ABSTRACT

Title of dissertation: REASONING WITH CONFLICTING INFORMATION IN ARTIFICIAL INTELLIGENCE AND DATABASE THEORY

Shekhar Shantaram Pradhan, Doctor of Philosophy, 2001

Dissertation directed by: Professor Jack Minker
Department of Computer Science

We develop **C4** a logic for reasoning with information containing non-logical conflicts, where the information is encoded in the form of normal logic programs and the conflicts are represented using a construct called "contestation." We prove that the **C4** logic is inferentially conflict-free in the sense that the set of entailments of a normal logic program augmented with a set of contestations are guaranteed to be free of the conflicts specified by the set of contestations. We provide a sound and complete procedure for answering ground queries to a ground and finite normal logic program augmented with a set of ground contestations.

We show that **C4** provides a new semantics for normal logic programs that subsumes both the stable model semantics and the well-founded semantics for

normal logic programs. We use **C4** to provide a new account of integrity constraint satisfaction for databases that may be inconsistent with their integrity constraints.

We extend **C4** to **C5**, a five valued logic, which is used to provide a new semantics for extended logic programs. We show that **C5** can be used to provide an inferentially conflict-free logic for reasoning with information containing both logical and non-logical conflicts.

REASONING WITH CONFLICTING INFORMATION IN ARTIFICIAL INTELLIGENCE AND DATABASE THEORY

by

Shekhar Shantaram Pradhan

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2001

Advisory Committee:

Professor Jack Minker, Chairman/Advisor
Professor John Grant
Professor John Horty
Professor Donald Perlis
Professor Louiqa Raschid

DEDICATION

To the memory of my father Shantaram Keshav Pradhan

ACKNOWLEDGEMENTS

I am grateful to Professors John Grant, John Horty, Donald Perlis, and Louiqa Raschid for serving as members of my dissertation committee. They very generously agreed to serve as members of my committee inspite of their very busy schedules.

Professor Jack Minker, my dissertation director, has been a source of much inspiration to me. I have tried to model myself after him in many aspects of my professional life, especially research, but also in many aspects of my non-professional life. He has been my dissertation supervisor, a mentor and a friend. I am very grateful to him for the role he has played in my life.

This work has also benefited from interaction with past members of Professor Minker's research group, in particular Carolina Ruiz, Jose Alberto Fernandez, Jarek Gryz, and, most especially, Parke Godfrey who provided valuable feedback and constructive criticism of many of the ideas worked out in this work.

I am also grateful to Professor Robert Kowalski for discussing the **C4** semantics with me on several occasions when I was at the Imperial College of Science, Technology and Medicine during the Fall of 1996. I am grateful to Professor Luis Monez Pereira for inviting me to give a talk at the New University of Lisbon and discussing the semantics of extended logic programs presented here.

And lastly I am thankful to my son Roy for putting up with many weeks of my absence from his life over the last several years when I had to visit the University of Maryland to work with Professor Jack Minker.

TABLE OF CONTENTS

LIST OF TABLES

Chapter 1

Introduction

1.1 Motivation

Logic programming was invented by Kowalski ([Kow74] and Colmerauer ([CKRP73]. It was recognized at the very inception of the logic programming paradigm that it provided a powerful and natural system for representing information and drawing inferences from this information. But any body of information is liable to contain conflicting information, and certain inferential mechanisms are liable to function pathologically when reasoning with conflicting information. For instance, classical logic licenses the inference of any sentence from a logically inconsistent set of sentences. Clearly, in any practical context this is a highly undesirable feature in an inferential mechanism. Thus, there are good practical reasons for seeking inferential mechanisms that can behave reasonably when reasoning with conflicting information.

Thus, given a set of sentences

$$S = \{HeartDisease,\ \sim HeartDisease,\ Insured,\ HeartDisease \rightarrow Transplant\}$$

where $\sim p$ is the logical negation of p and $p \rightarrow q$ is classical material implication, classical logic licenses the inference of $Insured$ from S, but it equally licenses the inference of $\sim Insured$.

