification. □

In other words, $\pi_0, \ldots, \pi_k$ is a stratification of Π if for all rules in Π, the predicates that appear only positively in the body of a rule are in strata lower than or equal to the stratum of the predicate in the head of the rule, and the predicates that appear under negation as failure are in strata lower than the stratum of the predicate in the head of the rule.

This stratification of the predicates defines a stratification of the rules to strata $\Pi_0, \ldots, \Pi_k$ where a strata Π_i contains rules whose heads are formed by predicates from π_i. Π_i can be viewed as a definition of relations from π_i. The above condition allows definitions which are mutually recursive but prohibits the use of negation as failure for the yet undefined predicates.

Example 54 An AnsProlog program Π consisting of rules

$p(f(X)) \leftarrow p(X), \mathbf{not}\ \ q(X)$

$p(a) \leftarrow$

$q(X) \leftarrow \mathbf{not}\ \ r(X)$

$r(a) \leftarrow$

is stratified with a stratification $\{r\}, \{q\}, \{p\}$. □

Given a program Π, the *dependency graph*, D_Π, of Π consists of the predicate names as the vertices and $< P_i, P_j, s >$ is a labeled edge in D_Π iff there is a rule r in Π with P_i in its head and P_j in its body and the label $s \in \{+, -\}$ denoting whether P_j appears in a positive or a negative literal in the body of r. Note that an edge may be labeled both by $+$ and $-$. A cycle in the dependency graph of a program is said to be a negative cycle if it contains at least one edge with a negative label.

Proposition 26 *[ABW88]* An AnsProlog program Π is stratified iff its dependency graph D_Π does not contain any negative cycles. □

As in the case of AnsProlog$^{-\mathbf{not}}$ programs, answer sets of stratified AnsProlog program can be computed in an iterated fashion. Recall that the operator used in the iterative characterization of AnsProlog$^{-\mathbf{not}}$ program in (1.3.4) was as follows:

$T^0_\Pi(I) = \{L_0 \in HB_\Pi \mid \Pi \text{ contains a rule } L_0 \leftarrow L_1, \ldots, L_m \text{ such that } \{L_1, \ldots, L_m\} \subseteq I \text{ holds }\}$.

We mentioned in Section 1.3.2 that this operator can be extended to AnsProlog programs in the following way:

$T_\Pi(I) = \{L_0 \in HB_\Pi \mid \Pi \text{ contains a rule } L_0 \leftarrow L_1, \ldots, L_m, \mathbf{not}\ L_{m+1}, \ldots, \mathbf{not}\ L_n \text{ such that } \{L_1, \ldots, L_m\} \subseteq I \text{ holds and } \{L_{m+1}, \ldots, L_n\} \cap I = \emptyset\}$.

But unlike T^0_Π, the operator T_Π is not monotone. Consider the program $\{a \leftarrow \mathbf{not}\ b\}$, and let $I = \emptyset$, and $I' = \{b\}$. It is easy to see that $T_\Pi(I) = \{a\}$ while $T_\Pi(I') = \emptyset$. Thus, even though $I \subset I'$, $T_\Pi(I) \not\subseteq T_\Pi(I')$. Since T_Π is not monotone, the Knaster-Tarski theorem is no longer applicable and the equation that the least Herbrand model is equal to $T_\Pi \uparrow \omega$ is equal to the least fixpoint of T_Π is no longer true.

Nevertheless, for a stratified AnsProlog program Π that can be stratified to strata $\Pi_0, \ldots, \Pi_k$, the answer set A of Π can be obtained as follows:

For any program P we define:

$T_P \uparrow 0(I) = I$
$T_P \uparrow (n+1)(I) = T_P(T_P \uparrow n(I)) \cup T_P \uparrow n(I)$
$T_P \uparrow \omega(I) = \bigcup_{n=0}^{\infty} n(I)$

and then we define

$M_0 = T_{\Pi_0} \uparrow \omega(\emptyset)$
$M_1 = T_{\Pi_1} \uparrow \omega(M_0)$
$\vdots$
$M_k = T_{\Pi_k} \uparrow \omega(M_{k-1})$
$A = M_k$

The above construction leads to the following theorem describing an important property of stratified programs.

Proposition 27 Any stratified AnsProlog program is categorical and A as defined above is its unique answer set. □

The following example illustrates the multi-strata iterated fixpoint computation of the unique answer set of AnsProlog programs.

Example 55 Consider the program Π consisting of the following rules:

$a(1) \leftarrow$ **not** $b(1)$.
$b(1) \leftarrow$ **not** $c(1)$.
$d(1) \leftarrow$.

Its predicates can be stratified to the strata: $\pi_0 = \{c, d\}$, $\pi_1 = \{b\}$ and $\pi_2 = \{a\}$. This leads to the following strata of programs. $\Pi_0 = \{d(1) \leftarrow .\}$, $\Pi_1 = \{b(1) \leftarrow$ **not** $c(1).\}$, and $\Pi_2 = \{a(1) \leftarrow$ **not** $b(1).\}$. We now use the iteration method to compute the answer set of Π.

$M_0 = T_{\Pi_0} \uparrow \omega(\emptyset)$ is computed as follows:
$T_{\Pi_0} \uparrow 0(\emptyset) = \emptyset$
$T_{\Pi_0} \uparrow 1(\emptyset) = T_{\Pi_0}(T_{\Pi_0} \uparrow 0(\emptyset)) \cup T_{\Pi_0} \uparrow 0(\emptyset) = T_{\Pi_0}(\emptyset) \cup \emptyset = \{d(1)\}$
$T_{\Pi_0} \uparrow 2(\emptyset) = T_{\Pi_0}(T_{\Pi_0} \uparrow 1(\emptyset)) \cup T_{\Pi_0} \uparrow 1(\emptyset) = T_{\Pi_0}(\{d(1)\}) \cup \{d(1)\} = \{d(1)\} = T_{\Pi_0} \uparrow 1(\emptyset)$
Hence, $M_0 = T_{\Pi_0} \uparrow \omega(\emptyset) = \{d(1)\}$.

$M_1 = T_{\Pi_1} \uparrow \omega(M_0)$ is computed as follows:
$T_{\Pi_1} \uparrow 0(M_0) = M_0$
$T_{\Pi_1} \uparrow 1(M_0) = T_{\Pi_1}(T_{\Pi_1} \uparrow 0(M_0)) \cup T_{\Pi_1} \uparrow 0(M_0) = T_{\Pi_1}(M_0) \cup M_0 = \{b(1)\} \cup \{d(1)\} = \{b(1), d(1)\}$
$T_{\Pi_1} \uparrow 2(M_0) = T_{\Pi_1}(T_{\Pi_1} \uparrow 1(M_0)) \cup T_{\Pi_1} \uparrow 1(M_0) = T_{\Pi_1}(\{b(1), d(1)\}) \cup \{b(1), d(1)\} = \{b(1), d(1)\} = T_{\Pi_1} \uparrow 1(M_0)$
Hence, $M_1 = T_{\Pi_1} \uparrow \omega(M_0) = \{b(1), d(1)\}$.

$M_2 = T_{\Pi_2} \uparrow \omega(M_1)$ is computed as follows:
$T_{\Pi_2} \uparrow 0(M_1) = M_1$
$T_{\Pi_2} \uparrow 1(M_1) = T_{\Pi_2}(T_{\Pi_2} \uparrow 0(M_1)) \cup T_{\Pi_2} \uparrow 0(M_1) = T_{\Pi_2}(M_1) \cup M_1 = \{\} \cup \{b(1), d(1)\} = \{b(1), d(1)\} = T_{\Pi_2} \uparrow 0(M_1)$
Hence, $A = M_2 = T_{\Pi_2} \uparrow \omega(M_1) = \{b(1), d(1)\}$ is the answer set of Π. □

The following proposition uses the notion of stratification of AnsProlog programs to give sufficiency conditions for the categoricity of AnsProlog$^{\neg}$ programs.

Proposition 28 *[GL90]* An AnsProlog$^{\neg}$ program Π is categorical if

(a) Π^+ is stratified, and

(b) The answer set of Π^+ does not contain atoms of the form $p(t)$, $p'(t)$. □

3.3.2 Stratification of AnsProlog^{or} programs

The definition of stratification can be extended to AnsProlog^{or} programs in a straight forward way by requiring that the dependency graph – whose definition is directly applicable to AnsProlog^{or} programs – does not contain any negative cycles. The following proposition guarantees existence of answer sets for stratified AnsProlog^{or} programs.

Proposition 29 Any stratified AnsProlog^{or} program has an answer set. □

Let us look at a few simple examples of AnsProlog^{or} programs, and their answer sets.

Let $\Pi_0 = \{p(a)\ or\ \ p(b) \leftarrow\}$.

It is easy to see that $\{p(a)\}$ and $\{p(b)\}$ are the only answer sets of Π_0 since they are the only minimal sets closed under its rule.

Let $\Pi_1 = \Pi_0 \cup \{r(X) \leftarrow \textbf{not}\ \ p(X)\}$.

Obviously, this program is stratified and hence by Proposition 29 has an answer set S. By part (a) of the Proposition 22, S must either contain $p(a)$ or contain $p(b)$. Part (b) of the Proposition 22 guarantees that S does not contain both. Suppose S contains $p(a)$. Then, by part (a), S contains $r(b)$, and by part (b), it contains nothing else, and hence, $\{p(a), r(b)\}$ is an answer set of Π_1. Similarly, we can show that $\{p(b), r(a)\}$ is an answer set of Π_1 and that there are no other answer sets.

3.3.3 Call-consistency

Stratified AnsProlog programs have a unique answer set. In this subsection we introduce a subclass of AnsProlog which contains many non-stratified programs but guarantees coherence – the existence of at least one answer set.

Definition 16 An AnsProlog program is said to be *call-consistent* if its dependency graph does not have a cycle with odd number of negative edges. □

Call-consistent AnsProlog programs are a super set of stratified AnsProlog programs. An example of a call-consistent program which is not stratified is as follows:

$p(a) \leftarrow q(a)$.
$q(a) \leftarrow p(a)$.

The following two propositions describe a property of call-consistent AnsProlog programs and a property of a more restricted class of call-consistent AnsProlog programs.

Proposition 30 If Π is a call-consistent AnsProlog program then $comp(\Pi)$ has Herbrand models. □

Proposition 31 *[Fag90]* A call-consistent AnsProlog program whose dependency graph does not have a cycle with only positive edges has at least one answer set. □

3.3.4 Local stratification and perfect model semantics

The notion of stratification partitions predicates to strata. The following more general notion of local stratification partitions atoms to strata and leads to the result that locally stratified AnsProlog programs preserve the categoricity property of stratified AnsProlog programs.

Definition 17 An AnsProlog program Π is *locally stratified* if there exists a mapping λ from HB_Π to the countable ordinal such that for every $A_0 \leftarrow A_1, \ldots, A_m, \textbf{not } A_{m+1}, \ldots, \textbf{not } A_n$ in $ground(\Pi)$, the following conditions hold for every $1 \leq i \leq n$:

- $1 \leq i \leq m$: $\lambda(A_0) \geq \lambda(A_i)$.
- $m+1 \leq i \leq n$: $\lambda(A_0) > \lambda(A_i)$. □

Note that the following program

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

is not locally stratified as there does not exist a mapping that satisfies the condition in Definition 17. Unlike the definition of stratification in terms of the dependency graph, we can not define local stratification in terms of not having a loop with negative edges in a more general dependency graph with atoms as nodes. (Such a definition would label the above program as locally stratified.) This is because while the dependency graph has a finite number of nodes, a more general atom dependency graph with atoms as nodes may have infinite number of nodes. In a later section (Section 3.3.6) we give an alternative definition of local stratification using the atom dependency graph.

On the other hand the following program

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

is locally stratified and the mapping $\lambda(q(a)) = 0$, and $\lambda(p(f^n(a))) = n$ satisfies the condition in Definition 17. Its unique answer set is $\{q(a), p(f(a)), p(f(f(f(a)))), \ldots\}$.

Proposition 32 Locally stratified AnsProlog programs are categorical. □

Proposition 33 The unique answer set of locally stratified AnsProlog programs, referred to as the *perfect model* is its least Herbrand model with respect to the following ordering ($\leq_\lambda$) that incorporates a mapping λ satisfying the conditions of Definition 17.

For an interpretation $I \subset HB_\Pi$ and an integer j , let $I[[j]] = \{a : a \in I$ such that $\lambda(a) = n\}$.

$I \leq_\lambda I'$ if there exists an integer k such that, for all $i \leq k$, $I[[i]] = I'[[i]]$ and $I[[k]] \subseteq I'[[k]]$ □

Example 56 Consider the program consisting of the following rule:

$a \leftarrow \textbf{not } b$.

It has three models $I = \{a\}$, $I' = \{b\}$, and $J = \{a, b\}$, out of which $I = \{a\}$ and $I' = \{b\}$ are minimal models. We will now show that it has a unique perfect model $\{a\}$ which is its answer set.

The above program is locally stratified and the mapping $\lambda(b) = 1$, $\lambda(a) = 2$ satisfies the conditions of Definition 17. It is easy to see that $I <_\lambda J$ and $I' <_\lambda J$. Now let us compare I and I'. We have $I[[1]] = \{\}$, $I[[2]] = \{a\}$, $I'[[1]] = \{b\}$, and $I'[[2]] = \{\}$. Since, $I[[1]] \subseteq I'[[1]]$ we have $I \leq_\lambda I'$. But since, $I'[[1]] \not\subseteq I[[1]]$ we have $I' \not\leq_\lambda I$. Hence, $I <_\lambda I'$ and I is the least among the three models. Therefore I is the perfect model of the above program. The reader can easily verify that I is also the unique answer set of this program. □

3.3.5 Acyclicity and tightness

All the subclasses of AnsProlog that we have considered so far do not guarantee that in the presence of function symbols determining an answer set or determining entailment is Turing computable. In this subsection we present such a subclass: acyclic AnsProlog programs, which is a sub-class of locally stratified programs. Such programs not only have unique answer sets but their unique answer set is computable by a Turing machine.

Definition 18 An AnsProlog program Π is *acyclic* if there exists a mapping λ from HB_Π to the the set of natural numbers such that for every $A_0 \leftarrow A_1, \ldots, A_m, \textbf{not } A_{m+1}, \ldots, \textbf{not } A_n$ in $ground(\Pi)$, and for every $1 \leq i \leq n$: $\lambda(A_0) > \lambda(A_i)$. □

Note that the program consisting of the single rule

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

is not acyclic. It is locally stratified though. Similarly the program consisting of the single rule

$p \leftarrow p$.

is also not acyclic but locally stratified. Another historical fact about properties of acyclic programs is that when they were discovered, most alternative semantics proposed for programs with AnsProlog syntax agreed with each other respect to acyclic programs. From that perspective, while the answer set of the non-acyclic program $\Pi = \{p \leftarrow p.\}$ is $\{\}$, and $\Pi \models \neg p$, some researchers argued that p should have the truth value *unknown* with respect to Π. We further discuss some of the alternative semantics of programs with AnsProlog syntax in Chapter 9.

Another important property about acyclic AnsProlog programs were that in conjunction with additional conditions, it guaranteed that the top down derivation procedure referred to as SLDNF [Cla78, Llo87] was sound and complete with respect to the answer set semantics. We now formally state these and other results about acyclic AnsProlog programs.

Proposition 34 *[AB90]* Let Π be an acyclic AnsProlog program. Then we have:
(i) Π has a unique Turing computable answer set;
(ii) The unique answer set of Π is the unique Herbrand model of $Comp(\Pi)$;
(iii) For all ground atoms A that do not flounder[2] , $\Pi \models A$ iff there is an SLDNF derivation of A from Π. □

Although at first glance, acyclic programs may seem to be too restrictive, many useful AnsProlog programs such as the one in part (1) of Section 2.2.2 are acyclic. Moreover, in Chapter 5 we show several AnsProlog programs to be acyclic and analyze them using Proposition 34.

Recently, in presence of several fast propositional solvers, one of the computation methods that have been used to compute answer sets of function-free AnsProlog programs is through computing the models of $Comp(\Pi)$. This has motivated identification of more general classes than acyclic programs that guarantee a one-to-one correspondence between answer sets and models of the completion. We now present two such generalizations.

[2]Intuitively, we say A flounders with respect to Π if while proving A from Π using SLDNF-derivation a goal is reached which contains only non-ground negative literals. For a precise definition see [Llo87].

Definition 19 *(Tight programs)* An AnsProlog program Π is said to be *tight (or positive order consistent)*, if there exists a function λ from HB_Π to the the set of natural numbers such that for every $A_0 \leftarrow A_1, \ldots, A_m, \textbf{not } A_{m+1}, \ldots, \textbf{not } A_n$ in $ground(\Pi)$, and for every $1 \leq i \leq m$: $\lambda(A_0) > \lambda(A_i)$. □

Proposition 35 *[Fag94]* For any propositional AnsProlog program, if Π is tight then X is an answer set of Π iff X is a model of $Comp(\Pi)$. □

Example 57 Consider the following program:

$p(a) \leftarrow \textbf{not } p(b)$.
$p(b) \leftarrow \textbf{not } p(a)$.

The above program is neither acyclic, not locally stratified. But it is tight. It has two answer sets $\{p(a)\}$ and $\{p(b)\}$, which are the two models of its completion consisting of the equality theory and the formula $\forall X.p(X) \Leftrightarrow (X = a \wedge \neg p(b)) \vee (X = b \wedge \neg p(a))$. □

The notion of tight programs was further generalized in two respects, to AnsProlog$^\neg$ program and with respect to a set of literal, by the following definition.

Definition 20 *(Tightness on a set of literals)* An AnsProlog$^{\neg,\bot}$ program Π is said to be *tight on a set X of literals*, if there exists a partial mapping λ with domain X from literals to the set of natural numbers such that for every rule $L_0 \leftarrow L_1, \ldots, L_m, \textbf{not } L_{m+1}, \ldots, \textbf{not } L_n$ in $ground(\Pi)$, if $L_0, \ldots, L_m \in X$, then for every $1 \leq i \leq m$: $\lambda(L_0) > \lambda(L_i)$. □

In the above definition it should be noted that $\bot$ is not considered a literal. The following example illustrate the difference between the original notion of tightness and the notion of tightness with respect to a set of literals.

Example 58 Consider the program consisting of the only rule

$p \leftarrow p$.

This program is obviously not tight. But it is tight on the set of literals $\{\}$. □

Proposition 36 *[BEL00]* For any AnsProlog$^{\neg,\bot}$ program Π and any consistent set X of literals such that Π is tight on X, X is an answer set of Π iff X is closed under and supported by Π.

Proposition 37 *[BEL00]* For any propositional AnsProlog program and any set X of atoms such that Π is tight on X, X is an answer set of Π iff X is a model of $Comp(\Pi)$. □

Example 59 Let us reconsider the program Π consisting of the only rule

$p \leftarrow p$.

This program is tight on the set of literals $S = \{\}$, and is not tight on the set of literals $S' = \{p\}$.

The completion of this program is $p \Leftrightarrow p$ and has the models S and S'. As suggested by Proposition 37 since Π is tight on S, S is an answer set of Π iff S is a model of $Comp(\Pi)$. Since Π is not tight on S', S' being a model of $Comp(\Pi)$ has no consequences. □

3.3.6 Atom dependency graph and order-consistency

One of the maximal subclass of AnsProlog programs that guarantee an answer set is the class of *order-consistent* (or local call-consistent) programs. To define this class of program we need to introduce the following notions.

- The *atom dependency graph* AD_Π of a program Π: The nodes are elements of HB_Π. $\langle A_i, A_j, s\rangle$ is a labeled edge in AD_Π iff there is a rule r in $ground(\Pi)$ with A_i in its head, and A_j in its body, and the label $s \in \{+, -\}$, denoting whether A_j appears in a positive or a negative literal in the body of r.
- We say q *depends evenly on* p denoted by $p \leq_+ q$ if there is a path from p to q in AD_Π with an even number of negative edges.
- We say q *depends oddly on* p denoted by $p \leq_- q$ if there is a path from p to q in AD_Π with an odd number of negative edges.
- We say q *depends on* p denoted by $p \leq q$ if $p \leq_+ q$ or $p \leq_- q$.
- We say q *depends even-oddly on* p denoted by $p \leq_{+-} q$ if $p \leq_+ q$ and $p \leq_- q$.
- We say q *depends positively on* p denoted by $p \leq_0 q$ if there is a non-empty path from p to q with all edges labeled as positive.
- A binary relation (not necessarily a partial order) is *well-founded* if there is no infinite decreasing chain $x_0 \geq x_1 \geq \ldots$. (Note: well-founded implies acyclic but not vice-versa.)

We are now ready to define order-consistency.

Definition 21 An AnsProlog program Π is said to be *order consistent* if the relation $\leq_{+-}$ in AD_Π is well-founded. □

The following two proposition define useful properties of order-consistent AnsProlog programs.

Proposition 38 *[Sat90, CF90]* If Π is an order-consistent AnsProlog program then $comp(\Pi)$ has a Herbrand model. □

Proposition 39 *[Fag90]* An order-consistent AnsProlog program has an answer set. □

A superclass of order-consistent programs which also has useful properties under certain restrictions is the class of negative cycle free programs defines as follows.

Definition 22 An AnsProlog program Π is said to be *negative cycle free* if $\leq_-$ is irreflexive in AD_Π. □

The following two proposition define useful properties of negative cycle-free AnsProlog programs under certain restrictions.

Proposition 40 *[Sat90]* If Π is a negative cycle free AnsProlog program and is either function free or internal variable free (i.e., for any rule the variables in the premise appear in the conclusion) then $Comp(\Pi)$ has a Herbrand model. □

Proposition 41 *[Fag90]* If Π is a negative cycle free and tight AnsProlog program and is either function free or internal variable free (i.e., for any rule the variables in the premise appear in the conclusion) then Π has an answer set. □

Similar to order consistency, we can define the notion of *predicate-order-consistency* for AnsProlog programs by defining the relation $\leq_{+-}$ among predicates in the dependency graph D_Π. This notion is used in Section 3.6.3 as a condition for language tolerance.

Definition 23 An AnsProlog program Π is said to be *predicate-order-consistent* if the relation $\leq_{+-}$ in D_Π is well-founded. □

The various orderings defined in this section can be used to give alternative definitions for local stratified programs and tight programs.

Proposition 42 An AnsProlog program Π is locally stratified iff the relation of dependency through at least one negative edge in AD_Π is well-founded. □

Proposition 43 An AnsProlog program is tight (or positive-order consistent) iff $\leq_0$ is well-founded. □

Example 60 Consider the following AnsProlog program:

$p(X) \leftarrow p(s(X))$
$p(X) \leftarrow \textbf{not}\ p(s(X))$

The above program is negative cycle-free, but not order consistent, nor tight because of the first rule, nor locally stratified because of the second rule.

Since the above program is internal variable free, its completion has a model. In fact in the only model of its completion p is true everywhere. However, this model is not well-supported, and hence is not an answer set. Therefore this program does not have an answer set. □

3.3.7 Signing

One of the early motivation behind studying signed programs was to find conditions on programs and predicates in them such that adding new facts about certain predicates only increased the set of ground atoms that could be concluded from the program. This property is a special case of the notion "restricted monotonicity". Besides that signed AnsProlog programs are coherent and some of their answer sets are related to the well-founded semantics. In this subsection we briefly discuss the last two properties and consider the restricted monotonicity aspect in a later section (Section 3.4).

Intuitively, a signed AnsProlog program is a program whose Herbrand base can be partitioned to two sets such that for any rule the atom in the head and the atoms in the body that are not preceded by **not** belong to the same partition and the atom in the head and the atoms in the body that are preceded by **not** belong to the opposite partitions. More formally,

Definition 24 An AnsProlog program is said to be signed if there is a set S (called signing) of ground atoms such that, for any ground instance of a rule of the type (1.2.2), either
$\{L_0, L_1, \ldots, L_m\} \subset S$ and $\{L_{m+1}, \ldots, L_n\} \cap S = \emptyset$
or
$\{L_0, L_1, \ldots, L_m\} \cap S = \emptyset$ and $\{L_{m+1}, \ldots, L_n\} \subset S$. □

We now present a result which shows that how to obtain one (or possibly two) answer sets of a signed AnsProlog program using $lfp(\Gamma^2_\Pi)$ and $gfp(\Gamma^2_\Pi)$ from Section 1.3.6; and since $lfp(\Gamma^2_\Pi)$ and $gfp(\Gamma^2_\Pi)$ always exist, we have the corollary that signed AnsProlog programs are coherent.

Proposition 44 For an AnsProlog program Π with signing S, the following are among the answer sets of Π:

1. $lfp(\Gamma^2_\Pi) \cup (gfp(\Gamma^2_\Pi) \cap \bar{S})$
2. $lfp(\Gamma^2_\Pi) \cup (gfp(\Gamma^2_\Pi) \cap S)$ □

Note that the above two may be the same, and the signed AnsProlog program may just have a single answer set. On the other hand a signed AnsProlog program may have additional answer sets beyond the ones described by the above theorem. For example, the following AnsProlog program has $\{a, d\}$ as one of its answer sets, which is not dictated by the above theorem.

$a \leftarrow$ **not** b.
$b \leftarrow$ **not** a.
$c \leftarrow$ **not** b, **not** d.
$d \leftarrow$ **not** c.
An additional connection between the answer set semantics of signed AnsProlog programs and the well-founded semantics of these programs is as follows:

Proposition 45 *[Dun92]* For a ground atom p, and a signed AnsProlog program Π, $\Pi \models p$ iff p is true with respect to the well-founded semantics of Π. □

3.3.8 The relation between the AnsProlog subclasses: a summary

In this section we have discussed several sub-classes of AnsProlog programs including stratified, acyclic, call-consistent, locally stratified, tight, signed, order-consistent, and negative cycle-free. We now give some examples that further illustrate the differences and relationship between these classes, show the relationship between the classes in a graph, and summarize the properties of these classes in a table.

1. AnsProlog$^{-\mathbf{not}}$ programs, also referred to as definite programs are a sub-class of signed programs, as for any definite program Π, its Herbrand base is one of its signing.

2. The class of stratified programs is a subclass of the class of call-consistent program. Checking whether a program is stratified and checking whether it is call-consistent are decidable problems. The program:

   $p(a) \leftarrow$ **not** $q(a)$
   $q(a) \leftarrow$ **not** $p(a)$.

   is call-consistent, but not stratified.

3. The class of acyclic programs is a subclass of the class of locally stratified programs, and also a subclass of the class of tight programs. The class of tight programs and the class of locally stratified programs are not related. The program:

   $p(a) \leftarrow p(a)$.

is locally stratified, but not acyclic and not tight. The program:

$p(a) \leftarrow$ **not** $p(b)$.
$p(b) \leftarrow$ **not** $p(a)$.

is tight, but not locally stratified and not acyclic. The program:

$p(a) \leftarrow$ **not** $p(a)$.

is tight, but not locally stratified and not acyclic.

Properties / Subclasses	Coherence (has ans set)	Categorical (One ans set)	Computing answer-set	Relation with completion	subclass determination
neg. cycle free (Defn 22)	under cond. (Prop 41)	not nec.	non-recursive	cons. under cond. (Prop 40)	non-recursive
order-consistent (Defn 21)	yes (Prop 39)	not nec.	non-recursive	consistent (Prop 38)	non-recursive
Call-consistent (Defn 16)	under cond. (Prop 31)	not nec.	non-recursive	consistent (Prop 30)	recursive
locally stratified (Defn 17)	yes	yes (Prop 32)	non-recursive	consistent	non-recursive
Stratified (Defn 15)	yes	yes (Prop 27)	non-recursive	consistent	recursive
signed (Defn 24)	yes (Prop 44)	not nec.	non-recursive	consistent	non-recursive?
tight (Defn 19)	under cond. (Prop 41)	not nec.	non-recursive	equivalent (Prop 35)	non-recursive
acyclic (Defn 18)	yes	yes (Prop 34)	recursive (Prop 34)	equivalent (Prop 34)	non-recursive
definite (AnsProlog$^{\neg}$**not**)	yes	yes	non-recursive	cons., subset	recursive

3.3.9 Head cycle free AnsProlog$^{\neg, or}$ programs

So far in this section we mostly discussed sub-classes of AnsProlog programs. We now define the subclass of head cycle free AnsProlog$^{\neg, or}$ programs which have some useful properties. To define that we first define the notion of a literal dependency graph. The *literal dependency graph* of an AnsProlog$^{\neg, or}$ program is a directed graph where each literal is a node and where there is an edge from L to L' iff there is a rule in which L appears positive (i.e, not preceded by **not**) in the body and L' appears in the head.

Definition 25 An AnsProlog$^{\neg, or}$ program is said to be *head cycle free* iff its literal dependency graph does not contain directed cycles that go through two literals that belong to the head of the same rule. □

Example 61 Consider the following program Π from Example 29.

p *or* $p' \leftarrow$.
q *or* $q' \leftarrow$.

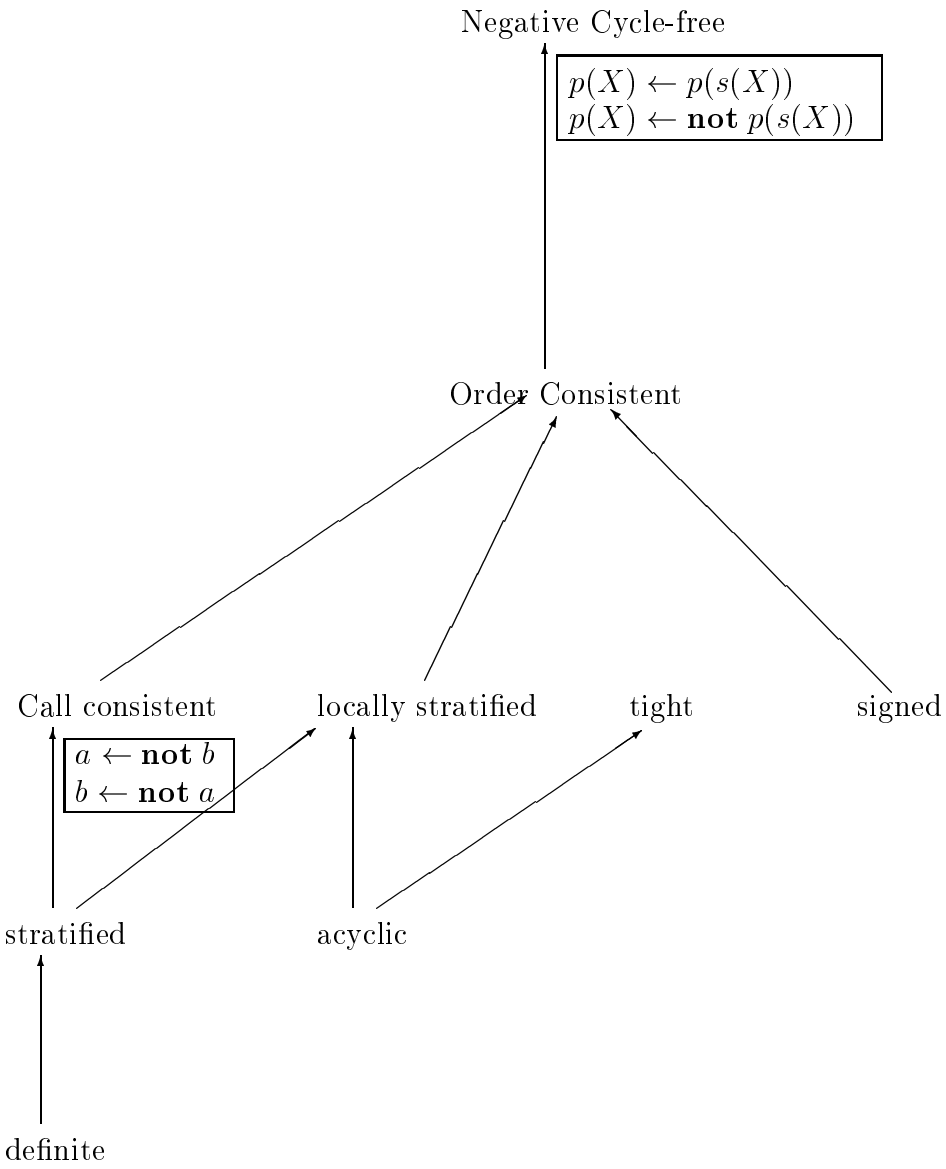

Figure 3.1: The ordering between AnsProlog subclasses

$not\_sat \leftarrow p, q.$
$not\_sat \leftarrow p', q'.$
$q \leftarrow not\_sat.$
$q' \leftarrow not\_sat.$

The above program is not head-cycle-free as its literal dependency graph has a cycle from *not_sat* to q to *not_sat*.

3.4 Restricted monotonicity and signed AnsProlog* programs

Intuitively, a theory is monotonic – normally meaning monotonically non-decreasing – if by adding new axioms to the theory we do not loose any of our original conclusions. In Chapter 1 we argued that suitable languages for knowledge representation should not have the monotonicity property, so that we can formulate bounded reasoning where reasoning is done with limited information and conclusions may be revised in presence of additional information. Although blanket monotonicity is not a desirable property, often we may want our theory to have a restricted kind of monotonicity. For example, we may not want a definition of a particular concept in our theory to change in presence of additional information. Thus we need a notion of 'restricted monotonicity' and need to study when AnsProlog* programs have such properties.

3.4.1 Restricted monotonicity

We start with a notion of restricted monotonicity for general declarative formalisms. We will later tailor the definition to particular classes of AnsProlog* programs.

Definition 26 A *declarative formalism* is defined by a set S of symbolic expressions called *sentences*, a set P of symbolic expressions called *postulates*, and a map Cn from sets of postulates to sets of sentences.

A set of postulates is referred to as a *theory*; and a sentence A is a consequence of a theory T if $A \in Cn(T)$. □

Definition 27 Let $\langle S, P, Cn \rangle$ be a declarative formalism; and let a subset S_0 of S be designated as the set of assertions (inputs), and a set P_0 of P as the set of parameters (outputs). A theory T is said to satisfy the restricted monotonicity condition with respect to S_0 and P_0 if, for any sets $p, q \subset P_0$,

$p \subset q \Rightarrow Cn(T \cup p) \cap S_0 \subset Cn(T \cup q) \cap S_0$ □

3.4.2 Signed AnsProlog$^{\neg, or}$ programs and their properties.

Earlier in Section 3.3.7 we had introduced the notion of signed AnsProlog programs. In this section we generalize the notion of signing to AnsProlog$^{\neg, or}$ programs and study properties of such programs, in particular the property of restricted monotonicity. We start with the generalized notion of signing.

Definition 28 Let Π be an AnsProlog$^{\neg, or}$ program, and S be a subset of Lit_Π, such that no literal in $S \cap Head(\Pi)$ appears complemented in $Head(\Pi)$. We say that S is a *signing* for Π if each rule $r \in \Pi$ satisfies the following two conditions:

- $head(r) \cup pos(r) \subseteq S$ and $neg(r) \subseteq \bar{S}$, or
  $head(r) \cup pos(r) \subseteq \bar{S}$ and $neg(r) \subseteq S$,
- if $head(r) \subseteq S$, then $head(r)$ is a singleton,

where $\bar{S} = Lit_\Pi \setminus S$. If a program has a signing then we say that it is *signed*. □

Example 62 Consider the following program Π_1.

$a \leftarrow$ **not** b
$b \leftarrow$ **not** a
$\neg a$

Program Π_1 has a signing $S = \{b\}$. Note that neither $\{a, \neg a\}$ nor $\{a\}$ is a signing. So the definition of signing for AnsProlog$^{\neg, or}$ programs in general is asymmetric. □

We now define an ordering between rules and programs which we will later use in defining $\leq_1$ to show the restricted monotonicity property.

Definition 29 Given rules r and r', we say that r *is subsumed by* r', and we write $r \preceq r'$, if the following three conditions hold:

1. $neg(r') \subseteq neg(r)$,
2. $pos(r') \subseteq pos(r)$, and
3. every literal in $head(r') \setminus head(r)$ appears complemented in $pos(r)$. □

Note that $\preceq$ is reflexive and transitive but not anti-symmetric. When dealing with AnsProlog$^\neg$ programs we replace the third condition in the above definition by $head(r) = head(r')$. In the following we define $h_S(\Pi)$, the reduction of a program Π with respect to a set S of literals.

Definition 30 Given programs Π and Π', we say Π is subsumed by Π', and we write $\Pi \preceq \Pi'$, if for each rule r in Π there is a rule r' in Π' such that $r \preceq r'$. □

Definition 31 Let Π be an AnsProlog$^{\neg, or}$ program. If S is a signing for Π, then

- $h_S(\Pi) = \{r \in \Pi \ : \ head(r) \subseteq S\}$,
- $h_{\bar{S}}(\Pi) = \{r \in \Pi \ : \ head(r) \subseteq \bar{S}\}$. □

For AnsProlog$^\neg$ programs Π with signing S, $h_S(\Pi)$ is alternatively denoted by Π_S and $h_{\bar{S}}(\Pi)$ is alternatively denoted by $\Pi_{\bar{S}}$. We now present two restricted monotonicity results about signed AnsProlog$^\neg$ programs.

Proposition 46 *[Tur93]* For AnsProlog programs P and Q with common signing S,
if $P_{\bar{S}} \preceq Q_{\bar{S}}$ and $Q_S \preceq P_S$ then P entails every ground atom in S that is entailed by Q, and Q entails every ground atom in $\bar{S}$ that is entailed by P. □

The above proposition implies the following two simple restricted monotonicity properties with S and P as set of ground atoms, and Cn as the consequence relation between AnsProlog programs and ground atoms that are entailed by it. (i) S_0, and P_0 are equal to $\bar{S}$. In this case the above proposition implies that if we start with a program P and add facts about $\bar{S}$ and obtain a new program Q, then Q still entails every ground atom in $\bar{S}$ that was entailed by P. (ii) S_0, and P_0 are equal to S. In this case the above proposition implies that if we start with a program Q and add facts about S and obtain a new program P, then P still entails every ground atom in S that was entailed by Q.

Proposition 47 *[Tur93]* For AnsProlog$^\neg$ programs P and Q with common signing S,
if $P_{\bar{S}} \preceq Q_{\bar{S}}$ and $Q_S \preceq P_S$ and $literals(P) \cap \bar{S} \subseteq literals(Q) \cap \bar{S}$, then Q entails every ground literal in $\bar{S}$ that is entailed by P. □

The above proposition implies the following simple restricted monotonicity property with S and P as set of ground literals, and Cn as the consequence relation between AnsProlog$^\neg$ programs and ground literals that are entailed – through $\models^*$ – by it. S_0, and P_0 are equal to $\bar{S}$. In this case the above proposition implies that if we start with a program P and add facts about $\bar{S}$ and obtain a new program Q, then Q still entails every ground literal in $\bar{S}$ that was entailed by P.

We now discuss an application of the above Proposition. Let us consider a simpler form of the Yale Turkey shoot problem from Section 2.2.2 where the only action we have is *shoot*. Then a formulation to reason about hypothetical situations in presence of incomplete information about the initial situation is given as follows:

r_1: $holds(alive, s_0) \leftarrow$

r_2 : $holds(F, res(A, S)) \leftarrow holds(F, S), \mathbf{not}\ \ ab(F, A, S)$
r_3 : $\neg holds(F, res(A, S)) \leftarrow \neg holds(F, S), \mathbf{not}\ \ ab(F, A, S)$

r_4 : $\neg holds(alive, res(shoot, S)) \leftarrow holds(loaded, S)$

r_5 : $ab(alive, shoot, S) \leftarrow \mathbf{not}\ \ \neg holds(loaded, S)$

As in Section 2.2.2, let us consider the case when we are given the additional oracle:

r_6 : $holds(alive, res(shoot, s_0)).$

Intuitively, we should now be able to conclude that the gun is not loaded in s_0. But $\Pi = \{r_1, r_2, r_3, r_4, r_5, r_6\}$ does not entail $\neg holds(loaded, s_0)$. In Section 2.2.2 we suggested using integrity constraints and enumeration with respect to the initial situation to be able to do such backward reasoning. Another alternative would be to add the following explicit rules for backward reasoning:

r_7: $holds(loaded, S) \leftarrow holds(alive, S), \neg holds(alive, res(shoot, S))$
r_8 : $\neg holds(loaded, S) \leftarrow holds(alive, res(shoot, S))$

r_9 : $\neg holds(F, S) \leftarrow \neg holds(F, res(A, S)), \mathbf{not}\ ab(F, A, S)$
r_{10} : $holds(F, S) \leftarrow holds(F, res(A, S)), \mathbf{not}\ ab(F, A, S)$

It can now be shown that $\Pi' = \Pi \cup \{r_7, r_8, r_9, r_{10}\}$ entails $\neg holds(loaded, s_0)$. Moreover, we can use Proposition 47 to show that Π' makes all the conclusions about *holds* and $\neg holds$ as made by Π and possibly more. More formally,

Proposition 48 The program Π' entails every *holds* and $\neg holds$ ground literals that is entailed by the program Π. □

Proof:
Let $S = \{a : a$ is an ab atom in our language $\}$. It is easy to see that S is a common signing for Π and Π'. It is also easy to see that $\Pi'_S = \Pi_S$ and $\Pi_{\bar{S}} \subseteq \Pi'_{\bar{S}}$. Hence, $\Pi_{\bar{S}} \preceq \Pi'_{\bar{S}}$ and $\Pi'_S \preceq \Pi_S$. Since $literals(\Pi) = literals(\Pi')$, it is clear that $literals(\Pi) \cap \bar{S} \subseteq literals(\Pi') \cap \bar{S}$.

Thus using Proposition 47 we have that Π' entails every ground literal in $\bar{S}$ that is entailed by the program Π. Since the *holds* and $\neg$*holds* ground literals are part of $\bar{S}$, our proposition holds. □

Now let us explore results about the answer sets of signed AnsProlog$^\neg$ programs. Recall that answer-sets of AnsProlog$^\neg$ programs are defined as sets of literals S, such that $S = \mathcal{M}^\neg(\Pi^S)$. Let us denote $\mathcal{M}^\neg(\Pi^S)$ as $\Gamma^\neg_\Pi(S)$.

Theorem 3.4.1 *[Tur94]* Let Π be an AnsProlog$^\neg$ program with signing S. Π is consistent iff $lfp(\Gamma^{\neg\,2}_\Pi) \cup (gfp(\Gamma^{\neg\,2}_\Pi) \cap S)$ is a consistent answer set for Π. □

Definition 32 We say that an AnsProlog$^{\neg,\,or}$ program Π is *head-consistent* if $Head(\Pi)$ is a consistent set. □

Proposition 49 *[Tur94]* Let Π be a head-consistent AnsProlog$^\neg$ program with signing S. The following three conditions hold.

1. Π is a consistent program.
2. $lfp(\Gamma^{\neg\,2}_\Pi) \cup (gfp(\Gamma^{\neg\,2}_\Pi) \cap S)$ and $lfp(\Gamma^{\neg\,2}_\Pi) \cup (gfp(\Gamma^{\neg\,2}_\Pi) \cap \bar{S})$ are consistent answer sets of Π.
3. $Cn(\Pi) = lfp(\Gamma^{\neg\,2}_\Pi)$. □

We will now define the notion of a cover of an AnsProlog$^{\neg,\,or}$ program and present a property of programs that have at least one head-consistent cover.

Definition 33 Let Π be an AnsProlog$^{\neg,\,or}$ program. An AnsProlog$^\neg$ program Π' is a cover of Π if Π' can be obtained from Π by replacing each rule $r \in \Pi$ with a rule r' such that $head(r')$ is a singleton, $head(r') \subseteq head(r)$, $pos(r') = pos(r)$, and $neg(r') = neg(r)$. □

Proposition 50 *[Tur94]* Every signed AnsProlog$^{\neg,\,or}$ program with at least one head-consistent cover is consistent. □

We now present two properties of AnsProlog$^{\neg,\,or}$ program which imply several specific restricted monotonicity properties.

Theorem 3.4.2 *[Tur94]* Let Π and Π' be AnsProlog$^{\neg,\,or}$ programs in the same language, both with signing S. If $h_{\bar{S}}(\Pi) \preceq h_{\bar{S}}(\Pi')$ and $h_S(\Pi') \preceq h_S(\Pi)$, then $Cn(\Pi) \cap \bar{S} \subseteq Cn(\Pi') \cap \bar{S}$. □

The above proposition implies the following simple restricted monotonicity property with S and P as set of ground literals, and Cn as the consequence relation between AnsProlog$^{\neg,\,or}$ programs and ground literals that are entailed – through $\models^*$ – by it. S_0, and P_0 are equal to $\bar{S}$. In this case the above proposition implies that if we start with a program Π and add facts about $\bar{S}$ and obtain a new program Π', then Π' still entails every ground literal in $\bar{S}$ that was entailed by Π. We now give a stronger result than Theorem 3.4.2 stated in terms of the answer sets of a program instead of consequences of a program.

Theorem 3.4.3 *[Tur94]* Let Π and Π' be AnsProlog$^{\neg,\,or}$ programs in the same language, both with signing S. If $h_{\bar{S}}(\Pi) \preceq h_{\bar{S}}(\Pi')$ and $h_S(\Pi') \preceq h_S(\Pi)$, then for every consistent answer set A' for program Π', there is a consistent answer set A for Π such that $A \cap \bar{S} \subseteq A' \cap \bar{S}$. □

3.5 Analyzing AnsProlog* programs using 'splitting'

To be able to analyze large AnsProlog* programs we need a way to break down an AnsProlog* program to smaller components in such a way that the analysis of the components can be carried over to the whole program. In this section we introduce such a notion which is called 'splitting', and use it to not only analyze AnsProlog* programs but also to construct answer sets of the whole program by computing answer sets of the smaller components. This provides us with an alternative way to compute answer sets for many programs beyond the general tedious guess-and-test approach that follow from the definition of answer sets, when we don't have a AnsProlog$^{-\mathbf{not}}$ program or a stratified AnsProlog program for which we have a constructive iterative fixpoint approach to compute answer sets.

The idea of splitting a program into sequences is a generalization of the idea of local stratification. Similar to local stratification, the ground program and the literals in the language are divided into strata such that the literals in the body of a rule in any stratum either belong to that stratum or a lower stratum, and the literals in the head of a rule belong to that stratum. But unlike local stratification there is no prohibition of recursion through negation. Thus while the program

$p \leftarrow a.$
$p \leftarrow b.$
$a \leftarrow \mathbf{not}\ b.$
$b \leftarrow \mathbf{not}\ a.$

is not locally stratified, it can be split into two strata, the top stratum consisting of the first two rules and the bottom stratum consisting of the other two rules. Unlike local stratification, splitting does not guarantee us a unique answer set, but it will be useful in computing the answer sets of the program by computing the answer sets layer by layer. For example, for the above program, we can compute the answer sets of the bottom layer, which are $\{a\}$ and $\{b\}$, first and then use each of those answer sets to partially evaluate the top layer and compute the answer set of the partially evaluated rules. Besides helping us in computing answer sets, we can use the notion of splitting to generalize many of the subclasses of AnsProlog programs in Section 3.3, by requiring that each stratum after partial evaluation with respect to an answer set of the subprogram consisting of all strata below that, belong to that subclass. In the rest of this section we will formalize and illustrate the above notions.

3.5.1 Splitting sets

We start with the notion of splitting sets that is used to split a program into two layers. Later we will generalize this to splitting a program into a sequence.

Definition 34 *(Splitting set)* A *splitting set* for an AnsProlog$^{\neg, or}$ program Π is any set U of literals such that, for every rule $r \in \Pi$, if $head(r) \cap U \neq \emptyset$ then $lit(r) \subset U$. If U is a splitting set for Π, we also say that U splits Π. The set of rules $r \in \Pi$ such that $lit(r) \subset U$ is called the *bottom* of Π relative to the splitting set U and denoted by $bot_U(\Pi)$. The subprogram $\Pi \setminus bot_U(\Pi)$ is called *the top of* Π relative to U and denoted $top_U(\Pi)$. □

Consider the following program Π_1:

$a \leftarrow b, \mathbf{not}\ c.$
$b \leftarrow c, \mathbf{not}\ a.$
$c \leftarrow.$

The set $U = \{c\}$ splits Π_1 such that the last rule constitutes $bot_U(\Pi_1)$ and the first two rules form $top_U(\Pi_1)$.

Once a program is split into top and bottom with respect to a splitting set, we can compute the answer sets of the bottom part and for each of these answer sets, we can further simplify the top part by partial evaluation before analyzing it further. We now present the formal definition of partial evaluation and then define how to compute the answers set of the original program using the partial evaluation of the top part with respect to answer sets of the bottom part.

Definition 35 *(Partial evaluation)* The partial evaluation of a program Π with splitting set U w.r.t. a set of literals X is the program $eval_U(\Pi, X)$ defined as follows.

$eval_U(\Pi, X) = \{r' \mid$

- there exist a rule r in Π such that $(pos(r) \cap U) \subset X$ and $(neg(r) \cap U) \cap X = \emptyset$, and
- $head(r') = head(r),\ pos(r') = pos(r) \setminus U,\ neg(r') = neg(r) \setminus U.\ \}$ □

For the program Π_1 mentioned above $eval_{\{c\}}(top_{\{c\}}(\Pi_1), \{c\}) = \{b \leftarrow \textbf{not}\ a\}$.

Definition 36 *(Solution)* Let U be a splitting set for a program Π. A solution to Π w.r.t. U is a pair $\langle X, Y \rangle$ of literals such that:

- X is an answer set for $bot_U(\Pi)$;
- Y is an answer set for $eval_U(top_U(\Pi), X)$;
- $X \cup Y$ is consistent. □

Continuing with Π_1 and $U = \{c\}$, the only answer set of $bot_U(\Pi_1)$ is $\{c\}$. Now the only answer set of $eval_U(top_U(\Pi_1), \{c\})$ is $\{b\}$. Hence, $\langle \{c\}, \{b\} \rangle$ is the only solution to Π_1 w.r.t. $\{c\}$.

Theorem 3.5.1 *(Splitting theorem) [LT94]* Let U be a splitting set for a program Π. A set A of literals is a consistent answer set for Π if and only if $A = X \cup Y$ for some solution $\langle X, Y \rangle$ to Π w.r.t. U. □

Continuing with Π_1 and $U = \{c\}$, we have that $\{c, b\}$ is the only answer set of Π_1.

Example 63 Consider the following program Π_2:

$\neg b \leftarrow$
$a\ or\ b \leftarrow$

Let $U = \{a, b\}$. We have $bot_U(\Pi_2) = \{a\ or\ b \leftarrow\}$ and $top_U(\Pi_2) = \{\neg b \leftarrow\}$. The two answer sets of $bot_U(\Pi_2)$ are $A_1 = \{a\}$ and $A_2 = \{b\}$. Now the answer set of $eval_U(top_U(\Pi_2), A_1)$ is $\{\neg b\}$, and the answer set of $eval_U(top_U(\Pi_2), A_2)$ is $\{\neg b\}$. Since $\{b, \neg b\}$ is inconsistent, the only solution to Π_2 with respect to U is $\langle \{a\}, \{\neg b\} \rangle$. □

Exercise 5 Consider the following program:

$p \leftarrow \textbf{not}\ q.$
$p \leftarrow \textbf{not}\ p.$
$q \leftarrow \textbf{not}\ r.$
$r \leftarrow \textbf{not}\ q.$

Using $\{q, r\}$ as a splitting set show that the only answer set of the above program is $\{r, p\}$. □

Exercise 6 Consider the following program:

$c \leftarrow$ **not** b.
$a \leftarrow b$.
a *or* $b \leftarrow$.

Compute the answer sets of the program by using $\{a, b\}$ as a splitting set. Explain why $\{a, b\}$ is not an answer set of this program. □

3.5.2 Application of splitting

In this subsection we illustrate a couple of applications of the notion of splitting to results about conservative extension and about adding CWA rules to a program.

Conservative Extension

We first show how the notion of splitting can be used to prove one of the conservative extension proposition (Proposition 24). First let us recall the statement of Proposition 24.

Let Π be an AnsProlog$^{\neg, or}$ program, and let C be a consistent set of literals that do not occur in Π and whose complements also do not occur in Π. Let Π' be an AnsProlog$^{\neg}$ program such that for every rule $r \in \Pi'$, $head(r) \subseteq C$, and $neg(r) \subseteq lit(\Pi)$. For any literal $L \notin C$, L is a consequence of $\Pi \cup \Pi'$ iff L is a consequence of Π. □

Proof of Proposition 24:

Consider the program $\Pi \cup \Pi'$. It is easy to see that $U = lit(\Pi)$ splits $\Pi \cup \Pi'$ with Π as $bot_U(\Pi \cup \Pi')$ and and Π' as $top_U(\Pi \cup \Pi')$. Let A be any consistent answer set of Π. The program $eval_U(\Pi', A)$ is an AnsProlog$^{\neg,-}$**not** program. That and since the head of the rules of $eval_U(\Pi', A)$ is from C and since C is consistent, $eval_U(\Pi', A)$ has a unique consistent answer set B which is a subset of C. Since neither the literals in C nor its complement appear in Π, the answer set A of Π has neither any literal from C nor any literal from the complement of C. Hence, $A \cup B$ is consistent, and $\langle A, B\rangle$ is a solution to $\Pi \cup \Pi'$ with respect to U. Thus, for every consistent answer set A of Π, there exists a $B \subseteq C$ such that $\langle A, B\rangle$ is a solution to $\Pi \cup \Pi'$ with respect to U. Now if $\langle A, B\rangle$ is a solution to $\Pi \cup \Pi'$ with respect to U then B must be a subset of C. It now follows from the splitting set theorem that a literal $L \notin C$ is a consequence of $\Pi \cup \Pi'$ iff it is a consequence of Π. □

Adding CWA rules

We now state a proposition about the relation between the answer sets of a program with the answer sets of a second program that is obtained by adding CWA rules to the first program.

Proposition 51 *(Adding CWA rules)* Let Π be an AnsProlog$^{\neg, or}$ program, and let C be a consistent set of literals that do not occur in Π and let Π' be the program given by $\Pi \cup \{L \leftarrow$ **not** $\bar{L} \mid L \in C\}$. If X is a consistent answer set for Π, then

$$X \cup \{L \in C \mid \bar{L} \notin X\} \tag{3.5.1}$$

is a consistent answer set for Π'. Moreover, every consistent answer set for Π' can be represented in the form (3.5.1) for some consistent answer set X for Π. □

Proof:

Let $U = lit(\Pi)$. Since by definition $U \cap C = \emptyset$, U splits Π' with Π as its bottom. Let A be a consistent answer set of Π. The program $eval_U(\Pi' \setminus \Pi, A)$ is the set $\{L \leftarrow . \ : \ \bar{L} \notin A\}$. The only answer set of $eval_U(\Pi' \setminus \Pi, A)$ is then $\{L \ : \ \bar{L} \notin A\}$. Since A and C are both consistent and $lit(\Pi) \cap C = \emptyset$, $A \cup C$ is consistent. Thus in general for any X, $\langle X, \{L \ : \ \bar{L} \notin X\}\rangle$ is a solution to Π' with respect to U if X is an answer set of Π. The proof then follows from the splitting set theorem. □

Example 64 Let Π_4 be the program

$p(1) \leftarrow$
$\neg q(2) \leftarrow$

Let $C = \{\neg p(1), \neg p(2), q(1), q(2)\}$. We can now obtain Π'_4 as the program $\Pi_4 \cup \{\neg p(1) \leftarrow$ **not** $p(1); \neg p(2) \leftarrow$ **not** $p(2); q(1) \leftarrow$ **not** $\neg q(1); q(2) \leftarrow$ **not** $\neg q(2)\}$.

Using Proposition 51 we can easily compute the answer set of Π' as $\{p(1), \neg q(2), \neg p(2), q(1)\}$. □

3.5.3 Splitting sequences

We now generalize the notion of splitting a program into top and bottom, to splitting it into a sequence of smaller programs.

Definition 37 A *splitting sequence* for an AnsProlog$^{\neg, or}$ program Π is a monotone, continuous sequence[3] $\langle U_\alpha\rangle_{\alpha<\mu}$ of splitting sets for Π such that $\bigcup_{\alpha<\mu} U_\mu = lit(\Pi)$. □

Example 65 Consider the following program Π_4 that defines even numbers.

$e(0) \leftarrow$
$e(s(X)) \leftarrow$ **not** $e(X)$.

The following sequence of length ω is a splitting sequence for Π_4:

$$\langle \{e(0)\}, \{e(0), e(s(0))\}, \{e(0), e(s(0)), e(s(s(0)))\}, \ldots\rangle \tag{3.5.2}$$

□

As before, we define solutions to a program Π with respect to U, and then relate it to the answer sets of Π.

Definition 38 Let $U = \langle U_\alpha\rangle_{\alpha<\mu}$ be a splitting sequence for an AnsProlog$^{\neg, or}$ program Π. A *solution* to Π with respect to U is a sequence $\langle X_\alpha\rangle_{\alpha<\mu}$ of sets of literals such that:

- X_0 is an answer set for $bot_{U_0}(\Pi)$,
- for any ordinal α such that $\alpha + 1 < \mu$, $X_{\alpha+1}$ is an answer set of the program:

$$eval_{U_\alpha}(bot_{U_{\alpha+1}}(\Pi) \setminus bot_{U_\alpha}(\Pi), \bigcup_{\nu\leq\alpha} X_\nu),$$

- for any limit ordinal $\alpha < \mu$, $X_\alpha = \emptyset$, and

[3] See Appendix A for the definition.

- $\bigcup_{\alpha \leq \mu} X_\alpha$ is consistent. □

The only solution $\langle X_0, X_1, \ldots \rangle$ to Π_4 w.r.t. the splitting sequence (3.5.2) can be given by the following:

$$X_n = \begin{cases} \{p(S^n(0))\}, & \text{if } n \text{ is even} \\ \emptyset, & \text{otherwise.} \end{cases}$$

Theorem 3.5.2 *(Splitting sequence theorem)* Let $U = \langle U_\alpha \rangle_{\alpha < \mu}$ be a splitting sequence for a AnsProlog$^{\neg, or}$ program Π. A set A of literals is a consistent answer set for Π iff $A = \bigcup_{\alpha < \mu} X_\alpha$ for some solution $\langle X_\alpha \rangle_{\alpha < \mu}$ to Π with respect to U. □

Example 66 Consider the following program Π:

r_1 : $e \leftarrow c, b.$
r_2 : $f \leftarrow d.$
r_3 : $c \leftarrow$ **not** $d.$
r_4 : $d \leftarrow$ **not** $c,$ **not** $b.$
r_5 : $a \leftarrow$ **not** $b.$
r_6 : $b \leftarrow$ **not** $a.$

This program has a splitting sequence $U_0 = \{a, b\}$, $U_1 = \{a, b, c, d\}$, and $U_2 = \{a, b, c, d, e, f\}$. It is easy to see that $bot_{U_0}(\Pi) = \{r_5, r_6\}$, $bot_{U_1}(\Pi) = \{r_3, r_4, r_5, r_6\}$, and $bot_{U_2}(\Pi) = \{r_1, r_2, r_3, r_4, r_5, r_6\}$.

The answer sets of $bot_{U_0}(\Pi)$ are $X_{0.0} = \{a\}$ and $X_{0.1} = \{b\}$.

$eval_{U_0}(bot_{U_1}(\Pi) \setminus bot_{U_0}(\Pi), X_{0.0}) = eval_{U_0}(\{r_3, r_4\}, \{a\}) = \{c \leftarrow$ **not** $d., d \leftarrow$ **not** $c.\}$.
It has two answer sets $X_{1.0} = \{c\}$ and $X_{1.1} = \{d\}$.

$eval_{U_0}(bot_{U_1}(\Pi) \setminus bot_{U_0}(\Pi), X_{0.1}) = eval_{U_0}(\{r_3, r_4\}, \{b\}) = \{c \leftarrow$ **not** $d.\}$.
It has one answer set $X_{1.2} = \{c\}$.

$eval_{U_1}(bot_{U_2}(\Pi) \setminus bot_{U_1}(\Pi), X_{0.0} \cup X_{1.0}) = eval_{U_1}(\{r_1, r_2\}, \{a, c\}) = \{\}$.
It has one answer set $X_{2.0} = \{\}$.

$eval_{U_1}(bot_{U_2}(\Pi) \setminus bot_{U_1}(\Pi), X_{0.0} \cup X_{1.1}) = eval_{U_1}(\{r_1, r_2\}, \{a, d\}) = \{f \leftarrow .\}$.
It has one answer set $X_{2.1} = \{f\}$.

$eval_{U_1}(bot_{U_2}(\Pi) \setminus bot_{U_1}(\Pi), X_{0.1} \cup X_{1.2}) = eval_{U_1}(\{r_1, r_2\}, \{b, c\}) = \{e \leftarrow .\}$.
It has one answer set $X_{2.2} = \{e\}$.

From the above analysis, Π has three solutions $\langle X_{0.0}, X_{1.0}, X_{2.0} \rangle$, $\langle X_{0.0}, X_{1.1}, X_{2.1} \rangle$, and $\langle X_{0.1}, X_{1.2}, X_{2.2} \rangle$. Thus, using Theorem 3.5.2 Π has three answer sets: $\{a, c\}$, $\{a, d, f\}$, and $\{b, c, e\}$. □

3.5.4 Applications of the Splitting sequence theorem

In this subsection we show that the notion of order consistency can be defined in terms of splitting the program to components which are signed. This leads to a different proof of coherence of order-consistent AnsProlog programs by using the proof of coherence of signed AnsProlog programs and the properties of a splitting sequence. We now formally define the notion of components.

Given a program Π and a set of literals X, $rem(\Pi, X)$ is the set of rules obtained by taking each of the rules in Π and removing from its body any literal in X, regardless of whether it is preceded by **not** or not. For any program Π and any splitting sequence $U = \langle U_\alpha \rangle_{\alpha < \mu}$ for P, the programs:

$bot_{U_0}(\Pi)$,

$rem(bot_{U_{\alpha+1}}(\Pi) \setminus bot_{U_\alpha}(\Pi), U_\alpha)$, for all $\alpha + 1 < \mu$

are called the U-components of Π.

For example, the U-components of Π_4 are the programs $\{p(S^n(0)) \leftarrow\}$ for all n.

Exercise 7 Using the notion of components and splitting sequence prove that every stratified AnsProlog program has a unique answer set. □

Earlier the notion of order-consistency was only defined for AnsProlog programs. Here we extend that definition to AnsProlog$^{\neg, or}$ programs and relate order-consistent AnsProlog$^{\neg}$ programs with signed AnsProlog$^{\neg}$ programs. We have the following notations first:

For any atom A, let P_A^+ and P_A^- be the smallest sets that satisfy the following conditions:

- $A \in P_A^+$,
- for every rule r: if $head(r) \subseteq P_A^+$ then $pos(r) \subseteq P_A^+$ and $neg(r) \subseteq P_A^-$, and
- for every rule r: if $head(r) \subseteq P_A^-$ then $pos(r) \subseteq P_A^-$ and $neg(r) \subseteq P_A^+$.

An AnsProlog$^{\neg, or}$ program P is called *order-consistent* if there exists a level mapping f such that $f(B) < f(A)$ whenever $B \in P_A^+ \cap P_A^-$.

Proposition 52 *[LT94]* An AnsProlog$^{\neg}$ program Π is order-consistent iff it has a splitting sequence U such that all U-components of Π are signed. □

Exercise 8 Generalize the notion of splitting sequences to splitting into a partial order of components. Formulate and prove the theorem about computing the answer set in this case.

Discuss the advantage of this notion over the notion of splitting sequences. □

3.6 Language independence and language tolerance

Recall from Section 1.2 that the answer-set language given by an alphabet is uniquely determined by its constants, function symbols, and predicate symbols. The consequences of an AnsProlog* program not only depend on the rules of the program, but because of presence of variables in those rules, may also depend on the language. For example, consider the program Π consisting of the following two rules:

$p(a) \leftarrow$ **not** $q(X)$.
$q(a) \leftarrow$.

Consider two different languages $\mathcal{L}_1$ and $\mathcal{L}_2$ of the above program, where both $\mathcal{L}_1$ and $\mathcal{L}_2$ have p and q as the predicate symbols, both have no function symbols, and $\mathcal{L}_1$ has the constant a, while $\mathcal{L}_2$ has the constants a and b. We can now give the grounding of Π with respect to $\mathcal{L}_1$ and $\mathcal{L}_2$ as follows:

$$ground(\Pi, \mathcal{L}_1) \ = \ \left\{ \begin{array}{l} p(a) \leftarrow \textbf{not } q(a). \\ q(a) \leftarrow . \end{array} \right.$$

$$ground(\Pi, \mathcal{L}_2) \;=\; \left\{ \begin{array}{l} p(a) \leftarrow \textbf{not}\ q(a). \\ p(a) \leftarrow \textbf{not}\ q(b). \\ q(a) \leftarrow . \end{array} \right.$$

It is easy to see that $p(a)$ is not a consequence of $ground(\Pi, \mathcal{L}_1)$, while it is a consequence of $ground(\Pi, \mathcal{L}_2)$. This illustrates that the consequences of an AnsProlog* program not only depend on the rules of the program, but may also depend on the language.

This raises the following question. Can we identify sufficiency conditions when the consequences of an AnsProlog* program only depends on the rules of the program, and not on the language? We will call such programs as *language independent* .

We are also interested in a weaker notion which we will call *language tolerant*. Intuitively, in language tolerant programs we allow the possibility that the conclusions may be different when the languages are different, but require the condition that they have the same conclusions about literals that are in the common language. For example, consider the program Π' consisting of the only rule:

$p(X) \leftarrow$.

It is clear that the above program is not language independent as the conclusions of this program with respect to two different languages $\mathcal{L}_3$ and $\mathcal{L}_4$, where the constants in $\mathcal{L}_3$ are a and c and in $\mathcal{L}_4$ are a and b, are different. But both make the same conclusion about the atom $p(a)$ which is in both languages. In fact, we can generalize and show that the above program is language tolerant.

One of the main reasons we would like to develop the notions of language tolerance and language independence is the fact that often in AnsProlog programs we envision the terms in the different position of an atom to be of particular types. For example, in Situation calculus, in the atom $holds(F, S)$, we expect F to be a fluent while S to be a situation. Now consider an AnsProlog* program formulating certain aspect of situation calculus. If we do the grounding of this program using the function $ground(\Pi, \mathcal{L})$, then F and S in $holds(F, S)$ can be grounded with any of the ground terms in$\mathcal{L}$. What we would like though is to ground F with fluent terms and S with situation terms. To achieve that we need to expand our notion of language by allowing sorts.

This leads to the question that under what restrictions (on the program) the conclusions of an AnsProlog* program made in presence of a many sorted language is same as the conclusion made in the absence of multiple sorts. The notions of language tolerance and language independence comes in handy in answering this question. The usefulness of the above is that many query evaluation procedures do not take sorts into account, while programmers often write programs with sorts in mind. Before proceeding further we need a definition of a language with sorts.

3.6.1 Adding sorts to answer-set theories

To specify the language $\mathcal{L}$ of a *sorted answer-set theory*, in addition to variables, connective, punctuation symbols, and a signature $\sigma_{\mathcal{L}}$, we have a non-empty set $I_{\mathcal{L}}$, whose members are called *sorts*, and a *sort specification* for each symbol of $\sigma_{\mathcal{L}}$ and the variables. When the language is clear from the context, we may drop the subscript writing σ and I.

The sort specification assigns each variable, and constant to a sort in I. Each n-ary function symbol is assigned an $n+1$-tuple $\langle s_1, \ldots, s_n, s_{n+1} \rangle$, where for each i, $1 \leq i \leq n+1, s_i \in I$. Each n-ary predicate symbol is assigned an n-tuple $\langle s_1, \ldots, s_n \rangle$, where for each i, $1 \leq i \leq n, s_i \in I$. In addition

it is stipulated that there must be at least one constant symbol of each sort in I. The terms, atoms, and literals are defined as before except that must respect the sort specifications. In the rest of this subsection the atoms and literals in an AnsProlog* program are either from a one-sorted language (i.e., multiple sorts) or a many-sorted language.

Example 67 Consider the following program Π in a many-sorted language $\mathcal{L}$.

$p(X,Y) \leftarrow r(X), \textbf{not } q(X,Y).$
$r(a) \leftarrow .$
$q(a,0) \leftarrow .$

The language $\mathcal{L}$ has variables X, and Y and has the signature $\sigma_{\mathcal{L}} = (\{a,0,1,2\},\{\},\{p/2,r/1,q/2\})$. The set of sorts $I_{\mathcal{L}}$ is $\{letter, number\}$. The sort specifications are as follows:

- $sort(X) = letter$; $sort(Y) = number$;
- $sort(a) = letter$; $sort(0) = sort(1) = sort(2) = number$;
- $sort(p) = sort(q) = \langle letter, number\rangle$; $sort(r) = \langle letter\rangle$.

The ground program $ground(\Pi, \mathcal{L})$ is

$p(a,0) \leftarrow r(a), \textbf{not } q(a,0).$
$p(a,1) \leftarrow r(a), \textbf{not } q(a,1).$
$p(a,2) \leftarrow r(a), \textbf{not } q(a,2).$
$r(a) \leftarrow .$
$q(a,0) \leftarrow .$

The above program has the unique answer set $\{r(a), q(a,0), p(a,1), p(a,2)\}$. □

We now define a notion of when a language is permissible for a program.

Definition 39 Let $\mathcal{L}$ be an arbitrary (one-sorted or many-sorted) language, and let Π be an AnsProlog* program. If every rule in Π is a rule in $\mathcal{L}$, we say that $\mathcal{L}$ is *permissible* for Π. □

3.6.2 Language Independence

Definition 40 An AnsProlog* program Π is *language independent* if, for two languages $\mathcal{L}_1$ and $\mathcal{L}_2$ that are permissible for Π, the ground programs $ground(\Pi, \mathcal{L}_1)$ and $ground(\Pi, \mathcal{L}_2)$ have the same consistent answer sets. □

Proposition 53 *[MT94b]* Let Π be a language independent AnsProlog$^{\neg, or}$ program, and let $\mathcal{L}_1$ and $\mathcal{L}_2$ be permissible languages for Π. Then $Cn(ground(\Pi, \mathcal{L}_1)) = Cn(ground(\Pi, \mathcal{L}_2))$. □

Ground programs are trivially language independent. Following is a result about an additional class of language independent programs whose grounding depends on the language, and hence does not lead to the same grounding regardless of the language.

Theorem 3.6.1 *[MT94b]* Every range-restricted AnsProlog$^{\neg, or}$ program is language independent. □

Consider the program $\{p(a) \leftarrow \textbf{not } q(X).; q(a) \leftarrow .\}$ in the beginning of this subsection. It is not range-restricted as the variable in the first rule does not appear in a positive literal in the body. In the beginning of the subsection we argued why this program is not language independent. Now consider the following range-restricted program Π:

$p(X) \leftarrow r(X), \textbf{not } q(X).$
$r(a) \leftarrow .$

Its grounding with respect to $\mathcal{L}_1$ and $\mathcal{L}_2$ is as follows:

$$ground(\Pi, \mathcal{L}_1) = \begin{cases} p(a) \leftarrow r(a), \textbf{not } q(a). \\ r(a) \leftarrow . \end{cases}$$

$$ground(\Pi, \mathcal{L}_2) = \begin{cases} p(a) \leftarrow r(a), \textbf{not } q(a). \\ p(a) \leftarrow r(b), \textbf{not } q(b). \\ r(a) \leftarrow . \end{cases}$$

Both programs have the unique answer set $\{r(a), p(a)\}$.

3.6.3 Language Tolerance

Definition 41 An AnsProlog* program Π is *language tolerant* if, for any two languages $\mathcal{L}_1, \mathcal{L}_2$ that are permissible for Π the following holds: If A_1 is a consistent answer set for the ground program $ground(\Pi, \mathcal{L}_1)$, then there is a consistent answer set A_2 for the ground program $ground(\Pi, \mathcal{L}_2)$ such that $A_1 \cap Lit(\mathcal{L}_2) = A_2 \cap Lit(\mathcal{L}_1)$. □

Proposition 54 *[MT94b]* Let Π be a language tolerant AnsProlog$^{\neg, or}$ program, and let $\mathcal{L}_1$ and $\mathcal{L}_2$ be permissible languages for Π. Then $Cn(ground(\Pi, \mathcal{L}_1)) \cap Lit(\mathcal{L}_2) = Cn(ground(\Pi, \mathcal{L}_2)) \cap Lit(\mathcal{L}_1)$. □

Example 68 Let us reconsider the program Π from Example 67. Let us have a different language $\mathcal{L}'$ that differs from $\mathcal{L}$ by having an additional constant b of sort *letter* and replacing the constant 1 by 4.
The ground program $ground(\Pi, \mathcal{L}')$ is

$p(a, 0) \leftarrow r(a), \textbf{not } q(a, 0).$
$p(a, 4) \leftarrow r(a), \textbf{not } q(a, 4).$
$p(a, 2) \leftarrow r(a), \textbf{not } q(a, 2).$
$p(b, 0) \leftarrow r(b), \textbf{not } q(b, 0).$
$p(b, 4) \leftarrow r(b), \textbf{not } q(b, 4).$
$p(b, 2) \leftarrow r(b), \textbf{not } q(b, 2).$
$r(a) \leftarrow .$
$q(a, 0) \leftarrow .$

The above program has the unique answer set $S' = \{r(a), q(a, 0), p(a, 4), p(a, 2)\}$ which although different from the answer set S of $ground(\Pi, \mathcal{L}')$, we have that $S \cap Lit(\mathcal{L}) = S' \cap Lit(\mathcal{L}') = \{r(a), q(a, 0), p(a, 2)\}$. Thus Π is not language independent. Our goal now is to show that it is language tolerant. □

Our next goal is to identify subclasses of AnsProlog* programs which are language tolerant. We first give a semantic condition. For that we need the following definition.

Definition 42 An AnsProlog* program Π' is a *part* of a program Π if Π' can be obtained from Π by (i) selecting a subset of the rules in Π, and (ii) deleting zero or more subgoals from each selected rule. □

Theorem 3.6.2 *[MT94b]* If an AnsProlog$^{\neg, or}$ program Π is stable and, for every permissible language $\mathcal{L}$ for Π, every part of $ground(\Pi, \mathcal{L})$ has a consistent answer set, then Π is language tolerant. □

The second condition of the above theorem needs exhaustive computation of answer sets of every part of Π. We now present a sufficiency condition that can be checked more easily.

Proposition 55 *[MT94b]* If Π is a predicate-order-consistent AnsProlog program, then for every permissible language $\mathcal{L}$ for Π, every part of $ground(\Pi, \mathcal{L})$ has a consistent answer set. □

The above sufficiency condition leads to the following theorem for language tolerance.

Theorem 3.6.3 *[MT94b]* If an AnsProlog program Π is stable ad predicate-order-consistent, then it is language tolerant. □

Example 69 The program Π from Example 67 is stable with respect to the mode $\Sigma_p = (-, +)$, $\Sigma_q = (-, -)$ and $\Sigma_r = (-)$. It is also predicate-order-consistent. Hence, it is language tolerant. □

Exercise 9 Show that the program $\{p(X) \leftarrow .\}$ is language tolerant. □

3.6.4 When sorts can be ignored

One of the most important application of the notion of language tolerance is that it leads us to conditions under which we can judiciously ignore the sorts.

Definition 43 let $\mathcal{L}$ and $\mathcal{L}'$ be languages. We say that $\mathcal{L}'$ is obtained from $\mathcal{L}$ by ignoring sorts if $\sigma_{\mathcal{L}} = \sigma_{\mathcal{L}'}$, $\mathcal{L}$ and $\mathcal{L}'$ have the same variables and $\mathcal{L}'$ does not have multiple sorts. □

Proposition 56 *[MT94b]* Let Π be a language tolerant AnsProlog$^{\neg, or}$ program in a language (possible with multiple sorts) $\mathcal{L}_\Pi$. If $\mathcal{L}$ is obtained from $\mathcal{L}_\Pi$ by ignoring sorts, then $ground(\Pi, \mathcal{L})$ is a conservative extension of $ground(\Pi, \mathcal{L}_\Pi)$. □

Example 70 Consider the program Π fro Example 67. Let us now consider the language $\mathcal{L}''$ obtained from $\mathcal{L}$ by having the same variables and signature but without any sorts. That means while grounding Π with respect to $\mathcal{L}''$, X and Y can take any value from $\{a, 0, 1, 2\}$. Thus $ground(\Pi, \mathcal{L}'')$ is the program:

$p(a, 0) \leftarrow r(a), \textbf{not } q(a, 0).$
$p(a, 1) \leftarrow r(a), \textbf{not } q(a, 1).$
$p(a, 2) \leftarrow r(a), \textbf{not } q(a, 2).$
$p(a, a) \leftarrow r(a), \textbf{not } q(a, a).$

$p(0, 0) \leftarrow r(0), \textbf{not } q(0, 0).$
$p(0, 1) \leftarrow r(0), \textbf{not } q(0, 1).$
$p(0, 2) \leftarrow r(0), \textbf{not } q(0, 2).$
$p(0, a) \leftarrow r(0), \textbf{not } q(0, a).$

$p(1,0) \leftarrow r(1), \textbf{not } q(1,0).$
$p(1,1) \leftarrow r(1), \textbf{not } q(1,1).$
$p(1,2) \leftarrow r(1), \textbf{not } q(1,2).$
$p(1,a) \leftarrow r(1), \textbf{not } q(1,a).$

$p(2,0) \leftarrow r(2), \textbf{not } q(2,0).$
$p(2,1) \leftarrow r(2), \textbf{not } q(2,1).$
$p(2,2) \leftarrow r(2), \textbf{not } q(2,2).$
$p(2,a) \leftarrow r(2), \textbf{not } q(2,a).$

$r(a) \leftarrow .$
$q(a,0) \leftarrow .$

The above program has the answer set $S'' = \{r(a), q(a,0), p(a,1), p(a,2), p(a,a)\}$. The answer set S'' differs from the answer set S of $ground(\Pi, \mathcal{L})$ by having the extra atom $p(a,a)$. But since $p(a,a) \notin Lit(\mathcal{L})$, $ground(\Pi, \mathcal{L}'')$ and $ground(\Pi, \mathcal{L})$ agree on $Lit(\mathcal{L})$. Thus $ground(\Pi, \mathcal{L}'')$ is a conservative extension of $ground(\Pi, \mathcal{L})$.

Intuitively the above means that by ignoring the sort we preserves the conclusions that would have been made if we considered sorts, and if we make any new conclusions then those literals are not part of the sorted language – i.e., they violate the sort conditions. □

3.7 Interpolating an AnsProlog program

So far in this chapter we have analyzed stand-alone AnsProlog* programs. As motivated in Section 1.4 often AnsProlog* programs are used to encode a function. This is particularly the case when AnsProlog* programs are used to express database queries and views. To date, most research in using AnsProlog* programs to express database queries has been with respect to AnsProlog programs. Considering that, our goal in this section is to develop techniques to transform AnsProlog programs so that they behave 'reasonably' when the CWA that is inherent in AnsProlog is removed. Currently there are a large number of queries that are expressed by AnsProlog programs, and at times there is a need to expand these queries so that they can accept incomplete inputs. The techniques of this section will be very useful for the above task.

To formulate the notions of 'reasonable behavior' in presence of incomplete input, expanding the query so that it accepts incomplete input, and transforming an AnsProlog program so that it interpolates the function represented by the original program, we will use many of the database notions such as *database, database instance*, and *query*, which we introduced in Section 1.4.1 and 1.4.2. We now start with a motivating scenario.

Consider a domain $\mathcal{U}$ of individuals, Sam, John, Peter, Mary, and Alberto. Now consider a database instance D consisting of the facts

$par(sam, john)$, $par(john, peter)$, and $par(mary, alberto)$;

where $par(X, Y)$ means that Y is a parent of X.

Now if we would like to ask a *query* Q about the ancestor and non-ancestor pairs in D, we can express it using the AnsProlog program Π_0:

$$\left.\begin{array}{l} anc(X,Y) \leftarrow par(X,Y) \\ anc(X,Y) \leftarrow par(X,Z), anc(Z,Y) \end{array}\right\} \Pi_0$$

The above representation of the *query* assumes the Close World Assumption (CWA) [Rei78] about the database instance D, which says that an atom f will be inferred to be $false$ w.r.t. D if it is not entailed by D. We also refer to D as a *complete database*. By $CWA(D)$ we denote the set $\{f \ : \ f$ is an atom in the language and $f \notin D\}$, and the meaning of a complete database D is expressed by the set $D \cup \neg CWA(D)$, where for any set S of atoms $\neg S$ denotes the set $\{\neg f \ : \ f \in S\}$.

The *query* Q represented by the AnsProlog program Π_0 can be considered as a function from instances of the relation *par* to instances of the relation *anc*. For example,

$Q(\{par(sam, john), par(john, peter), par(mary, alberto)\}) =$
$\{anc(sam, john), anc(john, peter), anc(sam, peter), anc(mary, alberto)\}$.

The function Q can be expressed by the AnsProlog program Π_0 in the following way:

$$Q(D) = \{anc(X, Y) \ : \ \Pi_0 \cup D \models anc(X, Y)\} \tag{3.7.3}$$

Now, let us consider the domain $\{a, b, c, d, e\}$, where a, b, c, d, and e denote remains of five different individuals found from an archaeological site. Using sophisticated tests scientists are able to determine that b is a parent of a and c is a parent of b, and neither e is parent of any of a, b, c and d nor any of them is the parent of e. This information can be represented by the following set S of literals:

$S = \{par(a, b), par(b, c), \neg par(a, e), \neg par(b, e), \neg par(c, e), \neg par(d, e), \neg par(e, a),$
$\neg par(e, b), \neg par(e, c), \neg par(e, d), \neg par(a, d)\}$.

The set S is not a database instance because it contains negative literals. Recall from Section 1.4.2 that sets of both positive and negative literals are referred to as *incomplete database instances* or simply incomplete databases, in database terminology. Hence, S is an incomplete database instance.

As before we are interested in ancestor and non-ancestor pairs. But this time with respect to the incomplete database S. The function Q and our earlier concept of *query* are no longer appropriate as they require database instances as input. To define ancestor and non-ancestor pairs using the information in S, we need to consider the notion of *extended query* – from Section 1.4.2 – which allows input to be incomplete databases. In this we can not ignore the negative literals in S and compute Q with respect to the database consisting of the positive literals in S. If we do that we will be able to infer $\neg anc(a, d)$. This is not correct intuitively, because it is possible that with further tests the scientist may determine $par(c, d)$, which will be consistent with the current determination and which will force us to infer $anc(a, d)$.

Since we can not ignore the negative literals in S, and AnsProlog programs such as $\Pi_0 \cup D$ do not allow rules with negative literals in their head we will use AnsProlog$^\neg$ programs to represent the query that determines ancestor and non-ancestor pairs from S. Let us denote this query by Q'. The next question is how are Q and Q' related?

While Q' allows incomplete databases as inputs, Q only allows complete databases. But they must coincide when the inputs are complete databases. Moreover, for any incomplete database X', $Q'(X')$ must not disagree with $Q(X)$, for any complete database X that extends X'. The intuition behind it is that if currently we have the incomplete database X' then our partial conclusion about ancestors and non-ancestors should be such that in presence of additional information we do not retract our earlier conclusion. Also, given an incomplete database X', the extended query Q' should be such that $Q'(X')$ contains all information about ancestors and non-ancestors that

'critically depend' on X'. In other words for any X', if for all complete database X that agree with X' an ancestor pair is true in $Q(X)$, then that ancestor pair must be true in $Q'(X')$. Similarly, about non-ancestor pairs. In general, given a query Q an extended query Q' that satisfies the above properties is referred to as its *expansion*. We formally define *expansion* of a query by an extended query and relate it to the above described properties in Definition 47 and Proposition 57 respectively.

Now that we have an intuitive idea about Q', the next question is how to express it using an AnsProlog$^\neg$ program.

Let us start with the AnsProlog program Π that expresses the query Q. Since, we would like to allow incomplete databases as inputs, let us consider Π as an extended logic program. But then we no longer have the CWA about ancestor. The next step would be to consider the AnsProlog$^\neg$ program obtained by adding explicit CWA about ancestors to the AnsProlog program Π representing Q. The resultant AnsProlog$^\neg$ program is given as follows:

$\neg anc(X, Y) \leftarrow$ **not** $anc(X, Y)$

$anc(X, Y) \leftarrow par(x, Y)$

$anc(X, Y) \leftarrow par(X, Z), anc(Z, Y)$

But the resultant AnsProlog$^\neg$ program is not an adequate expression of Q'. It only works when the input database instance is complete and does not work for S, as it incorrectly infers $\neg anc(a, d)$.

In this section we adequately express this extended query through an AnsProlog$^\neg$ program and argue that it expands the query expressed by Π. The more general question we answer in this section is how to expand arbitrary queries, from complete databases that are expressed by AnsProlog programs, so that they are applicable to incomplete databases. We use the term 'expand' because our goal is to expand the domain of the queries from complete databases only to databases that may be incomplete. We also refer to the AnsProlog$^\neg$ program T that expands Q as the *interpolation* of Π, or say T *interpolates* Π. The intuition behind the term 'interpolation' is that T agrees with Π on all inputs where Π is defined, and for inputs where Π is not defined T interpolates to a value based on the mappings of Π on the neighboring (complete) inputs.

With this motivation we first formulate the l-functions that are encoded by AnsProlog and AnsProlog$^\neg$ programs.

3.7.1 The l-functions of AnsProlog and AnsProlog$^\neg$ programs

Recall – from Section 1.4.2 – that an AnsProlog* program and two sets of literals $\mathcal{P}$ (called *parameters*) and $\mathcal{V}$ (called *values*) partially define an l-function. We need the following notation before presenting the complete definition of the l-function of our interest.

Definition 44 Let R and S be sets of ground literals over a language $\mathcal{L}$ of an AnsProlog$^\neg$ program Π. $\Pi_R \mid S$ the S-consequences of Π and R, is defined as follows:
$\Pi_R \mid S = \{s : s \in S \, and \, \Pi \cup R \models s\}$ □

Using the above notation we now precisely define an AnsProlog program that represents a query Q to be a (partial) function from complete sets of literals from the parameter (of Q) to sets of literals from the value (of Q). But, since both parameter and values may have literals, we will have to be careful about the domain of the query specified by a general logic program.

- The *first* requirement is that the elements of the domain must be *complete* w.r.t. $\mathcal{P}$. But, not all complete sets of literals from the parameter will be in the domain.
- We will *additionally require* that each element X of the domain should be a valid input in the sense that when added to Π the resulting AnsProlog program should not entail different literals from the parameter than that is in X.

The following definition makes the above ideas precise.

Definition 45 Let X be an arbitrary set from $2^{\mathcal{P}}$ and let $Y = \Pi_{atoms(X)}|\mathcal{V}$. We will say that X is a *valid input* of Π ($X \in Dom(\Pi)$) and Y is the value of Π at X (i.e. $Y = \Pi(X)$) if the following hold:

1. $X = \Pi_{atoms(X)} \mid \mathcal{P}$
2. X is complete w.r.t. $\mathcal{P}$.

Definition 46 For any set $X \in 2^{\mathcal{P}}$, a superset $\hat{X}$ of X is said to be a Π-*extension of* X if $\hat{X} \in Dom(\Pi)$. We denote the set of all Π-extension of X by $S_\Pi(X)$. We omit Π from $S_\Pi(X)$ when it is clear from context.

Intuitively, given a set $X \in 2^{\mathcal{P}}$, $S_\Pi(X)$ denotes the different ways X can be completed with additional consistent information, and still be a valid input for Π.

We are now almost ready to precisely define the interpolation of an AnsProlog program which we will specify through an AnsProlog$^\neg$ program.

We view an AnsProlog$^\neg$ program T to be a function from $2^{\mathcal{P}}$ to $2^{\mathcal{V}}$ such that $T(X) = T_X \mid \mathcal{V} = \{s \ : s \in \mathcal{V} \ and \ T \cup X \models s\}$. Since we are only interested in AnsProlog$^\neg$ programs that are interpolations, we do not restrict the domain of T.

3.7.2 Interpolation of an AnsProlog program and its properties

Definition 47 (Interpolation) Let Π be an AnsProlog program representing a query Q, with parameters $\mathcal{P}$ and values $\mathcal{V}$. We say that an AnsProlog$^\neg$ program T *interpolates* Π, w.r.t $\mathcal{P}$ and $\mathcal{V}$ if for every $X \in 2^{\mathcal{P}}$

$$T(X) = \bigcap_{\hat{X} \in S(X)} \Pi(\hat{X}) \tag{3.7.4}$$

Moreover, we also say that the extended query represented by (the AnsProlog$^\neg$ program) T expands the query represented by (the AnsProlog program) Π. □

For convenience, we will just say that T interpolates Π without mentioning the query Q, the parameters $\mathcal{P}$ and the values $\mathcal{V}$, whenever they are clear from the context. But it should be noted that T interpolates Π w.r.t. $\mathcal{P}$ and $\mathcal{V}$ does not necessarily mean that T will also be an interpolation of Π for a different query with a different pair of parameter and value. The choice of Q, $\mathcal{P}$ and $\mathcal{V}$ is an integral part of the program. The programmer designs the program with that choice in mind. This is similar to the choices and assumptions a Prolog programmer makes about whether a particular attribute of a predicate in the head of a rule will be ground or not when that predicate is invoked.

The following proposition breaks down the definition of interpolation to three different intuitive properties: equivalence, monotonicity and maximal informativeness. The equivalence property states that Π and T must be equivalent w.r.t. complete inputs. The monotonicity property states that T which accepts incomplete inputs should be monotonic, i.e. in the presence of additional consistent information it should not retract any of its earlier conclusions. The maximal informativeness property states that given an incomplete input X, the interpolation T should entail all literals that are entailed by Π w.r.t. all the complete extensions of X that are in the domain of Π. Intuitively, it means that if Π entails l regardless of what complete extension of X is given to Π as an input then X has enough information to make a decision on l and hence the interpolation T should entail l with X as the input.

Proposition 57 An AnsProlog$^\neg$ program T interpolates a general logic program Π iff the following conditions are satisfied:

1. (**Equivalence**) For every $X \in Dom(\Pi)$, $\Pi(X) = T(X)$.
2. (**Monotonicity**) T is monotonic, i.e. for every $X_1, X_2 \subseteq \mathcal{P}$, if $X_1 \subseteq X_2$ then $T(X_1) \subseteq T(X_2)$.
3. (**Maximal Informativeness**) For every $v \in \mathcal{V}$ and every $X \subseteq \mathcal{P}$, if for all $\hat{X} \in S(X)$ $v \in \Pi(\hat{X})$ then $v \in T(X)$. □

Proof "$\Longrightarrow$"
Since every $X \in Dom(\Pi)$ is complete $S(X) = X$. Hence, condition 1 holds. It is obvious that condition 3 holds. Condition 2 holds because $X_2 \subseteq X_1 \Rightarrow S(X_1) \subseteq S(X_2)$

"$\Longleftarrow$"
Condition 3 implies $T(X) \supseteq \bigcap_{\hat{X} \in S(X)} \Pi(\hat{X})$.
We now only need to show that $\forall X \subseteq \mathcal{P}$, $T(X) \subseteq \bigcap_{\hat{X} \in S(X)} \Pi(\hat{X})$.
(case 1) $S(X) = \emptyset$
$\Rightarrow \bigcap_{\hat{X} \in S(X)} \Pi(\hat{X}) = Lit(\mathcal{P})$
$\Rightarrow T(X) \subseteq \bigcap_{\hat{X} \in S(X)} \Pi(\hat{X})$
(case 2) $S(X) \neq \emptyset$
Let $s \in T(X)$. Since, for all $\hat{X} \in S(X)$, $X \subseteq \hat{X}$, by monotonicity (2.) we have $s \in T(\hat{X})$. But using equivalence (1.) we have $s \in \Pi(\hat{X})$. Hence,

$$\forall X \subseteq \mathcal{P}, T(X) \subseteq \bigcap_{\hat{X} \in S(X)} \Pi(\hat{X})$$

In the following example we give an interpolation of a non-stratified AnsProlog program.

Example 71 Consider the following AnsProlog program Π_2:

$$\left.\begin{array}{l} c(X) \leftarrow p(X) \\ c(X) \leftarrow q(X) \\ p(X) \leftarrow \textbf{not}\ \ q(X), r(X) \\ q(X) \leftarrow \textbf{not}\ \ p(X), r(X) \end{array}\right\} \Pi_2$$

Here $\mathcal{P} = Lit(r)$ and $\mathcal{V} = Lit(c)$. It is easy to see that for every $X \in 2^{\mathcal{P}}$, $\Pi_2 \cup X$ is call-consistent [Kun89]. Using results from [LT94] it is easy to check that Π_2 is defined for any complete and

consistent X, $X \subseteq \mathcal{P}$. (The program $\Pi_2 \cup X$ can then be split into three layers, the bottom layer consisting of X, the next layer consisting of the rules with either p or q in the head and the top layer consisting of the rules with c in the head. The answer sets of $\Pi_2 \cup X$ can then be computed bottom-up starting from the bottom layer.)

Consider T_2 obtained from Π_2 by adding to it the rule

$\neg c(X) \leftarrow \neg r(X)$. □

Proposition 58 *[BGK98]* T_2 is an interpolation of Π_2. □

So far in this section we have made precise the notion of interpolation (and query expansion) and proved the interpolation results for two particular AnsProlog programs. But we still do not know which program interpolates the program Π_0 that represents the ancestor query. Our goal now is to come up with a precise algorithm that constructs interpolations of AnsProlog programs.

3.7.3 An algorithm for interpolating AnsProlog programs

In this subsection we present an algorithm which constructs an interpolation of a large class of AnsProlog programs with some restrictions on its parameter and value. We now make these restrictions precise.

Let Π be an AnsProlog program in language $\mathcal{L}$. We only consider the query Q with values $\mathcal{V}$ consisting of all ground literals formed with predicates in the heads of the rules in Π (called IDB predicates), and parameters $\mathcal{P}$ consisting of all other ground literals in Lit (called EDB predicates). An AnsProlog program Π representing a query Q that satisfies the above property is called a *natural representation* of Q.

Before we give the algorithm we demonstrate the intuition behind the algorithm by using the ancestor query as an example.

Interpolation of the Transitive Closure Program

Recall the ancestor query represented as AnsProlog program was:

$$\left.\begin{array}{l} anc(X,Y) \leftarrow par(X,Y) \\ anc(X,Y) \leftarrow par(X,Z), anc(Z,Y) \end{array}\right\} \Pi_0$$

where *par* is the EDB predicate and *anc* is the IDB predicate.

Our goal is to construct an interpolation of this program. We refer to our interpolation program as $\mathcal{A}(\Pi_0)$.

The main idea behind our construction is to have a program which coincides with Π_0 on atoms of the form $anc(a,b)$ and derives $\neg anc(a,b)$ when no consistent extension of the original EDB may derive $anc(a,b)$. To achieve this we introduce a new predicate m_{anc}, where $m_{anc}(c,d)$ intuitively means that " c may bean ancestor of d". Our information about m_{anc} is complete hence $\neg m_{anc}$ coincides with **not** m_{anc} and hence we define $\neg anc$ using the rule:

$\neg anc(X,Y) \leftarrow$ **not** $m_{anc}(X,Y)$.

Next we modify the original rules to define m_{anc} and use another new predicate m_{par}, where $m_{par}(c,d)$ means that "c may be a parent of d". We now define m_{anc} using the rules:

$m_{anc}(X,Y) \leftarrow m_{par}(X,Y)$
$m_{anc}(X,Y) \leftarrow m_{par}(X,Z), m_{anc}(Z,Y)$

We now need to define m_{par}. Intuitively, in presence of incomplete information about parents we can say "e may be a parent of f if we do not know for sure that e is not a parent of f." This can be written as the rule:

$m_{par}(X,Y) \leftarrow \textbf{not } \neg par(X,Y)$

Putting all the above rules together we have an interpolation of Π_0 denoted by $\mathcal{A}(\Pi_0)$.

$$\left.\begin{array}{l} m_{par}(X,Y) \leftarrow \textbf{not } \neg par(X,Y) \\ m_{anc}(X,Y) \leftarrow m_{par}(X,Y) \\ m_{anc}(X,Y) \leftarrow m_{par}(X,Z), m_{anc}(Z,Y) \\ \neg anc(X,Y) \leftarrow \textbf{not } m_{anc}(X,Y) \\ anc(X,Y) \leftarrow par(X,Y) \\ anc(X,Y) \leftarrow par(X,Z), anc(Z,Y) \end{array}\right\} \mathcal{A}(\Pi_0)$$

where $\mathcal{P}$ and $\mathcal{V}$ are the same as for the program Π_0.

Theorem 3.7.1 *[BGK98][Interpolation]* $\mathcal{A}(\Pi_0)$ interpolates Π_0 with $\mathcal{P} = Lit(par)$ and $\mathcal{V} = Lit(anc)$. □

Since we were using the ancestor query as an example, the above theorem is a special case of a more general theorem (Theorem 3.7.2) to be stated later. Nevertheless, a separate proof of this theorem is important as it will illustrate proof techniques that can be used for proving interpolation results for more restricted class of AnsProlog programs such as, the class of Horn logic programs. Hence, a separate proof of this theorem is given in the Appendix.

The Interpolation Algorithm

Now we apply the idea from the previous section to arbitrary AnsProlog programs. We expand the language of such a program Π by introducing intermediate predicate m_p for every predicate p in Π. Intuitively $m_p(\bar{t})$ means that $p(\bar{t})$ may be true.

To define these predicates we consider two cases. Since negative information about EDB predicates is explicitly given, this intuition is captured by the rule:

$$m_q(\bar{t}) \leftarrow \textbf{not } \neg q(\bar{t}) \qquad \text{(v1)}$$

The definition of m_a for an IDB predicate a can not rely on negative information about a. Instead it uses the corresponding definition of a from Π. More precisely, $\mathcal{A}(\Pi)$ contains the rule

$$m_a(\bar{t}) \leftarrow m_{b_1}(\bar{t}), \ldots, m_{b_n}(\bar{t}), \textbf{not } c_1(\bar{t}), \ldots, \textbf{not } c_n(\bar{t}) \qquad \text{(v2)}$$

for every rule

$$a(\bar{t}) \leftarrow b_1(\bar{t}), \ldots, b_m(\bar{t}), \textbf{not } c_1(\bar{t}), \ldots, \textbf{not } c_n(\bar{t}) \qquad \text{(v3)}$$

in Π.

For any program Π, the program $\mathcal{A}(\Pi)$ contains the rule

$$a(\bar{t}) \leftarrow b_1(\bar{t}), \ldots, b_m(\bar{t}), \neg c_1(\bar{t}), \ldots, \neg c_n(\bar{t}) \qquad \text{(v4)}$$

for every rule (v3) in Π. Rule (v4) is intended to be a monotonic and weakened version of rule (v2). The intuition is that we no longer want to make hasty (revisable) conclusion using the negation as

failure operator (**not**) in the body. The negation as failure operator is therefore replaced by $\neg$ in (v4). On the other hand we would like $m_a(\bar{t})$ to be true if there is any possibility that $a(\bar{t})$ is true. Hence, in the body of (v2) we still have the negation as failure operator. Once the truth of $a(\bar{t})$ is established by $\mathcal{A}(\Pi)$ through the rule (v4) it will never be retracted in presence of additional consistent information about the EDB predicates.

The above definitions ensure that the interpolation does not entail $m_p(\bar{t})$ for an input X iff it does not entail $p(\bar{t})$ for any consistent extension Y of X. Hence it is safe to infer $\neg p(\bar{t})$ when the interpolation does not entail $m_p(\bar{t})$. This observation leads to the following definition of falsity of IDB predicates

$\neg p(\bar{t}) \leftarrow$ **not** $m_p(\bar{t})$. (v5)

The above intuitions can be summarized into the following algorithm that constructs the interpolations of a large class of AnsProlog programs.

Algorithm 1 *For any AnsProlog program* Π *the AnsProlog*$^\neg$ *program* $\mathcal{A}(\Pi)$ *contains the following rules:*

1. *If* q *is an EDB in the program* Π, $\mathcal{A}(\Pi)$ *contains the rule:*
   $m_q(\bar{t}) \leftarrow$ **not** $\neg q(\bar{t})$.
2. *For any rule* $a(\bar{t}) \leftarrow b_1(\bar{t}), \ldots, b_m(\bar{t}),$ **not** $c_1(\bar{t}), \ldots,$ **not** $c_n(\bar{t})$ *in* Π, $\mathcal{A}(\Pi)$ *contains the rules:*
   (a) $m_a(\bar{t}) \leftarrow m_{b_1}(\bar{t}), \ldots, m_{b_m}(\bar{t}),$ **not** $c_1(\bar{t}), \ldots,$ **not** $c_n(\bar{t})$,
   (b) $a(\bar{t}) \leftarrow b_1(\bar{t}), \ldots, b_m(\bar{t}), \neg c_1(\bar{t}), \ldots, \neg c_n(\bar{t})$.
3. *If* p *is an IDB in the program* Π, $\mathcal{A}(\Pi)$ *contains the rule:*
   $\neg p(\bar{t}) \leftarrow$ **not** $m_p(\bar{t})$.
4. *Nothing else is in* $\mathcal{A}(\Pi)$. □

Example 72 (Transitive Closure) Consider Π_0 from the previous subsection:
$anc(X,Y) \leftarrow par(X,Y)$
$anc(X,Y) \leftarrow par(X,Z), anc(Z,Y)$
Then the transformation $\mathcal{A}(\Pi)$ based on the above algorithm is exactly the AnsProlog$^\neg$ program $\mathcal{A}(\Pi_0)$ obtained in Section 3.7.3. □

3.7.4 Properties of the transformation $\mathcal{A}$

In this subsection we formalize several properties of the transformation $\mathcal{A}$. We show that for a large class of AnsProlog programs it constructs an interpolation and for all AnsProlog programs the transformed program satisfies weaker versions of interpolation.

Theorem 3.7.2 *[BGK98] [Properties of* $\mathcal{A}(\Pi)$ *for a signed program* Π*]* For any signed AnsProlog program Π that is a natural representation of a query Q, $\mathcal{A}(\Pi)$ is an interpolation of Π. □

The following example shows an unsigned but stratified AnsProlog program for which Algorithm 1 does not construct an interpolation.

Example 73 Consider the program Π_6:

$$\left.\begin{array}{l} p \leftarrow q \\ p \leftarrow \textbf{not } q \end{array}\right\} \Pi_6$$

Here $\mathcal{V} = \{p, \neg p\}, \mathcal{P} = \{q, \neg q\}$. It is easy to see that for any $X \subseteq \mathcal{P}$ $p \in \Pi(X)$.

Program $\mathcal{A}(\Pi_6)$ consists of the following rules:

$$\left.\begin{array}{l} m_q \leftarrow \textbf{not } \neg q \\ m_p \leftarrow m_q \\ m_p \leftarrow \textbf{not } q \\ p \leftarrow q \\ p \leftarrow \neg q \\ \neg p \leftarrow \textbf{not } m_p \end{array}\right\} \mathcal{A}(\Pi_6) = T$$

Consider $X = \emptyset$ and the AnsProlog$^\neg$ program $T \cup X = T$. It is easy to see that T^+ is a stratified program, therefore T has only one answer set A. Evidently, $A = \{m_p, m_q\}$ and $p \notin A$. Therefore

$$\mathcal{A}(\Pi)(X) \neq \bigcap_{\hat{X} \in S(X)} \Pi(\hat{X})$$

□

The following theorem states that the transformation $\mathcal{A}$ preserves categoricity for signed AnsProlog programs.

Theorem 3.7.3 *[BGK98] [Categoricity]* Let Π be a signed AnsProlog program with signing R, a natural representation of a query Q, and $X \subseteq \mathcal{P}$. Program $\Pi \cup atoms(X)$ is categorical if and only if program $\mathcal{A}(\Pi) \cup X$ is categorical. □

We now introduce some weaker notions of interpolation which will be satisfied by a larger class than the class of signed AnsProlog programs.

Definition 48 (Weak and Sound Interpolation) Let Π be a AnsProlog program, with parameter $\mathcal{P}$ and value $\mathcal{V}$. We say an AnsProlog$^\neg$ program T is a *weak interpolation* of Π w.r.t. $\mathcal{P}$ and $\mathcal{V}$ if the following three conditions are satisfied.

1. (**Monotonicity**) For every $X_1, X_2 \subseteq \mathcal{P}$, if $X_1 \subseteq X_2$ then $T(X_1) \subseteq T(X_2)$.
2. (**Equivalence**) For every $X \in Dom(\Pi)$, $\Pi(X) = T(X)$.
3. For any $X \subseteq \mathcal{P}$
$$T(X) \subseteq \bigcap_{\hat{X} \in S(X)} \Pi(\hat{X})$$

If only the first and the third conditions are satisfied then we say T to be a *sound interpolation* of Π w.r.t. $\mathcal{P}$ and $\mathcal{V}$. □

From the third condition above it is clear that if T is a sound interpolation of Π then it satisfies the weak equivalence property defined as:

For every $X \in Dom(\Pi)$, $T(X) \subseteq \Pi(X)$.

We now state two more results about the properties of the transformation $\mathcal{A}(\Pi)$.

Theorem 3.7.4 *[BGK98][Properties of $\mathcal{A}(\Pi)$ for a stratified general program Π]*Let Π be a stratified AnsProlog program and a natural representation of a query Q, then $\mathcal{A}(\Pi)$ is a weak interpolation of Π. □

Theorem 3.7.5 *[BGK98][Properties of $\mathcal{A}(\Pi)$ for an arbitrary AnsProlog program Π]* Let Π be an AnsProlog program and a natural representation of a query Q, then $\mathcal{A}(\Pi)$ is a sound interpolation of Π. □

In summary, the transformation $\mathcal{A}$ constructs interpolations for signed AnsProlog programs, weak interpolations for stratified AnsProlog programs and sound interpolations for all programs. Moreover, the transformation preserves categoricity for signed programs and preserves stratification (i.e. If Π is stratified then $\mathcal{A}(\Pi)^+$ is stratified) for stratified programs.

3.8 Building and refining programs from components: functional specifications and Realization theorems

In Section 3.2.3 we presented results about when a program Π enhanced with additional rules preserves the meaning of the original program. These results are part of a larger theme that will allow us to modularly build larger and/or refined programs from smaller components. In this section we continue with this theme. We follow Section 1.4.3 in viewing programs (together with a domain and an input and an output signature) as s-functions – which we also refer to as lp-functions, and define operators to compose and refine these functions and present results that state when such composition and refinement can be realized when using AnsProlog* programs.

3.8.1 Functional Specifications and lp-functions

To validate lp-functions with respect to an independent specification, we first introduce the notion of a functional specification.

A four-tuple $f = \{f, \sigma_i(f), \sigma_o(f), dom(f)\}$ where

1. $\sigma_i(f)$ and $\sigma_o(f)$ are signatures;
2. $dom(f) \subseteq states(\sigma_i(f))$;
3. f is a function which maps $dom(f)$ into $states(\sigma_o(f))$

is called an *f-specification (or functional specification)* with input signature $\sigma_i(f)$, output signature $\sigma_o(f)$ and domain $dom(f)$. States over $\sigma_i(f)$ and $\sigma_o(f)$ are called input and output states respectively.

We now formally define lp-functions.

A four-tuple $\pi = \{\pi, \sigma_i(\pi), \sigma_o(\pi), dom(\pi)\}$ where

1. π is an AnsProlog$^\neg$ program (with some signature $\sigma(\pi)$);
2. $\sigma_i(\pi), \sigma_o(\pi)$ are sub-signatures of $\sigma(\pi)$ called input and output signatures of π respectively;
3. $dom(\pi) \subseteq states(\sigma_i(\pi))$

is called an *lp-function* if for any $X \in dom(\pi)$ program $\pi \cup X$ is consistent, i.e., has a consistent answer set. For any $X \in dom(\pi)$,

$$\pi(X) = \{l : l \in lit(\sigma_o(\pi)), \pi \cup X \models l\}.$$

We say that an lp-function π *represents* an f-specification f if π and f have the same input and output signatures and domains and for any $X \in dom(f)$, $f(X) = \pi(X)$.

3.8.2 The compositional and refinement operators

In this section we discuss four operators on functional specifications that will allow us to compose and refine them. The four operators are *incremental extension*, *interpolation*, *input opening* and *input extension*. Among these the first one is a binary compositional operator that defines a particular composition of functional specifications. The rest three are unary refinement operators that refine a functional specification. All three of them focus on the input domain of the function. The interpolation operator refines a functional specification so that its domain is unchanged but for certain elements in the domain the function follows the notion of interpolation in Section 3.7. The input opening operator is a further refinement of the interpolation operator where besides interpolation, the enhanced function also reasons about the input itself and expands them to the maximum extent possible. The input extension operator refines a functional specification for particular enlargement of its domain. We now formally define these operators.

- **Incremental extension**

  Specifications f and g s.t. $\sigma_o(f) = \sigma_i(g)$ and $lit(\sigma_i(g)) \cap lit(\sigma_o(g)) = \emptyset$ can be combined into a new f-specification $g \circ f$ by a specification constructor $\circ$ called *incremental extension*. Function $g \circ f$ with domain $dom(f)$, $\sigma_i(g \circ f) = \sigma_i(f)$, $\sigma_o(g \circ f) = \sigma_o(f) + \sigma_o(g)$ is called the *incremental extension* of f by g if for any $X \in dom(g \circ f)$, $g \circ f(X) = f(X) \cup g(f(X))$.