Paraconsistent logics ([Cos74], [Arr79], [Bel77b]) attempt to remedy this defect in classical logic by preventing the licensing of any arbitrary sentence from an inconsistent set of sentences. Thus, from the above set of sentences paraconsistent logics would permit the inference of $Insured$, but would not permit the inference of $\sim Insured$.

However, almost all paraconsistent logics suffer from a related problem. From the above set of sentences S, they all (with the exception of [Lin96]) permit the inference of the inconsistent set $\{HeartDisease, \ \sim HeartDisease\}$. This is an undesirable result because if $HeartDisease$ can be inferred then it is possible to infer $Transplant$ from S. But it would not be wise to conclude that a patient should get a heart transplant unless there was no doubt about whether the patient has heart disease.

Thus, it would be desirable to have a logic that goes beyond paraconsistent logics in that not only does it not license the inference of any arbitrary sentence from an inconsistent set of sentences, but *the set of its inferences must itself be consistent*. Let us call a logic *inferentially consistent* if the set of inferences permitted by such a logic is consistent. A goal of this thesis is to develop an inferentially consistent logic.

Logical inconsistency among two statements is just one type of conflict among statements. Most generally, two statements p and q are in conflict if the truth of p undermines the truth of q and vice versa. This may happen because p and q are each other's negation, i.e., they are logically inconsistent. But two statements can conflict even if there is no logical inconsistency among them. Thus, the statement that "John is male" and the statement that "John is female" are in conflict; but this is because the semantics of "male" and "female" preclude any one's being both male and female. Clearly, this is not a case of logical inconsistency. We may consider this an instance of a *semantic conflict*. There can also be *evidential conflicts* among statements. Thus, the statement that "John solved the $P = NP$ problem" is in conflict with the statement that "John has an I.Q. of 70." But this is neither a logical conflict nor a semantic conflict. Rather, the truth of the first statement provides evidence against the truth of the second statement and vice versa. It would be desirable for a logic which permits reasonable inferences from a set of sentences containing possibly different types of conflicts. Thus, the concept of an inferentially consistent logic can be generalized to the concept of an *inferentially conflict-free* logic. A logic can be said to be inferentially conflict-free with respect to specified types of conflict if the set of inferences it permits is free of any conflicts of the specified type.

Furthermore, with certain types of conflicts, such as evidential conflicts, different sets of statements may be in evidential conflicts to different degrees. This opens the possibility that one statement, p, in a conflict may conflict with another statement, q, to one degree, but q may conflict with p to a different degree. For example, arguably, the statement "John has solved the $P = NP$ problem" undermines the truth of "John has an I.Q. of 70" to a much higher degree than the degree to which the latter statement can undermine the truth of the former. As a limiting case, p may conflict with q to a certain degree but q may not conflict with p at all. Thus, we can allow non-mutuality in conflicts. Thus, the claim that "John has a high fever now" might undermine the claim that "John has disease D," but the diagnosis of a disease, which is typically a matter of conjecture, is generally not taken as undermining the observation of such a simple bodily property as its temperature. Thus, it would be useful to develop a framework for representing all these different types of conflicts and for reasoning with information containing these different types of conflicts.

The main goal of this thesis then is to devise a logic for reasoning with knowledge encoded in the form of normal logic programs augmented with constructs for specifying different types of conflicts such that the logic is inferentially conflict-free with respect to the specified types of conflicts.

1.2 Research Contributions

Here we summarize the main research contributions of this dissertation.

- We develop **C4**, a logic for reasoning with conflicting information (Chapter 4).
  - We present a construct we call **contestations** for specifying different types of conflicts, including non-mutual conflicts (Subsection 4.4.1).
  - We develop **C4**, a four-valued semantics for normal logic programs augmented with a set of contestations representing different types of conflicts. This semantics is based on generalizing to the four-valued context the concept of a well-supported model ([Fag91]) and by introducing an ordering relation among the well-supported models. The canonical models of $P + \mathcal{C}$, where P is a normal logic program and $\mathcal{C}$ is a set of contestations, are the maximal models in terms of this ordering among

the well-supported models of $P + \mathcal{C}$ (Section 4.4).