  An example of incremental extension is as follows. Let f be a functional specification whose input signature consists of disjoint predicates corresponding to subclasses of birds such as eagles, canaries, pigeons, penguins, etc., and output signature consists of the predicate fly. Let g be a functional specification with input signature consisting of the predicate fly and the output signature consisting of the predicate *cage_needs_top*. The incremental extension of f by g can then be used to make inference about whether a particular bird's cage needs a top or not.

- **Interpolation and domain completion**

  Let D be a collection of states over some signature σ. A set $X \in states(\sigma)$ is called *D-consistent* if there is $\hat{X} \in D$ s.t. $X \subseteq \hat{X}$; $\hat{X}$ is called a *D-cover* of X. The set of all D-covers of X is denoted by $c(D, X)$.

  Definition 49 *(Interpolation)* Let f be a closed domain f-specification with domain D. F-specification $\tilde{f}$ with the same signatures as f and the domain $\tilde{D}$ is called the *interpolation* of f if

  $$\tilde{f}(X) = \bigcap_{\hat{X} \in c(D,X)} f(\hat{X}) \tag{3.8.5}$$

  This is a slight generalization of the notion of interpolation of Section 3.7 where we only considered interpolations of functions defined by AnsProlog programs.

  An example of interpolation is when we have a functional specification f that maps complete initial states to the value of fluents in the future states, and to be able to reason in presence of incomplete initial states we need an interpolation of the functional specification f.

Definition 50 A set $X \in states(\sigma)$ is called *maximally informative* w.r.t. a set $D \subseteq states(\sigma)$ if X is D-consistent and

$$X = \bigcap_{\hat{X} \in c(D,X)} \hat{X} \tag{3.8.6}$$

By $\tilde{D}$ we denote the set of states of σ maximally informative w.r.t. D. □

Definition 51 *(Domain Completion)*
Let D be a collection of complete states over signature σ. The *domain completion* of D is a function h_D which maps D-consistent states of σ into their maximally informative supersets. □

Our particular interest in domain completion is when dealing with the interpolation $\tilde{f}$ of a closed domain f-specification f with domain D. In that case we refer to the *domain completion* of D as the function $\tilde{f}_D$.

- **Input Opening**

The set of all D-consistent states of σ is called the *interior* of D and is denoted by D°. An f-specification f defined on a collection of complete states of $\sigma_i(f)$ is called *closed domain specification.*

Definition 52 *(Input Opening)* Let f be a closed domain specification with domain D. An f-specification f° is called the *input opening* of f if

$$\sigma_i(f^\circ) = \sigma_i(f) \qquad \sigma_o(f^\circ) = \sigma_i(f) + \sigma_o(f) \tag{3.8.7}$$

$$dom(f^\circ) = D^\circ \tag{3.8.8}$$

$$f^\circ(X) = \bigcap_{\hat{X} \in c(D,X)} f(\hat{X}) \cup \bigcap_{\hat{X} \in c(D,X)} \hat{X} \tag{3.8.9}$$

The following proposition follows immediately from the definitions.

Proposition 59 For any closed domain f-specification f with domain D

$$f^\circ = \tilde{f} \circ \tilde{f}_D \tag{3.8.10}$$

- **Input extension**

Definition 53 Let f be a functional specification with disjoint sets of input and output predicates. A f-specification f^* with input signature $\sigma_i(f) + \sigma_o(f)$ and output signature $\sigma_o(f)$ is called *input extension* of f if

1. f^* is defined on elements of $dom(f)$ possibly expanded by consistent sets of literals from $\sigma_o(f)$,
2. for every $X \in dom(f)$, $f^*(X) = f(X)$,
3. for any $Y \in dom(f^*)$ and any $l \in lit(\sigma_o(f))$,
   (i) if $l \in Y$ then $l \in f^*(Y)$
   (ii) if $l \notin Y$ and $\bar{l} \notin Y$ then $l \in f^*(Y)$ iff $l \in f(Y \cap lit(\sigma_i(f)))$ □

An example of a situation where input extension is necessary is when we have a functional specification f that maps possibly incomplete initial states to the value of fluents in the future states. Now suppose we would like to enhance our representation such that we can also have oracles about the value of fluents in the future states as input. In this case we will need the input extension of f.

Now that we have defined several operators on f-specifications, we now proceed to realize these operators when dealing with lp-functions. We start with a realization theorem for incremental extension which states how to build and when we can build an lp-function representing $g \circ f$ from lp-functions that represent f and g. Similarly, the realization theorems for the other unary operators (say, o) state how to build and when we can build an lp-function representing $o(f)$ from an lp-function that represent f.

3.8.3 Realization theorem for incremental extension

Definition 54 An lp-function $\{\pi, \sigma_i(\pi), \sigma_o(\pi), dom(\pi)\}$ is said to be *output-functional* if for any $X \in dom(\pi)$ and any answer sets A_1 and A_2 of $\pi \cup X$ we have $A_1 \cap lit(\sigma_o) = A_2 \cap lit(\sigma_o)$. □

Definition 55 We say that lp-functions $\{\pi_F, \sigma_i(\pi_F), \sigma_o(\pi_F), dom(\pi_F)\}$ and $\{\pi_G, \sigma_i(\pi_G), \sigma_o(\pi_G), dom(\pi_G)\}$ are *upward compatible* if

- $\sigma_o(\pi_F) = \sigma_i(\pi_G)$,
- $head(\pi_G) \cap (lit(\pi_F) \cup lit(\sigma_i(\pi_F)) \cup lit(\sigma_o(\pi_F))) = \emptyset$
- $(lit(\pi_G) \cup lit(\sigma_i(\pi_G)) \cup lit(\sigma_o(\pi_G))) \cap (lit(\pi_F) \cup lit(\sigma_i(\pi_F)) \cup lit(\sigma_o(\pi_F))) \subseteq lit(\sigma_o(\pi_F))$ □

Theorem 3.8.1 Let f and g be functional specifications represented by output functional lp-functions π_F and π_G, and let $g \circ f$ be the *incremental extension of f by g*. If π_F is upward compatible with π_G, then the lp-function $\pi_{G \circ F} = \langle \pi_G \cup \pi_F, \sigma_i(\pi_F), \sigma_o(\pi_F) \cup \sigma_o(\pi_G), dom(\pi_F) \rangle$ represents $g \circ f$. □

3.8.4 Realization theorem for interpolation

To give a realization theorem for the interpolation we need the following auxiliary notions.

Let D be a collection of complete states over a signature σ. Function f defined on the interior of D is called *separable* if

$$\bigcap_{\hat{X} \in c(D,X)} f(\hat{X}) \subseteq f(X)$$

or, equivalently, if for any $X \in dom(f)$ and any output literal l s.t. $l \notin f(X)$ there is $\hat{X} \in c(D, X)$ s.t. $l \notin f(\hat{X})$.
The following lemma and examples help to better understand this notion.

Lemma 3.8.2 Let D be the set of complete states over some signature σ_i and let π be an lp-function defined on $D^\circ = states(\sigma_i)$, s.t.

1. The sets of input and output predicates of π are disjoint and input literals do not belong to the heads of π;
2. for any $l \in \sigma_i$, $l \notin lit(\pi)$ or $\overline{l} \notin lit(\pi)$. (By $lit(\pi)$ we mean the collection of all literals which occur in the rules of the ground instantiation of π.)

Then π is separable. □

The following example shows that the last condition is essential.

Example 74 Let $D = \{\{p(a)\}, \{\neg p(a)\}\}$ and consider a function f_1 defined on D° by the program

$q(a) \leftarrow p(a)$

$q(a) \leftarrow \neg p(a)$

Let $X = \emptyset$. Obviously, $f_1(X) = \emptyset$ while $\bigcap_{\hat{X} \in c(D,X)} f_1(\hat{X}) = \{q(a)\}$ and hence f_1 is not separable.

Example 75 In some cases to establish separability of an lp-function π it is useful to represent π as the union of its independent components and to reduce the question of separability of π to separability of these components. Let π be an lp-function with input signature σ_i and output signature σ_o. We assume that the input literals of π do not belong to the heads of rules of π. We say that π is decomposable into independent components $\pi_0, \ldots, \pi_n$ if $\pi = \pi_0 \cup \ldots \cup \pi_n$ and $lit(\pi_k) \cap lit(\pi_l) \subseteq lit(\sigma_i)$ for any $k \neq l$. It is easy to check that, for any $0 \leq k \leq n$, four-tuple $\{\pi_k, \sigma_i, \sigma_o, dom(\pi)\}$ is an lp-function, and that if all these functions are separable then so is π. This observation can be used for instance to establish separability of function f_2 defined on the interior of the set D from the previous example by the program

$q_1(a) \leftarrow p(a)$

$q_2(a) \leftarrow \neg p(a)$

(The output signature of f_2 consists of a, q_1 and q_2).

□

Now we are ready to formulate our realization theorem for interpolation.

Theorem 3.8.3 *(Realization Theorem for Interpolation)* Let f be a closed domain specification with domain D represented by an lp-function π and let $\tilde{\pi}$ be the program obtained from π by replacing some occurrences of input literals l in $pos(\pi)$ by **not** $\overline{l}$. Then $\{\tilde{\pi}, \sigma_i(f), \sigma_o(f), dom(\tilde{f})\}$ is an lp-function and if $\tilde{\pi}$ is separable and monotonic then $\tilde{\pi}$ represents $\tilde{f}$. □

3.8.5 Representing domain completion and realization of input opening

Let C be a collection of constraints of the form $\leftarrow \Delta$ where $\Delta \subset lit(\sigma)$. A constraint is called *binary* if Δ consists of two literals. We say that a domain D is defined by C if D consists of complete sets from $states(\sigma)$ satisfying C.

Theorem 3.8.4 Let C be a set of binary constraints and D be the closed domain defined by C. Let $\tilde{\pi}_D$ be a program obtained from C by replacing each rule $\leftarrow l_1, l_2$ by the rules $\neg l_1 \leftarrow l_2$ and $\neg l_2 \leftarrow l_1$.

If for every $l \in lit(\sigma)$ there is a set $Z \in D$ not containing l then the lp-function $\{\tilde{\pi}_D, \sigma, \sigma, D^\circ\}$ represents domain completion of D. □

By virtue of Proposition 59 we can realize input opening by composing – through incremental extension – interpolation with domain completion.

3.8.6 Realization theorem for input extension

Definition 56 Let π be an lp-function. The result of replacing every rule

$$l_0 \leftarrow l_1, \ldots, l_m, \textbf{not } l_{m+1}, \ldots, \textbf{not } l_n$$

of π with $l_0 \in lit(\sigma_\circ(f))$ by the rule

$$l_0 \leftarrow l_1, \ldots, l_m, \textbf{not } l_{m+1}, \ldots, \textbf{not } l_n, \textbf{not } \overline{l}_0$$

is called the *guarded version* of π and is denoted by $\hat{\pi}$. □

Theorem 3.8.5 ([GP96]) *(Realization Theorem for Input Extension)*
Let f be a specification represented by lp-function π with signature σ. If the set $U = lit(\sigma) \setminus lit(\sigma_\circ)$ is a splitting set of π dividing π into two components $\pi_2 = top_U(\pi)$ and $\pi_1 = bot_U(\pi)$ then the lp-function $\pi^* = \pi_1 \cup \hat{\pi}_2$ represents the input extension f^* of f. □

3.9 Filter-Abducible AnsProlog$^{\neg, or}$ programs

Our concern in this section is to explore when an AnsProlog$^{\neg, or}$ program can incorporate abductive reasoning. Such reasoning is necessary to assimilate observations to a given theory and is a richer notion than the notion of input extension in Section 3.8.2 and 3.8.6 where we discuss the refinement of an lp-function so that the refined function allows input – without being inconsistent – that were only part of the output of the original lp-function.

Consider the following AnsProlog program Π_1:

$p \leftarrow a$
$p \leftarrow b$

Suppose we would like to assimilate our observation that p is true with respect to the knowledge encoded in Π_1. Then our intuitive conclusion would be that $a \vee b$ must be true. A reasoning mechanism that leads to such conclusions is called abductive reasoning and this particular example, we can not make the intended conclusion by just adding p to the program Π_1. Nor can we make the intended conclusion by adding the constraint $\leftarrow$ **not** p to Π_1. The later is referred to as *filtering* program Π_1 with the observation p.

Let us now consider the following program Π_1' that includes Π_1.

$p \leftarrow a$
$p \leftarrow b$
$a \leftarrow$ **not** $\neg a$

$\neg a \leftarrow$ **not** a
$b \leftarrow$ **not** $\neg b$
$\neg b \leftarrow$ **not** b

In this case, we can make the intended conclusion $(a \vee b)$ by adding the constraint $\leftarrow$ **not** p to Π_1'. Our goal in this section is to develop conditions when the intended abductive conclusions due to observations can be made by filtering a program with the observations. We refer to such programs as *filter-abducible*. We start with formalizing the notion of abduction in AnsProlog$^{\neg, or}$ programs.

3.9.1 Basic definitions: simple abduction and filtering

When formulating abduction in AnsProlog$^{\neg, or}$ programs we designate a complete subset *Abd Abd*) of *Lit* as abducible literals. The set of atoms in *Abd* are denoted by Abd_a. Similarly, we designate a complete subset Obs_l Obs_l) of *Lit* as observable literals. The set of atoms in Obs_l are denoted by Obs_a and the set of formulas made up of literals in Obs_l and classical connectives will be denoted by the set *Obs* . We will often refer to *Obs* as *the set of observables*, and a subset Q of *Obs* as an *observation*. Q may be represented by a set of formulas Q_f Q_f) with variables, where the variables serve as schema variables and are substituted with ground terms in the language to obtain Q.

Definition 57 (Explanation) Let Π be an AnsProlog$^{\neg, or}$ program and Q be an observation. A complete set of abducibles E (from *Abd*) is said to be an explanation of Q w.r.t. Π if $\Pi \cup E \models^{*} Q$ and $\Pi \cup E$ is consistent (i.e., it has a consistent answer-set). □

We would now like to define abductive entailment ($\models_{abd}$) with respect to the pair $\langle \Pi, Q \rangle$, which we refer to as an *abductive theory*. Reasoning using this abductive entailment relation will be our formulation of *abductive reasoning*.

Definition 58 *[Abductive Entailment]* (i) M is an answer set of $\langle T, Q \rangle$ if there exists an explanation E of Q w.r.t. T such that M is an answer set of $T \cup E$.
(ii) For any formula f, $\langle \Pi, Q \rangle \models_{abd} f$ if f is true in all answer sets of $\langle \Pi, Q \rangle$. □

Proposition 60 Abductive theories are monotonic with respect to addition of observations. □

Proof: Suppose we have $Q_1 \subseteq Q_2$. Then any explanation of Q_2 with respect to T is an explanation of Q_1 with respect to T. Thus answer sets of $\langle \Pi, Q_2 \rangle$ are answer sets of $\langle \Pi, Q_1 \rangle$ and hence, $\models_{\text{abd}}$ is monotonic with respect to Q. □

Definition 59 Let Π be an AnsProlog$^{\neg, or}$ program and Q be an observation. By $Filter(\Pi, Q)$, we refer to the subset of answer sets of Π which entail Q. □

Proposition 61 Entailment with respect to $Filter(\Pi, Q)$ is monotonic with respect to Q. □

Proof: Follows directly from the definition of $Filter(\Pi, Q)$. □

3.9.2 Abductive Reasoning through filtering: semantic conditions

We now present semantic conditions on AnsProlog$^{\neg, or}$ programs, abducibles and observables such that abductive reasoning can be done through filtering. We now formally define such triplets.

Definition 60 (Filter-abducible) An AnsProlog$^{\neg, or}$ program Π, a set Abd, and a set Obs are said to be *filter-abducible* if for all possible observations $Q \in Obs$; $Filter(\Pi, Q)$ is the set of answer sets of $\langle \Pi, Q \rangle$. □

Before we define conditions for filter-abducibility, we work out the filter-abducibility of the program Π_1 and Π_1' in the beginning of this section.

Example 76 Consider the AnsProlog$^{\neg}$ program [GL91] Π_1:
$p \leftarrow a$
$p \leftarrow b$
Let $Abd = \{a, b, \neg a, \neg b\}$, and $Obs_a = \{p\}$.

Let $Q = \{p\}$. it is easy to see that the answer sets of $\langle \Pi_1, Q \rangle$ are $\{\{p, a, \neg b\}, \{p, b, \neg a\}, \{p, a, b\}\}$, while $Filter(\Pi_1, Q) = \emptyset$.

Now consider Π_1' to be the following AnsProlog$^{\neg}$ program, where we have added four new rules to Π_1.

$p \leftarrow a$
$p \leftarrow b$
$a \leftarrow$ **not** $\neg a$
$\neg a \leftarrow$ **not** a
$b \leftarrow$ **not** $\neg b$
$\neg b \leftarrow$ **not** b

It is easy to see that the set of answer sets of $\langle \Pi_1', Q \rangle$ is $\{\{p, a, \neg b\}, \{p, b, \neg a\}, \{p, a, b\}\}$ which is same as $Filter(\Pi_1', Q)$.

Note that the set of answer sets of $\Pi_1' \cup Q$ is $\{\{p, a, \neg b\}, \{p, b, \neg a\}, \{p, a, b\}, \{p, \neg b, \neg a\}\}$ and is different from $Filter(\Pi_1', Q)$. □

Now let us compare Π_1 and Π_1' and analyze the differences. Syntactically, the difference between them is the last four rules of Π_1'. These four rules guarantee that Π_1' has at least one answer set corresponding to each potential explanation (i.e., interpretation of the abducibles). Since unlike during abductive reasoning, during filtering there is no scope to try each potential explanation so as not to miss any explanation, the answer sets of the theory should enumerate the potential explanations. This is missing in Π_1 and therefore Π_1 is not filter-abducible with respect to the above mentioned Abd and Obs. On the other hand, Π_1' satisfies this criteria and it is filter-abducible with respect to the same Abd and Obs. In the following paragraphs, we precisely state the above mentioned property as condition B. We now discuss additional conditions that may be important.

For filtering to be equivalent to abductive reasoning, each one of the answer sets obtained by filtering a theory Π with an observation Q should contain an explanation of the observation. In that case the abducibles in those answer sets consist of an explanation. For that to happen, the theory must be such that the abducibles uniquely determine the observables. If we want to avoid making any restrictions on the observables then the theory must be such that the abducibles uniquely determine the answer set. These two conditions are precisely stated below as Condition A' and Condition A, respectively.

We now formally state the above mentioned conditions.

Condition A
If M is an answer set of theory Π then M is the unique answer set of the theory $\Pi \cup (M \cap Abd)$.

Intuitively, Condition A means that the answer sets of a theory Π can be characterized by just the abducible literals in that answer set. It requires that if M is a answer set of a theory Π then M should be the only answer set of the theory $\Pi \cup (M \cap Abd)$. This is a strong condition, as in many cases $\Pi \cup (M \cap Abd)$ may have multiple answer sets. To take into account such cases, we can weaken condition A, by the following condition. But we will need to use Obs as part of our condition.

Condition A'
If M is an answer set of theory Π then:
(i) M is an answer set of $\Pi \cup (M \cap Abd)$;
(ii) all answer sets of $\Pi \cup (M \cap Abd)$ are also answer sets of Π; and
(iii) all answer sets of $\Pi \cup (M \cap Abd)$ agree on Obs; where two answer sets M_1 and M_2 of a theory are said to agree on Obs if for all $Q \in Obs$, we have $M_1 \models^* Obs$ iff $M_2 \models^* Obs$.

Condition B
For any complete subset E of Abd if $\Pi \cup E$ is consistent then there exists an answer set M of Π such that $M \cap Abd = E$

Intuitively, Condition B means that the theory Π has answer sets corresponding to each possible interpretation of the abducibles. I.e., the answer sets of the theory Π enumerate the possible explanations.

Lemma 3.9.1 Let Π be a theory satisfying Conditions A and B. Let E be any complete subset of Abd. If M is an answer set of $\Pi \cup E$, then $M \cap Abd = E$. □

Proof:
Let E be any complete subset of Abd.
From Condition B we have that there exists an answer set M' of Π such that $M' \cap Abd = E$.
But From Condition A, we have that M' is the unique answer set of $\Pi \cup E$.
Thus if M is an answer set of $\Pi \cup E$, then $M = M'$ and thus $M \cap Abd = E$. □

Theorem 3.9.2 If Π, Obs and Abd satisfy conditions A' and B then they are filter-abducible; i.e., they satisfy the following:
"for all observations $Q \in Obs$, $Filter(\Pi, Q)$ is the set of answer sets of $\langle \Pi, Q \rangle$." □

Proof:
(a) We first show that if Π, Obs and Abd satisfy conditions A' and B then all elements of $Filter(\Pi, Q)$ are answer sets of $\langle \Pi, Q \rangle$.

Let M be an element of $Filter(\Pi, Q)$
$\Rightarrow$ M is an answer set of Π and M entails Q.
From A' we have $\Pi \cup (M \cap Abd)$ is consistent and all answer sets of $\Pi \cup (M \cap Abd)$ are also answer sets of Π and agree on the observables.
Let $E = M \cap Abd$.
We then have $\Pi \cup E \models^* Q$ and $\Pi \cup E$ is consistent.
$\Rightarrow$ There exists an explanation E of Q w.r.t. Π such that M is an answer set of $\Pi \cup E$.
$\Rightarrow$ M is an answer set of $\langle \Pi, Q \rangle$.

(b) We will now show that if Π, Obs and Abd satisfy conditions A' and B then all answer sets of $\langle \Pi, Q \rangle$ are in $Filter(\Pi, Q)$.

Let M be an answer set of $\langle \Pi, Q \rangle$.
$\Rightarrow$ There exists an explanation E of Q w.r.t. Π such that M is an answer set of $\Pi \cup E$.

$\Rightarrow$ There exists a complete set of abducibles E such that $\Pi \cup E \models^{*} Q$ and $\Pi \cup E$ is consistent and M is an answer set of $\Pi \cup E$.
From Condition B we have that there exists an answer set M' of Π such that $M' \cap Abd = E$. But from Condition A' all answer sets of $\Pi \cup E$ are also answer sets of Π and agree on Q.
Thus M is an answer set of Π. Since M is also an answer set of $\Pi \cup E$ and $\Pi \cup E \models^{*} Q$, we have M entails Q.
Thus $M \in Filter(\Pi, Q)$. □

Lemma 3.9.3 If Π, *Obs* and *Abd* satisfy condition A then they also satisfies Condition A'. □

Proof: Straight forward.

Corollary 1 If an AnsProlog$^{\neg, or}$ program Π, and abducibles *Abd* satisfy conditions A and B then for any set of Obs_l in the language of Π, the theory Π, *Abd* and *Obs* are filter-abducible. □

Proof: Follows directly from Lemma 3.9.3 and Theorem 3.9.2.

The main significance of the above corollary is that by requiring the more restrictive condition A we have more flexibility with the observables.

We now give an example where we can verify filter-abducibility by verifying the above mentioned conditions.

Example 77 The following AnsProlog$^{\neg, or}$ program Π_2'
$q \ or \ r \leftarrow a$
$p \leftarrow a$
$p \leftarrow b$
$a \ or \ \neg a \leftarrow$
$b \ or \ \neg b \leftarrow$
with abducibles $\{a, b, \neg a, \neg b\}$ and $Obs_a = \{p\}$ satisfies Conditions A' and B.

This can be verified as follows. The program Π_2' has six answer sets, which are:
$\{a, b, p, q\}, \{a, b, p, r\}, \{a, \neg b, p, q\}, \{a, \neg b, p, r\}, \{\neg a, b, p\}$, and $\{\neg a, \neg b\}$. Let us refer to them as M_1, M_2, M_3, M_4, M_5 and M_6 respectively. Consider the answer set M_1. Let us verify that it satisfies Condition A'. It is easy to see that M_1 is an answer set of $\Pi_2' \cup (M_1 \cap Abd) = \Pi_2' \cup \{a, b\}$, and all answer sets of $\Pi_2' \cup \{a, b\}$ are answer sets of Π_2' and they agree on *Obs*. We can similarly verify that the other answer sets of Π_2' satisfy the conditions of A'. (Note that if we include q or r in Obs_a, condition A' will no longer be satisfied.)

In this example there are four complete subsets of *Abd*. These are: $\{a, b\}, \{a, \neg b\}, \{\neg a, b\}, \{\neg a, \neg b\}$. Consider $E = \{a, b\}$. Since $\Pi_2' \cup E$ is consistent, we need to verify that there exists an answer set M of Π_2' such that $M \cap Abd = E$. M_1 is such an answer set of Π_2'. We can similarly verify that the other complete subsets of *Abd* satisfy condition B. □

3.9.3 Sufficiency conditions for Filter abducibility of AnsProlog$^{\neg, or}$ Programs

In this subsection we will we will give some sufficiency conditions that guarantee that conditions A' and B holds in previous subsection hold. That will guarantee the filter-abducibility of AnsProlog$^{\neg, or}$ programs. The conditions involve the notion of splitting from Section 3.5 and the notion of an AnsProlog$^{\neg, or}$ program being functional from Section 1.4.4.

Proposition 62 An AnsProlog$^{\neg, or}$ program Π is filter-abducible with respect to abducibles *Abd* and observables *Obs* if
(i) Π is functional from *Abd* to Obs_l,
(ii) for all $l, \neg l \in Abd$, l *or* $\neg l \leftarrow$ is in Π, and
(iii) *Abd* is a splitting set for Π. □

Proof:
We prove this by showing that the conditions (i) – (ii) above imply the conditions A' and B which in turn guarantee filter abducibility of a theory.

(a) **Showing** $A'(i)$
We now show that the conditions (i) – (ii) implies $A'(i)$.
When Π is inconsistent this result trivially holds. Let us consider the case when Π is consistent.
Let M be a consistent answer set of Π
$\Rightarrow$ M is an answer set of Π^M.
$\Rightarrow$ M is an answer set of $\Pi^M \cup (M \cap Abd)$.
$\Rightarrow$ M is an answer set of $(T \cup (M \cap Abd))^M$.
$\Rightarrow$ M is an answer set of $\Pi \cup (M \cap Abd)$.
$\Rightarrow$ Condition $A'(i)$ holds.

(b) **Showing** $A'(ii)$
Let M be an answer set of Π. It is clear that $M \cap Abd$ is a complete set of abducible literals and is an answer set of $bot_{Abd}(T)$. Thus by Lemma 3.5.1, all answer set of $\Pi \cup (M \cap Abd)$ are answer sets of Π. Thus condition $A'(ii)$ holds.

(c) **Showing** $A'(iii)$
Since Π is functional from *Abd* to *Obs* and $M \cap Abd$ is a complete set of abducible literals, it is clear that condition $A'(iii)$ holds.

(d) **Showing B**
Let E be any arbitrary complete subset of *Abd*. From condition (ii) of the proposition, E is an answer set of $bot_{Abd}(T)$. Hence by Lemma 3.5.1, there exists an answer set M of Π, such that $M \cap Abd = E$. Thus condition B is satisfied. □

In Section 3.9.2 we show that the conditions A' and B are sufficient for filter-abducibility. We now show that they are also necessary.

3.9.4 Necessary conditions for filter-abducibility

Theorem 3.9.4 *Let* Π *be a theory, and Obs and Abd be observables and abducibles such that, for all* $Q \in Obs$, $Filter(\Pi, Q)$ *is equivalent to the set of answer sets of* $\langle \Pi, Q \rangle$. *Then* Π, *Obs and Abd satisfy the conditions B,* $A'(i), A'(ii)$ *and* $A'(iii)$. □

Proof:

(i) Suppose Π, *Obs* and *Abd* do not satisfy condition B. That means there exists an $E \subseteq Abd$, such that $\Pi \cup E$ is consistent, but there does not exist an answer set M of Π such that $M \cap Abd = E$.

Since $\Pi \cup E$ is consistent it has at least one answer set. Let M^* be an answer set of $\Pi \cup E$. Let Q be the conjunction of the literals in $M^* \cap Obs_l$. Obviously M^* is an answer set of $\langle \Pi, Q \rangle$. Since M^* is an answer set of $\Pi \cup E$, $M^* \cap Abd = E$. But then from our initial assumption, M^* can not be an answer set of Π. Hence M^* is not in $Filter(\Pi, Q)$. This contradicts the assumption in the

lemma that $Filter(\Pi, Q)$ is equivalent to the set of answer sets of $\langle \Pi, Q \rangle$. Hence Π must satisfy condition B.

(ii) Suppose Π, *Obs* and *Abd* do not satisfy condition $A'(i)$. That means there is an answer set M of Π which is not an answer set of $T \cup (M \cap Abd)$. Let $Q = M \cap Abd$. Obviously, M is in $Filter(\Pi, Q)$. We will now show that M is not an answer set of $\langle \Pi, Q \rangle$. Suppose M is an answer set of $\langle \Pi, Q \rangle$. That means there is an $E \subseteq Abd$, such that M is an answer set of $\Pi \cup E$ and $M \models^{*} Q$. But then $M \cap Abd = E$, and this contradicts our initial assumption that M is not an answer set of $T \cup (M \cap Abd)$. Hence M is not an answer set of $\langle \Pi, Q \rangle$. But this contradicts the assumption in the lemma that $Filter(\Pi, Q)$ is equivalent to the set of answer sets of $\langle \Pi, Q \rangle$. Hence Π must satisfy condition $A'(i)$.

(iii) Suppose Π, *Obs* and *Abd* do not satisfy condition $A'(ii)$. That means there is an answer set M of Π such that all answer sets of $T \cup (M \cap Abd)$ are not answer sets of Π. Let $Q = M \cap Abd$. Let M' be an answer set of $T \cup (M \cap Abd)$ which is not an answer set of Π. Obviously, M' is an answer set of $\langle \Pi, Q \rangle$. But it is not an element of $Filter(\Pi, Q)$. This contradicts the assumption in the lemma that $Filter(\Pi, Q)$ is equivalent to the set of answer sets of $\langle \Pi, Q \rangle$. Hence Π must satisfy condition $A'(ii)$.

(iv) Suppose Π, *Obs* and *Abd* do not satisfy condition $A'(iii)$. That means there is an answer set M of Π such that all answer sets of $T \cup (M \cap Abd)$ do not agree on the observables. This means $Obs_l \setminus Abd \neq \emptyset$. Let $Q = (M \cap Obs_l) \cup (M \cap Abd)$. Obviously, M is in $Filter(\Pi, Q)$. We will now show that M is not an answer set of $\langle \Pi, Q \rangle$. Suppose M is an answer set of $\langle T, Q \rangle$. That means there is a complete subset E of *Abd*, such that M is an answer set of $\Pi \cup E$ and $\Pi \cup E \models^{*} Q$. Since E is a complete subset of *Abd*, $E = M \cap Abd$. Since all answer sets of $T \cup (M \cap Abd)$ do not agree on observables $\Pi \cup E \not\models^{*} Q$. This contradicts our assumption and hence M is not an answer set of $\langle \Pi, Q \rangle$. But then, we have a contradiction to the assumption in the lemma that $Filter(\Pi, Q)$ is equivalent to the set of answer sets of $\langle \Pi, Q \rangle$. Hence Π must satisfy condition $A'(iii)$. □

We would like to mention that the known ways to satisfy condition B in logic programs are to have rules of the form l *or* $\neg l$ (or have two rules of the form $l \leftarrow$ **not** $\neg l; \neg l \leftarrow$ **not** l) for all abducible atoms l in logic programs. The former was first used in [Ino91] to relate semantics of abductive logic programs – based on the generalized stable models [KM90], and extended logic programs. Hence the necessity of condition B for filter-abducibility makes it necessary (to the best of our knowledge) to have such rules in filter-abducible logic programs.

3.9.5 Weak abductive reasoning vs filtering

Several instances of filtering used in the literature that define an intuitively meaningful entailment relation do not satisfy the conditions described earlier in this paper. In particular, when actions have non-deterministic effects (as in [Tur97]) filtering may still make intuitive sense, but our current definition of abductive reasoning is too strong to match the entailment defined through filtering. The following example illustrates our point.

Consider the AnsProlog$^{\neg, or}$ program: Π

a *or* $b \leftarrow p$
p *or* $\neg p$

where $Abd = \{p, \neg p\}$, and $Obs_a = \{a, b\}$. Suppose we observe a. Using filtering we would be able to conclude – in this case, intuitively explain our observation by – p. (I.e., p will be true all answer sets of $Filter(\Pi, \{a\})$.) But the current definition of abductive reasoning is too strong to explain

this observation by p. (Note that the above theory will violate our condition $A'(iii)$.) This rigidity of abductive reasoning has been noticed earlier and several suggestions for weaker versions have been made; for example in [Gel90, Rei87, Sha97]. In this section we define a weaker notion of abductive reasoning and show that it is equivalent to filtering under less restrictive conditions than given in the earlier sections; in particular, we no longer need condition $A'(iii)$. As a result we can also weaken the sufficiency conditions in Propositions 62. We now formally define *weak abductive entailment* and state theorems and propositions similar to the ones in the previous sections.

Definition 61 (Weak abductive Entailment) Let Π be an AnsProlog$^{\neg, or}$ program, and Q be an observation.
(i) M is a w-answer set (or weak-answer set) of $\langle \Pi, Q\rangle$ if there exists a complete subset E of abducibles such that M is an answer set of $\Pi \cup E$ and $M \not\models^{*} Q$.
(ii) For any formula f, $\langle \Pi, Q\rangle \models_{wabd} f$ if f is true in all w-answer sets of $\langle T, Q\rangle$. □

Definition 62 (Weak-filter-abducible) An AnsProlog$^{\neg, or}$ program Π, a set Abd, and a set Obs are said to be *weak-filter-abducible* if for all possible observations $Q \in Obs$; $Filter(\Pi, Q)$ is the set of w-answer sets of $\langle T, Q\rangle$. □

Theorem 3.9.5 (Sufficiency) Let Π be an AnsProlog$^{\neg, or}$ program, and Obs and Abd be observables. If Π, Obs and Abd satisfy conditions $A'(i)$, $A'(ii)$ and B then they are weak-filter-abducible. □

Theorem 3.9.6 (Necessity) Let Π be an AnsProlog$^{\neg, or}$ program, and Obs and Abd be observables and abducibles such that, for all $Q \in Obs$, $Filter(\Pi, Q)$ is equivalent to the set of weak-answer sets of $\langle \Pi, Q\rangle$. Then Π, Obs and Abd satisfy the conditions B, $A'(i), and A'(ii)$. □

Proposition 63 An AnsProlog$^{\neg, or}$ program Π is weak-filter-abducible with respect to abducibles Abd and observables Obs if
(i) for all $l, \neg l \in Abd$, l *or* $\neg l \leftarrow$ is in Π, and
(ii) Abd is a splitting set for Π. □

Exercise 10 Formulate a broader theory of abductive reasoning in AnsProlog$^{\neg, or}$ programs by expanding the notion of explanations, to allow incomplete subset of the abducibles. Such a notion of abductive reasoning in default theories is defined in [EGL97].

Hint: To do abductive reasoning using filtering in this case, the theory should be again such that its answer sets enumerate the various possible explanations. Suppose $Abd = \{p, \neg p\}$. Now that we intend to allow explanations to be incomplete subset of the abducibles, the set of possible explanations will $\{\{p\}, \{\neg p\}, \{\}\}$. Since answer sets are minimal sets, there does not exist a logic program whose answer sets will be these three possible explanations. The minimality condition will eliminate the first two, in presence of the third. One way to overcome this would be to use a special fluent u_p (meaning *uncommitted about* p) to represent the third explanation. Then we can have AnsProlog$^{\neg, or}$ programming rules of the form:

$$p \ or \ \neg p \ or \ u\_p \leftarrow$$

□

3.10 Equivalence of programs and semantics preserving transformations

In this section we explore the notions of when two AnsProlog* program are equivalent. There are several notions of equivalence, starting from the simple notion that they have the same answer sets to the more rigorous notion that they encode the same function. Having such notions of equivalence is very useful. We can use that to transform a program to an equivalent program that eliminates certain syntactic features, so that they are suitable for a more restricted interpreter. In this respect we discuss conditions when transformations that eliminate the **not** operator, rules with empty head, and disjunctions (*or*) respectively, result in an equivalent program.

3.10.1 Fold/Unfold transformations

Among the earliest transformations are notion of folding and unfolding. The following example illustrates both of these transformations.

Consider a program Π containing the following three rules and no other rules with p and q in its head.

$r_1 : p \leftarrow q, r.$
$r_2 : q \leftarrow s.$
$r_3 : q \leftarrow t.$

The first rule r_1 can be *unfolded* with respect to the other two rules r_2 and r_3 and we will have the following two rules:

$r_4 : p \leftarrow s, r.$
$r_5 : p \leftarrow t, r.$

which can replace the rule r_1 in Π without affecting the meaning of the program.

Similarly, consider a program Π' containing of $r_2 - r_5$. In that case we can replace r_4 and r_5 in Π' by the single rule r_1 without affecting the meaning of Π'. Replacing r_4 and r_5 by r_1 in the presence of r_2 corresponds to *folding*.

To formalize the above intuitive notions of folding and unfolding with respect to rules with variables, we need the notion of substitutions, unifiers, and most general unifiers.

Substitutions and Unifiers

A substitution is a finite mapping from variables to terms, and is written as

$\theta = \{X_1/t_1, \ldots, X_n/t_n\}$

The notation implies that $X_1, \ldots, X_n$ are distinct, and we assume that for $i = 1 \ldots n$, X_i is different from t_i. Substitutions operate on terms, a sequence of gen-literals or a rule as a whole. Substitutions can be composed. Given substitutions $\theta = \{X_1/t_1, \ldots, X_n/t_n\}$ and $\eta = \{Y_1/s_1, \ldots, Y_m/s_m\}$ their composition $\theta\eta$ is defined by removing from the set $\theta = \{X_1/t_1\eta, \ldots, X_n/t_n\eta, Y_1/s_1, \ldots, Y_m/s_m\}$ those pairs $X_i/t_i\eta$, where $X_i \equiv t_i\eta$ as well as those pairs Y_i/s_i for which $Y_i \in \{X_1, \ldots, X_n\}$.

For example if $\theta = \{X/3, Y/f(X, a)\}$ and $\eta = \{X/4, Y/5, Z/a\}$ then $\theta\eta = \{X/3, Y/f(4, a), Z/a\}$.

A substitution θ is said to be *more general* than a substitution η, if for some substitution γ we have $\eta = \theta\gamma$.

We say θ is an *unifier* for two atoms A and B if $A\theta \equiv B\theta$. A unifier θ of two atoms A and B is said to be the *most general unifier (or mgu)* of A and B if it is more general than any other unifier of A and B. Robinson in [Rob65] showed that if two atoms are unifiable then they have a most general unifier. The notion of unifiers and mgu is directly extendable to gen-literals, terms and rules.

Folding and Unfolding

Definition 63 *(Initial program)* An *initial program* Π_0 is an AnsProlog$^\neg$ program satisfying the following condition:

Π_0 can be partitioned to two programs Π_{new} and Π_{old} such that the set of predicates in the head of the rules of Π_{new} – referred to as the *new* predicates, neither appear in the body of the rules in Π_{new} nor appear in Π_{old}. The set of predicates in the head of the rules of Π_{old} are referred to as the *old* predicates □

Definition 64 *(Unfolding)* Let Π_i be an AnsProlog$^\neg$ program and C be a rule in Π_i of the form: $H \leftarrow A, L.$, where A is a literal, and L is a sequence of gen-literals. Suppose that $C_1, \ldots, C_k$ are all the rules in Π_i such that C_j is of the form $A_j \leftarrow K_j$, where K_j is a sequence of gen-literals, and A_j is unifiable with A, by an mgu θ_j for each j $(1 \leq j \leq k)$. Then $\Pi_{i+1} = (\Pi_i \setminus \{C\}) \cup \{H\theta_j \leftarrow K_j\theta_j, L\theta_j \ : \ 1 \leq j \leq k\}$, is a program obtained by unfolding Π, whose selected atom is A. C is called the *unfolded rule* and $C_1, \ldots, C_k$ are called the *unfolding rules.* □

We now discuss two different notions of folding: TSS-folding is due to Tamaki, Seki and Satoh and is defined in [TS84, Sek91, Sek93]; and MGS-folding is due to Maher, Gardner and Shepherdson and is defined [GS91, Mah87, Mah90, Mah93b].

Definition 65 *(TSS-Folding)* Let Π_i be an AnsProlog$^\neg$ program, C be a rule in Π_{new} (not necessarily in Π_i) of the form $A \leftarrow K, L.$, where K and L are sequences of gen-literals, and D be a rule of the form $B \leftarrow K'$, where K' is a sequence of gen-literal. Suppose there exists a substitution θ satisfying the following conditions:

1. $K'\theta = K$.
2. Let $X_1, \ldots, X_j, \ldots, X_m$ be variables that appear only in the body K' of D but not in B. Then, each $X_j\theta$ is a variable in C such that it appears in none of A, L and $B\theta$. Furthermore, $X_j\theta \neq X_{j'}\theta$ if $j \neq j'$.
3. D is the only clause in Π_i whose head is unifiable with $B\theta$.
4. Either the predicate of A is an old predicate, or C is the result of applying unfolding at least once to a clause in Π_0.

Then $\Pi_{i+1} = (\Pi_i \setminus C) \cup \{A \leftarrow B\theta, L.\}$ is called a TSS-folding of Π_i; C is called the folded rule and D is called the folding rule. □

Definition 66 *(MGS-Folding)* Let Π_i be an AnsProlog$^\neg$ program, C be a rule in Π_i of the form $A \leftarrow K, L.$, where K and L are sequences of gen-literals, and D be a rule of the form $B \leftarrow K'$, where K' is a sequence of gen-literal. Suppose there exists a substitution θ satisfying the following conditions:

1. $K'\theta = K$.

2. Let $X_1, \ldots, X_j, \ldots, X_m$ be variables that appear only in the body K' of D but not in B. Then, each $X_j\theta$ is a variable in C such that it appears in none of A, L and $B\theta$. Furthermore, $X_j\theta \neq X_{j'}\theta$ if $j \neq j'$.

3. D is the only clause in Π_i whose head is unifiable with $B\theta$.

4. C is different from D.

Then $\Pi_{i+1} = (\Pi_i \setminus C) \cup \{A \leftarrow B\theta, L.\}$ is called a MGS-folding of Π_i; C is called the folded rule and D is called the folding rule. □

Definition 67 Let Π_0 be an initial AnsProlog$^\neg$ program and Π_{i+1} $(i \geq 0)$ be obtained from Π_i by applying either unfolding, TSS-folding, or MGS-folding. Then the sequence of programs $\Pi_0, \ldots, \Pi_N$ is called a transformation sequence starting from Π_0. □

Proposition 64 *[AD95]* The answer sets of any program Π_i in a transformation sequence starting from an initial AnsProlog$^\neg$ program Π_0 are the same as the answer sets of Π_0. □

3.10.2 Replacing disjunctions in the head of rules

In this section we discuss two conditions which allow us to replace disjunctions in the head of rules by constraints, and AnsProlog$^\neg$ rules respectively. The motivation behind replacing disjunctions is that some interpreters either only allow AnsProlog$^\neg$ programs or are optimized for such programs. Moreover, as we will show in Chapter 6, in general AnsProlog$^{\neg, or}$ programs have a higher expressibility and complexity than AnsProlog$^\neg$ programs. Thus an interpreter for AnsProlog$^{\neg, or}$ programs may not be efficient for AnsProlog$^\neg$ programs, and therefore if efficiency is a concern we should eliminate the disjunctions if possible. Also, even when efficiency is not a concern, it is useful to find out while using disjunctions, whether the disjunction is mandatory – i.e., the problem needs the higher expressibility, or whether it is for convenience and ease of representation.

We now present the two transformations. Given a rule r of the form (1.2.2)

- $constraint(r)$ denotes the constraint:

  $\leftarrow$ **not** $L_0, \ldots,$ **not** $L_k, L_{k+1}, \ldots, L_m,$ **not** $L_{m+1}, \ldots,$ **not** L_n.

- and $disj\_to\_normal(r)$ denote the following $k+1$ rules:

  $L_i \leftarrow$ **not** $L_0, \ldots$ **not** $L_{i-1},$ **not** $L_{i+1}, \ldots,$ **not** $L_k, L_{k+1}, \ldots, L_m,$ **not** $L_{m+1}, \ldots,$ **not** L_n.

  where $0 \leq i \leq k$.

We now present the result about when AnsProlog$^{\neg, or}$ rules in a program can be replaced by AnsProlog$^\neg$ rules, while keeping the answer sets unchanged.

Theorem 3.10.1 *[BED92]* Let Π be an AnsProlog$^{\neg, or}$ program. Let Π' be the program obtained by replacing each rule r in Π with disjunctions in its head by the set of rules $disj\_to\_normal(r)$. If Π is head cycle free then Π and Π' have the same consistent answer sets. □

Example 78 Consider the following AnsProlog$^{\neg, or}$ program Π.

a *or* b *or* $c \leftarrow$.

It has three answer sets, $\{a\}, \{b\}$ and $\{c\}$. It is easy to check that Π is a head-cycle-free program. Let us consider the AnsProlog$^{\neg}$ obtained by replacing each rule r in Π by the set $disj\_to\_normal(r)$. We then obtain the following program.

$a \leftarrow$ **not** b, **not** c.
$b \leftarrow$ **not** c, **not** a.
$c \leftarrow$ **not** a, **not** b.

It is easy to check that it also has only three answer sets, the same as the answer sets of Π. □

Also recall that the program Π from Example 61 is not head-cycle-free and thus the above theorem is not applicable to it. This conforms with our discussion in Example 29 where we show that if we replace the rules r from Π with disjunctions in their head by the set of rules $disj\_to\_normal(r)$ then we do not have a program with the same answer sets as before.

We now present the result about when an AnsProlog$^{\neg, or}$ rule can be replaced by a constraint while keeping the answer sets unchanged.

Theorem 3.10.2 *[LT95]* Let Π and Π' be AnsProlog$^{\neg, or}$ programs, and let Γ be a saturated set of literals such that $Lit \setminus \Gamma$ is a signing for the program $\Pi \cup \Pi'$. If every answer set for Π is complete in Γ and if the head of every rule in Π' is a subset of Γ, then programs $\Pi \cup \Pi'$ and $\Pi \cup constraint(\Pi')$ have the same consistent answer sets. □

3.10.3 From AnsProlog to AnsProlog$^{or, \text{-}\mathbf{not}}$ and constraints

Any AnsProlog program P can be transformed to an AnsProlog$^{or, \text{-}\mathbf{not}}$ program P' and a set of constraints C such that the answer sets of P can be obtained by filtering the minimal models of P' by the constraints IC_P. The transformation is given as follows:

1. The AnsProlog$^{or, \text{-}\mathbf{not}}$ program P' consists of the following:

   (a) Replace each rule of the form $A_0 \leftarrow A_1, \ldots, A_m, \mathbf{not}\ A_{m+1}, \ldots, \mathbf{not}\ A_n$ in P by the following rule:

   A_0 *or* A'_{m+1} *or* $A'_n \leftarrow A_1, \ldots, A_m$.

   where if $A_i = p(\bar{X})$ then A'_i is obtained by replacing p by a new predicate symbol p' of the same arity as p.

   (b) For each predicate symbol p in P, the rule

   $p'(\bar{X}) \leftarrow p(\bar{X})$.

   is in P'.

2. The constraints IC_P consist of the following rules for each symbol p in P, the rule

   $\leftarrow p'(\bar{X}), \mathbf{not}\ p(\bar{X})$.

Theorem 3.10.3 *[FLMS93]* Let P be an AnsProlog program and P' and IC_P be as obtained above.

(i) M is an answer set of P iff $M' = M \cup \{p'(\bar{t}) : p(\bar{t}) \in M\}$ is an answer set of $P' \cup IC_P$.

(ii) M is an answer set of P iff $M' = M \cup \{p'(\bar{t}) : p(\bar{t}) \in M\}$ is a minimal model of P' and M' satisfies C. □

The following example illustrates the above.

Example 79 Consider the following AnsProlog program P:

$a \leftarrow$ **not** b.

The program P' consists of

$a \text{ or } b' \leftarrow$.
$a' \leftarrow a$.
$b' \leftarrow b$.

and IC_P consists of

$\leftarrow a', \textbf{not } a$.
$\leftarrow b', \textbf{not } b$.

There are two answer sets of P': $\{a, a'\}$ and $\{b'\}$; out of which the second one violates IC_P, while the first one does not.

Indeed $\{a\}$ is the answer set of P. □

Part (ii) of the above theorem is useful in computing answer sets of AnsProlog programs when there are methods available to compute minimal models of programs without **not** . In the following subsection we describe one such method.

3.10.4 AnsProlog and mixed integer programming

It is well known that models of propositional theories can be obtained by transforming the propositional theory to a set of integer linear constraints involving binary variables and solving these constraints. The transformation is quite straightforward and given as follows:

For each proposition p, we have a binary variable X_p that can only take values 1 or 0. Without loss of generality, we assume that our propositional theory is of the form

$(l_{11} \vee \ldots l_{1i_1}) \wedge \ldots \wedge (l_{m1} \vee \ldots l_{mi_m})$

where l_{jk}'s are propositional literals. For a negative literal $l = \neg p$, by X_l we denote $(1 - X_p)$ and for a positive literal $l = p$, by X_l we denote X_p.

The above propositional theory is then transformed to the following set of integer linear constraints:

$$X_{l_{11}} + \ldots + X_{l_{1i_1}} \geq 1$$
$$\vdots$$
$$X_{l_{m1}} + \ldots + X_{l_{mi_m}} \geq 1$$

The models of the propositional theory then correspond to the solution of the above constraints together with the restriction that each variable can only take the value 0 or 1. The later can be

expressed as $0 \leq X_p \leq 1$, for all propositions p. Given a propositional theory P, we refer to the above constraints as $ilp(P)$.

Now if we have the following minimization criteria

$$min \quad \Sigma_p \text{ is a proposition } X_p$$

then the solution of the resulting integer linear program (ILP) will correspond to the *cardinality minimal models* (i.e., minimal models based on the ordering $M_1 \leq M_2$ iff $|M_1| \leq |M_2|$) of the propositional theory.

The minimal models of a propositional theory can then be obtained by using an iterative procedure which computes one of the cardinality minimal models and updates the theory so as to eliminate that model in the next iteration. The algorithm is formally described as follows:

Algorithm 2 *Computing minimal models*
(1) $min\_model\_set = \emptyset$ and $Constarints = \emptyset$.
(2) Solve the ILP
$min \quad \Sigma_p$ is a proposition X_p
subject to $ilp(P) \cup Constraints$.
(3) If no (optimal) solution can be found, halt and return min_model_set as the set of minimal models of P.
(4) Otherwise, let M be the model corresponding to the optimal solution found in Step (2). Add M to min_model_set.
(5) Add the constraint $\Sigma_{A \in M} X_A \leq (k-1)$ to $Constarints$, where k is the cardinality of M. Go to Step (2). □

Example 80 Consider the propositional theory P given as $(a \vee b) \wedge (a \vee c)$. The constraints $ilp(P)$ then consists of the following:

$0 \leq X_a \leq 1$
$0 \leq X_b \leq 1$
$0 \leq X_c \leq 1$
$X_a + X_b \geq 1$
$X_a + X_c \geq 1$

Initially $Constraints = \emptyset$. Solving the ILP
$min \;\; X_a + X_b + X_c$
subject to $ilp(P) \cup Constraints$ will give us the solution $X_a = 1$ and $X_b = X_c = 0$. The corresponding model M is $\{a\}$. After adding M to min_model_set in Step (4) of the above algorithm in step (5) we will add $X_a \leq 0$ to $Constraints$. Now solving the ILP
$min \;\; X_a + X_b + X_c$
subject to $ilp(P) \cup Constraints$ will give us the solution $X_a = 0$ and $X_b = X_c = 1$.

The corresponding model M' is $\{b, c\}$. After adding M' to min_model_set in Step (4) of the above algorithm in step (5) we will add $X_b + X_c \leq 1$ to $Constraints$. Now solving the ILP
$min \;\; X_a + X_b + X_c$
subject to $ilp(P) \cup Constraints$ will not give us any solution. □

The above method for computing minimal models of propositional theories can be used for computing minimal models of ground AnsProlog^{or} programs. Using part (ii) of Theorem 3.10.3 we

can then compute answer sets of AnsProlog programs by transforming them to an AnsProlog^{or} program and a set of constraints; computing the minimal models of the AnsProlog^{or} program and checking if the minimal models violate the constraints. Alternative ways to compute answer sets of AnsProlog programs is to compute the minimal models of the program or its completion using the above mentioned technique and verifying if they are answer sets.

For AnsProlog$^{-\mathbf{not}}$ programs instead of solving the ILP, we can consider real values of the variables and will still obtain the unique answer set. More formally,

Theorem 3.10.4 Let P be an AnsProlog$^{-\mathbf{not}}$ program. Then:
(i) There is exactly one integer solution of $ilp(P)$ that minimizes Σ_p is a proposition X_p. And this solution corresponds to the unique minimal model of P.
(ii) Suppose the ILP constraints are relaxed such that the variables range over all real values in [0,1] instead of the integers 0, 1. Then there is exactly one solution to this problem, which is identical to the initial integer solution. □

Since it is well known that solving linear programming problems over real numbers is significantly easier than solving them over integers, the above theorem is significant.

3.10.5 Strongly equivalent AnsProlog* programs and the logic of here-and-there

The simple notion of equivalence that we discussed so far is not adequate if we were to treat AnsProlog* programs as functions. For example the two programs $\{p(X) \leftarrow q(X).; r(X) \leftarrow s(X).\}$ and $\{r(X) \leftarrow s(X).\}$ have the same answer sets, but definitely do not encode the same information. That is because, the meaning of the two programs together with a new fact $q(a)$ would be different. Similarly, the programs $\{p(X) \leftarrow q(X).; q(a) \leftarrow .; r(b) \leftarrow .\}$ and $\{p(a) \leftarrow .; q(a) \leftarrow .; r(b) \leftarrow .\}$ also have the same answer sets, but rarely will we consider replacing one with the other. The only time the simpler notion of equivalence is useful is when we are computing answer sets of the programs. If we were to treat programs as representing knowledge, or as encoding functions, we need a stronger notion of equivalence. Such a notion is defined below.

Definition 68 Two AnsProlog* programs Π_1 and Π_2 are said to be *strongly equivalent* if for every Π, $\Pi_1 \cup \Pi$ and $\Pi_2 \cup \Pi$ have the same answer sets. □

One of the reason behind the difference between the notions of equivalence and strong equivalence of AnsProlog* programs is due to the non-monotonicity of AnsProlog*. In a monotonic language if T_1 and T_2 have the same 'models' then for any T, $T_1 \cup T$ and $T_2 \cup T$ will have the same models. This implies that the strong equivalence of AnsProlog* programs can be inferred by transforming these programs to theories in a *suitable* monotonic logic and showing the equivalence of those theories.

In the rest of this sub-section we will pursue this with respect to propositional AnsProlog*. In this quest we rule out the possibility of using classical proposition logic with the straight forward transformation of replacing **not** by $\neg$ and $\leftarrow$ by $\Leftarrow$ as being the *suitable* approach. This is because the two programs $\{a \leftarrow \mathbf{not}\ b.\}$ and $\{b \leftarrow \mathbf{not}\ a\}$ are transformed to equivalent propositional theories and yet the program themselves are not even equivalent.

A logic that serves our purpose is the "logic of here-and-there" (HT), a stronger subsystem of classical propositional logic. We now describe this logic.

In HT formulas are built from propositional atoms and the 0-place connective $\bot$, using the binary connectives $\wedge$, $\vee$ and $\rightarrow$. The symbol $\top$ is written as a short-hand for $\bot \rightarrow \bot$, and $\neg F$ for $F \rightarrow \bot$. A theory in HT is a set of such formulas. A theory in HT can also be considered as a theory in classical propositional logic where the satisfaction relation between an interpretation I – thought of as a set of atoms, and a formula F can be defined as follows:

- If p is an atom, then $I \models p$ if $p \in I$,
- $I \not\models \bot$,
- $I \models F \wedge G$ if $I \models F$ and $I \models G$,
- $I \models F \vee G$ if $I \models F$ or $I \models G$, and
- $I \models F \rightarrow G$ if $I \not\models F$ or $I \models G$.

In HT interpretations, referred to as HT-interpretations, HT-interpretations are a pair (I^H, I^T) of sets of atoms such that $I^H \subseteq I^T$. Intuitively, H and T correspond to two worlds: *here* and *there*, and I^H and I^T correspond to the interpretation in the worlds H and T respectively. The satisfiability relation between HT-interpretation and formulas are defined in terms of the satisfiability relation between triplets (I^H, I^T, w) – where $w \in \{H, T\}$ is one of the two worlds, and formulas. We now define this relation.

- For any atom F, and $w \in \{H, T\}$, $(I^H, I^T, w) \models F$ if $F \in I^w$,
- $(I^H, I^T, w) \not\models \bot$,
- $(I^H, I^T, w) \models F \wedge G$ if $(I^H, I^T, w) \models F$ and $(I^H, I^T, w) \models G$,
- $(I^H, I^T, w) \models F \vee G$ if $(I^H, I^T, w) \models F$ or $(I^H, I^T, w) \models G$,
- $(I^H, I^T, H) \models F \rightarrow G$ if
  - $(I^H, I^T, H) \not\models F$ or $(I^H, I^T, H) \models G$, and
  - $(I^H, I^T, T) \not\models F$ or $(I^H, I^T, T) \models G$, and
- $(I^H, I^T, T) \models F \rightarrow G$ if $(I^H, I^T, T) \not\models F$ or $(I^H, I^T, T) \models G$.

An HT-interpretation (I^H, I^T) is said to satisfy a formula F if (I^H, I^T, H) satisfies F. An HT-model of a theory Γ is an HT-interpretation that satisfies every formula in Γ. A formula F is a consequence of a set Γ of formulas in logic HT, denoted by $\Gamma \models_{HT} F$, if every HT-model of Γ satisfies F. Two theories are said to be *HT-equivalent* (or equivalent in the logic of here-and-there) if they have the same HT-models.

Note that for a theory Γ, M is a model of Γ iff (M, M) is an HT-model of Γ. Hence, every consequence of Γ in the logic of here-and-there is a consequence of Γ in the sense of classical propositional logic. The converse is not true though.

We now present a deduction system for the logic of HT using the symbol $\vdash_{HT}$, where $\Gamma \vdash_{HT} F$ means that ϕ can be deduced from Γ in the deduction system of HT; when Γ is the empty set then it is written as $\vdash_{HT} F$.

- (**As**) If $F \in \Gamma$ then $\Gamma \vdash_{HT} F$.
- (**EFQ**) If $\Gamma \vdash_{HT} \bot$ then $\Gamma \vdash_{HT} F$.
- ($\neg$**-I**) If $\Gamma, H \vdash_{HT} \bot$ then $\Gamma \vdash_{HT} \neg H$.
- ($\neg$**-E**) If $\Gamma \vdash_{HT} F$ and $\Gamma \vdash_{HT} \neg F$ then $\Gamma \vdash_{HT} \bot$.
- ($\wedge$**-I**) If $\Gamma_1 \vdash_{HT} F$ and $\Gamma_2 \vdash_{HT} \neg G$ then $\Gamma_1, \Gamma_2 \vdash_{HT} F \wedge G$.
- ($\wedge$**-E**) If $\Gamma \vdash_{HT} F \wedge G$ then $\Gamma \vdash_{HT} F$; if $\Gamma \vdash_{HT} F \wedge G$ then $\Gamma \vdash_{HT} G$.
- ($\vee$**-I**) If $\Gamma \vdash_{HT} F$ then $\Gamma \vdash_{HT} F \vee G$; if $\Gamma \vdash_{HT} G$ then $\Gamma \vdash_{HT} F \vee G$.
- ($\vee$**-E**) If $\Gamma_1 \vdash_{HT} F \vee G$; $\Gamma_2, F \vdash_{HT} H$; and $\Gamma_3, G \vdash_{HT} H$ then $\Gamma_1, \Gamma_2, \Gamma_3 \vdash_{HT} H$.
- ($\rightarrow$**-I**) If $\Gamma, H \vdash_{HT} F$ then $\Gamma \vdash_{HT} (H \rightarrow F)$.
- ($\rightarrow$**-E**) If $\Gamma_1 \vdash_{HT} (H \rightarrow F)$ and $\Gamma_2 \vdash_{HT} H$ then $\Gamma_1, \Gamma_2 \vdash_{HT} F$.
- (**HTA**) $\vdash_{HT} F \vee (F \rightarrow G) \vee \neg G$.

Theorem 3.10.5 $\Gamma \models_{HT} F$ iff $\Gamma \vdash_{HT} F$. □

Note that the deduction system of classical propositional logic consists of all the above deduction rules – with the notation $\vdash_{HT}$ replaced by $\vdash$, except that (HTA) is replaced by $\vdash F \vee \neg F$, the law of excluded middle; and a natural system of intutionistic logic is obtained by removing (HTA) from the deduction system for HT. Moreover, in both HT and intutionistic logic F and $\neg\neg F$ are not equivalent, while they are equivalent is classical propositional logic. We now list some consequences and equivalences that can be derived in the deduction system for HT and that are useful for our purpose of showing strong equivalence of AnsProlog^{or} programs.

Proposition 65 (i) $\vdash_{HT} \neg H \vee \neg\neg H$.

(ii) $\neg(F \vee G)$ and $\neg F \wedge \neg G$ are HT-equivalent.

(iii) $\neg(F \wedge G)$ and $\neg F \vee \neg G$ are HT-equivalent.

(iv) $\neg F \vee G$ and $\neg\neg F \rightarrow G$ are HT-equivalent. □

Proof:
To show the HT-equivalences of two theories T and T' we can use $\vdash_{HT}$ and show that for all formulas Fin T, $T' \vdash_{HT} F$, and for all formulas G in T', $T \vdash_{HT} G$. The rest of the proof is left as an exercise. □

Note that the classical version of (ii) and (iii) are the well known De Morgan's laws.

Theorem 3.10.6 *[LPV00]* Two propositional AnsProlog^{or} programs are strongly equivalent iff the theory obtained by replacing *or* by $\vee$, **not** by $\neg$ and $Head \leftarrow Body$ by $Body \rightarrow Head$ are equivalent in the logic of here-and-there. □

We now use the above theorem to show strong equivalence of some AnsProlog^{or} programs.

Consider the program Π consisting of the following rules:

p *or* $q \leftarrow$.
$\bot \leftarrow p, q$.

and the program Π' consisting of the following rules:

$p \leftarrow$ **not** q.
$q \leftarrow$ **not** p.
$\bot \leftarrow p, q$.

The HT-theory obtained from Π and Π' are $\{p \vee q, \neg(p \wedge q)\}$ and $\{\neg p \rightarrow q, \neg q \rightarrow p, \neg(p \wedge q)\}$, respectively, which can be re-written using Proposition 65 as $T = \{p \vee q, \neg p \vee \neg q\}$ and $T' = \{\neg p \rightarrow q, \neg q \rightarrow p, \neg p \vee \neg q\}$. We now argue that these two theories are HT-equivalent.

To show T and T' are HT-equivalent. We show that (i) For all formulas F in T, $T' \vdash_{HT} F$, and (ii)) For all formulas G in T', $T \vdash_{HT} G$.

- (i) There are two formulas in T. $T' \vdash_{HT} \neg p \vee \neg q$ because of (As). We will now show that $T' \vdash_{HT} p \vee q$.
  - (a) $T' \vdash_{HT} \neg p \vee \neg q$ Using (As).
  - (b) $\{\neg p\} \vdash_{HT} \neg p$ Using (As).
  - (c) $T' \vdash_{HT} \neg p \rightarrow q$ Using (As).
  - (d) $T', \neg p \vdash_{HT} q$ Using (b), (c) and $\rightarrow$-E.
  - (e) $T', \neg p \vdash_{HT} p \vee q$ Using (d) and $\vee$-I.
  - (f) $\{\neg q\} \vdash_{HT} \neg q$ Using (As).
  - (g) $T' \vdash_{HT} \neg q \rightarrow p$ Using (As).
  - (h) $T', \neg q \vdash_{HT} p$ Using (f), (g) and $\rightarrow$-E.
  - (i) $T', \neg q \vdash_{HT} p \vee q$ Using (h) and $\vee$-I.
  - (j) $T' \vdash_{HT} p \vee q$ Using (a), (e), (i), and $\vee$-E, by having $\Gamma_1 = \Gamma_2 = \Gamma_3 = T'$.
- (ii) There are three formulas in T'. $T \vdash_{HT} \neg p \vee \neg q$ because of (As). We will now show that $T \vdash_{HT} \neg p \rightarrow q$.
  - (k) $T \vdash_{HT} p \vee q$ Using (As).
  - (l) $\neg p, p \vdash_{HT} q$ Using (EFQ) and $\neg$-E.
  - (m) $T, q \vdash_{HT} q$ Using (As).
  - (n) $T, \neg p \vdash_{HT} q$. Using (k), (l), (m), and $\vee$-E, by having $\Gamma_1 = \Gamma_3 = T$, and $\Gamma_2 = \{\neg p\}$.
  - (o) $T \vdash_{HT} \neg p \rightarrow q$. From (n) and $\rightarrow$-I.

  $T \vdash_{HT} \neg q \rightarrow p$, can be shown in a similar way.

3.11 Notes and references

Among the early results about coherence and categoricity of AnsProlog programs was the result about stratified and locally stratified programs. Stratification was defined and its properties were studied in [CH85, ABW88]. The notion of stratification was extended to local stratification in [Prz88b, Prz88a, Prz89b] and further extended to weak stratification in [PP90]. Acyclic programs were originally introduced by Cavedon [Cav89] and were called locally hierarchical programs. Its properties was extensively studied explored in [AB90, AB91], the properties of well-moded programs were explored in [AP91, AP94]. Existence of answer sets and various sub-classes of AnsProlog programs and their properties were further studied in [Fag90, CF90, Dun92, Sat90]. The notion of call-consistency was defined in [Kun89, Sat87]. The papers [Fag90, Fag94] present a nice summary of these results. Fages in [Fag94] used the term 'positive-order-consistency' which was later referred to as 'tight' in [Lif96] and the notion was further extended in [BEL00]. The notion of stable programs was originally defined in [Str93], and extended by McCain and Turner in [MT94b] where they discuss language independence and language tolerance. The ideas of language independence and language tolerance had their origin in the earlier deductive database research on domain independent databases [TS88]. Head-cycle-free programs and their properties were studied in [BED92]. One of the main properties studied in that paper was about when disjunctions can be eliminated. Another result on eliminating disjunctions was presented in [LT95].

Marek and Subrahmanian's result [MS89] relating answer sets and AnsProlog program rules was one of the most widely used result to analyze AnsProlog programs. Its extension to AnsProlog$^{\neg, or}$ programs is reported in [BG94]. Restricted monotonicity was first studied in [Lif93b]. The notion of signing was introduced in [Kun89] and its properties were studied in [Dun92, Tur93, Tur94]. The notion of splitting was introduced in [LT94]. A similar notion was independently developed in [EGM97].

The notion of conservative extension was first studied in [GP91] and further generalized in [GP96]. The notion of interpolation of AnsProlog programs was presented in [BGK93, BGK98]. Conservative extension and interpolation were both generalized as operations on functional specifications and results about realizing functional specifications using AnsProlog* programs were presented in [GG97, GG99].

The notion of filter-abducibility and assimilation of observations was explored in [Bar00]. An alternative analysis of relation between disjunction and abduction is presented in [LT95].

Various fold/Unfold transformations were described in [GS91, Mah87, Mah90, Mah93b, Sek91, Sek93, TS84]. Their properties with respect to AnsProlog$^{\neg}$ programs were compiled and extended in [AD95]. The paper [Mah88] discusses several different kinds of equivalences between logic programs. The paper [FLMS93] presents the transformation from AnsProlog programs to AnsProlog$^{or, \textbf{not}}$ programs and constraints. The use of integer linear programming and mixed integer programming in computing answer sets of AnsProlog programs is discussed in [BNNS94]. Conditions for strong equivalence of AnsProlog* programs were presented in [LPV00].

Chapter 4

Declarative problem solving and reasoning in AnsProlog*

In this chapter we formulate several knowledge representation and problem solving domains using AnsProlog*. Our focus in this chapter is on program development. We start with three well known problems from the literature of constraint satisfaction, and automated reasoning: placing queens in a chess board, determining who owns the zebra, and finding tile covering in a mutilated chess board. We show several encodings of these problems using AnsProlog*. We then discuss a general methodology for representing constraint satisfaction problems (CSPs) and show how to extend it to dynamic CSPs. We then present encodings of several combinatorial graph problems such as k-colarability, Hamiltonian circuit, and K-clique. After discussing these problem solving examples, we present a general methodology of reasoning with prioritized defaults, and show how reasoning with inheritance hierarchies is a special case of this.

4.1 Three well known problem solving tasks

A well known methodology for declarative problem solving is the *generate and test* methodology where by possible solutions to the problem are generated and non-solutions are eliminated by testing. This is similar to the common way of showing a problem is in the class NP, where it is shown that after the non-deterministic choice the testing can be done in polynomial time. The 'generate' part in an AnsProlog* formulation of a problem solving task is achieved by enumerating the possibilities, and the 'test' part is achieved by having constraints that eliminate the possibilities that violate the test conditions. Thus the answer sets of the resulting program correspond to solutions of the given problem. We refer to the AnsProlog* implementation of generate and test as the *enumerate and eliminate* approach.

Given a problem solving task its AnsProlog* formulation primarily depends on how the possibilities are enumerated. I.e., what variables are used and what values these variables can take. Often explicit or implicit knowledge about the domain can be used to reduce the size of the possibility space (or state space). Also, often some of the test conditions can be pushed inside the generate phase. One needs to be very careful in this though, as some times the pushing of the test conditions may be done wrong although at first glance it may appear to be right. We give such an example for the n-queens problem in Section 4.1.1. In representing the constraints, one also needs to be careful in whether to represent the constraint as a fact or as a rule with empty. For example, consider the following program Π, where the possibility space is $\{a, p\}$ and $\{b, q\}$.

$p \leftarrow a.$
$q \leftarrow b.$
$a \leftarrow$ **not** $b.$
$b \leftarrow$ **not** $a.$

Now, suppose our constraint is 'p must be true'. I.e, we would like to eliminate the possibilities where p is not true. If we try to do this by adding the fact $p \leftarrow .$ to Π, then the resulting program will have the answer sets $\{a, p\}$, and $\{b, q, p\}$, which is not what we want. What we want is to eliminate $\{b, q\}$ from the possibility space and have $\{a, p\}$ as the answer. The right way to achieve this is to add $\leftarrow$ **not** p to Π.

4.1.1 N-queens

In the n-queens problem we have an $n \times n$ board and we have to place n queens such that no two queens attack each other. I.e., there are no more than 1 queen in each row and column and no two queens are along the same diagonal line.

Following the enumerate and eliminate approach we first need to enumerate the placing of n queens in the $n \times n$ board and then eliminate those possibilities or configurations where two queens may attack each other. We now present several different encodings of this. These encodings differ in their formulation of the 'enumerate' and 'eliminate' parts. In some the 'enumerate' part it self consists of a weaker enumeration and elimination; in others part of the 'eliminate' conditions are pushed into the 'enumerate' part.

1. Placing queens one by one with possibility space based on the squares: In this formulation we name the queens from $1 \ldots n$ and use the predicate $at(I, X, Y)$ to mean that queen I is in location (X, Y). The 'enumerate' part in this formulation consists of a weaker enumeration of the possibility space which specifies that each location may or may not have queen I, and then has elimination constraints to force each queen to be in exactly one location, and no two queens in the same location. The 'elimination' part is the usual one. The formulation is as follows:

   (a) Declarations: We have the following domain specifications.

   $queen(1) \leftarrow .$ $\quad \ldots \quad$ $queen(n) \leftarrow .$
   $row(1) \leftarrow .$ $\quad \ldots \quad$ $row(n) \leftarrow .$
   $col(1) \leftarrow .$ $\quad \ldots \quad$ $col(n) \leftarrow .$

   (b) Enumeration: The enumeration rules create the possibility space such that the n different queens are placed in the $n \times n$ board in different locations. The rules with their intuitive meaning are as follows:

      i. For each locations (X, Y) and each queen I, either I is in location (X, Y) or not.

      $at(I, X, Y) \leftarrow queen(I), row(X), col(Y),$ **not** $not\_at(I, X, Y).$
      $not\_at(I, X, Y) \leftarrow queen(I), row(X), col(Y),$ **not** $at(I, X, Y).$

      ii. For each queen I it is placed in at most one location.

      $\leftarrow queen(I), row(X), col(Y), row(U), col(Z), at(I, X, Y), at(I, U, Z), Y \neq Z.$
      $\leftarrow queen(I), row(X), col(Y), row(Z), col(V), at(I, X, Y), at(I, Z, V), X \neq Z.$

iii. For each queen I it is placed in at most one location.

$placed(I) \leftarrow queen(I), row(X), col(Y), at(I, X, Y).$
$\leftarrow queen(I), \textbf{not } placed(I).$

iv. No two queens are placed in the same location.

$\leftarrow queen(I), row(X), col(Y), queen(J), at(I, X, Y), at(J, X, Y), I \neq J.$

(c) Elimination:

i. No two distinct queens in the same row.

$\leftarrow queen(I), row(X), col(Y), col(V), queen(J), at(I, X, Y), at(J, X, V), I \neq J.$

ii. No two distinct queens in the same column.

$\leftarrow queen(I), row(X), col(Y), row(U), queen(J), at(I, X, Y), at(J, U, Y), I \neq J.$

iii. No two distinct queens attack each other diagonally.

$$\leftarrow row(X), col(Y), row(U), col(V), queen(I), queen(J), at(I, X, Y), at(J, U, V), I \neq J, abs(X - U) = abs(Y - V).$$

Note that the rule 1 (b) (iv) is subsumed by both 1(c)(i) and 1(c)(ii). In other words, in presence of 1(c)(i) or 1(c)(ii) we do not need 1 (b) (iv). It is needed in 1 (b) if our only goal is enumeration.

Exercise 11 Explain why replacing the second rule of 1 (b) (iii) by the following rule makes the program incorrect.
$placed(I) \leftarrow queen(I).$ □

2. Placing queens one by one in unique locations: We no present a formulation where while enumerating the possibility space care is taken such that queens are placed in unique locations

(a) Declarations: As in the previous formulation.

(b) Enumeration:

i. The combined effect of 1 (b) (i) -(iii) is achieved by the following rules which ensure that each queen is uniquely placed. the first two rules define $other\_at(I, X, Y)$ which intuitively means that the queen I has been placed in a location other than (X, Y). The third rule enforces the condition that if queen I has not been placed in a location different from (X, Y), then it must be placed in (X, Y).

$other\_at(I, X, Y) \leftarrow queen(I), row(X), col(Y), row(U), col(Z), at(I, U, Z), Y \neq Z.$
$other\_at(I, X, Y) \leftarrow queen(I), row(X), col(Y), row(Z), col(V), at(I, Z, V), X \neq Z.$
$at(I, X, Y) \leftarrow queen(I), row(X), col(Y), \textbf{not } other\_at(I, X, Y).$

ii. The following rule is same as in 1 (b) (iv) and it forces two distinct queens to be placed in different locations.

$\leftarrow queen(I), row(X), col(Y), queen(J), at(I, X, Y), at(J, X, Y), I \neq J.$

(c) Elimination: As in the previous formulation.

3. Placing queens one by one in unique locations so that they don't attack each other horizontally or vertically: In this formulation we push two of the elimination constraints into the enumeration phase. Thus while enumerating the possibility space we ensure that no two queens are in the same row or same column.

(a) Declarations: As in the previous formulation.

(b) Enumeration:

i. As in 2 (b) (i)

$other\_at(I, X, Y) \leftarrow queen(I), row(X), col(Y), row(U), col(Z), at(I, U, Z), Y \neq Z.$
$other\_at(I, X, Y) \leftarrow queen(I), row(X), col(Y), row(Z), col(V), at(I, Z, V), X \neq Z.$
$at(I, X, Y) \leftarrow queen(I), row(X), col(Y), \textbf{not}\ other\_at(I, X, Y).$

ii. The first two constraints in 2 (c) – same as 1 (c) (i) and 1 (c) (ii) – are replaced by the following. We also no longer need 2 (b) (ii) which is subsumed by the following.

$other\_at(I, X, Y) \leftarrow queen(I), row(X), col(Y), col(V), queen(J), at(J, X, V), I \neq J.$
$other\_at(I, X, Y) \leftarrow queen(I), row(X), col(Y), row(U), queen(J), at(J, U, Y), I \neq J.$

(c) Elimination: We now need only one elimination rule, the same one as 1 (c) (iii).

$$\leftarrow row(X), col(Y), row(U), col(V), queen(I), queen(J), at(I, X, Y), at(J, U, V), I \neq J, \\ abs(X - U) = abs(Y - V).$$

4. Placing queens with possibility space based on the squares: In the last three formulations we named the queens and placed them one by one. We can get both computational efficiency and a smaller encoding by not naming the queens. For example, when queens are numbered the 4-queens problem has 48 solutions while if we do not distinguish the queens, it has only 2 solutions. In the following we simplify the formulation in (1) by not distinguishing the queens. We use the predicate $in(X, Y)$ to mean that a queen is placed in location (X, Y).

(a) Declarations: The simpler domain specification is as follows:

$row(1) \leftarrow .$ ... $row(n) \leftarrow.$
$col(1) \leftarrow .$ ... $col(n) \leftarrow.$

(b) Enumeration: The enumeration now has two parts. Since we do not distinguish between the queens we no longer need 1(b) (ii)-(iv).

i. Following is the simplification of 1 (b) (i) which specifies that each square either has a queen or does not.

$not\_in(X, Y) \leftarrow row(X), col(Y), \textbf{not}\ in(X, Y).$
$in(X, Y) \leftarrow row(X), col(Y), \textbf{not}\ not\_in(X, Y).$

ii. To make sure that we placed all the n queens, instead of counting, we use the knowledge that for the queens to not attack each other they must be in different rows, and hence to place all the n queens we must have a queen in each row. We specify this using the following:

$has\_queen(X) \leftarrow row(X), col(Y), in(X, Y).$
$\leftarrow row(X), \textbf{not}\ has\_queen(X).$

Note that we did not need rules similar to the above in 1 (b) as there we numbered the queens and by ensuring that each queen at at least one location, we made sure that all the queens were placed.

(c) Elimination: The elimination rules below are simplified versions of 1 (c).

i. Two queens can not be placed in the same column.

$\leftarrow row(X), col(Y), col(YY), Y \neq YY, in(X, Y), in(X, YY).$

ii. Two queens can not be placed in the same row.

$\leftarrow row(X), col(Y), row(XX), X \neq XX, in(X,Y), in(XX,Y).$

iii. Two queens can not be placed so that they attack each other diagonally.

$\leftarrow row(X), col(Y), row(XX), col(YY), X \neq XX, Y \neq YY, in(X,Y), in(XX,YY),$
$abs(X - XX) = abs(Y - YY).$

Exercise 12 Explain why we can not replace the rules in 4 (b) (ii) by the following:

$\leftarrow row(X), col(Y), \textbf{not } in(X,Y).$ □

5. Placing queens so that they don't attack each other horizontally or vertically: Similar to the formulation in (3) we now push the constraints that no two queens are in the same row or same column into the enumeration phase.

   (a) Declarations: As in the previous formulation:

   (b) Enumeration: We now push the constraints in 4 c (i) and (ii) into the enumeration phase, and generate possibilities of queen placement such that no two queens are in the same column or row, and at least one is in each row and column. We do this by using an auxiliary predicate $not\_in(X,Y)$ which intuitively means that a queen should not be placed in location (X,Y).

      i. A queen should not be placed in (X,Y) if there is queen placed in the same row.

      $not\_in(X,Y) \leftarrow row(X), col(Y), col(YY), Y \neq YY, in(X,YY)$

      ii. A queen should not be placed in (X,Y) if there is queen placed in the same column.

      $not\_in(X,Y) \leftarrow row(X), col(Y), row(XX), X \neq XX, in(XX,Y)$

      iii. A queen must be placed in (X,Y) if it is not otherwise prevented.

      $in(X,Y) \leftarrow row(X), col(Y), \textbf{not } not\_in(X,Y)$

   (c) Elimination: We now need only one elimination constraint, the same one as in 4 (c) (iii) which prevents placements where queens can attack each other diagonally.

   $\leftarrow row(X), col(Y), row(XX), col(YY), X \neq XX, Y \neq YY, in(X,Y), in(XX,YY),$
   $abs(X - XX) = abs(Y - YY).$

6. A non-solution: Let us continue the process of pushing the elimination phase into the enumeration phase one more step by removing 5 (c) and adding the following rule:

   $not\_in(X,Y) \leftarrow row(X), col(Y), row(XX), col(YY), X \neq XX, Y \neq YY, in(X,Y), in(XX,YY),$
   $abs(X - XX) = abs(Y - YY).$

   This will result in the following overall formulation.

   $row(1) \leftarrow .$ ... $row(n) \leftarrow.$
   $col(1) \leftarrow .$ ... $col(n) \leftarrow.$

   $not\_in(X,Y) \leftarrow row(X), col(Y), col(YY), Y \neq YY, in(X,YY).$
   $not\_in(X,Y) \leftarrow row(X), col(Y), row(XX), X \neq XX, in(XX,Y).$

$$not\_in(X,Y) \leftarrow row(X), col(Y), row(XX), col(YY), X \neq XX, Y \neq YY, in(X,Y), in(XX,YY), abs(X-XX) = abs(Y-YY).$$
$$in(X,Y) \leftarrow dim(X), dim(Y), \textbf{not } not\_in(X,Y).$$

Unfortunately the above formulation is *not correct* in the sense that it has answer sets where not all the n queens are placed. For example, if $n = 4$, then one of the answer sets encode the following placement: $\{in(1,1), in(3,2), in(2,4)\}$, where only three queens are placed, instead of four.

Further reasoning reveals that the above encoding places queens into a valid configuration, where a valid configuration is a configuration of queens that do not attack each other and to which additional queens can not be added without that queen attacking one or more of the already placed queens. The configuration $\{in(1,1), in(3,2), in(2,4)\}$ is valid in that sense, as we can not add a new queen to it in such a way that it does not attack the already placed ones. Hence, we need to be careful in pushing elimination constraints into the enumeration phase.

Exercise 13 Add additional rules to the formulation in (6) to make it work. □

4.1.2 Tile Covering of boards with missing squares

Consider covering slightly broken $n \times n$ checker boards with tiles of size 1 by 2. We need to find a covering of the board,if exists, using 1×2 tiles so that all the good squares are covered. Otherwise we need to report that no covering exists. A particular case is a 6×6 board. We need to show that if the board is missing square (1,6) and (1,5) then there is a covering; and if the board is missing square (1,6) and (6,1) then there is no covering.

1. A formulation using two squares to denote a tile: In this formulation we use the predicate $rt_iop(X,Y)$ meaning that Y is an adjacent square to X, either on the right of X or on the top of X. The enumeration phase involves selecting pairs (X,Y) to cover with a tile so that $rt_iop(X,Y)$ is true. In the elimination phase we make sure that two different tiles do not cover the same square, and all squares are covered.

   (a) We have the following domain specification.

   $row(1) \leftarrow.$ ... $row(6) \leftarrow.$
   $col(1) \leftarrow.$ ... $col(6) \leftarrow.$

   (b) We use the predicate sq to define when a square is not missing, and the predicate $missing$ to denote that the square is missing in the board.

   $sq(X,Y) \leftarrow row(X), col(Y), \textbf{not } missing(X,Y).$

   (c) The two particular instances in a 6×6 board is expressed by specifying the missing squares as given below:

   i. $missing(1,6) \leftarrow.$ $missing(6,1) \leftarrow .$
   ii. $missing(1,6) \leftarrow.$ $missing(1,5) \leftarrow .$

   (d) The following two rules define when two squares are adjacent with the second square to the right or top of the first one.

$rt\_top(X, Y, XX, Y) \leftarrow sq(X, Y), sq(XX, Y), XX = X + 1.$
$rt\_top(X, Y, X, YY) \leftarrow sq(X, Y), sq(X, YY), YY = Y + 1.$

(e) Enumeration: Given two right-top adjacent squares the following rules either select it to be covered by a tile or leave it out as a pair. (Each individual square in that pair may be selected together with another square for being covered by a tile.)

$sel\_rt\_top(X, Y, XX, YY) \leftarrow rt\_top(X, Y, XX, YY), \textbf{not } n\_sel\_rt\_top(X, Y, XX, YY).$
$n\_sel\_rt\_top(X, Y, XX, YY) \leftarrow rt\_top(X, Y, XX, YY), \textbf{not } sel\_rt\_top(X, Y, XX, YY).$

(f) Generalizing *sel_rt_top* to *sel*: To simplify the elimination rules we define the predicate *sel* which holds for a pair of adjacent squares if the right-top representation of the two squares is selected.

$sel(X, Y, XX, YY) \leftarrow sq(X, Y), sq(XX, YY), sel\_rt\_top(X, Y, XX, YY).$
$sel(X, Y, XX, YY) \leftarrow sq(X, Y), sq(XX, YY), sel\_rt\_top(XX, YY, X, Y).$

(g) Elimination of selections that conflict: We eliminate possibilities where a square (X, Y) is covered by two different tiles. We represent (U, V) is different from (W, Z) by encoding it as $U \neq W$ or $V \neq Z$.

$\leftarrow sel(X, Y, U, V), sel(X, Y, W, Z), sq(X, Y), sq(U, V), sq(W, Z), U \neq W.$
$\leftarrow sel(X, Y, U, V), sel(X, Y, W, Z), sq(X, Y), sq(U, V), sq(W, Z), V \neq Z.$

(h) Finally, we elimination selections where some square is left uncovered.
$covered(X, Y) \leftarrow sq(X, Y), sq(XX, YY), sel(X, Y, XX, YY).$
$\leftarrow sq(X, Y), notcovered(X, Y).$

The above formulation is not very efficient when it is run through the Smodels system. We believe the reason is that the ground version of the above program is quite large. For example, the constraints in part (g) have 6 variables and each of them can take n values; thus a naive grounding of it results in 6^n ground rules. We found that for $n = 8$, the above program did not return an answer within 2 minutes when we ran it in a WinTel 400 MHz laptop with 64 MB of RAM. (The lparse grounding took 30 second, the smodels reading took another 60 sec, and there was no answer for the next 60 seconds.)

2. An efficient formulation with tiles represented by a single co-ordinate: To decrease the size of the grounding we present a formulation where tiles are represented only by their left or bottom co-ordinate. To distinguish between horizontal and vertical covering we now have two – instead of the one in the previous formulation – predicates.

(a)-(c) are as before.

(d) Selecting horizontal coverings: For each square (X, Y) the following two rules enumerate the possibilities that either there is a horizontal tile with its left end at (X, Y) or not.

$sel\_rt(X, Y) \leftarrow sq(X, Y), sq(X + 1, Y), \textbf{not } n\_sel\_rt(X, Y).$
$n\_sel\_rt(X, Y) \leftarrow sq(X, Y), sq(X + 1, Y), \textbf{not } sel\_rt(X, Y).$

(e) Selecting vertical coverings: For each square (X, Y) the following two rules enumerate the possibilities that either there is a vertical tile with its bottom end at (X, Y) or not.

$sel\_top(X, Y) \leftarrow sq(X, Y), sq(X, Y + 1), \textbf{not } n\_sel\_top(X, Y).$
$n\_sel\_top(X, Y) \leftarrow sq(X, Y), sq(X, Y + 1), \textbf{not } sel\_top(X, Y).$

(f) Elimination of coverings that conflict: The following rules eliminate the possibilities where a square (X, Y) is covered by two different tiles.

$\leftarrow sq(X,Y), sel\_rt(X,Y), sel\_rt(X+1,Y).$
$\leftarrow sq(X,Y), sel\_top(X,Y), sel\_top(X,Y+1).$
$\leftarrow sq(X,Y), sel\_rt(X,Y), sel\_top(X,Y).$
$\leftarrow sq(X,Y), sel\_rt(X,Y), sel\_top(X+1,Y).$
$\leftarrow sq(X,Y), sel\_rt(X,Y), sel\_top(X,Y-1).$
$\leftarrow sq(X,Y), sel\_rt(X,Y), sel\_top(X+1,Y-1).$

(g) Elimination of selections that do not cover some square: The following rule eliminates possibilities where some square (X, Y) is left uncovered. A square (X, Y) can be covered in four different ways: a horizontal tile with its left end in $(X-1, Y)$ or in (X, Y) and a vertical tile with its bottom end in $(X, Y-1)$ or in (X, Y). In neither of these four cases hold then we reason that (X, Y) is left uncovered.

$\leftarrow sq(X,Y), \textbf{not } sel\_rt(X,Y), \textbf{not } sel\_rt(X-1,Y),$
$\textbf{not } sel\_top(X,Y), \textbf{not } sel\_top(X,Y-1).$

The above formulation leads to a small number of groundings, when compared with respect to the previous formulation. To be specific, it leads to $12 \times n^2$ ground rules beyond the domain representation. For $n = 8$, when we ran it through the Smodels system in a WinTel 400 MHz laptop with 64 MB of RAM, it took 0.87 seconds

4.1.3 Who let the Zebra out?

In this well known problem, there are five houses each of a different color (red, green, ivory, blue, and yellow) and inhabited by a person of a different nationality (Japanese, Englishman, Norwegian, Ukrainian, and Spaniard), with a different pet (horse, snails, zebra, fox, and dog), drink (water, coffee, tea, milk, and orange juice) and brand of cigarettes (Lucky strike, Winston, Chesterfields, Kools, and Parliaments).

It is given that

1. The Englishman lives in the red house.
2. The Spaniard owns the dog.
3. The Norwegian lives in the first house on the left.
4. Kools are smoked in the yellow house.
5. The man who smokes Chesterfields lives in the house next to the man with the fox.
6. The Norwegian lives next to the blue house.
7. The Winston smoker owns snails.
8. The Lucky Strike smoker drinks orange juice.
9. The Ukrainian drinks tea.
10. The Japanese smokes Parliaments.

11. Kools are smoked in the house next to the house where the horse is kept.
12. Coffee is drunk in the green house.
13. The green house is immediately to the right (your right) of the ivory house.
14. Milk is drunk in the middle house.

A zebra is found wandering in the streets and the animal shelter wants to find out who let the zebra out. I.e., which house the zebra belongs to. The solution of this problem is given by the following table:

number	cigarette brand	country	color	pet	drink
1	Kools	Norway	yellow	fox	water
2	Chesterfields	Ukraine	blue	horse	tea
3	Winston	UK	red	snails	milk
4	Lucky-strike	Spain	ivory	dog	oj
5	Parliaments	Japan	green	zebra	coffee

Our goal is to come up with an AnsProlog encoding of the above problem solving task. We rule out an encoding based on a predicate $together(U, V, W, X, Y, Z)$ meaning that the house number U is together with the cigarette brand V, country W, color X, pet Y, and drink Z. The reason behind ruling such an encoding is to limit the number of variables in rules, and the use of $together(U, V, W, X, Y, Z)$ leads to some rules – where we want to say that no to together atoms can have one of the parameter same and another parameter different – with at least seven variables. Such a rule instantiates to $5^7 = 78125$ instances. We now present two different encodings for this problem where instead of using *together* we use the house number as an anchor and associate the other features with the house number.

1. An encoding with house number as anchor and separate predicates for each association: In the following encoding we use house number as the anchor, and have binary predicates *has_color*, *has_drink*, *has_animal*, *has_country*, and *has_smoke* where the first parameter is the house number and the second parameters are color, drink, pet, country, and smoke respectively.

   (a) We have the following domain specification:

   $house(1) \leftarrow.$ $\quad \ldots \quad$ $house(5) \leftarrow.$

   $right(X, X+1) \leftarrow house(X), house(X+1).$

   $left(X+1, X) \leftarrow house(X), house(X+1).$

   $next(X, Y) \leftarrow right(X, Y).$
   $next(X, Y) \leftarrow left(X, Y).$

   $color(red) \leftarrow.$ $\quad color(green) \leftarrow.$ $\quad color(blue) \leftarrow.$
   $color(ivory) \leftarrow.$ $\quad color(yellow) \leftarrow.$

   $country(norway) \leftarrow.$ $\quad country(japan) \leftarrow.$ $\quad country(uk) \leftarrow.$
   $country(spain) \leftarrow.$ $\quad country(ukraine) \leftarrow.$

   $smoke(parliaments) \leftarrow.$ $\quad smoke(lucky\_strike) \leftarrow.$ $\quad smoke(kools) \leftarrow.$
   $smoke(chesterfield) \leftarrow.$ $\quad smoke(winston) \leftarrow.$

$drink(coffee) \leftarrow.$ $drink(tea) \leftarrow.$ $drink(oj) \leftarrow.$
$drink(milk) \leftarrow.$ $drink(water) \leftarrow.$

$animal(dog) \leftarrow.$ $animal(snails) \leftarrow.$ $animal(horse) \leftarrow.$
$animal(fox) \leftarrow.$ $animal(zebra) \leftarrow.$

(b) The enumerations: We now enumerate the predicates *has_color*, *has_drink*, *has_animal*, *has_country*, and *has_smoke*. For the predicate *has_color*, we make sure that every house has a unique color assigned to it, and every color corresponds to a unique house. This is achieved by the following three rules:

$other\_color(H, C) \leftarrow house(H), color(C), color(CC), has\_color(H, CC), C \neq CC.$
$other\_color(H, C) \leftarrow house(H), color(C), house(HH), has\_color(HH, C), H \neq HH.$
$has\_color(H, C) \leftarrow house(H), color(C), \textbf{not}\ other\_color(H, C).$

The enumeration rules for the predicates *has_drink*, *has_animal*, *has_country*, and *has_smoke* are similar to that of the enumeration rules for *has_color* and is given below:

$other\_drink(H, C) \leftarrow house(H), drink(C), drink(CC), has\_drink(H, CC), C \neq CC.$
$other\_drink(H, C) \leftarrow house(H), drink(C), house(HH), has\_drink(HH, C), H \neq HH.$
$has\_drink(H, C) \leftarrow house(H), drink(C), \textbf{not}\ other\_drink(H, C).$

$other\_animal(H, C) \leftarrow house(H), animal(C), animal(CC),$
$\qquad has\_animal(H, CC), C \neq CC.$
$other\_animal(H, C) \leftarrow house(H), animal(C), house(HH),$
$\qquad has\_animal(HH, C), H \neq HH.$
$has\_animal(H, C) \leftarrow house(H), animal(C), \textbf{not}\ other\_animal(H, C).$

$other\_country(H, C) \leftarrow house(H), country(C), country(CC),$
$\qquad has\_country(H, CC), C \neq CC.$
$other\_country(H, C) \leftarrow house(H), country(C), house(HH),$
$\qquad has\_country(HH, C), H \neq HH.$
$has\_country(H, C) \leftarrow house(H), country(C), \textbf{not}\ other\_country(H, C).$

$other\_smoke(H, C) \leftarrow house(H), smoke(C), smoke(CC), has\_smoke(H, CC), C \neq CC.$
$other\_smoke(H, C) \leftarrow house(H), smoke(C), house(HH), has\_smoke(HH, C), H \neq HH.$
$has\_smoke(H, C) \leftarrow house(H), smoke(C), \textbf{not}\ other\_smoke(H, C).$

(c) The elimination constraints: The fourteen given facts (or observations) are each encoded by the following 14 rules. If the fact i is satisfied then it forces s_i to be true.

$s_1 \leftarrow house(H), has\_country(H, uk), has\_color(H, red).$
$s_2 \leftarrow house(H), has\_animal(H, dog), has\_country(H, spain).$
$s_3 \leftarrow has\_country(1, norway).$
$s_4 \leftarrow house(H), has\_smoke(H, kools), has\_color(H, yellow).$
$s_5 \leftarrow house(H), house(HH), has\_smoke(H, chesterfield), has\_animal(HH, fox), next(H, HH).$
$s_6 \leftarrow house(H), house(HH), has\_color(H, blue), has\_country(HH, norway), next(H, HH).$
$s_7 \leftarrow house(H), has\_smoke(H, winston), has\_animal(H, snails).$
$s_8 \leftarrow house(H), has\_smoke(H, lucky\_strike), has\_drink(H, oj).$
$s_9 \leftarrow house(H), has\_country(H, ukraine), has\_drink(H, tea).$
$s_{10} \leftarrow house(H), has\_smoke(H, parliaments), has\_country(H, japan).$
$s_{11} \leftarrow house(H), house(HH), has\_smoke(H, kools), has\_animal(HH, horse), next(H, HH).$

$s_{12} \leftarrow house(H), has\_color(H, green), has\_drink(H, coffee).$
$s_{13} \leftarrow house(H), house(HH), has\_color(HH, ivory), has\_color(H, green), right(HH, H).$
$s_{14} \leftarrow has\_drink(3, milk).$

The following rule encodes that $satisfied$ is true iff all the 14 constraints are individually satisfied.

$satisfied \leftarrow s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8, s_9, s_{10}, s_{11}, s_{12}, s_{13}, s_{14}.$

The following rule eliminates possibilities were $satsified$ is not true. I.e., when one of the 14 conditions does not hold.

$\leftarrow$ **not** $satisfied.$

The above encoding finds the unique solution. We now present another encoding which instead of using the five association predicates: has_color, has_drink, has_animal, $has\_country$, and has_smoke; uses only a single association predicate.

2. An encoding with house number as anchor and a single association predicate: In the following encoding we use a single association predicate, and use two new predicates $object$ and $same\_type$ to distinguish the association of a house number with objects of different types. The main advantage of using a single association predicate is that it makes it easier to express the observations as we do not have to remember the particular association predicate for the object being described.

   (a) Besides the domain specification in 1 (a) we have the following additional rules.

   $object(X) \leftarrow color(X).$
   $object(X) \leftarrow drink(X).$
   $object(X) \leftarrow animal(X).$
   $object(X) \leftarrow country(X).$
   $object(X) \leftarrow smoke(X).$

   $same\_type(X, Y) \leftarrow color(X), color(Y).$
   $same\_type(X, Y) \leftarrow drink(X), drink(Y).$
   $same\_type(X, Y) \leftarrow animal(X), animal(Y).$
   $same\_type(X, Y) \leftarrow country(X), country(Y).$
   $same\_type(X, Y) \leftarrow smoke(X), smoke(Y).$

   (b) We have a single association predicate $has(X, Y)$ which intuitively means that house number X is associated with object Y. The following three enumeration rules ensure that each house number is associated with a unique object of a particular type, and each object of a particular type is associated with only a particular house number.

   $other\_has(X, Y) \leftarrow house(X), object(Y), house(Z), Z \neq X, has(Z, Y).$
   $other\_has(X, Y) \leftarrow house(X), object(Y), object(Z), Z \neq Y, same\_type(Y, Z), has(X, Z).$
   $has(X, Y) \leftarrow house(X), object(Y),$ **not** $other\_has(X, Y).$

   (c) The elimination constraints: The fourteen given facts (or observations) are each encoded by the following 14 rules. If the fact i is satisfied then it forces s_i to be true. The following rules are simplified versions of the rules in 1 (c). The simplification is that we no longer have to use different association predicates for objects of different types.

$s_1 \leftarrow house(H), has(H, uk), has(H, red).$
$s_2 \leftarrow house(H), has(H, dog), has(H, spain).$
$s_3 \leftarrow has(1, norway).$
$s_4 \leftarrow house(H), has(H, kools), has(H, yellow).$
$s_5 \leftarrow house(H), house(HH), has(H, chesterfield), has(HH, fox), next(H, HH).$
$s_6 \leftarrow house(H), house(HH), has(H, blue), has(HH, norway), next(H, HH).$
$s_7 \leftarrow house(H), has(H, winston), has(H, snails).$
$s_8 \leftarrow house(H), has(H, lucky_strike), has(H, oj).$
$s_9 \leftarrow house(H), has(H, ukraine), has(H, tea).$
$s_{10} \leftarrow house(H), has(H, parliaments), has(H, japan).$
$s_{11} \leftarrow house(H), house(HH), has(H, kools), has(HH, horse), next(H, HH).$
$s_{12} \leftarrow house(H), has(H, green), has(H, coffee).$
$s_{13} \leftarrow house(H), house(HH), has(H, green), has(HH, ivory), right(HH, H).$
$s_{14} \leftarrow has(3, milk).$

The following rules are same as in 1 (c).

$satisfied \leftarrow s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8, s_9, s_{10}, s_{11}, s_{12}, s_{13}, s_{14}.$

$\leftarrow$ **not** $satisfied.$

4.2 Constraint Satisfaction Problems

Many problem solving tasks can be cast as a constraint satisfaction problem (CSP) and the solution of the CSP then leads to the solution of the original problem. In this section we formally define a CSP, and show how to encode a CSP in AnsProlog show that there is a one to one correspondence between the solutions to the CSP and the answer sets of the AnsProlog encoding. We then demonstrate this technique with respect to two problem solving tasks. We now formally define a CSP.

A constraint satisfaction problem (CSP) consists of

- a set of *variables* ;
- a set of possible values for each variable, called the *domain* of the variable;
- and a set of two kinds of *constraints*: a set of allowed combinations of variables and values, and disallowed combinations of variables and values.

A solution to a CSP is an assignment to the variables (among the possible values) such that the constraints are satisfied.

We now give an encoding of a CSP problem P using AnsProlog such that there is a 1-1 correspondence between solutions of the CSP and answer sets of the encoded AnsProlog program $\Pi(P)$.

1. For each domain value c in the CSP we include the constant C in the language of $\Pi(P)$.
2. For each domain d in the CSP the program $\Pi(P)$ has a unary predicate d and the following set of facts

   $d(c_1) \leftarrow$

$\vdots$

$d(c_n) \leftarrow$

where $c_1, \ldots, c_n$ are the possible values of domain d.

3. For each variable v with the domain d in the CSP the program $\Pi(P)$ has the unary predicates v and $other\_v$ and the rules:

   $v(X) \leftarrow d(X), \textbf{not } other\_v(X)$

   $other\_v(X) \leftarrow d(X), d(Y), v(Y), X \neq Y$

4. For each constraint co giving a set of allowed value combinations for a set of variables $v_1, \ldots, v_j$ $\Pi(P)$ has the fact

   $constraint(co) \leftarrow$

   and for each allowed value combination $v_1 = c_1, \ldots, v_j = c_j$ $\Pi(P)$ has the rule

   $sat(co) \leftarrow v_1(c_1), \ldots, v_j(c_j)$

   and finally $\Pi(P)$ has the rule:

   $\leftarrow constraint(C), \textbf{not } sat(C)$

5. For each constraint that disallows combinations $v_1 = c_1, \ldots, v_j = c_j$ the program $\Pi(P)$ has the rule:

   $\leftarrow v_1(c_1), \ldots, v_j(c_j)$

Although many problem solving tasks can be formulated as CSPs, often the CSP encoding is more cumbersome and results in a larger encoding than if we were to encode the problem directly in AnsProlog. This happens because often the constraints can be more succinctly expressed by exploiting the relationship between the variables. This relationship is not encoded in a CSP and hence the number of constraints becomes larger. We illustrate this with respect to the n-queens example.

4.2.1 N-queens as a CSP instance

The n-queens problem can be cast as a CSP instance as follows:

- We have n variables which we denote by $r[1], \ldots, r[n]$.
- Each of these variables can take a value between 1 and n. Intuitively, $r[i] = j$ means that in the ith row the queen is placed in the jth column.
- The constraints are (a) No two rows can have a queen in the same column, I.e. for $X \neq Y$, the value of $r[X]$ must be different from $r[Y]$; and (b) There should not be two queens that attack each other diagonally. I.e., if $r[X] = Y$ and $r[XX] = YY$ and $X \neq XX$ and $Y \neq YY$ then $abs(X - XX) \neq abs(Y - YY)$.

To conform with the CSP notation we either need a preprocessor or need multiple explicit rules to express the above constraints. For example, the constraint (a) is expressible through the following $\frac{n\times(n-1)}{2}$ explicit constraints:

$r[1] = r[2]$, $r[1] = r[3]$, ..., $r[1] = r[n]$, $r[2] = r[3]$, ..., $r[2] = r[n]$, ..., $r[n-1] = r[n]$ are not allowed.

In contrast a direct representation in AnsProlog allows us to treat the X in $r[X]$ as a variable, and thus we can represent the constraints much more succinctly. For example, we can view the predicate $in(X, Y)$ in the encoding (4) of Section 4.1.1 to mean that in row X, the queen is in column Y. Since $in(X, Y)$ is a relation that encodes a function we need the following:

$\leftarrow in(X, Y), in(X, YY), Y \neq YY.$

Now we can write the constraints simply as:

$\leftarrow in(X, Y), in(XX, Y), X \neq XX.$
$\leftarrow in(X, Y), in(XX, YY), abs(X - XX) = abs(Y - YY).$

4.2.2 Schur as a CSP instance

In this problem we are required to assign a set $N = \{1, 2, \ldots, n\}$ of integers into b boxes such that for any $x, y \in N$: (a) x and $2x$ are in different boxes and (b) if x and y are in the same box, then $x + y$ is in a different box. We can cast this problem as a CSP in the following way:

- We have n variables denoted by $assign[1], \ldots, assign[n]$.
- Each of these variables can take a value between $1 \ldots b$.
- To represent the constraint "x and $2x$ are in different boxes" we need the following explicit disallowed combinations in the CSP notation:

  $assign[1] = assign[2]$, $assign[2] = assign[4]$, $assign[3] = assign[6]$, ..., $assign[\lfloor \frac{n}{2} \rfloor] = assign[2 \times \lfloor \frac{n}{2} \rfloor]$ are not allowed combinations.

  Similarly, to represent the other constraint "if x and y are in the same box, then $x + y$ is in a different box" we need multiple explicit disallowed combinations in CSP notation.

We now present an AnsProlog$^{\perp}$ encoding of the above problem that exploits the relationship between the variables. In the following program $in(X, B)$ means that the integer X is assigned to box B.

1. The domain specifications:

   $num(1) \leftarrow .$ ... $num(n) \leftarrow .$
   $box(1) \leftarrow .$ ... $box(b) \leftarrow .$

2. Assigning integers to boxes:

   $not\_in(X, B) \leftarrow num(X), box(B), box(BB), B \neq BB, in(X, BB).$
   $in(X, B) \leftarrow num(X), box(B),$ **not** $not\_in(X, B).$

3. Constraints:

(a) The first constraint: x and $2x$ are in different boxes.

$\leftarrow num(X), box(B), in(X, B), in(X + X, B).$

(b) The second constraint: if x and y are in the same box, then $x + y$ is in a different box.

$\leftarrow num(X), num(Y), box(B), in(X, B), in(Y, B), in(X + Y, B).$

Although in the above two examples, the direct encoding in AnsProlog$^{\perp}$ was more succinct than representing it in a CSP and then translating it to an AnsProlog program, there are many problems with good CSP representations and they are good candidates for using the translation method.

4.3 Dynamic Constraint Satisfaction Problem

A dynamic constraint satisfaction problem (DCSP) is an extension of a CSP and consists of:

1. a set $V = \{v_1, \ldots, v_n\}$ of variables;
2. a set $\mathcal{D} = \{D_1, \ldots, D_n\}$ of domains of the variables, each domain consisting of a finite set of values that the corresponding variable can take;
3. a set $V_I \subseteq V$ of initial variables;
4. a set of *compatibility constraints* each of which specifies the set of allowed combinations of values for a set of variable $v_1, \ldots, v_j$; and
5. a set of *activity constraints* that prescribes conditions when a variable must have an assigned value (i.e., be active) and when it must not be active. An activity constraint that activates a variable v is of the form **if** c **then** v, where c is of the form of a compatibility constraint. Similarly, an activity constraint that deactivates a variable v is of the form **if** c **then not** v. The former is referred to as a *require activity constraint* and the later is referred to as a *not-require activity constraint.*

Unlike a CSP, all the variables in a DCSP need not be active, but the set of *initial variables* must be active in every solution. A solution to a DCSP problem is an assignment A of values to variables such that it

1. satisfies the compatibility constraints (i.e., for all compatibility constraint c either some variable in c is inactive or A satisfies c);
2. satisfies the activity constraints (i.e., for all activity constraint of the form **if** c **then** v, if A satisfies c then v is active in A);
3. contains assignments for the initial variables; and
4. is subset minimal.

4.3.1 Encoding DCSP in AnsProlog

In this section we only give an encoding of DCSP in AnsProlog$_\oplus$, where $\oplus$ denotes the ex-or operator. The resulting AnsProlog$_\oplus$ program can be translated to an AnsProlog program using the translation described in Section 2.1.16. The mapping of a DCSP to an AnsProlog$_\oplus$ program is as follows:

1. The language:
   (a) a new distinct atom for each variable v_i to encode its activity,
   (b) a new distinct atom $sat(c_i)$ for each compatibility constraint c_i, and
   (c) a new distinct atom $v_i(val_{i,j})$ for each variable v_i and value $val_{i,j}$ in the domain of v_i.
2. The rules:
   (a) Each initially active variable is mapped to a fact

   $v_i \leftarrow$

   (b) Each variable v_i and its domain $\{val_{i,1}, \ldots, val_{i,n}\}$ is mapped to the following rule:

   $v_i(val_{i,1}) \oplus \ldots \oplus v_i(val_{i,n}) \leftarrow v_i$

   (c) A compatibility constraint on variables $v_1, \ldots, v_n$ is represented using a set of rules of the form:

   $sat(c_i) \leftarrow v_1(val_{1,j}), v_2(val_{2,k}), \ldots, v_n(val_{n,l})$

   for each allowed value combination $val_{1,j}, val_{2,k}, \ldots, val_{n,l}$.