- We introduce two types of entailment relations: strong entailment and weak entailment. We prove that the inferences permitted in terms of these entailment relations are conflict-free with respect to the types of conflicts specified in terms of $\mathcal{C}$ (Subsection 4.4.3).
- We prove that for any normal logic program augmented with a certain type of contestations there is at least one canonical model of the augmented program (Section 4.4).
- We show how the four truth values of **C4** and the associated ordering between them can naturally be derived from the classical truth values T and F in the context of two players assigning the classical truth values to the same set of statements, where one player's assignment is allowed to dominate the other player's assignment without outright winning against the other player's assignment (Section 4.2).

- We investigate **C4** as a new semantics for normal logic programs (Chapter 5).
  - We prove that every definite logic program has a unique **C4** canonical model (Section 5.2).
  - We prove that **C4** regarded as a semantics of normal logic programs (without any contestations) has the property that every normal logic program has at least one **C4** canonical model (Section 5.2).
  - We prove that the **C4** semantics of normal logic programs subsumes the stable model semantics of normal logic programs ([GL88]) . More precisely, we show that for a normal logic program P with any two-valued stable models, a literal l is true in every stable model of P iff l

is weakly entailed by P under the **C4** semantics (Section 5.3).

  – We prove that the **C4** semantics of normal logic programs subsumes the well-founded semantics ([GRS88]) of normal logic programs. More precisely, we show that a literal l is true in the well-founded semantics of P iff l is strongly entailed by P under the **C4** semantics (Section 5.4).

  – We have devised a formalism to express hybrid conjunctive queries one part of which must be answered in terms of strong entailment and another part of which may be answered in terms of weak entailment (Section 5.5).

- We develop three proof procedures. These proof procedures use a bottom-up computation strategy and are based on making assumptions of literals and keeping track of which literal is inferred on the basis of which assumptions.

  – We provide a proof procedure for answering whether a ground query consisting of a conjunction or a disjunction of ground literals is *weakly* entailed by a finite and ground normal logic program without any contestations which has at least one stable model. If the program has no stable models the procedure detects that (Section 6.3). We prove that this proof procedure is sound and complete with respect to the **C4** semantics for normal logic programs (Section 6.4). We prove that the worst-case complexity of this procedure is $O(n^2 \times 2^n)$, where n is the cardinality of the Herbrand base of the program (Section 6.6). We modify this proof procedure to compute all the stable models of a program (Section 6.3).

  – We provide a proof procedure for answering whether a ground query consisting of a conjunction or a disjunction of ground literals is *strongly* entailed by a finite and ground normal logic program without any contestations (Section 7.3). We prove that this proof procedure is sound and complete with respect to the **C4** semantics for normal logic programs (Section 7.4). We prove that the worst-case complexity of this procedure is $O(n^3)$, where n is the cardinality of the Herbrand base of the program (Section 7.5). We prove that this proof procedure also computes the well-founded semantics of a normal logic program (Section 7.4).

  – We provide a proof procedure for answering a ground query to a finite and ground normal logic program P augmented with a set of ground contestations $\mathcal{C}$. The proof procedure can answer whether the query is strongly entailed by $P + \mathcal{C}$ or weakly entailed by $P + \mathcal{C}$(Section 8.3). We prove the soundness and completeness of this procedure with respect to the **C4** semantics for $P + \mathcal{C}$ (Section 8.4). We prove that the worst-case complexity of this procedure is $O(n^2 \times 2^n)$ for both weak and strong entailment, where n is the cardinality of the Herbrand base of the program (Section 8.5).

- We show the connection between integrity constraints and contestations. We use the **C4** semantics for normal logic programs augmented with a set of contestations to provide an account of propositional integrity constraint satisfaction for deductive databases that may be inconsistent with their own integrity constraints (Chapter 9).

  – We propose that integrity constraints be viewed as constraints on what can be proven from a database rather than constraints on the state of a database. We propose a new account of integrity constraint satisfaction in terms of this reinterpretation of the role of integrity constraints (Section 9.2).

  – We show how to translate a wide range of propositional integrity constraints into contestations (Section 9.3).

  – We show that the **C4** semantics for normal logic programs augmented with a set of contestations can be used as a semantics for deductive databases augmented with a set of propositional integrity constraints (Section 9.4).

- We develop an approach to reasoning with normal logic programs augmented with contestations and preferences (Chapter 10). We provide a language for expressing preferences among statements (Section 10.2). We extend **C4** to provide two semantics for a normal logic program, LP, augmented with a set of contestations, $\mathcal{C}$, and a set of preferences, $\mathcal{P}$. The first semantics is based on using the preferences of $\mathcal{P}$ to induce an ordering among the well-supported models of $LP + \mathcal{C}$. The second semantics is based on the idea of a well-supported model of $LP + \mathcal{C}$ satisfying the preferences of $\mathcal{P}$. Although these two semantics are based on different ways of factoring in the role of preferences, we prove that these two semantics are equivalent (Section 10.3).

- Finally, we extend **C4** to provide a semantics for extended logic programs, which contain both a default and a non-default negation (Chapter 11).

  – We develop a five-valued semantics **C5** which is an extension of **C4** (Section 11.3).

- We prove that every extended logic program has at least one (consistent) canonical model under **C5** (Section 11.3).
- We show how to capture part of the logical force of non-default negation in terms of contestations. If non-default negation is viewed as an approximation of classical negation, then logical conflict in a logic program can be represented in terms of the derivability of a literal and its non-default negation from the program. Thus, logical conflicts as well as non-logical conflicts can be represented in terms of contestations. This establishes that contestations provide a flexible framework for expressing and reasoning with a wide variety of conflicts among statements (Section 11.3).
- We prove that **C5** is inferentially conflict-free with respect to the approximation of logical conflicts in terms of non-default negation (Section 11.3).
- We prove that for any extended logic program P which has a consistent answer set, a literal l is strongly entailed by P under the answer set semantics ([GL90]) if and only if l is weakly entailed under **C5** (Section 11.4).
- We have shown how the five truth values of **C5** and orderings associated among these truth values can be derived from the truth values $\{F, U, T\}$ of Kleene ([Kle50]) and the truth and knowledge orderings among these truth values (Section 11.5).

1.3 Outline

In Chapter 2 we provide a brief description of some background work. In Chapter 3 we introduce some basic logic programming terminology and describe some well-known semantics of normal logic programs. In Chapter 4 we introduce contestations, a key concept of this dissertation. Contestations are a general way of expressing conflicts among sets of statements. We introduce **C4**, a semantics of normal logic programs augmented with contestations which express non-logical conflicts. In terms of this semantics we introduce two entailment relations: strong entailment and weak entailment. In Chapter 5 we discuss **C4** as a semantics of normal logic programs. We examine the relation between **C4** as a semantics of normal logic programs and the stable model semantics and the well-founded semantics of normal logic programs. In Chapter 6 we describe a proof procedure for determining whether a query is *weakly* entailed by a finite and ground normal logic program. We prove that this procedure is sound and complete with respect to the **C4** semantics for normal logic programs. In Chapter 6 we describe a proof procedure for determining whether a query is *strongly* entailed by a finite and ground normal logic program. We prove that this procedure is sound and complete with respect to the **C4** semantics for normal logic programs. We also show that this procedure computes the well-founded semantics of a normal logic program. In Chapter 8 we describe two proof procedures. The first proof procedure is to determine whether a query is *weakly* entailed by a finite and ground normal logic program augmented with a set of contestations. The second proof procedure is to determine whether a query is *strongly* entailed by a finite and ground normal logic program augmented with a set of contestations. We prove that both of these procedures are sound and complete with respect to the **C4** semantics for normal logic programs augmented with a set of contestations. In Chapter 9 we consider a semantics for deductive databases that may be inconsistent with a set of propositional integrity constraints. We introduce a new account of integrity constraint satisfaction that can apply even to databases that may be inconsistent with their own integrity constraints. We show how to express a wide variety of propositional integrity constraints in terms of the language of contestations. In Chapter 10 we introduce preferences among statements, which express the idea that the reasoner prefers that a normal logic program augmented with contestations should entail the preferred statement rather than the non-preferred statement. We show how to reason with normal logic programs augmented with a set of contestations and a set of preferences. In Chapter 11 we show how the **C4** semantics for normal logic programs can be extended to the **C5** semantics for extended logic programs containing both a default negation and a non-default negation. We show how we can use the concept of contestations and of non-default negation to capture the idea of logical conflicts. Thus, we provide a framework for reasoning with logical as well as non-logical conflicts among statements. In Chapter 12 we state the major conclusions of this work and state some lines of future work.