   In addition we have the following rule that forces each answer set to satisfy the compatibility constraints.

   $\leftarrow v_1, \ldots, v_n, \textbf{not } sat(c_i)$

   (d) A require activity constraint of the form **if** c **then** v is represented by a set of rules of the form

   $v \leftarrow v_1(val_{1,j}), \ldots, v_n(val_{n,k})$

   for each allowed combination (as per c) $val_{1,j}, \ldots, val_{n.k}$ of the variables $v_1, \ldots v_n$.

   (e) A not-require activity constraint of the form **if** c **then** **not** v is represented by a set of rules of the form

   $\leftarrow v, v_1(val_{1,j}), \ldots, v_n(val_{n,k})$

   for each allowed combination (as per c) $val_{1,j}, \ldots, val_{n.k}$ of the variables $v_1, \ldots v_n$.

Example 81 *[SN99]* Consider a DCSP with two variable, *package* and *sunroof*, whose domains are $\{luxury, deluxe, standard\}$ and $\{sr_1, sr_2\}$, respectively, a set of initial variables $\{package\}$ and a require activity constraint **if** *package* has value *luxury* **then** *sunroof*. Mapping it to AnsProlog$_\oplus$ results in the following program:

$package \leftarrow$
$package(luxury) \oplus package(deluxe) \oplus package(standard) \leftarrow package$
$sunroof(sr_1) \oplus sunroof(sr_2) \leftarrow sunroof$
$sunroof \leftarrow package(luxury)$ □

4.4 Combinatorial Graph Problems

In this section we consider several combinatorial NP-Complete graph problems and encode them in AnsProlog.

4.4.1 K-Colorability

The first problem that we consider is the k-colarability problem. Given a positive integer k, and a graph G, we say G is k-colorable if each vertex can be assigned one of the k colors so that no two vertex connected by an edge are assigned the same color. The decision problem is to find if a graph is k-colorable. We encode this problem by the following AnsProlog program:

1. vertices: Each vertex v of the graph is denoted by the fact

   $vertex(v)$.

2. edges: Each edge u, v of the graph is denoted by the fact

   $edge(u, v)$.

3. colors: Each of the k colors are denoted by facts

   $col(c_1)$. $\quad \ldots \quad$ $col(c_k)$.

4. Assigning colors to vertices: The following two rules assign a color to each vertex.

   $another\_color(V, C) \leftarrow vertex(V), col(C), col(D), color\_of(V, D), C \neq D$

   $color\_of(V, C) \leftarrow vertex(V), col(C), \textbf{not}\ another\_color(V, C)$

5. Constraint: The following constraint eliminate the color assignments that violate the rule "no two vertices connected by an edge have the same color".

   $\leftarrow col(C), vertex(U), vertex(V), edge(U, V), color\_of(U, C), color\_of(V, C)$

The answer sets of the above program have a 1-1 correspondence with valid color assignments.

4.4.2 Hamiltonian Circuit

A Hamiltonian circuit is a path in a graph that visits each vertex of the graph exactly once and returns to the starting vertex. The decision problem is to find if a graph has a Hamiltonian circuit. We encode this problem by the following AnsProlog program:

1. vertices: Each vertex v of the graph is denoted by the fact

   $vertex(v)$.

2. edges: Each edge u, v of the graph is denoted by the fact

   $edge(u, v)$.

3. An initial node: We arbitrarily pick one of the node u as the initial node and label it as reachable.

   $reachable(u)$.

4. For each node u of the graph we pick exactly one outgoing edge from that node and label it as *chosen*.

   $other(U,V) \leftarrow vertex(U), vertex(V), vertex(W), V \neq W, chosen(U,W).$

   $chosen(U,V) \leftarrow vertex(U), vertex(V), edge(U,V), \textbf{not}\ other(U,V).$

5. Using the following constraint we enforce that there is only one incoming edge to each vertex.

   $\leftarrow chosen(U,W), chosen(V,W), U \neq V.$

6. We define the vertices that are reachable from our initial vertex and enforce that all vertices be reachable.

   $reachable(V) \leftarrow reachable(U), chosen(U,V).$

   $\leftarrow vertex(U), \textbf{not}\ reachable(U).$

Each answer sets of the above program encodes a Hamiltonian circuit of the graph and for each Hamiltonian circuit of the graph there is at least one answer set that encodes it.

4.4.3 K-Clique

Given a number k and a graph G, we say G has a clique of size k, if there is a set of k different vertices in G such that each pair of vertices from this set is connected through an edge. The decision problem is to find if a graph has a clique of size k. We encode this problem by the following AnsProlog program:

1. vertices: Each vertex v of the graph is denoted by the fact

   $vertex(v).$

2. edges: Each edge u, v of the graph is denoted by the fact

   $edge(u,v).$

3. label: We define $k+1$ labels $(0, l_1, \ldots, l_k)$ with the intention to label each vertex using one of the labels with the restriction that each vertex has a unique label and each non-zero label is assigned to only one vertex. The k vertices with the k non-zero labels will be our candidate for constituting the clique.

   $label(0).$
   $label(l_1).$ $\quad\ldots\quad$ $label(l_k).$

4. The following two rules make sure that each vertex is assigned a unique label.

   $label\_of(V,L) \leftarrow vertex(V), label(L), \textbf{not}\ other\_label(V,L)$

   $other\_label(V,L) \leftarrow vertex(V), label(LL), label(L), label\_of(V,LL), L \neq LL$

5. To enforce that no two vertices have the same non-zero label we have the following constraint.

   $\leftarrow label\_of(V,L), label\_of(VV,L), V \neq VV, L \neq 0$

6. The following enforce that each non-zero label is assigned to some vertex.

   $assigned(L) \leftarrow label\_of(V, L)$

   $\leftarrow label(L), L \neq 0, \textbf{not } assigned(L)$

7. We now test if the chosen k vertices – the ones with non-zero labels – form a clique. If two of the chosen vertices do not have an edge between them, then the set of chosen vertices do not form a clique.

   $\leftarrow label\_of(V, L), label\_of(VV, LL), L \neq 0, LL \neq 0, V \neq VV, \textbf{not } edge(V, VV)$

Each answer sets of the above program encodes a clique of size k of the graph and for each clique of size k of the graph there is at least one answer set that encodes it.

4.4.4 Vertex Cover

Given a positive integer k and a graph G, we say G has a vertex cover of size k, if there is a set of k different vertices in the graph such that at least one end point of each edge is among these vertices. The decision problem is to find if a graph has a vertex cover of size k. This problem is encoded in AnsProlog by using the rules 1-6 of Section 4.4.3 and the following, which tests if the chosen vertices form a vertex cover.

$\leftarrow edge(V, VV), label\_of(V, L), label\_of(VV, LL), L = 0, LL = 0.$

If there is an edge such that both its vertices are not chosen (i.e., are labeled as 0), then the chosen vertices do not form a vertex cover.

4.4.5 Feedback vertex set

Given a positive integer k and a graph G we say that G has a Feedback vertex cover of size k, if there is a set S of k different vertices in G such that every cycle of G contains a vertex in S. The decision problem is to find if G has a Feedback vertex set of size k. This problem is encoded in AnsProlog by using the rules 1-6 of Section 4.4.3 and the following additional rules.

1. Defining edges of a new graph (obtained from the original graph) where we eliminate the edges that come into vertices that have a non-zero label.

   $new\_edge(X, Y) \leftarrow edge(X, Y), label\_of(Y) = 0.$

2. Defining the transitive closure of the new graph.

   $trans(X, Y) \leftarrow new\_edge(X, Y).$
   $trans(X, Y) \leftarrow new\_edge(X, Z), trans(Z, Y).$

3. Checking if the new graph has a cycle or not.

   $\leftarrow trans(X, X).$

4.4.6 Kernel

Given a directed graph $G = (V, A)$ does there exists a subset $V' \subseteq V$ such that no two vertices are joined by an arc in A and such that for every vertex $v \in V - V'$ there is a vertex $u \in V'$, for which $(u, v) \in A$.

1. vertices: Each vertex v of the graph is denoted by the fact

   $vertex(v)$.

2. edges: Each edge u, v of the graph is denoted by the fact

   $edge(u, v)$.

3. Choosing a subset of the vertices.

   $chosen(X) \leftarrow$ **not** $not\_chosen(X)$.
   $not\_chosen(X) \leftarrow$ **not** $chosen(X)$.

4. No two chosen vertices are joined by an edge.

   $\leftarrow chosen(X), chosen(Y), X \neq Y, edge(X, Y)$.

5. For all non-chosen vertices there is an edge connecting them to a chosen vertex.

   $supported(X) \leftarrow not\_chosen(X), edge(X, Y), chosen(Y)$.
   $\leftarrow not\_chosen(X),$ **not** $supported(X)$.

Each answer set of the above program encodes a kernel of the graph and for each kernel of the graph there is at least one answer set that encodes it.

4.4.7 Exercise

Represent the following combinatorial graph problems in AnsProlog.

1. Independent Set

   Given a graph $G = (V, E)$ and a positive integer $k \leq |V|$, does there exists a subset $V' \subseteq V$ such that $|V'| \geq k$ and such that no two vertices in V' are joined by an edge in E?

2. Maximal Matching

   Given a graph $G = (V, E)$ and a positive integer $k \leq |E|$, does there exists a subset $E' \subseteq E$ with $|E'| \leq k$ such that no two edges in E' share a common endpoint and every edge in $E - E'$ shares a common endpoint with some edge in E'.

4.5 Prioritized defaults and inheritance hierarchies

In Section 2.2.1 we considered AnsProlog* representation of normative statements of the form "normally elements belonging to class c have the property p" and exceptions to such statements. Often we may need more than that, such as we may have contradictory normative statements and we may have some rules specifying when one of them should be preferred over the others. We may also have sub-class information and need to encode inheritance together with ways to avoid contradictory inheritances. The early approach to such formalizations was based on directly translating such statements to AnsProlog* rules. One of the limitations to this approach was to include preferences between possibly contradictory normative statements the language of AnsProlog* had to be extended to allow specification of such preferences. In the absence of that the preferences had to be hard-coded to the rules violating the elaboration tolerance principles. In this section we discuss a recent approach where normative statements, exceptions, preferences, etc. are represented as facts, and there are general purpose rules that take this facts and reason with them. (This is similar to our planning modules where we separate the domain dependent part and the domain independent part.) The facts encode a particular domain, while the general purpose rules encode particular kind of reasoning, such as cautious and brave reasoning.

4.5.1 The language of prioritized defaults

The language consists of four kinds of facts.

1. $rule(r, l_0, [l_1, \ldots, l_m])$
2. $default(d, l_0, [l_1, \ldots, l_m])$
3. $conflict(d_1, d_2)$
4. $prefer(d_1, d_2)$

where l_i's are literals, d and d_i's are default names, and r is a rule name.

Intuitively, (1) means that if $l_1, \ldots, l_m$ (the body) are true then l_0 (the head) must be also true, and r is the label (or name) of this rule. Similarly, (2) means that *normally* if $l_1, \ldots, l_m$ are true then l_0 should be true, and d is the label (or name) of this default. The facts (3) and (4) mean d_1 and d_2 are in conflict, and d_1 is to be preferred over d_2, respectively.

A description D of a prioritized default theory is a set facts of type (1) and (2). Literals made up of the predicates $conflict$ and $prefer$ in (3) and (4) can appear in facts of the type (1) and (2). If we simply want to say $conflict(d_1, d_2)$ we may say it by having the following rule of type (1):

$rule(con(d_1, d_2), conflict(d_1, d_2), [])$.

Similarly, the fact $prefer(d_1, d_2)$ is expressed by the following rule of type (1):

$rule(pref(d_1, d_2), prefer(d_1, d_2), [])$.

4.5.2 The axioms for reasoning with prioritized defaults

Given a description D of a prioritized default theory we have additional domain independent axioms that allow us to reason with D. In these axioms we have two distinct predicates *holds* and *holds_by_default* which define when a literal holds for sure and when it holds by default, respectively. The later is a weaker conclusion than the former and may be superseded under certain conditions.

1. Axioms for non-defeasible inference: The following rules define when a literal and a set of literals *holds*. Sets are implemented using the list construct. Intuitively, a set of literals holds if each of them holds, and a particular literal l holds if there is a rule with l in its head such that each literal in the body of the rule holds.

   (a) $holds\_list([])$.

   (b) $holds\_list([H|T]) \leftarrow holds(H), holds\_list(T)$.

   (c) $holds(L) \leftarrow rule(R, L, Body), holds\_list(Body)$.

2. Axioms for defeasible inference: The following rules define when a literal and a set of literals $holds\_by\_default$. A literal that holds is also assumed to hold by default. Besides that, a set of literals holds by default if each of them holds by default, and a particular literal l holds by default if there is a default with l in its head such that each literal in the body of the rule holds by default, and the rule is not defeated and the complementary of l does not hold by default.

   (a) $holds\_by\_default(L) \leftarrow holds(L)$.

   (b) $holds\_by\_default(L) \leftarrow rule(R, L, Body), holds\_list\_by\_default(Body)$.

   (c) $holds\_by\_default(L) \leftarrow default(D, L, Body), holds\_list\_by\_default(Body),$
   $\textbf{not } defeated(D), \textbf{not } holds\_by\_default(neg(L))$.

   (d) $holds\_list\_by\_default([])$.

   (e) $holds\_list\_by\_default([H|T]) \leftarrow holds\_by\_default(H), holds\_list\_by\_default(T)$.

   The third rule above will lead to multiple answer sets if there is a chance that both l and its complement may hold by default. In one set of answer sets l will hold by default while in the other set of answer sets the complement of l will hold by default.

3. Asymmetry of the preference relation: The following rules make sure that we do not have contradictory preferences. If we do then the following rules will make our theory inconsistent.

   (a) $\neg holds(prefer(D_1, D_2)) \leftarrow holds(prefer(D_2, D_2), D_1 \neq D_2$

   (b) $\neg holds\_by\_default(prefer(D_1, D_2)) \leftarrow holds\_by\_default(prefer(D_2, D_1)), D_1 \neq D_2$

4. Defining conflict between defaults: The following rules define when conflicts between two defaults hold.

   (a) $holds(conflict(d, d')) \leftarrow default(d, l_0, B), default(d', l_0', B'), contrary(l_0, l_0'), d \neq d'$

   (b) $holds(conflict(d, d')) \leftarrow prefer(d, d'), prefer(d', d), d \neq d'$

   (c) $\neg holds(conflict(d, d)) \leftarrow$

   (d) $holds(conflict(d, d')) \leftarrow holds(conflict(D', d))$

5. Axioms that defeat defaults: We are now ready to define the notion of when a default is defeated. Different definitions of this notion lead to different kinds of reasoning.

(a) The Brave approach: In the brave approach if we have two conflicting defaults one will be considered defeated if the other is preferred over the former and the other is not itself defeated. Thus if we have two conflicting defaults such that neither is preferred over the other, then neither will be defeated, and we can apply both defaults. The rule 2(c) may then lead to two sets of answer sets each reflecting the application of one of the defaults.

i. $defeated(D) \leftarrow default(D, L, Body), holds(neg(L))$.

ii. $defeated(D) \leftarrow default(D, L, Body), default(D', L', Body'),$
$holds(conflict(D', D)), holds\_by\_default(prefer(D', D)),$
$holds\_list\_by\_default(Body'), \mathbf{not}\ defeated(D')$.

(b) The cautious approach: In the cautious approach if we have two conflicting defaults such that neither is preferred over the other, then both will be defeated. Thus we can not use 2(c) with respect to either of the defaults.

i. $defeated(D) \leftarrow default(D, L, Body), default(D', L', Body'), holds(conflict(D', D)),$
$\mathbf{not}\ holds\_by\_default(prefer(D', D)),$
$\mathbf{not}\ holds\_by\_default(prefer(D, D')),$
$holds\_list\_by\_default(Body), holds\_list\_by\_default(Body')$.

6. Uniqueness of names for defaults and rules: The following constraints make sure that rules and defaults have unique names.

(a) $\leftarrow rule(R, F, B), default(R, F', B')$.

(b) $\leftarrow rule(R, F, B), rule(R', F', B'), F \neq F'$.

(c) $\leftarrow rule(R, F, B), rule(R', F', B'), B \neq B'$.

(d) $\leftarrow default(R, F, B), default(R', F', B'), F \neq F'$.

(e) $\leftarrow default(R, F, B), default(R', F', B'), B \neq B'$.

7. Auxiliary rules: We need the following self-explanatory additional rules.

(a) $contrary(l, neg(l))$.

(b) $contrary(neg(l), l)$.

(c) $\neg holds(L) \leftarrow holds(neg(L))$

(d) $\neg holds\_by\_default(L) \leftarrow holds\_by\_default(neg(L))$

We will refer to the set of rules consisting of rules 1-4, 5(a), 6 and 7 as $\pi_{pd}^{indep.brave}$ and the set of rules 1-4, 5(b), 6 and 7 as $\pi_{pd}^{indep.cautious}$.

Example 82 *(Legal reasoning [Bre94, GS97a])* Consider the following legal rules and facts about a particular case.

1. Uniform commercial code (UCC): A security interest in goods may be perfected by taking possession of the collateral.

2. Ship mortgage act (SMA): A security interest in a ship may only be perfected by filing a financial statement.

3. Principle of Lex Posterior: A newer law has preference over an older law.

4. Principle of Lex Posterior gives precedence to laws supported by the higher authority. Federal laws have a higher authority than state laws.
5. A financial statement has not been filed.
6. John has possession of the ship johns_ship.
7. UCC is more recent than SMA.
8. SMA is a federal law.
9. UCC is a state law.

From the above we would like to find out if John's security interest in johns_ship is perfected.

The legal rules and the fact of the case can be represented by the following set $\pi^{dep}_{pd.1}$ of prioritized default facts.

1. $default(d_1, perfected, [possession])$.
2. $default(d_2, \neg perfected, [\neg filed])$.
3. $default(d_3(D_1, D_2), prefer(D_1, D_2), [more\_recent(D_1, D_2)])$.
4. $default(d_4(D_1, D_2), prefer(D_1, D_2), [federal(D_1), state(D_2)])$.
5. $\neg filed$.
6. $possession$.
7. $more\_recent(d_1, d_2)$
8. $federal(d_2)$.
9. $state(d_1)$.

The program $\pi^{indep.brave}_{pd} \cup \pi^{dep}_{pd.1}$ has two answer sets one containing $holds\_by\_default(perfected)$ and another containing $\neg holds\_by\_default(perfected)$.

But if we add the fact $rule(r, prefer(d4(D_1, D_2), d3(D_2, D_1)), [])$ to $\pi^{indep.brave}_{pd} \cup \pi^{dep}_{pd.1}$, then the resulting program has only one answer set that contains $\neg holds\_by\_default(perfected)$. □

4.5.3 Modeling inheritance hierarchies using prioritized defaults

In this section we show how general principles about reasoning with inheritance hierarchies can be expressed using prioritized defaults. An inheritance hierarchy contains information about *subclass* relationship between classes, membership information about classes, and when elements of a class normally have or normally do not have a particular property. An example of a particular inheritance hierarchy $\pi^{dep}_{pd.ih1}$ is as follows:

1. $subclass(a, b)$
2. $subclass(c, b)$
3. $is\_in(x_1, a)$

4. $is\_in(x_2, c)$
5. $default(d_1(X), has\_prop(X, p), [is\_in(X, b)])$
6. $default(d_2(X), \neg has\_prop(X, p), [is\_in(X, c)])$

To reason with inheritance hierarchies we have have three basic rules: (i) transitivity about the subclass relationship, (ii) inheritance due to the subclass relationship, and (iii) the preference principle based on specificity which says that the default property of a more specific class should be preferred over the default property of a less specific class. These three rules can be expressed by the following rules in the prioritized default language.

1. $rule(trans(C_0, C_2), subclass(C_0, C_2), [subclass(C_0, C_1), subclass(C_1, C_2).$
2. $rule(inh(X, C_1), is\_in(X, C_1), [subclass(C_0, C_1), is\_in(X, C_0)]).$
3. $rule(pref(D_1(X), D_2(X)), prefer(D_1(X), D_2(X)),$
   $[default(D_1(X), \_, [is\_in(X, A)]), default(D_2(X), \_, [is\_in(X, B)]), subclass(A, B)]).$

In addition we may have the following two defaults which play the role of closed world assumption (CWA) with respect to the predicates *is_in* and *subclass*. These defaults can be expressed by the following defaults from the prioritized default language.

1. $default(d_3(X), \neg is\_in(X), []).$
2. $default(d_4, \neg subclass(A, B), []).$

We refer to the last 5 rules as the domain independent formulation to reason with inheritance hierarchies, and denote it by $\pi^{dep}_{pd.ih.indep}$. The *indep* in the subscript refers to this independence. The *dep* in the superscript refers to the fact that inheritance hierarchies are a particular case of prioritized defaults. It is easy to check that the program $\pi^{dep}_{pd.ih.indep} \cup \pi^{dep}_{pd.ih1} \cup \pi^{indep.brave}_{pd}$ has a unique answer set containing $holds\_by\_default(has(x_1, p))$ and $holds\_by\_default(has(x_2, p))$.

4.5.4 Exercise

1. Consider extended defaults of the form $default(d, l_0, [l_1, \ldots, l_m], [l_{m+1}, \ldots, l_n])$ whose intuitive meaning is that *normally* if $l_1, \ldots, l_m$ are true and there is no reason to believe $l_{m+1}, \ldots, l_n$, then l_0 can be assumed to be true by default.

   Define AnsProlog* rules for $defeated$ and $holds\_by\_default$ to reason with such extended defaults.

2. Consider weak exceptions to default which are of the form $exception(d, l_0, [l_1, \ldots, l_m], [l_{m+1}, \ldots, l_n])$ and which intuitively mean that default d is not applicable to an object about which $l_1, \ldots, l_m$ are true and there is no reason to believe $l_{m+1}, \ldots, l_n$ to be true.

   Define AnsProlog* rules for $defeated$ to reason with prioritized defaults in presence of such weak exceptions.

3. Define conditions that guarantee consistency of descriptions in the language of prioritized defaults and the above mentioned extensions. Prove your claims. **

4.6 Implementing description logic features

4.7 Querying Databases

4.7.1 User defined aggregates and Data mining operators

4.7.2 Null values in databases

4.8 Case Studies

4.8.1 Circuit delay

4.8.2 Cryptography and Encryption

4.8.3 Characterizing monitors in policy systems

4.8.4 Product Configuration

4.8.5 Deadlock and Reachability in Petri nets

4.9 Notes and References

Representing constraint satisfaction problems is studied in [Nie99] and extended to dynamic constraint satisfaction problems in [SGN99]. Product configuration problems are studied in [SN99, NS98]. Reasoning about prioritized defaults is studied in [GS97a]. Representing null values in AnsProlog* is studied in [GT93]. Representation of graph problems in non-monotonic languages is first done in [CMMT95]. Deadlock and reachability in Petri nets is studied in [Hel99]. Representation of aggregates is studied in [WZ00b, WZ00c, WZ98].

Chapter 5

Reasoning about actions and planning in AnsProlog*

In Chapter 4 we formulated several knowledge representation and problem solving domains using AnsProlog* and focused on the program development aspect. In this chapter we consider reasoning about actions in a dynamic world and its application to plan verification, simple planning, planning with various kinds of domain constraints, observation assimilation and explanation, and diagnosis. We do a detailed and systematic formulation - in AnsProlog* – of the above issues starting from the simplest reasoning about action scenarios and gradually increasing its expressiveness by adding features such as causal constraints, and parallel execution of actions. We also prove properties of our AnsProlog* formulations using the results in Chapter 3.

Our motivation behind the choice of a detailed formulation of this domain is two fold. (i) Reasoning about actions captures both major issues of this book: knowledge representation and declarative problem solving. To reason about actions we need to formulate the frame problem whose intuitive meaning is that objects in the worlds do not normally change their properties. Formalizing this has been one of the benchmark problem of knowledge representation and reasoning formalisms. We show how AnsProlog* is up to this task. Reasoning about actions also form the ground work for planning with actions, an important problem solving task. We present AnsProlog encodings of planning such that the answer sets each encode a plan. (ii) Our second motivation is in regards to the demonstration of the usefulness of the results in Chapter 3. We analyze and prove properties of our AnsProlog* formulations of reasoning about actions and planning by using the various results in Chapter 3, and thus illustrate their usefulness. For this we also start with simple reasoning about action scenarios and then in later sections we consider more expressive scenarios.

5.1 Reasoning in the action description language $\mathcal{A}$

To systematically reason about actions, we follow the dual characterization approach where there are two separate formalizations, one in a high level English-like language with its own intuitive semantics, and the other in a logical language which in this book is AnsProlog*; so that the the second one can be validated with respect to the first one.

We choose the language $\mathcal{A}$, proposed by Gelfond and Lifschitz in [GL92], as the starting point of our high level language. This language has a simple English like syntax to express effect of actions on a world and the initial state of the world, and an automata based semantics to reason about effect of a sequence of actions on the world. Historically, $\mathcal{A}$ is remarkable for its simplicity

and has been later extended in several directions to incorporate additional features of dynamic worlds and to facilitate elaboration tolerance. We now present the language $\mathcal{A}$. Our presentation of $\mathcal{A}$ will be slightly different from [GL92] and would follow the current practices of presenting action languages through three distinct sub-languages: *domain description language*, *observation language*, and *query language*.

5.1.1 The language $\mathcal{A}$

The alphabet of the language $\mathcal{A}$ consists of two non-empty disjoint sets of symbols $\mathbf{F}$, and $\mathbf{A}$. They are called the set of fluents, and the set of actions. Intuitively, a fluent expresses the property of an object in a world, and forms part of the description of states of the world. A *fluent literal* is a fluent or a fluent preceded by $\neg$. A *state* σ is a collection of fluents. We say a fluent f holds in a state σ if $f \in \sigma$. We say a fluent literal $\neg f$ holds in σ if $f \notin \sigma$.

For example, in the blocks world domain where there are many blocks in a table, some of the fluents are: $ontable(a)$, meaning that block a is on the table; $on(a, b)$, meaning that block a is on block b; $clear(a)$, meaning that the top of block a is clear; $handempty$, meaning that the hand is empty. These fluents are also fluent literals, and some other examples of fluent literals are: $\neg on(a, b)$, meaning that block a is not on top of block b; $\neg clear(b)$, meaning that the top of block b is not clear. An example of state where we have only two blocks a and b is $\sigma = \{ontable(b), on(a, b), clear(a), handempty\}$. Besides the fluent literals that are in σ, we also have that $\neg ontable(a), \neg on(b, a)$, and $\neg clear(b)$ hold in σ. Examples of actions in this domain are: $pickup(a)$, which picks up block a from the table; $putdown(b)$, which puts down block b on the table; $stack(a, b)$, which stacks block a on top of block b; and $unstack(a, b)$, which unstacks block a from the top of block b. Actions, when successfully executed change the state of the world. For example, the action $unstack(a, b)$ when performed in the state σ results in the state $\sigma' = \{ontable(b), clear(b)\}$.

Situations are representations of history of action execution. In the initial situation no action has been executed and it is represented by the empty list [ ]. The situation $[a_n, \ldots, a_1]$ corresponds to the history where action a_1 is executed in the initial situation, followed by a_2, and so on until a_n. There is a simple relation between situations and states. In each situation s certain fluents are true and certain others are false, and this 'state of the world' is the state corresponding to s. In the above example if the situation [ ] correspond to the state σ, then the situation $[unstack(a, b)]$ correspond to the state σ'. Since the state of the world could be same for different histories, different situations may correspond to the same state. Thus the situations [ ], $[stack(a, b), unstack(a, b)]$, $[stack(a, b), unstack(a, b), stack(a, b), unstack(a, b)]$, ... correspond to σ, and the situations $[unstack(a, b)]$, $[unstack(a, b), stack(a, b), unstack(a, b)]$, ... correspond to σ'.

We now present the three sub-languages of $\mathcal{A}$.

1. Domain description language: The domain description language is used to succinctly express the transition between states due to actions. A straight forward representation of this transition would be a 3-dimensional table of the size mn^2 where m is the number of actions and n is the number of states. Since the number of states can be of the order of $2^{|\mathbf{F}|}$, such a representation is is often shunned. Besides the space concerns, another concern is the issue of elaboration tolerance: how easy it is to update the representation in case we have elaborations, such as a few more fluents or a few more actions. The domain description sub-language of $\mathcal{A}$ is concerned with both issues.

A domain description D consists of effect propositions of the following form:

$$a \textbf{ causes } f \textbf{ if } p_1, \ldots, p_n, \neg q_1, \ldots, \neg q_r \tag{5.1.1}$$

where a is an action, f is a fluent literal, and $p_1, \ldots, p_n, q_1, \ldots, q_r$ are fluents. Intuitively, the above effect proposition means that if the fluent literals $p_1, \ldots, p_n, \neg q_1, \ldots, \neg q_r$ hold in the state corresponding to a situation s then in the state corresponding to the situation reached by executing a in s (denoted by $[a|s]$) the fluent literal f must hold. If both n and r are equal to 0 in (5.1.1) then we simply write:

$$a \textbf{ causes } f \tag{5.1.2}$$

Moreover, we often condense a set of effect propositions $\{a \textbf{ causes } f_1, \ldots, a \textbf{ causes } f_m\}$ by

$$a \textbf{ causes } f_1, \ldots, f_m \tag{5.1.3}$$

We also allow a, f and p_i's in (5.1.1) to have schema variables. Such an effect proposition with schema variables is a short-hand for the set of ground effect propositions obtained by substituting the variables with objects in the domain.

Example 83 Consider the blocks world domain. Let us assume that we have three blocks a, b, and c and a table. The effect of the action $pickup(X)$ can be expressed by the following effect propositions with schema variables:

$pickup(X)$ **causes** $\neg ontable(X), \neg handempty, \neg clear(X), holding(X)$

For this domain with three blocks a, b, and c, the above is a short-hand for the following three effect propositions with out any schema variables.

$pickup(a)$ **causes** $\neg ontable(a), \neg handempty, \neg clear(a), holding(a)$
$pickup(b)$ **causes** $\neg ontable(b), \neg handempty, \neg clear(b), holding(b)$
$pickup(c)$ **causes** $\neg ontable(c), \neg handempty, \neg clear(c), holding(c)$ □

As mentioned earlier, the role of effect propositions is to define a transition function from states and actions to states. Given a domain description D, such a transition function Φ should satisfy the following property. For all actions a, fluents g, and states σ:

- if D includes an effect proposition of the form (5.1.1) where f is the fluent g and $p_1, \ldots, p_n, \neg q_1, \ldots, \neg q_r$ hold in σ then $g \in \Phi(a, \sigma)$;
- if D includes an effect proposition of the form (5.1.1) where f is a negative fluent literal $\neg g$ and $p_1, \ldots, p_n, \neg q_1, \ldots, \neg q_r$ hold in σ then $g \notin \Phi(a, \sigma)$;
- if D does not include such effect propositions, then $g \in \Phi(a, \sigma)$ iff $g \in \sigma$.

For a given domain description D, there is at most one transition function that satisfies the above properties. If such a transition function exists, then we say D is consistent, and refer to its transition function by Φ_D.

An example of an inconsistent domain description D_1 consists of the following.

a **causes** f
a **causes** $\neg f$.

In the rest of this section, we only consider consistent domain descriptions.

2. Observation language: A set of observations O consists of value propositions of the following form:

$$f \textbf{ after } a_1, \ldots, a_m \tag{5.1.4}$$

where f is a fluent literal and $a_1, \ldots, a_m$ are actions. Intuitively, the above value proposition means that if $a_1, \ldots, a_m$ would be executed in the initial situation then in the state corresponding to the situation $[a_m, \ldots, a_1]$, f would hold.

When $a_1, \ldots, a_m$ is an empty sequence, we write the above as follows:

$$\textbf{initially } f \tag{5.1.5}$$

In this case the intuitive meaning is that f holds in the state corresponding to the initial situation. Here also we condense a set of value propositions of the form
$\{$ **initially** $f_1, \ldots,$ **initially** $f_k\}$ by

$$\textbf{initially } f_1, \ldots, f_k \tag{5.1.6}$$

Given a consistent domain description D the set of observations O are used to determine the state corresponding to the initial situations, refereed to as the *initial state* and denoted by σ_0. While D determines a unique transition function, an O may not always lead to a unique initial state.

We say σ_0 is an initial state corresponding a consistent domain description D and a set of observations O, if for all observations of the form (5.1.4) in O, the fluent literal f holds in the state $\Phi(a_m, \Phi(a_{m-1}, \ldots \phi(a_1, \sigma_0)\ldots))$. (We will denote this state by $[a_m, \ldots, a_1]\sigma_0$.) We then say that (σ_0, Φ_D) *satisfies* O.

Given a consistent domain description D and a set of observations O, we refer to the pair (Φ_D, σ_0), where Φ_D is the transition function of D and σ_0 is an initial state corresponding to D and O, as a *model* of D, O. We say D, O is *consistent* if it has a model and say it is *complete* if it has a unique model.

Example 84 Consider the Yale turkey shooting domain where we have the fluents *alive*, *loaded* and actions *load* and *shoot*. The effect of the actions can be expressed by the following domain description D:

load **causes** *loaded*
shoot **causes** $\neg$*alive* **if** *loaded*

Suppose we have the set of observations $O = \{$ **initially** *alive*$\}$. There are two initial states corresponding to D and O: $\sigma_0 = \{alive\}$ and $\sigma_0' = \{alive, loaded\}$. □

3. Query language: Queries also consist of value propositions of the form (5.1.4).

   We say a consistent domain description D in presence of a set of observations O entails a query Q of the form (5.1.4) if for all initial states σ_0 corresponding to D and O, the fluent literal f holds in the state $[a_m, \ldots, a_1]\sigma_0$. We denote this as $D \models_O Q$.

   Example 85 Consider D and O from Example 84. An example of a query Q in this domain is $\neg alive$ **after** *shoot*. Since there are two initial states, σ_0 and σ_0', corresponding to D and O, and while $\neg alive$ holds in $[shoot]\sigma_0'$, it does not hold in $[shoot]\sigma_0$. Hence, $D \not\models_O Q$.

   On the other hand $D \models_O \neg alive$ **after** *load*, *shoot* and also if $O' = \{$ **initially** *alive*; **initially** *loaded*$\}$, then $D \models_{O'} Q$. □

We can use the above formulation to do several different kinds of reasoning about actions such as predicting the future from information about the initial state, assimilating observations to deduce about the initial state, a combination of both, and planning.

- **Temporal projection**: In temporal projection, the observations are only of the form (5.1.5) and the only interest is to make conclusions about the (hypothetical) future. There are two particular cases of temporal projection: when the observations give us a complete picture of the initial state, and when they do not. The former is referred to as the initial state being complete, and is formally defined below.

  A set of observations O is said to be *initial state complete*, if O only consists of propositions of the form (5.1.5) and for all fluents f, either **initially** f is in O, or **initially** $\neg f$ is in O, but not both. For example, the set of observations O' in Example 85 is initial state complete.

  In the later case we say an initial state complete set of observations $\hat{O}$ extends O if $O \subseteq \hat{O}$. For example, the set of observations O' in Example 85 extends the set of observations O in Example 84. Moreover, the entailment in Examples 84 and 85 are examples of temporal projection.

- **Reasoning about the initial situation**: In reasoning about the initial situation, the observations can be about any situation, but the queries are only about the initial state. The following example illustrates this.

  Example 86 Let $O_1 = \{$ **initially** *alive*; $\neg alive$ **after** *shoot*$\}$ and D be the domain description from Example 84. In this case there is exactly one initial state, $\{alive, loaded\}$, corresponding to D and O_1 and we have $D \models_{O_1}$ **initially** *loaded*. Hence, from the observations O_1 we can reason backwards and conclude that the gun was initially loaded. □

- **Observation assimilation**: This generalizes temporal projection and reasoning about the initial situation. In this case both the observations and the queries can be about any situation.

  Example 87 Let $O_2 = \{$ **initially** *alive*; *loaded* **after** *shoot*$\}$ and D be the domain description from Example 84. In this case there is also exactly one initial state, $\{alive, loaded\}$, corresponding to D and O_2 and we have $D \models_{O_2} \neg alive$ **after** *shoot*. Hence, assimilating the observations O_2 we can conclude that if shooting happens in the initial situation then after that the turkey will not be alive. □

- **Planning**: In case of planning we are given a domain description D , a set of observations about the initial state O, and a collection of fluent literals $G = \{g_1, \ldots, g_l\}$, which we will refer to as a goal. We are required to find a sequence of actions $a_1, \ldots, a_n$ such that for all $1 \leq i \leq l$, $D \models_O g_i$ **after** $a_1, \ldots, a_n$. We then say that $a_1, \ldots, a_n$ is a plan for goal G (or that achieves goal G) with respect to D and O.

  If D, O have multiple models each with a different initial state then if we are able to find a sequence of actions $a_1, \ldots, a_n$ such that for all $1 \leq i \leq l$, $D \models_O g_i$ **after** $a_1, \ldots, a_n$, the sequence $a_1, \ldots, a_n$ is referred to as a conformant plan.

  Example 88 Recall that $O = \{$ **initially** *alive*$\}$ and $O' = \{$ **initially** *alive*; **initially** *loaded*$\}$ in Examples 84 and 85. Let D be the domain description from Example 84. Now suppose $G = \{\neg alive\}$. In that case *shoot* is a plan for G with respect to D and O'; *shoot* is not a plan for G with respect to D and O; and *load*; *shoot* is a plan for G with respect to D and O. □

We now present several simple results relating models of domain descriptions and observations. These results will be later used in analyzing the correctness of AnsProlog* encodings of the various kinds of reasoning about actions.

Lemma 5.1.1 Let D be a consistent domain description and O be an initial state complete set of observations such that D, O is consistent. D, O has a unique model (σ_0, Φ_D), where $\sigma_0 = \{f$: **initially** $f \in O\}$. □

Lemma 5.1.2 Let D be a consistent domain description and O be a set of observations about the initial state such that D, O is consistent. (σ_0, Φ_D) is a model of D, O iff $\{f$: **initially** $f \in O\} \subseteq \sigma_0$ and $\sigma_0 \cap \{f$: **initially** $\neg f \in O\} = \emptyset$. □

Corollary 2 Let D be a consistent domain description and O be a set of observations about the initial state such that D, O is consistent. (σ_0, Φ_D) is a model of D, O iff there exist an extension $\hat{O}$ of O such that (σ_0, Φ_D) is the unique model of $D, \hat{O}$. □

Corollary 3 Let D be a consistent domain description and O be a set of observations about the initial state such that D, O is consistent. For any fluent f and sequence of actions $a_1, \ldots, a_n$, $D \models_O (\neg)f$ **after** $a_1, \ldots, a_n$ iff $\forall \hat{O}$ such that $\hat{O}$ extends O, $D \models_{\hat{O}} (\neg)f$ **after** $a_1, \ldots, a_n$. □

Lemma 5.1.3 Let D be a consistent domain description and O be a set of observations. $M = (\sigma_0, \Phi_D)$ is a model of D, O iff M is the model of D, O_M, where $O_M = \{$ **initially** f : $f \in \sigma_0\} \cup \{$ **initially** $\neg f$: $f \notin \sigma_0\}$. □

In the next several sections (Section 5.1.2 – 5.1.6) we give AnsProlog* formulations that (partially) compute the entailment relation $\models_O$, given a domain description D and a set of observations O. The programs that we present ($\pi_1(D, O)$ – $\pi_6(D, O)$) are all sorted programs as introduced in Section 3.6.1, which means that variables in these programs are grounded based on their sorts. We have three sorts: *situations*, *fluents* and *actions*, and variables of these sorts are denoted by $S, S' \ldots$; $F, F', \ldots$; and $A, A', \ldots$, respectively. The sort of the arguments of the function symbols and predicates in the programs $\pi_1(D, O)$ – $\pi_6(D, O)$ are clear from the context. In these formulations we use the notation $[a_n, \ldots, a_1]$ to denote the situation $res(a_n \ldots res(a_1, s_0) \ldots)$. The following

table gives a summary of which AnsProlog* sub-class the programs $\pi_1(D,O)$ – $\pi_6(D,O)$ belong to and what kind of reasoning about actions they do.

Programs	Class	Applicability
π_1	AnsProlog	temporal projection from a complete initial state
π_2	AnsProlog$^\neg$	temporal projection from a complete initial state
π_3	AnsProlog$^\neg$	temporal projection from (possibly incomplete) initial state
π_4	AnsProlog$^\neg$	temporal projection, backward reasoning
π_5	AnsProlog$^\neg$	temporal projection, assimilation of observations
π_6	AnsProlog$^{\neg, or}$	temporal projection, assimilation of observations

5.1.2 Temporal projection and its acyclicity in an AnsProlog formulation: π_1

In this section we give an AnsProlog formulation of temporal projection. In particular, given a domain description D and a set of observations O which is initial state complete, we construct an AnsProlog program $\pi_1(D,O)$ and show the correspondence between query entailment from D,O and entailment in $\pi_1(D,O)$. The AnsProlog program $\pi_1(D,O)$ consists of three parts π_1^{ef}, π_1^{obs}, and π_1^{in} as defined below.

1. Translating effect propositions: The effect propositions in D are translated as follows and are collectively referred to as π_1^{ef}.

   For every effect proposition of the form (5.1.1) if f is a fluent then π_1^{ef} contains the following rule:

   $holds(f, res(a,S)) \leftarrow holds(p_1,S), \ldots, holds(p_n,S)$, **not** $holds(q_1,S), \ldots,$ **not** $holds(q_r,S)$.

   else, if f is the negative fluent literal $\neg g$ then π_1^{ef} contains the following rule:

   $ab(g,a,S) \leftarrow holds(p_1,S), \ldots, holds(p_n,S)$, **not** $holds(q_1,S), \ldots,$ **not** $holds(q_r,S)$.

2. Translating observations: The value propositions in O are translated as follows and are collectively referred to as π_1^{obs}.

   For every value proposition of the form (5.1.5) if f is a fluent then π_1^{obs} contains the following rule:

   $holds(f, s_0) \leftarrow$.

   Note that if f is a negative fluent literal, we do not add any rule corresponding to it to π_1^{obs}.

3. Inertia rules: Besides the above we have the following inertia rule referred to by π_1^{in}.

   $holds(F, res(A,S)) \leftarrow holds(F,S)$, **not** $ab(F,A,S)$.

Example 89 Consider D from Example 84, and let $O_3 = \{$ **initially** $alive$; **initially** $\neg loaded\}$. The program $\pi_1(D,O_3)$, which is same as the program in part (1) of Section 2.2.2, consists of the following:

$holds(loaded, res(load,S)) \leftarrow$.
$ab(alive, shoot, S) \leftarrow holds(loaded,S)$.

$holds(alive, s_0) \leftarrow .$

$holds(F, res(A, S)) \leftarrow holds(F, S), \mathbf{not}\ ab(F, A, S).$

It is easy to see that $\pi_1(D, O_3) \models holds(alive, [load])$, $\pi_1(D, O_3) \models holds(loaded, [load])$, and $\pi_1(D, O_3) \models \neg holds(alive, [shoot, load])$. (Note that $\pi_1(D, O_3) \not\models \neg holds(alive, [shoot, load])$.) □

We now analyze $\pi_1(D, O)$ and relate entailments with respect to $\pi_1(D, O)$ with queries entailed by D, O.

Proposition 66 Let D be a consistent domain description and O be an initial state complete set of observations. Then $\pi_1(D, O)$ is acyclic. □

Proof:
To show $\pi_1(D, O)$ to be acyclic we need to give a level mapping $|\ |$, to atoms in the language of $\pi_1(D, O)$ so that the acyclicity condition holds. To facilitate that we first give a level mapping to terms.

We assign $|s_0| = 1$, and for any actions a and situation s, $|res(a, s)| = |s| + 1$. For any fluent f, action a, and situation s, we assign $|holds(f, s)| = 2 \times |s|$ and $|ab(f, a, s)| = 2 \times |s| + 1$. It is easy to see that this level assignment satisfies the acyclicity condition for the AnsProlog program $\pi_1(D, O)$. □

Corollary 4 Let D be a consistent domain description and O be an initial state complete set of observations. Then $\pi_1(D, O)$ has the following properties.

1. It has a unique answer set. (Let us call it M.)
2. $A \in M$ iff $Comp(\pi_1(D, O)) \models A$.
3. For all variable-free atoms A, $A \in M$ iff there exists an SLDNF-refutation of $\pi_1(D, O) \cup \leftarrow A$.

□

Lemma 5.1.4 Let D be a consistent domain description and O be an initial state complete set of observations such that D, O is consistent. Let (σ_0, Φ_D) be the unique model of D, O and M be the answer set of $\pi_1(D, O)$. Let f be a fluent.

$f \in \Phi_D(a_n, \ldots, \Phi_D(a_1, \sigma_0)\ldots)$ iff $holds(f, [a_n, \ldots, a_1]) \in M$. □

Proof: We prove this by induction on the length of the action sequence.

Base Case: $n = 0$.
$\Longrightarrow$ $f \in \sigma_0$ implies that **initially** $f \in D$ which implies that $holds(f, s_0) \leftarrow . \in \pi_1(D, O)$. This implies $holds(f, s_0) \in M$.

$\Longleftarrow$ $holds(f, s_0) \in M$ implies that $holds(f, s_0) \leftarrow . \in \pi_1(D, O)$, as no other rule in $\pi_1(D, O)$ has $holds(f, s_0)$ in its head. This implies that **initially** $f \in D$ which implies that $f \in \sigma_0$.

Inductive case: Assuming the lemma hold for all $i < n$, let us show that it holds for $i = n$. Let us denote $\Phi_D(a_i, \ldots, \Phi_D(a_1, \sigma_0)\ldots)$ by σ_i.

($\Longrightarrow$) $f \in \sigma_n$ implies two cases:
(i) $f \in \sigma_{n-1}$ and there does not exist an effect proposition of the form a **causes** $\neg f$ **if** , $p_1, \ldots, p_k$, $\neg q_1, \ldots, \neg q_r$ such that $p_1, \ldots, p_k, \neg q_1, \ldots, \neg q_r$ hold in σ_{n-1}.

(ii) there exists an effect proposition of the form a **causes** f **if** $p_1, \ldots, p_k, \neg q_1, \ldots, \neg q_r$ such that $p_1, \ldots, p_k, \neg q_1, \ldots, \neg q_r$ hold in σ_{n-1}.

Case (i) : By induction hypothesis, $f \in \sigma_{n-1}$ implies $holds(f, [a_{n-1}, \ldots, a_1]) \in M$. We will now argue that the second part of (i) implies that $ab(f, a_n, [a_{n-1}, \ldots, a_1]) \notin M$. Lets assume the contrary. I.e., $ab(f, a_n, [a_{n-1}, \ldots, a_1]) \in M$. Then by part (b) of Proposition 20 there must be a rule in $\pi_1(D, O)$ whose head is $ab(f, a_n, [a_{n-1}, \ldots, a_1])$ and whose body evaluates to true with respect to M. That means there must be an an effect proposition of the form a **causes** $\neg f$ **if** , $p_1, \ldots, p_k$, $\neg q_1, \ldots, \neg q_r$ such that $\{holds(p_1, [a_{n-1}, \ldots, a_1]) \ldots, holds(p_k, [a_{n-1}, \ldots, a_1])\} \subseteq M$ and $\{holds(q_1, [a_{n-1}, \ldots, a_1]) \ldots, holds(q_r, [a_{n-1}, \ldots, a_1])\} \cap M = \emptyset$. By induction hypothesis, this means there exists an effect proposition of the form a **causes** $\neg f$ **if** , $p_1, \ldots, p_k$, $\neg q_1, \ldots, \neg q_r$ such that $p_1, \ldots, p_k, \neg q_1, \ldots, \neg q_r$ hold in σ_{n-1}. This contradicts the second part of (i). Hence our assumption that $ab(f, a_n, [a_{n-1}, \ldots, a_1]) \in M$ must be wrong. Therefore, $ab(f, a_n, [a_{n-1}, \ldots, a_1]) \notin M$. Now using this, the earlier conclusion that $holds(f, [a_{n-1}, \ldots, a_1]) \in M$, the inertia rule of the program $\pi_1(D, O)$ and part (a) of Proposition 20 we can conclude that $holds(f, [a_n, \ldots, a_1]) \in M$.

Case (ii) : Using the induction hypothesis, the first rule of of the program π_1^{ef} and part (a) of Proposition 20 we can conclude that $holds(f, [a_n, \ldots, a_1]) \in M$.

($\Longleftarrow$) From $holds(f, [a_n, \ldots, a_1]) \in M$, using part (b) of Proposition 20 there are two possibilities:

(a) $holds(f, [a_{n-1}, \ldots, a_1]) \in M$ and $ab(f, a_n, [a_{n-1}, \ldots, a_1]) \notin M$.

(b) there exists an effect proposition of the form a **causes** f **if** $p_1, \ldots, p_k, \neg q_1, \ldots, \neg q_r$ such that $\{holds(p_1, [a_{n-1}, \ldots, a_1]) \ldots, holds(p_k, [a_{n-1}, \ldots, a_1])\} \subseteq M$ and $\{holds(q_1, [a_{n-1}, \ldots, a_1]) \ldots, holds(q_r, [a_{n-1}, \ldots, a_1])\} \cap M = \emptyset$.

(case a): By induction hypothesis, $holds(f, [a_{n-1}, \ldots, a_1]) \in M$ implies that $f \in \sigma_{n-1}$. We will now argue that $ab(f, a_n, [a_{n-1}, \ldots, a_1]) \notin M$ implies that there does not exist an effect proposition of the form a **causes** $\neg f$ **if** , $p_1, \ldots, p_k$, $\neg q_1, \ldots, \neg q_r$ such that $p_1, \ldots, p_k, \neg q_1, \ldots, \neg q_r$ hold in σ_{n-1}. Suppose to the contrary. In that case using induction hypothesis we will have $\{holds(p_1, [a_{n-1}, \ldots, a_1]) \ldots, holds(p_k, [a_{n-1}, \ldots, a_1])\} \subseteq M$ and $\{holds(q_1, [a_{n-1}, \ldots, a_1]) \ldots, holds(q_r, [a_{n-1}, \ldots, a_1])\} \cap M = \emptyset$. Now using part (a) of Proposition 20 we will be forced to conclude $ab(f, a_n, [a_{n-1}, \ldots, a_1]) \in M$, which results in a contradiction. Hence, there does not exist an effect proposition of the form a **causes** $\neg f$ **if** , $p_1, \ldots, p_k$, $\neg q_1, \ldots, \neg q_r$ such that $p_1, \ldots, p_k, \neg q_1, \ldots, \neg q_r$ hold in σ_{n-1}. This together with the fact that $f \in \sigma_{n-1}$ implies that $f \in \sigma_n$.

(case b): By using the induction hypothesis we can conclude that there exists an effect proposition of the form a **causes** f **if** $p_1, \ldots, p_k, \neg q_1, \ldots, \neg q_r$ such that $p_1, \ldots, p_k, \neg q_1, \ldots, \neg q_r$ hold in σ_{n-1}. This implies that $f \in \sigma_n$.

(end of proof) □

Proposition 67 Let D be a consistent domain description and O be an initial state complete set of observations such that D, O is consistent. Let f be a fluent.

(i) $D \models_O f$ **after** $a_1, \ldots, a_n$ iff $\pi_1(D, O) \models holds(f, [a_n, \ldots, a_1])$.

(ii) $D \models_O \neg f$ **after** $a_1, \ldots, a_n$ iff $\pi_1(D, O) \not\models holds(f, [a_n, \ldots, a_1])$. □

Proof: Follows from Proposition 66, and Lemmas 5.1.1 and 5.1.4. □

5.1.3 Temporal projection in an AnsProlog$^\neg$ formulation: π_2

In this section we give an AnsProlog$^\neg$ formulation of temporal projection in presence of a complete initial state which can be also used with additional rules to do temporal projection when the initial state is incomplete. Given a domain description D and a set of observations O which is initial state complete, we construct an AnsProlog$^\neg$ program $\pi_2(D, O)$ consisting of three parts π_2^{ef}, π_2^{obs}, and π_2^{in} as defined below.

1. Translating effect propositions: The effect propositions in D are translated as follows and are collectively referred to as π_2^{ef}.

   For every effect proposition of the form (5.1.1) if f is a fluent then π_2^{ef} contains the following rules:

   $holds(f, res(a, S)) \leftarrow holds(p_1, S), \ldots, holds(p_n, S),\ \neg holds(q_1, S), \ldots, \neg holds(q_r, S).$

   $ab(f, a, S) \leftarrow holds(p_1, S), \ldots, holds(p_n, S),\ \neg holds(q_1, S), \ldots, \neg holds(q_r, S).$

   else, if f is the negative fluent literal $\neg g$ then π_2^{ef} contains the following rules:

   $\neg holds(g, res(a, S)) \leftarrow holds(p_1, S), \ldots, holds(p_n, S),\ \neg holds(q_1, S), \ldots, \neg holds(q_r, S).$

   $ab(g, a, S) \leftarrow holds(p_1, S), \ldots, holds(p_n, S),\ \neg holds(q_1, S), \ldots, \neg holds(q_r, S).$

2. Translating observations: The value propositions in O are translated as follows and are collectively referred to as π_2^{obs}.

   For every value proposition of the form (5.1.5) if f is a fluent then π_2^{obs} contains the following rule:

   $holds(f, s_0) \leftarrow.$

   else, if f is the negative fluent literal $\neg g$ then π_2^{obs} contains the following rule:

   $\neg holds(g, s_0) \leftarrow.$

3. Inertia rules: Besides the above we have the following inertia rules referred to as π_2^{in}.

   $holds(F, res(A, S)) \leftarrow holds(F, S), \mathbf{not}\ ab(F, A, S).$

   $\neg holds(F, res(A, S)) \leftarrow \neg holds(F, S), \mathbf{not}\ ab(F, A, S).$

Example 90 Consider D from Example 84, and $O_3 = \{$ **initially** $alive$; **initially** $\neg loaded\}$ from Example 89. The program $\pi_2(D, O_3)$ consists of the following:

$holds(loaded, res(load, S)) \leftarrow.$
$ab(loaded, load, S) \leftarrow.$
$\neg holds(alive, res(shoot, S)) \leftarrow holds(loaded, S).$
$ab(alive, shoot, S) \leftarrow holds(loaded, S).$

$holds(alive, s_0) \leftarrow .$
$\neg holds(loaded, s_0) \leftarrow .$

$holds(F, res(A, S)) \leftarrow holds(F, S), \textbf{not } ab(F, A, S).$
$\neg holds(F, res(A, S)) \leftarrow \neg holds(F, S), \textbf{not } ab(F, A, S).$

It is easy to see that $\pi_2(D, O_3) \models^* holds(alive, [load])$, $\pi_2(D, O_3) \models^* holds(loaded, [load])$, and $\pi_2(D, O_3) \models^* \neg holds(alive, [shoot, load])$.
(Recall from Example 89 that $\pi_1(D, O_3) \not\models \neg holds(alive, [shoot, load])$.) □

We now analyze $\pi_2(D, O)$ and relate entailments with respect to $\pi_2(D, O)$ with queries entailed by D, O.

Lemma 5.1.5 Let D be a consistent domain description and O be an initial state complete set of observations. Then $\pi_2(D, O)^+$ is acyclic. □

Proof:
To show $\pi_2(D, O)^+$ to be acyclic we need to give a level mapping $| \ |$, to atoms in the language of $\pi_2(D, O)^+$ so that the acyclicity condition holds. To facilitate that we first give a level mapping to terms.

We assign $|s_0| = 1$, and for any actions a and situation s, $|res(a, s)| = |s|+1$. For any fluent f, action a, and situation s, we assign $|holds(f, s)| = 2\times|s|$, $|holds'(f, s)| = 2\times|s|$ and $|ab(f, a, s)| = 2\times|s|+1$. It is easy to see that this level assignment satisfies the acyclicity condition for the AnsProlog program $\pi_2(D, O)^+$. □

Lemma 5.1.6 Let D be a consistent domain description and O be an initial state complete set of observations. Let M be the answer set of $\pi_2(D, O)^+$.

For any fluent f, and sequence of actions $a_1, \ldots, a_n$, at least one, but not both of $holds(f, [a_n, \ldots, a_1])$ and $holds'(f, [a_n, \ldots, a_1])$ belong to M. □

Proof: (sketch)
This can be proved by induction on n. The base case ($n = 0$) holds because of the translation π_2 and the assumption that O is an initial state complete set of observations. Assuming that the conclusion of the lemma holds for all $i < n$, we will argue that it holds for n. To show that at least one of $holds(f, [a_n, \ldots, a_1])$ and $holds'(f, [a_n, \ldots, a_1])$ belongs to M we can use the induction hypothesis and consider the two cases: (i) $holds(f, [a_{n-1}, \ldots, a_1]) \in M$; (ii) $holds'(f, [a_{n-1}, \ldots, a_1]) \in M$ and argue that in either case our desired conclusion holds. The argument is that if we are unable to use the inertia rules, then $ab(f, a_n, [a_{n-1}, \ldots, a_1]) \notin M$, and that means there must be an effect axiom which either causes f or causes $\neg f$ and whose preconditions hold in the situation $[a_{n-1}, \ldots, a_1]$.

Next we need to show that only one of $holds(f, [a_n, \ldots, a_1])$ and $holds'(f, [a_n, \ldots, a_1])$ belong to M. By the induction hypothesis we can show that they do not both hold because of two inertia rules. Because of the consistency of D, they can not both hold because of two effect axioms. Finally, they can not both hold because of one effect axioms and one inertia rule as the presence of $ab(f, a_n, [a_{n-1}, \ldots, a_1])$ will block the inertia rule. □

Proposition 68 Let D be a consistent domain description and O be an initial state complete set of observations such that D, O is consistent. Then $\pi_2(D, O)$ has a unique answer set which is consistent. □

Proof: Follows from Lemma 5.1.6 and Proposition 6. If M is the answer set of $\pi_2(D, O)^+$, then the answer set of $\pi_2(D, O)$ is the following:

$M \setminus \{holds'(f, s) \ : \ holds'(f, s) \in M\} \cup \{\neg holds(f, s) \ : \ holds'(f, s) \in M\}$ □

Lemma 5.1.7 Let D be a consistent domain description and O be an initial state complete set of observations such that D, O is consistent. Let (σ_0, Φ_D) be the unique model of D, O and M be the answer set of $\pi_2(D, O)$. Let f be a fluent.

$f \in \Phi_D(a_n, \ldots, \Phi_D(a_1, \sigma_0)\ldots)$ iff $holds(f, [a_n, \ldots, a_1]) \in M$. □

Proof (sketch): Can be shown using induction on n. The proof is similar to the proof of Lemma 5.1.4.

Proposition 69 Let D be a consistent domain description and O be an initial state complete set of observations such that D, O is consistent. Let f be a fluent.

(i) $D \models_O f$ **after** $a_1, \ldots, a_n$ iff $\pi_2(D, O) \models holds(f, [a_n, \ldots, a_1])$.

(ii) $D \models_O \neg f$ **after** $a_1, \ldots, a_n$ iff $\pi_2(D, O) \models \neg holds(f, [a_n, \ldots, a_1])$. □

Proof: Follows from Lemma 5.1.7, Proposition 68, and Lemma 5.1.1.

5.1.4 Temporal projection in AnsProlog$^\neg$ in presence of incompleteness: π_3

We now present a modification of $\pi_2(D, O)$ from the previous section so that it can reason correctly when the initial state is incomplete. To show what goes wrong if we reason with $\pi_2(D, O)$, when O is not initial state complete, consider the following example.

Example 91 Let D consist of the following two propositions:

a **causes** f **if** p

a **causes** f **if** $\neg p$

and $O = \{$ **initially** $\neg f\}$.

The program $\pi_2(D, O)$ consists of the following:

$holds(f, res(a, S)) \leftarrow holds(p, S)$.
$ab(f, a, S) \leftarrow holds(p, S)$.
$holds(f, res(a, S)) \leftarrow \neg holds(p, S)$.
$ab(f, a, S) \leftarrow \neg holds(p, S)$.

$\neg holds(f, s_0) \leftarrow$.

$holds(F, res(A, S)) \leftarrow holds(F, S)$, **not** $ab(F, A, S)$.
$\neg holds(F, res(A, S)) \leftarrow \neg holds(F, S)$, **not** $ab(F, A, S)$.

It is easy to see that $\pi_2(D, O) \models^* \neg holds(f, [a])$, while $D \models_O f$ **after** a. Thus $\pi_2(D, O)$ makes a wrong conclusion. □

Example 92 Consider D from Example 84, and $O_4 = \{$ **initially** $alive\}$. The program $\pi_2(D, O_4)$ consists of the following:

$holds(loaded, res(load, S)) \leftarrow$.
$ab(loaded, load, S) \leftarrow$.
$\neg holds(alive, res(shoot, S)) \leftarrow holds(loaded, S)$.
$ab(alive, shoot, S) \leftarrow holds(loaded, S)$.

$holds(alive, s_0) \leftarrow$.

$holds(F, res(A, S)) \leftarrow holds(F, S), \mathbf{not}\ ab(F, A, S).$
$\neg holds(F, res(A, S)) \leftarrow \neg holds(F, S), \mathbf{not}\ ab(F, A, S).$

It is easy to see that $\pi_2(D, O_4) \models^{\star} holds(alive, [shoot])$, while $D \not\models_{O_4} alive$ **after** $shoot$. □

In both of the above examples the fault lies in not being able to block the inertia rules. The new program $\pi_3(D, O)$ consists of π_2^{obs} and π_2^{in} from the previous section and π_3^{ef} as defined below, where we modify the definition of ab so that it can block the inertia in cases such as the above examples.

1. Translating effect propositions: The effect propositions in D are translated as follows and are collectively referred to as π_3^{ef}.

   For every effect proposition of the form (5.1.1) if f is a fluent then π_3^{ef} contains the following rules:

   $holds(f, res(a, S)) \leftarrow holds(p_1, S), \ldots, holds(p_n, S), \neg holds(q_1, S), \ldots, \neg holds(q_r, S).$

   $ab(f, a, S) \leftarrow \mathbf{not}\ \neg holds(p_1, S), \ldots, \mathbf{not}\ \neg holds(p_n, S), \mathbf{not}\ holds(q_1, S), \ldots, \mathbf{not}\ holds(q_r, S).$

   else, if f is the negative fluent literal $\neg g$ then π_3^{ef} contains the following rules:

   $\neg holds(g, res(a, S)) \leftarrow holds(p_1, S), \ldots, holds(p_n, S), \neg holds(q_1, S), \ldots, \neg holds(q_r, S).$

   $ab(g, a, S) \leftarrow \mathbf{not}\ \neg holds(p_1, S), \ldots, \mathbf{not}\ \neg holds(p_n, S), \mathbf{not}\ holds(q_1, S), \ldots, \mathbf{not}\ holds(q_r, S).$

Example 93 Let D and O be as in Example 91. The program $\pi_3(D, O)$ consists of the following:

$holds(f, res(a, S)) \leftarrow holds(p, S).$
$ab(f, a, S) \leftarrow \mathbf{not}\ \neg holds(p, S).$
$holds(f, res(a, S)) \leftarrow \neg holds(p, S).$
$ab(f, a, S) \leftarrow \mathbf{not}\ holds(p, S).$

$\neg holds(f, s_0) \leftarrow .$

$holds(F, res(A, S)) \leftarrow holds(F, S), \mathbf{not}\ ab(F, A, S).$
$\neg holds(F, res(A, S)) \leftarrow \neg holds(F, S), \mathbf{not}\ ab(F, A, S).$

It is easy to see that $\pi_3(D, O) \not\models^{\star} holds(f, [a])$, and also $\pi_3(D, O) \not\models^{\star} \neg holds(f, [a])$; even though $D \models_O f$ **after** a.

Thus $\pi_3(D, O)$ does not make a wrong conclusion as $\pi_2(D, O)$ in Example 91. On the other hand it is not able to completely capture $\models_O$. Hence, although it is sound it is not complete with respect to $\models_O$. □

Example 94 Consider D from Example 84, and $O_4 = \{$ **initially** $alive\}$. The program $\pi_3(D, O_4)$, which is same as the program in part (2) of Section 2.2.2, consists of the following:

$holds(loaded, res(load, S)) \leftarrow.$
$ab(loaded, load, S) \leftarrow.$
$\neg holds(alive, res(shoot, S)) \leftarrow holds(loaded, S).$
$ab(alive, shoot, S) \leftarrow \mathbf{not}\ \neg holds(loaded, S).$

$holds(alive, s_0) \leftarrow .$

$holds(F, res(A, S)) \leftarrow holds(F, S), \textbf{not } ab(F, A, S).$
$\neg holds(F, res(A, S)) \leftarrow \neg holds(F, S), \textbf{not } ab(F, A, S).$

Now $\pi_3(D, O_4) \models^{*} ab(alive, shoot, s_0)$ and $\pi_3(D, O_4) \not\models holds(alive, [shoot])$ which agrees with $D \not\models_{O_4} alive$ **after** $shoot$. □

We now analyze $\pi_3(D, O)$ and relate entailments with respect to $\pi_3(D, O)$ with queries entailed by D, O.

Proposition 70 Let D be a consistent domain description and O be an initial state complete set of observations such that D, O is consistent. Let f be a fluent.

(i) $D \models_O f$ **after** $a_1, \ldots, a_n$ iff $\pi_3(D, O) \models holds(f, [a_n, \ldots, a_1])$.

(ii) $D \models_O \neg f$ **after** $a_1, \ldots, a_n$ iff $\pi_3(D, O) \models \neg holds(f, [a_n, \ldots, a_1])$. □

Proof: (Sketch)
Similar to the proof of Proposition 69 and uses lemmas similar to the lemmas in Section 5.1.3.

Lemma 5.1.8 Let D be a consistent domain description and O be a (possibly incomplete) set of observations about the initial state such that D, O is consistent. Then,

if $\pi_3(D, O) \models (\neg)holds(f, [a_n, \ldots, a_1)$ then for all extensions $\hat{O}$ of O, $\pi_3(D, \hat{O}) \models (\neg)holds(f, [a_n, \ldots, a_1)$.
□

Proof: Given D, O satisfying the conditions of the statement of the lemma, let $\hat{O}$, be an arbitrary extension of O. Let us now consider the two programs $\pi_3(D, \hat{O})$ and $\pi_3(D, O)$. It is easy to see that both of them have the signing S, where S is the set of all ground atoms about the predicate ab. Then, $\bar{S}$ is the set of all ground literals about the predicate $holds$. Next, since $\pi_3(D, \hat{O})$ has some extra ground atoms about the predicate $holds$ than $\pi_3(D, O)$, it follows that $\pi_3(D, O)_{\bar{S}} \preceq \pi_3(D, \hat{O})_{\bar{S}}$. Furthermore, $\pi_3(D, \hat{O})_S = \pi_3(D, O)_S$, which implies, $\pi_3(D, \hat{O})_S \preceq \pi_3(D, O)_S$. Thus by Proposition 47, $\pi_3(D, \hat{O})$ entails every ground literal in $\bar{S}$ that is entailed by $\pi_3(D, O)$. The statement of the lemma follows from this. □

Proposition 71 Let D be a consistent domain description, and O be a (possibly incomplete) set of observations about the initial state such that D, O is consistent. Let f be a fluent.

(i) If $\pi_3(D, O) \models holds(f, [a_n, \ldots, a_1])$ then $D \models_O f$ **after** $a_1, \ldots, a_n$.

(ii) If $\pi_3(D, O) \models \neg holds(f, [a_n, \ldots, a_1])$ then $D \models_O \neg f$ **after** $a_1, \ldots, a_n$. □

Proof: Follows from Lemma 5.1.8, Proposition 70, and Corollary 3. □

We now give an example which shows that $\pi_3(D, O)$ is not complete in the sense that $D \models_O (\neg)f$ **after** $a_1, \ldots, a_n$ but $\pi_3(D, O) \not\models (\neg)holds(f, [a_n, \ldots, a_1])$.

5.1.5 Sound reasoning with non-initial observations in AnsProlog$^{\neg}$: π_3 and π_4

In all of the previous formulations we had assumed that observations were only about the initial state. We now lift that assumption and present the first two of four sound formulations that can reason with general observations. The first two formulations are weaker than the other two but computationally superior.

Given a domain description D and a set of observations O $\pi_3(D, O)$ is same as in the previous section except that non-initial observations of the form $(\neg)f$ **after** $a_1, \ldots, a_n$ are represented as $(\neg)holds(f, [a_n, \ldots, a_1])$.

Example 95 Consider D from Example 84, and $O_1 = \{$ **initially** $alive; \neg alive$ **after** $shoot\}$ from Example 86. The program $\pi_3(D, O_1)$ consists of the following:

$holds(loaded, res(load, S)) \leftarrow.$
$ab(loaded, load, S) \leftarrow.$
$\neg holds(alive, res(shoot, S)) \leftarrow holds(loaded, S).$
$ab(alive, shoot, S) \leftarrow$ **not** $\neg holds(loaded, S).$

$holds(alive, s_0) \leftarrow .$
$\neg holds(alive, res(shoot, s_0)) \leftarrow.$

$holds(F, res(A, S)) \leftarrow holds(F, S),$ **not** $ab(F, A, S).$
$\neg holds(F, res(A, S)) \leftarrow \neg holds(F, S),$ **not** $ab(F, A, S).$

Now although, $D \models_{O_1}$ **initially** $loaded$, $\pi_3(D, O_1) \not\models holds(loaded, s_0)$. Hence, $\pi_3(D, O_1)$ is not complete with respect to D and O_1. As we will formally prove later, it is sound though. □

Example 96 Consider D from Example 84, and $O_2 = \{$ **initially** $alive; loaded$ **after** $shoot\}$ from Example 87. The program $\pi_3(D, O_2)$ consists of the following:

$holds(loaded, res(load, S)) \leftarrow.$
$ab(loaded, load, S) \leftarrow.$
$\neg holds(alive, res(shoot, S)) \leftarrow holds(loaded, S).$
$ab(alive, shoot, S) \leftarrow$ **not** $\neg holds(loaded, S).$

$holds(alive, s_0) \leftarrow .$
$holds(loaded, res(shoot, s_0)) \leftarrow.$

$holds(F, res(A, S)) \leftarrow holds(F, S),$ **not** $ab(F, A, S).$
$\neg holds(F, res(A, S)) \leftarrow \neg holds(F, S),$ **not** $ab(F, A, S).$

Now although, $D \models_{O_2}$ **initially** $loaded$, and $D \models_{O_2} \neg alive$ **after** $shoot$, we have $\pi_3(D, O_2) \not\models holds(loaded, s_0)$, and $\pi_3(D, O_2) \not\models \neg holds(alive, res(shoot, s_0))$. Hence, $\pi_3(D, O_2)$ is not complete with respect to D and O_2. As we will formally prove later, it is sound though. □

We now present another formulation $\pi_4(D, O)$ that is sound with respect to D and O and show that it extends $\pi_3(D, O)$. The program $\pi_4(D, O)$ consists of five parts π_3^{ef}, and π_2^{in} from the previous sections and π_4^{obs}, π_4^{-in}, and π_4^{back}, as defined below.

1. Backward reasoning rules: The backward reasoning rules are rules that can reason from observations about non-initial situations and make conclusions about their past up to the initial situation. They are collectively denoted by π_4^{back} and consist of the following rules.

   For every effect proposition of the form (5.1.1), if f is a fluent then π_4^{back} contains the following rules:

   (a) For each i, $1 \leq i \leq n$

   $holds(p_i, S) \leftarrow holds(f, res(a, S)), \neg holds(f, S).$

   $\neg holds(p_i, S) \leftarrow \neg holds(f, res(a, S)), holds(p_1, S), \ldots, holds(p_{i-1}, S),$
   $holds(p_{i+1}, S), \ldots, holds(p_n, S), \neg holds(q_1, S), \ldots, \neg holds(q_r, S).$

(b) For each j, $1 \leq j \leq r$

$$\neg holds(q_j, S) \leftarrow holds(f, res(a, S)), \neg holds(f, S).$$

$$holds(q_j, S) \leftarrow \neg holds(f, res(a, S)), \neg holds(q_1, S), \ldots, \neg holds(q_{j-1}, S), \neg holds(q_{j+1}, S), \ldots, \neg holds(q_r, S),\ holds(p_1, S), \ldots, holds(p_n, S).$$

For every effect proposition of the form (5.1.1), if f is the negative fluent literal $\neg g$ then π_4^{back} contains the following rules:

(a) For each i, $1 \leq i \leq n$

$$holds(p_i, S) \leftarrow \neg holds(g, res(a, S)), holds(g, S).$$

$$\neg holds(p_i, S) \leftarrow holds(g, res(a, S)), holds(p_1, S), \ldots, holds(p_{i-1}, S), holds(p_{i+1}, S), \ldots, holds(p_n, S), \neg holds(q_1, S), \ldots, \neg holds(q_r, S).$$

(b) For each j, $1 \leq j \leq r$

$$\neg holds(q_j, S) \leftarrow \neg holds(g, res(a, S)), holds(g, S).$$

$$holds(q_j, S) \leftarrow holds(g, res(a, S)), \neg holds(q_1, S), \ldots, \neg holds(q_{j-1}, S), \neg holds(q_{j+1}, S), \ldots, \neg holds(q_r, S),\ holds(p_1, S), \ldots, holds(p_n, S).$$

2. Backward inertia: The backward inertia rules are similar to the original inertia rules, except that they are in the backward direction. They are collectively denoted by π_4^{-in} and consist of the following:

$$holds(F, S) \leftarrow holds(F, res(A, S)), \textbf{not } ab(F, A, S).$$
$$\neg holds(F, S) \leftarrow \neg holds(F, res(A, S)), \textbf{not } ab(F, A, S).$$

3. Translating observations: The value propositions in O of the form are translated as follows and are collectively referred to as π_4^{obs}.

For every value proposition of the form (5.1.4) if f is a fluent then π_4^{obs} contains the following rule:

$$holds(f, [a_m, \ldots, a_1]) \leftarrow.$$

else, if f is the negative fluent literal $\neg g$ then π_4^{obs} contains the following rule:

$$\neg holds(g, [a_m, \ldots, a_1]) \leftarrow.$$

Example 97 Consider D from Example 95, and $O_1 = \{$ **initially** *alive*; $\neg$*alive* **after** *shoot*$\}$ from Example 95. The program $\pi_4(D, O_1)$ consists of $\pi_3(D, O_1)$ from Example 95 and the following rules. (Recall that we analyzed this program earlier in Section 3.4.2 to demonstrate the impact of the notion of signing on restricted monotonicity.)

$$holds(loaded, S) \leftarrow holds(alive, S), \neg holds(alive, res(shoot, S)).$$
$$\neg holds(loaded, S) \leftarrow holds(alive, res(shoot, S)).$$

$$holds(F, S) \leftarrow holds(F, res(A, S)), \textbf{not } ab(F, A, S).$$
$$\neg holds(F, S) \leftarrow \neg holds(F, res(A, S)), \textbf{not } ab(F, A, S).$$

Now although $\pi_3(D, O_1) \not\models^{\star} holds(loaded, s_0)$, $\pi_4(D, O_1) \models^{\star} holds(loaded, s_0)$, which agrees with $D \models_{O_1}$ **initially** *loaded*. □

Example 98 Consider D from Example 96, and $O_2 = \{$ **initially** $alive; loaded$ **after** $shoot\}$ from Example 96. The program $\pi_4(D, O_2)$ consists of $\pi_3(D, O_2)$ from Example 96 and the following rules.

$holds(loaded, S) \leftarrow holds(alive, S), \neg holds(alive, res(shoot, S))$.
$\neg holds(loaded, S) \leftarrow holds(alive, res(shoot, S))$.

$holds(F, S) \leftarrow holds(F, res(A, S)),$ **not** $ab(F, A, S)$.
$\neg holds(F, S) \leftarrow \neg holds(F, res(A, S)),$ **not** $ab(F, A, S)$.

Now although, $\pi_3(D, O_2) \not\models^{*} holds(loaded, s_0)$, and $\pi_3(D, O_2) \not\models^{*} \neg holds(alive, res(shoot, s_0))$; we have $\pi_4(D, O_2) \models^{*} holds(loaded, s_0)$, and $\pi_4(D, O_2) \models^{*} \neg holds(alive, res(shoot, s_0))$. □

We now show the soundness of entailments of $\pi_3(D, O)$ and $\pi_4(D, O)$ with respect queries entailed by D, O. We also relate the entailments of $\pi_3(D, O)$ with that of $\pi_4(D, O)$.

Proposition 72 Let D be a consistent domain description, and O be a set of observations such that D, O is consistent. Let f be a fluent.

(i) If $\pi_3(D, O) \models holds(f, [a_n, \ldots, a_1])$ then $D \models_O f$ **after** $a_1, \ldots, a_n$.

(ii) If $\pi_3(D, O) \models \neg holds(f, [a_n, \ldots, a_1])$ then $D \models_O \neg f$ **after** $a_1, \ldots, a_n$. □

Proposition 73 Let D be a consistent domain description and O be a set of observations such that D, O is consistent. Let f be a fluent.

(i) If $\pi_3(D, O) \models holds(f, [a_n, \ldots, a_1])$ then $\pi_4(D, O) \models holds(f, [a_m, \ldots, a_1])$.

(ii) If $\pi_3(D, O) \models \neg holds(f, [a_n, \ldots, a_1])$ then $\pi_4(D, O) \models \neg holds(f, [a_m, \ldots, a_1])$. □

Proof:
Let D, O satisfy the conditions of the statement of the lemma. Consider the two programs $\pi_4(D, O)$ and $\pi_3(D, O)$. It is easy to see that both of them have the signing S, where S is the set of all ground atoms about the predicate ab. Then, $\bar{S}$ is the set of all ground literals about the predicate $holds$. Next, since $\pi_4(D, O)$ has some extra ground rules with $holds$ in their head than $\pi_3(D, O)$, it follows that $\pi_3(D, O)_{\bar{S}} \preceq \pi_4(D, \hat{O})_{\bar{S}}$. Furthermore, $\pi_4(D, O)_S = \pi_3(D, O)_S$, which implies, $\pi_4(D, O)_S \preceq \pi_3(D, O)_S$. Thus by Proposition 47, $\pi_4(D, O)$ entails every ground literal in $\bar{S}$ that is entailed by $\pi_3(D, O)$. The statement of the lemma follows from this. □

The program $\pi_4(D, O)$ is not always sound with respect to the queries entailed by D, O. For example, for the domain description $D = \{a$ **causes** f **if** $p; a$ **causes** f **if** $\neg p\}$, and for the observation $O = \{f$ **after** $a\}$, $\pi_4(D, O)$ entails both $holds(p, s_0)$ and $\neg holds(p, s_0)$ and hence is inconsistent. For this reason the soundness of $\pi_4(D, O)$ with respect to the queries entailed by D, O is conditional. The following proposition states this.

Proposition 74 Let D be a consistent domain description such that for every action we only have one effect axiom, and O be a set of observations such that D, O is consistent. Let f be a fluent.

(i) If $\pi_4(D, O) \models holds(f, [a_n, \ldots, a_1])$ then $D \models_O f$ **after** $a_1, \ldots, a_n$.

(ii) If $\pi_4(D, O) \models \neg holds(f, [a_n, \ldots, a_1])$ then $D \models_O \neg f$ **after** $a_1, \ldots, a_n$. □

5.1.6 Assimilating observations using enumeration and constraints: π_5 and π_6

The incompleteness of $\pi_3(D, O)$ pointed out in Example 93 also plagues $\pi_4(D, O)$. We now present two formulations $\pi_5(D, O)$ and $\pi_6(D, O)$ which overcome this limitation and can reason in presence of general observations and incomplete knowledge about the initial state. They are sound and complete with respect to the entailment relation of D, O. $\pi_5(D, O)$ is an AnsProlog$^{\neg,\perp}$ program consisting of π_2^{ef}, and π_2^{in} from Sections 5.1.3 and π_5^{obs}, and π_5^{en} as described below. $\pi_6(D, O)$ is an AnsProlog$^{\neg, or\ ,\perp}$ program consisting of π_2^{ef}, and π_2^{in} from Sections 5.1.3 and π_5^{obs}, and π_6^{en} as described below. The role of π_5^{en} in $\pi_5(D, O)$ is to enumerate the various possible values a fluent can have in the initial state. The role of π_6^{en} in $\pi_6(D, O)$ is similar.

- Enumeration in $\pi_5(D, O)$: The enumeration rules in $\pi_5(D, O)$ collectively denoted by π_5^{en} consists of the following rules:

  $holds(F, s_0) \leftarrow \textbf{not}\ \neg holds(F, s_0).$

  $\neg holds(F, s_0) \leftarrow \textbf{not}\ holds(F, s_0).$

- Enumeration in $\pi_6(D, O)$: The enumeration rules in $\pi_6(D, O)$ collectively denoted by π_6^{en} consists of the following rule:

  $holds(F, s_0)\ or\ \neg holds(F, s_0).$

- Observations as constraints: The value propositions in O are translated as follows and are collectively referred to as π_5^{obs}.

  For an observation of the form (5.1.4) if f is a fluent then π_5^{obs} contains the following rule:

  $\leftarrow \textbf{not}\ holds(f, [a_m, \ldots, a_1]).$

  else if f is the fluent literal $\neg g$, then π_5^{obs} contains the following rule:

  $\leftarrow \textbf{not}\ \neg holds(g, [a_m, \ldots, a_1]).$

Example 99 Let D and O be as in Example 91. The program $\pi_5(D, O)$ consists of the following:

$holds(f, s_0) \leftarrow \textbf{not}\ \neg holds(f, s_0).$
$\neg holds(f, s_0) \leftarrow \textbf{not}\ holds(f, s_0).$
$holds(p, s_0) \leftarrow \textbf{not}\ \neg holds(p, s_0).$
$\neg holds(p, s_0) \leftarrow \textbf{not}\ holds(p, s_0).$

$\leftarrow \textbf{not}\ \neg holds(f, s_0).$

$holds(f, res(a, S)) \leftarrow holds(p, S).$
$ab(f, a, S) \leftarrow holds(p, S).$
$holds(f, res(a, S)) \leftarrow \neg holds(p, S).$
$ab(f, a, S) \leftarrow \neg holds(p, S).$

$holds(F, res(A, S)) \leftarrow holds(F, S), \textbf{not}\ ab(F, A, S).$
$\neg holds(F, res(A, S)) \leftarrow \neg holds(F, S), \textbf{not}\ ab(F, A, S).$

$\pi_5(D, O)$ has two answer sets $\{\neg holds(f, s_0), holds(p, s_0), ab(f, a, s_0), holds(f, [a]), holds(p, [a]), ab(f, a, [a]), holds(f, [a, a]), holds(p, [a, a]), ab(f, a, [a, a]), \ldots\}$ and $\{\neg holds(f, s_0), \neg holds(p, s_0),$

$ab(f,a,s_0), holds(f,[a]), \neg holds(p,[a]), ab(f,a,[a]), holds(f,[a,a]), \neg holds(p,[a,a]), ab(f,a,[a,a]), \ldots\}$.
Thus, $\pi_5(D,O) \models^{*} holds(f,[a])$, which agrees with $D \models_O f$ **after** a.
(Recall that $\pi_3(D,O) \not\models^{*} holds(f,[a])$.)

The program $\pi_5(D,O)$ is exactly same as $\pi_4(D,O)$ except that the first two rules of $\pi_5(D,O)$ are replaced by the following two rules:

$holds(f,s_0)$ *or* $\neg holds(f,s_0) \leftarrow$.
$holds(p,s_0)$ *or* $\neg holds(p,s_0) \leftarrow$.

The programs $\pi_5(D,O)$ and $\pi_5(D,O)$ have the same answer sets and thus $\pi_6(D,O) \models^{*} holds(f,[a])$.
□

Lemma 5.1.9 Let D be a consistent domain description, and O be a set of observations such that D,O is consistent. Let (σ_0,Φ_D) be a model of D,O and M be a consistent answer set of $\pi_5(D,O)$ such that $\sigma_0 = \{f \ : \ holds(f,s_0) \in M\}$. Let f be a fluent and $a_1,\ldots,a_n$ be a sequence of actions.

(i) $f \in \Phi_D(a_n,\ldots,\Phi_D(a_1,\sigma_0)\ldots)$ iff $holds(f,[a_n,\ldots,a_1]) \in M$.

(ii) $f \notin \Phi_D(a_n,\ldots,\Phi_D(a_1,\sigma_0)\ldots)$ iff $\neg holds(f,[a_n,\ldots,a_1]) \in M$ □

Proof: Can be shown by induction on n.

Lemma 5.1.10 Let D be a consistent domain description, and O be a set of observations such that D,O is consistent. For every model $M = (\sigma_0,\Phi_D)$ of D,O there exists a consistent answer set A of $\pi_5(D,O)$ such that $\sigma_0 = \{f \ : \ holds(f,s_0) \in A\}$. □

Proof: Using Lemma 5.1.3 we have that M is a model of D,O_M, where $O_M = \{$ **initially** $f \ : \ f \in \sigma_0\} \cup \{$ **initially** $\neg f \ : \ f \notin \sigma_0\}$. Consider $\pi_5(D,O_M)$.

Let $U = \{holds(f,s_0) \ : \ f$ is a fluent $\} \cup \{\neg holds(f,s_0) \ : \ f$ is a fluent $\}$. It is easy to see that U splits $\pi_5(D,O_M)$ such that $bot_U(\pi_5(D,O_M)) = \pi_5^{en}(D,O_M)$ and $top_U(\pi_5(D,O_M)) = \pi_2^{ef}(D,O_M) \cup \pi_2^{in}(D,O_M) \cup \pi_5^{obs}(D,O_M)$. It is easy to see that $A_0 = \{holds(f,s_0) \ : \ f \in \sigma_0\} \cup \{\neg holds(f,s_0) \ : \ f \notin \sigma_0\}$ is an answer set of $bot_U(\pi_5(D,O_M))$. From Theorem 3.5.1, an answer set of $top_U(\pi_5(D,O_M)) \cup A_0$ is an answer set of $\pi_5(D,O_M)$. Notice that $top_U(\pi_5(D,O_M)) \cup A_0$ is same as $\pi_2(D,O_M) \cup \pi_5^{obs}(D,O_M)$, and $\pi_5^{obs}(D,O_M)$ is a set of constraints. Thus answer sets of $top_U(\pi_5(D,O_M)) \cup A_0$ and answer sets of $\pi_2(D,O_M)$ that satisfy the constraints $\pi_5^{obs}(D,O_M)$ coincide. Since $A_0 \subseteq \pi_2(D,O_M)$, all answer sets of $\pi_2(D,O_M)$ satisfy the constraints $\pi_5^{obs}(D,O_M)$. Thus answer sets of $top_U(\pi_5(D,O_M)) \cup A_0$ and answer sets of $\pi_2(D,O_M)$ coincide. Since from Proposition 68 the program $\pi_2(D,O_M)$ has a unique consistent answer set, $top_U(\pi_5(D,O_M)) \cup A_0$ has a unique consistent answer set. Let us refer to this answer set as A. We will show that A is a consistent answer set A of $\pi_5(D,O)$.

It is clear that A_0 is an answer set of $bot_U(\pi_5(D,O))$. We will now argue that A is an answer set of $top_U(\pi_5(D,O)) \cup A_0$, and hence by Theorem 3.5.1 is an answer set of $\pi_5(D,O)$.

It is easy to see that the answer sets of $top_U(\pi_5(D,O)) \cup A_0$ and $top_U(\pi_5(D,O_M)) \cup A_0 \cup \pi_5^{obs}(D,O)$ coincide, as the only difference between them is that the later contains the additional constraints $\pi_5^{obs}(D,O_M)$, which are satisfied by the answer sets of both programs because of the presence of A_0.

By definition, A is the answer set of $top_U(\pi_5(D,O_M)) \cup A_0$. Using Lemma 5.1.9 with respect to D,O_M, and the fact that M, the model of D,O_M is is a model of D,O, it is easy to see that A satisfies the constraints $\pi_5^{obs}(D,O)$. Hence, A is the answer set of $top_U(\pi_5(D,O_M)) \cup A_0 \cup \pi_5^{obs}(D,O)$, and therefore the answer set of $top_U(\pi_5(D,O)) \cup A_0$. Thus A is an answer set of $\pi_5(D,O)$. □

Lemma 5.1.11 Let D be a consistent domain description, and O be a set of observations such that D, O is consistent. For every consistent answer set A of $\pi_5(D, O)$ there exists a model $M = (\sigma_0, \Phi_D)$ of D, O such that $\sigma_0 = \{f \ : \ holds(f, s_0) \in A\}$. □

Proof: (sketch)
Consider D, O_A, where, $O_A = \{$ **initially** $f \ : \ holds(f, s_0) \in A\} \cup \{$ **initially** $\neg f \ : \ \neg holds(f, s_0) \in A\}$. Let M be the model of D, O_A. We then show that A is an answer set of $\pi_5(D, O_A)$. Using Lemma 5.1.9 with respect to D, O_A, M and A we show that M satisfies O, and hence conclude that M is a model of $\pi_5(D, O)$. □

Proposition 75 Let D be a consistent domain description, and O be a set of observations such that D, O is consistent. Let f be a fluent.

(i) $\pi_5(D, O) \models holds(f, [a_n, \ldots, a_1])$ iff $D \models_O f$ **after** $a_1, \ldots, a_n$.

(ii) $\pi_5(D, O) \models \neg holds(f, [a_n, \ldots, a_1])$ iff $D \models_O \neg f$ **after** $a_1, \ldots, a_n$. □

Proof: Follows from Lemmas 5.1.9, 5.1.10, and 5.1.11. □

Proposition 76 Let D be a consistent domain description, and O be a set of observations such that D, O is consistent. Let f be a fluent.

(i) $\pi_6(D, O) \models holds(f, [a_n, \ldots, a_1])$ iff $D \models_O f$ **after** $a_1, \ldots, a_n$.

(ii) $\pi_6(D, O) \models \neg holds(f, [a_n, \ldots, a_1])$ iff $D \models_O \neg f$ **after** $a_1, \ldots, a_n$. □

Proof: Similar to the proof of Proposition 75 using lemmas similar to the ones used there. □

Note that the equivalence of $\pi_5(D, O)$ and $\pi_6(D, O)$ in terms of both having the same answer sets follows from the fact that $\pi_6(D, O)$ is head cycle free. This can be easily verified by observing that the only literals that appear as disjuncts in the head of rules are of the form $holds(F, s_0)$, and these literals do not appear in the head of any other rule. So there can not be a cycle involving these literals in the literal dependency graph of $\pi_6(D, O)$. Now since, by applying *disj_to_normal* to the rule in π_6^{en} we obtain π_5^{en}, using Theorem 3.10.1 we have that $\pi_5(D, O)$ and $\pi_6(D, O)$ have the same answer sets.

5.1.7 Ignoring sorts through language tolerance

Recall that the programs $\pi_1(D, O)$ – $\pi_6(D, O)$ in the previous sections are with respect to a sorted theory. We would now use the formulation of 'language tolerance' from Section 3.6.3 to show that the AnsProlog programs $\pi_2^+(D, O)$ – $\pi_5^+(D, O)$ are language tolerant, and thus if we were to ignore the sorts of these programs we will still make the same conclusions that we would have made if we respected the sorts. To show language tolerance, we will use the conditions in Theorem 3.6.3 which is applicable only to AnsProlog programs. Thus we consider the programs $\pi_1(D, O)$, and $\pi_2^+(D, O)$ – $\pi_5^+(D, O)$. The AnsProlog program, $\pi_1(D, O)$ is not predicate-order-consistent as $holds \leq_{+-} holds$ is true in its dependency graph. Thus we can not use Theorem 3.6.3 to show that $\pi_1(D, O)$ is language tolerant. It is easy to see that the AnsProlog programs $\pi_2^+(D, O)$ – $\pi_5^+(D, O)$ are predicate-order-consistent as the relation $\leq_{+-}$ in the dependency graph of those programs is empty.

The second condition in Theorem 3.6.3 for showing language tolerance is that the AnsProlog program be stable. We will now argue that the AnsProlog programs $\pi_1(D, O)$, and $\pi_2^+(D, O)$ – $\pi_5^+(D, O)$ are all stable with respect to the following mode m:

$holds(-,+)$
$holds'(-,+)$
$ab(+,+,+)$

Consider $\pi_1(D,O)$. Both rules in π_1^{ef} are stable because S, the only variable that occurs in it, occurs in the input position – marked by $+$ – in the literal in the head. The rules in π_1^{obs} are stable as they do not have variables. The rule in π_1^{in} has three variables: F, A and S. The variables A and S occur in the input position in the literal in the head. The variable F in the output position of the positive subgoal $holds(F,S)$ in the body, and since there is no other subgoal before it in the body, it satisfies the second condition of Definition 12. Hence, $\pi_1(D,O)$ is a stable program with respect to m. Similarly, we can show that $\pi_2^+(D,O)$ – $\pi_4^+(D,O)$ are stable with respect to the mode m.

But the program $\pi_5^+(D,O)$ is not stable with respect to m. This is because the rules π_5^{en} are not stable with respect to m. The following changes will make it stable.

1. Introduce a predicate $fluent$, and for all the fluents $f_1, \ldots, f_n$, add the following rules to the program.

   $fluent(f_1) \leftarrow$

   $\vdots$

   $fluent(f_n) \leftarrow$

2. Replace π_5^{en} by the following rules

   $holds(F, s_0) \leftarrow fluent(F), \mathbf{not}\ \neg holds(F, s_0)$

   $\neg holds(F, s_0) \leftarrow fluent(F), \mathbf{not}\ holds(F, s_0)$

3. Enhance m by adding the following I/O specification for $fluent(-)$. Let us refer to this enhanced mode by m'.

We refer to the modified program by $\pi_{5.1}$. It is easy to see that $\pi_{5.1}^+$ is stable with respect to m' and is also predicate-order-consistent and hence language tolerant.

The following table summarizes our result about the language tolerance of the various programs. The '?' in the table denotes that we do not know if $\pi_1(D,O)$ and $\pi_{5.1}^+(D,O)$ are stable or not. We only know that they do not satisfy a particular set of sufficiency condition – being predicate-order-consistent and stable – for language tolerance.

program	predicate-order-consistent	stable	language tolerant
$\pi_1(D,O)$	no	yes	?
$\pi_2^+(D,O)$	yes	yes	yes
$\pi_3^+(D,O)$	yes	yes	yes
$\pi_4^+(D,O)$	yes	yes	yes
$\pi_5^+(D,O)$	yes	no	?
$\pi_{5.1}^+(D,O)$	yes	yes	yes

In Section 8.3.3 we further study the properties of $\pi_1(D,O)$, $\pi_2^+(D,O)$ and $\pi_3^+(D,O)$ and show the correctness of the Prolog interpreter and its LDNF-resolution with respect to these AnsProlog programs.

5.1.8 Filter-abducibility of π_5 and π_6

In Section 3.9 we present conditions on AnsProlog$^{\neg, or}$ programs so that abductive reasoning with respect to such programs and particular kind of observations can be done through filtering. We now show how this is relevant with respect to the program $\pi_6(D, O)$. Recall that $\pi_6(D, O)$ consists of four parts: π_2^{ef}, π_2^{in}, π_5^{obs} and π_6^{en}. Since the rules π_2^{ef}, π_2^{in}, and π_6^{en} only depend on D, and π_5^{obs} only depends on O, let us refer to them as $\pi_6(D)$, and $\pi_6(O)$ respectively. Let us also denote π_4^{obs} by $Obs(O)$. Recall that if f **after** $a_1, \ldots, a_m$ is in O, then its translation in $Obs(O)$ will be $holds(f, [a_1, \ldots, a_m]) \leftarrow$ while its translation in $\pi_6(O)$ will be $\leftarrow$ **not** $holds(f, [a_1, \ldots, a_m])$. In the following program we show the connection between the abductive program $\langle \pi_6(D), Obs(O) \rangle$, and the filtering of $\pi_6(D)$ with $Obs(O)$.

Proposition 77 Let D be a consistent domain description, Abd be the set of ground literals about $holds$ in the situation s_0 and Obs be the set of ground atoms about $holds$. The AnsProlog$^{\neg, or}$ program $\pi_6(D)$ is filter-abducible with respect to Abd and Obs. □

Proof: (sketch) We use the sufficiency conditions in Proposition 62. The sufficiency condition (i) can be shown to hold by the direct application of Proposition 68 and Lemma 5.1.6. It is straight forward to show that the other two conditions hold. □

Since filtering $\pi_6(D)$ with $Obs(O)$ means adding $\pi_6(O)$ to $\pi_6(D)$ – as filtering in logic programs is done through adding constraints, the above proposition shows that the program $\pi_6(D, O)$ which is the union of $\pi_6(O)$ and $\pi_6(D)$ is equivalent to the abductive program $\langle \pi_6(D), Obs(O) \rangle$. In other words $\pi_6(D)$ is a 'good' program which allows easy assimilation of observations by just using them as filtering constraints. Although we can not use the sufficiency conditions in Proposition 62 for showing filter-abducibility of π_5, a proposition similar to it can be proven and it can be shown that π_5 is also filter-abducible.

5.1.9 An alternative formulation of temporal projection in AnsProlog: $\pi_{2.nar}$

The formulation of temporal projection in the programs $\pi_1 - \pi_6$ in the previous sections closely follows the Situation Calculus, where each situation is identified by a sequence of actions from the initial situation. These formulations are most appropriate for verifying the correctness of simple plans consisting of actions sequences, by checking if the program entails $holds(goal, action\_sequence)$. To do planning – where we are required to find a sequence of actions that lead to a situation where the goal conditions holds – with these formulations we either need interpreters that can do answer extraction, and then use such an interpreter to instantiate the variable $Plan$ in the query $holds(goal, Plan)$, or use a generate and test paradigm where possible action sequences are generated and tested to find out if they form a plan.

An alternative approach to formulate temporal projection is to use a linear time line, record action occurrences in this time line, and reason about fluents at different time points. Such an approach is used in event calculus, and in reasoning with narratives. Planning with such a formulation can be done by enumerating different action occurrences in different answer sets and eliminating the ones where the goal is not true in the final time point. The action occurrences in each of the resulting answer sets will correspond to plans. This approach, which is referred to as 'Answer-set planning' in recent literature, is similar to the planning with satisfiability approach where propositional logic is used instead.

In this section we give such an alternative AnsProlog formulation of temporal projection in presence of a complete initial state. In a subsequent section we will discuss how it can be easily modified to

do planning. Our formulation in this section is based on the program π_2 from Section 5.1.3. Besides the change in the approach, we make one additional change in the notation by introducing new predicates such as *not_holds* to avoid using $\neg$ in our program. This is done to make the program acceptable to interpreters such as 'smodels'.

Given a domain description D and a set of observations O which is initial state complete, we construct an AnsProlog program $\pi_{2.nar}(D, O)$ consisting of three parts $\pi^{ef}_{2.nar}$, $\pi^{obs}_{2.nar}$, and $\pi^{in}_{2.nar}$ as defined below.

1. Translating effect propositions: The effect propositions in D are translated as follows and are collectively referred to as $\pi^{ef}_{2.nar}$.

   For every effect proposition of the form (5.1.1) if f is a fluent then $\pi^{ef}_{2.nar}$ contains the following rules:

   $$holds(f, T+1) \leftarrow occurs(a, T), holds(p_1, T), \ldots, holds(p_n, T), \\ not\_holds(q_1, T), \ldots, not\_holds(q_r, T).$$

   $$ab(f, a, T) \leftarrow occurs(a, T), holds(p_1, T), \ldots, holds(p_n, T), \\ not\_holds(q_1, T), \ldots, not\_holds(q_r, T).$$

   else, if f is the negative fluent literal $\neg g$ then π^{ef}_2 contains the following rules:

   $$not\_holds(g, T+1) \leftarrow occurs(a, T), holds(p_1, T), \ldots, holds(p_n, T), \\ not\_holds(q_1, T), \ldots, not\_holds(q_r, T).$$

   $$ab(g, a, T) \leftarrow occurs(a, T), holds(p_1, T), \ldots, holds(p_n, T), \\ not\_holds(q_1, T), \ldots, not\_holds(q_r, T).$$

2. Translating observations: The value propositions in O are translated as follows and are collectively referred to as $\pi^{obs}_{2.nar}$.

   For every value proposition of the form (5.1.5) if f is a fluent then $\pi^{obs}_{2.nar}$ contains the following rule:

   $$holds(f, 1) \leftarrow.$$

   else, if f is the negative fluent literal $\neg g$ then π^{obs}_2 contains the following rule:

   $$not\_holds(g, 1) \leftarrow.$$

3. Inertia rules: Besides the above we have the following inertia rules referred to as $\pi^{in}_{2.nar}$.

   $$holds(F, T+1) \leftarrow occurs(A, T), holds(F, T), \mathbf{not}\ ab(F, A, T).$$

   $$not\_holds(F, T+1) \leftarrow occurs(A, T), not\_holds(F, T), \mathbf{not}\ ab(F, A, T).$$

Lemma 5.1.12 Let D be a consistent domain description and O be an initial state complete set of observations. Then $\pi_{2.nar}(D, O) \cup \{occurs(a_1, 1), \ldots, occurs(a_n, n)\}$ is acyclic. □

Lemma 5.1.13 Let D be a consistent domain description and O be an initial state complete set of observations. Let M be the answer set of $\pi_{2.nar}(D,O) \cup \{occurs(a_1,1),\ldots,occurs(a_n,n)\}$.

For any fluent f, and $i \leq l$, at least one, but not both of $holds(f,i)$ and $not\_holds(f,i)$ belong to M. □

Lemma 5.1.14 Let D be a consistent domain description and O be an initial state complete set of observations such that D,O is consistent. Let (σ_0,Φ_D) be the unique model of D,O and M be the answer set of $\pi_{2.nar}(D,O) \cup \{occurs(a_1,1),\ldots,occurs(a_n,n)\}$. Let f be a fluent.

(i) $f \in \Phi_D(a_n,\ldots,\Phi_D(a_1,\sigma_0)\ldots)$ iff $holds(f,n+1) \in M$.

(ii) $f \notin \Phi_D(a_n,\ldots,\Phi_D(a_1,\sigma_0)\ldots)$ iff $not\_holds(f,n+1) \in M$. □

The proof of the above three lemmas is similar to the proof of the Lemmas 5.1.5, 5.1.6, and 5.1.7 respectively.

Proposition 78 Let D be a consistent domain description and O be an initial state complete set of observations such that D,O is consistent. Let f be a fluent.

(i) $D \models_O f$ **after** $a_1,\ldots,a_n$ iff $\pi_{2.nar}(D,O) \cup \{occurs(a_1,1),\ldots,occurs(a_n,n)\} \models holds(f,n+1)$.

(ii) $D \models_O \neg f$ **after** $a_1,\ldots,a_n$ iff $\pi_{2.nar}(D,O) \cup \{occurs(a_1,1),\ldots,occurs(a_n,n)\} \models not\_holds(f,n+1)$. □

Proof: (sketch) Follows from the Lemmas 5.1.1 and 5.1.14. □

Exercise 14 Consider $\pi_{2.nar'}$ which is obtained from $\pi_{2.nar}$ by removing the rules with ab in their head,and replacing the rules in $\pi^{in}_{2.nar}$ by the following two rules.

$holds(F,T+1) \leftarrow occurs(A,T), holds(F,T), \textbf{not}\ not\_holds(F,T+1).$
$not\_holds(F,T+1) \leftarrow occurs(A,T), not\_holds(F,T), \textbf{not}\ holds(F,T+1).$

Show one-to-one correspondence between the answer sets of $\pi_{2.nar}$ and $\pi_{2.nar'}$ when we only consider the $holds$ and not_holds facts. □

5.1.10 Modifying $\pi_{2.nar}$ for answer set planning: $\pi_{2.planning}$

We now show how the program $\pi_{2.nar}$ from the previous section which reasons about fluent values at different time points, given a set of consecutive action occurrences starting from time point 1, can be enhanced to perform planning. The basic idea is that we decide on a plan length l, enumerate action occurrences up to time points corresponding to that length, and encode goals has constraints about the time point $l+1$, such that possible answer sets that encode action occurrence that do not satisfy the goal at time point $l+1$ are eliminated. Thus each of the answer sets – which survived the constraints – encode a plan. We now describe this modification to $\pi_{2.nar}$.

Given a domain description D, a set of observations O which is initial state complete, a plan length l, and a goal h, we construct the following AnsProlog program $\pi_{2.planning}(D,O,h,l)$ consisting of five parts: $\pi^{ef}_{2.nar}$, $\pi^{obs}_{2.nar}$, $\pi^{in}_{2.nar}$, are from the previous section, and $\pi^{choice}_{2.planning}$, and $\pi^{goal}_{2.planning}$ are described below.

1. Choice rules: We have the following choice rules that make sure that one and only one action occurs at each time point up to l. They are collectively referred to as $\pi^{choice}_{2.planning}$

   $not\_occurs(A,T) \leftarrow occurs(B,T), A \neq B$

   $occurs(A,T) \leftarrow T \leq l, \textbf{not } not\_occurs(A,T)$

2. Goal: Finally we have the following constraint, for our goal h. We refer to it as $\pi^{goal}_{2.planning}$.

   $\leftarrow \textbf{not } holds(h, l+1)$

Proposition 79 Let D be a consistent domain description, O be an initial state complete set of observations such that D, O is consistent, l be the length of the plan that we are looking for, and h be a fluent which is the goal.

(i) If there is a sequence of actions $a_1, \ldots, a_l$ such that $D \models_O h$ **after** $a_1, \ldots, a_l$ then there exists a consistent answer of $\pi_{2.planning}(D, O, h, l)$ containing $\{occurs(a_1, 1), \ldots, occurs(a_l, l)\}$ as the only facts about $occurs$.

(ii) If there exists a consistent answer of $\pi_{2.planning}(D, O, h, l)$ containing $\{occurs(a_1, 1), \ldots, occurs(a_l, l)\}$ as the facts about $occurs$ then $D \models_O h$ **after** $a_1, \ldots, a_l$. □

Proof (sketch): Follows from splitting he program to three layers, with the bottom layer consisting of $\pi^{choice}_{2.planning}$, the middle layer consisting of $\pi_{2.nar}(D, O)$ and the top layer consisting of $\pi^{goal}_{2.planning}$. It is easy to see that $\{occurs(a_1, 1), \ldots, occurs(a_l, l)\}$ is the set of $occur$ atoms in an answer set of $\pi^{choice}_{2.planning}$, and uniquely identifies that answer set. (I.e., no two different answer sets of $\pi^{choice}_{2.planning}$ can have the same set of $occur$.) Now we can use Proposition 78 to show the correspondence between $D \models_O h$ **after** $a_1, \ldots, a_l$ and $\pi_{2.nar}(D, O) \cup \{occurs(a_1, 1), \ldots, occurs(a_n, l)\} \models holds(h, l+1)$, and use that correspondence to complete our proof. □

One of the draw backs of the above formulation is that we need to give the exact plan length. One way to overcome this limitation is to have a 'no-op' action that has no effect on the world, and then have l as an upper bound. In a later section we discuss an alternative approach where l can be an upper bound and we do not need the 'no-op' action.

5.2 Reasoning about Actions and plan verification in richer domains

In the previous section we considered the simple action description language $\mathcal{A}$ and showed how reasoning about actions in $\mathcal{A}$ can be formulated in AnsProlog*. In the process we applied many results and notions from Chapter 3, thus demonstrating their usefulness. That was the reason why we used the simple language $\mathcal{A}$. In this section we consider extensions of $\mathcal{A}$ and their formulation in AnsProlog* and state the correspondences.

5.2.1 Allowing executability conditions

Our first extension to $\mathcal{A}$, which we will refer to as $\mathcal{A}_{ex}$, allows executability conditions in the domain description part. An executability condition is of the following form:

$$\textbf{executable } a \textbf{ if } p_1, \ldots, p_n, \neg q_1, \ldots, \neg q_r \tag{5.2.7}$$

where a is an action and, $p_1, \ldots, p_n, q_1, \ldots, q_r$ are fluents. Intuitively, the above executability condition means that if the fluent literals $p_1, \ldots, p_n, \neg q_1, \ldots, \neg q_r$ hold in the state σ of a situation s, then the action a is executable in s.