Chapter 2

Background

2.1 Introduction

It has long been recognized that the problem of reasoning with inconsistent information is of great practical importance in computer science ([Bel77b], [Bel77a], [Gra74], [Gra75], [Gra78]). The problem is that using classical logic all sentences can be inferred from an inconsistent set of sentences. One response is to prevent inconsistencies from arising or to remove inconsistencies. Truth maintenance ([Doy80]) and belief revision ([GR95]) fall under this type of effort, as does integrity constraint checking in databases. A different approach consists not in modifying the set of sentences from which the inferences are made, but instead in modifying what can be inferred from the set of sentences. Such logics are called paraconsistent logics ([Cos74], [Arr79]). A logic is called paraconsistent if it can form the basis for reasoning with an inconsistent set of sentences such that not all sentences can be derived from the set using this logic. The semantical foundation of most paraconsistent logics is based on departing from classical logic and instead adopting some version of multi-valued logic ([Gra75], [Bel77b], [FH85], [BS89], [KL92]. Hence in this chapter we provide some background on multi-valued logic.

2.2 Multi-valued Logics

In this section we describe work on three-valued and four valued logics.

The idea of multi-valued logic can be traced back to the work of the philosopher Aristotle in the 4^{th} century B.C. In his treatise *De Interpretatione* he considers the truth status of a future contingent sentence such as "There will be a sea-battle tomorrow." He notes that it seems not entirely correct to call this sentence true or false now. It seems to have some sort of a third truth status. Inspired by Aristotle's discussion of future contingents, the Polish logician Lukasiewicz proposed a logic based on a third truth value ([Luk20]) and began the modern era of multi-valued logics. However, it is Kleene's work on three-valued logic ([Kle50]) that has had a direct influence on computer science (see [Fit85]) for example).

Kleene proposed a third truth value **u**, which is supposed to mean *undefined* or *unknown*. He proposed a *strong* logic and a *weak* logic based on this third truth value. The main difference between the strong logic and the weak logic is that in the *weak* logic a truth-functionally compound sentence is always assigned **u** if one of its constituents is assigned **u**, regardless of the truth value assigned to the other constituents of the compound sentence.

Negation has the same truth table in both the strong and weak logics. The negation of T is F, the negation of F is T, and the negation of **u** is **u**.

The truth tables for conjunction, disjunction and implication in the strong Kleene logic and the weak Kleene logic are given below. Note that the only difference between the two truth tables is when one of the arguments to a truth function is **u**. In that case according to the weak logic the truth value returned by the function must be **u**.

$\wedge$	T	**u**	F
T	T	**u**	F
u	**u**	**u**	F
F	F	F	F

$\vee$	T	**u**	F
T	T	T	T
u	T	**u**	**u**
F	T	**u**	F

$\rightarrow$	T	**u**	F
T	T	**u**	F
u	T	**u**	**u**
F	T	T	T

Table 2.1: Conjunction, disjunction and implication in strong Kleene logic.

$\wedge$	T	**u**	F
T	T	**u**	F
u	**u**	**u**	**u**
F	F	**u**	F

$\vee$	T	**u**	F
T	T	**u**	T
u	**u**	**u**	**u**
F	T	**u**	F

$\rightarrow$	T	**u**	F
T	T	**u**	F
u	**u**	**u**	**u**
F	T	**u**	T

Table 2.2: Conjunction, disjunction and implication in weak Kleene logic.