A domain description of $\mathcal{A}_{ex}$ consists of a set of effect propositions and executability conditions. The transition function Φ_D defined by a domain description in $\mathcal{A}_{ex}$ has one more condition than it had originally when defined for domain descriptions in $\mathcal{A}$. This condition is as follows:

Given a domain description D in $\mathcal{A}_{ex}$, for any action a, and any state σ, $\Phi_D(a, \sigma)$ is said to be *defined* if there is an executability condition of the form (5.2.7) such that $p_1, \ldots, p_n, \neg q_1, \ldots, \neg q_r$ hold in σ. Otherwise, it is said to be undefined. When $\Phi_D(a, \sigma)$ is defined, the value of $\Phi_D(a, \sigma)$ is same as the value of $\Phi_{D'}(a, \sigma)$ in $\mathcal{A}$, where D' consists of only the effect propositions of D. To define models and the corresponding entailment relation, the only additional requirement is that we declare that no fluent literal holds in an undefined state and if $\Phi_D(a, \sigma)$ is undefined then $\phi_D(a', \Phi_D(a, \sigma))$ is undefined.

Given a domain description D, and a set of observations O, and a model $M = (\sigma_0, \Phi_D)$ of D, O, we say a sequence of actions $a_1, \ldots, a_n$ is executable in M iff $\Phi_D(a_n, \ldots \phi_D(a_1, \sigma_0) \ldots)$ is defined. We say $a_1, \ldots, a_n$ is executable in D, O, iff it is executable in all models of D, O.

Example 100 Consider the following domain description D in $\mathcal{A}_{ex}$.

drive_to_airport **causes** *at_airport*
executable *drive_to_airport* **if** *has_car*
rent_a_car **causes** *has_car*

Let $\sigma_1 = \{\}$ and $\sigma_2 = \{has\_car\}$. We have $\Phi_D(drive\_to\_airport, \sigma_1)$ as undefined, while $\Phi_D(drive\_to\_airport, \sigma_2) = \{has\_car, at\_airport\}$. □

To compute the entailment relation $\models_O$ using AnsProlog* we now describe the changes that need to be made to the various programs in the previous section.

1. $\pi_{1.exec}(D, O)$: To allow executability conditions $\pi_1(D, O)$ is modified as follows.

   - We add the following rules which we will collectively refer to as $\pi^{ex}_{1.exec}$.

     For each executability condition of the form (5.2.7) $\pi^{ex}_{1.exec}$ contains the following rule:

     $executable(a, S) \leftarrow holds(p_1, S), \ldots, holds(p_n, S)$,**not** $holds(q_1, S), \ldots,$ **not** $holds(q_r, S)$.
   - To the body of each of the rules in π^{ef}_1 and π^{in}_1, we add the literals $executable(a, S)$, and $executable(A, S)$, respectively.
   - The rules in π^{obs}_1 remain unchanged.

2. $\pi_{2.exec}(D, O)$: To allow executability conditions $\pi_2(D, O)$ is modified as follows.

   - We add the following rules which we will collectively refer to as $\pi^{ex}_{2.exec}$.

     For each executability condition of the form (5.2.7) $\pi^{ex}_{2.exec}$ contains the following rule:

     $executable(a, S) \leftarrow holds(p_1, S), \ldots, holds(p_n, S), \neg holds(q_1, S), \ldots, \neg holds(q_r, S)$.
   - To the body of each of the rules in π^{ef}_2 and π^{in}_2, we add the literals $executable(a, S)$, and $executable(A, S)$, respectively.

- The rules in π_2^{obs} remain unchanged.

3. $\pi_{3.exec}(D, O)$: To allow executability conditions $\pi_3(D, O)$ is modified as follows.

   - We add $\pi_{2.exec}^{ex}$.
   - To the body of each of the rules in π_3^{ef} and π_2^{in}, we add the literals $executable(a, S)$, and $executable(A, S)$, respectively.
   - The rules in π_2^{obs} remain unchanged.

4. $\pi_{4.exec}(D, O)$: To allow executability conditions $\pi_4(D, O)$ is modified as follows.

   - We add the following rules which we will collectively refer to as $\pi_{4.exec}^{ex}$.

     $reachable(s_0) \leftarrow .$

     For each executability condition of the form (5.2.7) $\pi_{4.exec}^{ex}$ contains the following rule:

     $$reachable(res(a, S)) \leftarrow reachable(S), holds(p_1, S), \ldots, holds(p_n, S), \\ \neg holds(q_1, S), \ldots, \neg holds(q_r, S).$$

   - To the body of each of the rules in π_3^{ef} and π_2^{in}, we add the literals $reachable(res(a, S))$, and $reachable(res(A, S))$, respectively.
   - To the body of each of the rules in π_4^{obs} we add the literal $reachable([a_m, \ldots, a_1])$.
   - The rules in π_4^{-in} and π_4^{back} remain unchanged.

5. $\pi_{5.exec}(D, O)$: To allow executability conditions $\pi_5(D, O)$ is modified as follows.

   - We add $\pi_{2.exec}^{ex}$.
   - To the body of each of the rules in π_2^{ef} and π_2^{in}, we add the literals $executable(a, S)$, and $executable(A, S)$, respectively.
   - The rules in π_5^{en} and π_5^{obs} remain unchanged.

6. $\pi_{6.exec}(D, O)$: To allow executability conditions $\pi_6(D, O)$ is modified as follows.

   - We add $\pi_{2.exec}^{ex}$.
   - To the body of each of the rules in π_2^{ef} and π_2^{in}, we add the literals $executable(a, S)$, and $executable(A, S)$, respectively.
   - The rules in π_6^{en} and π_5^{obs} remain unchanged.

7. $\pi_{2.n.exec}(D, O)$: To allow executability conditions $\pi_{2.nar}(D, O)$ is modified as follows.

   - We add the following rules which we will collectively refer to as $\pi_{2.n.exec}^{ex}$.

     For each executability condition of the form (5.2.7) $\pi_{2.n.exec}^{ex}$ contains the following rule:

     $$executable(a, T) \leftarrow holds(p_1, T), \ldots, holds(p_n, T), \\ not\_holds(q_1, T), \ldots, not\_holds(q_r, T).$$

   - The rules in $\pi_{2.nar}^{ef}$, $\pi_{2.nar}^{in}$, and $\pi_{2.nar}^{obs}$ remain unchanged.

- We add the following rule which we will refer to as $\pi^{constr}_{2.n.exec}$.

  $\leftarrow occurs(A,T), \textbf{not}\ executable(A,T).$

8. $\pi_{2.p.exec}(D,O,h,l)$: To allow executability conditions $\pi_{2.planning}(D,O,h,l)$ is modified as follows.

   - We add the rules in $\pi^{ex}_{2.n.exec}$.
   - The rules in $\pi^{ef}_{2.nar}$, $\pi^{in}_{2.nar}$, $\pi^{obs}_{2.nar}$, $\pi^{choice}_{2.planning}$, and $\pi^{goal}_{2.planning}$ remain unchanged.
   - We add the rule $\pi^{constr}_{2.n.exec}$.

As in the previous sections we can now state the correspondences between the entailment relation $\models_O$ and the AnsProlog* programs. For brevity we state only one of them. Before that we would like to point out that the programs $\pi_{2.p.exec}(D,O,h,l)$ and $\pi_{2.n.exec}(D,O)$ differ from the other programs above in an important way. To incorporate the notions of executability $\pi_{2.p.exec}(D,O,h,l)$ and $\pi_{2.n.exec}(D,O)$ are obtained by only adding new rules – without making any surgery on the original rules – to $\pi_{2.planning}(D,O,h,l)$ and $\pi_{2.nar}(D,O)$ respectively. This is not the case for the other programs, where some of the original rules are modified.

Proposition 80 Let D be a consistent domain description in $\mathcal{A}_{ex}$, O be an initial state complete set of observations such that D,O is consistent, l be the length of the plan that we are looking for, and h be a fluent which is the goal.

(i) If there is a sequence of actions $a_1,\ldots,a_l$ such that $D \models_O h$ **after** $a_1,\ldots,a_l$ then there exists a consistent answer of $\pi_{2.p.exec}(D,O,h,l)$ containing $\{occurs(a_1,1),\ldots,occurs(a_l,l)\}$ as the only facts about *occurs*.

(ii) If there exists a consistent answer of $\pi_{2.p.exec}(D,O,h,l)$ containing $\{occurs(a_1,1),\ldots,occurs(a_l,l)\}$ as the facts about *occurs* then $D \models_O h$ **after** $a_1,\ldots,a_l$. □

Exercise 15 Executability conditions of the form discussed in this section have the implicit assumption that normally actions are not executable in a state and the exceptions are listed as executability conditions. This assumption is also used in the STRIPS language.

But in some domains the opposite is true. I.e., normally actions are executable and the few exceptions can be expressed as statements of the form:

impossible a **if** $p_1,\ldots,p_n,\neg q_1,\ldots,\neg q_r$.

Define the semantics of an extension of $\mathcal{A}$ that allows such impossibility statements and list what changes need to be made to the various AnsProlog* formulations of the prior sections so as to reason and plan in this extended language. □

5.2.2 Allowing static causal propositions

In the language $\mathcal{A}$ and $\mathcal{A}_{ex}$ the transition between states due to actions is specified through effect axioms and executability conditions. The effect axioms explicitly state the transition while the executability conditions explicitly state the executability of an action in a state. It was recognized that often we can express knowledge about the states directly in terms of what are possible or valid states and this knowledge can be used to indirectly infer effects of actions, executability of actions, or both and thus result in a more succinct and elaboration tolerant representation.

An example of such a constraint is: "a person can be at one place at one time". This can be expressed in classical logic as $at(X) \wedge at(Y) \Rightarrow X = Y$. In the absence of such a constraint the effect propositions of an action $move\_to(Y)$ can be expressed as:

$move\_to(Y)$ **causes** $at(Y)$.
$move\_to(Y)$ **causes** $\neg at(X), X \neq Y$.

But in the presence of the constraint, we just need the first effect axiom, and $\neg at(X)$ can be indirectly derived from the constraint. At first glance this does not seem to be much of a reduction, as instead of two effect propositions we have one effect proposition and one constraint. But now if we consider a set of actions similar to $move\_to(Y)$ such as: $drive\_to(Y)$, $fly\_to(Y)$, $take\_a\_train\_to(Y)$, $take\_a\_bus\_to(Y)$ we realize that for each of them we only need one effect proposition and over all we need one constraint. This supports our assertion about the advantage of using constraints. The indirect effect of an action due to constraints is referred to as *ramification* due to the occurrence of that action.

Constraints can also indirectly force certain actions to be not executable in certain states. For example, consider the constraint that $married\_to(X) \wedge married\_to(Y) \Rightarrow X = Y$, which encodes the constraint: "a person can not be married to two different persons." The effect of the action $marry(Y)$ can be expressed by the following effect proposition:

$marry(Y)$ **causes** $married\_to(Y)$

In presence of a the constraint about the impossibility of being married to two persons, we do not need an explicit executability condition saying that the action $marry(Y)$ is executable only when the person is not married to any one else. This condition is indirectly enforced by the constraint, as otherwise there will be a violation of the constraint. Formalization of this indirect qualification of the executability of an action due to a constraint is referred to as the *qualification problem.*

An important point to note about the two constraints $at(X) \wedge at(Y) \Rightarrow X = Y$ and $married\_to(X) \wedge married\_to(Y) \Rightarrow X = Y$ is that they are syntactically similar; yet they have different purpose. The first results in ramifications while the second results in a qualification. This means the processing of these constraints by themselves can not be automatic and for any automatic processing they will need additional annotation. This demonstrates the inability of classical logic for adequately expressing constraints. It is argued in the literature that a causal logic was more appropriate. In a $\mathcal{A}$ like syntax the above two constraints will be differently expressed as follows:

$at(X)$ **causes** $\neg at(Y)$ **if** $X \neq Y$, and

$married\_to(X) \wedge married\_to(Y) \wedge X \neq Y$ **causes** $false$.

We now formally define the syntax and semantics of a causal logic that we will use to express which states are valid and that may result in ramifications, qualifications, or both.

A static causal proposition is of the form

$$p_1, \ldots, p_n, \neg q_1, \ldots \neg q_r \textbf{ causes } f \tag{5.2.8}$$

where $p_1, \ldots, p_n, q_1, \ldots q_r$ are fluents and f is either fluent literal, or a special symbol $false$.

We say a state σ satisfies a static causal proposition of the form (5.2.8) if: (i) f is a literal and $p_1, \ldots, p_n, \neg q_1, \ldots \neg q_r$ hold in σ implies that f holds in σ; or (ii) f is $false$ and at least one of $p_1, \ldots, p_n, \neg q_1, \ldots \neg q_r$ does not hold in σ. For a state σ, by $open(\sigma)$ we denote the set of fluent literals $\sigma \cup \{\neg f \ : \ f$ is a fluent and $f \notin \sigma\}$. We say a set $\mathtt{s}$ of fluent literals and the symbol $false$

satisfies a static causal proposition of the form (5.2.8) if $\{p_1, \ldots, p_n, \neg q_1, \ldots \neg q_r\} \subseteq \mathsf{s}$ implies that $f \in \mathsf{s}$.

Let s be a set of fluent literals and the symbol $false$ and Z be a set of static causal propositions. By $Cn_Z(\mathsf{s})$ we denote the smallest set of fluent literals and the symbol $false$ that contains s and satisfies all propositions in Z. This set can be obtained by starting with s and repeatedly going over the static causal propositions in Z and adding the right hand side if the literals in the left hand side are already there, and repeating this until a fixpoint is reached.

Given a domain description D consisting of effect propositions, and static causal propositions, for an action a, and a state σ, we define $e_a(\sigma)$ as the set $\{f$: there exists an effect proposition of the form (5.1.1) such that $p_1, \ldots, p_n, \neg q_1, \ldots \neg q_r$ hold in $\sigma\}$.

We say a state $\sigma' \in \Phi_D(a, \sigma)$ if $open(\sigma') = Cn_Z(e_a(\sigma) \cup (open(\sigma) \cap open(\sigma')))$.

Example 101 Consider the following domain description D of effect propositions and static causal propositions:

a **causes** c
$c, \neg f$ **causes** g
$c, \neg g$ **causes** f

Let $\sigma_0 = \{\}$. We will now illustrate that $\sigma_1 = \{c, f\}$ and $\sigma_2 = \{c, g\}$ are in $\Phi_D(a, \sigma_0)$, while $\sigma_3 = \{c, f, g\}$ is not in $\Phi_D(a, \sigma_0)$.

Let us refer to the set of static causal propositions in D as Z. Now, $open(\sigma_0) = \{\neg c, \neg f, \neg g\}$, $open(\sigma_1) = \{c, f, \neg g\}$, $open(\sigma_2) = \{c, \neg f, g\}$, and $open(\sigma_3) = \{c, f, g\}$.

$Cn_Z(e_a(\sigma_0) \cup (open(\sigma_0) \cap open(\sigma_1))) = Cn_Z(\{c\} \cup (\{\neg c, \neg f, \neg g\} \cap \{c, f, \neg g\}) = Cn_Z(\{c\} \cup \{\neg g\})$ $= Cn_Z(\{c, \neg g\} = \{c, f, \neg g\} = open(\sigma_1)$. Hence, $\sigma_1 \in \Phi_D(a, \sigma_0)$.

$Cn_Z(e_a(\sigma_0) \cup (open(\sigma_0) \cap open(\sigma_2))) = Cn_Z(\{c\} \cup (\{\neg c, \neg f, \neg g\} \cap \{c, \neg f, g\}) = Cn_Z(\{c\} \cup \{\neg f\})$ $= Cn_Z(\{c, \neg f\} = \{c, \neg f, g\} = open(\sigma_2)$. Hence, $\sigma_2 \in \Phi_D(a, \sigma_0)$.

$Cn_Z(e_a(\sigma_0) \cup (open(\sigma_0) \cap open(\sigma_3))) = Cn_Z(\{c\} \cup (\{\neg c, \neg f, \neg g\} \cap \{c, f, g\}) = Cn_Z(\{c\} \cup \{\}) =$ $Cn_Z(\{c, \} = \{c\} \neq open(\sigma_3)$. Hence, $\sigma_3 \notin \Phi_D(a, \sigma_0)$. □

The above example shows that in presence of static causal propositions, effects of actions could be non-deterministic. The following example illustrates that static causal propositions are not contrapositive. I.e., the static causal proposition f **causes** g is not equivalent to $\neg g$ **causes** $\neg f$ and the direct effect of $\neg g$ does not indirectly cause $\neg f$.

Example 102 Consider the following domain description D:

$shoot$ **causes** $\neg alive$
$make\_walk$ **causes** $walking$
$\neg alive$ **causes** $\neg walking$

Let us refer to the set of static causal constraints in D as Z, and let $\sigma_0 = \{alive, walking\}$, $\sigma_1 = \{\}$, $\sigma_2 = \{walking\}$.

Let us consider the effect of the action $shoot$ in the state σ_0. $e_{shoot}(\sigma_0) = \{\neg alive\}$. $Cn_Z(e_{shoot}(\sigma_0) \cup$ $(open(\sigma_0) \cap open(\sigma_1))) = Cn_Z(\{\neg alive\} \cup (\{alive, walking\} \cap \{\neg alive, \neg walking\}) = Cn_Z(\{\neg alive\} \cup$

$\{\}) = Cn_Z(\{\neg alive\}) = \{\neg alive, \neg walking\} = open(\sigma_1)$. Hence, $\sigma_1 \in \Phi_D(shoot, \sigma_0)$. We can similarly show that no other states belong to $\Phi_D(shoot, \sigma_0)$ and hence the set $\Phi_D(shoot, \sigma_0)$ is a singleton set implying that the effect of *shoot* in the state σ_0 is deterministic.

Now let us consider the effect of the action *make_walk* in the state σ_1. $e_{make\_walk}(\sigma_1) = \{walking\}$. $Cn_Z(e_{make\_walk}(\sigma_1) \cup (open(\sigma_1) \cap open(\sigma_2))) = Cn_Z(\{walking\} \cup (\{\neg alive, \neg walking\} \cap \{\neg alive, walking\})$ $= Cn_Z(\{walking\} \cup \{\neg alive\}) = Cn_Z(\{\neg alive, walking\}) = \{\neg alive, walking, \neg walking\} \neq open(\sigma_2)$. Hence, $\sigma_2 \notin \Phi_D(shoot, \sigma_0)$. We can similarly show that no other state belongs to $\Phi_D(make\_walk, \sigma_0)$ and hence the set $\Phi_D(shoot, \sigma_0)$ is an empty set implying that *make_walk* is not executable in the state σ_0. This means that the static causal proposition $\neg alive$ **causes** $\neg walking$ is neither equivalent nor it encodes its contrapositive *walking* **causes** *alive*. If it were, then the indirect effect of *make_walk* would be that the turkey becomes *alive*, which of course is not intuitive. We get the intuitive conclusion that if some one tries to make a dead turkey walk he will fail in his attempt. Thus the static causal proposition results in a qualification when reasoning about the action *make_walk* in the state σ_1. □

Exercise 16 Consider the *move_to* and *marry* domains represented using static causal propositions and compute the transition due to the actions $move\_to(b)$ and $marry(b)$ on the states $\{at(a)\}$ and $\{married\_to(a)\}$ respectively. □

Given a domain description D consisting of effect propositions, executability conditions, and static causal propositions, and a set of observations O about the initial situation we say σ_0 is an initial state corresponding to D and O if (i) for all the observations of the form **initially** f in O, f holds in σ_0, and (ii) σ_0 satisfies the static causal propositions in D. We say $\sigma_0, a_1, \sigma_1, a_2, \sigma_2 \ldots a_n, \sigma_n$ is a valid trajectory of D and O, if (a) σ_0 is an initial state corresponding to D and O, (b) for $1 \leq i \leq n$, a_i is executable in σ_{i-1}, and (c) for $1 \leq i \leq n$, $\sigma_i \in \Phi_D(a_i, \sigma_{i-1})$.

We now present an AnsProlog$^\perp$ program $\pi_{7.n.causal}(D, O)$ whose answer sets correspond to the valid trajectories of D and O. The program $\pi_{7.n.causal}(D, O)$ consists of the components $\pi^{en}_{7.n.causal}(D, O)$, $\pi^{ef}_{7.n.causal}(D, O)$, $\pi^{st}_{7.n.causal}(D, O)$, $\pi^{obs}_{7.n.causal}(D, O)$, $\pi^{in}_{7.n.causal}(D, O)$, $\pi^{ex}_{7.n.causal}(D, O)$, $\pi^{constr}_{7.n.causal}(D, O)$, and $\pi^{choice}_{7.n.causal}$ as described below.

1. Enumeration in $\pi_{7.n.causal}(D, O)$: The enumeration rules in $\pi_{7.n.causal}(D, O)$ collectively denoted by $\pi^{en}_{7.n.causal}$ consists of the following rules:

   $holds(F, s_0) \leftarrow$ **not** $not\_holds(F, s_0)$.

   $not\_holds(F, s_0) \leftarrow$ **not** $holds(F, s_0)$.

2. Translating effect propositions: The effect propositions in D are translated as follows and are collectively referred to as $\pi^{ef}_{7.n.causal}$.

   For every effect proposition of the form (5.1.1) if f is a fluent then $\pi^{ef}_{7.n.causal}$ contains the following rules:

   $$holds(f, T+1) \leftarrow occurs(a, T), holds(p_1, T), \ldots, holds(p_n, T), not\_holds(q_1, T), \ldots, not\_holds(q_r, T).$$

   else, if f is the negative fluent literal $\neg g$ then $\pi^{ef}_{7.n.causal}$ contains the following rules:

   $$not\_holds(g, T+1) \leftarrow occurs(a, T), holds(p_1, T), \ldots, holds(p_n, T), not\_holds(q_1, T), \ldots, not\_holds(q_r, T).$$

3. Translating static causal propositions: The static causal propositions in D are translated as follows and are collectively referred to as $\pi^{st}_{7.n.causal}$.

   For every static causal proposition of the form (5.2.8) if f is a fluent then $\pi^{st}_{7.n.causal}$ contains the following rule:

   $holds(f,T) \leftarrow holds(p_1,T),\ldots,holds(p_n,T),not\_holds(q_1,T),\ldots,not\_holds(q_r,T).$

   else, if f is the negative fluent literal $\neg g$ then $\pi^{st}_{7.n.causal}$ contains the following rule:

   $not\_holds(g,T) \leftarrow holds(p_1,T),\ldots,holds(p_n,T),not\_holds(q_1,T),\ldots,not\_holds(q_r,T).$

   else, if f is symbol $false$ then $\pi^{st}_{7.n.causal}$ contains the following rule:

   $\leftarrow holds(p_1,T),\ldots,holds(p_n,T),not\_holds(q_1,T),\ldots,not\_holds(q_r,T).$

4. Translating observations: The value propositions in O are translated as follows and are collectively referred to as $\pi^{obs}_{7.n.causal}$.

   For every value proposition of the form (5.1.5) if f is a fluent then $\pi^{obs}_{7.n.causal}$ contains the following rule:

   $\leftarrow$ **not** $holds(f,1).$

   else, if f is the negative fluent literal $\neg g$ then $\pi^{obs}_{7.n.causal}$ contains the following rule:

   $\leftarrow$ **not** $not\_holds(g,1).$

5. Inertia rules: Besides the above we have the following inertia rules referred to as $\pi^{in}_{7.n.causal}$.

   $holds(F,T+1) \leftarrow occurs(A,T),holds(F,T),$ **not** $not\_holds(F,T+1).$

   $not\_holds(F,T+1) \leftarrow occurs(A,T),not\_holds(F,T),$ **not** $holds(F,T+1).$

6. We add the following rules which we will collectively refer to as $\pi^{ex}_{7.n.causal}$.

   For each executability condition of the form (5.2.7) $\pi^{ex}_{7.n.causal}$ contains the following rule:

   $executable(a,T) \leftarrow holds(p_1,T),\ldots,holds(p_n,T),$
   $not\_holds(q_1,T),\ldots,not\_holds(q_r,T).$

7. We add the following rules which we will refer to as $\pi^{constr}_{7.n.causal}$.

   $\leftarrow holds(F,1),not\_holds(F,1).$
   $\leftarrow occurs(A,T),$ **not** $executable(A,T).$
   $\leftarrow occurs(A,T),holds(F,T+1),not\_holds(F,T+1).$

8. Choice rules: We have the following choice rules that make sure that one and only one action occurs at each time point up to l. They are collectively referred to as $\pi^{choice}_{7.n.causal}$.

   $not\_occurs(A,T) \leftarrow occurs(B,T),A \neq B.$

   $occurs(A,T) \leftarrow$ **not** $not\_occurs(A,T).$

We now formally relate valid trajectories of D and O and answer sets of $\pi_{7.n.causal}(D, O)$.

Proposition 81 Let D be a domain description of effect propositions, executability conditions, and static causal propositions. Let O be a set of observations about the initial state.

(i) $\sigma_0, a_1, \sigma_1, \ldots, a_n, \sigma_n$ is a valid trajectory of D and O implies that there exists an answer set A of $\pi_{7.n.causal}(D, O)$ containing $\{occurs(a_1, 1), \ldots, occurs(a_n, n)\}$ as the only facts about *occurs* during the time points 1 to n and for $0 \leq i \leq n$, $\sigma_i = \{f \ : \ holds(f, i+1) \in A\}$ and $\sigma_i \cap \{f \ : \ not\_holds(f, i+1) \in A\} = \emptyset$.

(ii) Let A be an answer set of $\pi_{7.n.causal}(D, O)$ with $\{occurs(a_1, 1), \ldots, occurs(a_n, n)\}$ as the set of facts about *occurs* during the time points 1 to n; then $\sigma_0, a_1, \sigma_1, \ldots, a_n, \sigma_n$ is a valid trajectory of D and O, where for $0 \leq i \leq n$, $\sigma_i = \{f \ : \ holds(f, i+1) \in A\}$. □

5.2.3 Reasoning about parallel execution of actions

So far in this chapter we have reasoned about only sequential occurrences of actions and planned with them. In a more realistic scenario actions may be executed in parallel. Reasoning about such executions is the goal of this sub-section. When actions that do not interfere, which is often the case, are executed in parallel their effect can be computed as the cumulative effects of the individual actions. In some cases though they may interfere with each other and the effect may be different from their cumulative effect. An example of the later are the actions $left\_lift$ and $right\_lift$ with respect to a large bowl of soup. Individually each of these two action will cause the bowl of soup to spill, while when done in parallel the effect is different, instead of the soup getting spilled the bowl gets lifted. A straight forward approach of formalizing effects of parallel execution of actions would be to have effect propositions for each possible parallel execution of actions. But that would lead to explicitly representing the effect of 2^n action combinations, when our domain has n actions. This can be avoided by the use of normative statements and exceptions. The normative statement would be: Normally parallel execution of a set of actions inherits the individual effect of the actions in the set. We can then have exceptions to such inheritance, which encode the cases when inheritance should not be used, including when two actions interfere with each other, or complement each other resulting in a different effect.

To formulate the reasoning about such parallel actions in an $\mathcal{A}$ like language we minimally extend $\mathcal{A}$ and allow the actions in the effect propositions of the form (5.1.1) and observations of the form (5.1.4) to be compound actions, which we represent by a set of basic actions. We refer to this extension of $\mathcal{A}$ as $\mathcal{A}_c$.

Example 103 Consider the domain of lifting a bowl of soup. We can express the effect of the actions as follows:

$\{left\_lift\}$ **causes** *spilled*
$\{right\_lift\}$ **causes** *spilled*
$\{left\_lift, right\_lift\}$ **causes** *lifted* □

As in the case of $\mathcal{A}$ the role of the effect propositions is to define a transition function from states and actions – which now are compound actions represented by a set of basic actions, to states. We now define a transition function for domain descriptions in $\mathcal{A}_c$ which differs from the transition functions of domain descriptions in $\mathcal{A}$, in that when we have effect propositions a **causes** f **if** p and a **causes** $\neg f$ **if** p, we now treat it to mean a is not executable in a state where p is true,

rather than interpreting it as the domain being inconsistent. This isolates the 'badness' of having the effect propositions a **causes** f **if** p and a **causes** $\neg f$ **if** p in a domain description.

We say that a fluent literal f is an immediate effect of an action a in a state σ, if there exists an effect proposition of the form (5.1.1) such that $p_1, \ldots, p_n, \neg q_1, \ldots, \neg q_r$ hold in σ. For a state σ and an action a, the set of positive (and negative resp.) fluent literals that are immediate effects of a on σ is denoted by $direct^+(a, \sigma)$ (and $direct^-(a, \sigma)$ resp.). We say that a fluent literal f is an inherited effect of an action a in a state σ, if there exists $b \subset a$ such that f is an immediate effect of b and there is no c, $b \subset c \subseteq a$ such that the complement of f is an immediate effect of c. For a state σ and an action a, the set of positive (and negative resp.) fluent literals that are inherited effects of a on σ is denoted by $inherited^+(a, \sigma)$ (and $inherited^-(a, \sigma)$ resp.). We say that a fluent literal f is an effect of an action a in a state σ, if f is either an immediate effect or an inherited effect of a in σ. We then define $E^+(a, \sigma)$ as the set fluents that are effects of a in σ, and $E^-(a, \sigma)$ as the set fluents f such that $\neg f$ is an effect of a in σ. In other words, $E^+(a, \sigma) = direct^+(a, \sigma) \cup inherited^+(a, \sigma)$ and $E^-(a, \sigma) = direct^-(a, \sigma) \cup inherited^-(a, \sigma)$. $\Phi(a, \sigma)$ is said to be undefined if $E^+(a, \sigma)$ and $E^-(a, \sigma)$ intersect; otherwise $\Phi(a, \sigma)$ is defined as the set $\sigma \cup E^+(a, \sigma) \setminus E^-(a, \sigma)$. Note that unlike domain descriptions in $\mathcal{A}$, a domain description D in $\mathcal{A}_c$ always has a transition function, and we denote it by Φ_D.

We say σ_0 is an initial state corresponding a domain description D and a set of observations O, if for all observations of the form (5.1.4) in O, $[a_m, \ldots, a_1]\sigma_0$ is defined and the fluent literal f holds in it. We then say that (σ_0, Φ_D) *satisfies* O. Given a domain description D and a set of observations O, we refer to the pair (Φ_D, σ_0), where Φ_D is the transition function of D and σ_0 is an initial state corresponding to D and O, as a *model* of D, O. We say D, O is *consistent* if it has a model and is *complete* if it has a unique model.

Example 104 Let D_1 be the domain description $\{\{lift\}$ **causes** $lifted, \{open\}$ **causes** $opened\}$, and $O = \{$ **initially** $\neg lifted$, **initially** $\neg opened\}$. The only model of D_1 and O is given by (σ_0, Φ), where $\sigma_0 = \emptyset$ and Φ is defined as follows:

$\Phi(open, \sigma) = \sigma \cup \{opened\}$

$\Phi(lift, \sigma) = \sigma \cup \{lifted\}$

$\Phi(\{open, lift\}, \sigma) = \sigma \cup \{opened, lifted\}$

$D_1 \models_O opened$ **after** $\{open, lift\}$ and $D_1 \models_O lifted$ **after** $\{open, lift\}$. □

Example 105 Consider a domain containing three unit actions *paint*, *close* and *open*, and two fluents, *opened* and *painted*. The effects of these actions are described by the following domain description D_3:

$\{close\}$ **causes** $\neg opened$

$\{open\}$ **causes** $opened$

$\{paint\}$ **causes** $painted$.

Let O be the empty set. The transition function Φ of D_3 is defined as follows:

$\Phi(\emptyset, \sigma) = \sigma$

$\Phi(\{paint\}, \sigma) = \sigma \cup \{painted\}$

$\Phi(\{close\}, \sigma) = \sigma \setminus \{opened\}$

$\Phi(\{open\}, \sigma) = \sigma \cup \{opened\}$

$\Phi(\{paint, close\}, \sigma) = \sigma \cup \{painted\} \setminus \{opened\}$

$\Phi(\{paint, open\}, \sigma) = \sigma \cup \{painted\} \cup \{opened\}$

$\Phi(\{close, open\}, \sigma)$ and $\Phi(\{close, open, paint\}, \sigma)$ are undefined.

D_3 and O have four models, which are of the form (σ, Φ) where $\sigma \subseteq \{opened, painted\}$. □

We now present an AnsProlog$^{\neg,\perp}$, formulation that computes the entailment relation $\models_O$, given a domain description D and a set of observations O. The program we present is also a sorted program like the ones in Section 5.1.1; the only small difference being in the sort action, which now refers to compound actions that represent parallel execution of a set of basic actions. We refer to the AnsProlog$^{\neg,\perp}$ of this section as $\pi_{5.compound}(D, O)$ implying that it is similar to $\pi_5(D, O)$ from Section 5.1.1 and it allows compound actions. The program $\pi_{5.compound}(D, O)$ consists of $\pi^{ef.dep}_{5.compound}$, $\pi^{obs}_{5.compound}$, $\pi^{ef.indep}_{5.compound}$, $\pi^{in}_{5.compound}$, $\pi^{inh}_{5.compound}$, $\pi^{en}_{5.compound}$, and $\pi^{aux}_{5.compound}$ as defined below.

1. Translating effect propositions: The effect propositions in D are translated as follows and are collectively referred to as $\pi^{ef.dep}_{5.compound}$.

   For every effect proposition of the form (5.1.1) if f is a fluent then $\pi^{ef.dep}_{5.compound}$ contains the following rule:

   $causes(a, f, S) \leftarrow holds(p_1, S), \ldots, holds(p_n, S), \neg holds(q_1, S), \ldots, \neg holds(q_r, S).$

   else, if f is the negative fluent literal $\neg g$ then $\pi^{ef.dep}_{5.compound}$ contains the following rule:

   $causes(a, neg(g), S) \leftarrow holds(p_1, S), \ldots, holds(p_n, S), \neg holds(q_1, S), \ldots, \neg holds(q_r, S).$

2. Observations as constraints: The value propositions in O are translated as follows and are collectively referred to as $\pi^{obs}_{5.compound}$.

   For an observation of the form (5.1.4) if f is a fluent then $\pi^{obs}_{5.compound}$ contains the following rule:

   $\leftarrow$ **not** $holds(f, [a_m, \ldots, a_1])$.

   else if f is the fluent literal $\neg g$, then $\pi^{obs}_{5.compound}$ contains the following rule:

   $\leftarrow$ **not** $\neg holds(g, [a_m, \ldots, a_1])$.

3. Domain independent effect rules: The domain independent effect rules in $\pi_{5.compound}(D, O)$ collectively denoted by $\pi^{ef.indep}_{5.compound}$ contains the following rules:

   $holds(F, res(A, S)) \leftarrow causes(A, F, S),$ **not** $undefined(A, S)$.
   $\neg holds(F, res(A, S)) \leftarrow causes(A, neg(F), S),$ **not** $undefined(A, S)$.
   $undefined(A, S) \leftarrow causes(A, F, S), causes(A, neg(F), S)$.
   $undefined(A, res(B, S)) \leftarrow undefined(B, S)$

4. Inertia rules: The inertia rules in $\pi_{5.compound}(D,O)$ collectively denoted by $\pi^{in}_{5.compound}$ consists of the following:

   $holds(F, res(A,S)) \leftarrow holds(F,S), \mathbf{not}\ causes(A, neg(F), S), singleton(A), \mathbf{not}\ undefined(A,S).$

   $\neg holds(F, res(A,S)) \leftarrow \neg holds(F,S), \mathbf{not}\ causes(A, F, S), singleton(A), \mathbf{not}\ undefined(A,S).$

5. Inheritance axioms: The inheritance rules in $\pi_{5.compound}(D,O)$ collectively denoted by $\pi^{inh}_{5.compound}$ consists of the following rules:

   $holds(F, res(A,S)) \leftarrow subset(B,A), holds(F, res(B,S)), \mathbf{not}\ noninh(F,A,S), \mathbf{not}\ undefined(A,S).$
   $\neg holds(F, res(A,S)) \leftarrow subset(B,A), \neg holds(F, res(B,S)), \mathbf{not}\ noninh(neg(F),A,S),$
   $\mathbf{not}\ undefined(A,S).$
   $cancels(X,Y,F,S) \leftarrow subset(X,Z), subseteq(Z,Y), cause(Z, neg(F), S).$
   $cancels(X,Y,neg(F),S) \leftarrow subset(X,Z), subseteq(Z,Y), cause(Z,F,S).$
   $noninh(F,A,S) \leftarrow subseteq(U,A), causes(U, neg(F), S), \mathbf{not}\ cancels(U,A,neg(F),S).$
   $noninh(neg(F),A,S) \leftarrow subseteq(U,A), causes(U,F,S), \mathbf{not}\ cancels(U,A,F,S).$
   $undefined(A,S) \leftarrow noninh(F,A,S), noninh(neg(F),A,S).$

6. Enumeration about the initial situation in $\pi_{5.compound}(D,O)$: The enumeration rules in $\pi_{5.compound}(D,O)$ collectively denoted by $\pi^{en}_{5.compound}$ consists of the following rules:

   $holds(F, s_0) \leftarrow \mathbf{not}\ \neg holds(F, s_0).$

   $\neg holds(F, s_0) \leftarrow \mathbf{not}\ holds(F, s_0).$

7. Auxiliary rules: The auxiliary rules in $\pi_{5.compound}(D,O)$ collectively denoted by $\pi^{aux}_{5.compound}$ consists of the following rules:

   $\neg subseteq(U,V) \leftarrow in(X,U), \mathbf{not}\ in(X,V).$
   $subseteq(U,V) \leftarrow \mathbf{not}\ \neg subseteq(U,V).$
   $eq(X,X) \leftarrow.$
   $\neg singleton(X) \leftarrow in(Y,X), in(Z,X), \mathbf{not}\ eq(Y,Z).$
   $singleton(X) \leftarrow \mathbf{not}\ \neg singleton(X).$
   $subset(X,Y) \leftarrow subseteq(X,Y), \mathbf{not}\ subseteq(Y,X).$

We now analyze $\pi_{5.compound}(D,O)$ and relate entailments with respect to $\pi_{5.compound}(D,O)$ with queries entailed by D,O.

Lemma 5.2.1 Let D be a domain description and O be a set of observations. A set A is a consistent answer set of $\pi_{5.compound}(D,O)$ iff A is the consistent answer set of $\pi_{5.compound}(D,O) \cup \{holds(f,s_0) : holds(f,s_0) \in A\} \cup \{\neg holds(f,s_0) : \neg holds(f,s_0) \in A\}$. □

Lemma 5.2.2 Let D be a domain description, and O be a set of observations. $M = (\sigma_0, \Phi)$ is a model of D,O iff M is the model of (D, O_M), where $O_M = O \cup \{$ **initially** $f : f \in \sigma_0\} \cup \{$ **initially** $\neg f : f \notin \sigma_0\}$. □

In the following we denote the situation $[a_1, \ldots, a_n]$ by s_n, and the state $[a_m, \ldots, a_1]\sigma_0$ by σ_m.

Lemma 5.2.3 Models of D vs Answer sets of πD
Let D be a domain description and O be a set of observations. Let $M = (\sigma_0, \Phi)$ be a model of D,O and A be a consistent answer set of $\pi_{5.compound}(D,O)$ such that $\sigma_0 = \{f : holds(f,s_0) \in A\}$.

1. If $a_1, \ldots, a_n$ is executable in M then

   (a) $f \in direct^+(a_n, \sigma_{n-1})$ iff $causes(a_n, f, s_{n-1}) \in A$.

   (b) $f \in direct^-(a_n, \sigma_{n-1})$ iff $causes(a_n, neg(f), s_{n-1}) \in A$.

   (c) $f \in inherited^+(a_n, \sigma_{n-1})$ iff $noninh(neg(f), a_n, s_{n-1}) \in A$.

   (d) $f \in inherited^-(a_n, \sigma_{n-1})$ iff $noninh(f, a_n, s_{n-1}) \in A$.

   (e) $undefined(a_n, s_{n-1}) \notin A$.

   (f) $f \in \sigma_n \Leftrightarrow holds(f, s_n) \in A$

   (g) $f \notin \sigma_n \Leftrightarrow \neg holds(f, s_n) \in A$.

2. If $a_1, \ldots, a_n$ is not executable in M then
   $holds(f, s_n) \notin A$ and $\neg holds(f, s_n) \notin A$ and $undefined(a_n, s_{n-1}) \in A$. □

Lemma 5.2.4 Let D be a domain description and O be a set of observations such that D, O is consistent. For every model $M = (\sigma_0, \Phi)$ of D, O, there exists a consistent answer set A of $\pi_{5.compound}(D, O)$ such that $\sigma_0 = \{f \ : \ holds(f, s_0) \in A\}$. □

Lemma 5.2.5 Let D be a domain description and O be a set of observations such that D, O is consistent. For every consistent answer set A of $\pi_{5.compound}(D, O)$ there exists a model $M = (\sigma_0, \Phi)$ of D, O such that $\sigma_0 = \{f \ : \ holds(f, s_0) \in A\}$. □

Theorem 5.2.6 Soundness and Completeness of $\pi_{5.compound}$.
Let D be a domain description, O be a set of observations, f be a fluent, and $a_1, \ldots, a_n$ be a sequence of actions that is executable in all models of D, O. Then
(i) $\pi_{5.compound}(D, O) \models holds(f, [a_1, \ldots, a_n])$ iff $D \models f$ **after** $a_1, \ldots, a_n$.
(ii) $\pi_{5.compound}(D, O) \models \neg holds(f, [a_1, \ldots, a_n])$ iff $D \models \neg f$ **after** $a_1, \ldots, a_n$. □

Proof (sketch): Directly from Lemma 5.2.3, Lemma 5.2.4 and Lemma 5.2.5.

5.3 Answer set planning examples in extensions of $\mathcal{A}$ and STRIPS

In Sections 5.1.10 and 5.2.1 we discussed how to formulate planning using AnsProlog and in Section 5.1.9 we briefly mentioned the notion of 'Answer-set planning'. In this section we use the answer set planning methodology for planning with the Blocks world example. We now further motivate answer set planning and our use of it in this section.

In answer set planning, each plan corresponds to an answer set of the program. In this program a linear time line is used and the initial situation corresponds to the time point 1. Action occurrences at different time points are enumerated using the notion of 'choice' and possible answer sets (and the actions occurrences encoded in them) that do not lead to the holding of the goal at a required time point – defined by a given plan length – are eliminated using constraints. Thus the answer sets each encode a plan that achieves the goal at the required time point. As before, given what holds in a time point, reasoning about what holds in the next time point is formulated using effect rules and inertia rules.

Recently, there has been a lot of focus on answer set planning. Some of the main reasons behind this are:

- We now have the interpreters dlv and smodels which can generate one or more answer sets of an AnsProlog program. These interpreters are different from the query answering type of interpreters such as Prolog. The former is suitable for answer set planning, while the later is more appropriate for planning using variable instantiation.
- Answer set planning is similar to the satisfiability based planning, while planning in Prolog using variable instantiation is based on the principles of theorem proving. Theorem proving approach to planning was tried and abandoned because of its failure to plan in large domains. On the other hand satisfiability based planning has enjoyed tremendous success in recent years.

In this section we start with blocks world planning in STRIPS using the PDDL syntax that has been the standard in recent planning contests. We encode the blocks world example in AnsProlog and run it using smodels. The program that we present is an improvement over the program $\pi_{2.p.exec}$ from Section 5.2.1. The improvements are in allowing the user to give an upper bound of the plan length rather than the exact plan length, and disallowing further action occurrences in an answer set once the plan is found. Besides this we make additional modifications to reduce the number of predicates, and to make it easier to match with the PDDL syntax.

We then incorporate domain specific temporal constraints into the program so as to make the planning more efficient, and thus demonstrate the ease with each such constraints can be encoded in AnsProlog.

Finally, we consider richer action language features such as defined fluents, causal qualification and ramification constraints, and conditional effects, and show how they can be formulated in answer set planning. We also show that by allowing these features the programs become shorter and the execution time to find plans also reduces. We would like to point out here that it is not yet known how to incorporate causal constraints and the use of an upper bound of the plan length (instead of the plan length) in propositional logic formulations used in satisfiability based planning.

5.3.1 A blocks world example in PDDL

In this section we introduce PDDL by giving the specification of the blocks world domain in it. We first give the domain description part.

```
(define (domain BLOCKS)
  (:requirements :strips :typing)
  (:types block)
  (:predicates (on ?x - block ?y - block)
               (ontable ?x - block)
               (clear ?x - block)
               (handempty)
               (holding ?x - block)
               )

  (:action pick-up
            :parameters (?x - block)
            :precondition (and (clear ?x) (ontable ?x) (handempty))
            :effect
```

```
            (and (not (ontable ?x))
                 (not (clear ?x))
                 (not (handempty))
                 (holding ?x)))

  (:action put-down
            :parameters (?x - block)
            :precondition (holding ?x)
            :effect
            (and (not (holding ?x))
                 (clear ?x)
                 (handempty)
                 (ontable ?x)))
  (:action stack
            :parameters (?x - block ?y - block)
            :precondition (and (holding ?x) (clear ?y))
            :effect
            (and (not (holding ?x))
                 (not (clear ?y))
                 (clear ?x)
                 (handempty)
                 (on ?x ?y)))
  (:action unstack
            :parameters (?x - block ?y - block)
            :precondition (and (on ?x ?y) (clear ?x) (handempty))
            :effect
            (and (holding ?x)

                 (clear ?y)
                 (not (clear ?x))
                 (not (handempty))
                 (not (on ?x ?y)))))
```

The above PDDL description can be encoded in $\mathcal{A}$ as follows:

1. Sort: $block(X)$. In the rest of the descriptions X and Y range over the sort $block$.

2. fluents: $on(X), ontable(X), clear(X), handempty, holding(X)$.

3. Executability Conditions:

   executable $pick\_up(X)$ **if** $clear(X), ontable(X), handempty.$
   executable $put\_down(X)$ **if** $holding(X).$
   executable $stack(X, Y)$ **if** $holding(X), clear(Y).$
   executable $unstack(X, Y)$ **if** $on(X, Y), clear(X), handempty.$

4. Effect propositions:

$pick\_up(X)$ **causes** $\neg ontable(X), \neg clear(X), \neg handempty, holding(X)$.
$put\_down(X)$ **causes** $\neg holding(X), clear(X), handempty, ontable(X)$.
$stack(X,Y)$ **causes** $\neg holding(X), \neg clear(Y), clear(X), handempty, on(X,Y)$.
$unstack(X,Y)$ **causes** $holding(X), clear(Y), \neg clear(X), \neg handempty, \neg on(X,Y)$.

We denote the above by D_{bw}. We now give the initial conditions and the goals in PDDL.

```
(define (problem BLOCKS-4-0)
(:domain BLOCKS)
(:objects D B A C - block)
(:INIT (CLEAR C) (CLEAR A) (ONTABLE C)
       (ONTABLE B) (ON A B) (HANDEMPTY))
(:goal (AND (ON A C) (ON C B) )) )
```

In the language of $\mathcal{A}$ the sort *block* has the extent $\{a, b, c, d\}$ and the initial conditions are described as:

initially $clear(c)$
initially $clear(a)$
initially $ontable(c)$
initially $ontable(b)$
initially $on(a, b)$
initially $handempty$

We denote such a set of initial conditions by O, and this particular set by $O_{bw.1}$. In STRIPS/PDDL the assumption is that only fluents that are true are specified and all other fluents are false. For our convenience by $Comp(O)$ we denote the set $O \cup \{$ **initially** $\neg f \ : \ f$ is a fluent, and $f \notin O\}$. The goal is the set of literals $\{on(a, c), on(c, b)\}$. We denote such sets by G, and this particular one by $G_{bw.1}$.

5.3.2 Simple blocks world in AnsProlog: $\pi_{strips1}(D_{bw}, O_{bw}, G_{bw})$

In this sub-section we show how to encode the simple blocks world planning problem described in STRIPS using PDDL Section 5.3.1. We divide our encoding to two parts: the domain dependent part and the domain independent part. The former is a direct translation of the domain description (whether in $\mathcal{A}$ or in PDDL). The later is independent of the domain and may be used for planning with other domains.

1. The Domain dependent part $\pi^{dep}_{strips1}(D_{bw}, O_{bw}, G_{bw})$: This consists of five parts, defining the domain, the initial state, the goal conditions, the executability conditions, and the dynamic causal laws.

   (a) The domain $\pi^{dep.dom}_{strips1}(D_{bw})$: This part defines the objects in the world, the fluents and the actions.

   $block(a)$.
   $block(b)$.

$block(c)$.
$block(d)$.

$fluent(on(X,Y)) \leftarrow block(X), block(Y)$.
$fluent(ontable(X)) \leftarrow block(X)$.
$fluent(clear(X)) \leftarrow block(X)$.
$fluent(holding(X)) \leftarrow block(X)$.
$fluent(handempty)$.

$action(pick\_up(X)) \leftarrow block(X)$.
$action(put\_down(X)) \leftarrow block(X)$.
$action(stack(X,Y)) \leftarrow block(X), block(Y)$.
$action(unstack(X,Y)) \leftarrow block(X), block(Y)$.

(b) The executability conditions $\pi^{dep.exec}_{strips1}(D_{bw})$: This part states the executability conditions. Note that the simple form that is used here is only appropriate for STRIPS domains where the executability condition of an action is a conjunction of fluents. (In Section 5.3.4 and 5.3.5 we consider blocks world encodings that use more general executability conditions such as the one given in Section 5.2.1.) Here, $exec(a, f)$ means that f is among the executability conditions of a, and intuitively, a is executable in a state if all its executability conditions hold in that state. The later part will be encoded as a domain independent rule.

$exec(pick\_up(X), clear(X)))$.
$exec(pick\_up(X), ontable(X))$.
$exec(pick\_up(X), handempty)$.

$exec(put\_down(X), holding(X))$.

$exec(stack(X,Y), holding(X))$.
$exec(stack(X,Y), clear(Y))$.

$exec(unstack(X,Y), clear(X))$.
$exec(unstack(X,Y), on(X,Y))$.
$exec(unstack(X,Y), handempty)$.

(c) The dynamic causal laws: $\pi^{dep.dyn}_{strips1}(D_{bw})$: This part states the effects of the actions. Note that the simple form that is used here is only appropriate for STRIPS domains where the effects are not conditional. Blocks world planning using encodings that have conditional effects – as in $\mathcal{A}$ and ADL – will be discussed in Section 5.3.5.

$causes(pick\_up(X), neg(ontable(X)))$.
$causes(pick\_up(X), neg(clear(X)))$.
$causes(pick\_up(X), holding(X))$.
$causes(pick\_up(X), neg(handempty))$.

$causes(put\_down(X), ontable(X))$.
$causes(put\_down(X), clear(X))$.
$causes(put\_down(X), neg(holding(X)))$.
$causes(put\_down(X), handempty)$.

$causes(stack(X,Y), neg(holding(X)))$.
$causes(stack(X,Y), neg(clear(Y)))$.

$causes(stack(X,Y), clear(X))$.
$causes(stack(X,Y), handempty)$.
$causes(stack(X,Y), on(X,Y))$.

$causes(unstack(X,Y), holding(X))$.
$causes(unstack(X,Y), clear(Y))$.
$causes(unstack(X,Y), neg(clear(X)))$.
$causes(unstack(X,Y), neg(handempty))$.
$causes(unstack(X,Y), neg(on(X,Y)))$.

(d) The initial state $\pi^{dep.init}_{strips1}(O_{bw.1})$: This part defines the initial state by explicitly listing which fluents are true in the initial state. It is assumed that the fluents not explicitly listed to be true are false in the initial state. Thus the knowledge about the initial state is assumed to be complete.

$initially(handempty)$.
$initially(clear(a))$.
$initially(clear(c))$.
$initially(ontable(c))$.
$initially(ontable(b))$.
$initially(on(a,b))$.

(e) The goal conditions $\pi^{dep.goal}_{strips1}(G_{bw.1})$: This part lists the fluent literals that must hold in a goal state.

$finally(on(a,c))$.
$finally(on(c,b))$.

2. The domain independent part $\pi^{indep}_{strips1}$: As mentioned before and is evident from the label, this part is independent of the content of particular domain. It does depend on the assumption that the domain dependent part is a STRIPS problem.

(a) Defining time: In answer set planning, we need to give either the exact bound or at least an upper bound of the plan lengths that we want to consider. This is what makes each answer set finite. The encoding $\pi^{indep}_{strips1}$ depends on a constant referred to as *length* which is the upper bound of the length of plans that we intend to consider. Using this *length* we define a predicate *time* which specifies the times points of our interest.

$time(1)$. ... $time(length)$.

(b) Defining $goal(T)$: The following rules define when all the goal conditions are satisfied.

$not\_goal(T) \leftarrow time(T), finally(X), \mathbf{not}\ holds(X,T)$
$goal(T) \leftarrow time(T), \mathbf{not}\ not\_goal(T)$

(c) Eliminating possible answer sets which do not have a plan of the given length: The following constraint eliminates possible answer sets where the goal is not satisfied in the last time point of interest.

$\leftarrow \mathbf{not}\ goal(length)$.

(d) Defining contrary: The following facts define when a fluent literal is the negation of the other.

$contrary(F, neg(F))$.
$contrary(neg(F), F)$.

(e) Defining executability: The following two rules use the executability conditions to define when an action A is executable in a time T. Note that we are only interested in times before the time point denoted by *length*, as no actions are supposed to occur after that.

$not\_executable(A,T) \leftarrow exec(A,F), \textbf{not}\ holds(F,T).$
$executable(A,T) \leftarrow T < length, \textbf{not}\ not\_executable(A,T).$

(f) Fluent values at time-point 1:

$holds(F,1) \leftarrow initially(F).$

(g) Effect axiom: The following rule describes the change in fluent values due to the execution of an action.

$holds(F,T+1) \leftarrow T < length, executable(A,T), occurs(A,T), causes(A,F).$

(h) Inertia: The following rule describes which fluents do not change there values after an action is executed. In the literature, this is referred to as the frame axiom.

$holds(F,T+1) \leftarrow contrary(F,G), T < length, holds(F,T), \textbf{not}\ holds(G,T+1).$

(i) Occurrences of actions: The following rules enumerate action occurrences. They encode that in each answer set at each time point only one of the executable actions occurred. Also, for each action that is executable in an answer set at a time point, there is an answer set where this action occurs at that time point.

$occurs(A,T) \leftarrow action(A), time(T), \textbf{not}\ goal(T), \textbf{not}\ not\_occurs(A,T).$
$not\_occurs(A,T) \leftarrow action(A), action(AA), time(T), occurs(AA,T), A \neq AA.$
$\leftarrow action(A), time(T), occurs(A,T), \textbf{not}\ executable(A,T).$

Proposition 82 Let D_{bw} be the consistent domain description obtained from the STRIPS/PDDL blocks world specification in Section 5.3.1. Let O_{bw} be a set of observations about the initial state obtained from a STRIPS/PDDL blocks world specification of the initial state, $G_{bw} = \{g_1, \ldots, g_m\}$ be the set of literals that are obtained from a STRIPS/PDDL blocks world specification of the goal, and *length* be a number denoting the upper bound of plan lengths that we are interested in. For any $n < length$,

$\forall k.1 \leq k \leq m\ (D_{bw} \models_{Comp(O_{bw})} g_k\ \textbf{after}\ \ a_1, \ldots, a_n)$ and
$\forall j < n\ \ (\exists l.1 \leq l \leq m\ D_{bw} \not\models_{Comp(O_{bw})} g_l\ \textbf{after}\ \ a_1, \ldots, a_j)$ iff
$\pi_{strips1}(D_{bw}, O_{bw}, G_{bw})$ has an answer set A with $\{occurs(a_1,1), \ldots, occurs(a_n,n)\}$ as the set of facts about *occurs* in it. □

Note that unlike the choice rules in $\pi_{2.planning}$ from Section 5.1.10 we can not split $\pi_{strips1}(D_{bw}, O_{bw}, G_{bw})$ with 2 the first two rules of 2 (i) at the bottom. This makes the proof of the above proposition harder.

Exercise 17 Replace 2(i) in $\pi_{strips1}(D_{bw}, O_{bw}, G_{bw})$ so that the later can be split with the rules defining *occurs* and *not_occurs* in the bottom part. □

5.3.3 Simple blocks World with domain constraints

Planning in the blocks world domain in the general case is known to be NP-complete. One way for efficient blocks world planning is to use domain dependent knowledge about the blocks world to cut down on the search for the right action in each step. A particular set of such knowledge is

based on defining the notion of good and bad towers and put temporal constraints on the plans so that good towers are not destroyed, bad towers are not further built up, and blocks are not held if they have to be finally on top of another block and the later block is not on top of a good tower yet.

A good tower is defined as a tower whose top block is not required to be held in the goal state and the tower below it is not a bad tower. The tower below a block (X) is said to be a bad tower if one of the following holds. (i) X is on the table and it is supposed to be on top of another block in the goal state; (ii) X is on top of block Y, and X is supposed to be on the table or supposed be held in the goal state; (iii) X is on top of block Y, and Y is supposed to have nothing on top in the final state; (iv) X is on top of block Y, and X is supposed to be on top of some other block in the final state; (v) X is on top of block Y, and some other block is supposed to be on top of Y in the final state; (vi) X is on top of block Y, and there is a bad tower below Y.

We now first present the above knowledge about good towers and bad towers using AnsProlog rules; and then represent the temporal domain constraints as integrity constraints with the intention that it will speed up the process of finding an answer set by eliminating non-answer sets earlier in the process of determining that it does not lead to a plan. We refer to the resulting program as $\pi_{strips.cons1}(D_{bw}, O_{bw}, G_{bw})$.

1. Defining *bad_tower_below*

$$holds(bad\_tower\_below(X), T) \leftarrow holds(ontable(X), T), finally(on(X, Y)).$$
$$holds(bad\_tower\_below(X), T) \leftarrow holds(on(X, Y), T), finally(ontable(X)).$$
$$holds(bad\_tower\_below(X), T) \leftarrow holds(on(X, Y), T), finally(holding(Y)).$$
$$holds(bad\_tower\_below(X), T) \leftarrow holds(on(X, Y), T), finally(clear(Y)).$$
$$holds(bad\_tower\_below(X), T) \leftarrow holds(on(X, Y), T), finally(on(X, Z)), Z \neq Y.$$
$$holds(bad\_tower\_below(X), T) \leftarrow holds(on(X, Y), T), finally(on(Z, Y)), Z \neq X.$$
$$holds(bad\_tower\_below(X), T) \leftarrow holds(on(X, Y), T), holds(bad\_tower\_below(Y), T).$$

2. Defining *goodtower*

$$holds(goodtower(X), T) \leftarrow holds(clear(X), T), \textbf{not } finally(holding(X)), \textbf{not } holds(bad\_tower\_below(X), T).$$

3. Defining *badtower*

$$holds(badtower(X), T) \leftarrow holds(clear(X), T), \textbf{not } holds(goodtower(X), T).$$

4. The temporal constraints.

$$\leftarrow holds(goodtower(X), T), holds(on(Y, X), T + 1), \textbf{not } holds(goodtower(Y), T + 1).$$
$$\leftarrow holds(badtower(X), T), holds(on(Y, X), T + 1).$$
$$\leftarrow holds(ontable(X, T)), finally(on(X, Y)), \textbf{not } holds(goodtower(Y), T), holds(holding(X), T + 1).$$

Proposition 83 Let D_{bw} be the consistent domain description obtained from the STRIPS/PDDL blocks world specification in Section 5.3.1. Let O_{bw} be a set of observations about the initial state obtained from a STRIPS/PDDL blocks world specification of the initial state, $G_{bw} = \{g_1, \ldots, g_m\}$ be the set of literals that are obtained from a STRIPS/PDDL blocks world specification of the goal,

and *length* be a number denoting the upper bound of plan lengths that we are interested in. For any $n < length$,

$\forall k.1 \leq k \leq m$ $(D_{bw} \models_{Comp(O_{bw})} g_k$ **after** $a_1, \ldots, a_n)$ and
$\forall j < n$ $(\exists l.1 \leq l \leq m\ D_{bw} \not\models_{Comp(O_{bw})} g_l$ **after** $a_1, \ldots, a_j)$ iff
$\pi_{strips.cons.1}(D_{bw}, O_{bw}, G_{bw})$ has an answer set A with $\{occurs(a_1, 1), \ldots, occurs(a_n, n)\}$ as the set of facts about *occurs* in it. □

The temporal conditions in (4) above can be replaced by the following direct constraints on action occurrences. This further cuts down on the planning time.

$\leftarrow holds(goodtower(X), T), occurs(stack(Y, X), T),$ **not** $holds(goodtower(Y), T + 1).$
$\leftarrow holds(badtower(X), T), occurs(stack(Y, X), T).$
$\leftarrow holds(ontable(X, T)), finally(on(X, Y)),$ **not** $holds(goodtower(Y), T), occurs(pick\_up(X), T).$

5.3.4 Adding defined fluents, qualification and ramification to STRIPS

In this section we consider representing the Blocks World planning problem in a richer language that allows defined fluents, qualification and ramification constraints. We separate the fluents to two categories: basic fluents, and defined fluents. Intuitively, the defined fluents are completely defined in terms of the basic fluents and thus if our formulation includes those definition, then when expressing the effect of actions we only need to express the effect on the basic fluents. The qualification and ramification constraints (also referred to as static causal laws) state causal relationship between the basic fluents and by having them we can further simplify the effect axioms and executability conditions.

In the blocks world domain of Section 5.3.1 the fluents $on(X)$, $ontable(X)$ and $holding(X)$ can be considered as basic fluents, while the fluents $clear(X)$ and *handempty* can be thought of as defined fluents. The intuition is that we can define the later in terms of the former. For example, $clear(X)$ is true in a state iff there does not exist any block Y such that $on(Y, X)$ in that state. Similarly, *handempty* is true in a state iff there does not exists a block Y such that $holding(Y)$ is true in that state.

We refer to this richer language as ADL1, meaning 'action description language 1'. In this richer language the Blocks world domain D_{bw1} can be described as follows:

1. Sort: $block(X)$
2. Basic fluents: $on(X), ontable(X), holding(X)$.
3. Defined fluents: $clear(X), handempty$.
4. Definition of defined fluents.
   - $clear(X)$ iff $\nexists Y : on(Y, X)$
   - $handempty$ iff $\nexists Y : holding(Y)$
5. Executability Conditions:

   executable $pick\_up(X)$ **if** $clear(X), ontable(X), handempty$
   executable $put\_down(X)$ **if** $holding(X)$

executable $stack(X,Y)$ **if** $holding(X), ontable(Y)$
executable $stack(X,Y)$ **if** $holding(X), on(Y,Z)$
executable $unstack(X,Y)$ **if** $clear(X), on(X,Y), handempty$

The difference between the the above conditions and the executability conditions of Section 5.3.1 are the third and fourth conditions above, where the condition $clear(Y)$ is removed, and $ontable(Y) \vee on(Y,Z)$ is added. $clear(Y)$ is removed to demonstrate the usefulness of *qualification* constraints. The rule is split to two and we add the conditions $ontable(Y)$ and $on(Y,Z)$ so as to prevent the block X from being stacked on a non-existing block.

6. Effect propositions:

   $pick\_up(X)$ **causes** $holding(X)$
   $put\_down(X)$ **causes** $ontable(X)$
   $stack(X,Y)$ **causes** $on(X,Y)$
   $unstack(X,Y)$ **causes** $holding(X)$

7. Causal ramification constraints:

   $on(X,Y)$ **causes** $\neg holding(X)$
   $holding(X)$ **causes** $\neg on(X,Y)$
   $ontable(X)$ **causes** $\neg holding(X)$
   $holding(X)$ **causes** $\neg ontable(X)$

8. Causal qualification constraint:

   $on(X,Y), on(Z,Y), X \neq Z$ **causes** $false$

The representation of a set of initial conditions (O_{bw1}) and a goal (G_{bw1}) is similar to that of $O_{bw.1}$ and $G_{bw.1}$, resp. from Section 5.3.1.

We now describe how we can use AnsProlog to encode planning in an ADL1 domain. Given a domain D_{bw1}, a set of initial conditions O_{bw1} and a goal G_{bw1}, our AnsProlog encoding will be referred to as $\pi_{adl1}(D_{bw1}, O_{bw1}, G_{bw1})$ and as in Section 5.3.2 our encoding will consist of two parts, the domain dependent part and the domain independent part.

1. The domain dependent part $\pi^{dep}_{adl1}(D_{bw1}, O_{bw1}, G_{bw1})$:

   (a) The domain: $\pi^{dep.dom}_{adl1}(D_{bw1})$: This part defines the objects in the world, the basic fluents, the defined fluents and actions. The definition for blocks and actions are as in $\pi^{dep.dom}_{strips}(D_{bw})$ of Section 5.3.2. We use the predicate $fluent$ to denote the basic fluents, and the predicate $defined\_fluent$ to denote defined fluents.

   $fluent(on(X,Y))$.
   $fluent(ontable(X))$.
   $fluent(holding(X))$.

   $defined\_fluen(clear(X))$.
   $defined\_fluent(handempty)$.

(b) The executability conditions $\pi^{dep.exec}_{adl1}(D_{bw1})$: The executability conditions in D_{bw1} are more general than in D_{bw} in the sense that in D_{bw1} the action $stack(X,Y)$ has two executability conditions while in D_{bw} each action has only one executability condition. In the following encoding instead of representing executability conditions as facts and then having domain independent rules to define when an action is executable, we directly translate the executability conditions to rules.

$executable(pick\_up(X),T) \leftarrow T < length, holds(clear(X),T), holds(ontable(X),T), holds(handempty,T).$
$executable(put\_down(X),T) \leftarrow T < length, holds(holding(X),T).$
$executable(stack(X,Y),T) \leftarrow T < length, holds(holding(X),T), holds(ontable(Y),T).$
$executable(stack(X,Y),T) \leftarrow T < length, holds(holding(X),T), holds(on(Y,Z),T).$
$executable(unstack(X,Y),T) \leftarrow T < length, holds(clear(X),T), holds(on(X,Y),T), holds(handempty,T).$

(c) The dynamic causal laws $\pi^{dep.dyn}_{adl1}(D_{bw1})$: The use of defined fluents and causal ramification constraints drastically cuts down on the number of dynamic causal laws. Instead of 18 of them, we now have only 4 of them.

$causes(pick\_up(X), holding(X)).$
$causes(put\_down(X), ontable(X)).$
$causes(stack(X,Y), on(X,Y)).$
$causes(unstack(X,Y), holding(X)).$

(d) Defined fluents $\pi^{dep.def}_{adl1}(D_{bw1})$: The following rules define the defined fluents in terms of the basic fluents.

$holds(neg(clear(X)),T) \leftarrow holds(holding(X),T).$
$holds(neg(clear(X)),T) \leftarrow holds(on(Y,X),T).$
$holds(clear(X),T) \leftarrow holds(ontable(X),T), \textbf{not}\ holds(neg(clear(X)),T).$
$holds(clear(X),T) \leftarrow holds(on(X,Y),T), \textbf{not}\ holds(neg(clear(X)),T).$
$holds(neg(handempty),T) \leftarrow holds(holding(X),T).$
$holds(handempty,T) \leftarrow \textbf{not}\ holds(neg(handempty),T).$

(e) Qualification constraints $\pi^{dep.qual}_{adl1}(D_{bw1})$

$\leftarrow holds(on(X,Y),T), holds(on(Z,Y),T), neq(X,Z).$

(f) Static ramification constraints $\pi^{dep.ram}_{adl1}(D_{bw1})$

$static\_causes(on(X,Y), neg(holding(X))).$
$static\_causes(holding(X), neg(on(X,Y))).$
$static\_causes(ontable(X), neg(holding(X))).$
$static\_causes(holding(X), neg(ontable(X))).$

(g) The initial state $\pi^{dep.init}_{adl1}(O_{bw1})$: The initial state can be defined as in $\pi^{dep.init}_{strips}(O_{bw})$ of Section 5.3.2, or we may simplify it by only stating the truth about the basic fluents.

(h) The goal state $\pi^{dep.goal}_{adl1}(G_{bw1})$: The goal state is defines exactly as in $\pi^{dep.init}_{strips}(G_{bw})$) of Section 5.3.2.

2. The domain independent part π^{indep}_{adl1}.

(a) Defining *time*: Same as in 2 (a) of Section 5.3.2.

(b) Defining $goal(T)$: Same as in 2 (b) of Section 5.3.2.

(c) Defining plan existence: Same as in 2 (c) of Section 5.3.2.

(d) Defining contrary: Same as in 2 (d) of Section 5.3.2.

(e) Fluent values at time point 1: Same as in 2 (f) of Section 5.3.2.

(f) Defining literal: The following two rules define the predicate *literal* in terms of the predicate $fluent$. Thus $literal(F)$ denotes that F is a literal made up of basic fluent.

$literal(G) \leftarrow fluent(G).$

$literal(neg(G)) \leftarrow fluent(G).$

(g) Effect axioms: Unlike in 2(g) of Section 5.3.2 the effect axioms only define the effect of actions on literals made up of basic fluents.

$holds(F, T+1) \leftarrow literal(F), T < length, executable(A, T), occurs(A, T), causes(A, F).$

(h) Inertia: Unlike in 2(h) of Section 5.3.2 the inertia axioms are also defined only with respect to literals made up of basic fluents.

$holds(F, T+1) \leftarrow literal(F), contrary(F, G), T < length, holds(F, T),$
not $holds(G, T+1).$

(i) Effect axiom for static causal laws:
$holds(F, T) \leftarrow T < length, holds(G, T), static\_causes(G, F).$

(j) Occurrences of actions: Same as in 2 (i) of Section 5.3.2.

Proposition 84 Let D_{bw1} be the consistent domain description in Section 5.3.4. Let O_{bw1} be a set of observations about the initial state, $G_{bw1} = \{g_1, \ldots, g_m\}$ be a set of literals specifying the goal, and *length* be a number denoting the upper bound of plan lengths that we are interested in. For any $n < length$,

$\forall k.1 \leq k \leq m$ $(D_{bw1} \models_{Comp(O_{bw1})} g_k$ **after** $a_1, \ldots, a_n)$ and
$\forall j < n$ $(\exists l.1 \leq l \leq m\ D_{bw1} \not\models_{Comp(O_{bw1})} g_l$ **after** $a_1, \ldots, a_j)$ iff
$\pi_{adl1}(D_{bw1}, O_{bw1}, G_{bw1})$ has an answer set A with $\{occurs(a_1, 1), \ldots, occurs(a_n, n)\}$ as the set of facts about *occurs* in it. □

5.3.5 Allowing Conditional Effects

In this section we consider another dimension in enriching STRIPS. We allow conditional effects and refer to this language as ADL2. In the ADL2 representation of the blocks world problem instead of having four different actions we only need a single action $puton(X, Y)$. Intuitively, in the action $puton(X, Y)$, X is a block and Y can be either a block or the table, and $puton(X, Y)$ means that we put the block X on top of Y. The executability conditions and effects of this action can be described by the following domain description D_{bw2}.

executable $puton(X, Y)$ **if** $clear(X), clear(Y), X \neq Y \neq table$
executable $puton(X, table)$ **if** $clear(X), X \neq table$

$puton(X, Y)$ **causes** $on(X, Y)$ **if** $X \neq Y, X \neq table).$
$puton(X, Y)$ **causes** $\neg on(X, Z)$ **if** $on(X, Z), X \neq Y \neq Z.$
$puton(X, Y)$ **causes** $clear(Z)$ **if** $on(X, Z), X \neq Y \neq Z, X \neq table, Z \neq table.$
$puton(X, Y)$ **causes** $\neg clear(Y)$ **if** $X \neq Y \neq table$

In the above description we have to take special care as Y in $puton(X,Y)$ can be either a block or the table, and we need to distinguish between $clear(X)$ when X is a block and $clear(table)$. In particular, we need to encode that putting a block on the table neither affects nor depends on $clear(table)$.

As before we can use AnsProlog to encode planning in ADL2. Given a domain D_{bw2}, a set of initial conditions O_{bw2} and a goal G_{bw2}, our AnsProlog encoding will be referred to as $\pi_{adl2}(D_{bw2}, O_{bw2}, G_{bw2})$ and as in Section 5.3.2 our encoding will consist of two parts, the domain dependent part and the domain independent part.

1. Domain dependent part $\pi^{dep}_{adl2}(D_{bw2}, O_{bw2}, G_{bw2})$:

   (a) The domain, initial state and goal conditions are expressed as in 1(a), 1(d) and 1(e) of Section 5.3.2, respectively.

   (b) The executability conditions $\pi^{dep.exec}_{adl2}(D_{bw2})$:

   $exec(puton(X,Y),[clear(X),clear(Y)]) \leftarrow neq(X,Y), neq(X,table), neq(Y,table).$
   $exec(puton(X,Y),[clear(X)]) \leftarrow neq(X,Y), eq(Y,table).$

   (c) The dynamic causal laws $\pi^{dep.dyn}_{adl2}(D_{bw2})$:

   $causes(puton(X,Y), on(X,Y), []) \leftarrow neq(X,Y), neq(X,table).$
   $causes(puton(X,Y), neg(on(X,Z)), [on(X,Z)]) \leftarrow neq(X,Y), neq(X,Z), neq(Z,Y).$
   $causes(puton(X,Y), clear(Z), [on(X,Z)]) \leftarrow neq(X,Y), neq(X,Z), neq(Z,Y), neq(Z,table), neq(X,table).$
   $causes(puton(X,Y), neg(clear(Y)), []) \leftarrow neq(X,Y), neq(Y,table), neq(X,table).$

2. Domain independent part π^{indep}_{adl2}:

   The definition of time, the definition of $goal(T)$, the constraint that eliminates non-plans, the definition of contrary, the rule defining fluent values at time point 1, the inertia rule, and the rules that enumerate action occurrences are exactly the same as in 2(a), 2(b), 2(c), 2(d), 2(f), 2(h) and 2(i), of Section 5.3.2 respectively. The only changes are to 2(e) and 2(g) that define executability and effect of actions. In addition we need to define when a set of fluent literals hold at a time point.

   (a) Defining when a set of fluent holds at a time point:

   $not\_holds\_set(S,T) \leftarrow literal(L), in(L,S), notholds(L,T).$

   $holds\_set(S,T) \leftarrow notnot\_holds\_set(S,T).$

   (b) Defining executability:
   $executable(A,T) \leftarrow T < length, exec(A,S), holds\_set(S,T).$

   (c) Effect of actions:

   $holds(F,T{+}1) \leftarrow T < length, action(A), executable(A,T), occurs(A,T), causes(A,F,S), holds\_set(S,T).$

One of the plans generated by this AnsProlog program with $length = 5$ is as follows:

$occurs(puton(c,table),1)$
$occurs(puton(b,c),2)$
$occurs(puton(a,b),3)$

Proposition 85 Let D_{bw2} be the consistent domain description in Section 5.3.5. Let O_{bw2} be a set of observations about the initial state, $G_{bw2} = \{g_1, \ldots, g_m\}$ be a set of literals specifying of the goal, and *length* be a number denoting the upper bound of plan lengths that we are interested in. For any $n < length$,

$\forall k.1 \leq k \leq m$ $(D_{bw2} \models_{Comp(O_{bw2})} g_k$ **after** $a_1, \ldots, a_n)$ and
$\forall j < n$ $(\exists l.1 \leq l \leq m\ D_{bw2} \not\models_{Comp(O_{bw2})} g_l$ **after** $a_1, \ldots, a_j)$ iff
$\pi_{adl2}(D_{bw2}, O_{bw2}, G_{bw2})$ has an answer set A with $\{occurs(a_1, 1), \ldots, occurs(a_n, n)\}$ as the set of facts about *occurs* in it. □

5.3.6 Navigating a downtown with one-way streets

In the section we consider another domain in a language similar to STRIPS, the only difference being that instead of describing when actions are executable, in this language conditions are given when actions are not executable. We refer to this language as STRIPS2. The domain we consider is the domain of navigating a downtown area with one-way streets in a vehicle. The only action in this domain is $move(V, L1, L2)$ and the the fluents are $at(V, L)$ and $edge(L1, L2)$. Given the domain (D_{nav}), the initial position of a vehicle and the one-way description (O_{nav}), and the final location of the vehicle (G_{nav}), we would like to find a plan to get to the final location from the initial location obeying the one-way descriptions. Here, we directly give the encoding $\pi_{strips2}(D_{nav}, O_{nav}, G_{nav})$.

1. The domain dependent part: $\pi^{dep}_{strips2}(D_{nav}, O_{nav}, G_{nav})$

   (a) The domain $\pi^{dep.dom}_{strips2}(D_{nav})$:

   $vehicle(v)$

   $location(l_1). \ldots location(l_{12}).$

   $fluent(at(V, L)) \leftarrow vehicle(V), location(L).$

   $action(move(V, L1, L2)) \leftarrow vehicle(V), location(L1), location(L2).$

   (b) When actions are not executable $\pi^{dep.exec}_{strips2}(D_{nav})$:

   $impossible\_if(move(V, L1, L2), neg(at(V, L1))).$
   $impossible\_if(move(V, L1, L2), neg(edge(L1, L2))).$

   It should be noted that unlike 1(c) of Section 5.3.2, here we express when an action is impossible to execute instead of saying when they are executable.

   (c) The effect of actions $\pi^{dep.dyn}_{strips2}(D_{nav})$:

   $causes(move(V, L1, L2), at(V, L2)).$
   $causes(move(V, L1, L2), neg(at(V, L1))).$

   (d) The initial street description: and the initial position of the vehicle $\pi^{dep.init}_{strips2}(O_{nav})$:

   $initially(edge(l_1, l_2)).$
   $initially(edge(l_2, l_3)).$
   $initially(edge(l_3, l_4)).$
   $initially(edge(l_4, l_8)).$
   $initially(edge(l_8, l_7)).$
   $initially(edge(l_7, l_6)).$

$initially(edge(l_6, l_5)).$
$initially(edge(l_5, l_1)).$
$initially(edge(l_2, l_6)).$
$initially(edge(l_7, l_3)).$
$initially(edge(l_1, l_9)).$
$initially(edge(l_9, l_{10})).$
$initially(edge(l_{10}, l_{11})).$
$initially(edge(l_{11}, l_{12})).$
$initially(edge(l_{12}, l_2)).$
$initially(edge(l_{12}, l_9)).$

$initially(at(v, l_3)).$

(e) The goal state $\pi^{dep.goal}_{strips2}(G_{nav})$:

$finally(at(v, l_2)).$

2. The domain independent part $\pi^{indep}_{strips2}$:

   (a) The rules in 2(a)-2(d) and 2(g)-2(i) of Section 5.3.2 belong to the domain independent part.

   (b) Instead of 2(e) of Section 5.3.2 we need new rules that define executability in terms of the *impossible_if* conditions given in the domain dependent part. These rules are:

   $not\_executable(A, T) \leftarrow impossible\_if(A, B), holds(B, T)$

   $executable(A, T) \leftarrow \textbf{not}\ not\_executable(A, T)$

   (c) In lieu of 2(f) Section 5.3.2 we have two rules defining both what holds and what does not hold in time point 1. Thus the following substitutes 2(f) of Section 5.3.2.

   $holds(F, 1) \leftarrow initially(F)$

   $holds(neg(F), 1) \leftarrow \textbf{not}\ holds(F, 1)$

One of the plan generated by this AnsProlog program with $length = 9$ is as follows:

$occurs(move(v, l_3, l_4), 1).$
$occurs(move(v, l_4, l_8), 2).$
$occurs(move(v, l_8, l_7), 3).$
$occurs(move(v, l_7, l_6), 4).$
$occurs(move(v, l_6, l_5), 5).$
$occurs(move(v, l_5, l_1), 6).$
$occurs(move(v, l_1, l_2), 7).$

5.3.7 Downtown navigation: planning while driving

Consider the case that an agent uses the planner in the previous section and makes a plan. It now executes part of the plan, and hears in the radio that *an accident occurred between point* l_1 *and* l_2 and that section of the street is blocked. The agent now has to make a new plan from where it is to its destination. To be able to encode observations and make plans from the current situation we need to add the following to our program in the previous section.

1. The domain dependent part
   (a) The observations

   $happened(move(v, l_3, l_4), 1).$

   $happened(move(v, l_4, l_8), 2).$

   $happened(acc(l_1, l_2), 3).$

   (b) Exogenous actions

   $causes(acc(X, Y), neg(edge(X, Y)))$

2. The domain independent part
   (a) Relating *happened* and *occurs*

   $occurs(A, T) \leftarrow happened(A, T).$

With these additions one of the plan generated by this AnsProlog program with $length = 13$ is as follows:

$occurs(move(v, l_8, l_7), 4).$
$occurs(move(v, l_7, l_6), 5).$
$occurs(move(v, l_6, l_5), 6).$
$occurs(move(v, l_5, l_1), 7).$
$occurs(move(v, l_1, l_9), 8).$
$occurs(move(v, l_9, l_{10}), 9).$
$occurs(move(v, l_{10}, l_{11}), 10).$
$occurs(move(v, l_{11}, l_{12}), 11).$
$occurs(move(v, l_{12}, l_2), 12).$

Although it does not matter in the particular example described above, we should separate the set of actions to *agent_actions* and *exogenous_actions*, and in the planning module require that while planning we only use *agent_actions*. This can be achieved by replacing the first and third rule about *occurs* in 2(i) of Section 5.3.2 by the following two rules.

1. $occurs(A, T) \leftarrow time(T), happened(A, T).$
2. $occurs(A, T) \leftarrow agent\_action(A), time(T), executable(A, T), \textbf{not}\ goal(T), \textbf{not}\ not\_occurs(A, T)$

5.4 Approximate planning when initial state is incomplete

The planning encodings in Section 5.3 assume that the initial state is complete. When we remove this assumption those encodings are no longer appropriate. In fact since planning in this case belongs to a complexity class that is not expressible in AnsProlog, in general we can not have a sound and complete AnsProlog encoding that will give us all the plans. The best we can do is to have encodings that find at least some of the plans. We present such an encoding here and show that the encoding is sound in the sense that the set of *occurs* fact in any answer set of this encoding does indeed give us a plan. There may not be answer sets corresponding to some plans though. In that sense it is not complete. We consider the language from Section 5.3.5 and remove the completeness

assumption about the initial state, and allow fluents to be unknown in the goal state. We refer to this language as ADL3. Given a domain description D, a set of initial state observations O, and a set of 3-valued ($f, neg(f)$, and $unk(f)$) fluent literals G, our encoding $\pi_{adl3}(D, O, G)$ consist of the following:

1. Domain dependent part $\pi^{dep}_{adl3}(D, O, G)$:

   (a) The domain is expressed similar to 1(a) of Section 5.3.5.

   (b) The executability conditions and dynamic causal laws are similar to 1(b) and 1(c) of Section 5.3.5.

   (c) The initial state $\pi^{dep.init}_{adl3}(O)$: There is no longer the assumption that the initial state is complete. Hence, the initial state is a set of atoms of the form $initially(l)$, where l is a fluent literal.

   (d) The goal state $\pi^{dep.goal}_{adl3}(G)$: In addition to specifying that certain fluents should be true, and certain others should be false in the final state, we may say that truth value of certain fluents be *unknown* in the final state. In that case we say:

   $finally(unk(f))$.

2. Domain independent part $\pi^{indep}_{adl3}(D, O, G)$:

   The definition of time, the definition of $goal(T)$, the constraint that eliminates non-plans, the definition of contrary, the rule defining fluent values at time point 1, and the rules that enumerate action occurrences are exactly the same as in 2(a), 2(b), 2(c), 2(d), 2(f), and 2(i) of Section 5.3.2 respectively.

   The definition of when a set of fluents hold at a time point, the definition of executability, and the effect axiom are exactly the same as in 2(a), 2(b) and 2(c) of Section 5.3.5 respectively.

   The inertia rule is different, and we have additional rules for blocking inertia, rules that define when a set of fluent literals may hold at a time point, and rules that define when a fluent is unknown at a time point. These rules are given below:

   (a) Defining abnormality:

   $ab(F, T+1) \leftarrow T < length, action(A), executable(A,T), occurs(A,T), causes(A,F,S),$
   $m\_holds\_set(S,T).$

   (b) Inertia:

   $holds(F, T+1) \leftarrow contrary(F,G), T < length, holds(F,T), \mathbf{not}\ ab(G, T+1).$

   The inertia and abnormality rules above are similar to the encoding in Section 5.1.4. The purpose is to be conservative in using the inertia rules. Thus, if we have an effect proposition a **causes** $\neg f$ **if** $p_1, \ldots p_n$, and if there is a possibility that $p_1, \ldots p_n$ may hold in time T, f is inferred to be abnormal in time T+1. This blocks the inference of f being true in time T+1 due to inertia.

   (c) Defining when a set of fluents may hold:
   $m\_not\_holds\_set(S,T) \leftarrow in(L,S), contrary(L,LL), holds(LL,T).$
   $m\_holds\_set(S,T) \leftarrow \mathbf{not}\ m\_not\_holds\_set(S,T).$

(d) Defining when a fluent value is unknown:

$$holds(unk(F), T) \leftarrow \textbf{not } holds(F, T), contrary(F, G), \textbf{not } holds(G, T).$$

Proposition 86 Let D be a consistent domain description obtained from an ADL3 specification, Let O be a set of observations about the initial state, and $G = \{g_1, \ldots, g_m\}$ be a set of fluent literals, and *length* be a number denoting the upper bound of plan lengths that we are interested in. For any $n < length$,

$\pi_{adl3}(D, O, G)$ has an answer set A with $\{occurs(a_1, 1), \ldots, occurs(a_n, n)\}$ as the set of facts about *occurs* in it implies $\forall k.1 \leq k \leq m$ $(D \models_O g_k$ **after** $a_1, \ldots, a_n)$. □

5.5 Planning with procedural constraints

In Section 5.3.3 we discussed the use of temporal constraints in planning in the blocks world domain. In this section we discuss planning in presence of procedural constraints. An example of a simple procedural constraint is $a_1; a_2; (a_3|a_4|a_5); \neg f$. Intuitively, it means that the plan must have a_1 as the first action, a_2 as the second action, one of a_3, a_4 and a_5 as the third action, and $\neg f$ must be true after the third action. Such procedural constraints allow a domain expert to state a (non-deterministic) plan which is almost like a program in a procedural language except that it may have a few non-deterministic choices. The later are to be explored by the interpreter so as to make the plan executable and make sure the fluent formulas in the procedural constraint are true at the appropriate moment. Our procedural constraints will have the following forms, where a is an action ϕ is a formula, and p, and p_i's are procedural constraints.

- a
- ϕ
- $p_1; \ldots; p_m$ (denoted using list notation)
- $(p_1| \ldots |p_n)$ (denoted using set notation)

  In AnsProlog encodings we encode such a procedural constraint by giving it a name say p and have the following facts:

  $choice\_st(p)$.
  $in(p_1, p)$. ... $in(p_n, p)$.

- $if(\phi, p_1, p_2)$
- $while(\phi, p)$
- $choice\_arg(\phi, p)$

Formulas are bounded classical formulas with each bound variable associated with a sort. They are defined as follows:

- a literal is a formula.
- negation of a formula is a formula.

- a finite conjunction of formulas is a formula.
- a finite disjunction of formulas is a formula.
- If $X_1, \ldots, X_n$ are variables that can have values from the sorts $s_1, \ldots, s_n$, and $f_1(X_1, \ldots, X_n)$ is a formula then $\forall X_1, \ldots, X_n.f_1(X_1, \ldots, X_n)$ is a formula.

  In AnsProlog encodings we encode such a formula by giving it a name say f and have the following rule:

  $forall(f, f_1(X_1, \ldots, X_n)) \leftarrow in(X_1, s_1), \ldots, in(X_n, s_n)$
- If $X_1, \ldots, X_n$ are variables that can have values from the sorts $S_1, \ldots, S_n$, and $F_1(X_1, \ldots, X_n)$ is a formula then $\exists X_1, \ldots, X_n.F_1(X_1, \ldots, X_n)$ is a formula.

  In AnsProlog encodings we encode such a formula by giving it a name say f and have the following rule:

  $exists(f, f_1(X_1, \ldots, X_n)) \leftarrow in(X_1, s_1), \ldots, in(X_n, s_n)$

Before we present an AnsProlog encoding that generates plans by exploiting procedural constraints we first give some examples.

Example 106 Consider the following domain description D:

a **causes** $p(f)$ **if** $\neg p(g)$
b **causes** $\neg p(g)$
c **causes** $\neg p(f)$
d **causes** $p(f), p(g), p(h)$
e_1 **causes** $p(g)$
e_2 **causes** $p(h)$
e_3 **causes** $p(f)$
executable $a, b, c, d, e_1, e_2, e_3$

Let the initial state specification O consist of the following:

initially $\neg p(f)$
initially $p(g)$
initially $\neg p(h)$

Consider the following five procedural constraints

1. $p_1 : b; (a|c|d); p(f)$
2. $p_2 : b; (a|c|d); \forall X.[X \in \{f, g, h\}]p(X)$
3. $p_3 : b; (a|c|d); \exists X.[X \in \{f, g, h\}]p(X)$
4. $p_4 : b; (a|c|d); \forall X.[X \in \{f, g, h\}]\neg p(X)$
5. $p_5 : while(\exists X.[X \in \{f, g, h\}]\neg p(X), (a|b|c|e_1|e_2))$

The plans generated by the planning constraint p_1 are $b; a$ and $b; d$. The sequence $b; c$ is not a plan because it does not make $p(f)$ true, thus violating the plan constraint that $p(f)$ must be true at the end. On the other hand $b; e_3$ is not a plan because e_3 does not agree with the plan constraint which requires that the second actions must be one of a, c and d.

Similar, the plan generated by p_2 is $b; d$; the plans generated by p_3 are $b; a$ and $b; d$; and the plan generated by p_4 is $b; c$.

The planning constraint p_5 can be simply considered as a planner for the goal $\exists X.[X \in \{f, g, h\}]\neg p(X)$ using only the actions a, b, c, e_1, and e_2. Thus planning constraint can express classical planning. There are four plans (of length less than 5) that are generated by p_5. They are: (i) $e_2; b; a; e_1$, (ii) $b; e_2; a; e_1$, (iii) $b; a; e_2; e_1$, and (iv) $b; a; e_1; e_2$. The effect of actions are such that to achieve this goal any minimal plan must have b before a and a before e_1. Thus, e_2 can fit in four different slots resulting in four different plans. □

We now present an AnsProlog encoding of planning with procedural constraints and compare it with earlier encodings.

1. The domain dependent part illustrating a particular domain: It consists of five parts: the domain, the executability conditions, the dynamic causal laws, the description of the initial state, and the procedural constraints. Among them, the domain, the description of the initial state, the executability conditions and the dynamic causal laws are similar to the ones in the domain dependent part of Section 5.3.5. The only difference is that instead of goal conditions here we have the more general procedural constraints. We now encode the domain dependent part of planning with respect to the procedural constraints in Example 106.

   (a) The domain:

   $fluent(p(f)).$
   $fluent(p(g)).$
   $fluent(p(h)).$

   $action(a).$
   $action(b).$
   $action(c).$
   $action(d).$
   $action(e1).$
   $action(e2).$

   (b) The executability conditions:

   $exec(a, []).$
   $exec(b, []).$
   $exec(c, []).$
   $exec(d, []).$
   $exec(e1, []).$
   $exec(e2, []).$

   (c) The dynamic causal laws:

   $causes(a, p(f), [neg(p(g))]).$
   $causes(b, neg(p(g)), []).$

$causes(c, neg(p(f)), [])$.
$causes(d, p(f), [])$.
$causes(d, p(g), [])$.
$causes(d, p(h), [])$.
$causes(e1, p(g), [])$.
$causes(e2, p(h), [])$.

(d) Description of the initial state:

$initially(neg(p(f)))$.
$initially(p(g))$.
$initially(neg(p(h)))$.

(e) Five particular procedural constraints: In the following we define five different procedural constraints. In each case they are identified through the predicate $main\_cons$.

i. Representing $b; (a|c|d); p(f)$

$main\_cons([b, choice, p(f)])$
$choice\_st(choice)$.
$in(a, choice)$.
$in(c, choice)$.
$in(d, choice)$.

ii. Representing $b; (a|c|d); \forall X.[X \in \{f, g, h\}]p(X)$

$main\_cons([b, choice, rest3])$
$choice\_st(choice)$.
$in(a, choice)$.
$in(c, choice)$.
$in(d, choice)$.
$forall(rest3, p(X)) \leftarrow in(X, \{f, g, h\})$.

iii. Representing $b; (a|c|d); \exists X.[X \in \{f, g, h\}]p(X)$

$main\_cons([b, choice, rest4])$
$choice\_st(choice)$.
$in(a, choice)$.
$in(c, choice)$.
$in(d, choice)$.
$exists(rest4, p(X)) \leftarrow in(X, \{f, g, h\})$.

iv. Representing $b; (a|c|d); \forall X.[X \in \{f, g, h\}]\neg p(X)$

$main\_cons([b, choice, rest5])$
$choice\_st(choice)$.
$in(a, choice)$.
$in(c, choice)$.
$in(d, choice)$.
$forall(rest5, neg(p(X))) \leftarrow in(X, \{f, g, h\})$.

v. Representing $while(\exists X.[X \in \{f, g, h\}]\neg p(X), ch2)$

$main\_cons(while(rest6, ch2))$
$exists(rest6, neg(p(X))) \leftarrow in(X, \{f, g, h\})$.
$choice\_st(ch2)$.

$in(a, ch2)$.
$in(b, ch2)$.
$in(c, ch2)$.
$in(e1, ch2)$.
$in(e2, ch2)$.

2. Domain independent part: We now present the domain independent part of AnsProlog encodings that can plan using procedural constraints, and make comparison with the domain independent part of the previous encodings.

   (a) Defining time: This is exactly the same as 2(a) of Section 5.3.2.

   $time(1). \quad \ldots \quad time(length).$

   (b) Defining if and when the procedural constraints are satisfied:

   Unlike the goal conditions in Section 5.3.2, the notion of satisfiability of a procedural constraint is not about a particular time point; but about a time interval. Thus the first rule below says that the goal is reached at time point T if the main procedural constraint X is satisfied between 1 and T. The second rule says that if the goal is already reached by time T then it is also reached by time $T+1$. We did not need such a rule in Section 5.3.2 as there the goal was about reaching a state where the goal conditions were satisfied and once such a state was reached, action executions were blocked and the goal remained satisfied in subsequent time points.

   $goal(T) \leftarrow time(T), main\_cons(X), satisfied(X, 1, T).$
   $goal(T+1) \leftarrow time(T), T < length, goal(T).$

   (c) Eliminating possible answer sets that do not satisfy the goal: This is exactly the same as 2(c) of Section 5.3.2.

   $\leftarrow$ **not** $goal(length)$.

   (d) Auxiliary rules:

   $literal(G) \leftarrow fluent(G).$
   $literal(neg(G)) \leftarrow fluent(G).$
   $contrary(F, neg(F)) \leftarrow fluent(F).$
   $contrary(neg(F), F) \leftarrow fluent(F).$
   $def\_literal(G) \leftarrow def\_fluent(G).$
   $def\_literal(neg(G)) \leftarrow def\_fluent(G).$
   $contrary(F, neg(F)) \leftarrow def\_fluent(F).$
   $contrary(neg(F), F) \leftarrow def\_fluent(F).$
   $leq(T, T) \leftarrow$
   $leq(Tb, Te) \leftarrow Tb < Te.$

   (e) Defining executability: This is similar to the rule 2(b) of Section 5.3.5.

   $executable(A, T) \leftarrow action(A), set(S), T < length, exec(A, S), holds\_set(S, T).$

   (f) Fluent values at time point 1: This is similar to the rule 2(f) of Section 5.3.2.

   $holds(F, 1) \leftarrow literal(F), initially(F).$

(g) Effect axiom: This is similar to the rule 2(c) of Section 5.3.5.

$holds(F, T{+}1) \leftarrow literal(F), set(S), T < length, action(A), executable(A,T), occurs(A,T),$
$causes(A,F,S), holds\_set(S,T).$

(h) Static causal laws: This is similar to the rule 2(i) of Section 5.3.4.

$holds(F,T) \leftarrow literal(F), literal(G), T < length, holds(G,T), static\_causes(G,F).$

(i) Inertia: This is exactly same as the rule 2(h) of Section 5.3.2.

$holds(F, T+1) \leftarrow literal(F), literal(G), contrary(F,G), T < length, holds(F,T),$
not $holds(G, T+1).$

(j) Enumeration of action occurrences: This is similar to the rule 2(i) of Section 5.3.2.

$occurs(A,T) \leftarrow action(A), executable(A,T),$ **not** $goal(T),$ **not** $not\_occurs(A,T).$
$not\_occurs(A,T) \leftarrow action(A), action(AA), occurs(AA,T), neq(A,AA).$

(k) Defining satisfaction of procedural constraints: The following rules define when a procedural constraint can be satisfied during the interval Tb and Te. For example, the intuitive meaning of the first rule is that the procedural constraint with the first part as P_1 and the rest as P_2 is satisfied during the interval Tb and Te, if there exists $Te1$, $Tb \leq Te1 \leq Te$ such that P_1 is satisfied in the interval Tb and $Te1$, and P_2 is satisfied in the interval $Te1$ and Te. Similarly, the second rule says that a procedural constraint consisting of only action A is satisfied during the interval Tb and $Tb+1$ is A occurs at time Tb. The intuitive meaning of the other rules are similar.

$satisfied([P1|P2], Tb, Te) \leftarrow leq(Tb,Te), leq(Tb,Te1), leq(Te1,Te),$
$satisfied(P1,Tb,Te1), satisfied(P2,Te1,Te).$
$satisfied(A, Tb, Tb+1) \leftarrow action(A), occurs(A,Tb).$
$satisfied([], Tb, Tb) \leftarrow$
$satisfied(N,Tb,Te) \leftarrow leq(Tb,Te), choice\_st(N), in(P1,N), satisfied(P1,Tb,Te).$
$satisfied(F,Tb,Tb) \leftarrow formula(F), holds\_formula(F,Tb).$
$satisfied(F,Tb,Tb) \leftarrow literal(F), holds(F,Tb).$
$satisfied(if(F,P1,P2),Tb,Te) \leftarrow leq(Tb,Te), holds\_formula(F,Tb), satisfied(P1,Tb,Te).$
$satisfied(if(F,P1,P2),Tb,Te) \leftarrow leq(Tb,Te),$ **not** $holds\_formula(F,Tb), satisfied(P2,Tb,Te).$
$satisfied(while(F,P),Tb,Te) \leftarrow leq(Tb,Te), holds\_formula(F,Tb), leq(Tb,Te1),$
$leq(Te1,Te), satisfied(P,Tb,Te1), satisfied(while(F,P),Te1,Te).$
$satisfied(while(F,P),Tb,Tb) \leftarrow$ **not** $holds\_formula(F,Tb).$
$satisfied(choice\_arg(F,P),Tb,Te) \leftarrow leq(Tb,Te), holds(F,Tb), satisfied(P,Tb,Te).$

(l) Defining when a formula holds: The rules for satisfiability of the 'if' and 'while' procedural constraints need us to define when a formula holds at a time point. We do that using the following rules. Among these rules the non-obvious ones are the rules for formulas with a quantification. Let us consider formulas with an existential quantifier. In that case the existential formula F leads to a collection of ground facts of the form $exists(F,F1)$. The variable instantiations are done in the domain dependent part 1(e) through rules with $exists(F,\_)$ in the head. Thus formula F is said to hold in time T, if there exist an $F1$ such that $exist(F,F1)$ is true and $F1$ holds in T. Similarly a universally quantified formula F is said to hold in time T, if for all $F1$ such that $forall(F,F1)$ is true and $F1$ holds in T.

$holds\_formula(F,T) \leftarrow disj(F), in(F1,F), holds\_formula(F1,T).$
$not\_holds\_conj\_formula(F,T) \leftarrow time(T), conj(F), in(F1,F), \textbf{not}\ holds\_formula(F1,T).$
$holds\_formula(F,T) \leftarrow conj(F), \textbf{not}\ not\_holds\_conj\_formula(F).$
$holds\_formula(F,T) \leftarrow negation(F,F1), \textbf{not}\ holds\_formula(F1,T).$
$holds\_formula(F,T) \leftarrow literal(F), holds(F,T).$
$holds\_formula(F,T) \leftarrow exists(F,F1), holds\_formula(F1,T).$
$not\_holds\_forall\_formula(F,T) \leftarrow forall(F,F1), \textbf{not}\ holds\_formula(F1,T).$
$holds\_formula(F,T) \leftarrow forall(F,F1), \textbf{not}\ not\_holds\_forall\_formula(F,T).$

$forumla(F) \leftarrow disj(F).$
$forumla(F) \leftarrow conj(F).$
$formula(F) \leftarrow literal(F).$
$formula(F) \leftarrow negation(F,F1).$
$formula(F1) \leftarrow negation(F,F1).$
$formula(F) \leftarrow exists(F,F1).$
$formula(F1) \leftarrow exists(F,F1).$
$formula(F) \leftarrow forall(F,F1).$
$formula(F1) \leftarrow forall(F,F1).$

(m) Defining when a set of fluents hold: This is exactly same as the rule 2(a) of Section 5.3.5.

$not\_holds\_set(S,T) \leftarrow set(S), in(L,S), \textbf{not}\ holds(L,T).$
$holds\_set(S,T) \leftarrow set(S), \textbf{not}\ not\_holds\_set(S,T).$

As expected, the above encoding with $length = 5$ give us answer sets that encode the corresponding plans from Example 106.

5.6 Scheduling and planning with action duration

5.7 Explaining observations through action occurrences

5.8 Action based diagnosis

5.9 Reasoning about sensing actions

5.10 Case study: Planning and plan correctness in a Space shuttle reaction control system

The reaction control system (RCS) of a space shuttle has the primary responsibility for maneuvering the shuttle while it is in space. It consists of fuel and oxidizer tanks, valves and other plumbing necessary for the transmission of the propellant to the maneuvering jets of the shuttle and electronic circuits for controlling the valves and fuel lines, and also to prepare the jets to receive firing commands. The RCS is computer controlled during take-off and landing and the astronauts have the primary control during the rest of the flight. Normally the astronauts follow pre-scripted plans. But in presence of failures, the astronauts often have to fall back on the ground flight controllers, as the possible set of failures is to large to have scripts for all of them.

The United space alliance, a major NASA contractor, together with faculty and students from Texas Tech University and the University of Texas at El Paso, built an AnsProlog system to verify

plans and to generate plans for the RCS. Work on its deployment is scheduled to start in December 2000. We will refer to this system as the RCS-AnsProlog system.

The RCS-AnsProlog systems has 3 modeling modules: (i) Plumbing module PM, that models the plumbing system of RCS, (ii) Valve control module VCM that models the valves and the impact of their opening and closing, and (iii) Circuit theory module CTM that models the electrical circuit; and a planning module PM. We now describe each of these modules in greater detail.

The plumbing module PM has a description of the structure of the plumbing systems as a directed graph whose nodes represent tanks, jets and pipe junctions and whose arcs are labeled by valves. Possible faults that may change the graph are leaky valves, damaged jets, and stuck valves. The PM encodes the condition of fluid flow from one node to another node of the graph, which is that there exists a path without leaks from the source to the destination, with all open valves along the path. More generally, the PM encodes a function from the given graph and the state of the valves and faulty components to the pressures through the nodes in the graph, readiness of jets for firing and executability of maneuvers that can be performed. Following is a simplified instance of an AnsProlog rule in the TM, which encodes the condition that a node $N1$ is pressurized by a tank Tk at time T, if $N1$ is not leaking, and is connected by an open valve to a node $N2$ which is pressurized by Tk.

$$holds(pressurized\_by(N1, Tk), T) \leftarrow \textbf{not}\ holds(leaking(N1, T), link(N2, N1, Valve), holds(state\_of(Valve, open), T), holds(pressurized\_by(N1, Tk), T).$$

The valve control module VCM is divided into two parts: the basic VCM and the extended VCM. The basic VCM assumes all electrical circuits connecting switches and computer commands to the valves to be working properly, and does not include them in the representation. The extended VCM includes information about the electrical circuits and is normally used when circuits malfunction. We now describe both parts in greater detail.

The basic VCM encodes a function from initial positions and faults of switches and valves, and the history of the actions and events that have taken place to position of valves at the current moment. The output of this function is used as an input to the plumbing module. The VCM encoding is similar to the encoding of effects of action and inertia rules as described earlier in this chapter. Following are examples of two rules from the basic VCM module:

$$holds(state\_of(Sw, S), T + 1) \leftarrow occurs(flip(Sw, S), T), \textbf{not}\ holds(state\_of(Sw, stuck), T).$$

$$holds(state\_of(V, S), T) \leftarrow controls(Sw, V), holds(state\_of(Sw, S), T), \textbf{not}\ holds(ab\_input(V), T), S \neq no\_con, \textbf{not}\ holds(state\_of(Sw, stuck), T), \textbf{not}\ holds(bad\_circuitry(Sw, V), T).$$

The first rule encodes that a switch Sw is in the state S at time $T + 1$, if it is not stuck at time T, and the action of flipping it to state S occurs at time T. The second rule encodes that under normal conditions – i.e., the circuit connecting Sw and the valve V it controls is working properly, switch Sw is not stuck, and V does not have an abnormal input – if switch Sw that controls valve V is in some state S which is different from no_con then V is also in the same state. Here, a switch can be in one of three positions: open, closed, or no_con. When it is in the state no_con, it has no control over the state of the valve.

The extended VCM is similar to the basic VCM except that it allows additional information about electrical circuits, power and control buses, and the wiring connections among all the components of the system in its input. Part of this input information comes from the CTM which we will discuss later. The output is the same as in the basic VCM. Following is an example of a rule from

the extended VCM which encodes the condition that the value of the output wire W_o of a switch Sw is same as the value of its input wire W_i if the switch is in a state that connects W_i and W_o.

$holds(value(W_o, Val), T) \leftarrow holds(value(W_i, Val), T), holds(state\_of(Sw, S), T), connects(S, S_w, W_i, W_o).$

The circuit theory module CTM models the electrical circuits of the RCS, which are formed by the digital gates and other electrical components, connected by wires. The CTM describes the normal and faulty behavior of electrical circuits with possible propagation delays and 3-valued logic. It encodes a function from the description of a circuit, values of signals present on its input wires and the set of faults affecting its gates to the values on the output wires. Following is an example of a rule from the CTM which encodes the condition that the value of a wire W is X if W is the output of a gate G with delay D , whose input are $W1$ and $W2$, and $W1$ is stuck at X, $W2$ has the value 1, and W is not stuck.

$holds(value(W, Val), T+D) \leftarrow delay(G, D), input(W2, G), input(W1, G), output(W, G), W1 \neq W2,$
$holds(value(W2, 1), T), holds(value(W1, Val), T), holds(stuck(W1), T),$ **not** $holds(stuck(W), T).$

The planning module of RCS-AnsProlog is based on the answer set planning paradigm discussed earlier in this chapter where action occurrences encoding possible plans are enumerated, and enumerations that do not lead to the goal are eliminated resulting in a answer sets each of which encode a plan. In addition the planning module of RCS-AnsProlog encodes several domain independent and domain dependent heuristics that narrows the search. An example of a domain independent heuristics is: "Under normal conditions do not perform two different actions with the same effects." In RCS a valve V can be put into state S either by flipping the switch Sw that controls V or by issuing a command CC to the computer that can move V to state S. The following AnsProlog$^{\perp}$ rule encodes the above mentioned heuristics information.

$\leftarrow occurs(flip(Sw, S), T), controls(Sw, V), occurs(CC, T), commands(CC, V, S),$
not $holds(bad\_circuit(V), T).$

An example of a domain dependent heuristics is the notion that for a normally functioning valve connecting node $N1$ to $N2$, the valve should not be opened if $N1$ is not pressurized. This is indirectly encoded as follows:

$\leftarrow link(N1, N2, V), holds(state\_of(V, open), T),$ **not** $holds(pressurized\_by(N1, Tk), T),$
not $holds(has\_leak(V), T),$ **not** $holds(stuck(V), T).$

5.11 Notes and references

The simple action description language $\mathcal{A}$ was proposed and a sound formulation of it in AnsProlog$^{\neg}$ was presented in [GL92, GL93]. The relationship between the AnsProlog$^{\neg}$ formulation and partial order planning was studied in [Bar97b]. The language $\mathcal{A}$ was extended to $\mathcal{A}_C$ to allow compound actions and sound and complete formulations of it in AnsProlog* was given in [BG93, BG97, BGW99]. It was extended to $\mathcal{L}$ in another dimension to allow interleaving of execution and planning in [BGP97]. Encoding of static causal information and reasoning about them during action execution was was done in [Bar95, MT95, Tur97]. The representation in [Bar95] was inspired by the notion of revision programming [MT94a] and its encoding in AnsProlog [Bar94, Bar97a, PT95]. Reasoning about defeasible effects of actions was presented in [BL97]. Approximate reasoning about actions in presence of incompleteness and sensing actions was formulated in [BS97]. The papers [LW92, LMT93] presented implementations of reasoning about actions using AnsProlog. The volume [Lif97]had several papers on reasoning about actions and logic programming. Transition

systems are represented in logic programming in [LT99]. Reasoning about complex plans consisting of hierarchical and procedural constructs is studied in [BS99].

The generate and test approach to planning was first presented in [SZ95] and then taken up in [DNK97]. Recently it has been taken up with new vigor in [Lif99a, Lif99b, EL99]. In [TB01] the impact of knowledge representation aspects on answer set planning is studied and in [SBM01] answer set planning is extended to allow procedural constraints. Reasoning about actions in a dynamic domain in a generate and test setting is studied in [BG00].

The case study of the RCS-AnsProlog system is based on the reported work in [BW99, Wat99, BGN$^+$01].