A Kleene model of a set of sentences S is a three-valued interpretation which assigns a designated truth value to all sentences of S. The Kleene logics can be a paraconsistent logic only if both T and $\mathbf{u}$ are taken as designated truth values. To see this consider the set $S = \{p, \ \sim p, \ q\}$. No interpretation which assigns T or F to p can satisfy both p and $\sim p$ and thus cannot be a model of S. Hence, for S to have a model it must assign $\mathbf{u}$ to p and, thus, $\mathbf{u}$ must be regarded as a designated truth value. Thus, on the Kleene logics there are two models of S: a model which assigns $\mathbf{u}$ to both p and q, and a model which assigns $\mathbf{u}$ to p and T to q.

A set of sentences S entails a sentence s in the Kleene logics iff s is assigned a designated truth value in all the three-valued models of S. Hence in our example

S entails p, $\sim p$, and q. Thus, the Kleene logics are not inferentially consistent logics, although they are paraconsistent logics.

Belnap ([Bel77b]) generalizes Kleene's three valued logics to a four valued logic. Belnap's four truth values are F, $\perp$, T and $\top$. Intuitively, assigning F (T) means judging the sentence to be false (resp., true) in the classical logic sense of the term; assigning $\perp$ to a sentence means no information to judge the sentence as being true or false or that the truth value of the sentence is under-determined by the available information; assigning $\top$ to a sentence means that the truth value of the sentence is over-determined by the available information. $\top$ is also sometimes designated by $\{T, F\}$, which clearly indicates that this truth value is to be assigned to a sentence when there is information to assign it T and information to assign it F. This truth value plays a crucial role in providing a semantics for an inconsistent set of sentences.

There are two types of orderings associated among these four truth values. According to the truth ordering, $\leq_T$, $F \leq_T \top \leq_T T$ and $F \leq_T \perp \leq_T T$. According to the knowledge ordering, $\leq_K$, $\perp \leq_K T \leq_K \top$ and $\perp \leq_K F \leq_K \top$. Similar truth value orderings can be associated with Kleene's three truth values.

Although Belnap originally specified the logical negation of $\top$ to be $\perp$ and the logical negation of $\perp$ to be $\top$, other logicians ([Fit85], [BS89]) noted that it was more in keeping with the intuitive meaning of $\top$ and $\perp$ that the logical negation of $\top$ ($\perp$) should be $\top$ (resp., $\perp$).

This way of interpreting the logical negation of $\top$ also provides a straightforward semantics for logical inconsistent theories. Thus, consider the set $S = \{p,\ \sim p,\ q\}$. A Belnap-type four-valued model of this program assigns $\top$ to p and T to q. For this to be a model, both $\top$ and T must be the designated truth values of this logic. The only other model of S is one which assigns $\top$ to both p and q. Hence, it follows that on this semantics S entails p, $\sim p$ and q. But S does not entail $\sim q$. Thus, although Belnap's logic provides a paraconsistent logic, it does not provide an inferentially consistent logic.

Recently Lin ([Lin96]) has used Belnap's four truth values to provide an inferentially consistent semantics for logically inconsistent theories. He augments the language of propositional logic with two modal operators $\mathbf{B}$ and $\mathbf{B^c}$, which mean believes and believes consistently, respectively. Although he does not explicitly define the consistent entailments of a set of sentences S, it is easily seen that on his account S consistently entails a sentence p iff S entails $\mathbf{B^c}p$. The operator $\mathbf{B^c}$ is defined in terms of $\models_T$ and $\models_F$. Given, an assignment $\mathcal{I}$ of truth-values to the sentences of S, $\mathcal{I} \models_T p$ iff $\mathcal{I}$ assigns either T or $\{T, F\}$ to p and $\mathcal{I} \models_F p$ iff $\mathcal{I}$ assigns either F or $\{T, F\}$ to p. Relative to a set of sentences S, p is consistently believed iff in all the canonical models $\mathcal{I}$ of S, $\mathcal{I} \models_T p$ and $\mathcal{I} \not\models_F p$. Thus, S consistently entails p iff in every canonical model $\mathcal{I}$ of S, $\mathcal{I} \models_T p$ and $\mathcal{I} \not\models_F p$.