Chapter 6

Complexity, expressibility and other properties of AnsProlog* programs

Earlier in Chapter 3 we discussed several results and properties of AnsProlog* programs that help in analyzing and step-by-step building of these programs. In this chapter we consider some broader properties that help answer questions such as: (a) how difficult it is to compute answer sets of various sub-classes of AnsProlog*; (b) how expressive are the various sub-classes AnsProlog*; (c) does the use of AnsProlog* lead to compact representation or can it be compiled to a more tractable representation; and (d) what is the relationship between AnsProlog* and other knowledge representation formalisms.

The answers to these questions are important in many ways. For example, if we know the complexity of a problem that we want to solve then the answer to (a) will tell us which particular subset of AnsProlog* will be most efficient, and the answer to (b) will tell us the most restricted subset that we can use to represent that problem. With respect to (c) we will discuss results that show that for AnsProlog* leads to a compact representation. This clarifies the misconception that since many AnsProlog* classes belong to a higher complexity class, they are not very useful. For specifications where AnsProlog* leads to (exponentially) compact representation the fact that they are computationally harder is cancelled out and they become preferable as compact representation means that the programmer has to write less. So the burden is shifted from the programmer to the computer, which is often desirable.

To make this chapter self complete we start with the basic notions of complexity and expressibility, and present definitions of the polynomial, arithmetic and analytical hierarchy and their normal forms. We later use them in showing the complexity and expressibility of AnsProlog* subclasses.

6.1 Complexity and Expressibility

Intuitively, the notion of complexity of an AnsProlog* sub-class characterizes how hard it is to compute an entailment with respect to programs in that sub-class and the notion of expressibility characterizes what all can be expressed in that sub-class. Since AnsProlog* programs can also be viewed as a function, there are three different complexity measures associated with AnsProlog* sub-classes: data complexity, program complexity and combined complexity. In the first two cases, the AnsProlog* program is considered to consist of two parts, a set of *facts*, and a set of *rules* referred to as the program. In case of data complexity the program part is fixed and the facts are varied and the complexity measure is with respect to the size of the facts, while in case of

program complexity the facts are fixed and the program is varied and the complexity measure is with respect to the size of the program. In case of combined complexity the complexity measure is with respect to both the program and the facts. The following example illustrates the difference.

Example 107 Consider Π consisting of a set of ground facts and a rule with a non-empty body:

$p(a_1) \leftarrow.$ $\quad \ldots \quad$ $p(a_k) \leftarrow.$

$q(X_1, \ldots, X_l) \leftarrow p(X_1), \ldots, p(X_l).$

It is easy to see that $ground(\Pi)$ consists of $k + k^l = k + 2^{lg_2(k) \times l}$ rules; the size of the ground facts in Π is $c_1 \times k$, for some positive constant c_1; and the size of the rules with non-empty body in Π is $c_2 \times l$, for some positive constant c_2.

Now if we consider the facts part to be constant then the size of $ground(\Pi)$ is of the order of $O(2^{lg_2(k) \times n})$ where n is the size of the rule with non-empty body, and k is a constant. Similarly, if we consider the rule with non-empty body part to be constant then the size of $ground(\Pi)$ is of the order of $O(m^l)$ where m is the size of the facts, and l is a constant. Thus, if we keep the facts fixed then the size of $ground(\Pi)$ is exponential in terms of the size of rest of Π and if we keep the rule with non-empty body part fixed then the size of $ground(\Pi)$ is polynomial in terms of the size of the facts in Π. □

We now formally define data complexity and program complexity of AnsProlog* sub-classes. Although in our definition below L is a considered to be a sub-class of AnsProlog*, the definition holds if we generalize L to be any language like AnsProlog* which has two parts and has an entailment relation.

Definition 69 Let L be a sub-class of AnsProlog*, Π be a program in L, D_{in} be a set of ground facts in L, and A be a literal.

The *data complexity* of L is the complexity of checking $D_{in} \cup \Pi \models A$, in terms of the length of the input $\langle D_{in}, A \rangle$, given a fixed Π in L.

The *program complexity* of L is the complexity of checking $D_{in} \cup \Pi \models A$, in terms of the length of the input $\langle \Pi, A \rangle$, given a fixed D_{in} in L.

The *combined complexity* of L is the complexity of checking $D_{in} \cup \Pi \models A$, in terms of the length of the input $\langle \Pi, D_{in}, A \rangle$. □

One of our goals in this chapter is to associate complexity classes with the various AnsProlog* sub-classes. In this we are interested in both membership in the complexity classes, and completeness with respect to these complexity classes. Thus we extend the notion of data complexity and program complexity to notions of an AnsProlog* sub-class being *data-complete* or *program-complete* with respect to a complexity class C. In this we first start with the definitions for data-completeness. In general, our focus in this chapter will be more on data complexity than on program complexity for two reasons: (a) Often the size of data is much larger than the size of the program. (ii) For the applications that we focus on throughout the book the more appropriate measure is data complexity. For example, suppose we want to measure the complexity of our AnsProlog* encodings of planning problems. There the variable part is the description of the domain which can be encoded by facts. Similarly, for the graph domains, the complexity of the graph problems correspond to the data complexity of their AnsProlog* encodings; as when the graph is changed, the facts in the program change, and the rules remain the same.

Definition 70 Given an AnsProlog* program Π the recognition problem associated with it is to determine if $\Pi \cup I \models A$, when given some facts I and a literal A. Alternatively, the recognition problem of Π is to determine membership in the set $\{\langle I, A\rangle \mid \Pi \cup I \models A\}$. □

Definition 71 An AnsProlog* program Π is said to be in complexity class C if the recognition problem associated with Π is in C. □

It should be noted that in the above definitions, our focus is on data complexity, as we are keeping the program fixed and varying the facts.

Definition 72 (Data-complete) An AnsProlog* sub-class L is *data-complete for complexity class* C (or equivalently, the data complexity class of L is C-complete) if

(i) *(membership)*: each program in L is in C, and

(ii) *(hardness)*: there exists a program in L for which the associated recognition problem is complete with respect to the class C. □

We now define the notion of program-completeness in a slightly different manner.

Definition 73 (Program-complete) An AnsProlog* sub-class L is *program-complete for complexity class* C (or equivalently, the program complexity class of L is C-complete) if

(i) *(membership)*: the program complexity of L is C, and

(ii) *(hardness)*: there is a complete problem P in class C, which can expressed by a program in L, over a fixed set of facts. □

For example, if we can use the above definition to show that AnsDatalog$^{\neg\mathbf{not}}$ is program-complete for EXPTIME, by showing that (i) the program complexity of AnsDatalog$^{\neg\mathbf{not}}$ is EXPTIME and (ii) if a deterministic Turing machine (DTM) M halts in less than $N = 2^{n^k}$ transitions on a given input I, where $|I| = n$, then we can construct a AnsDatalog$^{\neg\mathbf{not}}$ program $\pi(I)$ with a fixed set of ground facts F such that $\pi(I) \models \mathit{accept}$ iff M accepts the input I.

We now define the notion of expressibility and explain how it differs from the notion of complexity.

Definition 74 (Expressibility) An AnsProlog* sub-class L is said to *capture the complexity class* C if

(i) each program in L is also in C, and

(ii) every problem of complexity C can be expressed in L. □

Although the notions of being data-complete for complexity class C and capturing the class C are close, they are not equivalent. We elaborate on this now.

- L is data-complete in C does not imply that L captures C.

  This is because, even though there may exist queries in L for which the associated recognition problem is complete with respect to the class C, there may be problems in the class C which can not be expressed in L.

  An example of this is the query classes *fixpoint* and *while* which are data-complete for $PTIME$ and $PSPACE$ respectively, but yet neither can express the simple query:

  even(R) = true if $|R|$ is even, and false otherwise.

- L captures C does not imply that L is data-complete in C.

  This is because even though L captures C there may not exist a problem that is C-complete. For example, as mentioned in [DEGV99] second-order logic over finite structures captures the polynomial hierarchy PH, for which no complete problem is known, and the existence of a complete problem of PH would imply that it collapses at some finite level, which is widely believed to be false.

To further explain the difference let us discuss the familiar definition of NP-completeness. The *definition* of NP-completeness is that a problem X is NP-complete if (i) it belongs to NP, and (ii) all problems in NP can be reduced feasibly (polynomially) to an instance of X. This definition is close to the definition of *expressibility* where we use the phrase *can be expressed* instead of the phrase *can be reduced feasibly.* Now let us consider how we normally prove that a problem X is NP-complete. We show it belongs to NP and then take a known NP-complete problem Y and give a polynomial time reduction of Y to X. This approach is close to the definition of data completeness above. Why is then the two notions of *expressibility* and *complexity* are different? One of the reasons is that data complexity of L being NP-complete does not imply L captures NP. This is because when we are dealing with query languages, there is no guarantee that the query language of our focus can express the polynomial reducibility that is used in the definition and proof strategy of NP-completeness.

We now define several important complexity classes: the polynomial hierarchy, exponential classes, arithmetic hierarchy, and analytical hierarchy.

6.1.1 The Polynomial Hierarchy

A decision problem is a problem of deciding whether a given input w satisfies a certain property Q. I.e., in set-theoretic terms, whether it belongs to the corresponding set $S = \{w \mid Q(w)\})$.

Definition 75 The basic complexity classes

- A decision problem is said to belong to the class **P** if there is a polynomial-time algorithm in a deterministic machine for solving this problem.
- A decision problem is said to belong to the class **NP** if there is a polynomial-time algorithm in a non-deterministic machine for solving this problem.
- A decision problem is said to belong to the class **coNP** if the complement of the problem is in **NP**.
- A decision problem is said to belong to the class **PSPACE** if there is a polynomial-space algorithm in a deterministic machine for solving this problem.
- For any deterministic or non-deterministic complexity class C, the class C^A is defined to be the class of all languages decided by machines of the same sort and time bound as in C, except that the machine now has an oracle A.
- The Polynomial hierarchy is defined as follows:
  - $\Sigma_0\mathbf{P} = \Pi_0\mathbf{P} = \mathbf{P}$
  - $\Sigma_{i+1}\mathbf{P} = \mathbf{NP}^{\Sigma_i\mathbf{P}}$

– $\Pi_{i+1}\mathbf{P} = \mathbf{coNP}^{\Sigma_i \mathbf{P}}$ □

In this book we will use the following alternative characterization of the polynomial hierarchy in our proofs. In these characterizations we will use the notions: *polynomially decidable* and *polynomially balanced*; which we define now. A k+1-ary relation P on strings is called polynomially decidable if there is a DTM deciding the language $\{u_1, \ldots, u_k, w \mid (u_1, \ldots, u_k, , w) \in P\}$. We say P is polynomially balanced if $(u_1, \ldots, u_k, w) \in P$ implies that the size of u_i's is bounded by a polynomial in the size of w.

Proposition 87 Alternative characterization of the polynomial hierarchy

- A problem belongs to the class **NP** iff the formula $w \in S$ (equivalently, $Q(w)$) can be represented as $\exists u P(u, w)$, where $P(u, w)$ is polynomially decidable and polynomially balanced. The class **NP** is also denoted by $\Sigma_1\mathbf{P}$ to indicate that formulas from this class can be defined by adding 1 existential quantifier (hence Σ and 1) to a polynomial (hence **P**) predicate .
- A problem belongs to the class **coNP** iff the formula $w \in S$ (equivalently, $Q(w)$) can be represented as $\forall u P(u, w)$, where $P(u, w)$ is polynomially decidable and polynomially balanced. The class **coNP** is also denoted by $\Pi_1\mathbf{P}$ to indicate that formulas from this class can be defined by adding 1 universal quantifier (hence Π and 1) to a polynomial predicate.
- For every positive integer k, a problem belongs to the class $\Sigma_k\mathbf{P}$ iff the formula $w \in S$ (equivalently, $Q(w)$) can be represented as $\exists u_1 \forall u_2 \ldots P(u_1, u_2, \ldots, u_k, w)$, where $P(u_1, \ldots, u_k, w)$ is polynomially decidable and polynomially balanced.
- Similarly, for every positive integer k, a problem belongs to the class $\Pi_k\mathbf{P}$ iff the formula $w \in S$ (equivalently, $Q(w)$) can be represented as $\forall u_1 \exists u_2 \ldots P(u_1, u_2, \ldots, u_k, w)$, where $P(u_1, \ldots, u_k, w)$ is polynomially decidable and polynomially balanced.
- A problem belongs to the class **PSPACE** iff the formula $w \in S$ (equivalently, $Q(w)$) can be represented as $\forall u_1 \exists u_2 \ldots P(u_1, u_2, \ldots, u_k, w)$, where the number of quantifiers k is bounded by a polynomial of the length of the input, and $P(u_1, \ldots, u_k, w)$ is polynomially decidable and polynomially balanced. □

In this chapter we often use the above alternative characterizations of the polynomial hierarchy to show the membership of particular problems in classes in the polynomial hierarchy. The following example illustrates this approach.

Example 108 It is well known that the problem of satisfiability of a propositional formula is in **NP**, the problem of unsatisfiability of a propositional formula is in **coNP**. We will now use the above proposition in showing these.

(i) Let w be a propositional formula, and $Q(w)$ be true iff w is satisfiable. To show that the problem of satisfiability of a propositional formula is in **NP** we need to come up with a polynomially decidable and polynomially balanced property $P(u, w)$ such that w is satisfiable iff $\exists u P(u, w)$ is true.

Let u be assignment of true or false to the propositions, and $P(u, w)$ be the evaluation of the formula w using the assignments in u. Obviously, $P(u, w)$ is polynomially decidable and polynomially balanced. Moreover, w is satisfiable iff $\exists u P(u, w)$ is true. Hence, the problem of satisfiability of a propositional formula is in **NP**.

(ii) Let w be a propositional formula, and $Q'(w)$ be true iff w is unsatisfiable. To show that the problem of unsatisfiability of a propositional formula is in **coNP** we need to come up with a polynomially decidable and polynomially balanced property $P'(u, w)$ such that w is unsatisfiable iff $\forall u P'(u, w)$ is true.

Let u be assignment of true or false to the propositions, and $P'(u, w)$ be the complement of the evaluation of the formula w using the assignments in u. Obviously, $P'(u, w)$ is polynomially decidable and polynomially balanced. Moreover, w is unsatisfiable iff $\forall u P'(u, w)$ is true. Hence, the problem of unsatisfiability of a propositional formula is in **coNP**. □

Definition 76 A problem is called *complete* in a class C of the polynomial hierarchy if, any other problem from this class can be reduced to it by a polynomial-time[1] reduction. □

Proposition 87 also leads us to several basic problems that are complete with respect to the various classes in the polynomial hierarchy. We enumerate them in the following proposition and use them later in the chapter when showing that particular AnsProlog* sub-classes are data-complete with respect to complexity classes in the polynomial hierarchy.

Proposition 88 Given a boolean expression ϕ, with boolean variables partitioned into sets $X_1, \ldots X_i$, let E_QSAT_i denote the satisfiability of the formula $\exists X_1, \forall X_2, \ldots Q X_i \phi$, where Q is $\exists$ if i is odd and $\forall$ otherwise.; and let F_QSAT_i denote the satisfiability of the formula $\forall X_1, \exists X_2, \ldots Q X_i \phi$, where Q is $\exists$ if i is even and $\forall$ otherwise.

(i) For all $i \geq 1$ E_QSAT_i is $\Sigma_i \mathbf{P}$ complete.

(ii) For all $i \geq 1$ F_QSAT_i is $\Pi_i \mathbf{P}$ complete. □

It should be noted that it is still not known whether we can solve any problem from the class **NP** in polynomial time (i.e., in precise terms, whether **NP**=**P**). However, it is widely believed that we cannot, i.e., **NP**≠**P**. It is also believed that to solve a **NP**-complete or a **coNP**-complete problem, we need $O(2^n)$ time, and that solving a complete problem from one of the second-level classes $\Sigma_2 \mathbf{P}$ or $\Pi_2 \mathbf{P}$ requires more computation time than solving **NP**-complete problems and solving complete problems from the class **PSPACE** takes even longer.

6.1.2 Polynomial and exponential Classes

We now define some complexity classes which are beyond the polynomial hierarchy, but are recursive. To define that let us first define the following:

$TIME(f(n)) = \{L \mid L$ is decided by some DTM in time $O(f(n))\}$,
$NTIME(f(n)) = \{L \mid L$ is decided by some NDTM in time $O(f(n))\}$,
$SPACE(f(n)) = \{L \mid L$ is decided by some DTM within space $O(f(n))\}$,
$NSPACE(f(n)) = \{L \mid L$ is decided by some NDTM within space $O(f(n))\}$.

Using the above notations we can now formally define the classes **P**, **NP**, EXPTIME, NEXPTIME, and PSPACE as follows:

$\mathbf{P} = \bigcup_{d>0} TIME(n^d)$
$\mathbf{NP} = \bigcup_{d>0} NTIME(n^d)$
EXPTIME $= \bigcup_{d>0} TIME(2^{n^d})$

[1] Alternatively, many authors use O(log n) space reduction.

NEXPTIME $= \bigcup_{d>0} NTIME(2^{n^d})$
PSPACE $= \bigcup_{d>0} SPACE(n^d)$

The notion of EXPTIME-complete and NEXPTIME-complete is based on Definition 76.

6.1.3 Arithmetical and Analytical hierarchy

The arithmetical and analytical hierarchy are similar to the polynomial hierarchy. The starting point in the arithmetical hierarchy is the class of *recursive* problems – defined in Section 11.1 – instead of the class **P** in the polynomial hierarchy. The arithmetical hierarchy is denoted by Σ_i^0 and Π_i^0, where i is non-negative integer, while the analytical hierarchy is denoted by Σ_i^1 and Π_i^1. We first define the arithmetical hierarchy.

1. A k-ary relation R is said to be recursive if the language $L_R = \{x_0; \ldots; x_{k-1} : (x_0, \ldots, x_{k-1}) \in R\}$ is recursive.

2. We say a decision problem belongs to the class Σ_0^0 iff it is *recursive*.

3. We say a decision problem belongs to the class Σ_1^0 iff it is *recursively enumerable*.

4. Σ_{n+1}^0 denotes the class of all languages L for which there is a (n+2)-ary *recursive* relation R such that $L = \{y : \exists x_0 \forall x_1 \ldots Q_n x_n R(x_0, \ldots, x_n, y\}$ where Q_k is $\forall$ if k is odd, and $\exists$, if k is even. Alternatively, Σ_{n+1}^0 denotes the relations over the natural numbers that are definable in arithmetic by means of a *first-order formula* $\Phi(\mathbf{Y}) = \exists \mathbf{X}_0 \forall \mathbf{X}_1 \ldots Q_k \mathbf{X}_n \psi(\mathbf{X}_0, \ldots, \mathbf{X}_n, \mathbf{Y})$ with free variables $\mathbf{Y}$, Q_k is as before, and ψ is quantifier free.

5. $\Pi_{n+1}^0 = \mathbf{co}\Sigma_{n+1}^0$, that is, Π_{n+1}^0 is the set of all complements of languages in Σ_{n+1}^0. It can be easily shown that Π_{n+1}^0 denotes the class of all languages L for which there is a (n+2)-ary *recursive* relation R such that $L = \{y : \forall x_0 \exists x_1 \ldots Q_n x_n R(x_0, \ldots, x_n, y\}$ where Q_k is $\exists$ if k is odd, and $\forall$, if k is even. Alternatively, Π_{n+1}^0 denotes the relations over the natural numbers that are definable in arithmetic by means of a *first-order formula* $\Phi(\mathbf{Y}) = \forall \mathbf{X}_0 \exists \mathbf{X}_1 \ldots Q_k \mathbf{X}_n \psi(\mathbf{X}_0, \ldots, \mathbf{X}_n, \mathbf{Y})$ with free variables $\mathbf{Y}$, and ψ is quantifier free.

Note that the difference between the classes $\Sigma_{n+1}\mathbf{P}$ and $\Pi_{n+1}\mathbf{P}$ from the polynomial hierarchy and the classes Σ_{n+1}^0 and Π_{n+1}^0 from the arithmetical hierarchy is that the relation ψ has to be decidable (i.e., we can determine if a tuple belongs to that relation or not) in polynomial time in the first case, while it must be recursive in the second case. We now define the analytical hierarchy.

1. The class Σ_1^1 belongs to the *analytical hierarchy* (in a relational form) and contains those relations which are definable by a *second order formula* $\Phi(\mathbf{X}) = \exists \mathbf{P} \phi(\mathbf{P}; \mathbf{X})$, where $\mathbf{P}$ is a tuple of predicate variables and ϕ is a first order formula with free variables $\mathbf{X}$.

2. The class Π_1^1 belongs to the *analytical hierarchy* (in a relational form) and contains those relations which are definable by a *second order formula* $\Phi(\mathbf{X}) = \forall \mathbf{P} \phi(\mathbf{P}; \mathbf{X})$, where $\mathbf{P}$ is a tuple of predicate variables and ϕ is a first order formula with free variables $\mathbf{X}$.

3. Σ_{n+1}^1 denotes the set of relations which are definable by a *second-order formula* $\Phi(\mathbf{Y}) = \exists \mathbf{P}_0 \forall \mathbf{P}_1 \ldots Q_k \mathbf{P}_n \psi(\mathbf{P}_0, \ldots, \mathbf{P}_n, \mathbf{Y})$, where $\mathbf{P}_i s$ are tuples of predicate variables and ψ is a first order formula with free variables $\mathbf{Y}$.

4. Π^1_{n+1} denotes the set of relations which are definable by a *second-order formula* $\Phi(\mathbf{Y}) = \forall \mathbf{P}_0 \exists \mathbf{P}_1 \ldots Q_k \mathbf{P}_n \psi(\mathbf{P}_0, \ldots, \mathbf{P}_n, \mathbf{Y})$, where $\mathbf{P}_i s$ are tuples of predicate variables and ψ is a first order formula with free variables $\mathbf{Y}$.

Note that the difference between the classes Σ^0_{n+1} and Π^0_{n+1} from the arithmetical hierarchy and the classes Σ^1_{n+1} and Π^1_{n+1} from the analytical hierarchy is that Φ is a first-order formula in the first case, while it is a second-order formula in the second case; and the quantification is over variables in the first case and is over predicates in the second case.

Definition 77 A problem L_1 is called *complete* in a class C of the arithmetical or analytical hierarchy if, any other problem L_2 from this class can be reduced to it (L_1) by Turing reduction.

By Turing reduction of L_2 to L_1 we mean that L_2 can be decided by a deterministic Turing machine with oracle L_1. □

6.1.4 Technique for proving expressibility: general forms

To show that an AnsProlog* sub-class L captures the complexity class C, we need to show (i) that every program in L is in C, and (ii) every query of complexity C can be expressed in L. To show the first condition we consider arbitrary programs from L and show that entailment with respect to them can be computed as per C. To show the second condition, we will often use general (or normal) forms of complexity classes. A general form of a complexity class C, is a form in which all problems in class C can be expressed. All complexity classes may not have a general form. But when a complexity class has one, to show (ii) we only need to consider arbitrary expressions in a general form of C, and show how to express it in L. In the following proposition we list a few general forms that we will use in our proofs.

Proposition 89 General form of some complexity classes
Consider a signature $\sigma = (O, F, P)$, where O is finite, and $F = \emptyset$ meaning that there are no function symbols. By a finite database over $\sigma = (O, F, P)$ we mean a finite subset of the Herbrand Base over σ.

1. [Fag74] A collection S of finite databases over the signature $\sigma = (O, \emptyset, P)$ is in $\Sigma_k \mathbf{P}$ iff there exists a $\Sigma^1_k(\sigma)$ sentence Ψ such that for any finite database w over σ, $w \in S$ iff w satisfies Ψ.

2. [Fag74, KP88] A collection S of finite databases over the signature σ is in **NP** iff it is definable by an existential second order formula over σ, i.e., iff there is a formula of the form $\exists U_1, \ldots U_m \forall \bar{x} \exists \bar{y} (\theta_1(\bar{x}, \bar{y}) \vee \ldots \vee \theta_k(\bar{x}, \bar{y}))$, where $\theta_i(\bar{x}, \bar{y})$'s are conjunctions of literals involving the predicates in σ and $\{U_1, \ldots, U_m\}$ such that for any finite database w over σ
$w \in S$ iff w satisfies $\exists U_1, \ldots U_m \forall \bar{x} \exists \bar{y} (\theta_1(\bar{x}, \bar{y}) \vee \ldots \vee \theta_k(\bar{x}, \bar{y}))$.

3. [EGM94] Assume that *succ*, $first$ and *last* are predicates that do not occur in P. By an enumeration literal (or literal in an enumerated theory) we mean literals in $(O, \emptyset, \{succ, first, last\})$ with the following conditions:

   - *succ* describes an enumeration of all elements in O, where $(x, y) \in succ$ means that y is the successor of x,
   - and the unary relation $first$ and *last* contain the first and last element in the enumeration respectively.

A collection S of finite databases over the signature $\sigma = (O, \emptyset, P)$ is in $\Sigma_2\mathbf{P}$ iff there is a formula of the form $\exists U_1, \ldots U_m \forall V_1, \ldots, V_n \exists \bar{x}(\theta_1(\bar{x}) \vee \ldots \vee \theta_k(\bar{x}))$, where $\theta_i(\bar{x})$ are conjunction of enumeration literals or literals involving predicates in $P \cup \{U_1, \ldots, U_m, V_1, \ldots, V_n\}$ such that for any finite database w over σ, $w \in S$ iff w satisfies $\exists U_1, \ldots U_m \forall V_1, \ldots, V_n \exists \bar{x}(\theta_1(\bar{x}) \vee \ldots \vee \theta_k(\bar{x}))$. □

Once we have proved that a certain language L captures a complexity class C by showing that (i) every program in a language L is in class C and (ii) the general form of a complexity class C is expressible in a language L, to show that L is C-complete only additional result we need to show is that there exists a C-complete problem. This is because, this C-complete problem, by definition, belongs to class C. Hence, it can be expressed in the general form, which we would have already shown to be expressible in L. Hence, we have a C-complete problem expressible in L. This together with (i) proves that L is C-complete.

6.2 Complexity of AnsDatalog* sub-classes

In this section we consider several sub-classes of AnsDatalog* – AnsProlog* programs with no function symbols – and explore their data and program complexity classes. We start with propositional AnsDatalog programs.

6.2.1 Complexity of propositional AnsDatalog$^{\neg\mathbf{not}}$

In case of propositional programs our main interest lies in the combined complexity. We now state and prove the combined complexity of AnsDatalog$^{\neg\mathbf{not}}$ programs.

Theorem 6.2.1 Given a propositional AnsDatalog$^{\neg\mathbf{not}}$ program Π and a literal A the complexity of deciding $\Pi \models A$ is P-complete. □

Proof:

(membership): The answer set of a propositional AnsDatalog$^{\neg\mathbf{not}}$ program can be obtained by the iterated fixpoint approach. Each iteration step can be done in polynomial time. The total number of iterations is bound by the number of rules plus one. Hence, $\Pi \models A$ can be determined in polynomial time.

(hardness): Let L be a language in $\mathbf{P}$. Thus L is decidable in $p(n)$ steps by a DTM M for some polynomial p. We now present a feasible transformation of each instance I of L into a propositional AnsDatalog$^{\neg\mathbf{not}}$ program $\pi(M, I, p(|I|))$ such that $\pi(M, I, p(|I|)) \models \mathit{accept}$ iff M with input I reaches an accepting step within $N = p(|I|)$ steps.

1. We first define the various propositions that we will use and their intuitive meaning.
   (a) $\mathit{state}_s[t]$ for $0 \leq t \leq N$: at time point t the state of the Turing machine is s.
   (b) $\mathit{cursor}[n, t]$ for $0 \leq t \leq N$ and $0 \leq n \leq N$: at time point t the cursor is at the cell number n (counted from the left).
   (c) $\mathit{symbol}_\alpha[n, t]$ for $0 \leq t \leq N$ and $0 \leq n \leq N$: at time point t, α is the symbol in cell number n (counted from the left) of the tape.
   (d) *accept*: the DTM M has reached state *yes*.

2. We now present the part of $\pi(M, I, p(|I|))$ that describes the initial state, the initial cursor position, the input, and the transition dictated by the transition function δ of M.

   (a) $state_{s_0}[0] \leftarrow$. : at time point 0 the state of the Turing machine is s_0.

   (b) $cursor[0, 0] \leftarrow$. : at time point 0 the cursor is at the left most end of the tape.

   (c) If the input is $I = \alpha_1 \ldots \alpha_{|I|}$, then we have the following:

   $symbol_{\alpha_1}[1, 0] \leftarrow$. $\quad \ldots \quad$ $symbol_{\alpha_{|I|}}[|I|, 0] \leftarrow$.

   (d) $symbol_{>}[0, 0] \leftarrow$. and $symbol_{\sqcup}[|I|+1, 0] \leftarrow$.: beginning of the tape, and end of the input.

   (e) For each $\delta(s, \alpha) = (s', \beta, p)$, we have the following rules:

   $symbol_{\beta}[n, t+1] \leftarrow symbol_{\alpha}[n, t], state_s[t], cursor_{[}n, t]$.

   $state_{s'}[t+1] \leftarrow symbol_{\alpha}[n, t], state_s[t], cursor[n, t]$.

   $cursor[X, t+1] \leftarrow symbol_{\alpha}[n, t], state_s[t], cursor[n, t]$.

   where, X is $n+1$ if p is $\rightarrow$, X is $n-1$ if p is $\leftarrow$, and X is n if p is $-$.

3. We also need inertia rules for the propositions $symbol_{\beta}[n, t]$ to describe which cell numbers keep their previous symbols after a transition. For $0 \leq t < N$, $n \neq n'$, and $n, n' \leq N$, we have the rules:

   $symbol_{\alpha}[n, t+1] \leftarrow symbol_{\alpha}[n, t], cursor_{[}n', t]$.

4. Finally, we need a rule to derive *accept*, when an accepting state is reached.

   $accept \leftarrow state_{yes}[t]$, for $0 \leq t \leq N$.

The above program has $O(N^2)$ rules which is polynomial in the size of $|I|$. Hence, it can be constructed in polynomial time. Moreover, if we compute the answer set of the above program using the iterated fixpoint approach the computation mimics the transitions in the DTM, and the answer set contains *accept* iff an accepting configuration is reached by the DTM M with input I in at most N computation steps. □

Exercise 18 Find the exact correspondence between the iterated fixpoint computation and the DTM configuration in the above proof. □

The proof of the above theorem can also be directly used to show that entailment with respect to propositional AnsDatalog$^{-\mathbf{not}}(3)$ programs is **P**-Complete. This is because all the rules in $\pi(M, I, p(|I|))$ have at most 3 literals in their body.

6.2.2 Complexity of AnsDatalog$^{-\mathbf{not}}$

We start with the data complexity of AnsDatalog$^{-\mathbf{not}}$ programs.

Theorem 6.2.2 AnsDatalog$^{-\mathbf{not}}$ programs are data-complete for **P**. □

Proof:

(membership): Following the computations in Example 107 it is clear that the Herbrand Base of a program is polynomial in size of the input ground facts. Now, given a set of ground facts I and a literal A, we can determine if $\Pi \cup I \models A$ – where Π is an AnsDatalog$^{\neg\mathbf{not}}$ program by – computing the answer set of $\Pi \cup I$. We can obtain this answer set by the iterated fixpoint approach, and the maximum number of iterations that we may need is bounded by the size of the Herbrand Base, as in each iteration we must add at least one new atom for the iteration to continue. Each iteration takes polynomial amount of time. Thus determining if $\Pi \cup I \models A$ can be done in time polynomial in the size of I.

(hardness): To show the hardness we use the result that entailment with respect to propositional AnsDatalog$^{\neg\mathbf{not}}(3)$ programs is **P**-Complete. Let us consider an arbitrary propositional AnsDatalog$^{\neg\mathbf{not}}(3)$ program Π. We will represent this program as facts $D_{in}(\Pi)$ as follows:

For each rule $A_0 \leftarrow A_1, \ldots, A_i$, $0 \leq i \leq 3$, we have the fact $R_i(A_0, A_1, \ldots, A_i) \leftarrow$.

Let us now consider the following AnsDatalog$^{\neg\mathbf{not}}$ program Π_{meta}:

$T(X_0) \leftarrow R_0(X_0)$.
$T(X_0) \leftarrow T(X_1), R_1(X_0, X_1)$.
$T(X_0) \leftarrow T(X_1), T(X_2), R_2(X_0, X_1, X_2)$.
$T(X_0) \leftarrow T(X_1), T(X_2), T(X_3), R_3(X_0, X_1, X_2, X_3)$.

It is easy to see that $\Pi \models A$ iff $D_{in}(\Pi) \cup \Pi_{meta} \models A$. Since the size of Π is of the same order as the size of $D_{in}(\Pi)$ and we keep Π_{meta} fixed, the data complexity of AnsDatalog$^{\neg\mathbf{not}}$ – to which Π_{meta} belongs – is same as the complexity of entailment in propositional AnsDatalog$^{\neg\mathbf{not}}(3)$. Hence, AnsDatalog$^{\neg\mathbf{not}}$ is data-complete in **P**. □

Exercise 19 It is well known that AnsDatalog$^{\neg\mathbf{not}}$ can not express the query which given a set of ground facts, determines if the number of facts are even or not. This obviously can be done in polynomial time. Explain why this does not contradict with our theorem above. □

We now state and prove the program complexity of AnsDatalog$^{\neg\mathbf{not}}$ programs.

Theorem 6.2.3 AnsDatalog$^{\neg\mathbf{not}}$ programs are program-complete for EXPTIME. □

Proof:

(membership): Recall from Example 107 that given a program Π, and a set of ground facts D, the size of $ground(\Pi)$ is exponential with respect to the size of Π. For an AnsDatalog$^{\neg\mathbf{not}}$ program Π, $ground(\Pi)$ is a propositional program and from Theorem 6.2.1 entailment with respect to $ground(\Pi) \cup D$ will be polynomial in the size of $ground(\Pi) \cup D$. Since we fix D, and the size of $ground(\Pi)$ is exponential with respect to the size of Π, program complexity of AnsDatalog$^{\neg\mathbf{not}}$ programs is in EXPTIME.

(hardness): Let L be a language in **EXPTIME**. Thus L is decidable in 2^{n^k} steps by a DTM M for some positive integer k. We now present a feasible transformation of each instance I of L into an AnsDatalog$^{\neg\mathbf{not}}$ program $\pi(M, I)$ with a fixed input database D, such that $\pi(M, I) \models accept$ iff M with input I reaches an accepting step within $N = 2^{|I|^k}$ steps. The fixed input database that we use is the empty database with the universe $U = \{0, 1\}$.

The AnsDatalog$^{\neg\mathbf{not}}$ program $\pi(M, I)$ is similar to the program in the hardness part of the proof of Theorem 6.2.1. But instead of the propositional symbols $state_s[t]$, $cursor[n, t]$, and $symbol_\alpha[n, t]$ used there, we now have predicates $state(s, t)$, $cursor(n, t)$, and $symbol(\alpha, n, t)$. The main difficulty now is to be able to define a predicate $succ(t, t')$ for $t' = t + 1$ without using function symbols. To achieve this we represent n and t as tuples of arity $m = |I|^k$, where each element of the tuple can take the value 0 or 1. Thus we can represent numbers from 0 to $2^m - 1$. Now we need to define a successor relationship between such binary representation of numbers. We achieve this by introducing predicates $succ^i$, for $i = 1 \ldots m$, which defines the successor relationship between i-bit numbers. We also have predicates $first^i$ and $last^i$ to define the smallest and largest i bit number. We now recursively define $succ^i$, $first^i$ and $last^i$ and defined a predicate $less^m$ between m-bit numbers. (In the following variables written in bold face represent m distinct variables, and constants written in bold face represent that number written in binary as m 0-1 constants.)

1. The base case:

   $succ^1(0, 1) \leftarrow.$
   $first^1(0) \leftarrow.$
   $last^1(1) \leftarrow.$

2. Defining $first^i$ and $last^i$:

   $first^{i+1}(0, \mathbf{X}) \leftarrow first^i(\mathbf{X}).$
   $last^{i+1}(1, \mathbf{X}) \leftarrow last^i(\mathbf{X}).$

3. Defining $succ^i$:

   $succ^{i+1}(Z, \mathbf{X}, Z, \mathbf{Y}) \leftarrow succ^i(\mathbf{X}, \mathbf{Y}).$
   $succ^{i+1}(Z, \mathbf{X}, Z', \mathbf{Y}) \leftarrow succ^1(Z, Z'), last^i(\mathbf{X}), first^i(\mathbf{Y}).$

4. Defining $less^m$:

   $less^m(\mathbf{X}, \mathbf{Y}) \leftarrow succ^m(\mathbf{X}, \mathbf{Y}).$
   $less^m(\mathbf{X}, \mathbf{Y}) \leftarrow less^m(\mathbf{X}, \mathbf{Z}), succ^m(\mathbf{Z}, \mathbf{Y}).$

We now present the part of $\pi(M, I)$ that describes the initial state, the initial cursor position, the input, and the transition dictated by the transition function δ of M.

1. $state(s_0, \mathbf{0}) \leftarrow.$: at time point $\mathbf{0}$ the state of the Turing machine is s_0.

2. $cursor(\mathbf{0}, \mathbf{0}) \leftarrow.$: at time point $\mathbf{0}$ the cursor is at the left most end of the tape.

3. If the input is $I = \alpha_1 \ldots \alpha_r$, then we have the following:

   $symbol(\alpha_1, \mathbf{1}, \mathbf{0}) \leftarrow.$ $\quad \ldots \quad$ $symbol(\alpha_r, \mathbf{r}, \mathbf{0}) \leftarrow.$

4. $symbol(>, 0, 0) \leftarrow.$ and $symbol(\sqcup, \mathbf{r} + \mathbf{1}, \mathbf{0}) \leftarrow.$: beginning of the tape, and end of the input.

5. The transition function δ is specified as atoms of the form:

   $trans(s, \alpha, s', \beta, p) \leftarrow.$

6. We have the following rules that define the transition.

   $symbol(B, \mathbf{N}, \mathbf{T}') \leftarrow symbol(A, \mathbf{N}, \mathbf{T}), state(S, \mathbf{T}), cursor(\mathbf{N}, \mathbf{T}), succ^m(\mathbf{T}, \mathbf{T}'), trans(S, A, \_, B, \_).$

   $state_{S'}(\mathbf{T}') \leftarrow symbol(A, \mathbf{N}, \mathbf{T}), state(S, \mathbf{T}), cursor(\mathbf{N}, \mathbf{T}), succ^m(\mathbf{T}, \mathbf{T}'), trans(S, A, S', \_, \_).$

   $cursor(\mathbf{N}', \mathbf{T}') \leftarrow symbol(A, \mathbf{N}, \mathbf{T}), state(S, \mathbf{T}), cursor(\mathbf{N}, \mathbf{T}), succ^m(\mathbf{T}, \mathbf{T}'), succ^m(\mathbf{N}, \mathbf{N}'),$
   $trans(S, A, \_, \_, \rightarrow).$

   $cursor(\mathbf{N}', \mathbf{T}') \leftarrow symbol(A, \mathbf{N}, \mathbf{T}), state(S, \mathbf{T}), cursor(\mathbf{N}, \mathbf{T}), succ^m(\mathbf{T}, \mathbf{T}'), succ^m(\mathbf{N}', \mathbf{N}),$
   $trans(S, A, \_, \_, \leftarrow).$

   $cursor(\mathbf{N}, \mathbf{T}') \leftarrow symbol(A, \mathbf{N}, \mathbf{T}), state(S, \mathbf{T}), cursor(\mathbf{N}, \mathbf{T}), succ^m(\mathbf{T}, \mathbf{T}'), trans(S, A, \_, \_, -).$

We also need the following inertia rules to describe which cell numbers keep their previous symbols after a transition.

$symbol(A, \mathbf{N}, \mathbf{T}') \leftarrow symbol(A, \mathbf{N}, \mathbf{T}), cursor(\mathbf{N}', \mathbf{T}), less^m(\mathbf{N}, \mathbf{N}').$

$symbol(A, \mathbf{N}, \mathbf{T}') \leftarrow symbol(A, \mathbf{N}, \mathbf{T}), cursor(\mathbf{N}', \mathbf{T}), less^m(\mathbf{N}', \mathbf{N}).$

Finally, we need a rule to derive *accept*, when an accepting state is reached.

$accept \leftarrow state(yes, \mathbf{T}).$

The above program can be constructed in constant (hence, polynomial) time. Moreover, if we compute the answer set of the above program using the iterated fixpoint approach the computation mimics the transitions in the DTM, and the answer set contains *accept* iff an accepting configuration is reached by the DTM M with input I in at most N computation steps. □

6.2.3 Complexity of AnsDatalog

Theorem 6.2.4 For any AnsDatalog program Π, and an input set of facts D determining if $\Pi \cup D$ has an answer set is **NP**-complete. I.e., determining if the set of answer sets of $\Pi \cup D$, denoted by $SM(\Pi \cup D)$ is $\neq \emptyset$ is **NP**-complete. □

Proof:

(membership) : Determining if $SM(\Pi \cup D) \neq \emptyset$ can be expressed as $\exists M.P_1(M, D)$, where $P_1(M, D)$ is true if the least model of $ground(\Pi \cup D)^M$ is equal to M. Since the size of the Herbrand base of $\Pi \cup D$ is polynomial in the size of D, the size of M - a subset of the Herbrand Base, is polynomial the in size of D. Moreover, the size of $ground(\Pi \cup D)$ is also polynomial in the size of D. Hence, computing $ground(\Pi \cup D)^M$ is polynomial in the size of D and obtaining the least model through the iterated fixpoint approach of the AnsDatalog$^{\neg}$**not** program $ground(\Pi \cup D)^M$ is also polynomial time. Hence, P_1 is polynomially decidable, and polynomially balanced. Therefore, determining if $SM(\Pi \cup D) \neq \emptyset$ is in NP.

An informal way of showing the above is to say that after guessing an M, we can verify if M is an answer set or not in polynomial time.

(hardness): We show this by considering the well-known NP-complete problem 3-Sat. Let $\{a_1, \ldots, a_n\}$ be a set of propositions. Given any instance ϕ of 3-Sat consisting of propositions from $\{a_1, \ldots, a_n\}$ we construct an AnsDatalog program $\Pi_1 \cup D$, – D dependent on ϕ, and Π fixed, such that ϕ is satisfiable iff $\Pi_1 \cup D$ has an answer set.

Let ϕ be of the form $(l_{11} \vee l_{12} \vee l_{13}) \wedge \ldots \wedge (l_{m1} \vee l_{m2} \vee l_{m3})$, where l_{ij}'s are literals made of the propositions $a_1, \ldots, a_n$.

D consists of the facts:

$conj(l_{11}, l_{12}, l_{13}) \leftarrow$. $\quad \ldots \quad$ $conj(l_{m1}, l_{12}, l_{m3}) \leftarrow$.

The fixed program Π_1 consists of the following:

1. It has rules of the following form – where $h(a)$ means a holds, or a is $true$, and $n\_h(a)$ means a does not hold, or a is $false$ – to enumerate truth of propositions in the language:

   $h(a_i) \leftarrow$ **not** $n\_h(a_i)$.

   $n\_h(a_i) \leftarrow$ **not** $h(a_i)$.

   for $i = 1 \ldots n$.

2. It has the following rules which makes q true if one of the conjunct is false:

   $q \leftarrow conj(X, Y, Z), n\_h(X), n\_h(Y), n\_h(Z)$.

3. Finally we have the following rule which rules out any answer set where q may be true, thus making sure that if an assignment to the propositions does not make ϕ true then no answer set exist corresponding to that assignment.

   $p \leftarrow$ **not** p, q

Now ϕ is satisfiable *implies* there exists an assignment A of truth values to the a_i's that makes ϕ true *implies* $\{h(a_i) : A$ assigns a_i $true\} \cup \{n\_h(a_j) : A$ assigns a_j $false\}$ is an answer set of $\Pi_1 \cup D$.

ϕ is not satisfiable *implies* there does not exists an assignment A of truth values to the a_i's that makes ϕ true *implies* there are no answer sets of Π_1. This is because if there was an answer set S of Π, in that answer set q would be true, which will then – because of the $p \leftarrow$ **not** p construction – not be an answer set. □

Theorem 6.2.5 AnsDatalog is data-complete for **coNP**. □

Proof:

(membership) $\Pi \cup D \models A$ can be written as $\forall M.P_2(M, D, A)$, where $P_2(M, D, A)$ is true if the least model of $(\Pi \cup D)^M$ is equal to M implies $M \models A$. Computing $ground(\Pi \cup D)^M$ is polynomial time (in the size of D) and obtaining the least model through the iterated fixpoint approach of the AnsDatalog$^{\neg\textbf{not}}$ program $ground(\Pi \cup D)^M$ is also polynomial time (in the size of D). Determining if $M \models A$ is polynomial time (in the size of D and A). Hence, membership in P_2 is polynomially decidable. Since M is polynomial in the size of D, P_2 is also polynomially balanced. Thus determining if $\Pi \cup D \models A$ is in $\Pi_1 P$, which is same as co-NP.

An equivalent way of showing the above is to show that the complement of the above problem is in NP. We can do that informally by guessing an M, and showing that M is an answer set of $\Pi \cup D$ and $M \not\models A$. The later two can be done in polynomial time. Thus the complement of the above problem is in NP. Hence, determining if $\Pi \cup D \models A$ is in co-NP.

(hardness): Showing Complexity of $\Pi \cup D \models A$ is co-NP Complete, for AnsDatalog programs Π.

We show this by considering the well-known co-NP Complete problem Unsatisfiability. Let $\{a_1, \ldots, a_n\}$ be a set of propositions. Given any instance ϕ of 3-Sat consisting of propositions from $\{a_1, \ldots, a_n\}$ we construct an AnsDatalog program $\Pi_2 \cup D$ – where D is dependent on ϕ and Π_2 is fixed, such that ϕ is unsatisfiable iff $\Pi_2 \cup D \models unsat$.

Let ϕ be of the form $(l_{11} \vee l_{12} \vee l_{13}) \wedge \ldots \wedge (l_{m1} \vee l_{m2} \vee l_{m3})$, where l_{ij}'s are literals made of the propositions $a_1, \ldots, a_n$.

D consists of the facts:

$conj(l_{11}, l_{12}, l_{13}) \leftarrow$. $\quad \ldots \quad$ $conj(l_{m1}, l_{12}, l_{m3}) \leftarrow$.

The fixed program Π_2 consists of the following:

1. It has rules of the following form – where $h(a)$ means a holds, or a is *true*, and $n\_h(a)$ means a does not hold, or a is $false$ – to enumerate truth of propositions in the language:

   $h(a_i) \leftarrow$ **not** $n\_h(a_i)$.

   $n\_h(a_i) \leftarrow$ **not** $h(a_i)$.

   for $i = 1 \ldots n$.

2. It has the following rules which makes *unsat* true if one of the conjunct is false:

   $unsat \leftarrow conj(X, Y, Z), n\_h(X), n\_h(Y), n\_h(Z)$. □

Theorem 6.2.6 AnsDatalog is program-complete for co-NEXPTIME. □

Proof: (sketch)

(membership): For an AnsDatalog program Π the size of $ground(\Pi)$ is exponential with respect to the size of Π. Given an atom A, we will argue that determining if $\Pi \not\models A$ is in NEXPTIME. This means we need to find an answer set of Π where A is not true. To find an answer set we need to guess and check. Since the checking part involves computing answer set of an AnsDatalog$^{\neg}$**not** program, from Theorem 6.2.3 checking is exponential time. Hence, determining if $\Pi \not\models A$ is in NEXPTIME, and therefore determining if $\Pi \models A$ is in co-NEXPTIME.

(hardness): The proof of hardness is similar to the proof of the hardness part in Theorem 6.2.3, except that we now need to simulate a non-deterministic Turing machine (NDTM). In an NDTM δ is a relation instead of a function. Thus in part (5) of the hardness proof of Theorem 6.2.3 we may have multiple facts of the form $trans(s, \alpha, s_1, \beta_1, p_1) \ldots trans(s, \alpha, s_k, \beta_k, p_k)$. Thus the rules in part (6) of the hardness proof of Theorem 6.2.3 are no longer adequate. What we need to do is to simulate the multiple branches of computation that an NDTM can take. This is done by introducing a predicate *occurs* as follows and replacing *trans* in the bodies of the rules in part (6) of the hardness proof of Theorem 6.2.3 by *occurs*.

$other\_occurs(S, A, S_1, B_1, P_1, T) \leftarrow occurs(S, A, S_2, B_2, P_2, T), S_1 \neq S_2.$
$other\_occurs(S, A, S_1, B_1, P_1, T) \leftarrow occurs(S, A, S_2, B_2, P_2, T), B_1 \neq B_2.$
$other\_occurs(S, A, S_1, B_1, P_1, T) \leftarrow occurs(S, A, S_2, B_2, P_2, T), P_1 \neq P_2.$
$occurs(S, A, S_1, B_1, P_1, T) \leftarrow symbol(A, N, T), state(S, T), cursor(N, T), trans(S, A, S_1, B_1, P_1),$
not $other\_occurs(S, A, S_1, B_1, P_1, T).$

The above rules ensure that at each time T there is exactly one answer set mimicking a possible transition. Thus the various computation paths have a 1-1 correspondence with the various answer sets.

Finally we replace the rule with accept in the head in the hardness proof of Theorem 6.2.3 by the following rules:

$reject \leftarrow state(no, T)$.

Now *reject* is true in one of the answer sets of the above program iff the NDTM rejects the input. Thus, rejection will be in NEXPTIME and hence acceptance will be in co-NEXPTIME. □.

Theorem 6.2.7 Stratified AnsDatalog is data-complete for **P**. □

Proof (sketch):

(membership): There are polynomial (in the size of the facts) number of strata. Iteration in each strata is polynomial time. So the unique answer set can be obtained in polynomial time.

(hardness): Same as the proof of the hardness part of Theorem 6.2.2. □

Theorem 6.2.8 Stratified AnsDatalog is program-complete for EXPTIME. □

Proof (sketch):

(membership): There could be an exponential (in the size of the program) number of strata in the grounding of the program. Iteration in each strata is polynomial time. So the unique answer set can be obtained in exponential time.

(hardness): Same as the proof of the hardness part of Theorem 6.2.3. □

The above results also hold for the well-founded semantics of AnsDatalog where instead of static stratas we have stratas that are determined dynamically.

Theorem 6.2.9 AnsDatalog with respect to well-founded semantics is data-complete for **P**. □

Theorem 6.2.10 AnsDatalog with respect to well-founded semantics is program-complete for EXPTIME. □

6.2.4 Complexity of AnsDatalog$^{or,\neg\mathbf{not}}$

Unlike AnsDatalog$^{\neg\mathbf{not}}$ programs, AnsDatalog$^{or,\neg\mathbf{not}}$ programs may have multiple answer sets. The multiplicity of the answer sets is due to the *or* connective that is now allowed in the head of rules. Moreover, as discussed in Example 29 a disjunctive fact of the form $a\ or\ b \leftarrow$. in a program can not be in general replaced by the two AnsDatalog rules in $\{a \leftarrow \mathbf{not}\ b.; b \leftarrow \mathbf{not}\ a\}$. In this section we formally show that entailment of negative literals with respect to AnsDatalog$^{or,\neg\mathbf{not}}$ programs is more complex than with respect to AnsDatalog programs. This is due to the extra minimality condition in the definition of answer sets of AnsDatalog$^{or,\neg\mathbf{not}}$ programs. Interestingly, this does not affect the complexity of entailment with respect to positive literals, as an AnsDatalog$^{or,\neg\mathbf{not}}$ program entails a positive literal A iff all its answer sets entail A iff all its models entail A. The last 'iff' does not hold when A is a negative literal. We now formally state and prove the complexity results about entailment with respect to AnsDatalog$^{or,\neg\mathbf{not}}$ programs. In this we only consider the data complexity.

Theorem 6.2.11 Given an AnsDatalog$^{or,\neg\textbf{not}}$ program Π, a set of facts D and a positive literal A, determining $\Pi \cup D \models A$ is **coNP**-complete with respect to the size of D. □

Proof:

(membership): $\Pi \cup D \models A$ can be written as $\forall M.P_3(M, D, A)$, where $P_3(M, D, A)$ is true if M is a model of $\Pi \cup D$ implies $M \models A$. Checking if M is a model of $ground(\Pi) \cup D$ and determining if $M \models A$ is polynomial in the size of D. Hence, membership in P_3 is polynomially decidable. Moreover, M is polynomial in the size of D, and hence P_3 is polynomially balanced. Thus determining if $\Pi \models A$ is in $\Pi_1 P$, which is same as co-NP.

An equivalent way of showing the above is to show that the complement of the above problem is in NP. For that we have to guess an M, show that M is a model of $\Pi \cup D$ and $M \not\models A$. The later two can be done in polynomial time. Thus the complement of the above problem is in NP. Hence, determining if $\Pi \cup D \models A$ is in co-NP.

(hardness): We show this by considering the well-known co-NP Complete problem unsatisfiability. Let $\{a_1, \ldots, a_n\}$ be a set of propositions. Given any instance ϕ of 3-Sat consisting of propositions from $\{a_1, \ldots, a_n\}$ we construct an AnsDatalog$^{or,\neg\textbf{not}}$ program $\Pi_3 \cup D$ – where D is dependent on ϕ and Π_3 is fixed, such that ϕ is unsatisfiable iff $\Pi_3 \cup D \models unsat$.

Let ϕ be of the form $(l_{11} \vee l_{12} \vee l_{13}) \wedge \ldots \wedge (l_{m1} \vee l_{m2} \vee l_{m3})$, where l_{ij}'s are literals made of the propositions $a_1, \ldots, a_n$.

D consists of the facts:

$conj(l_{11}, l_{12}, l_{13}) \leftarrow.$ $\quad \ldots \quad$ $conj(l_{m1}, l_{12}, l_{m3}) \leftarrow.$

The fixed program Π_3 consists of the following:

1. It has rules of the following form – where $h(a)$ means a holds, or a is $true$, and $n\_h(a)$ means a does not hold, or a is $false$ – to enumerate truth of propositions in the language:

   $h(a_i)\ or\ n\_h(a_i) \leftarrow.$

   for $i = 1 \ldots n$.

2. It has the following rules which makes $unsat$ true if one of the conjunct is false:

   $unsat \leftarrow conj(X, Y, Z), n\_h(X), n\_h(Y), n\_h(Z).$ □

Theorem 6.2.12 Given an AnsDatalog$^{or,\neg\textbf{not}}$ program Π, a set of facts D and a negative literal $\neg A$, determining $\Pi \cup D \models \neg A$ is Π_2**P**-complete with respect to the size of D. □

Proof:

(membership): $\Pi \cup D \models \neg A$ can be written as $\forall M \exists M' P_4(M, M', D, A)$, where $P_4(M, M', D, A)$ is true if M is a model of $\Pi \cup D$ implies M' is a model of $\Pi \cup D$ and $M' \subseteq M$ and $M' \not\models A$. Checking if M and M' are models of $\Pi \cup D$ and determining if $M' \not\models A$ takes time polynomial in the size of D. Hence, P_4 is polynomially decidable. Moreover, since M and M' are polynomial in the size of D, P_4 is polynomially balanced. Thus determining if $\Pi \cup D \models \neg A$ is in $\Pi_2 P$, which is same as co-NP^{NP}.

(hardness): We show this by considering the $\Pi_2 P$-complete problem, satisfiability of quantified boolean formulas ψ of the form $\forall x_1, \ldots x_n \exists y_1, \ldots, y_m \phi$, where ϕ is a propositional formula made up of the propositions $x_1, \ldots, x_n, y_1, \ldots, y_m$. Given a formula ψ of the above form we construct an AnsDatalog$^{or, \neg \mathbf{not}}$ program $\Pi_4 \cup D$ – where D is dependent on ψ and Π_4 is fixed, such that ψ is satisfiable iff $\Pi_4 \cup D \models \neg unsat$.

Let ϕ be of the form $(l_{11} \vee \ldots \vee l_{1k_1}) \wedge \ldots \wedge (l_{m1} \vee \ldots \vee l_{mk_m})$, where l_{ij}'s are literals made of the propositions $x_1, \ldots, x_n, y_1, \ldots, y_m$.

D consists of the following facts:

1. $conj(l_{11}, l_{12}, l_{13}) \leftarrow$. $\quad \ldots \quad$ $conj(l_{m1}, l_{12}, l_{m3}) \leftarrow$.
2. $forall(x_1) \leftarrow$. $\quad \ldots \quad$ $forall(x_n) \leftarrow$.
3. $exists(y_1) \leftarrow$. $\quad \ldots \quad$ $exists(y_m) \leftarrow$.

The fixed program Π_4 consists of the following:

1. It has rules of the following form to enumerate truth of propositions.

   $h(X)$ *or* $n\_h(X) \leftarrow forall(X)$.
   $h(X)$ *or* $n\_h(X) \leftarrow exists(X)$.

2. It has the following rule which makes *unsat* true if one of the conjunct is false:

   $unsat \leftarrow conj(X, Y, Z), n\_h(X), n\_h(Y), n\_h(Z)$.

3. Rules of the form:

   $h(X) \leftarrow exists(X), unsat$.
   $n\_h(X) \leftarrow exists(X), unsat$.

□

Exercise 20 Let Π be an AnsDatalog$^{or, \neg \mathbf{not}}$ program. Show that $\Pi \models \neg A$ iff for all models M of Π there exists a model $M' \subseteq M$ of Π such that $M' \not\models A$.

□

6.2.5 Complexity of AnsDatalog^{or}

Theorem 6.2.13 AnsDatalog^{or} is data-complete for Π_2**P**. □

Proof:

(membership) : Let Π be an AnsDatalog^{or} program and A be an atom. $\Pi \models A$ can be written as
$\forall M$. M is an answer set of Π^M implies $M \models A$
$\equiv \forall M$. [M is a model of Π^M and $\neg$ $(\exists M', M' \subset M$ and M' is a model of $\Pi^M)$] implies $M \models A$.
$\equiv \forall M$. [$\neg$ (M is a model of Π^M) or $(\exists M', M' \subset M$ and M' is a model of Π^M) or $(M \models A)$.
$\equiv \forall M$. [$\neg$ ((M is a model of Π^M) and $\neg(M \models A)$) or $(\exists M', M' \subset M$ and M' is a model of Π^M)]
$\equiv \forall M$. [(M is a model of Π^M) and $(M \not\models A)$] implies $(\exists M', M' \subset M$ and M' is a model of Π^M).
$\equiv \forall M$. $\exists M'$ (M is an a model of Π^M and $M \not\models A$) implies ($M' \subset M$ and M' is a model of Π^M).

Thus $\Pi \cup D \models A$ can be written as $\forall M \exists M' P_5(M, M', D, A)$, where $P_5(M, M', D, A)$ is true if (M is a model of $\Pi^M \cup D$ and $M \not\models A$) implies ($M' \subset M$ and M' is a model of $\Pi^M \cup D$). It is easy to see that given M, M' and A, whether $P_5(M, M', D, A)$ holds or not can be determined in time polynomial in the size of D. Hence, P_5 is polynomially decidable. Moreover, since M and M' are polynomial in the size of D, P_5 is polynomially balanced. Hence for AnsDatalog^{or} programs $\Pi \cup D$ determining if $\Pi \cup \models A$ is in $\Pi_2 P$. In exactly the same way we can show that for AnsDatalog^{or} programs $\Pi \cup D$ determining if $\Pi \cup D \models \neg A$ is in $\Pi_2 P$.

(hardness) : The hardness proof is same as the proof of hardness for Theorem 6.2.12. □

Theorem 6.2.14 AnsDatalog^{or} is program-complete for co-NEXPTIME^{NP}. □

The proof of the above theorem is similar to the proof of Theorem 6.2.6. The additional NP in the exponent of co-NEXPTIME^{NP} is due to the additional minimality necessary in AnsDatalog^{or} programs, and necessitates an NP oracle together with the NDTM when showing the hardness part.

We now summarize the various complexity results for the different AnsDatalog* sub-classes.

6.2.6 Summary of the complexity results of AnsDatalog* sub-classes

AnsDatalog* Class	complexity type	complexity class
AnsDatalog$^{\neg}$**not**	Data complexity	P-complete
AnsDatalog$^{\neg}$**not**	Program Complexity	EXPTIME-complete
Stratified AnsDatalog	Data Complexity	P-complete
Stratified AnsDatalog	Program complexity	EXPTIME-complete
AnsDatalog (under WFS)	Data Complexity	P-complete
AnsDatalog (under WFS)	Program Complexity	EXPTIME-complete
AnsDatalog (answer set existence)	Complexity of $SM(\Pi) \neq \emptyset$	NP-complete
AnsDatalog	Data Complexity	Co-NP Complete
AnsDatalog	Program Complexity	Co-NEXPTIME Complete
AnsDatalog$^{\neg}$	Existence of answer set	NP-complete
AnsDatalog$^{\neg}$	Data complexity	coNP-complete
AnsDatalog$^{or,\neg}$**not**	Deciding $\Pi \models_{GCWA} A$	Co-NP Complete
AnsDatalog$^{or,\neg}$**not**	Deciding $\Pi \models_{GCWA} \neg A$	Π_2P-complete
AnsDatalog^{or}	Data complexity	Π_2P-complete
AnsDatalog^{or}	Program complexity	Co-NEXPTIME^{NP}-complete

6.3 Expressibility of AnsDatalog* sub-classes

In this section our interest is in the expressibility of AnsDatalog* sub-classes. In this we are interested in two kinds of results: When a particular sub-class (completely) captures a complexity class or can only express a strict subset of it; and how a AnsDatalog* sub-class relates to other query languages such as First-order logic (FOL), fixpoint logic (FPL), relational algebra, and relational calculus. FPL is an extension of first-order logic by a least fixpoint operator.

There are two mismatches between Turing machines and some of the AnsDatalog* sub-classes. These mismatches are: (i) the input in a Turing machine automatically encodes a linear order among the constituent of the input, while such an order between the constants in the AnsDatalog* program is not automatically given; and (ii) a Turing machine can check what is in its input and

what is not, while certain sub-classes of AnsDatalog* do not have a way to find out if some atom is $false$.

The issue in (i) can be overcome in AnsDatalog* sub-classes that can encode non-determinism. Then the program can have multiple answer sets each encoding a particular ordering. An example of such an encoding is given in Section 2.1.12, and we need the use of either '**not** ' or ' or ' for encoding non-determinism. In the absence of non-determinism the only other option is to assume the presence of an ordering. The issue in (ii) can be overcome either by using '**not** ' or some weaker negation.

Because of the above although AnsDatalog$^{\neg\mathbf{not}}$ – known in the literature as Datalog – is data-complete in **P** it can not capture **P** and only captures a strict subset of **P**. In particular, it cannot express the query about whether the universe of the input database has an even number of elements. The book [AHV95] and the survey articles [DEGV97, DEGV99] discuss this and similar results in further detail. When Datalog is augmented with the assumption that the input predicates may appear negated in rule bodies then it still only captures a strict subset of **P**. This extension of Datalog is referred to as Datalog^{+}. It is known that Datalog^{+} is equivalent in expressiveness to FPL$^{+}\exists$), a fragment of FPL where negation is restricted to the input relations and only existential quantifiers are allowed. On the other hand Datalog^{+} together with the assumption that the input database is ordered captures **P**.

In the following table we summarize several of the expressibility results. Among these results we present the proof of the expressibility results about AnsDatalog and AnsDatalog^{or} . In those results by brave semantics we mean that $\Pi \models L$ iff there exists an answer set A of Π such that $A \models L$. Also, by AnsDatalog$^{or,\neg\mathbf{not},\neq}$ we refer to the extension of AnsDatalog$^{or,\neg\mathbf{not}}$ with the $\neq$ predicate. The other results in the table, some with proofs, are discussed in greater detail in [AHV95, DEGV97, DEGV99].

AnsDatalog* sub-class	relation	complexity class (or a non-AnsProlog* class)
Datalog^{+}	$\subset\neq$	P
Datalog^{+} (on ordered databases)	captures	P
Datalog^{+}	equal	$FPL^{+}(\exists)$
Stratified AnsDatalog	$\subset\neq$	FPL
Non-recursive range restr AnsDatalog	equal	relational algebra
Non-recursive range restr AnsDatalog	equal	relational calculus
Non-recursive range restr AnsDatalog	equal	FOL (without function symbols)
AnsDatalog (under WFS)	equal	FPL
Stratified AnsDatalog (on ordered databases)	captures	P
AnsDatalog under WFS (on ordered databases)	captures	P
AnsDatalog under brave semantics	captures	NP
AnsDatalog	captures	Co-NP
AnsDatalog$^{or,\neg\mathbf{not},\neq}$ (under brave semantics)	captures	Σ_2P
AnsDatalog$^{or,\neg\mathbf{not},\neq}$	captures	Π_2P
AnsDatalog^{or} (under brave semantics)	captures	Σ_2P
AnsDatalog^{or}	captures	Π_2P

6.3.1 Expressibility of AnsDatalog

Theorem 6.3.1 AnsDatalog under the brave semantics captures **NP**. □

Proof:

• *(membership)*: Same as the membership part of the proof of Theorem 6.2.5.

• *(expresses all* **NP** *relations)*: To show every problem of complexity class **NP** can be expressed in AnsDatalog we show that the general form of a problem in the complexity class **NP** as shown in part (2) of Proposition 89 can be expressed in AnsDatalog. Thus, we construct an AnsDatalog program Π such that a finite database w satisfies $\exists U_1, \ldots U_m \forall \bar{x} \exists \bar{y}(\theta_1(\bar{x}, \bar{y}) \vee \ldots \vee \theta_k(\bar{x}, \bar{y}))$ iff $\Pi \cup w$ has an answer set containing *yes*.

1. For enumeration of the predicates $U_1, \ldots, U_m$, we have the rules:

   $U_j(\bar{w_j}) \leftarrow$ **not** $U'_j(\bar{w_j})$.
   $U'_j(\bar{w_j}) \leftarrow$ **not** $U_j(\bar{w_j})$.

   for $j = 1 \ldots m$

2. For a given $\bar{x}$, we define $p(\bar{x})$ to hold when $\exists \bar{y}(\theta_1(\bar{x}, \bar{y}) \vee \ldots \vee \theta_k(\bar{x}, \bar{y}))$ holds by the following rules:

   $p(\bar{x}) \leftarrow \theta_i(\bar{x}, \bar{y})$.

   for $i = 1 \ldots k$.

3. To make q true if for some $\bar{x}$, $p(\bar{x})$ does not hold we have the rule:

   $q \leftarrow$ **not** $p(\bar{x})$.

4. To eliminate answer sets where q may be true we have:

   inconsistent $\leftarrow q,$ **not** *inconsistent*.

5. Finally to include *yes* in the answer sets which survive the elimination above we have:

   yes $\leftarrow$. □

Corollary 5 AnsDatalog captures **co-NP**. □

6.3.2 Expressibility of AnsDatalog ^{or}

Theorem 6.3.2 AnsDatalog ^{or} under the brave semantics captures $\Sigma_2\mathbf{P}$. □

Proof:

• *(membership)*: Same as the membership part of the proof in Theorem 6.2.13.

• *(expresses all* $\Sigma_2\mathbf{P}$ *relations)*: Following part (3) of Proposition 89 we construct an AnsDatalog ^{or} program Π such that a finite database w satisfies the formula $\exists U_1, \ldots U_m \forall V_1, \ldots, V_n \exists \bar{x}(\theta_1(\bar{x}) \vee \ldots \vee \theta_k(\bar{x}))$ (where $\theta_i(\bar{x})$s are of the form described in part (3) of Proposition 89) iff $\Pi \cup w$ has an answer set containing *sat*. This will prove that any problem in $\Sigma_2\mathbf{P}$ can be expressed in AnsDatalog ^{or} under the brave answer set semantics.

1. For enumeration of the predicates $U_1, \ldots, U_m$, we have the rules:

   $U_j(\bar{w_j})$ *or* $U'_j(\bar{w_j}) \leftarrow$.

   for $j = 1 \ldots m$

2. For enumeration of the predicates $V_1, \ldots, V_n$, we have the rules:

   $V_j(\bar{s_j})$ *or* $V'_j(\bar{s_j}) \leftarrow$.

   for $j = 1 \ldots n$

3. Definition of linear ordering: Since $\theta_i(\bar{x})$s may include enumeration literals made up of predicates *succ*, $first$ and *last*, we need to define these predicates. The following rules achieve that. They are similar to the rules in Section 2.1.12, except that use **not** in only one place, and use *or* .

   $prec(X, Y)$ *or* $prec(Y, X) \leftarrow$ **not** $eq(X, Y)$.
   $eq(X, X) \leftarrow$.
   $prec(X, Z) \leftarrow prec(X, Y), prec(Y, Z)$.
   $not\_succ(X, Z) \leftarrow prec(Z, X)$.
   $not\_succ(X, Z) \leftarrow prec(X, Y), prec(Y, Z)$.
   $not\_succ(X, X) \leftarrow$.
   $succ(X, Y)$ *or* $not\_succ(X, Y) \leftarrow$.
   $not\_first(X) \leftarrow prec(Y, X)$.
   $first(X)$ *or* $not\_first(X) \leftarrow$.
   $not\_last(X) \leftarrow prec(X, Y)$.
   $last(X)$ *or* $not\_last(X) \leftarrow$.
   $reachable(X) \leftarrow first(X)$.
   $reachable(Y) \leftarrow reachable(X), succ(X, Y)$.
   $linear \leftarrow last(X), reachable(X)$.

4. Definition of satisfiability

   $sat \leftarrow \theta_i(\bar{x}), linear$.

   for $i = 1 \ldots k$.

5. To eliminate potential answer sets where for particular instances of U_i's not all interpretations of the V_j's lead to *sat*, we add the following rules:

   $V_j(\bar{s_j}) \leftarrow sat$.
   $V'_j(\bar{s_j}) \leftarrow sat$. □

Corollary 6 AnsDatalog^{or} captures $\Pi_2\mathbf{P}$. □

Theorem 6.3.3 AnsDatalog$^{or,\neg\mathbf{not},\neq}$ under the brave semantics captures $\Sigma_2\mathbf{P}$. □

Proof: Almost same as the proof of Theorem 6.3.2. The only change is that we need to replace the first two rules in the program constructed in the item (3) of the second part of the proof by the following rule:

$prec(X,Y)$ *or* $prec(Y,X) \leftarrow X \neq Y$.

With this change the program constructed in the second part of the proof is an AnsDatalog$^{or\ ,\neg\mathbf{not},\neq}$ program.

Corollary 7 AnsDatalog$^{or\ ,\neg\mathbf{not},\neq}$ captures $\Pi_2\mathbf{P}$. □

6.4 Complexity and expressibility of AnsProlog* sub-classes

In this section we consider the complexity and expressibility of AnsProlog* sub-classes when we allow function symbols. In this case, even for AnsProlog$^{\neg\mathbf{not}}$ programs we may need infinite iterations of the iterative fixpoint operator to get to the answer set. Thus the set of answer sets of AnsProlog$^{\neg\mathbf{not}}$ programs is recursively enumerable. In general, when we allow function symbols the complexity and expressibility classes of AnsProlog* sub-classes are no longer in the polynomial hierarchy; rather they are in the arithmetic and analytical hierarchy.

6.4.1 Complexity of AnsProlog* sub-classes

We start with the complexity of AnsProlog$^{\neg\mathbf{not}}$.

Theorem 6.4.1 AnsProlog$^{\neg\mathbf{not}}$ is r.e. complete. □

Proof As before, to show AnsProlog$^{\neg\mathbf{not}}$ is r.e. complete we will show that (a) membership: answer set of a AnsProlog$^{\neg\mathbf{not}}$ program is an r.e. set; and (b)hardness: The r.e. complete problem of Turing acceptability can be expressed in AnsProlog$^{\neg\mathbf{not}}$.

• *(membership)*: Recall that AnsProlog$^{\neg\mathbf{not}}$ programs have unique answer sets that can be determined by iterative application of an operator until the least fixpoint is reached. Each application of the operator is computable, but the fixpoint may not be reached after a finite number of application of the operator, although it is reached eventually. Thus in general the answer set of a AnsProlog$^{\neg\mathbf{not}}$ program is an r.e. set, but for some programs it may not be recursive, as the fixpoint may not be reached after a finite number of applications of the operator.

• *(hardness)*: In the appendix we define a Turning machine M and when it accepts an input I. We now present a translation of an arbitrary Turing machine M and an arbitrary input I to an AnsProlog$^{\neg\mathbf{not}}$ program $\pi(M,I)$ and argue that $M(I) =$ 'yes' iff $\pi(M,I) \models accept$.

1. We first define the various predicates that we will use and their intuitive meaning.
   (a) $state(s,t)$: at time point t the state of the Turing machine is s.
   (b) $cursor(n,t)$: at time point t the cursor is at the cell number n (counted from the left).
   (c) $symbol(\alpha,n,t)$: at time point t, α is the symbol in cell number n (counted from the left) of the tape.
   (d) $trans(s,\alpha,s',\beta,p)$: if s is the current state of the Turing machine, and α is the symbol pointed to by the cursor, then the new state should be s', α should be over-written by β, and the cursor should move as dictated by p.
2. We now present the part of $\pi(M,I)$ that describes the initial state, the initial cursor position, the input, and the transition function δ of M.

(a) $state(s_0, 0) \leftarrow$: at time point 0 the state of the Turing machine is s_0.

(b) $cursor(0, 0) \leftarrow$: at time point 0 the cursor is at the left most end of the tape.

(c) $symbol(\alpha_1, 1, 0) \leftarrow$. ... $symbol(\alpha_k, k, 0) \leftarrow$.
if the input $I = \alpha_1 \ldots \alpha_k$.

$symbol(>, 0, 0) \leftarrow$.
$symbol(\sqcup, k+1, 0) \leftarrow$.

(d) $trans(s, \alpha, s', \beta, p)$: as specified by δ. Note that there are restriction on this such as $\delta(S, >) = (S', >, \rightarrow)$.

3. We now describe the remaining rules of $\pi(M, I)$ that describe how the states, the symbols in the tape and the position of the cursor changes.

(a) State change:

$state(S, T+1) \leftarrow state(S', T), cursor(N, T), symbol(A, N, T), trans(S', A, S, \_, \_)$.

(b) Cursor location change:

$cursor(N, T+1) \leftarrow state(S', T), cursor(N, T), symbol(A, N, T), trans(S', A, S, \_, -)$.
$cursor(N-1, T+1) \leftarrow state(S', T), cursor(N, T), symbol(A, N, T),$
$\qquad trans(S', A, S, \_, \leftarrow)$.
$cursor(N+1, T+1) \leftarrow state(S', T), cursor(N, T), symbol(A, N, T),$
$\qquad trans(S', A, S, \_, \rightarrow)$.

(c) Which symbols change and which do not:

$symbol(B, N, T+1) \leftarrow state(S', T), cursor(N, T), symbol(A, N, T), trans(S', A, S, B, \_)$.
$symbol(A, N, T+1) \leftarrow cursor(N', T), symbol(A, N, T), less\_than(N, N')$.
$symbol(A, N, T+1) \leftarrow cursor(N', T), symbol(A, N, T), less\_than(N', N)$.

(d) Auxiliary predicates. (Note that we can not define *eq* and then *neq* as the negation of *eq*, as we do not have **not** in our language.)

$less\_than(N, N+1)$.
$less\_than(X, Y) \leftarrow less\_than(X, Z), less\_than(Z, Y)$

$accept \leftarrow state(yes, T)$.

The change in the states, tape symbols and position of the cursor as defined by the above rules mimic the corresponding changes in the Turing machines. Hence, $M(I) =$ '*yes*' iff $\pi(M, I) \models accept$. □

We now consider the complexity of AnsProlog programs.

Theorem 6.4.2 AnsProlog is Π^1_1 complete. □

Proof: (sketch) We show AnsProlog is Π^1_1 complete by first showing that entailment in AnsProlog is in Π^1_1.

• *(membership)*: We show how an AnsProlog program Π can be transformed to a formula Φ_Π of the form $\forall \mathbf{P} \phi(\mathbf{P}; \mathbf{X})$, such that $\Phi_\Pi(a)$ is *true* iff $\Pi \models R(a)$. The transformation is based on characterizing $\Pi \models R(a)$ in SOL as $\forall S, S'.S$ is a model of Π implies ($S \models R(a)$ or (S' is a model of Π^S implies $S \subseteq S'$)).

Without loss of generality let us assume that Π has two sets of predicates, EDBs and IDBs with the restriction that EDBs do not appear in the head of rules. Let $\mathbf{P}$ be the set of IDB predicates in Π. Let us denote by $\phi_\Pi(\mathbf{S})$ the universal first-order formula obtained from Π, by treating $\leftarrow$ as classical reverse implication, by replacing each P_i from $\mathbf{P}$ by a new predicate variable S_i of the same arity, and then doing the conjunction of all the clauses obtained. Let us denote by $\psi_\Pi(\mathbf{S}, \mathbf{S}')$ the universal first-order formula obtained from Π, by treating $\leftarrow$ as classical reverse implication, by replacing each P_i from $\mathbf{P}$ by a new predicate variable S'_i of the same arity, for every naf-literal **not** $P_i(\mathbf{t})$, adding the literal $\neg S_i(\mathbf{t})$ to the body and then doing the conjunction of all the clauses obtained. The desired formula $\Phi_\Pi(\mathbf{x})$ is

$\forall \mathbf{S}, \mathbf{S}'.\phi_\Pi(\mathbf{S}) \Rightarrow (R(\mathbf{x}) \vee (\psi_\Pi(\mathbf{S}, \mathbf{S}') \Rightarrow \mathbf{S} \leq \mathbf{S}'))$.

where $(\mathbf{S} \leq \mathbf{S}')$ stands for $\bigwedge_i(\forall \mathbf{x}_i.(S_i(\mathbf{x}_i) \Rightarrow S'_i(\mathbf{x}_i)))$.

(Note that the intuitive meaning of the formulas $\phi_\Pi(\mathbf{S})$ and $\psi_\Pi(\mathbf{S}, \mathbf{S}')$ are that $\mathbf{S}$ is a model of Π and $\mathbf{S}'$ is a model of $\Pi^{\mathbf{S}}$ respectively. More precisely, $\mathbf{S}'$ is a model of $\Pi^{\mathbf{S}}$ iff $\mathbf{S} \cup \mathbf{S}'$ with the transformation is a model of $\psi_\Pi(\mathbf{S}, \mathbf{S}')$. The Example 109 below illustrates this further.)

It is easy to see that $\Phi_\Pi(\mathbf{x})$ is a Π^1_1 formula, $\Phi_\Pi(a)$ is *true* iff $\Pi \models R(a)$.

This completes the proof of membership. The following example illustrates the transformation in the above proof.

Example 109 Consider the following AnsProlog program Π:

$p_1(a) \leftarrow$ **not** $p_2(a)$.
$p_2(a) \leftarrow$ **not** $p_1(a)$.
$p_1(b) \leftarrow$ **not** $p_2(b)$.
$p_2(b) \leftarrow$ **not** $p_1(b)$.

Let $S = \{p_1(a), p_1(b), p_2(b)\}$ and $S' = \{p_1(a)\}$. It is easy to see that S' is a model of Π^S.

The formula $\psi_\Pi(\mathbf{S}, \mathbf{S}')$ is the conjunction of the following four:

$s'_1(a) \Leftarrow \neg s'_2(a), \neg s_2(a)$
$s'_2(a) \Leftarrow \neg s'_1(a), \neg s_1(a)$
$s'_1(b) \Leftarrow \neg s'_2(b), \neg s_2(b)$
$s'_2(b) \Leftarrow \neg s'_1(b), \neg s_1(b)$

It can be now easily shown that the transformation of $S \cup S'$ which is $\{s_1(a), s_1(b), s_2(b), s'_1(a)\}$ is a model of $\psi_\Pi(\mathbf{S}, \mathbf{S}')$. □

• Next we need to show that a Π^1_1 complete problem is expressible in AnsProlog. We show this by first pointing out that the XYZ problem is Π^1_1 complete []. Next we point to our proof of Theorem 6.4.6 where we will show that the general form of Π^1_1 problems is expressible in AnsProlog. Hence, the Π^1_1 complete problem XYZ is expressible in AnsProlog. This completes our proof. □

Theorem 6.4.3 AnsProlog under well-founded semantics is Π^1_1 complete. □

Theorem 6.4.4 AnsProlog^{or} is Π^1_1 complete. □

Proof: (sketch) We show AnsProlog^{or} is Π^1_1 complete by first showing that entailment in AnsProlog is in Π^1_1.

• *(membership)*: We show that an AnsProlog^{or} program Π can be transformed to a formula Φ_Π of the form $\forall \mathbf{P}\phi(\mathbf{P};\mathbf{X})$, such that $\Phi_\Pi(a)$ is *true* iff $\Pi \models R(a)$. The transformation is based on characterizing $\Pi \models R(a)$ in SOL as $\forall S.\ \exists S'$ (S is a model of Π) implies ($S \models R(a)$ or ($S' \subset S$ and S' is a model of Π^S)).

Thus $\Phi_\Pi(\mathbf{x})$ is given by the following formula that uses the formulas $\phi_\Pi(\mathbf{S})$ and $\psi_\Pi(\mathbf{S},\mathbf{S}')$ defined earlier.

$\forall \mathbf{S} \exists \mathbf{S}'.\phi_\Pi(\mathbf{S}) \Rightarrow [R(\mathbf{x}) \vee (\psi_\Pi(\mathbf{S},\mathbf{S}') \wedge (\mathbf{S}' < \mathbf{S}))]$

where $(\mathbf{S}' < \mathbf{S})$ stands for $\bigwedge_i(\forall \mathbf{x}_i.(S'_i(\mathbf{x}_i) \Rightarrow S_i(\mathbf{x}_i))) \wedge \bigwedge_i \exists \mathbf{y}_i.(S_i(\mathbf{y}_i) \wedge \neg S'_i(\mathbf{y}_i))$.

Although, $\Phi_\Pi(a)$ is *true* iff $\Pi \cup A \models R(a)$ holds, $\Phi_\Pi(\mathbf{x})$ is not of the form $\forall \mathbf{P}\phi(\mathbf{P};\mathbf{X})$, and hence not a Π^1_1 formula. Fortunately, it is a $\Pi^1_2(bool)$ formula, which is defined as a collection of Π^1_2 formulas whose first-order parts are boolean combinations of existential formulas. Eiter and Gottlob in [EG97] show that such formulas have equivalent Π^1_1 formulas. Hence there exist a Π^1_1 formula Ψ such that $\Psi(a)$ iff $\Pi \cup A \models R(a)$. This completes our proof.

• Next we need to show that a Π^1_1 complete problem is expressible in AnsProlog^{or}. Since AnsProlog is a sub-class of AnsProlog^{or}, the above follows from the earlier result that a Π^1_1 complete problem is expressible in AnsProlog. □

We now discuss some decidable sub-classes of AnsProlog* that allow function symbols. In [Sha84] Shapiro uses alternating Turing machines to show that AnsProlog$^{-\mathbf{not}}$ programs which satisfy certain restrictions are PSPACE-complete.

Theorem 6.4.5 *[Sha84]*AnsProlog$^{-\mathbf{not}}$ is PSPACE-complete if each rule is restricted as follows: the body contains only one atom, the size of the head is greater than or equal to that of the body, and the number of occurrences of any variable in the body is less than or equal to the number of its occurrences in the head. □

Dantsin and Voronkov [DV97] and Vorobyov and Voronkov [VV98] studied the complexity of non-recursive AnsProlog. In [VV98] complexity of non-recursive AnsProlog is classified based on the number of constants in the signature (k), number of unary functions (l), number of function symbols (m) of arity ≥ 2, presence of negation and range-restriction. We reproduce a summary of the classification from [DEGV99].

Signature	$(\geq 2, 0, 0)$	$(\_, 1, 0)$	$(\_, \geq 2, 0)$	$(\_, \_, \geq 1)$
			not range-restricted	
no negation	PSPACE	PSPACE	NEXPTIME	NEXPTIME
with negation	PSPACE	PSPACE	$TA(2^{O(n/logn)}, O(n/logn))$	NONELEMENTARY(n)
			range-restricted	
no negation	PSPACE	PSPACE	PSPACE	NEXPTIME
with negation	PSPACE	PSPACE	PSPACE	$TA(2^{n/logn}, n/logn)$

6.4.2 Summary of complexity results

AnsProlog* Class (With Functions)	complexity type	Complexity Class
AnsProlog$^{\neg\mathbf{not}}$	complexity	r.e. Complete
AnsProlog$^{\neg\mathbf{not}}$ (without recursion)	complexity	NEXPTIME-complete
AnsProlog$^{\neg\mathbf{not}}$ (with restrictions[2])	complexity	PSPACE-complete
Stratified AnsProlog (n levels of stratification)	complexity	Σ^0_{n+1}-complete
Non-recursive AnsProlog	Data complexity	P
AnsProlog (under WFS)	complexity	Π^1_1-complete
AnsProlog	complexity	Π^1_1 Complete
AnsProlog$^{or,\neg\mathbf{not}}$	under GCWA	Π^0_2 Complete
AnsProlog^{or}	complexity	Π^1_1-complete

6.4.3 Expressibility of AnsProlog* sub-classes

In this section we present the expressibility of AnsProlog* sub-classes when they have function symbols.

Theorem 6.4.6 AnsProlog captures Π^1_1. □

Proof: To show AnsProlog captures Π^1_1 complete, we have to show (i) (membership): any relation expressed using AnsProlog is in the class Π^1_1; and (ii) (captures Π^1_1): a general Π^1_1 relation can be expressed in AnsProlog.

• *(membership)*: Shown in the proof of Theorem 6.4.2.

• *(expresses all Π^1_1 relations)*: We present a transformation of general Π^1_1 relation Φ of the form $\forall\mathbf{P}\phi(\mathbf{P};\mathbf{X})$ – where $\mathbf{P}$ is a tuple of predicate variables and ϕ is a first order formula with free variables $\mathbf{X}$, to an AnsProlog program $\Pi(\Phi)$ and show that $\Phi(a)$ is *true* iff $\Pi(\Phi) \models R(a)$. In this we will use the result from [vBK83] that states that using second order skolemization a Φ is equivalent to a formula of the form $\forall\mathbf{P}\exists\mathbf{y}\forall\mathbf{z}.\phi(\mathbf{x},\mathbf{y},\mathbf{z})$. Using this result, assuming that ϕ is of the form $\bigwedge_{1\leq j\leq m}(l_{j1}(\mathbf{x},\mathbf{y},\mathbf{z}) \vee \ldots \vee l_{jk_j}(\mathbf{x},\mathbf{y},\mathbf{z}))$, and assuming that $\mathbf{P} = P_1, \ldots P_n$, we construct the program $\Pi(\Phi)$ as follows: (Note that when writing AnsProlog programs we use the convention that variables are in capital letters.)

1. For enumerating the predicate P_is, we have the following rules for $1 \leq i \leq n$.

   $P_i(\bar{X}) \leftarrow \mathbf{not}\ P'_i(\bar{X})$
   $P'_i(\bar{X}) \leftarrow \mathbf{not}\ P_i(\bar{X})$

   The above will guarantee that we will have different answer sets each expressing a particular interpretation of the P_is.

2. For $j = 1 \ldots m$, we have the following:

   $s_j(X,Y,Z) \leftarrow l'_{j1}(X,Y,Z), l'_{j2}(X,Y,Z), \ldots, l'_{jk_j}(X,Y,Z).$

   $unsat(X,Y) \leftarrow s_j(X,Y,Z).$

3. $R(X) \leftarrow \mathbf{not}\ unsat(X,Y).$

Now $\Phi(a)$ is *true* iff
$\forall \mathbf{P} \exists \mathbf{y} \forall \mathbf{z}.\phi(a, \mathbf{y}, \mathbf{z})$ is *true* iff
for all possible interpretations of $P_1, \ldots, P_n$, $\exists \mathbf{y} \forall \mathbf{z}.\phi(a, \mathbf{y}, \mathbf{z})$ is *true* iff
for all possible interpretations of $P_1, \ldots, P_n$, $\exists \mathbf{y} \neg (\exists \mathbf{z} \neg \phi(a, \mathbf{y}, \mathbf{z}))$ is *true* iff
for some Y, $unsat(a, Y)$ is $false$ in all answer sets of $\Pi(\Phi)$ iff
$R(a)$ is *true* in all answer sets of $\Pi(\Phi)$ iff
$\Pi(\Phi) \models R(a)$. □

Theorem 6.4.7 AnsProlog^{or} captures Π_1^1. □

Proof:

• *(membership)*: Shown in the proof of Theorem 6.4.4.

• *(expresses all Π_1^1 relations)*: Since AnsProlog programs are also AnsProlog^{or} programs, and in the last theorem we proved that AnsProlog expresses all Π_1^1 relations, we have that AnsProlog^{or} expresses all Π_1^1 relations. □

Theorem 6.4.8 AnsProlog (under well-founded semantics) captures Π_1^1. □

The following table summarizes the expressive power of AnsProlog* sub-classes.

AnsProlog* Class	relation	Complexity Class
AnsProlog (under WFS)	captures	Π_1^1
AnsProlog	captures	Π_1^1
AnsProlog^{or}	captures	Π_1^1

6.5 Compact representation and compilability of AnsProlog

In this section our focus is on the compactness properties of AnsProlog* as a knowledge representation language. In particular, knowing that entailment in AnsProlog is co-NP complete, we would like to know if there is a way to represent the same information in some other way with at most polynomial increase in size so that inferences can be made in polynomial time with respect this new representation. We will show that the answer to this question is negative, subject to the widely held belief that $\Sigma_2\mathbf{P} \neq \Pi_2\mathbf{P}$. This implies that for an alternative polynomial time inferencing it is very likely that there would be an exponential blow-up in the representation.

In the following we use the formulation from [CDS96, CDS94] to formalize the notion of compact representation and compilability. A problem P with fixed part F and varying part V will be denoted as $[P, F, V]$. Intuitively, a problem $[P, F, V]$ is compilable, if for each instance f of the fixed part F there is a data structured D_f of size polynomial in $|f|$ such that D_f can be used to solve the problem P in polynomial time. More formally,

Definition 78 *[CDS96]* A problem $[P, F, V]$ is compilable if there exists two polynomials p_1, p_2 and an algorithm ASK such that for each instance f of F there is a data structure D_f such that:

1. $|D_f| \leq p_1(|f|)$;
2. for each instance v of V the call $ASK(D_f, v)$ returns yes iff $\langle f, v \rangle$ is a "yes" instance of P; and

3. $ASK(D_f, v)$ requires time $\leq p_2(|v| + |D_f|)$. □

Example 110 Consider the problem $[T \models_{prop} q, T, q]$, where T is a propositional theory, $\models_{prop}$ is propositional entailment and q is a literal. Determining $T \models_{prop} q$ is co-NP complete. But the problem $[T \models_{prop} q, T, q]$ is compilable, as we can compile it to a set of literals entailed by T. This set is polynomial in size of the theory T (which includes representation of the alphabet of T), and whether a literal belongs to this set can be determined in polynomial time. The same reasoning would apply to any knowledge representation language with an entailment relation $\models$ if the space of possible queries is polynomial in the size of T.

Now consider the case where q is a conjunction of literals. In that case the space of possible queries is exponential in the size of T. But even then $[T \models_{prop} q, T, q]$ is compilable, as we can compile it to a set S of literals entailed by T and to answer any query which is a conjunction of literals we just have to check if each of the literals in the query is in S. This of course can be done in polynomial time.

The above reasoning no longer applies if q can be an arbitrary propositional formula. In [CDS96] it is shown that if $[T \models_{prop} q, T, q]$, where q is an arbitrary propositional formula, is compilable then $\Sigma_2\mathbf{P} \neq \Pi_2\mathbf{P}$, which is believed to be false. □

We now list similar results about AnsProlog.

Theorem 6.5.1 The problem $[\Pi \models q, \Pi, q]$, where Π is an AnsProlog program, and q is conjunction of literals, is compilable. □

Proof (sketch): As in Example 110 we can compile the above problem $[\Pi \models q, \Pi, q]$ to a set S of literals entailed by Π. S is obviously polynomial in the size of Π and to answer any query which is a conjunction of literals we just have to check if each of the literals in the query is in S. This of course can be done in polynomial time. □

Theorem 6.5.2 Unless $\Sigma_2\mathbf{P} \neq \Pi_2\mathbf{P}$, the problem $[\Pi \models q, \Pi, q]$, where Π is an AnsProlog program, and q is either a disjunction of positive literals or a disjunction of negative literals, is not compilable. □

Theorem 6.5.3 The problem $[\Pi \models_{brave} q, \Pi, q]$, where Π is an AnsProlog program, and q is disjunction of positive literals, is compilable. □

Proof (sketch): We can compile the above problem $[\Pi \models q, \Pi, q]$ to a set S of literals which is the union of all the answer sets of Π. S is obviously polynomial in the size of Π and to answer any query which is a disjunction of positive literals we just have to check if one of the literals in the query is in S. This of course can be done in polynomial time. □

Theorem 6.5.4 Unless $\Sigma_2\mathbf{P} \neq \Pi_2\mathbf{P}$, the problem $[\Pi \models_{brave} q, \Pi, q]$, where Π is an AnsProlog program, and q is either a conjunction of positive literals or a disjunction of negative literals, is not compilable. □

6.6 Relationship with other knowledge representation formalisms

In this section we relate AnsProlog* sub-classes with various other knowledge representation formalisms that have been proposed. In particular we consider classical logic formalisms, several non-monotonic formalisms and description logics. There are two directions in relating AnsProlog* with other formalisms: (i) Translating a program from a particular AnsProlog* sub-class to another formalism and showing the correspondence between answer sets of the original program and the 'models' of the translation. (ii) Translating a theory in another formalism to an AnsProlog* program and showing the correspondence between the 'models' of the original theory and the answer sets of the translation.

In both cases the expressibility results about languages shed light on the existence of such translations. For example, it was shown in [CEG97] that entailment in propositional default logic captures the class Π_2P. Since AnsDatalog^{or} also captures Π_2P, it means that any propositional default theory can be translated to an AnsDatalog^{or} program and vice-versa with a direct correspondence between their entailments. Similarly the result that AnsProlog captures the class Π^1_1 implies that the circumscriptive formalisms that are within Π^1_1 can be translated to an AnsProlog program and vice-versa with a direct correspondence between their entailments.

The important question about the translations is whether they are 'modular' or not. In Section 6.6.1 we formally define the notion of modular translation and show that there is no modular translations from AnsProlog to any monotonic logic, such as propositional and first order logic.

Because most of the other formalisms are syntactically more complex than AnsProlog* most of the results – in the literature – relating AnsProlog* with other non-monotonic formalisms are of the type (i). Our focus in this section will also be on these type of results.

We now briefly discuss the implications of such results. These results will tell a practioneer of another logic how to interpret AnsProlog* rules without having to master the semantics of AnsProlog*. Thus they lead to alternative intuitions about AnsProlog* constructs and also suggest how notions of minimization, default reasoning and knowledge modalities are expressed in AnsProlog*. In the past they have also led to transporting of notions from AnsProlog* to the other logics. For example, the idea of stratification was transported to auto-epistemic logic in [Gel87], the idea of splitting was transported to default logic in [Tur96], default logic was extended with the AnsProlog* connective '*or*' in [GLPT91], and well-founded semantics for default logic and auto-epistemic logic were developed in [BS93].

6.6.1 Inexistence of modular translations from AnsProlog* to monotonic logics

We now formally define the notion of a modular translation from one language to another and show that there is no modular translation from propositional AnsProlog to propositional logic.

Definition 79 A mapping $T(.)$ from the language L_1 to L_2 is said to be *rule-modular* if for any theory (or program) Π in L_1, for each set (possibly empty) of atomic facts F, the "models" of $\Pi \cup F$ and $T(\Pi) \cup F$ coincide. □

Proposition 90 There is no rule-modular mapping from propositional AnsProlog to propositional logic. □

Proof: Consider the AnsProlog program $\Pi = \{p \leftarrow \textbf{not } p\}$. Suppose there exists a modular mapping T from AnsProlog to propositional logic. Since Π does not have an answer set $T(\Pi)$ is

unsatisfiable. But $\Pi \cup \{p\}$ has the answer set $\{p\}$ while because of monotonicity of propositional logic, $T(\Pi) \cup \{p\}$ must remain unsatisfiable. Thus T can not be a modular mapping from AnsProlog to propositional logic. □

The above proof also suffices – with very little modification – to show that there can not be any modular translation from AnsProlog to any monotonic logic.

6.6.2 Classical logic and AnsProlog*

In Section 2.1.6 we showed how propositional theories can be mapped to AnsProlog so that there is a one-to-one correspondence between the models of a propositional theory and the answer sets of the corresponding AnsProlog program. In Section 2.1.7 we discussed how to express the entailment of closed first-order queries from AnsProlog* programs. In this section our goal is slightly different. We would like to discuss translation of classical theories – first-order and beyond, to AnsProlog* theories and vice-versa so that there is a one-to-one correspondence between the models of a classical theory and the answer sets of the corresponding AnsProlog* program.

In this quest the translations presented in the expressibility and complexity results (in particular, in Theorems 6.4.2, 6.4.4, 6.4.6, and 6.4.7) are adequate if we only consider Herbrand models.

Example 111 Consider the first-order theory T given by $\exists X.p(X)$.

This can be translated to the following AnsProlog program Π with a one-to-one correspondence between the Herbrand models of T and the answer sets of Π.

$p(X) \leftarrow$ **not** $n\_p(X)$.
$n\_p(X) \leftarrow$ **not** $p(X)$.
$good\_model \leftarrow p(X)$.
$\leftarrow$ **not** $good\_model$. □

Since the semantics of AnsProlog* is based on the Herbrand universe and answer sets of AnsProlog programs are Herbrand Interpretations there is often a mismatch between classical theories and AnsProlog* programs if we do not restrict ourselves to Herbrand models. The following example illustrates this.

Example 112 Consider the first-order theory T_1 given by $ontable(a) \wedge ontable(b)$ and the AnsProlog* program Π_1 obtained by translating T_1 using the method suggested in the proof of Theorem 6.4.6 given as follows:

$ontable(X) \leftarrow$ **not** $n\_ontable(X)$.
$n\_ontable(X) \leftarrow$ **not** $ontable(X)$.
$good\_model \leftarrow ontable(a), ontable(b)$.
$\leftarrow$ **not** $good\_model$.

Let us consider the query Q given by $\forall X.ontable(X)$. Since entailment in AnsProlog* is defined with respect to Herbrand models only we have[3] $\Pi_1 \models \forall X.ontable(X)$. But since the entailment with respect to first-order theories is not limited to Herbrand models we have $T_1 \not\models \forall X.ontable(X)$. For that reason while using resolution with respect to T_1 and Q, the clauses obtained from $T_1 \cup \{\neg Q\}$ is the set $\{ontable(a), ontable(b), \neg ontable(c)\}$ – where c is a skolem constant, which does not lead to a contradiction. □

[3] This is no longer true if we have other object constants besides a and b in the language of Π_1.

Since in some cases it may be preferable to allow non-Herbrand models, one way to get around it while using AnsProlog* is to judiciously introduce skolem constants, perhaps derived by transforming the query into a clausal form as done during resolution.

Exercise 21 Explain why the transformations from AnsProlog and AnsProlog^{or} to classical theories given in Theorems 6.4.2 and 6.4.4 are not rule-modular. □

Exercise 22 Define a new entailment relation $\models_{nh}$ from AnsProlog* programs, where entailment is not restricted to Herbrand models.
Hint: Define $\Pi \models_{nh} Q$, by first transforming $\neg Q$ using the standard techniques in classical logic, to a clausal form possibly including Skolem constants. Use this transformed formula and the standard AnsProlog* entailment relation '$\models$' to define '$\models_{nh}$'. □

Before moving onto to the next section we would like to point out that the method suggested in the proof of Theorem 6.4.6 is one of the better ways to encode classical logic theories in AnsProlog*. Intuitively this method consists of enumerating each predicates in the language and then representing the classical theory as a constraint so as to eliminate any enumeration that does not satisfy the given classical theory. The answer sets of the resulting program all satisfy the classical theory, and there is an answer set corresponding to each Herbrand model of the classical theory as they are not eliminated by the constraints.

6.6.3 Circumscription and Stratified AnsProlog

Circumscription was proposed as a non-monotonic reasoning methodology by McCarthy [McC80]. True to its literal meaning, the idea behind circumscription is to circumscribe one or more predicates in a first-order theory. For example, if a first order theory T consists of the formula $ontable(a) \wedge ontable(b)$, then circumscribing the predicate $ontable$ in T would limit the extent of $ontable$ so that it is true only for those constants for which it needs to be true. In this case the circumscription of $ontable$ in T, denoted by $Circ(T; ontable)$, is equivalent to the formula $\forall X.ontable(X) \Longleftrightarrow (X = a) \vee (X = b)$. In general, circumscription of a predicate p in a theory A containing p, denoted by $Circ(A; p)$, is given by the second order sentence $A(p) \wedge \neg\exists P(A(P) \wedge P < p)$, where $P < p$ means that the extent of P is a strict subset of the extent of p. This can be expressed in classical logic as $(\forall X.P(X) \supset p(X)) \wedge \neg(\forall X.P(X) \equiv p(X))$.

The basic definition of circumscription, as in $Circ(A; p)$, minimizes the extent of p with the stipulation that the the interpretation of other predicates, constants, and functions, remain unchanged. Often we are willing to vary the interpretation of some of the other predicates, constants, and functions, in order to make the extent of p smaller. In that case we have the more general notion $Circ(A; p; z_1, \ldots, z_n)$, where we minimize p while varying $z_1, \ldots, z_n$. $Circ(A; p; z_1, \ldots, z_n)$ is then given by the second order sentence $A(p, z_1, \ldots, z_n) \wedge \neg\exists P, Z_1, \ldots, Z_n(A(P, Z_1, \ldots, Z_n) \wedge P < p)$.

A model theoretic characterization of $Circ(A; p; z_1, \ldots, z_n)$ is given by defining an ordering $\leq^{p;z}$ between structures, where a structure M is determined by its universe $|M|$ and by the interpretations $M[[c]]$ of all function and predicate constants c in the language. $M_1 \leq^{p;z} M_2$ is then said to hold if (i) $|M_1| = |M_2|$, (ii) $M_1[[c]] = M_2[[c]]$ for every constant c which is different from p and not in z, and (iii) $M_1[[p]] \subseteq M_2[[p]]$.

Proposition 91 *[Lif85b]* A structure M is a model of $Circ(A; p; z)$ iff M is minimal relative to $\leq^{p;z}$. □

The next generalization of circumscription is to minimize a set of predicates in a theory. We may then have priorities between these predicates. In the absence of priorities the second order definition of $Circ(A; p; z_1, \ldots, z_n)$ remains same except that p now represents a tuple of predicates $p_1, \ldots, p_m$. In that case $p < P$, where P is a tuple of predicates $P_1, \ldots, P_m$, is defined as $p \leq P \wedge \neg(p = P)$ where $p \leq P$ stands for $p_1 \leq P_1 \wedge \ldots \wedge p_m \leq P_m$ and $p = P$ stands for $p_1 = P_1 \wedge \ldots \wedge p_m = P_m$. The notation $Circ(A; p_1, \ldots, p_m; z_1, \ldots, z_n)$ then denotes the parallel circumscription of $p_1, \ldots, p_m$ in theory A while varying $z_1, \ldots, z_n$. If $n = 0$, we simply write it as $Circ(A; p_1, \ldots, p_m)$.

Theorem 6.6.1 Let Π be an AnsProlog$^{-\mathbf{not}}$ program and A_Π be the first order theory obtained from Π by replacing $\leftarrow$ by the classical connective $\Leftarrow$. M is an answer set of Π iff M is a Herbrand model of $Circ(A_\Pi; p_1, \ldots, p_m)$ where $p_1, \ldots, p_m$ are all the predicates in Π. □

The above theorem follows directly from Definition 2. It illustrates that as in circumscription, the basic characterization of AnsProlog$^{-\mathbf{not}}$ programs is based on minimality, whereby minimal (with respect to subset ordering) Herbrand models are selected. Selecting minimal models correspond to parallel circumscription of all the predicates. In case of AnsProlog programs, because of the **not** operator, simply selecting minimal models is not enough. This was illustrated in Example 12. But for stratified AnsProlog programs, a more restricted notion of minimality, referred to as perfect models in Section 3.3.4 is adequate. This notion of minimality corresponds to a form of circumscription referred to as prioritized circumscription.

In prioritized circumscription, there is a priority between the predicates that are to be circumscribed. Let us assume that the tuple of predicates p can be partitioned into smaller parts $p^1, \ldots, p^k$. Our goal is to circumscribe p, but with the predicates in p^1 being circumscribed at a higher priority than the the predicates in p^2 and so on. This is denoted by $Circ(A; p^1 > \ldots > p^k; z_1, \ldots, z_n)$, and its definition given by $A(p, z_1, \ldots, z_n) \wedge \neg\exists P, Z_1, \ldots, Z_n (A(P, Z_1, \ldots, Z_n) \wedge P \prec p)$ is almost similar as before, except the meaning of $\prec$. If q is a tuple of predicates of the same kind as p, and q is partitioned into smaller parts $q^1, \ldots, q^k$, then $p \preceq q$ denotes:

$$\bigwedge_{i=1}^{k} (\bigwedge_{j=1}^{i-1} p^j = q^j \supset p^i \leq q^i)$$

and $p \prec q$ denotes $p \preceq q \wedge \neg(p = q)$. We can now relate answer sets of stratified AnsProlog programs with prioritized circumscription of the transformation of the AnsProlog program into a first-order theory, where the priorities are based on the stratification. More formally,

Theorem 6.6.2 Let Π be a stratified AnsProlog program with the stratification $\pi_1, \ldots, \pi_k$. Let A_Π be the first order theory obtained from Π by replacing **not** by $\neg$ and $\leftarrow$ by $\Leftarrow$. M is an answer set of Π iff M is a Herbrand model of $Circ(A_\Pi; \pi^1 > \ldots > \pi^k)$. □

There are relatively fewer results that related programs beyond stratified AnsProlog with circumscription. In [Prz89c] Przymusinski considers AnsProlog programs in general and relates the well-founded semantics with a 3-valued notion of circumscription.

In the above results we discussed transforming AnsProlog programs to a circumscriptive formalism. In regards to the opposite, it must be note that the second order formulation of circumscription in general does not fall within the general form of the Π^1_1 complexity class. In particular, it was shown in [Sch87] that all Δ^1_2 sets of natural numbers are definable by means of circumscription. Hence, circumscription can not be in general expressed in AnsProlog*. Nevertheless, several special

cases have been identified in the literature [Lif94] where the circumscribed theory is equivalent to a first-order formalism. Those theories can then be expressed in AnsProlog*. Similarly, when the theory to be circumscribed is of the form A_{Π} in the Theorems 6.6.1 and 6.6.2, then of course the results in Theorems 6.6.1 and 6.6.2 give the relation between such a circumscriptive theory and the corresponding AnsProlog program. It remains open to identify additional special cases where a circumscribed theory is equivalent to an AnsProlog program obtained by translating the former.

6.6.4 Autoepistemic Logic and AnsProlog*

Autoepistemic logic was proposed by Moore in [Moo85] to express non-monotonic reasoning through introspection. Although syntactically a theory in auto-epistemic logic is also a theory in (modal) nonmonotonic logics proposed by McDermott and Doyle in [MD80, McD82], Moore's autoepistemic logic avoids several pitfalls of the logics in [MD80, McD82]. Nevertheless, the ideas of McDermott and Doyle [MD80, McD82] were revived later and it was shown that the non-monotonic version of certain modal logics also avoid the pitfall of the original logics proposed in [MD80, McD82]. This is detailed in the book [MT89]. In this section we focus on the relation between autoepistemic logic and AnsProlog* as this relationship is well-studied compared to the other non-monotonic modal logics.

To motivate autoepistemic logic and differentiate it from nonmonotonic logics based on default reasoning Moore's wrote in [Moo85]:

> *Consider my reason for believing that I do not have an older brother. It is surely not that one of my parents casually remarked, "You know, you don't have any older brothers", nor have I pieced it together by carefully sifting other evidence. I simply believe that if I did have an older brother I would know about it; therefore, since I don't know of any older brothers, I must not have any.*

The language $\mathcal{L}_B$ of an auto-epistemic logic is defined – over a set of propositions S, as the least set $\mathcal{U}$ of strings such that:

1. $S \subset \mathcal{U}$,
2. if $\varphi \in \mathcal{U}$ then $\neg\varphi \in \mathcal{U}$,
3. if $\varphi_1, \varphi_2 \in \mathcal{U}$ then $(\varphi_1 \vee \varphi_2) \in \mathcal{U}$, $(\varphi_1 \wedge \varphi_2) \in \mathcal{U}$, and
4. if $\varphi \in \mathcal{U}$ then $B\varphi \in \mathcal{U}$.

Elements of $\mathcal{L}_B$ are called $formulas$, and formulas of the form $B\varphi$ are referred to as *modal atoms*. Intuitively, a modal atom $B\varphi$ means that the agent believes φ. Syntactically, an autoepistemic theory is a set of formulas constructed using propositions (in S) and modal atoms of $\mathcal{L}_B$ and propositional connectives $\neg, \vee, \wedge, \supset$, and $\equiv$. Thus an autoepistemic theory is like a propositional theory except that the 'propositions' in the autoepistemic theory can be either elements of S or modal atoms.

To define the meaning of autoepistemic theories we first define autoepistemic interpretations. An autoepistemic interpretation is similar to an interpretation of a propositional theory and maps each proposition and modal atom to either $true$ or $false$. As in case of propositional logic, an autoepistemic model M of an autoepistemic theory T must make T true. But there are two additional requirements that capture the notion that the reasoner has perfect introspection capability.

1. If a formula φ evaluates to $true$ with respect to M then $B\varphi$ must be mapped to $true$ in M.
2. If a formula φ evaluates to $false$ with respect to M then $B\varphi$ must be mapped to $false$ in M.

But as shown by the following example this is still not sufficient to capture the intuitions regarding non-monotonicity.

Example 113 Consider the following autoepistemic theory which is suppose to capture the statement that if a is a bird and we do not believe that a does not fly then a flies.

$bird(a) \wedge \neg B \neg flies(a) \supset flies(a)$

The above theory has an auto-epistemic model where $bird(a)$ is mapped to $true$ and $flies(a)$ is mapped to false, which prevents us to make a conclusion solely on the basis of auto-epistemic models that a flies. □

To overcome the above lacking, autoepistemic theories (T) are characterized using the notion of *expansions* (E) which are a set of formulas such that (i) $T \subseteq E$, (ii) E incorporates perfect introspection, and (iii) all elements of E can be derived using T and the beliefs and non-beliefs with respect to E. Condition (iii) is the one that was missing in the earlier notion of autoepistemic models. We now formally define the notion of expansions.

Definition 80 For any sets T and E of autoepistemic formulas, E is said to be an expansion of T iff $E = Cn(T \cup \{B\phi : \phi \in E\} \cup \{\neg B\psi : \psi \notin E\})$, where Cn is the propositional consequence operator. □

A formula F is said to be autoepistemically entailed by T if F belongs to all expansions of T.

Example 114 Let us reconsider the following autoepistemic theory T from Example 113.

$bird(a) \wedge \neg B \neg flies(a) \supset flies(a)$

The above theory can not have an expansion E containing $bird(a)$ and $\neg flies(a)$ as there is no way to derive $\neg flies(a)$ from T and $\{B\phi : \phi \in E\} \cup \{\neg B\psi : \psi \notin E\})$. On the other hand there is an expansion E' containing $bird(a)$ and $flies(a)$. In fact it can be shown that T autoepistemically entails $flies(a)$. □

We now relate answer sets of AnsProlog programs and expansions of autoepistemic theories obtained by a particular transformation.

Theorem 6.6.3 Let Π be an AnsProlog program. Let $T_1(\Pi)$ be an autoepistemic theory obtained by translating each rule in Π of the form

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

to the autoepistemic formula

$L_1 \wedge \ldots \wedge L_m \wedge \neg B L_{m+1} \wedge \ldots \wedge \neg B L_n \supset L_0$

M is an answer set of Π iff there is an expansion E of $T_1(\Pi)$ such that $M = E \cap HB_\Pi$. □

The above theorem suggests that the negation as failure operator in AnsProlog programs can be understood as an epistemic operator. In fact historically the definition of stable models in [GL88] was inspired by this transformation, which was earlier proposed in [Gel87] to show that the perfect models of stratified logic programs can be characterized in terms of expansions of the corresponding autoepistemic theory.

Theorem 6.6.4 *[Lif93a, Che93]* Let Π be an AnsProlog^{or} program. Let $T_2(\Pi)$ be an autoepistemic theory obtained by translating each rule in Π of the form

L_0 *or* $\ldots$ *or* $L_k \leftarrow L_{k+1}, \ldots, L_m,$ **not** $L_{m+1}, \ldots,$ **not** L_n.

to an autoepistemic formula of the form

$(L_{k+1} \wedge B\, L_{k+1}) \wedge \ldots \wedge (L_m \wedge B\, L_m) \wedge \neg B\, L_{m+1} \wedge \ldots \wedge \neg B\, L_n \supset (L_0 \wedge B\, L_0) \vee \ldots \vee (B\, L_k \wedge L_k)$.

M is an answer set of Π iff there exists an expansion E of $T_2(\Pi)$ such that $M = E \cap HB_\Pi$. □

In the quest for a direct relationship between auto-epistemic logic and Reiter's default theory, Marek and Truszczynski propose a more restricted notion of expansion, which they call *iterative expansion*. We now define iterative expansions of autoepistemic theories and relate them to answer sets.

Definition 81 For a set of formula S, let $D(S) = Cn(S \cup \{B\varphi : \varphi \in S\}$.
Given a set E, and A of formulas,
$D_0^E(A) = Cn(A \cup \{\neg B\varphi : \varphi \notin E\}$.
$D_{n+1}^E(A) = D(D_n^E(A))$.
$D^E(A) = \bigcup_{0 \leq n \leq \omega} D_n^E(A)$.
E is an iterative expansion of A iff $E = D^E(A)$. □

Theorem 6.6.5 Let Π be an AnsProlog program. Let $T_3(\Pi)$ be an autoepistemic theory obtained by translating each rule in Π of the form

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

to the autoepistemic formula

$BL_1 \wedge \ldots \wedge BL_m \wedge \neg BBL_{m+1} \wedge \ldots \wedge \neg BBL_n \supset L_0$

M is an answer set of Π iff there is an iterative expansion E of $T_3(\Pi)$ such that $M = E \cap HB_\Pi$. □

6.6.5 Default logic and AnsProlog*

The most widely studied default logic was proposed by Reiter in [Rei80]. A few other default logics were proposed by Lukaszewicz [Luk84] and Brewka [Bre91]. In this section our focus will be on Reiter's default logic, which we will simply refer to as default logic, and will relate it to AnsProlog*. Reiter's default logic is similar to AnsProlog* in its use of variables as schema variables. Thus in this section we will focus on the default theories where the variables[4] have been already instantiated, resulting in a propositional default logic.

A default theory consists of two parts: a propositional theory, and a set of non-standard inference rules referred to as *defaults*. The non-monotonicity of default logic is due to the role defaults play.

[4] A predicate default logic is proposed in [Lif90]. But its relationship with AnsProlog* is not well-studied. Hence we do not discuss it here.

The normative statement "normally birds fly" can be expressed as the default "if X is a bird and it is consistent to assume that X flies then conclude that X flies". Thus if we know tweety to be a bird and have no other information about tweety, then this default would lead us to make the conclusion that tweety flies. On the other hand, if additional information about tweety being a penguin, and penguins inability to fly is added to our knowledge base then it is no longer consistent to assume that tweety flies, hence the above default can no longer be used to conclude that tweety flies. This demonstrates the non-monotonicity of reasoning with defaults.

Syntactically, given a propositional language L a *default* d is an expression of the form

$$\frac{p(d) : j(d)}{c(d)} \tag{6.6.1}$$

where $p(d)$ and $c(d)$ are propositional formulas in L and $j(d)$ is a set of propositional formulas in L. The notation $p(d)$ is called the *prerequisite* of d, $j(d)$ is called the *justification* of d and $c(d)$ is called the *consequent* or *conclusion* of d.

The normative statement "normally birds fly" can then be expressed as the set of defaults given by the schema

$$\frac{bird(X) : fly(X)}{fly(X)}$$

A default theory is a pair (D, W) where D is a set of defaults, and W is a set of propositional formulas in L. The semantics of default theories are defined in terms of sets of formulas called *extensions*. Extensions of a default theory (D, W) are defined as fixpoints of a function $\Gamma_{(D,W)}$ between sets of propositional formulas. The function $\Gamma_{(D,W)}$ associated with the default theory (D, W) is defined as follows:

Given a set E of propositional formulas $\Gamma_{(D,W)}(E)$ is the smallest set of sentences such that

(i) $W \subseteq \Gamma_{(D,W)}(E)$,

(ii) for any default of the form (6.6.1) from D, if $p(d) \in \Gamma_{(D,W)}(E)$ and $\neg j(d) \cap E = \emptyset$ then $c(d) \in \Gamma_{(D,W)}(E)$, where $\neg j(d) = \{\neg\beta \mid \beta \in j(d)\}$; and

(iii) $\Gamma_{(D,W)}(E)$ is deductively closed.

A set of propositional formulas E is said to be an *extension* of a default theory (D, W) if $\Gamma_{(D,W)}(E) = E$. We now relate answer sets of AnsProlog* programs with the extensions of several translations of these programs to default theories.

Theorem 6.6.6 Let Π be an AnsProlog$^\neg$ program. Let $(D_1(\Pi), \emptyset)$ be a default theory obtained by translating each rule in Π of the form

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

to the default

$$\frac{L_1 \wedge \ldots \wedge L_m : \bar{L}_{m+1}, \ldots, \bar{L}_n}{L_0}$$

where $\bar{L}$ denotes the literal complementary to L.

(i) A set M of literals is an answer set of Π iff $Cn(M)$ is an extension of $(D_1(\Pi), \emptyset)$.

(ii) A set of formulas E is an extension of $(D_1(\Pi), \emptyset)$ iff $E = Cn(E \cap Lit_\Pi)$ and $E \cap Lit_\Pi$ is an answer set of Π.

(iii) If E_1 and E_2 are two extensions of $(D_1(\Pi), \emptyset)$ and $E_1 \cap Lit_\Pi = E_2 \cap Lit_\Pi$, then $E_1 = E_2$. □

Thus every AnsProlog$^\neg$ program can be identified with a particular default theory, and hence AnsProlog$^\neg$ programs can be considered as a special case of a default theory. On the other hand default theories all whose defaults have justifications and consequents as literals, preconditions as conjunction of literals, and the W part as empty can be thought of as AnsProlog$^\neg$ programs. We now present some additional translations from AnsProlog programs to default theories and relate them.

Theorem 6.6.7 Let Π be an AnsProlog program. Let $(D_2(\Pi), \emptyset)$ be a default theory obtained by translating each rule in Π of the form

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

to the default

$$\frac{: \neg L_{m+1}, \ldots, \neg L_n}{L_1 \wedge \ldots \wedge L_m \Rightarrow L_0}$$

A set M of atoms is an answer set of Π iff there is an extension E of $(D_2(\Pi), \emptyset)$ such that $M = E \cap HB_\Pi$. □

Theorem 6.6.8 Let Π be an AnsProlog program. Let $(D_3(\Pi), W_3(\Pi))$ be a default theory obtained by translating each rule in Π of the form

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

to the default

$$\frac{L_1 \wedge \ldots \wedge L_m : \neg L_{m+1}, \ldots, \neg L_n}{L_0}$$

when $m \neq n$, and to the formula $L_1 \wedge \ldots \wedge L_m \Rightarrow L_0$ in $W_3(\Pi)$, when $m = n$.

(i) A set M of atoms is an answer set of Π iff there is an extension E of $(D_3(\Pi), W_3(\Pi))$ such that $M = E \cap HB_\Pi$.

(ii) A set of formulas E is an extension of $(D_3(\Pi), W_3(\Pi))$ iff $E = Cn(W_3(\Pi) \cup (E \cap HB_\Pi))$ and $E \cap HB_\Pi$ is an answer set of Π.

(iii) If E_1 and E_2 are extensions of $(D_3(\Pi), W_3(\Pi))$ and $(E_1 \cap HB_\Pi) = (E_2 \cap HB_\Pi)$, then $E_1 = E_2$. □

Somewhat surprisingly, the above results are not easily generalized to AnsProlog^{or}. One of the problems in finding a natural translation from AnsProlog^{or} programs to default theories is related to the inability to use defaults with empty justifications in reasoning by cases: the default theory $(D, W) = (\{\frac{q:}{p}, \frac{r:}{p}\}, \{q \vee r\})$ does not have an extension containing p and therefore, does not entail p. It is easy to see that its AnsProlog^{or} counterpart entails p.

Two proposals have been made to overcome this. As pointed out in [Tur95], modifying (D, W) by adding $\{\frac{\neg q:}{\neg q}, \frac{q:}{q}, \frac{\neg r:}{\neg r}, \frac{r:}{r}\}$ to D will result in the intuitive conclusion of p.

In [GLPT91] a disjunctive default theory is proposed where disjunctions similar to the one in AnsProlog^{or} is added to default theory. In that formulation (D, W) would be instead written as $(\{\frac{q:}{p}, \frac{r:}{p}, \frac{:}{q|r}\}, \emptyset)$ and this theory would entail p.

6.6.6 Truth Maintenance Systems and AnsProlog*

In this section we will briefly discuss the relationship between AnsProlog programs and nonmonotonic truth maintenance systems (TMSs) [Doy79]. Systems of this sort, originally described by procedural (and sometimes rather complicated) means, commonly serve as inference engines of AI reasoning systems. We will follow a comparatively simple description of TMSs from [Elk90]. We will need the following terminology: a *justification* is a set of directed propositional clauses of the form $\alpha \wedge \beta \supset c$ where c is an atom, α is a conjunction of atoms and β is a conjunction of negated atoms. By an interpretation we will mean a set of atoms. The justification $\alpha \wedge \beta \supset c$ *supports* the atom c w.r.t. an interpretation M if $\alpha \wedge \beta$ is $true$ in M. A model M of a set of justifications Π is *grounded* if it can be written $M = \{c_1, \ldots, c_n\}$ such that each c_j has at least one justification $\alpha \wedge \beta \supset c_j$ that supports it whose positive antecedents α are a subset of $\{c_1, \ldots, c_{j-1}\}$. The task of a nonmonotonic TMS is to find a grounded model of a set of justifications Π.

The form of justifications suggests the obvious analogy with rules of AnsProlog programs where negated literals $\neg A$ from β are replaced by **not** A. For a nonmonotonic TMS Π let us denote the corresponding AnsProlog program by Π^*. The following theorem establishes the relationship between TMSs and AnsProlog programs:

Theorem 6.6.9 *[Elk90]* M is a grounded model of a collection of justifications Π iff it is an answer set of the program Π^*.

Similar results were obtained in [WB93], [GM90], [PC89], [RM89], and [FH89]. (The last two papers use autoepistemic logic instead of AnsProlog* programs). They led to a better understanding of the semantics of nonmonotonic truth maintenance systems, to their use in computing answer sets [Esh90] and autoepistemic expansions [JK91], for doing abductive reasoning [IS91], [RP91], and to the development of variants of TMSs based on other semantics of logic programs. A good description of one such system, based on the well-founded semantics, together with the proof of its tractability can be found in [Wit91].

6.6.7 Description logics and AnsProlog*

6.6.8 Answer set entailment as a non-monotonic entailment relation

In [KLM90] Kraus, Lehman and Magidor propose several intuitive properties that they suggest should be satisfied by non-monotonic entailment relations. Dix [Dix95a, Dix95b] compares several semantics of logic programming with respect to these properties, which he refers to as *structural properties*. In this section we list how the semantics of AnsProlog and the well-founded semantics fare with respect to these structural properties.

Kraus, Lehman and Magidor considered an entailment relation " $\mid\sim$ " *between single propositional*

formulae (α, β and γ) and defined the following structural properties.

Right Weakening	$\models \alpha \rightarrow \beta$	and	$\gamma \mid\sim \alpha$	imply	$\gamma \mid\sim \beta$
Reflexivity					$\alpha \mid\sim \alpha$
And	$\alpha \mid\sim \beta$	and	$\alpha \mid\sim \gamma$	imply	$\alpha \mid\sim \beta \wedge \gamma$
Or	$\alpha \mid\sim \gamma$	and	$\beta \mid\sim \gamma$	imply	$\alpha \vee \beta \mid\sim \gamma$
Left Logical Equivalence	$\models \alpha \leftrightarrow \beta$	and	$\alpha \mid\sim \gamma$	imply	$\beta \mid\sim \gamma$
Cautious Monotony	$\alpha \mid\sim \beta$	and	$\alpha \mid\sim \gamma$	imply	$\alpha \wedge \beta \mid\sim \gamma$
Cut	$\alpha \mid\sim \beta$	and	$\alpha \wedge \beta \mid\sim \gamma$	imply	$\alpha \mid\sim \gamma$
Rationality	not $\alpha \mid\sim \neg\beta$	and	$\alpha \mid\sim \gamma$	imply	$\alpha \wedge \beta \mid\sim \gamma$
Negation Rationality	$\alpha \mid\sim \beta$	implies	$\alpha \wedge \gamma \mid\sim \beta$	or	$\alpha \wedge \neg\gamma \mid\sim \beta$
Disjunctive Rationality	$\alpha \vee \beta \mid\sim \gamma$	implies	$\alpha \mid\sim \gamma$	or	$\beta \mid\sim \gamma$

The above properties were defined for single formulas, but could be easily extended to a relation between *finite* sets of formulae using the connective $\wedge$. For infinite sets of formulas, Makinson [Mak94] uses a *closure-operation Cn* to define several of the above mentioned properties and another property called *Cumulativity* which is defined as follows:

Cumulativity: $\Phi \subseteq \Psi \subseteq Cn(\Phi)$ implies $Cn(\Phi) = Cn(\Psi)$.

The following lemma relates Cn and $\mid\sim$ when dealing with finite sets.

Lemma 6.6.10 Relating Cn and $\mid\sim$ for finite sets
If And holds Cumulativity is equivalent to Cautious monotony and Cut. □

The above structural properties are not directly applicable to our AnsProlog* as its entailment relation is between AnsProlog* programs and queries. To make it consistent with the notations of $\mid\sim$, we adapt $\mid\sim$ to AnsProlog* and define an entailment relation.

Definition 82 Let Π be an AnsProlog or AnsProlog^{or} program. Let $U = \{u_1, \ldots, u_n\}$ be a set of positive literals and $X = \{x_1, \ldots, x_m\}$ be a set of literals. We define:

$(u_1 \wedge \ldots \wedge u_n) \mid\sim_\Pi (x_1 \wedge \ldots \wedge x_m)$

iff $\Pi \cup \{u_1, \ldots, u_n\} \models x_1 \wedge \ldots \wedge x_m$ □

This adaptation of $\mid\sim$ to $\mid\sim_\Pi$ results in one major difference between $\mid\sim$ to $\mid\sim_\Pi$. While $\mid\sim$ was a relation between propositional formulas, $\mid\sim_\Pi$ is a relation between conjunction of atoms and conjunction of literals. Because of this AnsProlog programs trivially satisfy the properties: Right weakening, Reflexivity, And, and Left logical equivalence. The only properties that remains to be considered are Cumulativity (i.e., Cautious monotony and Cut, since And is satisfied) and Rationality, which is considered as a strengthened form of Cautious monotony.

Theorem 6.6.11 For stratified AnsProlog programs, answer set semantics is *Cumulative* and *Rational*. □

Theorem 6.6.12 For AnsProlog programs answer-set semantics satisfies Cut but not Cautious monotony. Hence it is not Cumulative. □

Example 115 Consider the following program Π_2:

$a \leftarrow \textbf{not } b$
$b \leftarrow \textbf{not } a$
$p \leftarrow \textbf{not } p$
$p \leftarrow a$

The above program has the unique answer set $\{p, a\}$. But the program $\Pi_1 \cup \{p\}$ has two answer sets $\{p, a\}$ and $\{p, b\}$, and thus a is not entailed by $\Pi_1 \cup \{p\}$. □

Theorem 6.6.13 For AnsProlog programs, well-founded semantics is *cumulative* and *rational*. □

In the context of proof theory for their logical system **P** in [KLM90], Kraus, Lehman and Magidor describe another property defined below which they call *Loop*.

$$\frac{\alpha_0 \mid\sim \alpha_1,\ \alpha_1 \mid\sim \alpha_2,\ \ldots,\ \alpha_{k-1} \mid\sim \alpha_k,\ \alpha_k \mid\sim \alpha_0}{\alpha_0 \mid\sim \alpha_k} \qquad (Loop)$$

The following example shows that well-founded semantics does not satisfy Loop.

Example 116 Consider the following program Π:

$a_1 \leftarrow a_0, \textbf{not } a_2, \textbf{not } a_3.$
$a_2 \leftarrow a_1, \textbf{not } a_3, \textbf{not } a_0.$
$a_3 \leftarrow a_2, \textbf{not } a_0, \textbf{not } a_1.$
$a_0 \leftarrow a_3, \textbf{not } a_2, \textbf{not } a_1.$

$WFS(\Pi \cup \{a_0\}) = \{\neg a_2, \neg a_3, a_1\}$
$WFS(\Pi \cup \{a_1\}) = \{\neg a_0, \neg a_3, a_2\}$
$WFS(\Pi \cup \{a_2\}) = \{\neg a_1, \neg a_0, a_3\}$
$WFS(\Pi \cup \{a_3\}) = \{\neg a_2, \neg a_1, a_0\}$ □

Finally, the following theorem is due to Schlipf.

Theorem 6.6.14 *[Sch92]* For an AnsProlog program Π if a is true in the well-founded semantics of Π then the answers sets of Π and $\Pi \cup \{a\}$ coincide. □

6.7 Notes and references

The complexity and expressibility results in this chapter are based on the excellent survey article [DEGV99, DEGV97] which surveys the complexity and expressibility results of various different logic programming semantics. The book [Pap94] is a very good resource on complexity classes and the book [AHV95] has several illuminating chapters on expressibility of database query languages.

The P-completeness result about propositional AnsDatalog$^{-\textbf{not}}$ (Theorem 6.2.1) is implicit in [JL77, Var82, Imm86]. Moreover it is shown in [DG84, IM87] that using appropriate data structures, the answer set of a propositional AnsDatalog$^{-\textbf{not}}$ program can be obtained in linear time with respect to the size of the program. The P-data-completeness of AnsDatalog$^{-\textbf{not}}$ (Theorem 6.2.2) and EXPTIME-program-completeness (Theorem 6.2.3) is also implicit in [Var82, Imm86]. The complexity of existence of answer sets of AnsDatalog programs (Theorem 6.2.4) was shown in [MT91, BF91]. The coNP-data-completeness and coNEXPTIME-program-completeness of AnsDatalog (Theorem 6.2.5 and 6.2.6) was shown in [MT91, Sch95b, KP88, KP91]. The program and data

complexity result about Stratified AnsDatalog (Theorem 6.2.7 and 6.2.8) is implicit in [ABW88]. Similar results about the well-founded semantics of AnsDatalog (Theorem 6.2.9 and 6.2.10) are implicit in [VGRS91]. The results about complexity of entailment in AnsDatalog$^{or,\neg\mathbf{not}}$ programs (Theorem 6.2.11 and 6.2.12) are from [EG93a, EG93b, EG93c, EGM94]. The results about complexity of entailment in AnsDatalog^{or} programs (Theorem 6.2.13 and 6.2.14) are from [Got94, EG95, EGM94, EGM97].

The results about the general form of complexity classes in [Fag74, KP88, EGM94] form the basis of proving expressibility results of various subclasses of AnsProlog*. Expressibility of AnsDatalog (Theorem 6.3.1) is from [Sch95b]. Expressibility of locally stratified programs was first presented in [BMS95]. Expressibility of AnsDatalog^{or} (Theorem 6.3.2) is from [EGM94, EGM97]. Related expressibility results are presented in [GS97b, KV95, Sch90, Sac97].

The r.e.-completeness of AnsProlog$^{\neg\mathbf{not}}$ (Theorem 6.4.1) is from [AN78, Tar77]. The Π^1_1-completeness of AnsProlog (Theorem 6.4.2) is from [Sch95b, MNR94], and the same for the well-founded semantics is from [Sch95b]. The Π^1_1-completeness of AnsProlog^{or} (Theorem 6.4.4) is from [EG97]. The expressibility results for AnsProlog and AnsProlog^{or} (Theorems 6.4.6 and 6.4.7) are from [Sch95b, EG97]. Additional complexity and decidability results are presented in [Sch95a].

Section 6.5 on compactness and compilability are based on [CDS96, CDS94].

The result about the lack of a modular translation from AnsProlog to propositional logic is from [Nie99]. The issue of capturing non-Herbrand models in AnsProlog* was first Raised by Przymusinski in a technical report in 1987 and was later elaborated on by Ross in the appendix of [Ros89a]. Ross proposed a solution to this. Reiter in a personal conversations also raised these issues. We discuss this issue further in Section 9.5.

Even though some affinity between logic programs and nonmonotonic logics was recognized rather early [Rei82, Lif85a], the intensive work in this direction started in 1987 after the discovery of model theoretic semantics for stratified logic programs [Apt89]. Almost immediately after this notion was introduced, stratified logic programs were mapped into the three major nonmonotonic formalisms investigated at that time: circumscription [Lif88, Prz88a], autoepistemic logic [Gel87] and default theories [BF91, MT89]. Research in this area was stimulated by the workshop on *Foundations of Deductive Databases and Logic Programming* [Min88b] and by the workshops on *Logic Programming and Nonmonotonic Reasoning* [NMS91, PN93]. A 1993 special issue of *Journal of Logic Programming* devoted to "logic programming and nonmonotonic reasoning" includes an overview on the relations between logic programming and nonmonotonic reasoning [Min93] and an article on performing nonmonotonic reasoning with logic programming [PAA93]. Results relating logic programs with different semantics to various modifications of original nonmonotonic theories can be found in [PAA92a, Prz89c] among others.

The article [Lif94] is an excellent survey on circumscription and presents many results on special instances where a circumscribed theory can be equivalently expressed in first-order logic. The papers [GPP89, GL89] present additional relationships between circumscription and AnsProlog*.

The book [MT93] gives a comprehensive exposition of default logic and auto-epistemic logic and has its Chapter 12 discusses relation between these two logics, other nonmonotonic modal logics and AnsProlog*. Many of our results relating default logic, auto-epistemic logic and AnsProlog* are from [MS89]. The lack of a modular translation from propositional default logic to AnsProlog* – even though they express the same complexity class – was first pointed out by Gottlob in an invited talk in KR 96. Gottlob elaborated on this in a recent mail as follows: Since there is

a modular translation from AnsProlog to autoepistemic logic and it was shown in [Got95] that there does not exist a modular translation from default logic to autoepistemic logic, hence there can not be a modular translation from propositional default logic to AnsProlog*. The complexity and expressibility of default logics is discussed in [CEG94, CEG97]. The survey article [CS92] and Chapter 13 of [MT93] has a compilation of complexity results about various non-monotonic formalisms.

Among the impact of logic programming on the development of nonmonotonic logic were identification of special classes of theories such as stratified autoepistemic theories and their variants, with comparatively good computational and other properties, and development of new versions of basic formalisms, such as "default theories" [LY91, PP92], disjunctive defaults [GLPT91], reflexive autoepistemic logic [Sch91], introspective circumscription [Lif89], and MBNF [Lif91, LS90], to mention only a few.

Dix studied the structural properties of various logic programming semantics in a series of papers [Dix91, Dix92b, Dix92a, Dix95a, Dix95b]. Our discussion in Section 6.6.8 is based on the papers [Dix95a, Dix95b].

Chapter 7

Answer set computing algorithms

In this chapter we discuss four algorithms for computing answer sets of ground AnsProlog* programs. The first three algorithms compute answer sets of ground AnsProlog programs while the fourth algorithm computes answer sets of ground AnsProlog^{or} programs. In Chapter 8 we will discuss several implemented systems that compute answer sets and use algorithms from this chapter.

Recall that for ground AnsProlog and AnsProlog^{or} programs Π answer sets are finite sets of atoms and are subsets of HB_Π. In other words answer sets are particular *(Herbrand) interpretations* of Π which satisfy additional properties. Intuitively, for an answer set A of Π all atoms in A are viewed as $true$ with respect to A, and all atoms not in A are viewed as $false$ with respect to A. Most answer set computing algorithms – including the algorithms in this chapter – search in the space of partial interpretations, where in a partial interpretation some atoms have the truth value $true$, some others have the truth value $false$ and the remaining are considered to be neither $true$ nor $false$. In the first three algorithms in this chapter the partial interpretations are 3-valued and are referred to as *3-valued interpretations* , while in the fourth algorithm the partial interpretation that is used is 4-valued. In 3-valued interpretations the atoms which are neither $true$ not $false$ have the truth value *unknown*.

The common feature of the algorithms in this chapter are that given a partial interpretation they first try to *extend* them using some form or derivative of Propositions 20 and 22 from Chapter 3. If that fails they then arbitrarily select a naf-literal or use a heuristics to decide on a naf-literal to add to the current partial interpretation and then extend the resulting partial interpretation. These attempts to extend continues until an answer set is obtained or a contradiction is obtained.

We now give some formal definitions and notations that we will use in the rest of the chapter. A 3-valued interpretation I is often represented as a pair $\langle T_I, F_I \rangle$, with $T_I \cap F_I = \emptyset$, where T_I is the set of atoms that have the truth value $true$ and F_I is the set of atoms that have the truth value $false$. The atoms that are neither in T_I nor in F_I are said to have the truth value *unknown*. Sometimes a 3-valued interpretation is represented as a set S of naf-literals such that S does not contain both a and **not** a for any atom a. The two representations have a 1-1 correspondence. A 3-valued interpretation represented as a set S of naf-literals can be represented as the pair $\langle T_S, F_S \rangle$, where $T_S = \{a \ : \ a \in HB_\Pi \cap S\}$ and $F_S = \{a \ : \ \textbf{not } a \in S \text{ and } a \in HB_\Pi\}$. Similarly, a 3-valued interpretation I represented as a pair $\langle T_I, F_I \rangle$, can be represented by the set of naf-literals given by $T_I \cup \{\textbf{not } a \ : \ a \in F_I\}$. A 3-valued interpretation $\langle T_I, F_I \rangle$ is said to be 2-valued if $T_I \cup F_I = HB_\Pi$. In that case we say that the 3-valued interpretation $\langle T_I, F_I \rangle$ is equivalent to the interpretation T_I, and we often replace one by the other. A 3-valued interpretation $\langle T_I, F_I \rangle$ is said to *extend* (or *expand*) another 3-valued interpretation $\langle T'_I, F'_I \rangle$ if $T'_I \subseteq T_I$ and $F'_I \subseteq F_I$. In that case we also say

that $\langle T'_I, F'_I \rangle$ *agrees with* $\langle T_I, F_I \rangle$.

Example 117 Let $HB_\Pi = \{a, b, c, d, e, f\}$. Let I be a 3-valued interpretation given by $\langle \{a, b\}, \{c, d\} \rangle$. I can be alternatively represented as $\{a, b, \textbf{not } c, \textbf{not } d\}$.

Similarly a 3-valued interpretation represented as $S = \{a, b, e, \textbf{not } c, \textbf{not } d\}$ can be alternatively represented as $\langle \{a, b, e\}, \{c, d\} \rangle$.

The 3-valued interpretation $\langle \{a, b, e\}, \{c, d\} \rangle$ extends I but the 3-valued interpretation $\langle \{b, e\}, \{c, d\} \rangle$ does not.

The 3-valued interpretation $S' = \{a, b, e, \textbf{not } c, \textbf{not } d, \textbf{not } f\}$ is equivalent to the interpretation $\{a, b, e\}$.

I and S agree with S'. We can also say that I and S agree with the interpretation $\{a, b, e\}$. But S does not agree with I. □

7.1 Branch and bound with WFS: wfs-bb

The wfs-bb algorithm computes answer sets in two distinct phases. It first computes the well-founded semantics of the ground program. It exploits the fact that the well-founded semantics is sound with respect to answer set semantics. This means that the 3-valued interpretation corresponding to the well-founded semantics of a program agrees with any answer set of the program. After computing the well-founded semantics it extends the corresponding 3-valued interpretation to answer sets by using branch and bound strategy together with recursive calls to the module that computes the well-founded semantics.

7.1.1 Computing the well-founded semantics

A comparatively straightforward way to compute the well-founded semantics of an AnsProlog program is to use the characterization in Section 1.3.6, where it is mentioned that the well-founded semantics of AnsProlog programs is given by $\{lfp(\Gamma^2_\Pi), gfp(\Gamma^2_\Pi)\}$. It is easy to show that Γ_Π is an anti-monotonic operator, and hence Γ^2_Π is a monotonic operator. Thus the $lfp(\Gamma^2_\Pi)$ can be computed by iteratively applying Γ^2_Π starting from the empty set; and $gfp(\Gamma^2_\Pi) = \Gamma_\Pi(lfp(\Gamma^2_\Pi))$.

The wfs-bb algorithm computes the well-founded semantics by improving on the above algorithm in two ways.

Improvement 1

It first computes a 3-valued interpretation through an iterative procedure based on an operator defined due to Fitting. The original operator of Fitting takes a program P and a three-valued interpretation $I = \langle I^+, I^- \rangle$ and extends I. In each iteration it adds atoms p to I^+ if there is a rule in P whose head is p and whose body evaluates to *true* with respect to I, and adds atoms q to I^- if for all rules in P whose head is q and their body evaluates to $false$ with respect to I. The interpretation obtained by one iteration is denoted by $one\_step(P, I)$. Starting with an I where all atoms have the truth value *unknown* the *one_step* operator is iteratively applied until a fixpoint is reached.

This operator is monotonic and continuous with respect to I and the ordering $\preceq$ defined as $\langle T, F \rangle \preceq \langle T', F' \rangle$ iff $T \subseteq T'$ and $F \subseteq F'$. Hence if we start from $I_0 = \langle \emptyset, \emptyset \rangle$ and repeatedly apply *one_step* (keeping P) constant we reach the least fixpoint. Let us refer to this as $I^P_{Fitting}$. (Often if P is clear from context we will just write $I_{Fitting}$.)

Proposition 92 Let P be a AnsDatalog program. $I^P_{Fitting}$ agrees with the well-founded semantics of P. □

The following example illustrates the direct computation of $I^P_{Fitting}$, for a program P.

Example 118 Consider the following program P:

$r_1 : a \leftarrow.$
$r_2 : b \leftarrow a.$
$r_3 : d \leftarrow \textbf{not } e.$
$r_4 : e \leftarrow \textbf{not } d, c.$
$r_5 : f \leftarrow g, a.$
$r_6 : g \leftarrow f, d.$
$r_7 : h \leftarrow \textbf{not } h, f.$
$r_8 : i \leftarrow \textbf{not } j, b.$
$r_9 : j \leftarrow \textbf{not } i, \textbf{not } c.$
$r_a : k \leftarrow \textbf{not } l, i.$
$r_b : l \leftarrow \textbf{not } k, j.$

Let us now compute $I^P_{Fitting}$.

Initially $I_0 = \langle \emptyset, \emptyset \rangle$.

$I_1 = one\_step(P, I_0) = \langle \{a\}, \{c\} \rangle$, as the body of r_1 is true with respect to I_0 and there is no rule in P with c in its head.

$I_2 = one\_step(P, I_1) = \langle \{a, b\}, \{c, e\} \rangle$, as the body of r_2 is true with respect to I_1 and r_4 is the only rule in P with e in its head and the body of r_4 is false with respect to I_1,since c is false in I_1.

$I_3 = one\_step(P, I_2) = \langle \{a, b, d\}, \{c, e\} \rangle$, as the body of r_3 is true with respect to I_2.

$I^P_{Fitting} = I_4 = I_3$. □

The wfs-bb algorithm modifies the above steps to more efficiently compute $I^P_{Fitting}$. In the modified approach, after each iteration of *one_step* the program P undergoes a transformation so as to simplify it. The simplified program denoted by $modified(P, I)$ is obtained from P by the following steps:

1. All rules in P whose head consists of an atom that has a truth value of $true$ or $false$ in I or whose body evaluates to $false$ with respect to I are removed.
2. From the remaining rules, naf-literals in the body that evaluates to $true$ with respect I are removed.

Note that none of the atoms that have truth value $true$ or $false$ in I appear in the rules in $modified(P, I)$.

The modified approach can now be described by the following iteration leading to a fixpoint.

I_0 has all atoms as *unknown*.
$P_0 = P$.
$I_{j+1} = one\_step(P_j, I_j)$
$P_{j+1} = modified(P_j, I_j)$.

Proposition 93 The fixpoint interpretation obtained above is $I^P_{Fitting}$. □

The simplified program that is obtained at the end of the fixpoint computation will be referred to as $P_{modified}$.
The following example illustrates the computation of $I^P_{Fitting}$ using the modified approach.

Example 119 Consider the program P from Example 118.

Let us now compute $I^P_{Fitting}$ using the modified algorithm.

Initially $I_0 = \langle \emptyset, \emptyset \rangle$, and $P_0 = P$.

$I_1 = one\_step(P_0, I_0) = \langle \{a\}, \{c\} \rangle$, as the body of r_1 is true with respect to I_0 and there is no rule in P with c in its head.

$P_1 = modified(P_0, I_0) = P_0 = P$.

$I_2 = one\_step(P_1, I_1) = \langle \{a, b\}, \{c, e\} \rangle$, as the body of r_2 is true with respect to I_1 and r_4 is the only rule in P with e in its head and the body of r_4 is false with respect to I_1,since c is false in I_1.

$P_2 = modified(P_1, I_1) = \{r'_2, r_3, r'_5, r_6, r_7, r_8, r'_9, r_a, r_b\}$ as given below. The rule r_1 is not in P_2 as the head of r_1 is a, and a is $true$ in I_1. The rule r_4 is not in P_2 as the body of r_4 has c, and c is $false$ in I_1.

$r'_2 : b \leftarrow$.
$r_3 : d \leftarrow$ **not** e.
$r'_5 : f \leftarrow g$.
$r_6 : g \leftarrow f, d$.
$r_7 : h \leftarrow$ **not** h, f.
$r_8 : i \leftarrow$ **not** j, b.
$r'_9 : j \leftarrow$ **not** i.
$r_a : k \leftarrow$ **not** l, i.
$r_b : l \leftarrow$ **not** k, j.

$I_3 = one\_step(P_2, I_2) = \langle \{a, b, d\}, \{c, e\} \rangle$, as the body of r_3 is true with respect to I_2.

$P_3 = modified(P_2, I_2)$ is as given below.

$r'_3 : d \leftarrow$.
$r'_5 : f \leftarrow g$.
$r_6 : g \leftarrow f, d$.
$r_7 : h \leftarrow$ **not** h, f.
$r'_8 : i \leftarrow$ **not** j.
$r'_9 : j \leftarrow$ **not** i.
$r_a : k \leftarrow$ **not** l, i.
$r_b : l \leftarrow$ **not** k, j.

$I^P_{Fitting} = I_4 = I_3$, and $P_{modified} = P_4 = modified(P_3, I_3)$ is as given below.

$r'_5 : f \leftarrow g$.
$r'_6 : g \leftarrow f$.
$r_7 : h \leftarrow$ **not** h, f.
$r'_8 : i \leftarrow$ **not** j.
$r'_9 : j \leftarrow$ **not** i.
$r_a : k \leftarrow$ **not** l, i.
$r_b : l \leftarrow$ **not** k, j. □

Improvement 2

The well-founded semantics of P can be directly obtained by computing the well-founded semantics of $P_{modified}$ and adding $I^P_{Fitting}$ to it.

Proposition 94 Let P be an AnsDatalog program. Let $P_{modified}$ and $I^P_{Fitting}$ be as defined earlier. The well-founded semantics of P is equal to the union of $I^P_{Fitting}$ and the well-founded semantics of $P_{modified}$. □

The wfs-bb algorithm further optimizes in computing the well-founded semantics of $P_{modified}$. Let us refer to $P_{modified}$ by Π. Recall that the well-founded semantics of Π can be computed by starting from $\emptyset$ and repeatedly applying Γ^2_Π to it until a fixpoint is reached. This fixpoint is the $lfp(\Gamma^2_\Pi)$ and applying Γ_Π to $lfp(\Gamma^2_\Pi)$ gives us the $gfp(\Gamma^2_\Pi)$. The well-founded semantics is then a characterization where all atoms in $lfp(\Gamma^2_\Pi)$ are $true$ and all atoms not it $gfp(\Gamma^2_\Pi)$ are $false$. In other words we have the following two sequences:

$\Gamma^0_\Pi(\emptyset) \subseteq \Gamma^2_\Pi(\emptyset) \subseteq \ldots \subseteq \Gamma^{2i}_\Pi(\emptyset) \subseteq \ldots \subseteq lfp(\Gamma^2_\Pi)$
and

$HB_\Pi \setminus \Gamma^1_\Pi(\emptyset) \subseteq HB_\Pi \setminus \Gamma^3_\Pi(\emptyset) \subseteq \ldots \subseteq HB_\Pi \setminus \Gamma^{2i+1}_\Pi(\emptyset) \subseteq \ldots \subseteq HB_\Pi \setminus gfp(\Gamma^2_\Pi)$

the first giving us the set of atoms that are $true$ in the well-founded semantics and the second giving is the set of atoms that are $false$ in the well-founded semantics.

Example 120 Let us consider $\Pi = P_{modified}$ from Example 119 and compute its well-founded semantics using the above method.

$f \leftarrow g.$
$g \leftarrow f.$
$h \leftarrow$ **not** $h, f.$
$i \leftarrow$ **not** $j.$
$j \leftarrow$ **not** $i.$
$k \leftarrow$ **not** $l, i.$
$l \leftarrow$ **not** $k, j.$

The Gelfond-Lifschitz transformation of Π with respect to $\emptyset$ is the following program which we will denote by Π_1:

$f \leftarrow g.$
$g \leftarrow f.$
$h \leftarrow f.$
$i \leftarrow.$
$j \leftarrow.$
$k \leftarrow i.$
$l \leftarrow j.$

The answer set of the above program is $\{i, j, k, l\}$. Hence, $\Gamma_\Pi(\emptyset) = \{i, j, k, l\}$. Let us denote this by I_1. Now let us compute $\Gamma^2_\Pi(\emptyset)$. To compute this we first need to compute the Gelfond-Lifschitz transformation of Π with respect to I_1. After the transformation we obtain the following program which we will denote by Π_2:

$f \leftarrow g.$
$g \leftarrow f.$
$h \leftarrow f.$

The answer set of the above program is $\emptyset$. Hence, $\Gamma^2_\Pi(\emptyset) = \emptyset$. Thus the least fixpoint of Γ^2_Π is $\emptyset$ and the greatest fixpoint of Γ^2_Π is $\{i, j, k, l\}$. Hence, the well-founded semantics of Π is $\langle \emptyset, \{f, g, h\}\rangle$. □

The slightly modified approach used in the wfs-bb algorithm to compute the well-founded semantics of $P_{modified}$ modifies the program each time the Γ operator is applied. The modification done is different in the odd and even iteration of the program. We refer to this modified method as the *pruning oscillation* method.

Initially, we have $I_0 = \emptyset$; $\Pi_0 = \Pi$; $T_0 = \emptyset$; and $F_0 = \emptyset$. This corresponds to $\Gamma^0_\Pi(\emptyset)$ of the first sequence above.

Similarly, $HB_\Pi \setminus \Gamma^1_\Pi(\emptyset)$ from the second sequence corresponds to $I_1 = \Gamma_{\Pi_0}(I_0)$; $T_1 = \emptyset$; and $F_1 = HB_\Pi \setminus I_1$. We need to define Π_1 such that $\Gamma_{\Pi_1}(I_1)$ together with T_0 will give us $\Gamma^2_\Pi(\emptyset)$

Such a Π_1 is obtained by modifying Π_0 with respect to I_1. We refer to this modification as $modify^-(\Pi_0, I_1)$, where we modify Π_0 with the assumption that all atoms not in I_1 are false.

In general $modify^-(\Pi, I)$ is obtained from Π by removing all rules in Π whose head does not belong to I, or whose body contains an atom p, such that p does not belong to I. In addition naf-literals of the form **not** q in the body of the remaining rules are removed if q does not belong to I.

Similarly, $modify^+(\Pi, I)$ x is obtained from Π by removing all rules in Π whose head belongs to I, or whose body contains an naf-literal **not** p, such that p belongs to I. In addition atoms of the form q in the body of the remaining rules are removed if q belongs to I.

We now define the rest of the sequence.

For even j, $j \geq 0$, $I_{j+2} = \Gamma_{\Pi_{j+1}}(I_{j+1})$; $\Pi_{j+2} = modify^+(\Pi_{j+1}, I_{j+2})$; $T_{j+2} = T_j \cup I_{j+2}$; and $F_{j+2} = F_j$.

For odd j, $j \geq 1$, $I_{j+2} = \Gamma_{\Pi_{j+1}}(I_{j+1})$; $\Pi_{j+2} = modify^-(\Pi_{j+1}, I_{j+2})$; $T_{j+2} = T_j$; and $F_{j+2} = F_j \cup (HB_{\Pi_{j+1}} \setminus I_{j+2})$.

Let $T^{wfs}_\Pi = T_n$ where n is the smallest integer such that $T_n = T_{n+2}$. Let $F^{wfs}_\Pi = F_{n+1}$. Let us denote Π_{n+1} by $P_{simplified}$. We now have the following proposition.

Proposition 95 Let T^{wfs}_Π and F^{wfs}_Π as defined above. $T^{wfs}_\Pi = lfp(\Gamma^2_\Pi)$ and $F^{wfs}_\Pi = gfp(\Gamma^2_\Pi)$. □

The well-founded semantics of $\Pi = P_{modified}$ as computed by $\langle T^{wfs}_\Pi, F^{wfs}_\Pi\rangle$ together with $I_{fitting}$ gives us the well-founded semantics of our original program P. We refer to it as $T_{wfs}(P)$ and $F_{wfs}(P)$. To find the answer sets of P we only need to find the answer sets of $P_{simplified}$ and add the well-founded semantics of P to it. More formally,

Proposition 96 Let P be a ground AnsProlog program, and $P_{simplified}$, $T_{wfs}(P)$ and $F_{wfs}(P)$ are as defined above. M is an answer set of P iff there is an answer set M' of $P_{simplified}$ such that $M = M' \cup T_{wfs}(P)$. □

In the following example we illustrate the computation of $P_{simplified}$, T^{wfs}_Π, F^{wfs}_Π, $T_{wfs}(P)$ and $F_{wfs}(P)$, where $\Pi = P_{modified}$.

Example 121 Let us recompute the well-founded semantics of $\Pi = P_{modified}$ from Example 119 following the pruning oscillation method and contrast it with the computation in Example 120. Recall that $\Pi_0 = \Pi$ is as follows, $I_0 = \emptyset$, $T_0 = \emptyset$, and $F_0 = \emptyset$.

$f \leftarrow g.$
$g \leftarrow f.$
$h \leftarrow \textbf{not } h, f.$
$i \leftarrow \textbf{not } j.$
$j \leftarrow \textbf{not } i.$
$k \leftarrow \textbf{not } l, i.$
$l \leftarrow \textbf{not } k, j.$

Now $I_1 = \Gamma_{\Pi_0}(I_0)$, $\Pi_1 = modify^-(\Pi_0, I_1)$, $T_1 = \emptyset$, and $F_1 = HB_\Pi \setminus I_1$. We compute them as follows:

- The Gelfond-Lifschitz transformation $\Pi_0^{I_0}$ is the following program:

  $f \leftarrow g.$
  $g \leftarrow f.$
  $h \leftarrow f.$
  $i \leftarrow.$
  $j \leftarrow.$
  $k \leftarrow i.$
  $l \leftarrow j.$

  Its unique answer set is $\{i, j, k, l\}$. Hence $I_1 = \{i, j, k, l\}$.

- $\Pi_1 = modify^-(\Pi_0, I_1)$ is the following program:

  $i \leftarrow \textbf{not } j.$
  $j \leftarrow \textbf{not } i.$
  $k \leftarrow \textbf{not } l, i.$
  $l \leftarrow \textbf{not } k, j.$

- $T_1 = \emptyset$.

- $F_1 = HB_\Pi \setminus I_1 = \{f, g, h\}$.

Now $I_2 = \Gamma_{\Pi_1}(I_1)$, $\Pi_2 = modify^+(\Pi_1, I_2)$, $T_2 = T_0 \cup I_2$, and $F_2 = F_0$. We compute them as follows:

- The Gelfond-Lifschitz transformation $\Pi_1^{I_1}$ is the empty program.

  Its unique answer set is $\{\}$. Hence $I_2 = \{\}$.

- $\Pi_2 = modify^+(\Pi_1, I_2)$ is the following program:

  $i \leftarrow \textbf{not } j.$
  $j \leftarrow \textbf{not } i.$
  $k \leftarrow \textbf{not } l, i.$
  $l \leftarrow \textbf{not } k, j.$

  which is same as Π_1.

- $T_2 = T_0 \cup I_2 = \emptyset = T_0$.

- $F_2 = F_0 = \emptyset$.

Since $T_2 = T_0$ we have $T_\Pi^{wfs} = T_0$, and $F_\Pi^{wfs} = F_1$ and $P_{simplified} = \Pi_1$. Thus the well-founded semantics of $P_{modified}$ is $\langle \emptyset, \{f, g, h\}\rangle$.

Though we do not need to, for illustration purpose we compute I_3, Π_3, T_3, and F_3 and show that indeed they are equivalent to I_1, Π_1, T_1, and F_1 respectively. Recall that, $I_3 = \Gamma_{\Pi_2}(I_2)$, $\Pi_3 = modify^-(\Pi_2, I_3)$, $T_3 = T_1$, and $F_3 = F_1 \cup (HB_\Pi \setminus I_3)$. We compute them as follows:

- The Gelfond-Lifschitz transformation $\Pi_2^{I_2}$ is the following program:

  $i \leftarrow$.
  $j \leftarrow$.
  $k \leftarrow i$.
  $l \leftarrow j$.

  Its unique answer set is $\{i, j, k, l\}$. Hence $I_3 = \{i, j, k, l\}$.

- $\Pi_3 = modify^-(\Pi_2, I_3)$ is the following program:

  $i \leftarrow$ **not** j.
  $j \leftarrow$ **not** i.
  $k \leftarrow$ **not** l, i.
  $l \leftarrow$ **not** k, j.

  which is same as Π_1.

- $T_3 = T_1 = \emptyset$.

- $F_3 = F_1 \cup (HB_\Pi \setminus I_3) = F_1$.

Recall that our goal is to compute the well-founded semantics of P denoted by $\langle T_{wfs}(P), F_{wfs}(P)\rangle$ from Examples 118 and 119, which is obtained by adding $\langle T_\Pi^{wfs}, F_\Pi^{wfs}\rangle$ to $I^P_{fitting}$. From Example 119 we have $I^P_{fitting} = \langle\{a, b, d\}, \{c, e\}\rangle$. Hence, the well-founded semantics of P, denote by $\langle T_{wfs}(P), F_{wfs}(P)\rangle$ is $\langle\{a, b, d\}, \{c, e, f, g, h\}\rangle$. □

7.1.2 The branch and bound algorithm

The answer sets of $P_{simplified}$ is obtained by a straight forward branch and bound strategy where branching is done in terms of which atom to select next. After an atom is selected two branches arise, one where the selected atom is assumed to be *true* and the other where it is assumed to be $false$. We now present the branch and bound algorithm, whose input is a ground AnsProlog program P.

In this algorithm L is a list of triples, where in each triple the first element is a program, the second element is the set of atoms assigned the truth value *true* and the third element is the set of atoms assigned the truth value $false$. The term *Ans_sets* denotes the set of answer sets. Initially it is assigned the empty set and its value is returned at the end of the algorithm. In the algorithm by $wfs(\pi)$ we denote the well-founded semantics of the fragment π computed with respect to the Herbrand Base of the initial program P. Finally, given a program Π and a naf-literal l, by $reduced(\Pi, L)$ we denote the program obtained from Π by removing any rule with the complement of L in its body and removing L from the bodies of the remaining rules.

Algorithm 3 procedure $bb(P)$
(01) $L := [(P, \emptyset, \emptyset)]$
(02) $Ans\_sets := \emptyset$
(03) **while** $L \neq \emptyset$ **do**
(04) select the first node $Q = (\pi, T, F)$ from L;
(05) remove Q from L;
(06) **if** there is no $T_0 \in Ans\_sets$ such that $T_0 \subseteq T$ **then**
(07) Select an atom A from $HB_P \setminus \{T \cup F\}$;
(08) $Q^- := (\pi^-, T^-, F^-)$ where
(09) $\pi^- := reduced(\pi, \textbf{not } A)$.
(10) $T^- := T \cup$ the set of atoms true in $wfs(\pi^-)$, and
(11) $F^- := F \cup \{A\} \cup$ the set of atoms false in $wfs(\pi^-)$.
(12) **if** T^- is not a superset of any $T_0 \in Ans\_sets$ **then**
(13) **if** Q^- is consistent (i.e., $T^- \cap F^- = \emptyset$) **then**
(14) **if** $T^- \cup F^- = HB_P$ **then**
(15) $Ans\_sets := Ans\_sets \cup T^-$
(16) **else** append Q^- to the end of list L;
(17) **endif**
(18) **endif**
(19) **endif**
(20) $Q^+ := (\pi^+, T^+, F^+)$ where
(21) $\pi^+ := reduced(\pi, A)$.
(22) $T^+ := T \cup \{A\} \cup$ the set of atoms true in $wfs(\pi^+)$, and
(23) $F^+ := F \cup$ the set of atoms false in $wfs(\pi^+)$.
(24) **if** T^+ is not a superset of any $T_0 \in Ans\_sets$ **then**
(25) **if** Q^+ (i.e., $T^+ \cap F^+ = \emptyset$) is consistent **then**
(26) **if** $T^+ \cup F^+ = HB_P$ **then**
(27) $Ans\_sets := Ans\_sets \cup T^+$
(28) **else** append Q^+ to the end of list L;
(29) **endif**
(30) **endif**
(31) **endif**
(32) **endif**
(33) **endwhile**
(34) return Ans_sets.

The above algorithm outputs the set of answer sets of programs P' if $P' = P_{simplified}$ for some ground AnsProlog program P. The algorithm does not work for arbitrary ground AnsProlog programs.

Proposition 97 Let P be a ground AnsProlog program, and $P' = P_{simplified}$ be the program obtained from P as described in this section. The set Ans_sets returned by $bb(P')$ is the set of all the answer sets of P'. □

Thus following Propositions 96 and 97 the answer sets of arbitrary ground AnsProlog programs are obtained by first computing $T_{wfs}(P)$ and $P_{simplified}$, then computing the answer sets of $P_{simplified}$ and then adding $T_{wfs}(P)$ to each of them.

In the following example we give a brief illustration of the working of the above algorithm:

Example 122 Let us apply $bb(P')$ where $P' = P_{simplified}$ from Example 121 given by

$i \leftarrow$ **not** j.
$j \leftarrow$ **not** i.
$k \leftarrow$ **not** l, i.
$l \leftarrow$ **not** k, j.

When $bb(P')$ is called in steps (01) and (02) L is initialized to $[(P', \emptyset, \emptyset)]$ and Ans_sets is initialized to $\emptyset$. In steps (04) and (05) the node $(P', \emptyset, \emptyset)$ is selected and removed from L. Suppose in step (07) the atom i is selected.

In that case π^- is the following program

$i \leftarrow$ **not** j.
$j \leftarrow$.
$l \leftarrow$ **not** k, j.

The well-founded semantics of π^- is $\langle\{j, l\}, \{i, k\}\rangle$. Hence $T^- = \{j, l\}$ and $F^- = \{i, k\}$. Since the conditions in step (12), (13) and (14) are satisfied, we have $Ans\_sets = \{\{j, l\}\}$ due to the assignment in step (15).

In step (21) π^+ is computed, which is the following program

$i \leftarrow$ **not** j.
$k \leftarrow$ **not** l.
$l \leftarrow$ **not** k, j.

The well-founded semantics of π^+ is $\langle\{i, k\}, \{j, l\}\rangle$. Hence $T^+ = \{i, k\}$ and $F^+ = \{j, l\}$. Since the conditions in step (24), (25) and (126) are satisfied, we have $Ans\_sets = \{\{j, l\}, \{i, k\}\}$ due to the assignment in step (27).

Since neither steps (17) and (27) are used L remains empty and the while loop from (3)-(33) terminates. In step (34) $bb(P')$ returns the answer sets $\{\{j, l\}, \{i, k\}\}$.

Recall that the answer sets of P are obtained by adding $T_{wfs}(P)$ to each answer sets of $P' = P_{simplified}$. Hence, the answer sets of P are:
$\{\{j, l\} \cup \{a, b, d\}, \{i, k\} \cup \{a, b, d\}\} = \{\{a, b, d, j, l\}, \{a, b, d, i, k\}\}$. □

7.1.3 Heuristic for selecting atoms in wfs-bb

The wfs-bb algorithm is modification of the branch and bound algorithm in the previous section. In wfs-bb the atoms selection step (07) of $bb(\Pi)$ is done using a heuristic function. We now describe that function.

The heuristic function is based on partitioning the set of atoms into several levels, and choosing an atom for branching from the lowest level. To partition the set of atoms a relation *depends on* between atoms is defined. An atom a is said to directly depend on an atom b if there is a rule with a in the head and either b or **not** b in the body, or if $b = a$. An atom a is said to depend on an atom b if it directly depends on b, or if there is an atom c such that a directly depends on c and c depends on b. Using the depends on relation, we define equivalent classes, where the equivalent class of an atom a, denoted by $||a||$ is the set of atoms b such that b depends on a and a depends on b. Next we define a partial ordering $\trianglelefteq$ between these equivalence classes as: $||a|| \trianglelefteq ||b||$ iff there exist an atom p in $||a||$ and an atom q in $||b||$ such that q depends on p. The equivalence classes are partitioned into layers $E_0, E_1, \ldots$ as follows: E_0 is the set of all $\trianglelefteq$ minimal equivalence classes and,

for $i \geq 0$, E_{i+1} is the set of all $\trianglelefteq$ minimal members of the set of equivalence classes obtained after removing the ones in $\bigcup_{j \leq i} E_j$. Finally, the level of an atom a is given by the i, such that $||a|| \in E_i$. The following example illustrates this.

Example 123 Let us consider $\Pi = P_{simplified}$ from Example 121 given by

$i \leftarrow$ **not** j.
$j \leftarrow$ **not** i.
$k \leftarrow$ **not** l, i.
$l \leftarrow$ **not** k, j.

With respect to the above program Π, i depends on j, j depends on i, k depends on l and i, and l depends on k and j. Based on this dependency relation we have the following equivalence classes.

$||i|| = ||j|| = \{i, j\}$, and
$||k|| = ||l|| = \{k, l\}$.

Between these two equivalence classes we have $\{i, j\} \trianglelefteq \{k, l\}$. Thus we can partition the set of equivalence classes to layers E_0 and E_1, where $E_0 = \{\{i, j\}\}$ and $E_1 = \{\{k, l\}\}$. Based on this layering the levels of i and j are 0 and the levels of k and l are 1.

Hence in the selection step (07) the heuristics described in this section will lead us to choose either i or j in the first iteration of the while loop. □

7.2 The assume-and-reduce algorithm of SLG

The assume-and-reduce algorithm (of the SLG system) to compute answer sets of ground AnsProlog programs exploits the *observation* that to find an answer set one only needs to guess the truth values of the naf-literals that appear in the program. Unlike the wfs-bb algorithm it does not compute the well-founded semantics on its way to compute the answer sets. But it does use concepts very similar to the notions $reduced(P, L)$ and *one_step*, and the notion of $modified$ from the previous section to simplify programs based on the truth value of the known atoms, and to infer the truth value of additional atoms.

7.2.1 The main observation

The earlier mentioned observation is formalized as follows:

Lemma 7.2.1 Let P be a ground AnsProlog program, I be a 2-valued interpretation, and $N(P)$ be the set of ground atoms a, such that **not** a appears in a body of a rule in P. Then I is an answer set of P iff there is a 3-valued interpretation J that agrees with I such that $N(P) = \{a : a$ is an atom and has a truth value of either $true$ or $false$ in $J\}$ and I is the unique minimal model of of the program P^J obtained from P and J by

(i) removing from P any rule that has a literal **not** B in its body with B $true$ in J, and

(ii) removing all literals of the type **not** B in the bodies of the remaining rules if B is $false$ in J. □

The following example illustrates the application of the above lemma.

Example 124 Consider the following program P from Example 118:

$r_1 : a \leftarrow.$
$r_2 : b \leftarrow a.$
$r_3 : d \leftarrow \textbf{not}\ e.$
$r_4 : e \leftarrow \textbf{not}\ d, c.$
$r_5 : f \leftarrow g, a.$
$r_6 : g \leftarrow f, d.$
$r_7 : h \leftarrow \textbf{not}\ h, f.$
$r_8 : i \leftarrow \textbf{not}\ j, b.$
$r_9 : j \leftarrow \textbf{not}\ i, \textbf{not}\ c.$
$r_a : k \leftarrow \textbf{not}\ l, i.$
$r_b : l \leftarrow \textbf{not}\ k, j.$

$N(P) = \{e, d, c, h, i, j, k, l\}$.

Recall that the answer sets of P are $I_1 = \{a, b, d, i, k\}$ and $I_2 = \{a, b, d, j, l\}$.

Let $J_1 = \langle\{d, i, k\}, \{e, c, h, j, l\}\rangle$ be a 3-valued interpretation where all the atoms in $N(P)$ are assigned a truth value of $true$ or $false$. The program P^{J_1} is as follows:

$r_1 : a \leftarrow.$
$r_2 : b \leftarrow a.$
$r_3' : d \leftarrow.$
$r_5 : f \leftarrow g, a.$
$r_6 : g \leftarrow f, d.$
$r_7' : h \leftarrow f.$
$r_8' : i \leftarrow b.$
$r_a' : k \leftarrow i.$

The unique minimal model of P^{J_1} is $\{a, b, d, i, k\}$ which is equal to I_1 as dictated by the above lemma. We can similarly verify that I_2 is an answer set of P by having $J_2 = \langle\{d, j, l\}, \{e, c, h, i, k\}\rangle$. □

7.2.2 The SLG reduction: $reduce_{slg}$

The assume-and-reduce algorithm uses a slightly different reduction than the $reduced(P, L)$ from the previous section in simplifying a program after making an assumption about an atom. We refer to this reduction as $reduced_{slg}$ and define is as follows:

Definition 83 Given a ground AnsProlog program P and a naf-literal L, the program $reduced_{slg}(P, L)$ is defined as the program obtained from P by deleting every rule in P, whose body contains the complement of L, and removing every occurrence of L in P, if L is a negative naf-literal. □

Example 125 *[CW96]* Consider the following program P:

$p \leftarrow p.$
$r \leftarrow \textbf{not}\ p.$

Let us now compute $reduced_{slg}(P, p)$. In that case we remove the second rule from P but do not remove p from the body of the first rule. Thus $reduced_{slg}(P, p) = \{p \leftarrow p.\}$.

On the other hand when we compute $reduced_{slg}(P, \textbf{not } p)$ we remove the first rule and also remove **not** p from the body of the second rule. Hence, $reduced_{slg}(P, \textbf{not } p) = \{r \leftarrow .\}$. □

Although the purpose of *reduced* in the previous section and the purpose of $reduced_{slg}$ are similar, the difference in them is due to the fact that *reduced* is applied to programs for which certain simplifications have been already done, while $reduced_{slg}$ is applied to arbitrary ground AnsProlog programs. The following lemma formalizes the impact of $reduced_{slg}$.

Lemma 7.2.2 Let P be a ground AnsProlog program, I be a 2-valued interpretation,and A be a ground atom. Then I is an answer set of P iff either A is *true* in I and I is an answer set of $reduced_{slg}(P, A)$, or A is $false$ in I and I is an answer set of $reduced_{slg}(P, \textbf{not } A)$. □

Example 126 Consider the program P from Example 125. Its answer set is $I = \{r\}$.

The atom p is false in I. We will now show that I is an answer set of $reduced_{slg}(P, \textbf{not } p)$.

This is obvious as we recall that $reduced_{slg}(P, \textbf{not } p) = \{r \leftarrow .\}$. □

7.2.3 The SLG modification

In the previous section we iterated *one_step* and $modified$ to compute $I^P_{fitting}$ and $P_{modified}$ for a given program P. A similar computation is used by the assume-and-reduce algorithm. We now define this computation:

Let P be an AnsProlog program and U be the set of atoms that appear in P. By $one\_step_{slg}(P, U)$ we denote a 3-valued interpretation I, such that all atoms from U that appear as rules with empty body in P are assigned *true* in I and all atoms a in U, for which there is not a single rule in P with a in its head, are assigned $false$ in I.

Let us considered the following sequence,where $modified$ is as defined in Section 7.1.1.

$P_0 = P$; $U_0 = U$

$I_1 = one\_step_{slg}(P_0, U_0)$, $P_1 = modified(P_0, I_1)$, $U_1 =$ the set of atoms in P_1.

For $j \geq 1$, $I_{j+1} = one\_step_{slg}(P_j, U_j)$, $P_{j+1} = modified(P_j, I_{j+1})$, $U_{j+1} =$ the set of atoms in P_{j+1}.

which stops when for some k, I_k does no assignment, and thus, $P_k = P_{k-1}$. We then say that P is *SLG-modified* to the interpretation $I' = \bigcup_{1 \leq r \leq k} I_r$, and the program $P' = P_k$.

Example 127 Consider the program P from Example 118 and 119 and let us compute the interpretation and program by doing SLG-modification on P.

Initially $P_0 = P$, and $U_0 = \{a, b, c, d, e, f, g, h, i, j, k, l\}$.

$I_1 = one\_step_{slg}(P_0, U_0) = \langle\{a\}, \{c\}\rangle$, as the body of r_1 is empty and there is no rule in P with c in its head.

$P_1 = modified(P_0, I_1) = \{r'_2, r_3, r'_5, r_6, r_7, r_8, r'_9, r_a, r_b\}$ as given below. The rule r_1 is not in P_1 as the head of r_1 is a, and a is *true* in I_1. The rule r_4 is not in P_1 as the body of r_4 has c, and c is $false$ in I_1.

$r'_2 : b \leftarrow$.
$r_3 : d \leftarrow$ **not** e.
$r'_5 : f \leftarrow g$.

$r_6 : g \leftarrow f, d.$
$r_7 : h \leftarrow \textbf{not } h, f.$
$r_8 : i \leftarrow \textbf{not } j, b.$
$r'_9 : j \leftarrow \textbf{not } i.$
$r_a : k \leftarrow \textbf{not } l, i.$
$r_b : l \leftarrow \textbf{not } k, j.$

$U_1 = \{b, d, e, f, g, h, i, j, k, l\}.$

$I_2 = one\_step_{slg}(P_1, I_1) = \langle\ b\}, \{e\}\rangle$, as the body of r_2 is empty and there is the no rule with e in its head.

$P_2 = modified(P_1, I_2)$ as given below:

$r'_3 : d \leftarrow.$
$r'_5 : f \leftarrow g.$
$r_6 : g \leftarrow f, d.$
$r_7 : h \leftarrow \textbf{not } h, f.$
$r'_8 : i \leftarrow \textbf{not } j.$
$r'_9 : j \leftarrow \textbf{not } i.$
$r_a : k \leftarrow \textbf{not } l, i.$
$r_b : l \leftarrow \textbf{not } k, j.$

$U_2 = \{d, f, g, h, i, j, k, l\}.$

$I_3 = one\_step_{slg}(P_2, I_2) = \langle\{d\}, \{\}\rangle$, as the body of r'_3 is empty.

$P_3 = modified(P_2, I_3)$ is as given below.

$r'_5 : f \leftarrow g.$
$r'_6 : g \leftarrow f.$
$r_7 : h \leftarrow \textbf{not } h, f.$
$r'_8 : i \leftarrow \textbf{not } j.$
$r'_9 : j \leftarrow \textbf{not } i.$
$r_a : k \leftarrow \textbf{not } l, i.$
$r_b : l \leftarrow \textbf{not } k, j.$

$U_3 = \{f, g, h, i, j, k, l\}.$

$I_4 = one\_step_{slg}(P_3, I_3) = \langle\{\}, \{\}\rangle$. Hence, $P_4 = P_3$ and $U_4 = U_3$.

Thus P is SLG-modified to $I_1 \cup I_2 \cup I_3 = \langle\{a, b, d\}, \{c, e\}\rangle$ and P_3. □

The following lemma states the properties of SLG-modification.

Lemma 7.2.3 Let P be an AnsProlog program that is SLG-modified to the interpretation I' and P'. Then every answer set I of P is equal to $I' \cup J$, for some answer set J of P' and vice versa. □

7.2.4 The assume-and-reduce non-deterministic algorithm

We now present the assume-and-reduce non-deterministic algorithm, which takes an AnsProlog program P as input and outputs an answer set of P, or reports failure. The non-deterministic algorithm can be easily converted to a backtracking algorithm which will result in the enumeration of all the answer sets.

Algorithm 4 procedure *assume-and-reduce(P)*
(01) Let P be SLG-modified to an interpretation I' and a program P'.
(02) $derived := I'$; $program := P'$.
(03) $assigned := \emptyset$; $assume\_set = N(program)$;
(04) **while** $assume\_set \neq \emptyset$ **do**
(05) Delete an arbitrary element A, from $assume\_set$;
(06) **if** $A \notin derived$ and **not** $A \notin derived$ **then**
(07) $choice(A, L)$ /* choice point: L can be either A or **not** A.
(08) $assumed := assumed \cup \{L\}$;
(09) Let $reduced_{slg}(program, L)$ be SLG-modified to an interpretation I^*
(09) and a program P^*;
(10) $derived := derived \cup I^*$; $program := P^*$;
(11) **if** $derived \cup assumed$ is inconsistent **then**
(12) fail (and backtrack)
(13) **endif**
(13) **endif**
(14) **endwhile**
(15) **if** $A \in assumed$ and $A \notin derived$ for some atom A **then**
(16) fail (and backtrack)
(17) **else**
(18) return the set of positive naf-literals in $assumed \cup derived$ as an answer set of P;
(19) **endif**

Theorem 7.2.4 Let P be a ground AnsProlog program. If P has at least one answer sets then *assume-and-reduce(P)* will return an answer set of P □

7.2.5 From assume-and-reduce to SLG

SLG is a sound (but not complete in general) query evaluating system that answer queries with respect to non-ground programs. It uses some of the ideas from the assume-and-reduce algorithm and two backward propagation rules described below:

1. If a ground atom A is assumed to be *true* and the program P contains exactly one clause of the form $A \leftarrow L_1, \ldots, L_m$, then every L_i $(1 \leq i \leq n)$ is assumed to be *true*.
2. If a ground atom A is assumed to be $false$, then for every rule in P of the form $A \leftarrow L$ with only one naf-literal in its body, L is assumed to be $false$.

Variants of the above backward propagation techniques are integrated into the next two answer set computation algorithms.

7.3 The smodels algorithm

The main function of the smodels algorithm is *smodels* which takes as input a ground AnsProlog program P and a set of naf-literals A and either returns *true* if there is an answer set of Π that agrees with A, or if no such answer set exists then it returns $false$. The *smodels* function calls three important functions: *expand*, *lookahead* and *heuristics*. Before describing the smodels function we first describe these three functions and functions called by these functions.

7.3.1 The function $expand(P, A)$

Given a ground AnsProlog program P and a set of naf-literals A, the goal of the function $expand(P, A)$ is to extend A as much as possible and as efficiently as possible so that all answer sets of P that agree with A also agree with $expand(P, A)$. It is defined in terms of two other functions named $Atleast(P, A)$ and $Atmost(P, A)$. The function $Atleast(P, A)$ uses Fitting's operator and two additional backward propagation rules to extend A. The function $Atmost(P, A)$ gives an upper bound – referred to as the upper-closure – on the set of atoms that can be true in any answer set that extends A. Thus A can be extended with **not** a if a is not in $Atmost(P, A)$. We now give an algorithms for $expand(P, A)$, $Atleast(P, A)$, and $Atmost(P, A)$, and describe their properties.

function $expand(P, A)$
  repeat
    $A' := A$
    $A := Atleast(P, A)$
    $A := A \cup \{\textbf{not } x \mid x \in Atoms(P) \text{ and } x \notin Atmost(P, A)\}$
  until $A = A'$
  return A.

The function $Atleast(P, A)$ is defined as the least fixpoint of the operator F_A^P defined as follows:

$F_A^P(X) = A \cup X$
  $\cup\{a \in Atoms(P) \mid$ there is a rule r in P with a in its head
    and whose body is $true$ with respect to X $\}$
  $\cup\{\textbf{not } a \mid a \in Atoms(P)$ and for all rules r in P with a in its head,
    their body is $false$ with respect to X $\}$
  $\cup\{x \mid$ there exists an $a \in B$ such that there is only one rule r in P with a in its head
    and whose body has x as a naf-literal, and the body is not $false$ with respect to X.$\}$
  $\cup\{not(x) \mid$ there exists **not** $a \in B$ such that there is a rule r in P with a in its head
    and whose body is true with respect to $X \cup \{x\}\}$.

Note that the first two set constitute Fitting's operator and the other two are similar to the backward propagation rules of Section 7.2.5. The operator $F_A^P(X)$ is monotonic and continuous and hence its least fixpoint can be computed by the standard iteration method starting from the empty set. In fact it can be computed in linear time in the size of P. We discuss this later in Section 7.3.5.

It should be noted that when A is the empty set, then during the fixpoint computation of F_A^P the backward propagation rules are not much relevant. This leads to the following proposition.

Proposition 98 Let P be a ground AnsProlog program. $Atleast(P, \emptyset)$ is equal to $I^P_{Fitting}$. □

But when A is not the empty set and includes naf-literals that are assumed (or chosen) during the computation of the answer sets then the backward propagation rules can be exploited to extend A sooner and with fewer new assumptions. The following example illustrates this.

Example 128 Consider the following program P:

r_1 : $a \leftarrow \textbf{not } b.$
r_2 : $b \leftarrow \textbf{not } a, c.$
r_3 : $b \leftarrow \textbf{not } d.$

r_4 : $c \leftarrow$ **not** d.
r_5 : $d \leftarrow$ **not** c.

The well-founded semantics and $I^P_{Fitting}$ are both $\emptyset$ for this program. Thus both the *wfs-bb* and *assume-and-reduce* algorithms will get to the choice point where a naf-literal is assumed and further reasoning is done with respect to that assumption.

Suppose a is assumed and we have $A_1 = \{a\}$. In both *wfs-bb* and *assume-and-reduce* since only forward reasoning is done, A_1 can not be extended further without making new assumptions. We will now argue that the backward propagation steps in the computation of $Atleast(P, \{a\})$ is able to make additional conclusions.

We compute the least fixpoint of $F^P_{A_1}$ as follows:
$F^P_{A_1} \uparrow 0 = F^P_{A_1}(\emptyset) = A_1 = \{a\}$.
$F^P_{A_1} \uparrow 1 = F^P_{A_1}(F^P_{A_1} \uparrow 0) = \{a\} \cup \{$**not** $b\}$.
In the last computation there is only one rule r_1 with a in its head, and its body is not $false$ with respect to $\{a\}$ and contains the naf-literal **not** b. Hence $\{$**not** $b\}$was added.
$F^P_{A_1} \uparrow 2 = F^P_{A_1}(F^P_{A_1} \uparrow 1) = F^P_{A_1} \uparrow 1$ and we have the least fixpoint.

Thus $Atleast(P, \{a\}) = \{a,$ **not** $b\}$. The above illustrates the usefulness of one of the back propagation rules. We now illustrate the usefulness of the other back propagation rule.

Let us now assume **not** a instead of a and have $A_2 = \{$**not** $a\}$. As before, since in both *wfs-bb* and *assume-and-reduce* since only forward reasoning is done, A_2 can not be extended further without making new assumptions. We will now argue that the backward propagation steps in the computation of $Atleast(P, \{$**not** $a\})$ is able to make additional conclusions.

We compute the least fixpoint of $F^P_{A_2}$ as follows:
$F^P_{A_2} \uparrow 0 = F^P_{A_2}(\emptyset) = A_2 = \{$**not** $a\}$.
$F^P_{A_2} \uparrow 1 = F^P_{A_2}(F^P_{A_2} \uparrow 0) = \{$**not** $a\} \cup \{b\}$.
In the last computation there is rule r_1 with a in its head, and its body is true with respect to $\{a\} \cup \{$**not** $b\}$. Hence $not($**not** $b) = b$ was added.
$F^P_{A_2} \uparrow 2 = F^P_{A_2}(F^P_{A_2} \uparrow 1) = F^P_{A_2} \uparrow 1$ and we have the least fixpoint.

Thus $Atleast(P, \{$**not** $a\}) = \{$**not** $a, b\}$. □

The following proposition shows that the extension done by $Atleast(P, A)$ does not lose any answer sets.

Proposition 99 Let P be a ground AnsProlog program and A be a set of naf-literals. If S is an answer set of P such that S agrees with A then S agrees with $Atleast(P, A)$. □

Given a 3-valued interpretation $A = \langle A^+, A^- \rangle$ and a ground AnsProlog program P, the function $Atmost(P, A)$ is defined as the least fixpoint of the operator G^P_A defined as follows:

$$G^P_A(X) = \{a \in atoms(P) \mid \text{there is a rule } a \leftarrow b_1, \ldots, b_m, \textbf{not } c_1, \ldots, \textbf{not } c_n \text{ such that } \{b_1, \ldots, b_m\} \subseteq X \setminus A^- \text{ and } \{c_1 \ldots c_n\} \cap A^+ = \emptyset\} \setminus A^-.$$

The operator $G^P_A(X)$ is monotonic and continuous and hence its least fixpoint can be computed by the standard iteration method starting from the empty set. In fact it can also be computed in linear time. We discuss this later in Section 7.3.5. We now give an example that illustrates the computation of $Atmost(P, A)$.

Example 129 Consider the following program P:

$r_1:\ p \leftarrow p.$
$r_2:\ q \leftarrow \textbf{not}\ p.$
$r_3:\ r \leftarrow p.$

Let $A = \emptyset$. We compute the least fixpoint of G_A^P as follows:
$G_A^P \uparrow 0 = G_A^P(\emptyset) = \{q\}$.
The presence of q is explained by the fact that the q is in the head of r_2 and $\{p\} \cap A^+ = \emptyset$. On the other hand both p and r are absent because considering rules r_1 and r_3 respectively $\{p\} \not\subseteq \emptyset$.

$G_A^P \uparrow 1 = G_A^P(G_A^P \uparrow 0) = G_A^P \uparrow 0 = \{q\}$ and we have the least fixpoint. □

The following proposition characterizes the property of $Atmost(P, A)$.

Proposition 100 Let P be a ground AnsProlog program and A be a set of naf-literals. If A is 2-valued then $Atmost(P, A)$ is same as $\Gamma_P(A)$. □

Proposition 101 Let P be a ground AnsProlog program and A be a set of naf-literals. If S is an answer set of P such that S agrees with A then $S \subseteq Atmost(P, A)$. □

We will now illustrate the computation of *expand*.

Example 130 Consider the P from Example 129 and let us compute $expand(P, A)$ where $A = \emptyset$.

In the first iteration of the computation of $expand(P, A)$ we have the following:
A is assigned the value $Atleast(P, A) = \emptyset$.
$Atmost(P, A) = \{q\}$.
Thus A gets the value $\emptyset \cup \{\textbf{not}\ p, \textbf{not}\ r\} = \{\textbf{not}\ p, \textbf{not}\ r\}$.

In the second iteration of the computation of $expand(P, A)$ we have the following:
A is assigned the value $Atleast(P, A) = \{q, \textbf{not}\ p, \textbf{not}\ r\}$.
$Atmost(P, A) = \{q, \textbf{not}\ p, \textbf{not}\ r\}$.
Thus A keeps the value $\{q, \textbf{not}\ p, \textbf{not}\ r\}$.

The next iteration has the value of A unchanged and thus we obtain $expand(P, A) = \{q, \textbf{not}\ p, \textbf{not}\ r\}$, which is also the well-founded semantics of P. □

We now formally state the properties of the function $expand(\Pi, A)$. The first property states that $expand(\Pi, A)$ does not eliminate any answer set that agreed with A. The second property states that if $A = expand(\Pi, A)$ is a set of naf-literals that contain all the atoms of the program, then either A is inconsistent or A is an answer set. The third property states that $expand(\Pi, \emptyset)$ computes the well-founded semantics of Π.

Proposition 102 Let Π be a ground AnsProlog program and A be a set of naf-literals. Then, the following hold:

1. $A \subseteq expand(\Pi, A)$; and
2. Every answer set of Π that agrees with A also agrees with $expand(\Pi, A)$. □

Proposition 103 Let Π be a ground AnsProlog program and A be a set of naf-literals such that $A = expand(\Pi, A)$. Then, the following hold:

1. if $atoms(A) = atoms(\Pi)$ and there is no answer set that agrees with A, then A is inconsistent, and

2. if A is inconsistent, then there is no answer set of Π that agrees with A. □

Proposition 104 Let Π be a ground AnsProlog program. The value of $expand(\Pi, \emptyset)$ is equal to the well-founded semantics of Π. □

7.3.2 The function $lookahead(P, A)$

As we will see later the smodels function may call the *expand* function a large number of times. For that reason *expand* is defined such that it can be computed efficiently. The smodels function has another function called *lookahead* which it less frequently and which also "expands" a set of naf-literals. Although the function *lookahead* is less efficient than *expand*, it leads to early elimination of 3-valued interpretations that do not have any extension which are answer sets. Thus the smodels function balances the more efficiently implementable but less aggressive in expanding function *expand*, with the less efficiently implementable but more aggressive in expanding function *lookahead* by calling the former more frequently and the later much less frequently.

The basic idea behind the function *lookahead* is that to extend A with respect to P, beyond computing $expand(P, A)$, we can computed $expand(P, A \cup \{x\})$ for some naf-literal x and if we find that $expand(P, A \cup \{x\})$ is inconsistent then we can extend A by adding $not(x)$. This idea is supported by the following formal result.

Lemma 7.3.1 Let Π be a ground AnsProlog program and A be a set of naf-literals. Let y be an naf-literal such that $expand(P, A \cup \{y\})$ is inconsistent. Then, every answer set of Π that agrees with A also agrees with $expand(P, A \cup \{not(y)\})$. □

We now describe the function *lookahead*.

Algorithm 5 function *lookahead(P, A)*
  repeat
    $A' := A$
    $A := lookahead\_once(P, A)$
  until $A = A'$
  return A.

function *lookahead_once(P, A)*
  $B := Atoms(P) \setminus Atoms(A)$
  $B := B \cup not(B)$
  while $B \neq \emptyset$ **do**
    Select a literal x from B
    $A' := expand(P, A \cup \{x\})$
    $B := B \setminus A'$
    if A' is inconsistent **then**
      return $expand(P, A \cup \{not(x)\})$
    endif
  endwhile
  return A.

The following example illustrates the computation of *lookahead* and its usefulness.

Example 131 Consider the following program P:

$p \leftarrow$ **not** p.

The well-founded semantics of P is $\langle \emptyset, \emptyset \rangle$, and $expand(P, \emptyset)$ will also give us the same value. Now let us compute $lookahead(P, \emptyset)$.

When $lookahead\_once(P, \emptyset)$ is called, we have B as $\{p, \textbf{not } p\}$. Let us first *select* p from B and compute $expand(P, \{p\})$. Since $atleast(P, \{p\}) = \{p, \textbf{not } p\}$ we have $expand(P, \{p\}) = \{p, \textbf{not } p\}$ which is inconsistent. Thus $lookahead\_once(P, \emptyset)$ returns $expand(P, \{\textbf{not } p\})$ which is $\{p, \textbf{not } p\}$. Another call to *lookahead_once* does not change this and hence we have $lookahead(P, \emptyset) = \{p, \textbf{not } p\}$. The same would have happened if **not** p was *selected* instead.

The importance and usefulness of *lookahead* is that it spots inconsistencies like the one above early in its computation, thus avoiding exploration of large branches each ultimately ending in inconsistencies. □

The function *lookahead* also satisfies the main property of *expansions* in that it does not lose any answer sets. More formally,

Proposition 105 Let Π be a ground AnsProlog program and A be a set of naf-literals. Then, the following hold:

1. $A \subseteq lookahead(\Pi, A)$; and
2. Every answer set of Π that agrees with A also agrees with $lookahead(\Pi, A)$. □

7.3.3 The function $heuristic(P, A)$

In our effort to extend A after we have called *expand* and *lookahead* the next step is to assume the truth of one of the remaining literals. The smodels algorithm uses a function called $heuristic(P, A)$ to make this assumption or choice. The basic idea behind the heuristic is to choose an atom which will lead to a bigger expansion. Since once an atom x is chosen their may be two paths, one where x is assumed to be $true$ and another where x is assumed to be $false$, the function $heuristic(P, A)$ takes into account the size of both $expand(P, A \cup \{x\})$ and $expand(P, A \cup \{\textbf{not } x\})$.

Thus the function $heuristic(P, A)$ first selects an atom x from $atoms(P) \setminus atoms(A)$ such that it has the maximum value of $min(|expand(P, A \cup \{x\}) \setminus A|, |expand(P, A \cup \{\textbf{not } x\}) \setminus A|)$. If there are more than one such x, then it selects the one with the greater $max(|expand(P, A \cup \{x\}) \setminus A|, |expand(P, A \cup \{\textbf{not } x\}) \setminus A|)$. Once x is selected, $heuristic(P, A)$ returns x if $|expand(P, A \cup \{x\}) \setminus A| \geq |expand(P, A \cup \{\textbf{not } x\}) \setminus A|)$, otherwise it returns **not** x.

7.3.4 The main function: $smodels(P, A)$

We are now ready to describe the main function $smodels(P, A)$ in terms of *expand*, *lookahead* and *heuristic*.

Algorithm 6 function $smodels(P, A)$
(01) $A := expand(P, A)$
(02) $A := lookahead(P, A)$

(03) **if** A is inconsistent **then** return $false$
(04) **elseif** $atoms(A) = atoms(P)$
(05) **then** return $true$
(06) **else**
(07) $x := heuristic(P, A)$
(08) **if** $smodels(P, A \cup \{x\})$ **then** return $true$
(09) **else** return $smodels(P, A \cup \{not(x)\})$
(10) **endif**
(11) **endelse**
(12) **endif**

It should be noted that if the above function returns $true$, then the set of atoms in the then value of A is an answer set of P. Normally the initial call to the above function is made by having $A = \emptyset$. To compute more than one answer set, instead of returning $true$ in step (5), the algorithm can print the answer set $atoms(A)$, and return $false$ to force the searching of additional answer sets.

We now present the a lemma that justifies steps (08) and (09) of the algorithm.

Lemma 7.3.2 Let Π be an AnsProlog program, S be an answer set of Π and A be a set of naf-literals. If S agrees with A but not with $A \cup \{x\}$, for some naf-literal x, then S agrees with $A \cup \{not(x)\}$. □

Following theorem states the correctness of the smodels function. It can be proved using the above lemma and the properties of the functions *expand*, and *lookahead*.

Theorem 7.3.3 Let Π be a ground AnsProlog program and A be a set of naf-literals. Then, there is an answer set of Π agreeing with A iff $smodels(\Pi, A)$ returns $true$. □

7.3.5 Strategies and tricks for efficient implementation

In [Sim00] a detailed efficient implementation the smodels function is described. In this section we very briefly mention some of the strategies and tricks discussed there.

During the computation of $Atleast$ and $Atmost$ a variant of the Dowling-Galier algorithm is used for linear time computation. Recall that during the computation of $Atleast(P, A)$ an atom x is added to A if there exist a rule in P whose body holds with respect to A. This is efficiently implemented by using a counter for each rule in P. Initially the counter corresponding to a rule r, referred to as $r.literal$, has the value equal to the number of naf-literals in the body of r that are not true with respect to A. Every time a new naf-literal is added to A the counter corresponding to any rule whose body contains this literal is decremented by one. When the counter corresponding to a rule has the value 0, the head of that rule is added to A. Similarly, corresponding to each rule r, there is an inactivity counter $r.inactive$, whose value is the number of naf-literals in the body of r that are $false$ with respect to A. When $r.inactive > 0$ it means that the body of r is $false$ with respect to A, and we say r is inactive; otherwise r is said to be active. For each atom a, we have a counter $a.headof$ whose value indicates the number of active rules with head a. The naf-literal **not** a is added to A when the value of $a.headof$ becomes 0. The other two sets in the definition of F_A^P are similarly accounted for by the following:

(a) When $a.headof$ becomes one, and $a \in A$, then every naf-literal in the body of the only active rule with a in its head is added to A.

(b) For an active rule r with a in its head, if **not** $a \in A$, and $r.literal$ becomes 1, then for the only naf-literal x in the body of r which is not in A, $not(x)$ is added to A.

Similar techniques are used in computing *atmost* in linear time. The computation of *atmost* is further expedited by recognizing that it is fragments (of programs) of the kind in Example 131 or in the more general case of the kind below where *atmost* is useful.

$p_1 \leftarrow p_2.$
$p_2 \leftarrow p_3.$
$\vdots$
$p_{n-1} \leftarrow p_n.$
$p_n \leftarrow p_1.$

This observation is formalized and exploited by identifying strongly connected components of the atom-dependency-graph of a program sans its negative naf-literals and localizing the computation of *atmost* with respect to the strongly connected components before moving on to the rest of the graph.

Another optimization step is to reduce the branching due to lines (07)-(09) of Algorithm 6 by taking advantage of the observations in Section 7.2.1. In that case the function *heuristics* only considers a selected subset of atoms in a program P, not the whole set $Atoms(P)$.

7.4 The dlv algorithm

The dlv algorithm is similar to the smodels algorithm of Section 7.3 with the main differences being that it is targeted towards AnsProlog$^{\perp, or}$ programs (while smodels algorithm is only for AnsProlog programs). Thus the expand function of dlv has to be able to reason about disjunctions and empty heads in the head of the rules. Another distinguishing feature of dlv is that it uses a new truth value *mbt* (or simply M), meaning *must be true*, to do backward propagation using constraints. For example, if an AnsProlog$^{\perp, or}$ program contains the constraint '$\leftarrow$ **not** p', then p must be true in any answer set of Π. In that case, dlv will initially assign the truth value M to p. The truth value of M of p can be thought of as that p has been 'observed' to be true, and we need to find 'explanations' for that observation. The dlv algorithm also uses backward propagation similar to the kind described in Section 7.2.5 to make additional conclusions. For example, if dlv has assigned p the truth value of M and there is only one rule '$p \leftarrow q, r$' with p in its head in our program, then it assigns q and r with the truth value M.

To be able to reason with this extra truth value M we use some different notations in the rest of this subsection. An interpretation I, which we will refer to as a *dlv-interpretation*, is a mapping from HB_Π to $\{T, M, U, F\}$. Thus, p has the truth value of M in an interpretation I is represented as $I(p) = M$. The truth value of naf-literals, and heads and bodies of rules with respect to an interpretation I, is expressed using the function val_I. To define val_I we use the following: **not** $T = F$, **not** $M = F$, **not** $U = U$, and **not** $F = T$; and $T > M > U > F$. Now, for an atom p, $val_I(\textbf{not } p) = \textbf{not } I(p)$; $val_I(q_1 \& \ldots \& q_n) = min_{1 \leq i \leq n}\{val_I(q_i)\}$, where q_is are naf-literals; and $val_I(p_1 \ or \ \ldots \ or \ p_m) = max_{1 \leq i \leq m}\{val_I(p_i)\}$, where p_is are atoms. For a rule r of the form '$p_1 \ or \ \ldots \ or \ p_m \leftarrow q_1 \& \ldots \& q_n$', by $val_I(head(r))$ and $val_I(body(r))$ we refer to $val_I(p_1 \ or \ \ldots \ or \ p_m)$ and $val_I(q_1 \& \ldots \& q_n)$ respectively; and by $val_I(head(r) \setminus \{p_i\})$ we refer to $val_I(p_1 \ or \ \ldots p_{i-1} \ or \ p_{i+1} \ or \ p_m)$. If r is a constraint (i.e., has an empty head) then $val_I(head(r))$ is defined as F.

For a dlv-interpretation I, by I^T (resp. I^M) we denote the set of atoms that have the truth value T (resp. M) in the interpretation I. Also, by $I \diamond x : r$ we mean the dlv-interpretation obtained by assigning x the truth value r and making no other changes to I.

We have three additional definitions. For an atom p, $support(p)$ (with respect to an interpretation I) is the set of rules in the ground program such that $val_I(head(r) \setminus \{p\}) < M$ and $val_I(body(r) > F$. Intuitively, $support(p)$ consists of the set of rules in the ground program that *may* eventually force p to become *true*. We say a rule r is *satisfied* (with respect to an interpretation I) if $val_I(head(r)) \geq val_I(body(r))$. Intuitively, a rule r is *not satisfied* (with respect to an interpretation I) if its body may become true and its head may become false eventually. Given dlv-interpretations I and I', we say I' extends I if for all atoms p, $I(p) \preceq I'(p)$, where $U \preceq F$, $U \preceq M$, $U \preceq T$, and $M \preceq T$. Intuitively, $X \preceq Y$ means that Y is more concrete in knowledge terms than X and I' extends I means that I' represents more concrete knowledge than I.

The following examples illustrates the above definitions.

Example 132 Consider an AnsProlog$^{\perp, or}$ program consisting of the following rules:

r_1 : $a \text{ or } b \leftarrow c, d, \textbf{not } e.$
$r_2 : \leftarrow d, e.$

Let I be a dlv-interpretation given as: $I(a) = M$, $I(b) = F$, $I(c) = M$, $I(d) = M$, and $I(e) = F$.

$val_I(head(r_1)) = max\{M, F\} = M$, $val_I(body(r_1)) = min\{M, M, T\} = M$, $val_I(head(r_2)) = F$, $val_I(body(r_2)) = min\{M, F\} = F$.

With respect to I, we have $support(a) = \{r_1\}$, as $val_I(head(r_1) \setminus \{a\}) = val_I(b) = F < M$, and $val_I(body(r_1)) = M > F$. But $support(b) = \{\}$.

The rule r_1 is satisfied with respect to I as $val_I(head(r_1)) = M \geq val_I(body(r_1)) = M$. The rule r_2 is also satisfied with respect to I as $val_I(head(r_2)) = F \geq val_I(body(r_2)) = F$.

Now consider I' given as: $I(a) = T$, $I(b) = F$, $I(c) = M$, $I(d) = M$, and $I(e) = F$. I' extends I, as $I(a) = M \preceq I'(a) = T$, and I and I' have the identical mapping for the other atoms. □

As mentioned earlier, the dlv algorithm is similar to the smodels algorithm. Its main function *dlv* takes a ground AnsProlog$^{\perp, or}$ program Π and a dlv-interpretation I and prints all the answer sets of Π that extend I. During its execution it calls three other functions: $expand_{dlv}$, $heuristics_{dlv}$, and $isAnswerSet$. We first describe these functions and then given the main *dlv* function.

7.4.1 The function $expand_{dlv}(P, I)$

The function $expand_{dlv}(P, I)$ is similar to the *Atleast* function of Smodels and takes into account rules with disjunctions and empty heads that are not accounted for in the *Atleast* function of Smodels. It expands the dlv-interpretation I by adding deterministic consequences of I with respect to P. It can assign F, M or T to any atom that was previously assigned to U, and can assign T or F to atoms that were previously assigned M. In the last case – when an atom assigned M is re-assigned as F – we say an inconsistency is detected.

The $expand_{dlv}(P, I)$ function iteratively extends I using the following rules. Each iteration is a monotonic operator, and is repeated until a fixpoint is reached. Each iteration step does the following:

1. Each rule in P is considered one by one and I is expanded using the following:

(a) If the head of a rule is $false$ and its body is either $true$ or mbt, then the function exits by returning $I = Lit$, which means that there is a contradiction.

(b) If there is a rule r such that $val_I(body(r))$ is either M or T, and every atom in the head of r is $false$ except for one atom p, then $val_I(p)$ is assigned the value $val_I(body(r))$.

(c) If there is a rule r such that $val_I(head(r))$ is F, and every naf-literal in the body except one (l) is either T or M, then $val_I(l)$ is made false by either assigning l the value $false$, when l is an atom, or when l is an naf-literal **not** a, then assigning a the value M.

2. Each atom is considered one by one and its support is analyzed and I is expanded based on that using the following guidelines.

(a) If an atom p with truth value T or M has no support(i.e., $support(p) = \emptyset$) then the function exits by returning $I = Lit$, which means that there is a contradiction.

(b) If an atom p with truth value U has no support(i.e., $support(p) = \emptyset$) then $I(p)$ is assigned the value F.

(c) If an atom p with truth value T or M has exactly one supporting rule (i.e., $support(p) = \{r\}$) then,

i. if an atom q, different from p is in $head(r)$, then q is assigned the value F.

ii. if an atom a is in $body(r)$ then a is assigned the value M; and

iii. if an naf-literal **not** a is in $body(r)$ then a is assigned the value F.

We now briefly relate the above steps with the steps in the *expand* function of Section 7.3.1 and the back propagation steps of Section 7.2.5. The step 1 (a) is new to this algorithm. The step 1 (b) is similar to the first constituent of F_A^P of the $Atleast(P, A)$ function of Section 7.3.1. The step 1 (c) is a generalization of the second idea in Section 7.2.5 and is similar to the fourth constituent of F_A^P. The step 2 (a) is new to this algorithm. The step 2 (b) is similar to the second constituent of F_A^P. The step 2 (c) is similar to the third constituent of F_A^P and generalizes it to account for disjunctions in the head of rules. The following proposition characterizes $expand_{dlv}$.

Proposition 106 Let P be a ground AnsProlog$^{\perp, or}$ program and I be a dlv-interpretation.

1. If $expand_{dlv}(P, I) = Lit$ then no answer set of P extends I.

2. Otherwise, $expand_{dlv}(P, I)$ extends I; and every answer set S of P that extends I also extends $expand_{dlv}(P, I)$. □

7.4.2 The function $heuristic_{dlv}(P, I)$

The function $heuristic_{dlv}(P, I)$ is different from the function *heuristic* used in the smodels algorithm. It uses a notion of *possibly-true* (PT) naf-literals and analyzes the impact of $expand_{dlv}$ if I was to be updated by making one of the PT naf-literals $true$. Based on this analysis it defines an ordering $\geq$ on PT naf-literals. It then selects one of the PT naf-literals which is maximal with respect to $\geq$.

We now define PT naf-literals, and the ordering $\geq$.

PT naf-literals: A *positive PT literal* is an atom p with truth value U or M such that there exists a ground rule r with $p \in head(r)$, the head is not true with respect to I and the body is true with

respect to I. A *negative PT literal* is a naf-literal **not** q with truth value U such that there exists a ground rule r with **not** q in the body, the head is not true with respect to I, all the atoms in the body are true with respect to I, and no negative naf-literal in the body is false with respect to I. A *PT naf-literal* is either a positive PT literal or a negative PT literal. The set of all PT naf-literals of a program P with respect to I is denoted by $PT_P(I)$.

Ordering between PT naf-literals based on their impact: The impact of a PT naf-literal p is comparatively defined based on analyzing the set of literals that become M or lose their truth value of M, when p is assumed to be true.

An atom p with truth value M *(called an mbt atom)* is said to be of level n if $|support(p)| = n$. For a PT naf-literal p, $mbt^-(p)$ is the overall number of mbt atoms which become true (using $expand_{dlv}$) by assuming p to be true; $mbt^+(p)$ is the overall number of $undefined$ atoms which become M (using $expand_{dlv}$) by assuming p to be true; $mbt_i^-(p)$ is the number of mbt atoms of level i which become true (using $expand_{dlv}$) by assuming p to be true; $mbt_i^+(p)$ is the number of $undefined$ atoms of level i which become M (using $expand_{dlv}$) by assuming p to be true; $\Delta_{mbt}(p) = mbt^-(p) - mbt^+(p)$; $\Delta_{mbt2}(p) = mbt_2^-(p) - mbt_2^+(p)$; and $\Delta_{mbt3}(p) = mbt_3^-(p) - mbt_3^+(p)$.

Given two PT naf-literals a and b:

- If $(mbt^-(a) = 0 \wedge mbt^-(b) > 0) \vee (mbt^-(a) > 0 \wedge mbt^-(b) = 0)$ then
  $a > b$ if $mbt^-(a) > mbt^-(b)$;
- Otherwise $a > b$ if
  (i) $\Delta_{mbt}(a) > \Delta_{mbt}(b)$; or
  (ii) $\Delta_{mbt2}(a) > \Delta_{mbt2}(b)$ and $\Delta_{mbt}(a) = \Delta_{mbt}(b)$; or
  (iii) $\Delta_{mbt3}(a) > \Delta_{mbt3}(b)$ and $\Delta_{mbt2}(a) = \Delta_{mbt2}(b)$ and $\Delta_{mbt}(a) = \Delta_{mbt}(b)$.
- If $a \not> b \wedge b \not> a$ then $a = b$.

One of the intuition behind the above ordering is that the mbt atoms can be thought of as constraints that need to be satisfied. Thus lesser number of mbt atoms are preferable to more number of mbt atoms as the former suggests that to find an answer set lesser number of constraints need be satisfied. Another intuition is to prefer some elimination of mbt atoms over no elimination.

Example 133 *[FLP99]* Consider the ground version of the following program which we will refer to as P.

$node(a) \leftarrow.$
$node(b) \leftarrow.$
$node(c) \leftarrow.$
$node(d) \leftarrow.$
$node(e) \leftarrow.$

$arc(a, b) \leftarrow.$
$arc(a, c) \leftarrow.$
$arc(a, d) \leftarrow.$
$arc(a, e) \leftarrow.$
$arc(b, c) \leftarrow.$
$arc(c, d) \leftarrow.$

$arc(d, b) \leftarrow$.
$arc(d, e) \leftarrow$.

$start(a) \leftarrow$.

r_1 : $inpath(X, Y)$ *or* $outpath(X, Y) \leftarrow arc(X, Y)$.

r_2 : $reached(X) \leftarrow start(X)$.
r_3 : $reached(X) \leftarrow reached(Y), inpath(Y, X)$.

r_4 : $\leftarrow inpath(X, Y), inpath(X, Y1), Y \neq Y1$.

r_5 : $\leftarrow inpath(X, Y), inpath(X1, Y), X \neq X1$.

r_6 : $\leftarrow node(X), \textbf{not}\ reached(X)$.

Let I_0 be the dlv-interpretation which assigns the truth value U to all atoms. The call $expand_{dlv}(P, I_0)$ returns a dlv-interpretation with the *node*, *arc* and *start* facts assigned T, $reached(a)$ assigned T (because of a ground instance of r_2) and $reached(b)$, $reached(c)$, $reached(d)$, and $reached(e)$ assigned to M (because of ground instances of r_6). Let us refer to this resulting dlv-interpretation as I_1 and consider the computation of $heuristic_{dlv}(P, I_1)$.

One of the PT-literals is $inpath(a, b)$. Let us assume it to be *true* and illustrate the computation of $mbt^-(inpath(a, b))$, and $mbt^+(inpath(a, b))$.

Using r_3 we can derive $reached(b)$ to be T. Using r_4 and r_5 we can derive $inpath(a, c)$, $inpath(a, d)$, $inpath(a, e)$, $inpath(d, b)$ to be F. For further analysis let us consider the following ground instantiations of r_2 and r_3:

$reached(a) \leftarrow start(a)$.

$reached(b) \leftarrow reached(a), inpath(a, b)$.
$reached(b) \leftarrow reached(d), inpath(d, b)$. (*)

$reached(c) \leftarrow reached(a), inpath(a, c)$. (*)
$reached(c) \leftarrow reached(b), inpath(b, c)$.

$reached(d) \leftarrow reached(a), inpath(a, d)$. (*)
$reached(d) \leftarrow reached(c), inpath(c, d)$.

$reached(e) \leftarrow reached(a), inpath(a, e)$. (*)
$reached(e) \leftarrow reached(d), inpath(d, e)$.

Among the above rules, the bodies of ones marked by (*) evaluate to $false$. Hence for $reached(c)$, $reached(d)$ and $reached(e)$ there is only one supporting rule. Thus using step 2 (c) (ii) of $expand_{dlv}$ we assign M to $inpath(b, c)$, $inpath(c, d)$, and $inpath(d, e)$.

Now for each of $inpath(b, c)$, $inpath(c, d)$, and $inpath(d, e)$ occur in the head of exactly one rule (the corresponding instantiation of r_1) and the body of these rules are true. Thus using 2 (c) (i) $expand_{dlv}$ of we assign F to $outpath(b, c)$, $outpath(c, d)$, and $outpath(d, e)$ as $false$ and then using 1 (b) of $expand_{dlv}$ we assign T to $inpath(b, c)$, $inpath(c, d)$, and $inpath(d, e)$.

In the subsequent iteration $reached(c)$, $reached(d)$ and $reached(e)$ are assigned T.

In summary, the change from I_1 to $expand_{dlv}(I_1 \diamond inpath(a, b) : T)$ are as follows:

$reached(b)$, $reached(c)$, $reached(d)$ and $reached(e)$: From M to T.

$inpath(a, c), inpath(a, d), inpath(a, e)$, and $inpath(d, b)$: From U to F.

$inpath(b,c), inpath(c,d)$, and $inpath(d,e)$: From U to M and then to T.

$outpath(b,c), outpath(c,d)$, and $outpath(d,e)$: From U to to F.

Thus $mbt^{-}(inpath(a,b)) = 7$ and $mbt^{+}(inpath(a,b)) = 3$. □

7.4.3 The function $isAnswerSet(P,S)$

It should be noted that a proposition similar to Proposition 103 does not hold for $expand_{dlv}$. Hence, if $expand_{dlv}$ results in a dlv-interpretation I that is 2-valued (i.e., none of the atoms are mapped to U or M) it is not guaranteed that I would be an answer set. Thus the need for the function $isAnswerSet(P,S)$.

The function $isAnswerSet(P,S)$, where S is a set of atoms, and P is an AnsProlog$^{\perp, or}$ program returns the value $true$ if S is an answer set of P; otherwise it returns the value $false$. This verification can be done by constructing the AnsProlog$^{\perp, or, \mathbf{not}}$ program P^S and checking if S is a minimal model of P^S.

7.4.4 The main dlv function

We now present the main dlv algorithm that uses $expand_{dlv}(P,I)$ and $heuristic_{dlv}(P,I)$ and prints the answer sets of an AnsProlog$^{\perp, or}$ program.

Intuitively, the function $dlv(P,I)$ takes a ground program AnsProlog^{or} program P and an interpretation I, and first computes the function $expand_{dlv}(P,I)$. If it returns Lit meaning a contradiction then the function dlv returns $false$. Otherwise it checks if $expand_{dlv}(P,I)$ encodes an answer set of P. If that is the case the answer set is printed, and the function dlv returns $false$ so as to facilitate the generation of the other answer sets. If $I' = expand_{dlv}(P,I)$ does not encode an answer set of P then $heuristic_{dlv}(P,I')$ is used to pick a naf-literal x and dlv is called with two different updates of I', one where the truth value of x is T, and another where it is F, so as to print answers sets (if any) that can be reached from both interpretations.

Algorithm 7 function $dlv(P,I)$
(01) $I := expand_{dlv}(P,I)$.
(02) **if** $I = Lit$ **then** return $false$.
(03) **elseif** $PT_P(I) = \emptyset$
(04) **then if** $I^M = \emptyset$ and $isAnswerSet(P, I^T)$ **then** print(I^T), return($false$).
(05) **else**
(06) $x := heuristic_{dlv}(P,I)$.
(07) **if** x is an atom **then**
(08) **if** $dlv(P, I \diamond x : T)$ **then** return $true$.
(09) **else** return $dlv(P, I \diamond x : F)$.
(10) **end if**
(11) **elseif** x is an naf literal **not** p **then**
(12) **if** $dlv(P, I \diamond p : M)$ **then** return $true$.
(13) **else** return $dlv(P, I \diamond p : F)$.
(14) **end if**
(15) **end if**
(16) **end if**
(17) **end if**

The function $dlv(P, I)$ is initially called with an I where all atoms are assigned the truth value U, and a P which is a ground AnsProlog^{or} program. It always returns the value $false$, but prints all the answer set (if any) of P. More formally,

Theorem 7.4.1 Let Π be a ground AnsProlog$^{\perp, or}$ program and I be a dlv-interpretation. The function $dlv(\Pi, I)$ prints all and only the answer sets of Π that extend I. □

7.4.5 Comparing dlv with Smodels

The dlv algorithm is applicable to the larger class AnsProlog$^{\perp, or}$ than the class of AnsProlog programs that is targeted by the smodels algorithm. This leads to one of the main difference between $expand_{dlv}$ and $expand$. In addition, while $expand$ uses the upper-closure idea $expand_{dlv}$ does not. Thus, while $expand$ will infer p to be $false$ given a program consisting of the only rule $p \leftarrow p$, $expand_{dlv}$ will not. The heuristics used by the two algorithms are quite different, and dlv does not have a *lookahead* similar to the one used by *smodels*.

7.5 Notes and references

The wfs-bb algorithm is from [SNV95]. The assume-and-reduce algorithm is from [CW96]. The smodels algorithm is used in the smodels system described in [NS97]. The algorithm is described in great detail in [Sim00] which also includes a lot of implementation details. The dlv algorithm is used in the dlv system described in [CEF$^+$97, EFG$^+$00]. The algorithm is described in [FLP99].

7.5.1 Other query answering approaches

In this chapter we have described algorithms to compute answer sets of ground AnsProlog* programs. There are several issues that are not addressed in this chapter. We now briefly mention them.

When dealing with AnsProlog* programs that may have function symbols the answer sets may not have a finite cardinality and hence we need a way to finitely express such answer sets. Such an attempt was made in [GMN$^+$96].