If $S = \{p, \ \sim p, \ q\}$, then S consistently entails q, but does not consistently entail either p or $\sim p$. Lin proves that his semantics is inferentially consistent.

Lin's semantics is based on some unusual features. A type of entailment (consistent entailment) is defined, but not on the basis of a corresponding notion of satisfaction. Furthermore, it is clear that from the point of view of consistent entailment, the only designated truth value is T. Yet any model of S must assign $\{T, F\}$ to p and, thus, assigning $\{T, F\}$ to p must be taken as satisfying that sentence in that model. So in Lin's semantics the well-understood connections between the concepts of designated truth values, satisfaction and entailment do not obtain.

Lin's semantics can be extended to handle some types of non-logical conflicts. Let us suppose that the fact that there is a non-logical conflict between p and q can be specified in terms of the statement $\sim (p \wedge q)$. In any set of sentences S containing both p and q, a canonical model of S must assign $\{T, F\}$ to both p and q, if the extension of Lin's semantics to non-logical conflicts is to be inferentially conflict-free with regard to the conflict expressed by $\sim (p \wedge q)$. However, it follows then that $\sim (p \wedge q)$ also evaluates to $\{T, F\}$ and, thus, it follows on Lin's semantics that the fact that p and q non-logically conflict cannot be known consistently. This seems a paradoxical result: the non-logical conflict between p and q is supposed to determine which truth values can be assigned to p and q, but the fact that there is such a conflict between p and q cannot be consistently known from S. This result holds regardless of whether the statement specifying this conflict, $\sim (p \wedge q)$, is part of S, or not part of S but used to determine which interpretations of S count as canonical models of S.

Furthermore, it is not clear how Lin's framework can be extended to reason with non-logical conflicts of varying degrees of strength.

For all these reasons, even though Lin's semantics does provide an inferentially consistent semantics, there is a need for a different approach that will provide a framework for reasoning with conflicting information.

Chapter 3

Logic Programming Preliminaries

3.1 Basic Definition

By an *atom* we mean a sentence consisting of an n-ary predicate symbol followed by n terms. A term may contain function symbols, variables, and constants. If a term contains no variables, it is called a *ground* term. An atom consisting entirely of ground terms is called a *ground atom*. By a *literal* we mean an atom or an atom preceded by the default negation operator, **not**.

By a *normal logic rule* we mean a sentence of the following form

$$a \leftarrow b_1, \ldots, b_m, \ \mathbf{not}\ c_1, \ldots, \ \mathbf{not}\ c_n$$

with $m, n \geq 0$. Here $a, b_1, \ldots, b_m, c_1, \ldots, c_n$ are all *atoms*. In the above rule $b_1, \ldots, b_m, \ \mathbf{not}\ c_1, \ldots, \ \mathbf{not}\ c_n$ is meant to be a conjunction. The negation symbol **not** is the default negation.

Note that $a, b_1, \ldots, b_m, c_1, \ldots, c_n$ need not be ground atoms. In this work we use the lower case letters to stand for both ground and non-ground literals. When we want them to stand for ground literals we make this explicit unless this is already clear form the context.

A non-ground rule, i.e., a rule with a non-ground atom in it, is assumed to be implicitly universally quantified.

A normal logic program is a set of normal logic rules.

Given a normal logic rule $R = a \leftarrow b_1, \ldots, b_m,\ \textbf{not}\ c_1, \ldots,\ \textbf{not}\ c_n$,

- $head(R) = a$,
- $body(R) = \{b_1, \ldots, b_m,\ \textbf{not}\ c_1, \ldots,\ \textbf{not}\ c_n\}$,
- $posbody(R) = \{b_1, \ldots, b_m\}$,
- $negbody(R) = \{\textbf{not}\ c_1, \ldots,\ \textbf{not}\ c_n\}$.
- $Atoms(R) = \{a, b_1, \ldots, b_m, c_1, \ldots, c_n\}$

Given a normal logic program P,

$$Atoms(P) = \bigcup_{R \in P} Atoms(R)$$

We sometimes interpret $body(R), posbody(R)$, and $negbody(R)$ to be a set of literals and at other times to be a conjunction of literals. The context makes clear which interpretation is intended. Given a set of atom $\{a_1, \ldots, a_n\}$ we understand $\textbf{not}\ \{a_1, \ldots, a_n\}$ to be a shorthand for $\{\textbf{not}\ a_1, \ldots, \textbf{not}\ a_n\}$.