An alternative approach, which is also useful when we are only interested in computing entailment and not in computing one or all the answer sets is to develop derivation methods that compute the entailment. Several such methods have been proposed which are sound for restricted cases and complete for even more restricted cases. Some of these are the SLDNF resolution method [AD94, Str93] used in Prolog, the SLD and SLDNF calculus proposed in [Lif95, Lif96], and the integration of assume-and-reduce and SLG resolution in [CW96].

Magic set techniques [BR87] have been used in answering queries with respect to stratified AnsDatalog programs by a bottom-up approach similar to computing answer sets, but at the same time using query binding patterns in precompiling the program so as the make the bottom-up computation faster. To the best of our knowledge it has not been extended beyond stratified AnsDatalog programs.

Finally in recent years several very efficient propositional model generators [MMZ$^+$01, Zha97, MSS99] have been developed. For AnsProlog* subclasses that can be compiled to equivalent propositional theories, this suggests an alternative way to compute answer sets. Since model generation of propositional theories can be translated to solving a corresponding integer linear programming

problem, a further alternative is to use integer linear programming solvers such as CPLEX. Such an attempt is made in [BNNS94] and we discussed it earlier in Section 3.10.4.

Chapter 8

Query answering and answer set computing systems

In this chapter we discuss three query answering and answer set computing systems: smodels, dlv and Prolog. Both smodels and dlv are answer set computing systems and allow an input language with features and constructs not in AnsProlog*. While the smodels system extends AnsProlog$^{\perp}$ and AnsProlog$^{\perp,\neg}$, the dlv system extends AnsProlog$^{\perp, or}$ and AnsProlog$^{\perp, or, \neg}$. We describe the syntax and semantics of the input language of smodels and dlv and present several programs in their syntax. This chapter can be thought of as a quick introduction to programming in smodels and dlv, and not a full-fledged manual. Both smodels and dlv are evolving systems and the reader is recommended to visit their corresponding websites for their latest features.

Besides smodels and dlv we also give a brief introduction to the Prolog interpreter and its approach to answering queries with respect to AnsProlog programs. We present conditions for AnsProlog programs and queries for which the Prolog interpreter is sound and complete. We illustrate these conditions through several examples.

8.1 Smodels

The Smodels system is meant for computing the answer sets of AnsProlog$^{\perp}$ and AnsProlog$^{\perp,\neg}$ programs[1] and allows certain extensions to them. We refer to the extended language allowed by the Smodels system as AnsProlog$_{sm}$. The Smodels system consists of two main modules: lparse and smodels. The lparse module takes an AnsProlog$_{sm}$ program Π and grounds the variables in Π to produce, $ground(\Pi)$, a grounded version of Π and outputs a representation of $ground(\Pi)$ that is readable by the smodels module. The smodels module then computes the answer sets of $ground(\Pi)$. The Smodels system expects that the user input to be in the language expected by the lparse module, and not in the format expected by the smodels module. Thus the input language of lparse is a Prolog like programming language while the input language of smodels is like a machine language.

To make sure $ground(\Pi)$ is of finite length and to achieve fast grounding by processing each rule only once the lparse program requires that its input satisfy the property of being *strongly range restricted*, according to which in every rule in the input, any variable that appears in the rule must also appear in a positive domain literal in the body. A domain literal is made up of domain

[1]The Smodels system has primitive functionality with respect to AnsProlog ^{or} and we do not discuss it here.

predicates which are either defined through facts or through other domain predicates without any recursion through **not** . We give a formal definition of domain predicates in Section 8.1.2.

The rest of this section is structured as follows. In Section 8.1.1 we present the syntax and semantics of ground AnsProlog$_{sm}$ and other ground statements. In Section 8.1.2 we describe how AnsProlog$_{sm}$ programs and other statements are grounded. In Section 8.1.3 we briefly present some of the other constructs in the input language of lparse and smodels modules. In Section 8.1.4 we present command line options for the lparse and smodels modules. In the remaining sections we present several small AnsProlog$_{sm}$ programs showcasing the features of the Smodels system.

8.1.1 The ground subset of the input language of lparse

We first start with ground AnsProlog$_{sm}$ programs and describe their semantics. We will then allow variables and describe the semantics of the resulting programs by explaining the grounding procedure.

A ground program in the input language of lparse consists of a set of ground AnsProlog$_{sm}$ rules, a set of compute statements and a list of optimize statements. Ground AnsProlog$_{sm}$ rules are more general than ground AnsProlog rules in that they allow more than atoms and naf-literals in their head and body respectively. They allow two kinds of constraints; cardinality and weight constraints.

- Ground cardinality constraint: A ground cardinality constraint C is of the form

$$L\ \{a_1, \ldots, a_n, \textbf{not}\ b_1, \ldots, \textbf{not}\ b_m\}\ U \tag{8.1.1}$$

  where a_is andb_js are atoms and L and U are integers. Both L and U are allowed to be missing and in that case their value is understood as $-\infty$ and ∞ respectively. Given a set of atoms S, by the value of C w.r.t. S (denoted by val(C,S)) we refer to the number $|S \cap \{a_1, \ldots, a_n\}| + (m - |S \cap \{b_1, \ldots, b_m\}|)$. We say C holds in S if $L \leq val(C, S) \leq U$. We refer L and U as $lower(C)$ and $upper(C)$ respectively.

- Ground weight constraint: A weight constraint C is of the form

$$L\ [a_1 = w_{a_1}, \ldots, a_n = w_{a_n}, \textbf{not}\ b_1 = w_{b_1}, \ldots, \textbf{not}\ b_m = w_{b_m}]\ U \tag{8.1.2}$$

  where a_is and b_js are atoms and $w's$, L and U are integers[2]. The weight w_{a_i} denotes the weight of a_i being true in a set and the weight w_{b_j} denotes the weight of **not** b being true in a set. If the weight of a naf-literal is 1 then its explicit specification may be omitted. As before, both L and U are allowed to be missing and in that case their value is understood as $-\infty$ and ∞ respectively. Given a set of atoms S, by the value of C w.r.t. S (denoted by val(C,S)) we refer to the number

$$\sum_{a_i \in S, 1 \leq i \leq n} w_{a_i} + \sum_{b_j \notin S, 1 \leq j \leq m} w_{b_j}.$$

[2]The restriction to integers is limited to the current implementation of the Smodels system. From the semantical point of view real numbers are acceptable.

We say C holds in S if $L \leq val(C,S) \leq U$. We refer L and U as $lower(C)$ and $upper(C)$ respectively.

Exercise 23 Let $S_1 = \{a, b, c, d\}$, and $S_2 = \{b, c, e, f\}$. Consider the following ground cardinality and weight constraints.

$C_1 = 1\ \{a, b, \textbf{not}\ d\}\ 3.$
$C_2 = 2\ \{a, e, \textbf{not}\ d\}\ 4.$
$C_3 = 1\ \{c, \textbf{not}\ d, e\}\ 1.$
$C_4 = 2\ [a = 2, b = 3, \textbf{not}\ d = 2]\ 8.$
$C_5 = 6\ [a = 3, d = 2, \textbf{not}\ e = 1]\ 10.$
$C_6 = 1\ [a, b, \textbf{not}\ d]\ 3.$

Which of the above C_is hold with respect to S_1 and which of the C_is hold with respect to S_2? □

For a ground constraint C (of either kind), by $atoms(C)$ we denote the set $\{a_1, \ldots, a_n\}$.

A ground AnsProlog$_{sm}$ rule is of the form:

C_0 :- $C_1, \ldots, C_k$.

where C_0 is either a ground atom, or a ground weight constraint or a ground cardinality constraint, and C_i $(1 \leq i \leq k)$ is either a ground literal, a ground weight constraint or a ground cardinality constraint. If $k = 0$ then the above rule is written simply as:

C_0.

Consider the following ground AnsProlog$_{sm}$ rule

$1\ \{a, b, c\}\ 2$:- p.

Intuitively, the meaning of the above rule is that if p is true in any answer set of a program containing this rule then at least 1 and at most 2 among the set $\{a, b, c\}$ must be true in that answer set. In general, the truth of a cardinality constraint C with respect to a set of atoms S is defined by counting the number of naf-literals in C that evaluate to true with respect to S. If this number is (inclusively) in between the lower and upper bound of C, then we say that C holds in S.

Example 134 Consider the following ground AnsProlog$_{sm}$ program.

$1\ \{a, b, c\}\ 2$:- p.
p.

While $\{a, p\}$, and $\{a, b, p\}$ are among the answer sets of this program $\{a, b, c, p\}$ is not an answer set. Notice that unlike in AnsProlog here we can have answer sets which are proper subsets of other answer sets. □

We now give the semantics of ground AnsProlog$_{sm}$ programs which have only non-negative weights and whose rules do not have any naf-literals. The former restriction is for simplifying the definitions, and because negative weights can be eliminated through some transformations without changing the meaning of the program. Similarly, a positive literal a can be replaced $1\ \{a\}\ 1$ and a negative literal **not** a can be replaced $1\ \{\textbf{not}\ a\}\ 1$, without changing the meaning of the program.

The semantics is defined in a style similar to the definition of answer sets for AnsProlog programs by first defining the notion of reduct and then using the notion of deductive closure of a simpler class of programs. First we define reducts of constraints. Let S be a set of atoms, and C be a ground constraint. The reduct of C with respect to S denoted by C^S is defined as follows:

- When C is a ground cardinality constraint of the form (8.1.1), C^S is the constraint

  $L'\ \{a_1, \ldots, a_n\}$

  where $L' = L - (m - |S \cap \{b_1, \ldots, b_m\}|)$

- When C is a ground weight constraint of the form (8.1.2), C^S is the constraint

  $L'\ [a_1 = w_{a_1}, \ldots, a_n = w_{a_n}]$

$$\text{where } L' = L - \sum_{b_j \notin S, 1 \leq j \leq m} w_{b_j}$$

Definition 84 Let Π be a set of ground AnsProlog$_{sm}$ rules and S be a set of ground atoms. The reduct Π^S of Π with respect to S is the set given by:

$\{p \leftarrow C_1^S, \ldots, C_k^S.\ :\ C_0$:- $C_1, \ldots, C_k. \in \Pi,\ p \in atoms(C_0) \cap S$, and for each of the C_i, $1 \leq i \leq k$, $val(C_i, S) \leq upper(C_i).\ \}$ □

Example 135 Consider a ground AnsProlog$_{sm}$ Π consisting of the following rule:

$1\ \{a, b, c\}\ 2$:- $1\ \{a, c, \textbf{not}\ d\}\ 2, 3\ \{b = 2, c = 1, \textbf{not}\ e = 2\}\ 6.$

Let $S = \{a, b\}$.

The reduct Π^S consists of the following rules:

a :- $0\ \{a, c\}, 1\ \{b = 2, c = 1\}.$

b :- $0\ \{a, c\}, 1\ \{b = 2, c = 1\}.$ □

Notice that the constraints in the rules of the reduct do not have negative naf-literals and have only lower bounds, and the head of the rules are only atoms. Such rules are monotonic in the sense that if the body of the rule is satisfied by a set of atoms S, then it is satisfied by any superset of S. The *deductive closure* of such a program is defined as the unique smallest set of atoms S such that if for any of the rules all the constraints in the body hold with respect to S then the atom in the head also holds w.r.t. S. This deductive closure can be obtained by an iterative procedure that starts from the empty set of atoms and iteratively adds heads of rules to this set whose bodies are satisfied by the set until no unsatisfied rules are left. Using the notion of deductive closure we now define answer sets of ground AnsProlog$_{sm}$ programs.

Definition 85 A set of ground atoms S is an answer set of a set of ground AnsProlog$_{sm}$ rules Π iff the following two condition holds,

(i) For each rule in Π if each of the C_is in the body hold w.r.t. S, then the head also holds w.r.t. S, and

(ii) S is equal to the deductive closure of Π^S. □

The definition of reduct in Definition 84 does not pay attention to the lower and upper bound in the head of the rules. This is taken care of by condition (i) of the Definition 85. The following examples illustrates this.

Example 136 Consider a ground AnsProlog$_{sm}$ program Π consisting of the following rule:

$1\ \{a, b\}\ 1.$

Let $S_1 = \{a\}$, $S_2 = \{b\}$, and $S_3 = \{a, b\}$.

The reduct of Π with respect to S_1 consists of the only rule:

$a.$

whose deductive closure is $\{a\} = S_1$. Since S_1 also satisfies the condition 1 of Definition 85, S_1 is an answer set of Π. Similarly, it can be shown that S_2 is an answer set of Π.

Now let us consider S_3. The reduct of Π with respect to S_3 consists of the following two rules:

$a.$
$b.$

whose deductive closure is $\{a, b\} = S_3$. So although S_3 is equal to the deductive closure of Π^{S_3} (thus satisfying condition 2 of Definition 85), it is not an answer set of Π as it does not satisfy condition 1 of Definition 85. □

Exercise 24 Extend the definition of answer sets to ground AnsProlog$_{sm}$ programs whose rules allow constraints with negative weights and give a transformation tr which eliminates negative weights, such that for any ground AnsProlog$_{sm}$ program π, the answer sets of π and $tr(\pi)$ coincide. (Hint: See page 9 of [NS97].) □

Recall that besides AnsProlog$_{sm}$ rules an input to lparse may also contain compute and optimize statements. The ground version of these statements are of the following forms.

- Compute statement:

  compute *number* $\{a_1, a_2, \ldots, a_n, \textbf{not}\ b_1, \textbf{not}\ b_2, \ldots, \textbf{not}\ b_m\}$.

  A compute statement is a filter (with a role similar to integrity constraint but with a different interpretation of the set of literals) on the answer sets. It eliminates all answer sets that is missing one of the a_is or includes one of the b_js. In its presence the total number of answer sets that are generated is dictated by *number*. If *number* $= 0$ or *all* then all answer sets are generated. If *number* is absent then by default a single answer set is generated.

- Optimize statements: They are of the following four kinds.

$$\text{maximize}\ \{a_1, \ldots, a_n, \textbf{not}\ b_1, \ldots, \textbf{not}\ b_m\}. \tag{8.1.3}$$

$$\text{minimize}\ \{a_1, \ldots, a_n, \textbf{not}\ b_1, \ldots, \textbf{not}\ b_m\}. \tag{8.1.4}$$

$$\text{maximize}\ [a_1 = w_{a_1}, \ldots, a_n = w_{a_n}, \textbf{not}\ b_1 = w_{b_1}, \ldots, \textbf{not}\ b_m = w_{b_m}]. \tag{8.1.5}$$

$$\text{minimize}\ [a_1 = w_{a_1}, \ldots, a_n = w_{a_n}, \textbf{not}\ b_1 = w_{b_1}, \ldots, \textbf{not}\ b_m = w_{b_m}]. \tag{8.1.6}$$

If the program contains a single weight optimization statements, then it returns the answer sets that has the optimum (minimum or maximum as the case may be) value of $[a_1 = w_{a_1}, \ldots, a_n = w_{a_n}, \textbf{not } b_1 = w_{b_1}, \ldots, \textbf{not } b_m = w_{b_m}]$. With respect to an arbitrary set of atoms S this value is

$$\sum_{a_i \in S, 1 \leq i \leq n} w_{a_i} + \sum_{b_j \notin S, 1 \leq j \leq m} w_{b_j}.$$

If the program contains a list of optimizations statements, then the optimality is determined by a lexicographic ordering where the first optimization statement is most significant and the last optimization statement is the least significant.

Note that any maximize statement of the form (8.1.5) can be converted to a minimize statement of the following form:

minimize $[\textbf{not } a_1 = w_{a_1}, \ldots, \textbf{not } a_n = w_{a_n}, b_1 = w_{b_1}, \ldots, b_m = w_{b_m}]$.

Example 137 Consider a ground AnsProlog$_{sm}$ program Π consisting of the following rule:

$1\ \{a, b, c, d\}\ 4.$

This program will have 15 answer sets, each of the non-empty subset of $\{a, b, c, d\}$.

Now suppose we have the following statement:

minimize $\{a, b, c, d\}$.

In that case only four answer sets, each of cardinality 1 will be returned.

Now if in addition we also have the following statement:

maximize $[a = 2, b = 1, c = 2, d = 1]$.

Then the two answer sets that will be returned are $\{a\}$ and $\{c\}$. □

8.1.2 Variables and conditional literals and their grounding

For programming convenience and to enable writing of smaller programs the input language of lparse allows variables and conditional literals which are of the form:

$p(X_1, \ldots, X_n) : p_1(X_{i_1}) : \ldots : p_m(X_{i_m})$

where $\{i_1, \ldots, i_m\} \subseteq \{1, \ldots, n\}$. The predicate p is referred to as the *enumerated predicate* and p_is are referred to as *conditions of* p. . It should be noted that in a conditional literal a condition of the enumerating predicate could be the predicate itself. In other word $p(X) : p(X)$ is a syntactically correct conditional literal.

To make the grounding procedure – where variables and conditional literals are eliminated – efficient, lparse puts certain restrictions on the variables and the conditional literals.

Grounding of AnsProlog$_{sm}$ programs without constraints

Let us first consider the subset of AnsProlog$_{sm}$ programs where we only have atoms and naf-literals in the rules and no constraints. Intuitively, the main restriction in this case is that any variable that appears in a rule must appear in another atom in the rule whose predicate is a 'domain' or 'sort' predicate. This 'domain' or 'sort' predicate describes the range of values the variable can

take. Thus during grounding variables are replaced only by the values that their corresponding sort predicate defines. To make grounding efficient the Smodels systems requires that the 'domain' predicates be defined such that their extent can be computed efficiently.

Formally, a predicate 'p' of an AnsProlog$_{sm}$ program Π is a *domain predicate* iff in the predicate dependency graph of Π every path starting from 'p' is free of cycles that pass through a negative edge. Following Section 3.3.1 this means that the sub-program of Π that consists of rules with p or any predicate that can be reached from p – in the predicate dependency graph – in its head is stratified.

An AnsProlog$_{sm}$ rule is *strongly range restricted* if for every variable that appears in that rule, it also appears in a positive domain literal in the body of the rule. An AnsProlog$_{sm}$ program is *strongly range restricted* if all its rules are strongly range restricted.

The grounding of the rules are done by first computing the extent of the domain predicates and then grounding the rest of the rules by using the computed extent of the domain predicates. Since the definition of domain predicates do not involve recursion through **not** , their extent can be efficiently computed using the iteration method to compute answer sets of stratified programs.

Grounding of AnsProlog$_{sm}$ programs

In addition to variables, lparse allows the additional construct of 'conditional literals' to help in compactly writing constraints. For example, given that the extent of a domain predicate *col* is $\{c_1, c_2, \ldots, c_k\}$, the sequence $color(v, c_1), \ldots, color(v, c_k)$ can be succinctly represented by the conditional literal $color(v, C) : col(C)$. For this conditional literal *col* is the condition of the enumerated predicate *color*. With variables these literals allow great compactness in writing rules.

We now formally define weight and cardinality constraints that allow variable and conditional literals.

Cardinality and weight constraints are either in the form of ground cardinality and weight constraints or of the following forms:

$$L\ \{Cond\_Lit\}\ U \tag{8.1.7}$$

$$L\ [Cond\_Lit]\ U \tag{8.1.8}$$

respectively, where L and U are either integers or variables, and *Cond_Lit* is a conditional literal. For weight constraints of the form (8.1.8) the weights of naf-literals that may be generated in the grounding process may be explicitly give using weight definitions. The syntax of weight definitions is given in the next section.

Continuing with our example, consider the fact that we have a graph with n vertices $v_1, \ldots, v_n$ and we would like to test k-colorability of this graph. To express this as an AnsProlog$_{sm}$ program we would first like to enumerate all possible colorings. With conditional literals and variables the enumeration part can be expressed by the single rule:

$1\ \{color(X, C) : col(C)\}\ 1$:- $vertex(X)$

and the facts

$\{col(c_1), \ldots, col(c_k), vertex(v_1), \ldots, vertex(v_n)\}$.

The grounding of the above rule would produce the following n ground rules whose heads will have k atoms.

$1\ \{color(v_1, c_1), \ldots, color(v_1, c_k)\}\ 1$:- $vertex(v_1)$

$\vdots$

$1\ \{color(v_n, c_1), \ldots, color(v_n, c_k)\}\ 1$:- $vertex(v_n)$

The lparse module requires that its input rules with conditional literals be *domain restricted* in the sense that each variable in the rules appears in a domain predicate or in the condition part of some conditional literal in the rule. Grounding of such AnsProlog$_{sm}$ rules are then done as follows:

1. The variables an AnsProlog$_{sm}$ rule are first divided into two categories: *local* and *global*. Those variables that appear only in a particular conditional literal, and nowhere else in the rule are labeled to be local with respect to that literal and all other variables in the rule are labeled as global. The grounding of the rules is now done in two steps.
2. First, all global variables are eliminated by substituting them with ground terms as dictated by domain predicates corresponding to that variable.
3. Second, conditional literals in constraints are eliminated by replacing a conditional literal say of the form $l(c_1, \ldots, c_n, X) : d(X)$ by $l(c_1, \ldots, c_n, d_1), \ldots, l(c_1, \ldots, c_n, d_k)$, where the extent of d is $\{d_1, \ldots, d_k\}$. (This is generalized to multiple conditions in the obvious way.)

8.1.3 Other constructs of the lparse language

The input language of lparse allows many additional constructs which facilitates programming using the Smodels system. We list some of them below and explain how they are processed.

- Range:

  A range is of the form

  start .. *end*

  where *start* and *end* are constant valued expressions.

  It can be use for compact representation in certain cases. For example, instead of writing

  $p(5)$.
  $p(6)$.
  $p(7)$.
  $p(8)$

  we may simply write $p(5..8)$.

  Similarly, instead of writing

  a :- $p(3), p(4), p(5)$.

  we may simply write

  a :- $p(3..5)$.

- Multiple arguments:

  These are a list of arguments separated by semicolons and can be used for compact representation. For example,

  $p(5..8)$.

  can also be written as

  $p(5;6;7;8)$.

  Similarly, instead of writing

  a :- $p(3), p(4), p(5)$.

  we may write

  a :- $p(3;4;5)$.

- Declarations

  - Function declaration: function f

    This statement declares that f will be used as a numeric function throughout the program.

  - Constant declaration: const $ident = expr$.

    This statement declares that the identifier $ident$ is a numeric constant with value $expr$, which may be any constant valued mathematical expression.

  - Weight definition: There are two kind of weight definitions.

    weight $literal = expr$.

    weight $literal_1 = literal_2$.

    The first declares that the default weight of the literal $literal$ is given by a mathematical expression $expr$, which may include variables that appear in $literal$. The second declares the weight of $literal_1$ to be same as the weight of $literal_2$ which may be defined in another part of the program. If the weight of a literal is defined more than once, only the latest one is used.

    Now suppose we have a predicate $value(Item, V)$ that uniquely defines the Value of an Item. We can use the declaration

    weight $value(Item, V) = 2 * V$

    so as to assign weights to $value$ atoms based on the value of the particular item. This will allow to use value of items in weight constraints or optimization statements.

  - Hide declaration: hide $p(X_1, \ldots, X_n)$.

    This statement tells the smodels program to not display atoms of the n-ary predicate p when displaying the answer sets.

Alternatively, when the set of predicates to hide is large, then the statements

hide. show $p(X_1, \ldots, X_n)$.

will only show the atoms of the predicate p when displaying the answer sets.

- Functions

  The lparse module has 18 different built-in functions: $plus, minus, times, div$, mod, lt, gt, le, ge, eq, $neq, assign, abs$, and, or, xor, not, and $weight$. Among these only the comparison functions and the weight function allow symbolic constants. The weight function takes a naf-literal as its only argument and returns its weight.

  The lparse module also allows user-defined C or C++ functions. But in comparison to the role of functions in classical logic or in Prolog it has a very restricted view and usage. It distinguishes between *numerical* functions and *symbolic functions*. While numerical functions are used to compute something, symbolic functions are used to define new terms.

  Numerical functions can occur either in a term or as a literal and in either case they are eliminated in the grounding phase. The role of symbolic functions are severely limited. Basically, if f is a symbolic function, then $f(a)$ basically defines a new constant that gets the name $f(a)$. Thus symbolic functions are not used to build lists. But we can have simple uses such as:

  $holds(neg(F), T)$:- fluent(F), time(T), not $holds(F, T)$.

  where, $neg(f)$ is a term defined using the symbolic function neg.

- Comments: % a comment

  The symbol % indicates that any text following it and until the end of the line is a comment.

- Declaration identifier: # a declaration

  Since the 1.0.3 version of lparse the symbol # is allowed (and recommended) before declarations.

8.1.4 Invoking lparse and smodels

Given a program Π acceptable to lparse written in the file $file.sm$, an answer set of it are obtained by the following command:

lparse $file.sm$ | smodels

When the above command is given in the command line prompt, if $file.sm$ has an answer set then one of its answer set is displayed in the terminal. To display 5 answer sets, we need to type:

lparse $file.sm$ | smodels 5

in the command line. Replacing 5 by 0 results in the display of all the answer sets. To store the answer sets in a file we can use the standard UNIX output notation and need to type:

lparse $file.sm$ | smodels 5 > $output\_file$

We now give some of the options available with lparse and smodels and their meaning. The reader is advised to follow the lparse manual for the updated and extended information on the options. The general usage using those options is as follows:

lparse list_of_lparse_options $file1 \ldots filen$ | smodels list_of_smodels_options

1. Important lparse options

   (a) −g $file$

   Reads previously grounded $file$ to memory before grounding the program.

   (b) −v

   Prints the lparse version information.

   (c) −−true-negation[0|1]

   Enables the classical negation extension.

   (d) −t

   Prints the ground program in a human readable form.

2. Important smodels options

   (a) −nolookahead

   Commands not to use look ahead during answer set computation.

   (b) −backjump

   Commands not to backtrack chronologically.

   (c) −sloppy_heuristic

   Commands not to compute the full heuristic after look ahead.

   (d) −randomize

   Does a randomized but complete search.

   (e) −w

   Displays the well-founded model.

   (f) −tries *number*

   Tries to find a model stochastically a *number* of times.

   (g) −seed *number*

   Uses *number* as seed for random parts of the computation.

8.1.5 Programming in Smodels: Graph colorability

Following is a program that can determine if a given graph is colorable by a given set of colors. The graph is described by the facts with respect to predicates *vertex* and *edge* and the given set of colors are expressed using the predicate *col*. In the following AnsProlog$_{sm}$ program the only rule with a cardinality constraint in its head assigns a unique color to each vertex. The same can be achieved through the following set of AnsProlog rules:

$other\_color(X, Y) \leftarrow vertex(X), color(X, Z), Y \neq Z.$
$color(X, Y) \leftarrow vertex(X), col(Y), \textbf{not}\ other\_color(X, Y).$

But the AnsProlog$_{sm}$ rule is more succinct and intuitive and also leads to better timings.

```
 vertex(1..4).

 edge(1,2).
 edge(1,3).
 edge(2,3).

 edge(1,4).
 edge(2,4).
% edge(3,4).

 col(a;b;c).

 1 { color(X,C) : col(C) } 1 :- vertex(X).

 :- edge(X,Y), col(C), color(X,C), color(Y,C).

 :- edge(Y,X), col(C), color(X,C), color(Y,C).

 hide col(X).
 hide vertex(Y).
 hide edge(X,Y).
```

When we run the above program we get six answer sets, and in each of them the color assigned to nodes 3 and 4 are the same. But if we remove the % before edge(3,4). and run the resulting program then we do not get any answer set as the resulting graph is not colorable using three colors.

8.1.6 Programming with smodels: Round robin tournament scheduling

Consider scheduling a tournament with 10 teams that play in a round-robin schedule, where each team plays the other team exactly once. This obviously takes 9 weeks. We have five stadiums where these games are played and each team plays every week. Besides obvious restrictions, such as only two teams can play at a time in a stadium, we have the restriction that no team plays more than twice in the same field over the course of the 9 weeks. The following is an AnsProlog$_{sm}$ program

that finds schedules for such a tournament. We use comments to give the intuitive meaning of some of the rules.

```
team(1..10).
week(1..9).
field(1..5).

%schedule(X,Y,Z) means that Z is a team that plays in field X in week Y.

1 { schedule(X,Y,Z) : field(X) } 1 :- week(Y), team(Z).

% The schedule should be such that for a given week, a team is scheduled
% in only one field.

2 { schedule(X,Y,Z) : team(Z) } 2 :- week(Y), field(X).

%The schedule should be such that for a given week, and a given
%fields exactly two teams are scheduled. In other words, 3
%different teams can not be scheduled on the same field on the same
%week, and at least 2 teams are scheduled on the same field on the
%same  week.

1 { schedule(X,Y,Z) : week(Y) } 2 :- field(X), team(Z).

%The schedule should be such that no team plays more than twice in
%the same field over the course of the season; and every team plays
%at least once in each field.

hide week(X).
hide team(X).
hide field(X).
```

8.1.7 Programming with smodels: Mini-ACC tournament scheduling

ACC is the Atlantic Coast Conference, a group of nine universities in the east coast of United states that play various inter-collegiate games among themselves and share the revenue. The basketball teams of these universities are quite famous and they use a specific set of guidelines to schedule their basketball games. Based on these guidelines a operation research based solution package has been developed [Hen00] and used. In this section we give an Smodels solution for a simplified version of the problem. Although couple of my students have come up with a reasonable solution that calls Smodels multiple times and follows some of the ideas from [Hen00], we have not been able to come up with a direct formulation in Smodels that can come up with a schedule in 24 hours. We now describe the simplified version, which we will refer to as the mini-ACC tournament scheduling.

There are five teams: duke, fsu, umd, unc, uva; duke and unc are considered rivals and also umd and uva. They play 10 dates over five weeks. In each week there is a week day game and a weekend game. Besides the obvious restrictions such as a team can not play itself, we have the following restrictions.

Two teams play exactly twice over the season. Over the season a team may plays another team at home exactly once. No team may have more than two home matches in a row. No team can play away in both of the last two dates. No team may have more than 3 away matches or byes in a row. In the weekends, each team plays 2 at home, 2 away and one bye. In the last date, every team except fsu, plays against its rival, unless it plays against fsu or has a bye. No team plays in two consecutive days away against duke and unc. The following is an AnsProlog$_{sm}$ program whose answer sets encode schedules agreeing with the above restrictions and a few additional specific restrictions about particular teams.

```
date(1..10).
team(duke;fsu;umd;unc;uva).

rival(duke,unc).
rival(umd,uva).

rival(X,Y) :-  team(X), team(Y), rival(Y,X).

weekday(X) :- date(X), not weekend(X).
weekend(X) :- date(X), not weekday(X).

weekday(X+1) :- date(X), weekend(X).
weekend(X+1) :- date(X), weekday(X).

%sch(X,Y,D) means that Teams X and Y are scheduled to plays on date
%D at the home court of X.

plays(X,Y,D)   :-  team(X), team(Y), date(D), sch(X,Y,D).
plays(X,Y,D)   :-  team(X), team(Y), date(D), sch(Y,X,D).

plays(X,D)  :-  team(X), team(Y), date(D), plays(X,Y,D).

plays_at_home(X,D)  :-  team(X), team(Y), date(D), sch(X,Y,D).

plays_away(X,D)  :-  team(X), team(Y), date(D), sch(Y,X,D).

has_bye(X,D)   :-  team(X), date(D), not plays(X,D).


0 { sch(X,Y,D) : team(Y) } 1   :-  date(D),
                                   team(X).

%The schedule should be such that for a particular date a team may
%only play at most one more team at home.

:- team(X), team(Y), team(Z), date(D), not eq(Z,Y), plays(X,Y,D),
   plays(X,Z,D).
```

```
%The schedule should be such that for a particular date a team can
%not play more than one other team.

 :- team(X), team(Y), date(D), sch(X,Y,D), sch(Y,X,D).

%The schedule should be such that for a particular date a team
%cannot play both at home and away.

 :- team(X), team(Y), date(D), sch(X,Y,D), eq(X,Y).

%The schedule should be such that a team can not play itself.

 1 { sch(X,Y,D) : date(D) } 1 :-  team(X), team(Y), not eq(X,Y).

%The schedule should be such that over the season a team plays
%another team at home exactly once; and thus two teams play exactly
%twice over the season.

 :- team(X), plays_away(X,9), plays_away(X,10).

%No team plays away in both of the last two dates.

 :- team(X), date(T), date(T+1), date(T+2), plays_at_home(X,T),
    plays_at_home(X,T+1), plays_at_home(X,T+2).

%No team may have more than two home matches in a row.

 away_or_bye(X,D)    :-  team(X), date(D), plays_away(X,D).
 away_or_bye(X,D)    :-  team(X), date(D), has_bye(X,D).

 :- team(X), date(T), date(T+1), date(T+2), date(T+3), away_or_bye(X,T),
    away_or_bye(X,T+1), away_or_bye(X,T+2),  away_or_bye(X,T+3).

%No team may have more than 3 away matches or byes in a row.

 plays_at_home_weekend(X,D) :- team(X), date(D), weekend(D),
                               plays_at_home(X,D).
 plays_away_weekend(X,D) :- team(X), date(D), weekend(D),
                            plays_away(X,D).

 has_bye_weekend(X,D) :-    team(X), date(D), weekend(D),
                            has_bye(X,D).

 cond1(X) :-  2 {plays_at_home_weekend(X,D) : date(D)} 2, team(X).

 :- team(X), not cond1(X).
```

```
 cond2(X) :-  2 {plays_away_weekend(X,D) : date(D)} 2, team(X).

 :- team(X), not cond2(X).

 cond3(X) :-  1 {has_bye_weekend(X,D) : date(D)} 1, team(X).

 :- team(X), not cond3(X).

%The above set of rules enforce the condition that in the (five)
%weekends, each team plays 2 at home, 2 away and one bye.

 :- team(X), team(Y), plays(X,Y,10), not rival(X,Y),
        not eq(X,fsu), not eq(Y,fsu).

% In the last date, every team except fsu, plays against its
% rival, unless it plays against fsu or has a bye.

 plays_duke_unc(X,D) :-  team(X), date(D), sch(unc,X,D).
 plays_duke_unc(X,D) :-  team(X), date(D), sch(duke,X,D).

 :- team(X), date(T), date(T+1), plays_duke_unc(T),
    plays_duke_unc(T+1).

%No team plays in two consecutive days away against duke and unc.

 :- not plays(unc,duke,10). %unc must plays duke in date 10.
 :- not has_bye(duke,7).    %duke has a bye in date 7.
 :- plays_at_home(wfu,9).   %wake does not plays at home in date 9.
 :- plays_away(umd,10).     %umd does not plays away in date 10.
 :- has_bye(fsu,10).        %fsu does not have a bye in date 10.
 :- has_bye(umd,8).         %umd does not have a bye in date 8.
 :- plays(umd,uva,6).       %uva does not play at umd in date 6.
 :- not plays_away(uva, 7). %uva plays away in week 7.
 :- not plays_at_home(unc,3). %unc plays at home in week 6.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

 hide team(X).
 hide date(X).
 hide rival(X,Y).
 hide weekday(X).
 hide plays(X,Y).
 hide plays(X,Y,Z).
 hide has_bye(X,Y).
 hide plays_at_home(X,D).
 hide plays_away(X,D).
 hide away_or_bye(X,D).
```

```
hide home_or_bye(X,D).
hide plays_at_home_weekend(X,D).
hide plays_away_weekend(X,D).
hide has_bye_weekend(X,D).
hide plays_duke_unc(X,T).
hide plays_duke_unc_wfu(X,D).
hide cond1(X).
hide cond2(X).
hide cond3(X).
```

8.1.8 Programming with smodels: Knapsack problem

We now give an example of encoding with Smodels that uses optimize statements. We consider a simple knapsack problem where we have five items (1-5) and each item has a cost (or size) and value associated with it. We have a sack with a given capacity (12) and the goal is to select a subset of the items which can fit the sack while maximizing the total value. This can be encoded in Smodels as follows:

```
item(1..5).

weight val(1) = 5.
weight val(2) = 6.
weight val(3) = 3.
weight val(4) = 8.
weight val(5) = 2.

weight cost(1) = 4.
weight cost(2) = 5.
weight cost(3) = 6.
weight cost(4) = 5.
weight cost(5) = 3.

in_sack(X) :-  item(X), not not_in_sack(X).

not_in_sack(X) :- item(X), not in_sack(X).

val(X) :-  item(X), in_sack(X).

cost(X) :- item(X), in_sack(X).

cond1 :- [ cost(X) : item(X) ] 12.

:- not cond1.

maximize { val(X) : item(X) }.

hide item(X).
```

```
hide not_in_sack(X).
hide cost(X).
hide val(X).
```

8.1.9 Programming with smodels: Single unit combinatorial auction

In a combinatorial auction problem bidders are allowed to bid on a bundle of items. The auctioneer has to select a subset of the bids so as to maximize the price it gets, and making sure that it does not accept multiple bids that have the same item as each item can be sold only once. Following is an example of a single unit combinatorial auction problem. The auctioneer has the set of items $\{1, 2, 3, 4\}$, and the buyers submit bids $\{a, b, c, d, e\}$ where a constitutes of $\langle\{1, 2, 3\}, 24\rangle$, meaning that the bid a is for the bundle $\{1, 2, 3\}$ and its price is \$24. Similarly b constitutes of $\langle\{2, 3\}, 9\rangle$, c constitutes of $\langle\{3, 4\}, 8\rangle$, d constitutes of $\langle\{2, 3, 4\}, 25\rangle$, and e constitutes of $\langle\{1, 4\}, 15\rangle$. The *winner determination* problem is to accept a subset of the bids with the stipulation that no two bids containing the same item can be accepted, so as to maximize the total price fetched. We now present an AnsProlog$_{sm}$ encoding of this example.

```
bid(a;b;c;d;e).

item(1..4).

in(1,a).
in(2,a).
in(3,a).

in(2,b).
in(3,b).

in(3,c).
in(4,c).

in(2,d).
in(3,d).
in(4,d).

in(1,e).
in(4,e).

weight sel(a) = 24.
weight sel(b) = 9.
weight sel(c) = 8.
weight sel(d) = 25.
weight sel(e) = 15.

sel(X)  :-  bid(X), not not_sel(X).
```

```
not_sel(X)  :-  bid(X), not sel(X).

:- bid(X), bid(Y), sel(X), sel(Y), not eq(X,Y), item(I),
   in(I,X), in(I,Y).

maximize [ sel(X) : bid(X) ].


hide bid(X).
hide not_sel(X).
hide item(X).
hide in(X,Y).
```

8.1.10 Some AnsProlog$_{sm}$ programming tricks

In this section we present some AnsProlog$_{sm}$ programming tricks and show how certain programming constructs and data structures can be encoded using these tricks.

Aggregation: Weight constraints and cardinality constraints can be used to encode certain aggregations such as *count* and *sum*. Consider the following example from Section 2.1.14, where $sold(a, 10, Jan1)$ means that 10 units of item a was sold on Jan 1.

```
sold(a, 10, jan1).
sold(a, 21, jan5).
sold(a, 15, jan16).
sold(b, 16, jan4).
sold(b, 31, jan21).
sold(b, 15, jan26).
sold(c, 24, jan8).
```

We now show how to use weight constraints to compute the total units sold and total number of selling transactions for each of the items.

```
item(a;b;c).
number(1..100).
date(jan1;jan5;jan16;jan4;jan21;jan26;jan8).

weight sold(X,Y,Z) = Y.

total_sold(I, N) :- item(I), number(N),
                    N [ sold(I, X, D) : number(X) : date(D)  ] N.

total_sell_transactions(I, N) :- item(I), number(N),
                    N { sold(I, X, D) : number(X) : date(D)  } N.
```

Sets and Lists: Since Smodels has very limited facility for expressing symbolic functions, we can not use symbolic functions – as in Prolog – to express lists. Nevertheless, we can explicitly express lists and process them. In the following we show some of the techniques.

1. In representing effects of actions often the atom $causes(a, l_0, [l_1, \ldots, l_n])$ is used to express that the execution of action a will make l_0 true if $l_1, \ldots, l_n$ are initially $true$. In the absence of the [ ] notation (which is a short form of a term built using symbolic functions), we can express the above atom in AnsProlog$_{sm}$ in the following way:

```
causes(a, l0, s).

set(s).

in(l1, s).
...
in(ln, s).
```

2. Now suppose we want to verify that every element of the set (s) holds at time T. This can be expressed in AnsProlog$_{sm}$ as follows:

```
not_holds(S, T) :- set(S), time(T), in(I,S), not holds(I,T).

holds(S,T) :- set(S), time(T), not not_holds(S, T).
```

3. Now instead of sets, suppose we would like to deal with linear lists. For example, we may want to verify if l will be true after executing the sequence of actions $a_1, \ldots, a_n$ in time t. In other words we want to find out if $holds\_after(l, [a_1, \ldots, a_n], t)$ is true or not. In AnsProlog$_{sm}$ we will express this as

```
holds_after(l, list, t).

action(1, list, a1).
...
action(n, list, a_n).
```

and will have the following AnsProlog$_{sm}$ rules to reason with it.

```
not_holds_set(Set, N, List, T) :- in(L, Set),
                                  not holds_after(L, N, List, T).
holds_set(Set, N, List, T) :- not not_holds_set(Set, N, List, T).

not_last(List, N) :- action(M, List, A), N < M.
last(List, N) :- not  not_last(List, N).

holds_after(L, List, T) :- holds_after(L, N, List, T),
                           last(List, N).
holds_after(L, 0, List, T) :- holds(L, T).
holds_after(L, N, List, T) :- holds_after(L, N-1, List,T),
                              action(N, List, A), not ab(L, N, List,T).
holds_after(L, N, List, T) :- action(N, List, A), causes(A, L, Set),
```

```
                                holds_set(Set, N-1, List, T).

ab(LL, N, List, T) :-  action(N, List, A), causes(A, L, Set),
                       holds_set(Set, N-1, List, T).
```

4. Finally lists of arbitrary depth such as in the following atom $p([a,[b,[c,d]])$ can be expressed as follows:

```
p(l).

head(l,a).
body(l, l1).

head(l1, b).
body(l1, l2).

head(l2, c).
body(l2, l3).

head(l3, d).
body(l3, nil).
```

In Section 5.5 we use this notation in processing procedural constraints.

8.2 The dlv system

The dlv system is meant for computing answer sets of AnsProlog* programs and offers several front-ends for knowledge representation tasks such as query answering in the brave and cautious mode, diagnosis, planning, and answering a subset of SQL3 queries.

When the dlv system is invoked with an input AnsProlog* program it processes it in several steps. First it converts the input to an internal representation. Then it converts the program to an equivalent program with out variables. In the next step it generates possible answer sets and checks there validity. Finally the answer sets (or an answer set) goes through post-processing as dictated by the front-end.

The main difference between the smodels system of Section 8.1 and the dlv system is that the later is centered around AnsProlog^{or} programs, while the former has only primitive functionality with respect to disjunction. Among of the other differences between the two are that dlv allows specification of queries in the program and allows both brave and cautious reasoning modes. The following table elaborates the differences between smodels and dlv.

Issues	smodels system	dlv system
Main focus	AnsProlog	AnsProlog^{or}
Notation for disjunction		3 alternate notations (v, |, ;)
Query specification Notation for queries	indirectly using constraints	allows direct specification $p_1, \ldots, p_m$, **not** $q_1, \ldots,$ **not** q_m? (where p_is and q_js are literals.
Query answering modes		brave, cautious
Beyond AnsProlog*	Weight constraints, cardinality constraints, optimization statements	weak constraints, several front-ends (diagnosis, planning, SQL3)
Using arithmetic in rules	$p(T+1)$:- $p(T)$.	the one on the left is not acceptable; $p(TT)$:- $p(T), TT = T+1$. is.
Function symbols	allowed with limitations	not allowed
Explicit negation	allowed *Lit* could be an answer set.	allowed, uses the symbol '-' . *Lit* can not be an answer set.
Anonymous variables	not allowed	Can be specified using _ $student(X) : -takes\_class(X, \_)$.
Restriction on rules	strongly range restricted	range restricted
Input interface		Oracle, Objectivity (both experimental).
API	User defined C/C^{++} functions	

8.2.1 Some distinguishing features of dlv

In this section we discuss some of the distinguishing features of the dlv system in greater detail.

The dlv system requires that each rule in the input program be range restricted. Recall from Section 1.2.2 that a rule is said to be range-restricted if each variable occurring in the a rule, must occur at least in one of the non-built-in comparative positive literal in the body of the rule. Note that this restriction on rules is weaker than the restriction of 'strong range-restrictedness' imposed by the Smodels system.

There is one exception to the range restricted requirement in rules. The dlv system allows usage of *anonymous variables* denoted by _ (an underscore). For example, if we want to write a rule saying that X is a student if (s)he is taking some class Y. We can write a rule using the anonymous variable notation in the following way.

$student(X)$:- $takes\_class(X, \_)$.

The dlv system allows the usage of classical negation. Thus it accepts AnsProlog$^{\neg, or}$ programs. In its characterization of AnsProlog$^{\neg, or}$ programs it assumes that inconsistent models do not exist. In other words while as defined in this book, a program may have the set *Lit* consisting of all literals (positive and negative)as an answer set, in the dlv system such is not the case. For example, although the following program

p.
q.
$\neg q$.

has $Lit = \{p, q, \neg q, \neg p\}$ as its answer set, according to the dlv system it has no answer set. This is a minor draw back of the dlv system as it does not allow us to distinguish between programs

that do not have answer sets and programs that have only Lit as an answer set. In contrast in the smodels system there is an option to specify whether to consider Lit as an answer set or not.

The dlv system allows the specification of queries. A query is of the form:

$$p_1, \ldots, p_m, \textbf{not } q_1, \ldots, \textbf{not } q_n? \tag{8.2.9}$$

where p_i and q_j are literals. The dlv system allows two modes of querying: brave and cautious; which are specified by the command line options $-$FB and $-$FC respectively. A query of the form (8.2.9) is true in the brave mode if each of the p_is and **not** q_j's are satisfied in at least one answer set of the program. A query is true in the cautious mode if it is satisfied in all answer sets of the program.

In the brave mode of reasoning, a query of the form (8.2.9) can alternatively be encoded by the following set of constraints:

:- **not** p_1.
$\vdots$
:- **not** p_m.
:- q_1.
$\vdots$
:- q_n.

which will eliminate answer sets that do not satisfy the query.

The dlv system allows four arithmetic predicates: #int, #succ, +, and *. When one or more these predicates are used then dlv must be invoked with the option '-N=int-value'. For example, when dlv is invoked with '-N=20' it means that the system only considers integers between 0 and 20. Intuitively, #int(X) is true if X is an integer less than int-value; #succ(X,Y) is true if X + 1 = Y; +(X,Y,Z) is true if Z = X + Y; and *(X,Y,Z) is true if Z = X*Y. The dlv system does allow the writing Z = X + Y instead of +(X,Y,Z) and Z = X * Y instead of *(X,Y,Z). However, it should be noted that unlike in smodels, + and * can not be used as function symbols. Thus, the rule

$p(T + 1)$:- $p(T)$.

is not acceptable to dlv. An equivalent rule acceptable to dlv would be:

$p(TT)$:- $p(T), TT = T + 1$.

One of the other novel features of dlv is the notion of *weak constraints*. We discuss this further in the next section.

8.2.2 Weak constraints in dlv vs. optimization statements in smodels

Weak constraints in dlv are of the form

$$:\sim p_1, \ldots, p_m, \textbf{not } q_1, \ldots, \textbf{not } q_n.[weight : level] \tag{8.2.10}$$

where p_is and q_js are literals, and weight and level are integers or integer variables that appear in the p_is.

Given a program with weak constraints its *best* answer sets are obtained by first obtaining the answer sets with out considering the weak constraints and ordering each of them based on the weight and priority level of the set of weak constraints they violate, and then selecting the ones that

violate the minimum. In presence of both weight and priority level information, the minimization is done with respect to the weight of the constraints of the highest priority, then the next highest priority and so on. Note that the weak constraints may contain none, or one or both of the weight and priority information, but it is required that all the weak constraints have the same syntactic form. I.e., if one of them has only weight information then all of them can have only weight information, and so on. If both the weight and level information are omitted then they are set to the value 1 by default.

Consider the following program with weak constraints.

$a \ v \ b$
$-a \ v \ c$
$:\sim a$
$:\sim b, c$

The *best* answer set of the above program is $\{-a, b\}$, which does not violate any of the weak constraints, while the other two answer sets $\{a, c\}$ and $\{b, c\}$, each violate one weak constraint.

Weak constraints in dlv serve a similar purpose as optimization statements in dlv. For example, the following optimizations statement in smodels

minimize $\{a_1 = w_{a_1}, \ldots, a_n = w_{a_n}, \textbf{not } b_1 = w_{b_1}, \ldots, \textbf{not } b_m = w_{b_m}\}$

can be expressed by the following set of weak constraints in dlv.

$:\sim a_1.[w_{a_1} : 1].$
$\vdots$
$:\sim a_n.[w_{a_n} : 1].$
$:\sim \textbf{not } b_1.[w_{b_1} : 1].$
$\vdots$
$:\sim \textbf{not } b_m.[w_{b_m} : 1].$

Similarly, a set of weak constraints specifying the same priority level can be replaced by a minimize statement with some additional rules to link conjunctions in the weak constraints to individual atoms that are included in the minimize statement.

8.2.3 Invoking dlv

To process a program using dlv, it is invoke by the following command

dlv [front-end-options] [general-options] [file1, ..., filen]

Earlier we mentioned two of the front-end-options (-FB and -FC resp.) for brave and cautious reasoning. In addition to those front-end options following are some useful general options in dlv. The dlv manual has the exhaustive list of front-end options and general options.

1. −filter = p

   This option specifies that the literals made up of predicate p are not to be displayed. This option may be used multiple times to specify a set of predicates.

2. −n = number

   This option specifies the upper bound on the number of answer sets that will be displayed. If 'number' is 0, then all answer sets are displayed.

3. −N = number

   This specifies the maximum value of integers that the program should consider.

4. −OH−

   This option disables heuristics in the model generator. When this option is absent, by default, heuristics are used in the model generator.

5. −OMb−

   This option disables the use of a novel backtracking technique in the model generator. When this option is absent, by default, the novel backtracking technique is used in the model generator.

6. −−

   This option tells dlv to read input from the stdin.

8.2.4 Single unit combinatorial auction using weak constraints

In Section 8.1.9 we showed how to encode the single unit combinatorial auction problem using smodels. Here we encode it in the language of dlv. The main differences in the two encoding are that hear we use a rule with disjunction in the head for enumeration, and use weak constraints instead of the optimization statement in the AnsProlof$_{sm}$ encoding.

```
bid(a).
bid(b).
bid(c).
bid(d).
bid(e).

in(1,a).
in(2,a).
in(3,a).

in(2,b).
in(3,b).

in(3,c).
in(4,c).

in(2,d).
in(3,d).
in(4,d).

in(1,e).
in(4,e).

sel(X) v not_sel(X) :- bid(X).
```

```
:- bid(X), bid(Y), sel(X), sel(Y),
   X != Y, in(I,X), in(I,Y).

:~ not sel(a). [24:1]
:~ not sel(b). [9:1]
:~ not sel(c). [8:1]
:~ not sel(d). [25:1]
:~ not sel(e). [15:1]
```

8.2.5 Conformant planning using dlv

In Chapter 5 we discussed the issue of planning in great detail. But except in Section 5.4 we assumed that the planning agent has complete information about the initial state. In Section 5.4 we relaxed this assumption and introduced a notion of approximate planning whose complexity is NP-complete. In this section we consider the more general notion of conformant planning, but whose complexity is $\Sigma_2\mathbf{P}$-complete. In conformant planning, we look for a sequence of actions which is a plan for all possible complete initial states that agree with the (incomplete) knowledge that the agent has about the initial state. Conformant planning is not expressible in AnsProlog, but is expressible in AnsProlog^{or}, the focus of the dlv system. We illustrate how to encode conformant planning problems in dlv, through a simple example.

Consider a domain with actions a and b and fluents p and f, where the effect of the actions are described by the following propositions in the language $\mathcal{A}$ (from Chapter 5).

a **causes** f **if** p
a **causes** f **if** $\neg p$
b **causes** f **if** p

The (incomplete) information about the initial state is given by the following.

initially $\neg f$

The goal of the agent is to make f *true*.

Intuitive the only conformant plan of length 1 is the single action a. The following encoding has one answer set which encodes this plan. This encoding is similar to the encoding of exiestential-universal QBFs in Section 2.1.11, as a conformant plan can be charaterized as "there exists a sequence of actions (α) such that for all possible initial states (σ), execution of α in σ will takes us to a goal state."

```
time(1).

action(a).
action(b).

fluent(f).
fluent(p).
fluent(ff).
fluent(pp).
```

```
initially(ff).

holds(F,1) :- fluent(F), initially(F).

opp(f,ff).
opp(ff,f).
opp(p,pp).
opp(pp,p).

holds(F, TT) :- fluent(F), fluent(FF), opp(F,FF), time(T),
                TT = T + 1, holds(F,T), not holds(FF, TT).

holds(f, TT) :- time(T), TT = T + 1, occurs(a,T), holds(p,T).
holds(f, TT) :- time(T), TT = T + 1, occurs(a,T), holds(pp,T).
holds(f, TT) :- time(T), TT = T + 1, occurs(b,T), holds(p,T).

not_occurs(A,T) :- action(A), time(T), action(B),
                   A != B, occurs(B,T).

occurs(A,T) :- action(A), time(T), not not_occurs(A,T).

holds(p,1) | holds(pp, 1).

goal_sat :- holds(f,2).

holds(f,1)      :-  goal_sat.
holds(ff,1)     :-      goal_sat.
holds(p,1)      :-      goal_sat.
holds(pp,1)     :-  goal_sat.

holds(f,2)      :- goal_sat.
holds(ff,2)     :- goal_sat.
holds(p,2)      :- goal_sat.
holds(pp,2)     :- goal_sat.

:- not goal_sat.
```

We now show the output obtained by running the above program using dlv. We run two versions of the above program: planning10.dl and planning11.dl. The only difference between the two is that in the second one the constraint ':- not goal_sat' is commented out.

1. dlv -N=5 planning10.dl

   ```
   dlv [build DEV/Nov  7 2000   gcc egcs-2.91.57 19980901 (egcs-1.1
   release)]
   ```

```
{time(1), action(a), action(b), fluent(f), fluent(p), fluent(ff),
fluent(pp), initially(ff), opp(f,ff), opp(p,pp), opp(ff,f),
opp(pp,p), holds(ff,1), occurs(a,1), not_occurs(b,1), holds(p,1),
holds(pp,1), holds(ff,2), holds(f,2), holds(p,2), holds(pp,2),
goal_sat, holds(f,1)}
```

The reason we do not have an answer set with $occurs(b, 1)$ and $goal\_sat$ is that even though there seem to be an enumeration where p is $true$ at time point 1, this enumeration is not minimal, as it leads to $goal\_sat$ which makes all fluent literals to be $true$ at time point 1 and 2, and there is another enumeration where pp – inverse of p – is $true$ at time point 1 which is a subset of the previous enumeration. Thus the first enumeration does not lead to an answer set. The second one is eliminated as it does not have $goal\_sat$. As evident from the next item, when we remove the constraint $\leftarrow goal\_sat.$, we do have an answer set with $occurs(b, 1)$ and $holds(pp, 1)$ but without $goal\_sat$.

In the case where we have $occurs(a, 1)$, both possibilities with p true at time point 1, and its inverse pp true at time point 1, lead to $goal\_sat$ which makes all fluent literals to be $true$ at time point 1 and 2, leading to a single answer set.

2. dlv -N=5 planning11.dl

```
dlv [build DEV/Nov  7 2000   gcc egcs-2.91.57 19980901 (egcs-1.1
release)]

{time(1), action(a), action(b), fluent(f), fluent(p), fluent(ff),
fluent(pp), initially(ff), opp(f,ff), opp(p,pp), opp(ff,f),
opp(pp,p), holds(ff,1), not_occurs(a,1), occurs(b,1), holds(pp,1),
holds(ff,2), holds(pp,2)}

{time(1), action(a), action(b), fluent(f), fluent(p), fluent(ff),
fluent(pp), initially(ff), opp(f,ff), opp(p,pp), opp(ff,f),
opp(pp,p), holds(ff,1), occurs(a,1), not_occurs(b,1), holds(p,1),
holds(pp,1), holds(ff,2), holds(f,2), holds(p,2), holds(pp,2),
goal_sat, holds(f,1)}
```

8.3 Pure Prolog

Prolog is a programming language based on logic and the term Prolog is a short form of **Pro**gramming in **log**ic. A Prolog program consists of a set of rules similar to AnsProlog rules. Users interact with Prolog programs by asking Prolog queries with respect to the program, where a Prolog query is a sequence of naf-literals. The answer with respect to ground queries is usually (i.e., when the query processing terminates) $true$ or $false$. When the query has variables, along with the answer $true$ the Prolog interpreter returns an answer substitution for the variables in the query. The query answering approach of Prolog is very different from the approach in Smodels and dlv. While both Smodels and dlv compute the answer sets first, in Prolog query answering is driven by the query.

Prolog rules are more general than AnsProlog rules and have built-in predicates and non-logical features such as *cut*. In this section our focus is on *the behavior of the Prolog's execution mechanism on programs in AnsProlog syntax*. This language with syntax of AnsProlog and semantics

of Prolog is referred to as *Pure Prolog. The semantics of Pure Prolog is defined procedurally in terms of SLDNF-resolution with the leftmost selection rule (referred to as LDNF-resolution), with the exception that selection of non-ground literals is allowed (i.e., floundering is ignored), during resolution rules are tried from the beginning of the program to the end, and occur check is omitted during unification.* We will explain each of these terms in the rest of this section, and present sufficiency conditions when the semantics of Pure Prolog agrees with the semantics of AnsProlog.

We start with examples illustrating the procedural semantics of Pure Prolog and its deviation from the semantics of AnsProlog. The analysis of the deviations led to the development of the sufficiency conditions that guarantee conformity with AnsProlog.

In the rest of this section, by a query, we will mean a Prolog query, a sequence of naf-literals.

Example 138 Consider the following Pure Prolog program:

r_1 : $anc(X, Y) \leftarrow par(X, Y)$.
r_2 : $anc(X, Y) \leftarrow par(X, Z), anc(Z, Y)$.

f_1 : $par(a, b)$.
f_2 : $par(b, c)$.
f_3 : $par(h, c)$.
f_4 : $par(c, d)$.
f_5 : $par(e, f)$.
f_6 : $par(f, g)$.

If the query $anc(a, b)$ is asked with respect to this program, the interpreter looks for a rule with anc in its head and whose head unifies with $anc(a, b)$. (Recall that the notion of unification was defined in Section 3.10.1 and in this case it refers to substitution of variables by terms such that after the substitution we obtain the same atom.) It finds such a rule r_1, with the mgu $\{X/a, Y/b\}$. The interpreter now replaces $anc(a, b)$ in the query by the body of the rule with the mgu applied to it. So the new query is $par(a, b)$. The above step is referred to as: "The query $anc(a, b)$ resolves to the query $par(a, b)$ via $\{X/a, Y/b\}$ using r_1."

It now looks for a rule with par in its head and whose head unifies with $par(a, b)$. It finds such a rule f_1, with the mgu $\{\}$. As before, the interpreter now replaces $par(a, b)$ in the query by the body of the rule with the mgu applied to it. Since the body of f_1 is empty, the new query is empty. With an empty query the interpreter returns the answer $true$. I.e., the query $par(a, b)$ resolves to the empty query via $\{\}$ using f_1. Since the original query did not have any variables, the interpreter does not return any answer substitutions.

Now consider the query $anc(e, W)$. It resolves to $par(e, W)$ via $\{X/e, Y/W\}$ using r_1. In the next step $par(e, W)$ resolves to the empty query via $\{W/f\}$ using f_5. The interpreter then returns $true$ and the answer substitution $\{W/f\}$ meaning that f is an ancestor of e. If the interpreter is asked for another answer it backtracks its steps and resolves the query $anc(e, W)$ to the query $par(e, Z), anc(Z, W)$ via $\{X/e, Y/W\}$ using r_2. In the next step the query $par(e, Z), anc(Z, W)$ resolves to the query $anc(f, W)$ via $\{Z/f\}$ using f_5. The query $anc(f, W)$ then resolves to the query $par(f, W)$ via $\{X/a, Y/W\}$ using r_1. The query $par(f, W)$ then resolves to the empty query via $\{W/g\}$ using f_6. The interpreter then returns $true$ and the answer substitution $\{W/g\}$ meaning that g is another ancestor of e. If the interpreter is asked for another answer it backtracks and looks for another answer and it fails in its attempts and returns $false$. □

The above example illustrates some aspects of the execution mechanism of Pure Prolog. One important point of this illustration is that because of the query driven approach the facts $f_1 - f_4$ are not touched while answering the query $anc(e, W)$. This is not the case if Smodels or dlv is used. Both of them will compute the answer set of the program and in the process reason about $f_1 - f_4$ even though they are not relevant to the query.

Now let us illustrate the Pure Prolog execution mechanism with respect to a program that has **not** .

Example 139 Consider the following program:

$p \leftarrow q, \textbf{not } r.$
$q \leftarrow.$
$r \leftarrow s.$
$r \leftarrow t.$

Consider the query p. It resolves to $q, \textbf{not } r$ which resolves to **not** r. The execution mechanism treats negative naf-literals in a query differently. In this case it does not look for a rule whose head unifies with **not** r, as Pure Prolog rules do not have negative naf-literals in their head. Instead it attempts to answer a new query r with respect to the program.

The query r initially resolves to s and s does not resolve to anything else. This leads to a failure branch and the execution mechanism backtracks and now resolves r to t, which again does not resolve to anything else leading to another failure branch. Since no further backtracking is possible, the query r is said to *finitely fail*.

The finite failure of the query r is interpreted as a success in resolving the query **not** r to the empty query, and hence the answer to the original query p is given as $true$. □

The top-down execution mechanism of Pure Prolog does come with a price though. If the query p is asked with respect to the program consisting of the only rule $p \leftarrow p$ then the Pure Prolog execution mechanism goes into an infinite loop. This can also happen if the literals in the body of certain rules are not ordered correctly, or if the rule themselves are not ordered correctly. The following examples illustrate this

Example 140 *[SS86]* Consider the following Pure Prolog program that defines appending of lists.

$append([X|XL], YL, [X|ZL]) \leftarrow append(XL, YL, ZL).$
$append([], YL, YL).$

If the query $append(Xlist, [a, b, c], Zlist)$ is asked with respect to the above program then the Pure Prolog execution mechanism will not return any answers (nor will it say $false$). On the other hand if the order of the two rules in the program are swapped and we have the following program:

$append([], YL, YL).$
$append([X|XL], YL, [X|ZL]) \leftarrow append(XL, YL, ZL).$

and the same query is asked then the Pure Prolog execution mechanism will continually return answers, as there are indeed infinite number of answers to this query.

In contrast, from the AnsProlog point of view both programs are identical, as an AnsProlog program is a *set* of rules, and there is no ordering of elements in a set. □

Example 141 Consider a undirected graph whose edges are written using the predicated *edge*. Following is an encoding for a particular graph.

$edge(X, Y) \leftarrow edge(Y, X).$
$edge(a, b).$
$edge(b, c).$
$edge(d, e).$

Now if the query $edge(b, a)$ is asked then the Pure Prolog execution mechanism will get into an infinite loop. On the other hand, if we alter the program by putting the first rule at the end of the program resulting in the following program,

$edge(a, b).$
$edge(b, c).$
$edge(d, e).$
$edge(X, Y) \leftarrow edge(Y, X).$

then the query $edge(b, a)$ will be answered correctly by the Pure Prolog execution mechanism. □

To avoid the problem illustrated by the above example, we will present conditions – referred to as *termination conditions* – that guarantee that no matter how the rules are ordered inside the program, query processing mechanism will terminate. We now illustrate some of the other aspects of the procedural semantics of Pure Prolog and the problems that arise.

Example 142 Consider the following program.

$p \leftarrow$ **not** $q(X).$
$q(1).$
$r(2).$

When the query p is asked with respect to the above program, the Pure Prolog answering mechanism resolves p to **not** $q(X)$. It then tries to answer the query $q(X)$ which resolves to the empty query with the answer substitution $\{X/1\}$. Thus the answer to the query $q(X)$ is $true$ and hence the answer to **not** $q(X)$ is considered to be $false$. Since p does not resolve to any other query, the answer to the query p (with respect to the Pure Prolog semantics) is $false$.

This is unsound with respect to the AnsProlog semantics as the ground version of the above program as given below:

$p \leftarrow$ **not** $q(1).$
$p \leftarrow$ **not** $q(2).$
$q(1).$
$r(2).$

makes it clear that because of the second rule p should be $true$. □

The analysis of what went wrong results in the observation that the handling of negative naf-literals with variables by Pure Prolog is not right. The selection of negative naf-literals with variables is referred to as *floundering*. To eliminate floundering we will suggest conditions that will guarantee that during the LDNF-resolution any naf-literal that is picked does not have any variables.

The following two examples show that Pure Prolog also can not correctly handle many non-stratified programs.

Example 143 Consider the following program.

$p \leftarrow a.$
$p \leftarrow b.$
$a \leftarrow$ **not** $b.$
$b \leftarrow$ **not** $a.$

When the query p is asked with respect to the above program, the Pure Prolog answering mechanism resolves p to a and then to **not** b. It then tries to answer the query b, and resolves b to **not** a. It then tries to answer the query a, and resolves a to **not** b. It gets stuck in this loop. □

Example 144 Consider the following program.

$p \leftarrow$ **not** $p.$
$q.$

When the query p is asked with respect to the above program, the Pure Prolog answering mechanism resolves p to **not** p. It then tries to answer the query p, and resolves p to **not** p. It gets stuck in this loop.

When the query q is asked with respect to the above program, the Pure Prolog answering mechanism resolves q to the empty query and returns *true*. But according to the AnsProlog semantics this program does not have any answer set. □

In the above examples we have illustrated some aspects of LDNF-resolution, the notion of floundering, and non-termination due to ordering of rules in the program. It should be noted that the later two are not part of LDNF-resolution. They are used in the implementation of Pure Prolog and although they are useful in certain circumstances and contribute to an efficient implementation they lead to problems that we described. Before we formally define SLDNF and LDNF resolution we present an algorithm for unification and identify the *occur check* step that is bypassed in the Pure Prolog semantics for the sake of a simpler and efficient implementation, and illustrate the problem caused by it.

8.3.1 A unification algorithm and the occur-check step

In Section 3.10.1 we defined the notion of unification. In this subsection we give an algorithm for unifying two atoms. It is obvious that two atoms are unifiable only if they have the same predicate. While unifying two atoms $p(s_1, \ldots, s_n)$ and $p(t_1, \ldots, t_n)$, $\{s_1 = t_1, \ldots, s_n = t_n\}$ is referred to as the corresponding set of equations, and is often denoted by $p(s_1, \ldots, s_n) = p(t_1, \ldots, t_n)$. A substitution θ such that $s_1\theta = t_1\theta \ldots s_n\theta = t_n\theta$ is called a unifier of the set of equations $\{s_1 = t_1, \ldots, s_n = t_n\}$ and obviously these set of equations and the atoms $p(s_1, \ldots, s_n)$ and $p(t_1, \ldots, t_n)$ have the same unifiers. Two sets of equations are called equivalent if they have the same unifiers and a set of equations is said to be *solved* if it is of the form $\{x_1 = t_1, \ldots, x_n = t_n\}$ where the x_i's are distinct variables and none of the x_i's occur in a term t_j. An mgu θ of a set of equations E is called *relevant* if the variables in θ are a subset of the variables in E.

Lemma 8.3.1 If $E = \{x_1 = t_1, \ldots, x_n = t_n\}$ is solved, then $\theta = \{x_1/t_1, \ldots, x_n/t_n\}$ is a relevant mgu of E. □

The θ in the above lemma is referred to as the *unifier determined by E*. The following algorithm – due to Martelli and Montanari [MM82] – to determine the mgu of two atoms transforms the set of equations corresponding to the two atoms to an equivalent set which is solved, and thus obtains the mgu.

Algorithm 8 Nondeterministically choose from the set of equations an equation of a form below and perform the associated action. If the set of equations do not satisfy any of (1)-(6) then halt.

(1)	$f(s_1, \ldots, s_n) = f(t_1, \ldots, t_n)$	replace by the equations $s_1 = t_1, \ldots, s_n = t_n$
(2)	$f(s_1, \ldots, s_n) = f(t_1, \ldots, t_n)$ where $f \neq g$	halt with failure
(3)	$x = x$	delete the equation
(4)	$t = x$ where t is not a variable	replace by the equation $x = t$
(5)	$x = t$ where x is not the same as t, $\underline{x \text{ does not occur in } t}$, and x occurs else where	perform the substitution $\{x/t\}$ in every other equation
(6)	$x = t$ where x is not the same as t, and x occurs in t	halt with failure

Theorem 8.3.2 If $p(s_1, \ldots, s_n)$ and $p(t_1, \ldots, t_n)$ have a unifier then the above algorithm successfully terminates and produces a solved set of equations determining a relevant mgu; otherwise it terminates with failure. □

The test x does not occur in t, in step (5) of the above algorithm, which we have underlined, is refereed to as *occur check* and most implementation of Prolog omit this check and step (6). Although this omission results in constant time unification (as opposed to linear time) in certain cases, it may lead to incorrect results in other cases. Following is an example of the later.

Example 145 Consider the attempt to unify the atoms $p(y)$ and $p(f(y))$ using the above algorithm minus the occur check. Due to the modified Step (5) the modified algorithm will yield the substitution $\{y/f(y)\}$. This is incorrect as $p(y)$ and $p(f(y))$ are not unifiable. □

Since Pure Prolog inherits this omission of occur check from Prolog, to avoid problems due to occur check we will present sufficiency conditions in Section 8.3.3 that would guarantee that unification is correctly done even in the absence of occur check.

8.3.2 SLDNF and LDNF resolution

We now present the definitions that lead to the formulation of SLDNF and LDNF resolution.

Definition 86 We say that the rule
$p \leftarrow p_1, \ldots, p_k, \textbf{not } p_{k+1}, \ldots, \textbf{not } p_n.$
is a variant of the rule
$q \leftarrow q_1, \ldots, q_k, \textbf{not } q_{k+1}, \ldots, \textbf{not } q_n.$
if there exists substitutions θ and σ such that $p = q\theta$, $q = p\sigma$, $\{p_1, \ldots, p_k\} = \{q_1\theta, \ldots, q_k\theta\}$, $\{p_{k+1}, \ldots, p_n\} = \{q_{k+1}\theta, \ldots, q_n\theta\}$, $\{q_1, \ldots, q_k\} = \{p_1\sigma, \ldots, p_k\sigma\}$, and $\{q_{k+1}, \ldots, q_n\} = \{p_{k+1}\sigma, \ldots, p_n\sigma\}$.
□

Definition 87 A query Q *resolves* to another query Q' via substitution α with respect to Σ, denoted by $Q \stackrel{\alpha}{\Longrightarrow} Q'(\Sigma)$ (also referred to as (α, Q') is a resolvent of Q and Σ), if
either: $\Sigma = (L, R)$, L is a positive literal in Q, R is a rule and for some variant $A \leftarrow E$ (the *input rule*) of R, α is a mgu of L and A and $Q' = Q\alpha\{L\alpha/E\alpha\}$ is obtained from $Q\alpha$ by replacing $L\alpha$ by $E\alpha$,
or: Σ is a negative literal in Q, $\alpha = \epsilon$, and Q' is obtained by removing Σ from Q. □

Consider the program in Example 138. The query $anc(a, b)$ resolves to $par(a, Z), anc(Z, b)$ via substitution $\{X/a, Y/b\}$ with respect to $(anc(a, b), r_2)$.

Similarly consider the program in Example 139. the query $q, \mathbf{not}\ r$ resolves to q via substitution $\{\}$ (also denoted by ϵ) with respect to $\mathbf{not}\ r$.

Definition 88 A rule R is called *applicable* to an atom if the rule has a variant whose head unifies with the atom. □

Definition 89 A (finite or infinite) sequence $Q_0 \stackrel{\alpha_1}{\Longrightarrow} \ldots Q_n \stackrel{\alpha_{n+1}}{\Longrightarrow} Q_{n+1} \ldots$ of resolution steps is a *pseudo derivation* if for every step involving a rule:

- (standardisation apart) the input rule employed does not contain a variable from the initial query Q_0 or from an input rule used at some earlier step, and
- (relevance) the mgu employed is relevant. □

In the above definition the standardisation apart condition is to avoid any confusion with respect to answer substitutions. Thus the input rule used in any step is obtained by using variants of a rule in the program so that the variables in the input rule is a new one.

The notion of pseudo derivation is used to define SLDNF-derivation. Intuitively, an SLDNF-derivation is a pseudo derivation in which the deletion of ground negative literals are justified through finitely failed SLDNF-forests. We now define the notion of finitely failed SLDNF-forests.

Definition 90 A tree is called *successful* if it contains a leaf marked as success, and is called *finitely failed* if it is finite and all its leaves are marked as *failed*. □

Definition 91 A *forest* is a triple $(\mathcal{F}, T, subs)$ where $\mathcal{F}$ is a set of trees, $T \in \mathcal{F}$ and is called the *main* tree, and *subs* is a function assigning to some nodes of trees in $\mathcal{F}$ a tree from $\mathcal{F}$.

A path in $\mathcal{F}$ is a sequence of nodes $N_0, \ldots, N_i, \ldots$ such that for all i, N_{i+1} is either a child of N_i in some tree in $\mathcal{F}$ or the root of the tree $subs(N_i)$. □

Definition 92 A *pre-SLDNF-forest* relative to an AnsProlog program P is a forest whose nodes are queries of literals. The queries may be marked as: $failed$, $success$, or $floundered$, and one literal in each query may be marked as *selected*. The function *subs* assigns to nodes containing a marked negative ground naf-literal $\mathbf{not}\ A$ a tree in $\mathcal{F}$ with root A. □

Definition 93 An *extension* of a pre-SLDNF-forest $\mathcal{F}$ is a forest obtained by marking all empty queries as *success* and performing the following actions for every non-empty query C which is an unmarked leaf in some tree $T \in \mathcal{F}$.

First, if no literal in C is marked as selected, then one of them is marked as selected. Let L be the selected literal of C.

- If L is a positive naf-literal, and

- C has a resolvent with respect to L and some rules from P, then for every rule R from P which is applicable to L, choose one resolvent (α, D) of C with respect to L and R and add this as child of C in T. These resolvents are chosen in such a way that all branches of T remain pseudo derivations.
- otherwise (i.e., C has no resolvents with respect to L and a rule from P) C is marked as failed.

- If $L = \neg A$ is a negative naf-literal, and
  - A is non-ground, then C is marked as floundered.
  - A is ground, and
    - $subs(C)$ is undefined, then a new tree T' with a single node A is added to $\mathcal{F}$ and $subs(C)$ is set to T'.
    - $subs(C)$ is defined and successful, then C is marked as $failed$.
    - $subs(C)$ is defined and finitely failed, then the resolvent $(\epsilon, C - \{L\})$ of C is added as the only child of C in T. □

Definition 94 The set of pre-SLDNF-forests is defined inductively as follows:

1. For every query C, the forest consisting of the main tree which has the single node C is a pre-SLDNF-forest, referred to as an *initial* pre-SLDNF-forest.
2. If $\mathcal{F}$ is a pre-SLDNF-forest, then any extension of $\mathcal{F}$ is a pre-SLDNF-forest. □

Definition 95 SLDNF-forest

- An SLDNF-forest is a limit of a sequence $\mathcal{F}_0, \ldots, \mathcal{F}_i, \ldots$ such that $\mathcal{F}_0$ is an initial pre-SLDNF-forest, and for all i, $\mathcal{F}_{i+1}$ is an extension of $\mathcal{F}_i$.
- An SLDNF-forest for C is an SLDNF-forest $\mathcal{F}$ in which C is the root of the main tree of $\mathcal{F}$.
- A (pre-)SLDNF-forest $\mathcal{F}$ is called successful (resp. finitely failed) if the main tree of $\mathcal{F}$ is successful (resp. finitely failed).
- An SLDNF-forest is called *finite* if no infinite path exist in it. □

Definition 96 A (pre-)SLDNF-derivation for C is a branch in the main tree of a (pre-)SLDNF-forest $\mathcal{F}$ for C together with all trees in $\mathcal{F}$ whose roots can be reached from the nodes in this branch. It is called *successful* if it ends with the empty query. An SLDNF-derivation is called *finite* if all paths of $\mathcal{F}$ fully contained within this branch as these trees is finite. □

Definition 97 Consider a branch in the main tree of a (pre-)SLDNF-forest $\mathcal{F}$ for C which ends with the empty query. Let $\alpha_1, \ldots, \alpha_n$ be the consecutive substitutions along this branch. Then the restriction $(\alpha_1, \ldots, \alpha_n)|C$ of the composition $\alpha_1, \ldots, \alpha_n$ to the variables of C is called a *computed answer substitution* of C in $\mathcal{F}$. □

Theorem 8.3.3 1. Every pre-SLDNF-forest is finite.

2. Every SLDNF-forest is the limit of a unique sequence of pre-SLDNF-forests.

3. If the SLDNF-forest $\mathcal{F}$ is the limit of the sequence $\mathcal{F}_0, \ldots, \mathcal{F}_i, \ldots$, then for all τ

   (a) $\mathcal{F}$ is successful and yields τ as a computed answer substitution iff some $\mathcal{F}_i$ is successful and yields τ as a computed answer substitution, and

   (b) $\mathcal{F}$ is finitely failed iff some $\mathcal{F}_i$ is finitely failed. □

Definition 98 SLDNF-resolution Let P be a Pure Prolog program, and Q be a query.

$P \models_{SLDNF} \forall\ Q\theta$ if there exists a successful SLDNF-derivation for Q (with respect to P) with computed answer θ.

$P \models_{SLDNF} \forall$ **not** Q if there exists a finitely failed SLDNF-forest for Q (with respect to P). □

Definition 99 A query Q is said to flounder with respect to a program P if some SLDNF-forest for Q (with respect to P) contains a node consisting exclusively of non-ground negative naf-literals. □

Definition 100 A program is called *terminating* if all its SLDNF-forests for ground queries are finite. □

8.3.3 Sufficiency conditions

We now present results that specify under what conditions the Pure Prolog semantics agrees with the semantics of AnsProlog. This means under those conditions we can use the Prolog interpreter to correctly make conclusions about query entailment. Recall that pure Prolog semantics is based on (i) SLDNF resolution with the leftmost selection rule (i.e., LDNF resolution) where by only the leftmost literal in each query is marked selected (in the definition of pre-SLDNF-forest in Definition 92 and its extension in Definition 93); (ii) but ignoring floundering; (iii) omitting occur check during unification; and (iv) selecting input rules during resolution from the beginning of the program to the end.

Before presenting results about sufficiency of Pure Prolog let us first consider SLDNF with (ii), (iii) and (iv). In this we use the notion of acyclic programs from Section 3.3.5. We start with conditions that guarantee correctness of $\models_{SLDNF}$.

Proposition 107 Let Π be an acyclic AnsProlog program and G be a variable free atom that does not flounder. Then, $\Pi \models G$ iff $\Pi \models_{SLDNF} G$. □

To allow more general queries than ground queries, we have the following definition.

Definition 101 A literal L is called *bounded with respect to a level mapping* λ on ground literals, if λ is bounded on the set of ground instances $gr(L)$ of L.

A query is called *bounded with respect to a level mapping* λ, if all its literals are. □

Proposition 108 Let Π be an acyclic AnsProlog program and G be a bounded query. Every SLDNF forest of G (with respect to Π) is finite. □

Corollary 8 Every acyclic program is terminating. □

The above result about termination takes care of SLDNF together with (iv) above. To account for the omission of occur-check – condition (iii) and ignoring of floundering – condition (ii), we need the notion of well-moded from Section 3.2.4.

Theorem 8.3.4 If an AnsProlog program Π is well moded for some input-output specification and there is no rule in Π whose head contains more than one occurrence of the same variable in its output positions then Π is occur check free with respect to any ground query. □

Theorem 8.3.5 If an AnsProlog program Π is well moded for some input-output specification and all predicate symbols occurring under **not** are moded completely by input then a ground query to Π does not flounder. □

We now consider Pure Prolog. For termination of Pure Prolog programs we can use a more general notion than acyclic programs, referred to as *acceptable programs.*

Definition 102 Let P be a program in AnsProlog syntax, λ a level mapping for P, and I be a model of P. P is called *acceptable with respect to* λ *and* I if for every rule $A \leftarrow B_1, \ldots, B_n.$ in $ground(P)$ the following holds for $i = 1 \ldots n$:

if $I \models B_1, \ldots, B_{i-1}$ then $\lambda(A) > \lambda(B_i)$.

P is called *acceptable* if it is acceptable with respect to some level mapping and a model of P. □

Example 146 Let us consider the following program π_{anc}.

$par(a, b).$
$par(b, c).$
$anc(X, Y) \leftarrow par(X, Y).$
$anc(X, Y) \leftarrow par(X, Z), anc(Z, Y).$

We will show that π_{anc} is an acceptable program but not an acyclic program.

Let us consider the following level assignment, where all *par* atoms are assigned the level 0, $\lambda(anc(c, a)) = \lambda(anc(c, b)) = \lambda(anc(c, c)) = 2$, $\lambda(anc(b, a)) = \lambda(anc(b, b)) = \lambda(anc(b, c)) = 3$, $\lambda(anc(a, a)) = \lambda(anc(a, b)) = \lambda(anc(a, c)) = 4$, and all other *anc* atoms are assigned the level 1.

It is easy to see that the ground rules corresponding to the first three rules of π_{anc} satisfy the conditions of acyclicity and acceptability. Now let us consider the ground rules corresponding to the fourth rule. They are given below. The first rule below violates the acyclicity condition. To verify the acceptability conditions let us choose $I = \{par(a, b), par(b, c), anc(a, b), anc(b, c), anc(a, c)\}$.

$anc(a, a) \leftarrow par(a, a), anc(a, a).$
$anc(a, b) \leftarrow par(a, a), anc(a, b).$
$anc(a, c) \leftarrow par(a, a), anc(a, c).$
$anc(b, a) \leftarrow par(b, a), anc(a, a).$
$anc(b, b) \leftarrow par(b, a), anc(a, b).$
$anc(b, c) \leftarrow par(b, a), anc(a, c).$
$anc(c, a) \leftarrow par(c, a), anc(a, a).$
$anc(c, b) \leftarrow par(c, a), anc(a, b).$
$anc(c, c) \leftarrow par(c, a), anc(a, c).$

$anc(a,a) \leftarrow par(a,b), anc(b,a).$ (*)
$anc(a,b) \leftarrow par(a,b), anc(b,b).$ (*)
$anc(a,c) \leftarrow par(a,b), anc(b,c).$ (*)
$anc(b,a) \leftarrow par(b,b), anc(b,a).$
$anc(b,b) \leftarrow par(b,b), anc(b,b).$
$anc(b,c) \leftarrow par(b,b), anc(b,c).$
$anc(c,a) \leftarrow par(c,b), anc(b,a).$
$anc(c,b) \leftarrow par(c,b), anc(b,b).$
$anc(c,c) \leftarrow par(c,b), anc(b,c).$

$anc(a,a) \leftarrow par(a,c), anc(c,a).$
$anc(a,b) \leftarrow par(a,c), anc(c,b).$
$anc(a,c) \leftarrow par(a,c), anc(c,c).$
$anc(b,a) \leftarrow par(b,c), anc(c,a).$ (*)
$anc(b,b) \leftarrow par(b,c), anc(c,b).$ (*)
$anc(b,c) \leftarrow par(b,c), anc(c,c).$ (*)
$anc(c,a) \leftarrow par(c,c), anc(c,a).$
$anc(c,b) \leftarrow par(c,c), anc(c,b).$
$anc(c,c) \leftarrow par(c,c), anc(c,c).$

In the above rules, only in the ones marked with (*) the *par* atom in the body is entailed by I, and since $\lambda(anc(a,a)) > \lambda(anc(b,a)) > \lambda(anc(c,a))$, $\lambda(anc(a,b)) > \lambda(anc(b,b)) > \lambda(anc(c,b))$, and $\lambda(anc(a,c)) > \lambda(anc(b,c)) > \lambda(anc(c,c))$, the acceptability conditions are satisfied. Hence this program is acceptable.

It should be noted that if we change the ordering of literals in the fourth rule of π_{anc} to the following,

$anc(X,Y) \leftarrow anc(Z,Y), par(X,Z).$

the program is no longer acceptable; and indeed Prolog gets into an infinite loop when trying to answer a query (about *anc*) with respect to this program. □

Exercise 25 Generalize the π_{anc} program with respect to arbitrary sets of *par* atoms that do not have a cycle and show that such programs are acceptable. □

We now have the following result about termination of programs with respect to the pure Prolog interpreter.

Proposition 109 If Π is an acceptable AnsProlog program and Q is a ground query then all SLDNF derivations – with left most selection – of Q (with respect to Π) are finite and therefore the Pure Prolog interpreter terminates on Q. □

The earlier result accounting for the omission of occur-check in Theorem 8.3.4 and ignoring of floundering in Theorem 8.3.5 are also applicable to the execution mechanism of pure Prolog. *In summary, one way to show that the Pure prolog semantics of a program agrees with the AnsProlog semantics is by showing that the conditions in Theorems 8.3.4 and 8.3.5 and Proposition 109 are satisfied.* In the next section we illustrate this.

8.3.4 Examples of applying pure Prolog sufficiency conditions to programs

Let us consider the programs in Section 5.1.2-5.1.6. We start with program π_1 from Section 5.1.2. As shown there, this program is acyclic and hence is an acceptable program. We now need to check if the conditions in Theorems 8.3.4 and 8.3.5 are satisfied or not.

For this let us consider the mode assignment in Section 5.1.7, which is

$holds(-,+)$
$ab(+,+,+)$

With respect to this mode assignment π_1 is well-moded and it also satisfies the other condition of Theorems 8.3.4. But it does not satisfy the condition 'all predicate symbols occurring under **not** are moded completely by input' of Theorem 8.3.5 as the body of the rules in π_1^{ef} has **not** *holds*.

Let us now consider the program π_2^+ from Section 5.1.3. As shown there, this program is also acyclic and hence is an acceptable program. Now let us consider the mode assignment in Section 5.1.7, which is

$holds(-,+)$
$holds'(-,+)$
$ab(+,+,+)$

With respect to this mode assignment π_1 is well-moded and it also satisfies the other conditions of Theorems 8.3.4 and 8.3.5. *Hence, the Pure Prolog semantics of* π_2^+ *agrees with its AnsProlog semantics.*

Since unlike Smodels and dlv, Prolog can easily manipulate lists, we now consider a list based formulation of π_2, which we will refer to as $\pi_{2.lis}$, and analyze its Pure Prolog characterization vis-a-vis its AnsProlog semantics. The program $\pi_{2.list}$ then consists of the following rules.

1. For every effect proposition of the form (5.1.1) $\pi_{2.list}$ contains the following rule:

   $causes(a, f, [p_1, \ldots, p_n, not(q_1), \ldots, not(q_r)])$.

2. For every value proposition of the form (5.1.5) if f is a fluent then $\pi_{2.list}$ contains the following rule:

   $holds(f, [])$.

   else, if f is the negative fluent literal $\neg g$ then $\pi_{2.list}$ contains the following rule:

   $holds(not(g), [])$.

3. $\pi_{2.list}$ has the following rules to reason about effects and abnormality:

   $holds(F, [A|S]) \leftarrow causes(A, F, L), holds\_list(L, S)$.

   $ab(F, A, S) \leftarrow causes(A, F, L), holds\_list(L, S)$.
   $ab(F, A, S) \leftarrow causes(A, not(F), L), holds\_list(L, S)$.

4. $\pi_{2.list}$ has the following inertia rule:

   $holds(F, [A|S]) \leftarrow holds(F, S), \mathbf{not}\ ab(F, A, S)$.

5. $\pi_{2.list}$ has the following rules for reasoning about lists of fluents.

   $holds\_list([], S)$.
   $holds\_list([F|L], S) \leftarrow holds(F, S), holds\_list(L, S)$.

Let us consider the following mode assignment for $\pi_{2.list}$:

$holds(-, +)$
$holds\_list(-, +)$
$ab(+, +, +)$
$causes(+, -, -)$

With respect to this mode assignment $\pi_{2.list}$ is well-moded and it also satisfies the other conditions of Theorems 8.3.4 and 8.3.5.

Now we will show that $\pi_{2.list}$ is acyclic through the following level mapping.

Let c be the number of fluents in the language plus 1; p be a list of fluent literals, f be a fluent literal, and s be a sequence of actions.

For any action a, $\lambda(a) = 1$, $\lambda([]) = 1$, and for any list $[a|r]$ of actions, $\lambda([a|r]) = \lambda(r) + 1$. For any fluent literal f, $\lambda(f) = 1$, $\lambda([]) = 1$, and for any list $[f|p]$ of fluent literals, $\lambda([f|p]) = \lambda(p) + 1$.

$\lambda(holds\_list(p, s)) = 4c * \lambda(s) + \lambda(p) + 3$;
$\lambda(holds(f, s)) = 4c * \lambda(s) + 3$;
$\lambda(ab(f, a, s)) = 4c * \lambda(s) + 3c + 2$;
and all other literals are mapped to 0.

Now let us consider the rules which have non-empty bodies. These appear in (3), (4) and (5) above.

- For a ground instance of the first rule in (3):

  $\lambda(holds(f, [a|s])) == 4c * \lambda(s) + 4c + 3$.

  The maximum value of $\lambda(holds\_list(p, s))$ will be $4c*\lambda(s)+max(\lambda(p))+3 = 4c*\lambda(s)+c-1+3$

  Obviously, $4c * \lambda(s) + 4c + 3 > 4c * \lambda(s) + c + 2$. Hence, this rule satisfies the acyclicity condition.

- For a ground instance of the second rule in (3):

  $\lambda(ab(f, a, s)) = 4c * \lambda(s) + 3c + 2$.

  Since $c > 1$, $4c * \lambda(s) + 3c + 2 > 4c * \lambda(s) + c + 2$. Hence, this rule satisfies the acyclicity condition.

- For a ground instance of the rule in (4):

  $\lambda(holds(f, [a|s])) = 4c * \lambda(s) + 4c + 3 > \lambda(ab(f, a, s)) = 4c * \lambda(s) + 3c + 2$. Hence, this rule satisfies the acyclicity condition.

- For a ground instance of the second rule in (5):

  $\lambda(holds\_list([f|p], s)) = 4c * \lambda(s) + \lambda(p) + 1 + 3.$

  $\lambda(holds\_list(p, s)) = 4c * \lambda(s) + \lambda(p) + 3.$

  $\lambda(holds(f, s)) = 4c * \lambda(s) + 3.$

  Hence, $\lambda(holds\_list([f|p], s)) > \lambda(holds\_list(p, s))$ and $\lambda(holds\_list([f|p], s)) > \lambda(holds(f, s))$; and therefore this rule satisfies the acyclicity condition.

Hence, the Pure Prolog semantics of $\pi_{2.list}$ agrees with its AnsProlog semantics.

8.4 Notes and references

The Smodels system developed at Helsinki University of Technology is described in [NS96, NS97] and the Smodels web pages. Our presentation in this chapter is based on the lparse manual, [Sim00] and our experience with Smodels. The dlv system developed at TU Vienna and is described in [EFG+00, KL99, LRS96a, LRS96b] and the dlv manuals and tutorials available in the dlv web pages. Starting from [CM81], there are several good books such as [SS86, O'K90, Ste90] on the Prolog programming language and its applications. The notion of Pure Prolog that we use here is from [SS86]. The definition of SLDNF resolution that we presented is from [AB94, AD94], where it is also pointed out that most earlier definitions of SLDNF resolution do not allow correct reasoning about termination. The sufficiency conditions for correctness of Pure Prolog are from [AP94, AP93]. Our example in Section 8.3.4 is based on [BGP97].

Among the other systems that have similarity with the ones we presented in this chapter are DeRes [Tru99, CMT96, CMMT95], Ccalc [Lif95], XSB [CSW95] and LDL^{++} [WZ00a, Zan88]. The DeRes system generates extensions of Reiter's default theories. The Ccalc system implements a causal logic by translating it to propositional logic and then using propositional solvers. The XSB system is a top down system like Prolog but it uses tabling mechanism to correctly handle some of the queries that send Prolog interpreters to infinite loops. It also has options to compute the well-founded semantics and answer set semantics. The LDL^{++} system has an efficient implementation of a database query language that extends AnsDatalog with aggregate operators.

In the database community, there has been a lot of research on techniques to answer queries in a focused manner – as done in top-down query processing systems, but using the bottom-up approach this avoiding the non-termination issues. The technique that is used is referred to as 'magic sets' or 'magic templates' [Ull88a, Ull88b, BR86, BR87]. The basic idea behind this is to take a query binding pattern and transform a given program so that the bottom-up query processing with respect to the transformed program and queries that follow the initial binding pattern is as focused as it would have been with respect to top-down query processing. Such techniques have only been explored with respect to stratified programs. How to extend it beyond stratified programs, in particular with respect to programs that have multiple answer sets, remains an open question.

Chapter 9

Further extensions of and alternatives to AnsProlog*

In Chapter 8 we discussed two extensions of AnsProlog that are implemented in the Smodels and dlv systems. In this chapter we consider several additional proposals for extending AnsProlog* and also alternative semantics of programs in the syntax of AnsProlog*.

Some of the extensions that we consider are: (i) enriching the rules by allowing **not** in the head of rules, (ii) allowing nesting in the head and body of rules; (iii) allowing additional operators such as knowledge and belief operators; and (iv) enriching the framework by allowing abduction.

We consider the well-founded semantics of AnsProlog and its various characterizations. We then consider several alternative semantics – based on the well-founded semantics – of other AnsProlog* sub-classes. Finally we address approaches and alternative semantics to counter the so called universal query problem that arises due to the in-built Herbrand assumption in most characterizations of programs with AnsProlog* syntax.

9.1 AnsProlog$^{\mathbf{not}\ ,\ or\ ,\neg,\bot}$: allowing not in the head

The AnsProlog* programs do not allow **not** in their head, although they do allow $\neg$ in the head. But the AnsProlog$_{sm}$ programs allow cardinality and weight constraints in their head, and these constraints may contain **not** . In this section we consider extending AnsProlog* programs that allow **not** in their head. Such programs will be referred to as AnsProlog$^{\mathbf{not}\ ,\ or\ ,\neg,\bot}$ programs. These programs will form a first step in analyzing strong equivalence of AnsProlog$_{sm}$ programs which we will discuss in a later section.

An AnsProlog$^{\mathbf{not}\ ,\ or\ ,\neg,\bot}$ rule is of the form

$$L_0\ or\ \ldots\ or\ L_k\ or\ \mathbf{not}\ L_{k+1}\ or\ \ldots\ or\ \mathbf{not}\ L_l \leftarrow L_{l+1}, \ldots, L_m, \mathbf{not}\ L_{m+1}, \ldots, \mathbf{not}\ L_n \qquad (9.1.1)$$

where L_i's are literals or when $l = k = 0$, L_0 may be the symbol $\bot$, and $n \geq m \geq l \geq k \geq 0$. It differs from an AnsProlog* rule by allowing **not** in the head of the rules. An AnsProlog$^{\mathbf{not}\ ,\ or\ ,\neg,\bot}$ program is a collection of AnsProlog$^{\mathbf{not}\ ,\ or\ ,\neg,\bot}$ rules.

Let Π be an AnsProlog$^{\mathbf{not}\ ,\ or\ ,\neg,\bot}$ program and S be a set of literals. The reduct Π^S of Π by S is an AnsProlog$^{\neg\mathbf{not}\ ,\neg,\ or}$ program obtained as follows: A rule

$L_0\ or\ \ldots\ or\ L_k \leftarrow L_{l+1}, \ldots, L_m.$

is in Π^S iff there is a ground rule of the form (9.1.1) in Π such that $\{L_{k+1}, \ldots, L_l\} \subseteq S$ and $\{L_{m+1}, \ldots, L_n\} \cap S = \emptyset$.

The intuition behind this construction is as follows: *First*, rules of the form (9.1.1) in Π are not considered for Π^S if either (i) at least one of $\{L_{m+1}, \ldots, L_n\}$ is in S, or (ii) at least one of $\{L_{k+1}, \ldots, L_l\}$ is not in S. In the first case the body of that rule will evaluate to false and hence that rule will not make any contribution to the answer set. In the second case one of the disjuncts in the head is already true with respect to S, so no new disjuncts in the head need to be made true. *Next*, we remove literals following **not** in the body and head of the remaining rules in Π and put them in Π^S. This is because the ones in the body are already true with respect to S and the ones in the head are already false with respect to S.

To define answer sets of AnsProlog$^{\mathbf{not}, or, \neg, \bot}$ programs, recall that we have already defined – in Section 1.3.4 – answer sets for programs of the form Π^S. We now say S is an answer set of an AnsProlog$^{\mathbf{not}, or, \neg, \bot}$ program Π iff S is an answer set of Π^S.

Example 147 *[IS98]* Consider the following AnsProlog$^{\mathbf{not}, or, \neg, \bot}$ program π:

$r \leftarrow p.$
p *or* **not** $p \leftarrow.$

We will show that $S_1 = \emptyset$ and $S_2 = \{p, r\}$ are two answer sets of this program.

$\pi^{S_1} = \{r \leftarrow\}$, and the answer set of π^{S_1} is $\emptyset$.

$\pi^{S_2} = \{r \leftarrow; p \leftarrow\}$, and the answer set of π^{S_2} is $\{p, r\}$. □

On important feature of answer sets of AnsProlog$^{\mathbf{not}, or, \neg, \bot}$ programs that is evident from the above example is that they no longer have to be minimal. This is a departure from the answer sets of AnsProlog* programs.

There exist a mapping that can translate AnsProlog$^{\mathbf{not}, or, \neg, \bot}$ programs to AnsProlog* programs so that there is a one-to-one correspondence between their answer sets. But the answer sets are not exactly the same, as they may contain some additional literals that are introduced during the translation process. The translation is as follows:

For every rule of the form (9.1.1) in the AnsProlog$^{\mathbf{not}, or, \neg, \bot}$ program π, the translated program $tr(\pi)$ contains the following rules.

1. r_0 *or* $\ldots$ *or* r_k *or* r_{k+1} *or* $\ldots$ *or* $r_l \leftarrow L_{l+1}, \ldots, L_m, \mathbf{not}\ L_{m+1}, \ldots, \mathbf{not}\ L_n$
2. $L_i \leftarrow r_i.$ for $i = 0 \ldots k$
3. $r_i \leftarrow L_i, L_{k+1}, \ldots, L_l.$ for $i = 0 \ldots k$
4. $\bot \leftarrow r_i, \mathbf{not}\ L_j.$ for $i = 0 \ldots k$ and $j = k+1 \ldots l.$
5. $\bot \leftarrow r_j, L_j.$ for $j = k+1 \ldots l$

Proposition 110 Let π be an AnsProlog$^{\mathbf{not}, or, \neg, \bot}$ program and $tr(\pi)$ be its translation. A set S is an answer set of π iff S' is an answer set of $tr(\pi)$ such that $S = S' \cap Lit_\pi$. □

Example 148 Consider the following AnsProlog$^{\mathbf{not}\ ,\ or\ ,\neg,\bot}$ program π_1:

p *or* **not** $q \leftarrow$.
q *or* **not** $p \leftarrow$.

The answer sets of π_1 are $\{p, q\}$ and $\emptyset$.

The translation π_1 is as follows:

r_1 *or* $r_2 \leftarrow$.
r_3 *or* $r_4 \leftarrow$.
$p \leftarrow r_1$.
$r_1 \leftarrow p, q$.
$\bot \leftarrow r_1$, **not** q.
$\bot \leftarrow r_2, q$.
$q \leftarrow r_3$.
$r_3 \leftarrow q, p$.
$\bot \leftarrow r_3$, **not** p.
$\bot \leftarrow r_4, p$.

It has the answer sets $\{r_1, r_3, p, q\}$ and $\{r_2, r_4\}$. □

We now define a special class of AnsProlog$^{\mathbf{not}\ ,\ or\ ,\neg,\bot}$ programs for which we can do a simpler translation such that the answer sets of the two programs coincide.

An AnsProlog$^{\mathbf{not}\ ,\ or\ ,\neg,\bot}$ program is negative acyclic if there is a level mapping l for Π such that:
(i) for every $i = 0 \ldots k$, and $j = k+1 \ldots l$, $l(L_i) > l(L_j)$; and
(i) for every $i = 0 \ldots k$, and $j = l+1 \ldots m$, $l(L_i) \geq l(L_j)$.

Proposition 111 Let π be a negative acyclic AnsProlog$^{\mathbf{not}\ ,\ or\ ,\neg,\bot}$ program. Let $tr_2(\pi)$ be the AnsProlog program obtained from π by replacing every rule of the form (9.1.1) by the rule:

L_0 *or* $\ldots$ *or* $L_k \leftarrow L_{k+1}, \ldots, L_l, L_{l+1}, \ldots, L_m$, **not** $L_{m+1}, \ldots,$ **not** L_n.

Both π and $tr_2(\pi)$ have the exact same answer sets. □

9.2 AnsProlog$^{\{\mathbf{not}\ ,\ or\ ,\neg,\bot\}^*}$: allowing nested expressions

The nesting of operators in a logic programming setting started off with Prolog implementations where rules of the form

$s \leftarrow (p \rightarrow q; r), t$.

were allowed. The construct $(p \rightarrow q; r)$ represented the 'if-then-else' construct and meant *if* p *then* q *else* r. Using the operators in our language this rule will be expressed as

$s \leftarrow ((p, q)$ *or* $(\mathbf{not}\ p, r)), t$.

which has the equivalent (intuitive) meaning as the following two rules:

$s \leftarrow p, q, t$.
$s \leftarrow \mathbf{not}\ p, r, t$.

The Prolog implementation is more efficient with respect to the nested rule than with respect to the above two rules as in the former case it evaluates p only once. Another motivation behind

allowing nesting is the resulting compactness in representation, and a notion of equivalence that makes it easier to reason about strong equivalence of AnsProlog$_{sm}$ programs. In this section we will introduce the language AnsProlog$^{\{\mathbf{not}\ ,\ or\ ,\neg,\bot\}^*}$ that allows nesting, give its semantics, discuss its usefulness, and present a translation from AnsProlog$^{\{\mathbf{not}\ ,\ or\ ,\neg,\bot\}^*}$ to AnsProlog$^{\mathbf{not}\ ,\ or\ ,\neg,\bot}$. We will focus on the propositional case, as the programs with variables can be grounded to eliminate the variables.

We now start with several notions that are specific to this section only. Elementary formulas are literals and the 0-place connectives $\bot$ and $\top$, representing $false$ and $true$ respectively. *Formulas* are built from elementary formulas using the unary connective **not** , and the binary connectives , (conjunction) and *or* (disjunction). Often we will use the symbol ; for *or* . A *rule* is an expression of the form:

$F \leftarrow G$

where F and G are formulas referred to as the *head* and *body* of the rule respectively. Often we will use the short-hand $F \rightarrow G; H$ for $(F, G); (\mathbf{not}\ F, H)$. The rule $F \leftarrow \top$ will often be written as $F \leftarrow$ or simply F. Rules of the form $\bot \leftarrow G$ will be referred to as *constraints* and written as $\leftarrow G$. An AnsProlog$^{\{\mathbf{not}\ ,\ or\ ,\neg,\bot\}^*}$ program is a set of rules. An occurrence of a formula F in another formula or rule is said to be *singular* if the symbol before F in this occurrence is $\neg$; otherwise the occurrence is referred to as *regular*. It should be noted that in formulas the status of **not** and $\neg$ are different in the sense that $\neg$ can only precede an atom, while **not** can precede an arbitrary formula.

Formulas, rules and programs that do not contain **not** will be called *basic*. A consistent set X of literals is said to satisfy a basic formula F denoted by $X \models F$ if:

- F is an elementary formula, and $F \in X$ or $F = \top$.
- F is the formula (G, H), and $X \models G$ and $X \models H$.
- F is the formula $(G; H)$, and $X \models G$ or $X \models H$.

A consistent set X of literals is said to be *closed* under a basic program Π if, for every rule $F \leftarrow G$ in Π, $X \models F$ whenever $X \models G$. X is said to be an answer set of Π if X is minimal among the consistent set of literals closed under Π.

Definition 103 The *reduct* of a formula, rule or program relative to a consistent set X of literals, denoted by putting X as a superscript, is recursively defined as follows:

- If F is an elementary formula then $F^X = F$.
- $(F, G)^X = (F^X, G^X)$.
- $(F; G)^X = (F^X; G^X)$.
- $(\mathbf{not}\ F)^X$ is equal to $\bot$ if $X \models F^X$, and $\top$ otherwise.
- $(F \leftarrow G)^X = (F^X \leftarrow G^X)$.
- For a program Π, $\Pi^X = \{(F \leftarrow G)^X : F \leftarrow G \in \Pi\}$.

□

Notice that for a program Π and a consistent set X of literals, the program Π^X is *basic*.

Definition 104 A consistent set X of literals is an *answer set* for a program Π if X is an answer set of Π^X. □

Proposition 112 Let Π be an AnsProlog$^{\mathbf{not}\ ,\ or\ ,\neg,\bot}$ program. Its answer sets corresponding to Definition 104 are its consistent answer sets with respect to the definition in Section 9.1. □

In contrast to the rest of the book, the definition of answer set in this section precludes the possibility of inconsistent answer sets.

Example 149 Consider the following AnsProlog$^{\neg}$ program:

$p \leftarrow$.
$\neg p \leftarrow$.
$q \leftarrow$.

This program has the unique answer set *Lit*. But according to Definition 104, the corresponding AnsProlog$^{\{\mathbf{not}\ ,\ or\ ,\neg,\bot\}^*}$ program does not have any answer sets. □

The notion of strong equivalence in Section 3.10.5 can be transported to AnsProlog$^{\{\mathbf{not}\ ,\ or\ ,\neg,\bot\}^*}$ programs. We will discuss transformations that preserve strong equivalence of AnsProlog$^{\{\mathbf{not}\ ,\ or\ ,\neg,\bot\}^*}$ programs. We first define equivalence of formulas and transformations that preserve their equivalence.

Definition 105 Two formulas F and G are said to be equivalent, denoted by $F \Leftrightarrow H$, if for any consistent sets of literals X and Y, $X \models F^Y$ iff $X \models G^Y$. □

Proposition 113 For any formulas, F, G and H,

1. $F, G \Leftrightarrow G, F$ and $F; G \Leftrightarrow G; F$.
2. $(F, G), H \Leftrightarrow F, (G, H)$ and $(F; G); H \Leftrightarrow F; (G; H)$.
3. $F, (G; H) \Leftrightarrow (F, G); (F, H)$ and $F; (G, H) \Leftrightarrow (F; G), (F; H)$.
4. **not** $(F, G) \Leftrightarrow$ **not** F; **not** G and **not** $(F; G) \Leftrightarrow$ **not** F, **not** G.
5. **not not not** $F \Leftrightarrow$ **not** F.
6. $F, \top \Leftrightarrow F$ and $F; \top \Leftrightarrow \top$.
7. $F, \bot \Leftrightarrow \bot$ and $F; \bot \Leftrightarrow F$.
8. If p is an atoms then $p, \neg p \Leftrightarrow \bot$ and **not** p; **not** $\neg p \Leftrightarrow \top$.
9. **not** $\top \Leftrightarrow \bot$ and **not** $\bot \Leftrightarrow \top$.

□

The equivalence of formulas can be used to show the strong equivalence of programs. The following proposition states the necessary conditions.

Proposition 114 Let Π be a AnsProlog$^{\{\mathbf{not}\ ,\ or\ ,\neg,\bot\}^*}$ program, and let F and G be a pair of equivalent formulas. Any program obtained from Π by replacing some regular occurrences of F by G is strongly equivalent to Π. □

Following are some additional strong equivalence conditions for AnsProlog$^{\{\mathbf{not}\ ,\ or\ ,\neg,\bot\}^*}$ programs.

Proposition 115 1. $F, G \leftarrow H$ is strongly equivalent to

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

2. $F \leftarrow G; H$ is strongly equivalent to

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

3. $F \leftarrow G, \mathbf{not\ not}\ H$ is strongly equivalent to $F; \mathbf{not}\ H \leftarrow G$.

4. $F; \mathbf{not\ not}\ G \leftarrow H$ is strongly equivalent to $F \leftarrow \mathbf{not}\ G, H$.

□

Proposition 116 For every AnsProlog$^{\{\mathbf{not}\ ,\ or\ ,\neg,\bot\}^*}$ program there is a strongly equivalent program consisting of rules in the syntax of AnsProlog$^{\mathbf{not}\ ,\ or\ ,\neg,\bot}$. □

Note that in AnsProlog$^{\{\mathbf{not}\ ,\ or\ ,\neg,\bot\}^*}$ **not not** do not cancel out. For example the following two programs have different answer sets.

The program

$p \leftarrow \mathbf{not\ not}\ p$

has the two answer sets $\emptyset$ and $\{p\}$, while the program

$p \leftarrow p$

has the single answer set $\emptyset$. This example also show that AnsProlog$^{\{\mathbf{not}\ ,\ or\ ,\neg,\bot\}^*}$ programs may have non-minimal answer sets.

We now present a translation from AnsProlog$_{sm}$ programs to AnsProlog$^{\{\mathbf{not}\ ,\ or\ ,\neg,\bot\}^*}$ programs so that they have the same answer sets. The transformation is as follows:

1. The translation of a constraint of the form

$L \leq [c_1 = w_1, \ldots, c_n = w_n]$

is the nested expression

$$\overset{\cdot}{;}_{X : \Sigma X \geq L} \left(\overset{,}{}_{i \in X} c_i \right)$$

where X ranges over the subsets of $\{1, \ldots, m\}$ and ΣX stands for $\Sigma_{i \in X} w_i$. The translation of $L \leq S$ is denoted by $[[L \leq S]]$. In the above notation the empty conjunction is understood as $\top$ and the empty disjunction as $\bot$.

2. The translation of a constraint of the form

   $[c_1 = w_1, \ldots, c_n = w_n] \leq U$

   is the nested expression

   $$\underset{X : \Sigma X > U}{\overset{,}{}} \quad (\quad \underset{i \in X}{\overset{;}{}} \quad \textbf{not } c_i \ )$$

   where ΣX is as defined before. The translation of $S \leq U$ is denoted by $[[S \leq U]]$.

3. The translation of a general weight constraint $L \leq S \leq U$ denoted by $[[L \leq S \leq U]]$ is the nested expression $[[L \leq S]], [[S \leq U]]$.

4. For any AnsProlog$_{sm}$ program Ω, its translation $[[\Omega]]$ is the AnsProlog$^{\{\textbf{not}\ ,\ or\ ,\neg,\bot\}^*}$ program obtained from Ω by replacing each AnsProlog$_{sm}$ rule of the form

   $L \leq [l_1 = w_1, \ldots, l_m = w_m] \leq U \leftarrow C_1, \ldots, C_m$

   by

   $(l_1; \textbf{not } l_1), \ldots, (l_m; \textbf{not } l_m), [[L \leq [l_1 = w_1, \ldots, l_m = w_m] \leq U]] \leftarrow [[C_1]], \ldots, [[C_m]]$

The following example illustrates the above translation.

Example 150 Consider the AnsProlog$_{sm}$ program consisting of the following rule:

$2 \leq [a = 1, b = 1, c = 1] \leq 2 \leftarrow p.$

We translate the above program to an AnsProlog$^{\{\textbf{not}\ ,\ or\ ,\neg,\bot\}^*}$ program as follows:

$[[2 \leq [a = 1, b = 1, c = 1]]]$ is the nested expression

$(a, b); (b, c); (a, c); (a, b, c).$

$[[[a = 1, b = 1, c = 1] \leq 2]]$ is the nested expression

$(\textbf{not } a; \textbf{not } b; \textbf{not } c).$

Thus $[[2 \leq [a = 1, b = 1, c = 1] \leq 2 \leftarrow p]]$ is the program consisting of the following rule:

$(a; \textbf{not } a), (b; \textbf{not } b), (c; \textbf{not } c), ((a, b); (b, c); (a, c); (a, b, c)), (\textbf{not } a; \textbf{not } b; \textbf{not } c) \leftarrow p$ □

Theorem 9.2.1 For any AnsProlog$_{sm}$ program Ω, the programs Ω and $[[\Omega]]$ have the same answer sets. □

It should be noted that the translation $[[\ ]]$ in general results in an exponential blow-up. Nevertheless, when L and U are either infinite or bounded by a fixed constant, which is often the case, the resulting translation is of reasonable size. One advantage of this translation over the translation in Section 2.1.18 is that in case of the later the correspondence between the answer sets is not 1-1 and onto, and new literals not in the language of the original program are introduced. On the other hand the translation in Section 2.1.18 is more succinct, in particular is linear with respect to the size of the original program.

An important advantage of the translation mechanism of this section is that we can use the strong equivalence results of AnsProlog$^{\{\textbf{not}\ ,\ or\ ,\neg,\bot\}^*}$ programs to show the strong equivalence of AnsProlog$_{sm}$ programs.

9.3 AnsProlog$^{\neg, or, K, M}$: allowing knowledge and belief operators

Among the various AnsProlog* subsets and extensions that we considered so far, there are only two forms of negation, the classical $\neg$, and the non-monotonic **not** . Although the answer to a query with respect to an AnsProlog$^{\neg, or}$ program is *true* if it is *true* in all its answer sets, there is no way to reason within the language about a particular literal being *true* in all the (or some of the) answer sets.

The following example demonstrates the need for an extension of AnsProlog$^{\neg, or}$ that will allow such reasoning:

Example 151 Consider the following information. We know that (A) Either "john" or "peter" is guilty (of murder). (B) " a person is presumed innocent if (s)he cannot be proven to be guilty". (C) "a person can get a security clearance if we have no reason to suspect that (s)he is guilty."

Statement (A) can easily be written as an AnsProlog$^{\neg, or}$ rule:

$$A_1: \ guilty(john) \ or \ \ guilty(peter) \leftarrow .$$

If we try to write statement (B) as an AnsProlog$^{\neg, or}$ rule, we have:

$$B_1: \ presumed\_innocent(X) \leftarrow \textbf{not} \ \ guilty(X).$$

This however is not appropriate because the program consisting of A_1 and B_1 has two answer sets $\{guilty(john), presumed\_innocent(peter)\}$ and $\{guilty(peter), presumed\_innocent(john)\}$, and therefore $presumed\_innocent(john)$ is inferred to be *unknown*. Intuitively, we should be able to infer that $presumed\_innocent(john)$ is *true*. Hence, the operator **not** in the body of B_1 is not the one we want.

Similarly, if we consider representing statement C in the language AnsProlog$^{\neg, or}$ programs, we have :

$$C_1: \ cleared(X) \leftarrow \textbf{not} \ \ guilty(X).$$

But, C_1 is not appropriate because the program consisting of A_1 and C_1 has two answer sets: $\{guilty(john), cleared(peter)\}$ and $\{guilty(peter), cleared(john)\}$, and we infer $cleared(john)$ to be *unknown*. Intuitively, we would like to infer that $cleared(john)$ is $false$.

Our goal is to expand the language and redefine answer sets in such a way that: (B_2) We would infer $presumed\_innocent(a)$ iff there is at least one answer set that does not contain $guilty(a)$. (C_2) We would infer $cleared(a)$ iff none of the answer sets contain $guilty(a)$. □

To capture the intuition in (B_2) and (C_2) in the above example, we use two unary operators K and M [Gel91b] and add them to our language. Intuitively, KL stands for L is *known* and ML stands for L *may be* believed. For a literal L, and a collection of sets of literals S, we say that KL is *true* with respect to S ($S \models KL$) iff L is *true* in *all* sets in S. ML is *true* with respect to S ($S \models ML$) iff l is *true* in *at least one* set in S. We say $S \models \neg KL$ iff $S \not\models KL$ and we say $S \models \neg ML$ iff $S \not\models ML$. This means $\neg KL$ is *true* with respect to S iff there is at least one set in S where L is not *true*, and $\neg ML$ is *true* with respect to S iff there is *no* set in S where L is *true*.

Using K and M we can represent the statements (B) and (C) in the above example by the rules:

$$innocent(X) \leftarrow \neg K guilty(X)$$

and

$$cleared(X) \leftarrow \neg M guilty(X)$$

We now define the syntax and semantics of AnsProlog$^{\neg,\ or\ ,K,M}$ programs which are obtained by adding K and M to AnsProlog$^{\neg,\ or}$. We refer to a literal L (without K or M) as an *objective* literal, and we refer to formulas of the form KL, ML, $\neg KL$ and $\neg ML$ as *subjective* literals.

An AnsProlog$^{\neg,\ or\ ,K,M}$ logic program is a collection of rules of the form:

$$L_1\ or\ \ldots\ or\ L_k \leftarrow G_{k+1}, \ldots, G_m, \textbf{not}\ \ L_{m+1}, \ldots, \textbf{not}\ \ L_n \tag{9.3.2}$$

where the L's are objective literals and the G's are subjective or objective literals.

Let T be an AnsProlog$^{\neg,\ or\ ,K,M}$ program and S be a collection of sets of literals in the language of T. By T^S we will denote the AnsProlog$^{\neg,\ or}$ program obtained from T by:

1. removing from T all rules containing subjective literals G such that $S \not\models G$,
2. removing from rules in T all other occurrences of subjective literals.

Definition 106 A set S will be called a *world view* of T if S is the collection of all answer sets of T^S. Elements of S will be called *belief sets* of T. The program T^S will be called the *reduct* of T w.r.t. S. □

We now limit ourselves to AnsProlog$^{\neg,\ or\ ,K,M}$ programs with a unique world view.

An objective literal is said to be $true(false)$ with respect to an AnsProlog$^{\neg,\ or\ ,K,M}$ program if it is $true(false)$ in all elements of its world view; otherwise it is said to be unknown. A subjective literal is said to be $true(false)$ with respect to an AnsProlog$^{\neg,\ or\ ,K,M}$ program if it is $true(false)$ in its world view. Notice that subjective literals can not be unknown.

Example 152 Consider the AnsProlog$^{\neg,\ or\ ,K,M}$ program T_1:

1. $guilty(john)\ or\ \ guilty(peter) \leftarrow$.
2. $presumed\_innocent(X) \leftarrow \neg K guilty(X)$.
3. $cleared(X) \leftarrow \neg M guilty(X)$.
4. $\neg presumed\_innocent(X) \leftarrow \textbf{not}\ \ presumed\_innocent(X)$.
5. $\neg\ cleared(X) \leftarrow \textbf{not}\ \ cleared(X)$.

Let $S_1 = \{guilty(john), presumed\_innocent(john), presumed\_innocent(peter),$
$\neg cleared(john), \neg cleared(peter)\}$

and $S_2 = \{guilty(peter), presumed\_innocent(john), presumed\_innocent(peter),$
$\neg cleared(john), \neg cleared(peter)\}$

and $S = \{S_1, S_2\}$

Since, $M guilty(john)$ and $M guilty(peter)$ are both $true$ with respect to S,
$S \not\models \neg M guilty(john)$ and $S \not\models \neg M guilty(peter)$, and therefore, ${T_1}^S$ does not contain any ground

instance of rule 3. Similarly, $S \models \neg K guilty(john)$ and $S \models \neg K guilty(peter)$ and hence ${T_1}^S$ consists of the rules:

$guilty(john)$ *or* $guilty(peter) \leftarrow$.
$presumed\_innocent(john) \leftarrow$.
$presumed\_innocent(peter) \leftarrow$.
$\neg presumed\_innocent(X) \leftarrow$ **not** $presumed\_innocent(X)$.
$\neg\ cleared(X) \leftarrow$ **not** $cleared(X)$.

The answer sets of ${T_1}^S$ are S_1 and S_2. Hence, S is a world view of T_1. It is possible to show that S is the only world view of T_1 [GP91] and therefore $T_1 \models presumed\_innocent(john)$, $T_1 \models presumed\_innocent(peter)$, $T_1 \models \neg cleared(john)$ and $T_1 \models \neg cleared(peter)$ which corresponds to our specification. □

Example 153 *[Representing Unknown]* Consider the AnsProlog$^{\neg,\ or}$ program from Section 2.2.4. Recall that it consists of rules used by a certain college for awarding scholarships to its students, and a rule saying "if the three rules do not determine the eligibility of a student then (s)he should be interviewed."

In the formulation in Section 2.2.4 the above is encoded using following rule:

$interview(X) \leftarrow$ **not** $eligible(X)$, **not** $\neg eligible(X)$

We now argue that in presence of multiple answer sets the above rule is not appropriate. Assume that, in addition to the earlier formulation, we have the following additional disjunctive information.

5. $fairGPA(mike)$ *or* $highGPA(mike) \leftarrow$

The AnsProlog$^{\neg,\ or\ ,K,M}$ program $\pi_{gpa.1}$ consisting of (1) - (4) from π_{gpa} and (5), has two answer sets: $A_1 = \{highGPA(mike), eligible(mike)\}$ and $A_2 = \{fairGPA(mike), interview(mike)\}$, and therefore the reasoner modeled by $\pi_{gpa.1}$ does not have enough information to establish Mike's eligibility for the scholarship (i.e. answer to $eligible(mike)$ is *unknown*). Hence, intuitively the reasoner should answer *yes* to the query $interview(mike)$. But this is not achieved by the above representation.

The intended effect is achieved by replacing (4) by the following rule:

4′. $interview(X) \leftarrow \neg K\, eligible(X),\ \neg K \neg eligible(X)$

The AnsProlog$^{\neg,\ or\ ,K,M}$ program, $\pi_{gpa.epi}$, obtained by replacing (4) in $\pi_{gpa.1}$ by (4') has the world view $A = \{A_1, A_2\}$ where $A_1 = \{highGPA(mike), eligible(mike), interview(mike)\}$, and $A_2 = \{fairGPA(mike), interview(mike)\}$. Hence, $\pi_{gpa.epi}$ answers *unknown* to the query $eligible(mike)$ and *yes* to the query $interview(mike)$, which is the intended behavior of the system. □

Hence, in general (for theories with multiple answer sets), the statement "the truth of an atomic statement P is *unknown*" is appropriately represented by

$$\textbf{not}\ \ KP, \textbf{not}\ \ \neg KP. \tag{9.3.3}$$

So far we only considered AnsProlog$^{\neg,\ or\ ,K,M}$ programs with a unique world view. The following example shows AnsProlog$^{\neg,\ or\ ,K,M}$ programs may have multiple world views.

Example 154 Let T_2 consist of the rules

1. $p(a)\ or\ \ p(b) \leftarrow$
2. $p(c) \leftarrow$
3. $q(d) \leftarrow$
4. $\neg p(X) \leftarrow \neg Mp(X)$

The specification T_2 has three world views:

$A_1 = \{\{q(d),\ p(c),\ p(a),\ \neg p(b),\ \neg p(d)\}\}$,

$A_2 = \{\{q(d),\ p(c),\ p(b),\ \neg p(a),\ \neg p(d)\}\}$, and

$A_3 = \{\ \{q(d),\ p(a),\ p(c),\ \neg p(d)\},\ \{q(d),\ p(b),\ p(c),\ \neg p(d)\}\}$.

Intuitively A_3 is preferable to the other two world views of T_2 as it treats $p(a)$ and $p(b)$ in the same manner (unlike A_1 and A_2) and can be used to answer queries with respect to T_2. □

Exercise 26 Formulate a preference relation between world-views of AnsProlog$^{\neg,\ or\ ,K,M}$ programs, and develop conditions on the programs that guarantee unique preferred world-views. (Hint: [Gel91b].) □

9.4 Abductive reasoning with AnsProlog: AnsProlog^{abd}

Earlier in Section 3.9 we discussed a simple form of abduction using AnsProlog*. In this section we consider a notion of abductive logic programming which is more general than our earlier formulation in some aspects and less general in certain others. In this formulation a subset of the predicates in the language are referred to as the *abducible* predicates or *open predicates*. An AnsProlog^{abd} program is defined as a triple $\langle \Pi, A, O\rangle$, where A is the set of open predicates, Π is an AnsProlog program with only atoms of non-open predicates in its heads and O is a set of first order formulas. O is used to express *observations* and *constraints* in an abductive logic program. Abductive logic programs are characterized as follows:

Definition 107 Let $\langle \Pi, A, O\rangle$ be an abductive logic program. A set M of ground atoms is a *generalized stable model* of $\langle \Pi, A, O\rangle$ if there is a $\Delta \subset atoms(A)$ such that M is an answer set of $\Pi \cup \Delta$ and M satisfies O.

For an atom f, we say $\langle \Pi, A, O\rangle \models_{abd} f$, if f belongs to all generalized stable models of $\langle \Pi, A, O\rangle$. For a negative literal $\neg f$, we say $\langle \Pi, A, O\rangle \models_{abd} \neg f$, if f does not belong to any of the generalized stable models of $\langle \Pi, A, O\rangle$. □

9.5 Domain closure and the universal query problem

Consider the AnsProlog program Π consisting of the following rule:

$p(a) \leftarrow$.

Suppose we would like to ask if $\Pi \models \forall X.p(X)$. Since this query is not part of our query language presented in Section 1.3.5, let us use the technique in Section 2.1.7 and modify the program Π to Π^* which consists of the following rules:

$p(a) \leftarrow$.
$not\_all\_p \leftarrow$ **not** $p(X)$.
$all\_p \leftarrow$ **not** not_all_p.

and ask if $\Pi^* \models all\_p$. Since, when the language of a program is not explicitly stated, its language is inferred from the program itself, for this program the Herbrand Universe is $\{a\}$. Hence, we have $\Pi^* \models all\_p$. To many this answer is unintuitive, and they point to the fact that adding an unrelated $r(b) \leftarrow$ to Π^* result in the retraction of all_p. I.e., $\Pi^* \cup \{r(b) \leftarrow .\} \not\models all\_p$. Although we discussed this aspect in Section 3.6 and also briefly in Section 6.6.2 we consider a different angle here. Unlike in Section 3.6 where we presented sufficiency conditions that guarantee language independence and tolerance, in this section we propose alternative semantics and alternative ways to characterize entailment.

There are four main proposals to handle this problem, referred to in the literature as the 'Universal Query Problem'.

(i) One proposal, advocated in [Ros92], is to add to every program a fact $q(f(c))$, where q, f and c do not occur in the original program. The basic idea here is to introduce an infinite number of terms to the Herbrand Universe. Following this approach, we have that $\Pi^* \cup \{q(f(c)) \leftarrow .\} \not\models all\_p$.

(ii) The second proposal advocated in [Kun89] is to have a Universal language with an infinite number of terms in the Herbrand Universe in which all programs are expressed. In this case, $\Pi^* \not\models all\_p$, but $\Pi^* \models \neg all\_p$.

(iii) The third proposal advocated in [Kun87, Prz89b] is to consider arbitrary models instead of Herbrand models. In this case, $\Pi^* \not\models all\_p$, and $\Pi^* \not\models \neg all\_p$.

(iv) The fourth proposal by Gelfond articulated in a joint paper in [BG94], is to neither blindly assume a closed domain (as done by using Herbrand models), nor blindly assume an infinite domain (as done by arbitrarily enlarging the Herbrand Universe), but have a characterization with open domain – as also done in (iii), and in addition have a way to selectively specify if we want closed domain. With respect to this proposal Π^*'s answer to all_p will be 'unknown' and the reasoning is that we do not know if a is the only object or not. Both assuming it to be the only object and answering 'yes' and assuming the presence of infinite other objects and answering 'no' amounts to preferring one assumption over another in terms of whether the domain is closed or infinite. We now present this characterization.

9.5.1 Parameterized answer sets and $\models_{open}$

Let Π be an AnsProlog$^\neg$ program over the language $\mathcal{L}_0$. To give the semantics of Π, we will first expand the alphabet of $\mathcal{L}_0$ by an infinite sequence of new constants $c_1, \ldots, c_k, \ldots$ We will call these new constants *generic*. The resulting language will be denoted by $\mathcal{L}_\infty$. By $\mathcal{L}_k$ we will denote the expansion of $\mathcal{L}_0$ by constants $c_1, \ldots, c_k$. Π_k, where $0 \leq k \leq \infty$, will stand for the set of all ground instances of Π in the language $\mathcal{L}_k$. The entailment relation with respect to the language $\mathcal{L}_k$ will be denoted by $\models_k$.

Definition 108 Let Π be an AnsProlog or an AnsProlog$^\neg$ program. By *k-answer set* of Π we will mean a pair $< k, B >$, where B is an answer set of Π in the language $\mathcal{L}_k$. For a query q, we say $\Pi \models_{open} q$, if q is true in all consistent k-answer sets of Π, for all k. □

We will refer to the collection of all consistent k-answer sets as *parameterized* answer sets.

Example 155 Consider a language $\mathcal{L}_0$ over the alphabet $\{a\}$ and an AnsProlog program Π^* consisting of the rules

$p(a) \leftarrow$.
$not\_all\_p \leftarrow$ **not** $p(X)$.
$all\_p \leftarrow$ **not** not_all_p.

The following are parameterized answer sets of Π:
$\{< 0, \{p(a), all\_p\} >\}$,$\{< 1, \{p(a), not\_all\_p\} >\}$, $\{< 2, \{p(a), not\_all\_p\} >\}$, ...

Thus all_p is *true* in the first answer set as the only constant in the language $\mathcal{L}_0$ is a while it is not *true* in all other answer sets as the the corresponding languages contain constants other than a. Hence, as intended, Π's answer to the query all_p is *unknown*. I.e., $\Pi \not\models_{open} all\_p$, and $\Pi \not\models_{open} \neg all\_p$. □

9.5.2 Applications of $\models_{open}$

We first show that the $\models_{open}$ characterization allows the explicit specification – when desired — of domain-closure assumption.

Let Π be an arbitrary AnsProlog$^\neg$ program in a language $\mathcal{L}_0$. We expand $\mathcal{L}_0$ by the unary predicate symbol h which stands for *named elements of the domain*. The following rules can be viewed as the definition of h:

H_1. $h(t) \leftarrow$ (for every ground term t from $\mathcal{L}_0$)

H_2. $\neg h(X) \leftarrow$ **not** $h(X)$

The domain-closure assumption is then expressed by the rule:

DCA. **if** $\neg h(X)$

The following example illustrates the role of DCA.

Example 156 Let Π be an AnsProlog$^\neg$ program consisting of H_1, H_2 and the following rules:

$p(a) \leftarrow$.
$q(a) \leftarrow$ **not** $p(X)$.

$\neg p(X) \leftarrow$ **not** $p(X)$.
$\neg q(X) \leftarrow$ **not** $q(X)$

The k-answer sets of Π is

$\{< 0, \{h(a), p(a), \neg q(a)\} >\}$, if $k = 0$, and

$\{< k, \{h(a), \neg h(c_1) \ldots \neg h(c_k), p(a), q(a), \neg p(c_1), \neg q(c_1) \ldots \neg p(c_k), \neg q(c_k), \} >\}$, if $k > 0$,

and therefore, Π's answer to the query $q(a)$ with respect to $\models_{open}$ is *unknown*. I.e., $\Pi \not\models_{open} q(a)$, and $\Pi \not\models_{open} \neg q(a)$. The answer changes if Π is expanded by the domain closure assumption (DCA). The resulting program, Π_C, has the unique answer set $\{< 0, \{h(a), p(a), \neg q(a)\} >\}$ and therefore, Π_C's answer to $q(a)$ with respect to $\models_{open}$ is *no*, exactly the answer produced by the program $\{p(a) \leftarrow .; q(a) \leftarrow$ **not** $p(X)\}$ with respect to $\models$, which has the domain closure assumption built-in. □

Exercise 27 Show that for any AnsProlog$^\neg$ program Π, its answer to any query with respect to $\models$ is same as $\Pi \cup \{H_1, H_2, DCA\}$'s answer with respect to $\models_{open}$. □

Now we will briefly discuss an example that shows the use of domain assumptions and of the concept of named objects.

Example 157 Consider a departmental database containing the list of courses which will be offered by a department next year, and the list of professors who will be working for the department at that time. Let us assume that the database knows the names of all the courses which may be taught by the department but, since the hiring process is not yet over, it does not know the names of all of the professors. This information can be expressed as follows:

$course(a) \leftarrow.$
$course(b) \leftarrow.$

$prof(m) \leftarrow.$
$prof(n) \leftarrow$

$\neg\ course(X) \leftarrow \neg h(X)$

The k-answer set of this program is

$< k, \{course(a), course(b), \neg\ course(c_1) \ldots \neg course(c_k), prof(m), prof(n),$
$h(a), h(b), h(m), h(n), \neg h(c_1) \ldots \neg h(c_k)\} >$

and therefore, the above program answers *no* to the query

$\exists X\ (course(X) \wedge \neg h(X))$

and *unknown* to the query

$\exists X\ (prof(X) \wedge \neg h(X)).$

with respect to $\models_{open}$. Notice that in this example, it is essential to allow for the possibility of unknown objects.

Let us now expand the informal specification of our database by the closed world assumptions for predicates *course* and *prof*. The closed world assumption for *course* says that there are no other courses except those mentioned in the database and can be formalized by the standard rule

$\neg course(X) \leftarrow \mathbf{not}\ \ course(X).$

Using this assumption, we will be able to prove that a and b are the only courses taught in our department. In the case of predicate *prof*, however, this (informal) assumption is too strong – there may, after all, be some unknown professor not mentioned in the list. However, we want to be able to allow our database to conclude that *no one known to the database is a professor unless so stated*. For that we need a weaker form of the closed world assumption, which will not be applicable to generic elements. This can easily be accomplished by the following rule:

$\neg prof(X) \leftarrow h(X), \mathbf{not}\ \ prof(X).$

The k-answer set of the resulting program Π looks as follows:

$< k, \{c(a), c(b), \neg c(m), \neg c(n), \neg c(c_1) \ldots \neg c(c_k), p(m), p(n), \neg p(a), \neg p(b),$
$h(a), h(b), h(m), h(n), \neg h(c_1) \ldots \neg h(c_k)\} >,$

where c stands for *course* and p stands for *prof*. This allows us to conclude, say, that a is not a professor without concluding that there are no professors except m and n. □

9.6 Well-founded semantics of programs with AnsProlog syntax

In Section 1.3.6 we presented a definition of the well-founded semantics and Chapter 7 we used the computation of the well-founded semantics as a first step in computing the answer sets. Since the well-founded semantics, which we treat in this book as an approximate semantics, is widely preferred over the answer set characterization in circles – such as databases – where efficiency is a bigger concern than expressiveness, in this section we explore it in more detail. In particular we present several different characterization of the well-founded semantics of programs with AnsProlog syntax and in the subsequent section we discuss how this characterization is extended to programs with AnsProlog$^{\neg}$ syntax.

9.6.1 Original characterization using unfounded sets

The initial characterization of the well-founded semantics was done using a notion of *unfounded sets*. In this characterization a partial interpretation I is viewed as a pair $\langle T, F\rangle$, where $T \cap F = \emptyset$, and $T, F \subseteq HB$.

Definition 109 (Unfounded sets and greatest unfounded sets) Let Π be an AnsProlog program and I be a partial interpretation. We say $A \subseteq HB_\Pi$ is an unfounded set of Π with respect to I if each atom atom $p \in A$ satisfy the following condition: For each rule in $ground(\Pi)$ whose head is p, at least one of the following holds.

1. Some positive subgoal q or negative subgoal **not** q of the body is inconsistent with I.
2. Some positive subgoal of the body occurs in A.

The greatest unfounded set with respect to I is the union of all sets that are unfounded with respect to I. □

Definition 110 Let Π be an AnsProlog program and I be a partial interpretation.

$T_\Pi(I) = \{p :$ there is a rule r in Π with p in the head such that the body of r evaluates to true with respect to I $\}$.

$F_\Pi(I)$ is the greatest unfounded set with respect to I. □

Definition 111 For all countable ordinals α we define I_α as follows:

- $I_0 = \emptyset$.
- If α is a successor ordinal $k+1$ then $I_{k+1} = \langle T_\Pi(I_k), F_\Pi(I_k)\rangle$.
- If α is a limit ordinal then $I_\alpha = \bigcup_{\beta<\alpha} I_\beta$. □

Lemma 9.6.1 I_α is a monotonic sequence of partial interpretations. □

The above sequence reaches a limit $I^* = \langle T^*, F^*\rangle$ at some countable (possibly beyond the first limit ordinal ω) ordinal. The *well-founded* semantics of an AnsProlog program is defined as this limit I^*.

9.6.2 A slightly different iterated fixpoint characterization

We now present a slightly different fixpoint characterization where the notion of greatest unfounded set used in $F_\Pi(I)$ and $T_\Pi(I)$ from Definition 110 is replaced by a a fixpoint computation.

Definition 112 Let I be a partial interpretation, Π be an AnsProlog program and T' and F' be sets of ground atoms.

$\mathcal{T}_I(T') = \{p :$ p is not true in I and there is a rule r in Π with p in the head such that each literal in the body of r evaluates to true with respect to I or is in T'. $\}$

$\mathcal{F}_I(F') = \{p :$ p is not false in I and for every rule r in Π with p in the head there is at least one literal in the body of r that evaluates to false with respect to I or is in F'. $\}$ □

Lemma 9.6.2 The operators $\mathcal{T}_I$ and $\mathcal{F}_I$ are monotonic, i.e., $T' \subseteq T'' \Rightarrow \mathcal{T}_I(T') \subseteq \mathcal{T}_I(T'')$ and $F' \subseteq F'' \Rightarrow \mathcal{F}_I(F') \subseteq \mathcal{F}_I(F'')$ □

Definition 113 For a program Π and partial interpretation I,

$T_I^{\uparrow 0} = \emptyset$; $T_I^{\uparrow n+1} = \mathcal{T}_I(T_I^{\uparrow n})$; $T_I = \bigcup_{n<\omega} T_I^{\uparrow n}$

$F_I^{\downarrow 0} = HB_\Pi$; $F_I^{\downarrow n+1} = \mathcal{F}_I(F_I^{\downarrow n})$; $F_I = \bigcap_{n<\omega} F_I^{\downarrow n}$ □

Lemma 9.6.3 The transfinite sequence $\{T_I^{\uparrow n}\}$ is monotonically increasing and the transfinite sequence $\{F_I^{\uparrow n}\}$ is monotonically decreasing.

T_I is the least fixpoint of the operator $\mathcal{T}_I$ and F_I is the greatest fixpoint of the operator $\mathcal{F}_I$. □

Definition 114 Let $\mathcal{I}$ be the operator assigning to every partial interpretation I of P a new interpretation $\mathcal{I}$(I) defined by :

$\mathcal{I}(\mathrm{I}) = \mathrm{I} \cup < T_I; F_I >$. □

Lemma 9.6.4 The operator $\mathcal{A}$ is monotonic with respect to the ordering $\preceq$ defined as $\langle T, F\rangle \preceq \langle T', F'\rangle$ iff $T \subseteq T'$ and $F' \subseteq F$. □

Definition 115 *[Prz89a]* Let $M_0 = < \emptyset, \emptyset >$;

$M_{\alpha+1} = \mathcal{I}(M_\alpha) = M_\alpha \cup < T_{M_\alpha}; F_{M_\alpha} >$;

$M_\alpha = \bigcup_{\beta<\alpha} M_\beta$, *for limit ordinal* α. □

The sequence $\{M_\alpha\}$ of interpretations is monotonically increasing. Therefore there is a smallest ordinal δ such that M_δ is a fixpoint of the operator $\mathcal{I}$. Let is refer to this fixpoint as M_Π.

Theorem 9.6.5 Given an AnsProlog program Π, M_Π is its well-founded semantics. □

9.6.3 Alternating fixpoint characterization

We now give yet another characterization of the well-founded semantics.

Definition 116 Suppose I is an Herbrand interpretation and Π be an AnsProlog program. Construct a program Π' as follows: If

$$A \leftarrow B_1 \,\&\, \ldots \,\&\, B_n \,\&\, \textbf{not}\ D_1 \,\&\, \ldots \,\&\, \textbf{not}\ D_m$$

is in Π, then

$$A \leftarrow B_1 \,\&\, \ldots \,\&\, B_n \,\&\, \tilde{D_1} \,\&\, \ldots \,\&\, \tilde{D_m}$$

is in Π'. Here $\tilde{D_i} = \tilde{p}(\vec{t})$ iff $D_i = p(\vec{t})$. In addition, if $A \notin I$, then the unit clause $\tilde{A} \leftarrow$ is added to Π'.
Now define $S_\Pi(\overline{I}) = HB_\Pi \cap T_{\Pi'} \uparrow \omega$. □

Note that $T_{\Pi'} \uparrow \omega$ contains new atoms of the form $\tilde{p}(\vec{t})$ and that these atoms may be used in deriving atoms in $S_\Pi(\overline{I})$, but atoms of the form $\tilde{p}(\vec{t})$ are themselves not present in $S_\Pi(\overline{I})$.

Definition 117 Given an AnsProlog program Π, we associate an operator A_Π with Π as follows:
$A_\Pi(I) = \overline{S_\Pi(\overline{S_\Pi(I)})}$.
$A^* = lfp(A_\Pi)$.
$A^+ = S_\Pi(A^*)$. □

Theorem 9.6.6 [Gel89] A^+ is the set of atoms true in the well-founded model of Π. Similarly A^* is the set of atoms false in the well-founded model of Π. □

We now use Van Gelder's alternating fixpoint characterization to show that well-founded semantics is equivalent to a particular stable class as defined in the next subsection.

9.6.4 Stable Classes and duality results

Suppose, given a program Π, we associate an operator F_Π that maps Herbrand interpretations to Herbrand interpretations such that $F_\Pi(I) = M(\Pi^I)$. Answer sets are defined in terms of the fixed point of this operator, i.e. I is an answer set of P iff $F_\Pi(I) = M(\Pi^I) = I$. However, this operator may not always have fixed points.

Definition 118 Let $\mathcal{A}$ be a set of indices. Let $S = \{I_i \mid i \in \mathcal{A}\}$ be a finite set of interpretations. S is said to be a *stable class* of program Π iff S $= \{F_\Pi(I_i) \mid i \in \mathcal{A}\}$. □

Example 158 Consider the following AnsProlog program Π:

$a \leftarrow \textbf{not}\ \ a.$
$p \leftarrow.$

This program does not have any answer sets. But it has two stable classes: S_0 which is the empty collection of interpretations and $S_1 = \{I_1, I_2\}$ where:
$I_1 = \{p\}$
$I_2 = \{a, p\}$

Thus, Π has a unique non-empty stable class, viz. S_1, and p is *true* in all interpretations contained in S_1. □

Lemma 9.6.7 $S_\Pi(\overline{I}) = F_\Pi(I)$ □

Lemma 9.6.8 Let Π be an AnsProlog program. Then $F_\Pi(lfp(F_\Pi^2)) = gfp(F_\Pi^2)$ and $F_\Pi(gfp(F_\Pi^2)) = lfp(F_\Pi^2)$. i.e. $\{lfp(F_\Pi^2), gfp(F_\Pi^2)\}$ form a stable class of Π. □

Lemma 9.6.9 $lfp(F_\Pi^2) = A^+$
$gfp(F_\Pi^2) = \overline{A^*}$ □

Lemmas 9.6.7, 9.6.8, 9.6.9 and Theorem 9.6.6 are required to establish the following theorem.

Theorem 9.6.10 (Well-Founded Semantics is Captured by a Stable Class) Let Π be an AnsProlog program. The well-Founded semantics of Π is characterized by a particular stable class C of Π, i.e. a ground atom is true in the well-founded semantics of Π iff A is true in all interpretations in C, and A is false according to the well founded semantics of Π iff A is false in all interpretations in C. Moreover, $C = \{lfp(F_\Pi^2)\,,\ gfp(F_\Pi^2)\}$. □

Example 159 Consider the following program Π_4 :

$$\left.\begin{array}{l} p \leftarrow \textbf{not } a \\ p \leftarrow \textbf{not } b \\ a \leftarrow \textbf{not } b \\ b \leftarrow \textbf{not } a \end{array}\right\} \Pi_4$$

The above program has two answer sets $\{p, a\}$ and $\{p, b\}$. It has three stable classes, $\{\{p, a\}\}$, $\{\{p, b\}\}$, and $\{\{\}, \{p, a, b\}\}$. The stable class $\{\{\}, \{p, a, b\}\}$ corresponds to the well-founded semantics of the above program. Therefore, p is a consequence of Π_4 with respect to the answer set semantics, while the answer to p in the well-founded semantics is *undefined*. □

Example 160 Consider the following program Π_5 [VG88]:

$$\left.\begin{array}{l} q \leftarrow \textbf{not } r \\ r \leftarrow \textbf{not } q \\ p \leftarrow \textbf{not } p \\ p \leftarrow \textbf{not } r \end{array}\right\} \Pi_5$$

Π_5 has a unique answer set, viz. $\{p, q\}$. Π_5 has three strict stable classes (stable classes which have no proper subset which is also a stable class), namely, C_1, C_2 *and* C_3, where $C_1 = \{\{q, p\}\}$, $C_2 = \{\emptyset, \{p, q, r\}\}$ and $C_3 = \{\{r\}, \{r, p\}\}$. Of these, the class C_2 corresponds to the well-founded semantics which says that p, q, r are all *undefined*. Notice that even though p is the consequence of Π_5 in the answer set semantics, its addition to Π_5 alters the set of consequences of Π_5. In particular, we will no longer be able to conclude q. □

9.7 Well-founded semantics of programs with AnsProlog$^\neg$ syntax

The formulation of well-founded semantics of AnsProlog programs in [BS91] can be extended to define the well-founded semantics [Prz90a] of AnsProlog$^\neg$ programs. More precisely, let us consider $G_\Pi(S) = \mathcal{M}^{\neg,\perp}(\Pi^S)$. Then for any AnsProlog$^\neg$ program Π, the fixpoints of G_Π defines the answer-set semantics, and $\{lfp(G_\Pi^2), gfp(G_\Pi^2)\}$ defines the well-founded semantics. A literal l is *true* (resp. $false$) w.r.t. the well-founded semantics of an AnsProlog$^\neg$ program Π if $l \in lfp(G_\Pi^2)$ (resp. $l \notin gfp(G_\Pi^2)$). Otherwise l is said to be *undefined*.

Pereira et al. [PAA92a] show that this definition gives unintuitive characterizations for several programs.

Example 161 Consider the program Π_0
$a \leftarrow$ **not** b
$b \leftarrow$ **not** a
$\neg a \leftarrow$
The well-founded semantics infers $\neg a$ to be $true$ and a and b to be $unknown$ with respect to the above program. Intuitively, b should be inferred $true$ and a should be inferred $false$. □

Example 162 Consider the program Π_1
$b \leftarrow$ **not** $\neg b$
and the program Π_2
$a \leftarrow$ **not** $\neg a$
$\neg a \leftarrow$ **not** a
The well-founded semantics infers b to be $true$ with respect to Π_1 and infers b to be $undefined$ with respect to $\Pi_1 \cup \Pi_2$ even though Π_2 does not have b in its language. □

To overcome the unintuitiveness of the well-founded semantics Pereira et al. [PAA92a] propose an alternative semantics of AnsProlog$^\neg$ programs which we refer to as the Ω-well-founded semantics. We now define the Ω-well-founded semantics.

Definition 119 *[PAA92a]* Let Π be an AnsProlog$^\neg$ program. $S(\Pi)$ the semi-normal version of Π is obtained by replacing each rule of the form $L_0 \leftarrow L_1, \ldots, L_m,$ **not** $L_{m+1}, \ldots,$ **not** L_n by the rule:

$$L_0 \leftarrow L_1, \ldots, L_m, \textbf{not } L_{m+1}, \ldots, \textbf{not } L_n, \textbf{not } \neg L_0. \tag{9.7.4}$$

□

Definition 120 *[PAA92a]* For any AnsProlog$^\neg$ program Π, the function Ω_Π is defined as $\Omega_\Pi(X) = G_\Pi(G_{S(\Pi)}(X))$ □

Definition 121 *[PAA92a]* A set of literals E is said to be an Ω-extension of an AnsProlog$^\neg$ program Π iff
1. E is a fixpoint of Ω_Π.
3. E is a subset of $(G_{S(\Pi)}(E))$ □

Pereira et al. [PAA92a] show that if an AnsProlog$^\neg$ program has an Ω-extension then Ω_Π is a monotonic function and hence has a least fixpoint. The Ω-well-founded semantics is defined as $\{lfp(\Omega_\Pi),\ G_{S(\Pi)}(lfp(\Omega_\Pi))\}$. Entailment w.r.t. the Ω-well-founded semantics is defined as follows: A literal l is $true$ (resp. $false$) w.r.t. the Ω-well-founded semantics of an AnsProlog$^\neg$ program Π if $l \in lfp(\Omega_\Pi)$ (resp. $l \notin G_{S(\Pi)}(lfp(\Omega_\Pi))$). Otherwise l is $undefined$.

Example 163 *[PAA92a]* Consider the following program Π_3
$c \leftarrow$ **not** b
$b \leftarrow$ **not** a
$a \leftarrow$ **not** a
$\neg b$
The above program has $\{c, \neg b\}$ as the only Ω-extension. The Ω-well-founded semantics is given by $\{\{c, \neg b\}, \{c, a, \neg b\}\}$. □

Before we end this section we would like to briefly mention another class of semantics of AnsProlog$^{\neg}$ programs based on contradiction removal [Dun91, Wag93, PAA91a, GM90].

To illustrate the problem let us consider the program Π_4:
1. $p \leftarrow$ **not** q
2. $\neg p \leftarrow$
3. $s \leftarrow$

Obviously, under the answer set semantics this program is inconsistent. It is possible to argue however that inconsistency of Π_4 can be localized to the rules (1.) and (2.) and should not influence the behavior of the rest of the program, i.e. Π_4's answer to query s should be *yes* and the rules causing inconsistency should be neutralized. There are several approaches to doing that. One, suggested in [KS90], modifies the answer set semantics to give preference to rules with negative conclusions (viewed as exceptions to general rules). Under the corresponding entailment relation Π_4 concludes s and $\neg p$. Another possibility is to first identify literals responsible for contradiction, in our case q. After that q can be viewed as abducible[1] and hence Π_4 will entail s, $\neg p$ and q. Another possibility arises when Ω-well-founded semantics is used as the underlying semantics of Π_4. In this case we may want to have both q and $\neg q$ undefined. This can be achieved by expanding Π_4 by new statements $q \leftarrow$ **not** q and $\neg q \leftarrow$ **not** $\neg q$. The resulting program Π_5 entails (w.r.t. the Ω-well-founded semantics) $\neg p$ and s and infers p to be $false$. The last idea is developed to a considerable length in [PAA91a, PA93].

9.8 Notes and references

In this chapter we have presented only a small and incomplete set of extensions and alternatives to AnsProlog* based on our perception of closeness to the rest of the content of this book.

The material on the extension of AnsProlog* that allows **not** in the head is from [IS98]. The extension that allos nested expressions is from [LTT99, FL01]. The extension of AnsProlog* to allow knowledge and belief operators was proposed in [Gel91b] and further developed in [GP91]. There has not been much work on this language since then. In particular it remains to be seen if AnsProlog$^{\neg, or, K, M}$ programs with multiple world views will prove to be useful for knowledge representation. Moreover complexity and expressibility analysis of AnsProlog$^{\neg, or, K, M}$ programs also remain to be done.

The discussion regarding open domain and the universal query problem is based on the papers [Ros89a, Ros92, Kun89, Kun87, Prz89b, BG94, AB94]. Complexity of reasoning with open domains is explored in [Sch93]. Application of open domains is discussed in [GT93, GP93]. The paper [GT93] shows the usefulness of open domain semantics in representing certain types of null values in databases., while [GP93] discusses an application to formalization of anonymous exceptions to defaults.

Well-founded semantics of AnsProlog programs was initially defined by Van Gelder, Ross, and Schlifp in [vGRS88]. Later Van Gelder [Gel89] gave a different characterization of the well-founded semantics based on an alternating fixpoint approach. Some of the alternative characterization of the well-founded semantics that we presented in this paper are based on [Prz89a, Prz89d, BS92, BS91, Fit91]. Some of the additional characterization of the well-founded semantics that we did not discuss in this chapter are given in [Dun93] and [Prz90c]; the first one uses argumentation while the second

[1] Abducible literals are literals that can be assumed *true* if necessary. For more details see Section 3.9.

one characterizes well-founded semantic using 3-valued stable models. The well-founded characterization of AnsProlog$^{\neg}$ programs are discussed in [AP92, PAA91b, PAA92a, PAA92b, Prz90a, KS90, Dun91, Wag93, PAA91a, GM90, PA93]. We presented some of those results.

In the late eighties and early ninties several other semantics for programs with AnsProlog syntax were proposed. Some of these ones are [FBJ88, Fit85, Fit86, Kun87, Kun89, LM85, Myc83]. Similarly several alternative characterization of AnsProlog^{or} programs were also developed. Some of those are [Bar92, LMR92, RM90, BLM92, BLM91, Ros89b, Prz90b, Sak89, Prz91]. Przymusinski considered several extensions of AnsProlog^{or} [Prz95, BDT96, BDNT01] and proposed semantics for these languages which when restricted to AnsProlog syntax coincided with the well-founded semantics. Similar extensions were also studied by Minker and Ruiz in [MR93].

Among the major omissions in this chapter are meta logic programming [Kow79] and constraint logic programming [Mah93a]. Some of the met-logic programming languages and their analysis are given in [Kow90, AR89, Pet92, HL91, CKW93, CL89, MDS92a, MDS92b, Mil86, BM90, GO92, BMPT92, BK82]. A good discussion of either would need their own chapter.

Chapter 10

Appendix A: Ordinals, Lattices and fixpoint theory

10.1 Ordinals

The finite ordinals are the non-negative integers. The first ordinal 0 is defined by the empty set, $\emptyset$. The second ordinal $1 = \{0\} = \{\emptyset\}$. The third ordinal $2 = \{0, 1\} = \{\emptyset, \{\emptyset\}\}$. The fourth ordinal $3 = \{0, 1, 2\} = \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}\}$.

The first infinite ordinal is $\omega = \{0, 1, 2, \ldots\}$, the set of all non-negative integers.

The successor of an ordinal α is $\alpha + 1 = \alpha \cup \{\alpha\}$. Any ordinal which is the successor of another ordinal is referred to as a successor ordinal .

A limit ordinal is an ordinal other than 0 which is not the successor of any other ordinal. The first limit ordinal is ω. The successor ordinal of ω is $\omega + 1 = \omega \cup \{\omega\}$. The second limit ordinal is $\omega 2$, which is the set $\omega \cup \{\omega + n \mid n \in \omega\}$. The successor ordinal of $\omega 2$ is $\omega 2 + 1 = \omega 2 \cup \{\omega 2\}$.

10.2 Fixpoint theory

Let S be a set. A *relation* R on S is a subset $S \times S$. A relation R on S is a *partial order* if R is *reflexive*, *antisymmetric* and *transitive*. A binary relation (not necessarily a partial order) is *well-founded* if there is no infinite decreasing chain $x_0 \geq x_1 \geq \ldots$.

Let S be a set with partial order $\leq$. Then $a \in S$ is an *upper bound* of a subset X of S if $x \leq a$, for all $x \in X$. Similarly, $b \in S$ is a *lower bound* of X if $b \leq x$, for all $x \in X$. $a \in S$ is the *least upper bound (lub)* of a subset X of S if a is an upper bound of X and for all upper bound a' of X we have $a \leq a'$. Similarly, $b \in S$ is the *greatest lower bound (glb)* of a subset X of S if b is a lower bound of X and for all lower bound b' of X we have $b' \leq b$.

A partially ordered set L is a **complete lattice** if $lub(X)$ and $glb(X)$ exist for every subset X of L. The lub(L) is called the top element ($\top$) and the glb(L) is called the bottom element ($\bot$).

Example 164 For any set S, its power set 2^S under $\subseteq$ is a complete lattice with $lub(X)$ being the union of all elements of X and $glb(X)$ being the intersection of all element of L. The top element is S and the bottom element is $\emptyset$. □

Monotonicity : Let L be a complete lattice and $T : L \to L$ be a mapping. We say T is monotonic if $T(x) \leq T(y)$, whenever $x \leq y$.

Let L be a complete lattice and $X \subseteq L$. We say X is **directed** if every finite subset of X has an upper bound in X.

Continuity : Let L be a complete lattice and $T : L \to L$ be a mapping. We say T is continuous if $T(lub(X)) = lub(T(X))$, for every directed subset X of L.

fixpoint, least fixpoint : Let L be a complete lattice and $T : L \to L$ be a mapping. We say $a \in L$ is the least fixpoint (lfp) if a is a fixpoint of T (i.e. T(a) = a) and for all fixpoints b of T, we have $a \leq b$.

Theorem 10.2.1 *[Tar55]* Let L be a complete lattice and $T : L \to L$ be a monotonic mapping. Then T has a least fixpoint, $lfp(T)$, and furthermore $lfp(T) = glb\{x : T(x) = x\} = glb\{x : T(x) \leq x\}$. □

Definition 122 Let L be a complete lattice and $T : L \to L$ be a monotonic mapping. Then,

$T \uparrow o = \bot$
$T \uparrow \alpha = T(T \uparrow (\alpha - 1)$, if α is a successor ordinal
$T \uparrow \alpha = lub\{T \uparrow \beta : \beta < \alpha\}$ if α is a limit ordinal. □

Theorem 10.2.2 Let L be a complete lattice and $T : L \to L$ be monotonic. Then $lfp(T) = T \uparrow \alpha$, where α is a limit ordinal. □

Theorem 10.2.3 *[Tar55]* Let L be a complete lattice and $T : L \to L$ be continuous. Then $lfp(T) = T \uparrow \omega$, where ω is the first limit ordinal. □

10.3 Transfinite Sequences

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

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

Chapter 11

Appendix B: Decidability and Complexity

11.1 Turing Machines

Intuitively, a **deterministic Turing machine** (DTM) is an automata bundled with a semi-infinite tape with a cursor to read from and write to. So like an automata, there is a state and state transitions are based on the current state and what the cursor points to on the tape. But in addition to the state transition, there is an accompanying transition that dictates if the symbol in the tape location pointed to by the cursor should be overwritten and if the cursor should move to the left or right – by one cell – of its current position. Special symbols mark the beginning of the tape ($>$), the end of the input on the tape ($\sqcup$), and the output of the computation ('*halt*', '*yes*', '*no*'). We now give a formal definition of DTMs.

Definition 123 A DTM is a quadruple $M = (S, \Sigma, \delta, s_0)$ where S is a finite set of non-final states that includes s_0 the initial state, Σ is a finite alphabet of symbols including the special symbols, $>$, and $\sqcup$ and δ is a transition function that maps $S \times \Sigma$ to $S \cup \{halt, yes, no\} \times \Sigma \times \{\leftarrow, \rightarrow, -\}$. $\square$

Intuitively, δ is control or program of the machine that dictates how the machine behaves. If $\delta(s, \alpha) = (s', \beta, \leftarrow)$ then it means that if the current state is s and the cursor is pointed at the symbol α then the state should change to s', α should be overwritten by β and the cursor should move to the left of its current position. If instead of $\leftarrow$, we had $\rightarrow$ then that would dictate the cursor to move right, and if we had '$-$' instead, then that would dictate the cursor to remain where it is. The special symbol $>$ is used as the left marker of the tape. Hence, for any state s, we require that $\delta(s, >) = (s', >, \rightarrow)$, for some s'. This forces that whenever the cursor is at the left end of the tape, it must move right without overwriting the $>$ symbol at the left end.

Initially, the state of the machine is required to be s_0, and the cursor is required to be at the left end of the tape pointing to $>$. Moreover, the string starting after the left marker $>$, until the first $\sqcup$ is considered as the *input* I. From this initial configuration the machine makes the transition dictated by its δ, until it reaches one of the final states $\{halt, yes, no\}$. If the final state is *yes*, that means the machine accepted the input (i.e., M(I) = 'yes'), if it is *no*, that means the machine rejected the input (i.e., M(I) = 'no') , and if it is *halt*, that means the machine computed an output $M(I)$, which is defined as the string starting after the left end marker $>$ until the first $\sqcup$.

Note that it is possible that the machine never reaches a final state and keeps on computing for ever.

A set S is said to be recursive if there is a DTM M such that given an input x, $M(x) = `yes'$ iff $x \in S$, and $M(x) = `no'$ iff $x \notin S$. S is said to be r.e. (or recursively enumerable) if there is a Turing machine M such that given an input x, $M(x) = `yes'$ iff $x \in S$. Note that in this case if $x \notin S$ then either $M(x) = `no'$ or the Turing machine never reaches the 'halt' state.

A **non-deterministic Turing machine** (NDTM) is a quadruple (S, Σ, δ, s_0) like a DTM, except that δ is no longer a function; it is a relation given as a subset of $(S \times \Sigma) \times (S \cup \{halt, yes, no\}) \times \Sigma \times \{\leftarrow, \rightarrow, -\}$. As in a DTM, the role of δ is to be the control of the machine. When the machine is in state s and the cursor is pointed at the symbol α then the control considers all quadruples from δ whose first element is (s, α), and nondeterministically chooses one quadruple $((s, \alpha), s', \beta, dir)$ from it, and changes the state to s', replaces α by β and moves the cursor according to dir.

The time taken by a DTM M on an input I is defined as the number of transitions taken by M on I from the start till it stops. If it does not stop then the time is considered to be infinite. For a function f from positive integers to itself, we say that a DTM M takes $O(f(n))$ time, if there exists positive integers c and n_0 such that the time taken by M on any input of length n is not greater than $c \times f(n)$ for all $n \geq n_0$.

11.1.1 Oracle Turing Machines

An Oracle DTM M^A, also referred to as a DTM M with an oracle A can be thought of as a DTM with an additional write-only tape referred to as the query tape, and three special states $\{q, q_y, q_n\}$. When the state of M^A is different from $\{q, q_y, q_n\}$ the computation of the oracle DTM M^A is same except that M^A can write on the query tape. When the state is q, M^A moves to the state q_y or q_n depending on whether the current query string in the query tape is in A or not, while instantly erasing the query tape.

11.2 Computational Complexity

Definition 124 An existential second-order formula over the vocabulary σ is an expression of the form $\exists S_1, \ldots, S_m \phi(S_1, \ldots, S_m)$, where S_i's are relational predicate symbols different from those in σ and $\phi(S_1, \ldots, S_m)$ is an arbitrary first order formula with relational predicate symbols among those in σ and $\{S_1, \ldots, S_m\}$.

Chapter 12

Appendix C: Pointers to resources

- DBLP

  http://www.informatik.uni-trier.de/~ley/db/index.html

  http://www.acm.org/sigmod/dblp/db/index.html
- Michael Gelfond.

  http://earth.cs.ttu.edu/~mgelfond/
- Vladimir Lifschitz.

  http://www.cs.utexas.edu/users/vl/
- H. Turner

  http://www.d.umn.edu/~hudson/
- C. Baral.

  http://www.public.asu.edu/~cbaral/
- TAG

  http://www.cs.utexas.edu/users/vl/tag/
- ccalc

  http://www.cs.utexas.edu/users/tag/cc/
- I. Niemela

  http://saturn.hut.fi/~ini/
- Smodels

  http://www.tcs.hut.fi/Software/smodels/
- N. Leone

  http://www.dbai.tuwien.ac.at/staff/leone/

- G. Gottlob

  http://www.dbai.tuwien.ac.at/staff/gottlob/

- DBAI at Viena

  http://www.dbai.tuwien.ac.at/

- DLV

  http://www.dbai.tuwien.ac.at/proj/dlv/

- M. Truszczynski

  http://www.cs.engr.uky.edu/~mirek

- W. Marek

  http://www.cs.engr.uky.edu/~marek/

- DeRES

  http://www.cs.engr.uky.edu/ai/deres.html

- D. S. Warren

  http://www.cs.sunysb.edu:80/~warren/

- XSB

  http://xsb.sourceforge.net/

- C. Zaniolo

  http://www.cs.ucla.edu/~zaniolo/

  http://www.cs.ucla.edu/~zaniolo/cz/personal.html

- LDL^{++}

  http://www.cs.ucla.edu/ldl/

- J. Dix

  http://www.uni-koblenz.de/~dix/

- T. Przymusinski.

  http://www.cs.ucr.edu/~teodor/

- H. Przymusinska.

  http://www.informatik.uni-trier.de/~ley/db/indices/a-tree/p/Przymusinska:Halina.html

- V. S. Subrahmanian

  http://www.cs.umd.edu/users/vs/

- L. Pereira

  http://centria.di.fct.unl.pt/~lmp/

- J. Alferes

  http://www.dmat.uevora.pt/~jja/

- M. Cadoli

  http://www.dis.uniroma1.it/~cadoli/

- J. You

  http://www.cs.ualberta.ca/~you/

- Li-Yan Yuan

  http://web.cs.ualberta.ca:80/~yuan/

- C. Sakama

  http://www.wakayama-u.ac.jp/~sakama/

- K. Inoue

  http://www.informatik.uni-trier.de/~ley/db/indices/a-tree/i/Inoue:Katsumi.html

- Ken Satoh

  http://mhjcc3-ei.eng.hokudai.ac.jp/lab/ksatoh.html

- A. Kakas

  http://www.cs.ucy.ac.cy/kakas.html

- P. Dung

  http://www.cs.ait.ac.th/~dung/

- D. Pearce

  http://www.compulog.org/staff/DavidPearce.html

- G. Wagner

  http://www.inf.fu-berlin.de/~wagnerg/

Bibliography

[AB90] K. Apt and M. Bezem. Acyclic programs. In D. Warren and Peter Szeredi, editors, *Logic Programming: Proc. of the Seventh Int'l Conf.*, pages 617–633, 1990.

[AB91] K. Apt and M. Bezem. Acyclic programs. *New Generation Computing*, 9(3,4):335–365, 1991.

[AB94] K. Apt and R. Bol. Logic programming and negation: a survey. *Journal of Logic Programming*, 19,20:9–71, 1994.

[ABW88] K. Apt, H. Blair, and A. Walker. Towards a theory of declarative knowledge. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 89–148. Morgan Kaufmann, San Mateo, CA., 1988.

[AD94] K. Apt and K. Doets. A new definition of SLDNF resolution. *Journal of Logic Programming*, 18:177–190, 1994.

[AD95] C. Aravindan and P. Dung. On the correctness of unfold/fold transformation of normal and extended logic programs. *Journal of Logic Programming*, pages 201–217, 1995.

[AHV95] S. Abiteboul, R. Hall, and V. Vianu. *Foundations of Databases*. Addison Wesley, 1995.

[AN78] H. Andreka and I. Nemeti. The generalized completeness of Horn predicate logic as a programming language. *Acta Cybernetica*, 4:3–10, 1978.

[AP91] Krzysztof Apt and Dino Pedreschi. Proving termination in general prolog programs. In *Proc. of the Int'l Conf. on Theoretical Aspects of Computer Software (LNCS 526)*, pages 265–289. Springer Verlag, 1991.

[AP92] J. Alferes and L. Pereira. On logic program semantics with two kinds of negation. In K. Apt, editor, *Proc. of the Joint International Conference and Symposium on Logic Programming, Wash DC*, pages 574–588. MIT Press, Nov 1992.

[AP93] K. Apt and D. Pedreschi. Reasoning about termination of pure prolog programs. *Information and Computation*, 106(1):109–157, 1993.

[AP94] K. Apt and A. Pellegrini. On the occur-check free logic programs. *ACM Transaction on Programming Languages and Systems*, 16(3):687–726, 1994.

[Apt89] K.R. Apt. Introduction to Logic Programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*. North Holland, 1989.

[AR89] H. Abramson and M. Rogers, editors. *Meta-Programming in Logic Programming.* MIT Press, 1989.

[Bar92] C. Baral. Generalized Negation As Failure and Semantics of Normal Disjunctive Logic Programs. In A Voronkov, editor, *Proceedings of International Conference on Logic Programming and Automated Reasoning, St. Petersburg*, pages 309–319, 1992.

[Bar94] C. Baral. Rule based updates on simple knowledge bases. In *Proc. of AAAI94, Seattle*, pages 136–141, August 1994.

[Bar95] C. Baral. Reasoning about Actions : Non-deterministic effects, Constraints and Qualification. In C. Mellish, editor, *Proc. of IJCAI 95*, pages 2017–2023. Morgan Kaufmann, 1995.

[Bar97a] C. Baral. Embedding revision programs in logic programming situation calculus. *Journal of Logic Programming*, 30(1):83–97, Jan 1997.

[Bar97b] C. Baral. Relating logic programming theories of action and partial order planning. *Annals of Math and AI*, 21(2-4):131–153, 1997.

[Bar00] C. Baral. Abductive reasoning through filtering. *Artificial Intelligence Journal*, 120(1):1–28, 2000.

[BDNT01] S. Brass, J. Dix, I. Niemela, and Przymusinski. T. On the equivalence of the static and disjunctive well-founded semantics and its computation. *TCS*, 258(1-2):523–553, 2001.

[BDT96] S. Brass, J. Dix, and Przymusinski T. Super logic programs. In *Proc. of KR*, pages 529–540, 1996.

[BED92] R. Ben-Eliyahu and R. Dechter. Propositional semantics for disjunctive logic programs. In *Proceedings of the 1992 Joint International Conference and Symposium on Logic Programming*, pages 813–827, 1992.

[BEL00] Y. Babovich, E. Erdem, and V. Lifschitz. Fages' theorem and answer set programming. In *Proc. International workshop on non-monotonic reasoning*, 2000.

[BF91] N. Bidoit and C. Froidevaux. General logical databases and programs: Default logic semantics and stratification. *Journal of Information and Computation*, 91(1):15–54, 1991.

[BG93] C. Baral and M. Gelfond. Representing concurrent actions in extended logic programming. In *Proc. of 13th International Joint Conference on Artificial Intelligence, Chambery, France*, pages 866–871, 1993.

[BG94] C. Baral and M. Gelfond. Logic programming and knowledge representation. *Journal of Logic Programming*, 19,20:73–148, 1994.

[BG97] C. Baral and M. Gelfond. Reasoning about effects of concurrent actions. *Journal of Logic Programming*, 31(1-3):85–117, May 1997.

[BG00] C. Baral and M. Gelfond. Reasoning agents in dynamic domains. In J Minker, editor, *Logic Based AI.* Kluwer, 2000.

[BGK93] C. Baral, M. Gelfond, and O. Kosheleva. Approximating general logic programs. In *Proceedings of International Logic Programming Symposium*, pages 181–198, 1993.

[BGK98] C. Baral, M. Gelfond, and O. Kosheleva. Expanding queries to incomplete databases by interpolating general logic programs. *Journal of Logic Programming*, 35:195–230, 1998.

[BGN+01] M. Balduccini, M. Gelfond, M. Nogueira, R. Watson, and M. Barry. An a-prolog decision support system for the space shuttle, 2001. submitted for publication.

[BGP97] C. Baral, M. Gelfond, and A. Provetti. Representing Actions: Laws, Observations and Hypothesis. *Journal of Logic Programming*, 31(1-3):201–243, May 1997.

[BGW99] C. Baral, M. Gelfond, and R. Watson. Reasoning about actual and hypothetical occurrences of concurrent and non-deterministic actions. In B. Fronhofer and R. Pareschi, editors, *Theoretical approaches to dynamical worlds*, pages 73–110. Kluwer Academic, 1999.

[BK82] K. Bowen and R. Kowalski. Amalgamating language and metalanguage in logic programming. In K.L. Clark and S.A. Tarnlund, editors, *Logic Programming*, pages 153–173. Academic Press, 1982.

[BL97] C. Baral and J. Lobo. Defeasible specification in action theories. In *IJCAI 97*, pages 1441–1446, 1997.

[BLM91] C. Baral, J. Lobo, and J. Minker. WF^3: A Semantics for Negation in Normal Disjunctive Logic Programs with Equivalent Proof Methods. In *Proceedings of ISMIS 91*, pages 459–468, Dept of Computer Science, University of Maryland, October 1991. Springer-Verlag.

[BLM92] C. Baral, J. Lobo, and J. Minker. Generalized disjunctive well-founded semantics for logic programs. *Annals of Math and Artificial Intelligence*, 5:89–132, 1992.

[BM90] A. Bonner and L. McCarty. Adding negation as failure to intuitionsitic logic programming. In S. Debray and M. Hermenegildo, editors, *Logic Programming: Proc. of the 1990 North American Conf.*, pages 681–703, 1990.

[BMPT92] A. Brogi, P. Mancarella, D. Pedreschi, and F. Turini. Meta for modularising logic programming. In *Proc. META-92*, pages 105–119, 1992.

[BMS95] H. Blair, W. Marek, and J. Schlipf. The expressiveness of locally stratified programs. *Annals of Mathematics and Artificial Intelligence*, 15(2):209–229, 1995.

[BNNS94] C. Bell, A. Nerode, R. Ng, and V.S. Subrahmanian. Mixed integer programming methods for computing non-monotonic deductive databases. *Journal of ACM*, 41(6):1178–1215, 1994.

[BR86] F. Bancilhon and R. Ramakrishnan. An amateur's introduction to recursive query processing strategies. *Proc. of ACM SIGMOD '86*, pages 16–52, May 28-30, 1986.

[BR87] C. Beeri and R. Ramakrishnan. On the power of magic. *Proc. Principles of Database Systems*, March 1987.

[Bre91] G. Brewka. Cumulative default logic: in defense of nonmonotonic inference rules. *Artificial Intelligence*, 50:183–205, 1991.

[Bre94] G Brewka. Reasoning about priorities in default logic. In *AAAI 94*, 1994.

[BS91] C. Baral and V. S. Subrahmanian. Duality between alternative semantics of logic programs and nonmonotonic formalisms. *In the International Workshop in logic programming and nonmonotonic reasoning - pages 69-86- and to appear in Journal of Automated Reasoning*, 1991.

[BS92] C. Baral and V. S. Subrahmanian. Stable and Extension Class Theory for Logic Programs and Default Logics. *Journal of Automated Reasoning*, 8:345–366, 1992.

[BS93] C. Baral and V. S. Subrahmanian. Duality between alternative semantics of logic programs and nonmonotonic formalisms. *Journal of Automated Reasoning*, 10:399–420, 1993.

[BS97] C. Baral and T. Son. Approximate reasoning about actions in presence of sensing and incomplete information. In *Proc. of International Logic Programming Symposium (ILPS 97)*, pages 387–401, 1997.

[BS99] C. Baral and T. Son. Adding hierarchical task networks to congolog. In *Proc. of Agent, theories and Languages (ATAL) 99 (won one of the two best paper awards)*, 1999.

[BW99] M. Barry and R. Watson. Reasoning about actions for spacecraft redundancy management. In *Proceedings of the 199 IEEE Aerospace conference*, volume 5, pages 101–112, 1999.

[Cav89] L. Cavedon. Continuity, consistency, and completeness properties for logic programs. In Giorgio Levi and Maurizio Martelli, editors, *Logic Programming: Proc. of the Sixth Int'l Conf.*, pages 571–584, 1989.

[CDS94] M. Cadoli, F. Donini, and M. Schaerf. Is intractability of non-monotonic reasoning a real drawback. In *AAAI*, volume 2, pages 946–951, 1994.

[CDS96] M. Cadoli, F. Donini, and M. Schaerf. Is intractability of non-monotonic reasoning a real drawback. *Artificial Intelligence*, 88(1-2):215–251, 1996.

[CEF+97] 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 application front ends. In *Proceedings of the 12th Workshop on Logic Programming*, pages 128–137, 1997.

[CEG94] M. Cadoli, T. Eiter, and G. Gottlob. Default logic as a query language. In *KR*, pages 99–108, 1994.

[CEG97] M. Cadoli, T. Eiter, and G. Gottlob. Default logic as a query language. *IEEE TKDE*, 9(3):448–463, 1997.

[CF90] A. Cortesi and G. File. Graph properties for normal logic programs. In *Proc. of GULP 90*, 1990.

[CH85] A. Chandra and D. Harel. Horn clause queries and generalizations. *Journal of Logic Programming*, 2(1):1–5, 1985.

[Che93] J. Chen. Minimal knowledge + negation as failure = only knowing (sometimes). In *Proceedings of the Second Int'l Workshop on Logic Programming and Non-monotonic Reasoning, Lisbon*, pages 132–150, 1993.

[CKW93] W. Chen, M. Kifer, and D.S. Warren. A foundation for higher-order logic programming. *Journal of Logic Programming*, 15(3):187–230, 1993.

[CL89] S. Costantini and G.A. Lanzarone. A metalogic programming language. In G. Levi and M. Martelli, editors, *Proc ICLP'89*, pages 218–233, 1989.

[Cla78] K. Clark. Negation as failure. In Herve Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, New York, 1978.

[CM81] W. Clockin and C. Mellish. *Programming in Prolog.* Springer, 1981.

[CMMT95] P. Cholewinski, W. Marek, A. Mikitiuk, and M. Truszczynski. Experimenting with nonmonotonic reasoning. In *ICLP*, pages 267–281, 1995.

[CMT96] P. Cholewiński, W. Marek, and M. Truszczyński. Default reasoning system deres. In L. Aiello, J. Doyle, and S. Shapiro, editors, *Proc. of KR 96*, pages 518–528. Morgan Kaufmann, 1996.

[CS92] M. Cadoli and M. Schaerf. A survey on complexity results for nonmonotonic logics. Technical report, University di Roma "La Sapienza", Dipartiment o di Informatica e sistemistica, Roma, Italy, 1992.

[CSW95] W. Chen, T. Swift, and D. Warren. Efficient top-down computation of queries under the well-founded semantics. *Journal of Logic Programming*, 24(3):161–201, 1995.

[CW96] W. Chen and D. Warren. Computation of stable models and its integration with logical query processing. *IEEE TKDE*, 8(5):742–757, 1996.

[DEGV97] E. Dantsin, T. Eiter, G. Gottlob, and A. Voronkov. Complexity and expressive power of logic programming. In *Proc. of 12th annual IEEE conference on Computational Complexity*, pages 82–101, 1997.

[DEGV99] E. Dantsin, T. Eiter, G. Gottlob, and A. Voronkov. Complexity and expressive power of logic programming. Technical Report INFSYS 1843-99-05, Technische Universitat Wien, 1999.

[DG84] W. Dowling and H. Gallier. Linear time algorithms for testing the satisfiability of propositional horn formulae. *Journal of Logic Programming*, 1:267–284, 1984.

[Dix91] J. Dix. Classifying semantics of logic programs. In *Proceedings of* International Workshop in logic programming and nonmonotonic reasoning, *Washington D.C.*, pages 166–180, 1991.

[Dix92a] J. Dix. Classifying semantics of disjunctive logic programs. In *JICSLP*, pages 798–812, 1992.

[Dix92b] J. Dix. A framework for representing and characterizing semantics of logic programs. In *KR*, pages 591–602, 1992.

[Dix95a] J. Dix. A classification theory of semantics of normal logic programs: I. strong properties. *Fundamenta Informaticae*, 22(3):227–255, 1995.

[Dix95b] J. Dix. A classification theory of semantics of normal logic programs: Ii. weak properties. *Fundamenta Informaticae*, 22(3):257–288, 1995.

[DNK97] Y. Dimopoulos, B. Nebel, and J. Koehler. Encoding planning problems in nonmonotonic logic programs. In *Proc. of European conference on Planning*, pages 169–181, 1997.

[Doy79] J. Doyle. A truth-maintenance system. *Artificial Intelligence*, 12:231–272, 1979.

[Dun91] P. Dung. Well-founded reasoning with classical negation. In *Proc. of 1st international workshop on logic programming and non-monotonic reasoning*, 1991.

[Dun92] P. Dung. On the relations between stable and well-founded semantics of logic programs. *Theoretical Computer Science*, 105:7–25, 1992.

[Dun93] P. Dung. On the acceptability of arguments and its fundamental role in nonmonotonic reasoning and logic programming. In *Proc. of IJCAI 93*, pages 852–857, 1993.

[DV97] E. Dantsin and A. Voronkov. Complexity of query answering in logic databases with complex values. In S. Adian and A. Nerode, editors, *Proc. of 4th Intenational Symposium on logical foundations of computer science (LFCS'97)*, pages 56–66, 1997.

[EFG+00] T. Eiter, W. Faber, G. Gottlob, C. Koch, C. Mateis, N. Leone, G. Pfeifer, and F. Scarcello. The dlv system. In J. Minker, editor, *Pre-prints of Workshop on Logic-Based AI*, 2000.

[EG93a] T. Eiter and G. Gottlob. Complexity aspects of various semantics for disjunctive databases. In *PODS*, pages 158–167, 1993.

[EG93b] T. Eiter and G Gottlob. Complexity results for disjunctive logic programming and application to nonmonotonic logics. In D. Miller, editor, *Proceedings of the International Logic Programming Symposium (ILPS), Vancouver*, pages 266–178. MIT Press, 1993.

[EG93c] T. Eiter and G. Gottlob. Propositional Circumscription and Extended Closed World Reasoning are $\mathbf{\Pi}_2^P$-complete. *Theoretical Computer Science*, 114(2):231–245, 1993. Addendum 118:315.

[EG95] T. Eiter and G. Gottlob. On the computational cost of disjunctive logic programming: propositional case. *Journal of Logic Programming*, 15(3-4):289–323, 1995.

[EG97] T. Eiter and G. Gottlob. Expressiveness of stable model semantics for disjunctive logic programs with functions. *JLP*, 33(2):167–178, 1997.

[EGL97] T. Eiter, G. Gottlob, and N. Leone. Semantics and complexity of abduction from default theories. *Artificial Intelligence*, 90:177–223, 1997.

[EGM94] T. Eiter, G. Gottlob, and H. Mannila. Adding disjunction to datalog. In *PODS*, pages 267–278, 1994.

[EGM97] T. Eiter, G. Gottlob, and H. Mannila. Disjunctive datalog. *ACM TODS*, 22(3):364–418, 1997.

[EL99] E. Erdem and V. Lifschitz. Transformations of logic programs related to causality and planning. In *LPNMR*, pages 107–116, 1999.

[Elk90] C. Elkan. A rational reconstruction of non-monotonic truth maintenance systems. *Artificial Intelligence*, 43, 1990.

[Esh90] K. Eshghi. Computing Stable Models by Using the ATMS. In *Proc. AAAI-90*, pages 272–277, 1990.

[Fag74] R. Fagin. Generalized first-order spectra and polynomial-time recognizable sets. In R. Karp, editor, *Complexity of Computation*, pages 43–74. AMS, 1974.

[Fag90] F. Fages. Consistency of Clark's completion and existence of stable models. Technical Report 90-15, Ecole Normale Superieure, 1990.

[Fag94] F. Fages. Consistency of clark's completion and existence of stable models. *Journal of Methods of Logic in Computer Science*, 1:51–60, 1994.

[FBJ88] M. Fitting and M. Ben-Jacob. Stratified and Three-Valued Logic programming Semantics. In R.A. Kowalski and K.A. Bowen, editors, *Proc.* 5^{th} *International Conference and Symposium on Logic Programming*, pages 1054–1069, Seattle, Washington, August 15-19, 1988.

[FH89] Y. Fujiwara and S. Honiden. Relating the tms to autoepistemic logic. In *Proc. IJCAI-89*, pages 1199–1205, 1989.

[Fit85] M. Fitting. A Kripke-Kleene semantics for logic programs. *Journal of Logic Programming*, 2(4):295–312, 1985.

[Fit86] M. Fitting. Partial models and logic programming. *Theoretical Computer Science*, 48:229–255, 1986.

[Fit91] M. Fitting. Well-founded semantics, generalized. In *Proceedings of* International Symposium on Logic Programming*, San Diego*, pages 71–84, 1991.

[FL01] P. Ferraris and V. Lifschitz. Weight constraints as nested expressions, 2001. unpublished draft (http://www.cs.utexas.edu/users/vl/papers.html).

[FLMS93] J. Fernandez, J. Lobo, J. Minker, and V. S. Subrahmanian. Disjunctive LP + integrity constraints = stable model semantics. *Annals of Math and AI*, 18:449–479, 1993.

[FLP99] W. Faber, N. Leone, and G. Pfeifer. Pushing goal derivation in dlp computations. In M. Gelfond, N. Leone, and G. Pfeifer, editors, *Proc. of LPNMR 99*, pages 177–191. Springer, 1999.

[Gel87] M. Gelfond. On stratified autoepistemic theories. In *Proc. AAAI-87*, pages 207–211, 1987.

[Gel89] A. Van Gelder. The alternating fixpoint of logic programs with negation. In *Proceedings of the Symposium on Principles of Database Systems*, 1989.

[Gel90] M. Gelfond. Belief sets, deductive databases and explanation based reasoning. Technical report, University of Texas at El Paso, 1990.

[Gel91a] M. Gelfond. Epistemic semantics for disjunctive databases. Preprint, ILPS Workshop on Disjunctive Logic Programs, San Diego, Ca., 1991.

[Gel91b] M. Gelfond. Strong introspection. In *Proc. AAAI-91*, pages 386–391, 1991.

[Gel94] M. Gelfond. Logic programming and reasoning with incomplete information. *Annals of Mathematics and Artificial Intelligence*, 12:19–116, 1994.

[GG97] M. Gelfond and A. Gabaldon. From functional specifications to logic programs. In J. Maluszynski, editor, *Proc. of International symposium on logic programming*, pages 355–370, 1997.

[GG99] M. Gelfond and A. Gabaldon. Building a knowledge base: An example. *Annals of Mathematics and Artificial Intelligence*, 25(3-4):165–199, 1999.

[GL88] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In R. Kowalski and K. Bowen, editors, *Logic Programming: Proc. of the Fifth Int'l Conf. and Symp.*, pages 1070–1080. MIT Press, 1988.

[GL89] M. Gelfond and V. Lifschitz. Compiling circumscriptive theories into logic programs. In M Reinfrank, Johan de Kleer, M Ginsberg, and Erik Sandewall, editors, *Non-Monotonic Reasoning: 2nd International Workshop (Lecture Notes in Artificial Intelligence 346)*, pages 74–99. Springer-Verlag, 1989.

[GL90] M. Gelfond and V. Lifschitz. Logic programs with classical negation. In D. Warren and Peter Szeredi, editors, *Logic Programming: Proc. of the Seventh Int'l Conf.*, pages 579–597, 1990.

[GL91] M. Gelfond and V. Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9:365–387, 1991.

[GL92] M. Gelfond and V. Lifschitz. Representing actions in extended logic programs. In K. Apt, editor, *Joint International Conference and Symposium on Logic Programming.*, pages 559–573. MIT Press, 1992.

[GL93] M. Gelfond and V. Lifschitz. Representing actions and change by logic programs. *Journal of Logic Programming*, 17(2,3,4):301–323, 1993.

[GLPT91] M. Gelfond, V Lifschitz, H. Przymusińska, and M. Truszczyński. Disjunctive defaults. In J. Allen, R. Fikes, and E. Sandewall, editors, *Principles of Knowledge Representation and Reasoning: Proc. of the Second Int'l Conf.*, pages 230–237, 1991.

[GM90] L. Giordano and A. Martelli. Generalized stable models, truth maintainance and conflict resolution. In *Proc. of the Seventh International*, pages 427–441. The MIT Press, 1990.

[GMN+96] G. Gottlob, S. Marcus, A. Nerode, G. Salzer, and V. S. Subrahmanian. A non-ground realization of the stable and well-founded semantics. *TCS*, 166(1-2):221–262, 1996.

[GO92] L. Giordano and N. Olivetti. Negation as failure in intuitionistic logic programming. In *Proc. of the Joint International Conference and Symposium on Logic Programming*, pages 430–445. The MIT Press, 1992.

[Got94] G. Gottlob. Complexity and expressive power of disjunctive logic programming. In *SLP*, pages 23–42, 1994.

[Got95] G. Gottlob. Translating default logic into standard autoepistemic logic. *JACM*, 42(4):711–740, 1995.

[GP91] M. Gelfond and H. Przymusińska. Definitions in epistemic specifications. In A. Nerode, W. Marek, and Subrahmanian V. S., editors, *Logic Programming and Non–monotonic Reasoning: Proc. of the First Int'l Workshop*, pages 245–259, 1991.

[GP93] M. Gelfond and H. Przymusinska. Reasoning in open domains. In L. Pereira and A. Nerode, editors, *Proceedings of the Second International Workshop in Logic Programming and Nonmonotonic Reasoning*, pages 397–413, 1993.

[GP96] M. Gelfond and H. Przymusinska. Towards a theory of elaboration tolerance: logic programming approach. *Journal of Software and Knowledge Engineering*, 6(1):89–112, 1996.

[GPP89] M. Gelfond, H. Przymusinska, and T. Przymusinski. On the relationship between circumscription and negation as failure. *Artificial Intelligence*, 38(1):75–94, 1989.

[GS91] P. Gardner and J. Shepherdson. Unfold/Fold transformation in logic programs. In J-L. Lassez and G. Plotkin, editors, *Computational Logic: Essays in honor of Alan Robinson*, pages 565–583. MIT press, 1991.

[GS97a] M. Gelfond and T. Son. Reasoning with prioritized defaults. In J. Dix, L. Pereira, and T. Przymusinski, editors, *Proc. of the Workshop on Logic Programming and Knowledge Representation*, volume LNCS 1471, pages 89–109. Springer, 1997.

[GS97b] S. Greco and D. Sacca. Deterministic semantics for datalog¬: Complexity and expressive power. In *DOOD*, pages 337–350, 1997.

[GSZ93] S. Greco, D. Sacca, and Carlo Zaniolo. Dynamic programming optimization for logic queries with aggregates. In *ILPS*, pages 575–589, 1993.

[GT93] M. Gelfond and B. Traylor. Representing null values in logic programs. In *Workshop on Logic Programming with incomplete information, Vancouver, BC.*, 1993.

[Hel99] K. Heljanko. Using logic programs with stable model semantics to solve deadlock and reachability problems for 1-safe petri nets. *Fundamenta Informaticae*, 37(3):247–268, 1999.

[Hen00] M. Henz. Scheduling a major college basketball conference— revisited, 2000.

[HL91] P. Hill and J. Lloyd. The godel report. Technical Report TR-91-02, University of Bristol, 1991.

[IM87] A. Itai and J. Makowsky. Unification as a complexity measure for logic programming. *Journal of logic programming*, 4(105-117), 1987.

[Imm86] N. Immerman. Relational queries computable in polynomial time. *Information and Control*, 68:86–104, 1986.

[Ino91] K. Inoue. Extended logic programs with default assumptions. In *Proc. of ICLP91*, 1991.

[IS91] N. Iwayama and K. Satoh. Computing abduction using the TMS. In *Proc. of ICLP 91*, pages 505–518, 1991.

[IS98] K. Inoue and C. Sakama. Negation as failure in the head. *JLP*, 35(1):39–78, 1998.

[JK91] U. Junker and K. Konolige. Computing the extensions of autoepistemic and default logics with a truth maintenance systems. In *Proc. of AAAI 90*, pages 278–283, 1991.

[JL77] N. Jones and W. Lasser. Complete problems in deterministic polynomial time. *Theoretical Computer Science*, 3:105–117, 1977.

[KL99] C. Koch and N. Leone. Stable model checking made easy. In *IJCAI*, pages 70–75, 1999.

[KLM90] S. Kraus, D. Lehman, and M. Magidor. Nonmonotonic reasoning, preferential models and cumulative logics. *Artificial Intelligence*, 44(1):167–207, 1990.

[KM90] A. Kakas and P. Mancarella. Generalized stable models: a semantics for abduction. In *Proc. of ECAI-90*, pages 385–391, 1990.

[Kow79] R. Kowalski. *Logic for Problem Solving*. North-Holland, 1979.

[Kow90] R. Kowalski. Problems and promises of computational logic. In J. Lloyd, editor, *Computational Logic: Symposium Proceedings*, pages 80–95. Springer, 1990.

[KP88] P. Kolaitis and C. Papadimitriou. Why not negation by fixpoint? In *PODS*, pages 231–239, 1988.

[KP91] P. Kolaitis and C. Papadimitriou. Why not negation by fixpoint? *JCSS*, 43(1):125–144, 1991.

[KS90] R. Kowalski and F. Sadri. Logic programs with exceptions. In D. Warren and Peter Szeredi, editors, *Logic Programming: Proc. of the Seventh Int'l Conf.*, pages 598–613, 1990.

[KS92] H. Kautz and B. Selman. Planning as satisfiability. In *Proc. of ECAI-92*, pages 359–363, 1992.

[Kun87] K. Kunen. Negation in logic programming. *Journal of Logic Programming*, 4(4):289–308, 1987.

[Kun89] K. Kunen. Signed data dependencies in logic programs. *Journal of Logic Programming*, 7(3):231–245, 1989.

[KV95] P. Kolaitis and M. Vardi. On the expressive power of datalog: Tools and a case study. *JCSS*, 51(1):110–134, 1995.

[Lif85a] V. Lifschitz. Closed-world data bases and circumscription. *Artificial Intelligence*, 27, 1985.

[Lif85b] V. Lifschitz. Computing circumscription. In *Proc. of IJCAI-85*, pages 121–127, 1985.

[Lif88] V. Lifschitz. On the declarative semantics of logic programs with negation. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 177–192. Morgan Kaufmann, San Mateo, CA., 1988.

[Lif89] V. Lifschitz. Between circumscription and autoepistemic logic. In R. Brachman, H. Levesque, and R. Reiter, editors, *Proc. of the First Int'l Conf. on Principles of Knowledge Representation and Reasoning*, pages 235–244, 1989.

[Lif90] V. Lifschitz. On open defaults. In J. Lloyd, editor, *Computational Logic: Symposium Proceedings*, pages 80–95. Springer, 1990.

[Lif91] V. Lifschitz. Nonmonotonic databases and epistemic queries: Preliminary report. In *Proceedings of International Joint Conference on Artificial Intelligence*, pages 381–386, Sydney, Australia, 1991.

[Lif93a] V. Lifschitz. A language for representing actions. In *Working Papers of the Second Int'l Symp. on Logical Formalizations of Commonsense Knowledge*, 1993.

[Lif93b] V. Lifschitz. Restricted monotonicity. In *AAAI*, pages 432–437, 1993.

[Lif94] V. Lifschitz. Circumscription. In D. Gabbay, C. Hogger, and J. Robinson, editors, *Handbook of Logic in AI and Logic Programming*, volume 3, pages 298–352. Oxford University Press, 1994.

[Lif95] V. Lifschitz. SLDNF, Constructive Negation and Grounding. In *ICLP*, pages 581–595, 1995.

[Lif96] V. Lifschitz. Foundations of declarative logic programming. In G. Brewka, editor, *Principles of Knowledge Representation*, pages 69–128. CSLI Publications, 1996.

[Lif97] V. Lifschitz, editor. *Special issue of the Journal of Logic Programming on Reasoning about actions and change*, volume 31(1-3), May 1997.

[Lif99a] V. Lifschitz. Action languages, answer sets and planning. In K. Apt, V. Marek, M. Trucszczynski, and D. Warren, editors, *The Logic Programming Paradigm: a 25-Year perspective*, pages 357–373. Springer, 1999.

[Lif99b] V. Lifschitz. Answer set planning. In *Proc. International Conf. on Logic Programming*, pages 23–37, 1999.

[Llo84] J. Lloyd. *Foundations of logic programming.* Springer, 1984.

[Llo87] J. Lloyd. *Foundations of logic programming.* Springer, 1987. Second, extended edition.

[LM85] J. Lassez and M. Maher. Optimal fixedpoints of logic programs. *Theoretical Computer Science*, 39:15–25, 1985.

[LMR92] J. Lobo, J. Minker, and A. Rajasekar. *Foundations of disjunctive logic programming.* The MIT Press, 1992.

[LMT93] V. Lifschitz, N. McCain, and H. Turner. Automation of reasoning about action: a logic programming approach. In *Posters of the International Symposium on Logic Programming*, 1993.

[LPV00] V. Lifschitz, D. Pearce, and A. Valverde. Strongly equivalent programs, 2000. draft.

[LRS96a] N. Leone, P. Rullo, and F. Scarcello. On the computation of disjunctive stable models. In *DEXA*, pages 654–666, 1996.

[LRS96b] N. Leone, P. Rullo, and F. Scarcello. Stable model checking for disjunctive logic programs. In *Logic in Databases*, pages 265–278, 1996.

[LS90] F. Lin and Y. Shoham. Epistemic semantics for fixed-points nonmonotonic logics. In R. Parikh, editor, *Theoretical Aspects of Reasoning and Knowledge: Proc. of the Third Conf.*, pages 111–120, Stanford University, Stanford, CA, 1990.

[LT94] V. Lifschitz and H. Turner. Splitting a logic program. In Pascal Van Hentenryck, editor, *Proc. of the Eleventh Int'l Conf. on Logic Programming*, pages 23–38, 1994.

[LT95] V. Lifschitz and H. Turner. From disjunctive programs to abduction. In *Non-monotonic extensions of logic programming (Lecture notes in AI)*, pages 23–42, 1995.

[LT99] V. Lifschitz and H. Turner. Representing transition systems by logic programs. In *LPNMR*, pages 92–106, 1999.

[LTT99] V. Lifschitz, L. Tang, and Hudson Turner. Nested expressions in logic programs. *Annals of Mathematics and Artificial Intelligence*, 25(3-4):369–389, 1999.

[Luk84] W. Lukaszewicz. Considerations on default logic. In R. Reiter, editor, *Proc. of the international workshop on non-monotonic reasoning*, pages 165–193, 1984.

[LW92] V. Lifschitz and T. Woo. Answer sets in general nonmonotonic reasoning. In *Proc. of the Third Int'l Conf. on Principles of Knowledge Representation and Reasoning*, pages 603–614, 1992.

[LY91] L. Li and J. You. Making default inferences from logic programs. *Journal of Computational Intelligence*, 7:142–153, 1991.

[Mah87] M. Maher. Correctness of a logic program transformation system. Technical Report IBM Research Report RC 13496, IBM TJ Watson Research Center, 1987.

[Mah88] M. Maher. Equivalences of logic programs. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 627–658. Morgan Kaufmann, San Mateo, CA., 1988.

[Mah90] M. Maher. Reasoning about stable models (and other unstable semantics). Technical report, IBM TJ Watson Research Center, 1990.

[Mah93a] M. Maher. A logic programming view of clp. In *Proc. of ICLP*, pages 737–753, 1993.

[Mah93b] M. Maher. A transformation system for deductive database modules with perfect model semantics. *Theoretical computer science*, 110:377–403, 1993.

[Mak94] D. Makinson. General patterns in nonmonotonic reasoning. In D. Gabbay, C. Hogger, and J. Robinson, editors, *Handbook of Logic in AI and Logic Programming*, volume 3. Oxford University Press, 1994.

[McC80] J. McCarthy. Circumscription—a form of non-monotonic reasoning. *Artificial Intelligence*, 13(1, 2):27–39,171–172, 1980.

[McD82] D. McDermott. Nonmonotonic logic II: Nonmonotonic modal theories. *Journal of the ACM.*, 29(1):33–57, 1982.

[MD80] D. McDermott and J. Doyle. Nonmonotonic logic I. *Artificial Intelligence*, 13(1,2):41–72, 1980.

[MDS92a] B. Martens and D. De Schreye. A Perfect Herbrand Semantics for Untyped Vanilla Meta-Programming. In *Proc. of the Joint International Conference and Symposium on Logic Programming*. The MIT Press, 1992.

[MDS92b] B. Martens and D. De Schreye. Why untyped non-ground meta-programming is not (much of) a problem. Technical report, Department of Comp. Science, Katholieke Universiteit Leuven, Belgium, 1992.

[Mil86] D. Miller. A theory of modules in logic programming. In *Proc. of IEEE Symposium on Logic Programming*, pages 106–114, 1986.

[Min88a] J. Minker, editor. *Foundations of Deductive Databases and Logic Programming*. Morgan Kaufman, Washington DC, 1988.

[Min88b] J. Minker, editor. *Foundations of Deductive Databases and Logic Programming*. Morgan Kaufmann Pub., 1988.

[Min93] J. Minker. An overview of nonmonotonic reasoning and logic programming. *Journal of Logic Programming (special issue on nonmonotonic reasoning and logic programming)*, 17(2-4), 1993.

[MM82] A. Martelli and Montanari. An efficient unification algorithm. *ACM transaction on programming languages and systems*, 4:258–282, 1982.

[MMZ+01] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient sat solver. In *Design Automation Conference*, 2001.

[MNR94] W. Marek, A. Nerode, and J. Remmel. The stable models of a predicate logic program. *Journal of Logic programming*, 21:129–153, 1994.

[Moo85] R. Moore. Semantical considerations on nonmonotonic logic. *Artificial Intelligence*, 25(1):75–94, 1985.

[MR93] J. Minker and C. Ruiz. On extended disjunctive logic programs. In *ISMIS*, 1993. invited paper.

[MS89] W. Marek and V.S. Subrahmanian. The relationship between logic program semantics and non–monotonic reasoning. In G. Levi and M. Martelli, editors, *Proc. of the Sixth Int'l Conf. on Logic Programming*, pages 600–617, 1989.

[MSS99] J. Marques-Silva and K. Sakallah. GRASP: a search algorithm for propositional satisfiability. *IEEE transactions on computers*, 48:506–521, 1999.

[MT89] W. Marek and M. Truszczyński. Stable semantics for logic programs and default reasoning. In E. Lusk and R. Overbeek, editors, *Proc. of the North American Conf. on Logic Programming*, pages 243–257. MIT Press, 1989.

[MT91] W. Marek and M. Truszczyński. Autoepistemic logic. *Journal of the ACM.*, 3(38):588–619, 1991.

[MT93] W. Marek and M. Truszczyński. *Nonmonotonic Logic: Context dependent reasoning.* Springer, 1993.

[MT94a] W. Marek and M. Truszczyński. Revision programming, database updates and integrity constraints. In *In 5th International conference in Database theory, Prague*, 1994.

[MT94b] N. McCain and H. Turner. Language independence and language tolerance in logic programs. In *Proc. of the Eleventh Intl. Conference on Logic Programming*, pages 38–57, 1994.

[MT95] N. McCain and H. Turner. A causal theory of ramifications and qualifications. In *Proc. of IJCAI 95*, pages 1978–1984, 1995.

[MT99] W. Marek and M Truszczyński. Stable models and an alternative logic programming paradigm. In K. Apt, V. Marek, M. Trucszczynski, and D. Warren, editors, *The Logic Programming Paradigm: a 25-Year perspective*, pages 375–398. Springer, 1999.

[Myc83] A. Mycroft. Logic programs and many valued logics. In *Proc. of the 1st STACS conference*, 1983.

[Nie99] I. Niemela. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, 25(3-4):241–271, 1999.

[NMS91] A. Nerode, W. Marek, and V. S. Subrahamanian, editors. *Logic Programming and Nonmonotonic Reasoning: Proceedings of the First International Workshop.* MIT Press, 1991.

[NS96] I. Niemela and P. Simons. Efficient implementation of the well-founded and stable model semantics. In *Proc. Joint international conference and symposium on Logic programming*, pages 289–303, 1996.

[NS97] I. Niemela and P. Simons. Smodels – an implementation of the stable model and well-founded semantics for normal logic programs. In J. Dix, U. Furbach, and A. Nerode, editors, *Proc. 4th international conference on Logic programming and non-monotonic reasoning*, pages 420–429. Springer, 1997.

[NS98] I. Niemela and T. Soininen. Formalizing configuration knowledge using rules with choices. In *Proc of workshop on Formal Aspects and Applications of Nonmonotonic Reasoning, Trento, Italy*, 1998.

[NSS99] I. Niemela, P. Simons, and T. Soininen. Stable model semantics of weight constraint rules. In *LPNMR*, pages 317–331, 1999.

[O'K90] R. O'Keefe. *The craft of Prolog.* MIT press, 1990.

[PA93] L. Pereira and J. Alferes. Optative reasoning with scenario semantics. In *Proceedings of ICLP 93, Hungary*, pages 601–619, 1993.

[PAA91a] L. Pereira, J. Aparicio, and J. Alferes. Contradiction removal within well-founded semantics. In Anil Nerode, Victor Marek, and Subrahmanian V. S., editors, *Logic Programming and Non–monotonic Reasoning: Proc. of the First Int'l Workshop*, pages 105–119. MIT Press, 1991.

[PAA91b] L. Pereira, J. Aparicio, and J. Alferes. Non-monotonic reasoning with well-founded semantics. In *Proc. of the Eight International Logic Programming Conference*, pages 475–489, 1991.

[PAA92a] L. Pereira, J. Alferes, and J. Aparicio. Default theory for well founded semantics with explicit negation. In D. Pearce and G. Wagner, editors, *Logic in AI, Proc. of European Workshop JELIA'92 (LNAI, 633)*, pages 339–356, 1992.

[PAA92b] L. Pereira, J. Alferes, and J. Aparicio. Well founded semantics for logic programs with explicit negation. In *Proc. of European Conference on AI*, 1992.

[PAA93] L. Pereira, J. Alferes, and J. Aparicio. Nonmonotonic reasoning with logic programming. *Journal of Logic Programming (special issue on nonmonotonic reasoning and logic programming)*, 17(2-4):227–263, 1993.

[Pap94] C. Papadimitriou. *Computational Complexity.* Addison-Wesley, 1994.

[PC89] S. Pimental and J. Cuadrado. A truth maintainance system based on stable models. In *Proc. North American Conference on Logic Programming*, 1989.

[Pet92] A. Pettorossi, editor. *Meta-programming in Logic.* Springer Verlag, June 1992.

[PN93] L. Pereira and A. Nerode, editors. *Logic Programming and Non-monotonic Reasoning: Proceedings of the Second International Workshop.* MIT Press, 1993.

[PP90] H. Przymusinska and T. Przymusinski. Weakly stratified logic programs. *Fundamenta Informaticae*, 13:51–65, 1990.

[PP92] H. Przymusinska and T. Przymusinski. Stationary default extensions. In *Proceedings of 4th International Workshop on Non-monotonic reasoning*, pages 179–193, 1992.

[Prz88a] T. Przymusinski. On the declarative semantics of deductive databases and logic programs. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 193–216. Morgan Kaufmann, San Mateo, CA., 1988.

[Prz88b] T. Przymusinski. Perfect model semantics. In R. Kowalski and K. Bowen, editors, *Logic Programming: Proc. of the Fifth Int'l Conf. and Symp.*, pages 1081–1096, 1988.

[Prz89a] T. Przymusinski. Every logic program has a natural stratification and an iterated least fixed point model. In *Proc. of Principles of Database Systems*, 1989.

[Prz89b] T. Przymusinski. On the declarative and procedural semantics of logic programs. *Journal of Automated Reasoning*, 5:167–205, 1989.

[Prz89c] T. Przymusinski. Three-valued formalizations of non-monotonic reasoning and logic programming. In R. Brachman, H. Levesque, and R. Reiter, editors, *Proc. of the First Int'l Conf. on Principles of Knowledge Representation and Reasoning*, pages 341–348, 1989.

[Prz89d] T. Przymusinski. The well-founded semantics coincides with the three-valued stable semantics. *Fundamenta Informaticae*, 1989.

[Prz90a] T. Przymusinski. Extended stable semantics for normal and disjunctive programs. In D. Warren and Peter Szeredi, editors, *Logic Programming: Proc. of the Seventh Int'l Conf.*, pages 459–477, 1990.

[Prz90b] T. Przymusinski. Stationary semantics for disjunctive logic programs and deductive databases. In *Proc. of NACLP*, pages 40–59, 1990.

[Prz90c] T. Przymusinski. The well-founded semantics coincides with the three-valued stable semantics. *Fundamenta Informaticae*, 13(4):445–463, 1990.

[Prz91] T. Przymusinski. Stable semantics for disjunctive programs. *New generation computing*, 9(3,4):401–425, 1991.

[Prz95] T. Przymusinski. Static semantics for normal and disjunctive logic programs. *Annals of Mathematics and Artificial Intelligence*, 14(2-4):323–357, 1995.

[PT95] T. Przymusinski and H. Turner. Update by means of inference rules. In *Proc. of Int'l Conf.on Logic Programming and non-monotonic reasoning*, May-June 1995.

[Rei78] R. Reiter. On closed world data bases. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 119–140. Plenum Press, New York, 1978.

[Rei80] R. Reiter. A logic for default reasoning. *Artificial Intelligence*, 13(1,2):81–132, 1980.

[Rei82] R. Reiter. Circumscription implies predicate completion (sometimes). In *Proc. of IJCAI-82*, pages 418–420, 1982.

[Rei87] R. Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32(1):57–95, 1987.

[RM89] R. Reiter and A. Mackworth. A logical framework for depiction and image interpretation. *Artificial Intelligence*, 41(2):125–156, 1989.

[RM90] A. Rajasekar and J. Minker. On stratified disjunctive programs. *Annals of Mathematics and Artificial Intelligence*, 1(1-4):339–357, 1990.

[Rob65] J. Robinson. A machine-oriented logic based on the resolution principle. *JACM*, 12(1):23–41, 1965.

[Ros89a] K. Ross. A procedural semantics for well founded negation in logic programming. In *Proc. of the eighth Symposium on Principles of Database Systems*, pages 22–34, 1989.

[Ros89b] K. Ross. The well founded semantics for disjunctive logic programs. In *Proc. of DOOD*, pages 385–402, 1989.

[Ros92] K. Ross. A procedural semantics for well-founded negation in logic programs. *JLP*, 13(1):1–22, 1992.

[RP91] W. Rodi and S. Pimentel. A nonmonotonic assumption-based tms using stable bases. In J. Allen, R. Fikes, and Erik Sandewall, editors, *Proc. of the Second Int'l Conf. on Principles of Knowledge Representation and Reasoning*, 1991.

[Sac93] D. Saccá. Multiple stable models are needed to solve unique solution problems. In *Informal Proc. of the Second Compulog Net meeting on Knowledge Bases (CNKBS 93)*, 1993.

[Sac97] D. Sacca. The expressive powers of stable models for bound and unbound datalog queries. *JCSS*, 54(3):441–464, 1997.

[Sak89] C. Sakama. Possible model semantics for disjunctive databases. In *Proc. of the first international conference on deductive and object oriented databases*, pages 1055–1060, 1989.

[Sat87] T. Sato. On the consistency of first-order logic programs. Technical report, ETL, TR-87-12, 1987.

[Sat90] T. Sato. Completed logic programs and their consistency. *Journal of logic programming*, 9(1):33–44, 1990.

[SBM01] T. Son, C. Baral, and S. McIlraith. Extending answer set planning with sequence, conditional, loop, non-deterministic choice, and procedure constructs. In *Proc. of AAAI Spring symposium on Answer Set Programming*, 2001.

[Sch87] J. Schlipf. Decidability and definability with circumscription. *Annals of pure and applied logic*, 35, 1987.

[Sch90] J. Schlipf. The expressive powers of the logic programming semantics. In *PODS*, pages 196–204, 1990.

[Sch91] G. Schwarz. Autoepistemic logic of knowledge. In Anil Nerode, Victor Marek, and Subrahmanian V. S., editors, *Logic Programming and Non–monotonic Reasoning: Proc. of the First Int'l Workshop*, pages 260–274, 1991.

[Sch92] J. Schlipf. Formalizing a logic for logic programming. *Annals of Mathematics and Artificial Intelligence*, 5:279–302, 1992.

[Sch93] J. Schlipf. Some remarks on computability and open domain semantics, 1993. manuscript.

[Sch95a] J. Schlipf. Complexity and undecidability results for logic programming. *Annals of Mathematics and Artificial Intelligence*, 15(3-4):257–288, 1995.

[Sch95b] J. Schlipf. The expressive powers of the logic programming semantics. *Journal of computer and system sciences*, 51(1):64–86, 1995.

[Sek91] H. Seki. Unfold/Fold transformation of stratified programs. *Theoretical Computer Science*, 86:107–139, 1991.

[Sek93] H. Seki. Unfold/Fold transformation for the well-founded semantics. *Journal of Logic programming*, 16:5–23, 1993.

[SGN99] T. Soininen, E. Gelle, , and I. Niemela. A fixpoint definition of dynamic constraint satisfaction. In *Proceedings of the Fifth International Conference on Principles and Practice of Constraint Programming, Alexandria, Virginia, USA*. Springer-Verlag, October 1999.

[Sha84] E. Shapiro. Alternation and the computational complexity of logic programs. *Journal of Logic Programming*, 1:19–33, 1984.

[Sha97] M. Shanahan. *Solving the frame problem: A mathematical investigation of the commonsense law of inertia*. MIT press, 1997.

[Sim99] P. Simons. Extending the stable model semantics with more expressive rules. In *Proc. of International Conference on Logic Programming and Nonmonotonic Reasoning, LPNMR'99*, 1999.

[Sim00] P. Simmons. *Extending and implementing the stable model semantics*. PhD thesis, Helsinki University of Technology, 2000.

[SN99] T. Soininen and I. Niemela. Developing a declarative rule language for applications in product configuration. In G. Gupta, editor, *Proc. of Practical Aspects of Declarative Languages '99*, volume 1551, pages 305–319. Springer, 1999.

[SNV95] V. S. Subrahmanian, D. Nau, and C. Vago. Wfs + branch and bound = stable models. *IEEE Transactions on Knowledge and Data Engineering*, 7(3):362–377, 1995.

[SS86] L. Sterling and E. Shapiro. *The art of Prolog*. MIT press, 1986.

[Ste90] L. Sterling. *The practice of Prolog*. MIT press, 1990.

[Str93] K. Stroetman. A Completeness Result for SLDNF-Resolution. *Journal of Logic Programming*, 15:337–355, 1993.

[SZ90] D. Saccá and C. Zaniolo. Stable models and non-determinism in logic programs with negation. In *Proc. of the Ninth Symp. on Principles of Database Systems*, 1990.

[SZ95] V. Subrahmanian and C. Zaniolo. Relating stable models and ai planning domains. In L. Sterling, editor, *Proc. ICLP-95*, pages 233–247. MIT Press, 1995.

[Tar55] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.

[Tar77] S. Tarnlund. Horn clause computability. *BIT*, 17:215–216, 1977.

[TB01] L. Tuan and C. Baral. Effect of knowledge representation on model based planning : experiments using logic programming encodings. In *Proc. of AAAI Spring symposium on Answer Set Programming*, 2001.

[Tru99] M. Truszczynski. Computing large and small stable models. In *ICLP*, pages 169–183, 1999.

[TS84] H. Tamaki and T. Sato. Unfold/Fold transformation of logic programs. In *Proc. of 2nd International conference on logic programming*, pages 127–138, 1984.

[TS88] R. Topor and L. Sonenberg. On domain independent databases. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 217–240. Morgan Kaufmann, San Mateo, CA., 1988.

[Tur93] H. Turner. A monotonicity theorem for extended logic programs. In D. S. Warren, editor, *Proc. of 10th International Conference on Logic Programming*, pages 567–585, 1993.

[Tur94] H. Turner. Signed logic programs. In *Proc. of the 1994 International Symposium on Logic Programming*, pages 61–75, 1994.

[Tur95] H. Turner. Representing actions in default logic: A situation calculus approach. In *Proceedings of the Symposium in honor of Michael Gelfond's 50th birthday (also in Common Sense 96)*, 1995.

[Tur96] H. Turner. Splitting a default theory. In *Proceedings of AAAI*, 1996.

[Tur97] H. Turner. Representing actions in logic programs and default theories. *Journal of Logic Programming*, 31(1-3):245–298, May 1997.

[Ull88a] J. Ullman. *Principles of Database and Knowledge-base Systems, volume I.* Computer Science Press, 1988.

[Ull88b] J. Ullman. *Principles of Database and Knowledge-base Systems, volume II.* Computer Science Press, 1988.

[Var82] M. Vardi. The complexity of relational query languages. In *ACM symposium on theory of computing (STOC)*, pages 137–146, 1982.

[vBK83] J. van Benthem and Doets K. Higher order logic. In D. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic*, volume 1, pages 275–329. Reidel Publishing company, 1983.

[VG88] A. Van Gelder. Negation as failure using tight derivations for general logic programs. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 149–176. Morgan Kaufmann, San Mateo, CA., 1988.

[vGRS88] A. van Gelder, K. Ross, and Schlpif. Unfounded Sets and Well-founded Semantics for General Logic Programs. In *Proc.* 7^{th} *Symposium on Principles of Database Systems*, pages 221–230, 1988. to appear in JACM.

[VGRS91] A. Van Gelder, K. Ross, and J. Schlipf. The well-founded semantics for general logic programs. *Journal of ACM*, 38(3):620–650, 1991.

[VV98] S. Vorobyov and A. Voronkov. Complexity of nonrecursive logic programs with complex values. In *PODS*, pages 244–253, 1998.

[Wag93] G. Wagner. Reasoning with inconsistency in extended deductive databases. In *Proc. of 2nd International workshop on Logic programming and non-monotonic reasoning*, 1993.

[Wat99] R. Watson. An application of action theory to the space shuttle. In G. Gupta, editor, *Proc. of Practical Aspects of Declarative Languages '99*, volume 1551, pages 290–304. Springer, 1999.

[WB93] C. Witteveen and G. Brewka. Skeptical reason maintainance and belief revision. *Artificial Intelligence*, 61:1–36, 1993.

[Wit91] C. Witteveen. Skeptical reason maintenance system is tractable. In J. A, R. Fikes, and E. Sandewall, editors, *Proc. of KR'91*, pages 570–581, 1991.

[WZ98] H. Wang and C. Zaniolo. User defined aggregates for logical data languages. In *DDLP*, pages 85–97, 1998.

[WZ00a] H. Wang and C. Zaniolo. Nonmonotonic Reasoning in LDL++. In J. Minker, editor, *This volume*. Kluwer, 2000.

[WZ00b] H. Wang and C. Zaniolo. User defined aggregates in object-relational systems. In *ICDE*, pages 135–144, 2000.

[WZ00c] H. Wang and C. Zaniolo. Using sql to build new aggregates and extenders for object-relational systems. In *VLDB*, 2000.

[Zan88] C. Zaniolo. Design and implementation of a logic based language for data intensive applications. In R. Kowalski and K. Bowen, editors, *Logic Programming: Proc. of the Fifth Int'l Conf. and Symp.*, pages 1666–1687, 1988.

[ZAO93] C. Zaniolo, N. Arni, and K. Ong. Negation and aggregates in recursive rules: the LDL++ approach. In *DOOD*, pages 204–221, 1993.

[Zha97] H. Zhang. SATO: An efficient propositional prover. In *Proc. of CADE'97*, pages 272–275, 1997.

Index