We take the underlying language to be fixed by the language of the program P under discussion. By the *Herbrand universe* of P, we mean the set of all terms that can be formed using the language of P. By the *Herbrand base* of P (denoted by HB_P), we mean the set of all the ground atoms that can be formed using the predicates of P and the terms in the Herbrand universe of P.

By a *substitution* θ we mean a set of the form

$$\{v_1 = t_1, \ldots, v_k = t_k\}$$

where each v_i is a variable and each t_i is a term. If $\theta = \emptyset$ then it is called the empty substitution. By a *ground substitution* we mean a substitution in which all t_i are ground terms.

Given a rule R, by $R\theta$ we mean the result of applying the substitution θ to R. i.e., by replacing all the occurrences of each variable v_i in R by the corresponding term in θ. By *an instantiation* of R we mean a $R\theta$. We call an instantiation a *ground instantiation* if it is a ground rule.

Given a rule R and a program P, by $grd(R)$ we mean the set of all the ground instantiations of R with respect to the terms in the Herbrand universe of P. This need not be a finite set, but it will be a countable set. By $grd(P)$ we mean the set consisting of all the ground instantiations of all the rules in P with respect to the Herbrand universe of P.

A *minimal model* of a normal logic program, P, is a model of P such that no proper subset of that model is itself a model of P.

A *definite* logic program is a normal logic program that contains no negated atoms in its rules. It is well known that every *definite* logic program has a unique minimal model ([vEK76]).

A two-valued *Herbrand interpretation* of a logic program P is a subset of HB_P, the Herbrand base of P. Alternately, it can be understood as a mapping from HB_P to $\{T,\ F\}$, where T and F are the classical truth values.

3.2 Fixpoint Theory

Let $\leq$ be a binary relation on a set S which forms a partial order on the elements of S. For any subset X of S, $a \in S$ is an *upper bound* of X if $\forall x \in X, x \leq a$, and $a \in S$ is a *lower bound* of X if $\forall x \in X, x \geq a$. $a \in S$ is the *least upper bound (lub)* of a subset X of S if a is an upper bound of X and for all upper bounds a' of X, $a \leq a'$. The *greatest lower bound (glb)* can be defined in a similar way. S is a *complete lattice* if $lub(X)$ and $glb(X)$ exist for every subset X of S. Let S be a complete lattice. We say $X \subseteq S$ is *directed* if every finite subset of X has an upper bound in X. Given a complete lattice S, an operator $T : S \rightarrow S$ is said to be *continuos* if $T(lub(X)) = lub(T(X))$ for every directed subset X of L. For a lattice S, $x \in S$ is a *fixpoint* of T if $T(x) = x$. We say that x is the *least fixpoint (lfp)* of T if x is a fixpoint of T such that for all fixpoints x' of T, $x \leq x'$. The *greatest fixpoint* can be defined similarly.

For an operator T, the ordinal powers of T are defined as:

$$\begin{aligned} T \uparrow 0 &= glb(S) \\ T \uparrow \alpha &= T(T \uparrow (\alpha - 1)), \text{if } \alpha \text{ is a successor ordinal} \\ T \uparrow \alpha &= lub\{T \uparrow \beta | \beta < \alpha\} \text{if } \alpha \text{ is a limit ordinal} \end{aligned}$$

The following theorem states a well-known property of continuos operators.

Theorem 3.2.1 *([Llo87])*
For a continuos operator $T : S \rightarrow S$, $lfp(T) = T \uparrow \omega$, where ω is the first limit ordinal.

Van Emden and Kowalski ([vEK76]) defined an operator, T_P, of a program P that maps Herbrand interpretations of P into Herbrand interpretations of P thus:

$$T_P(I) = \{a \in HB_P \mid a \leftarrow body \in grd(P), I \models body\}$$

where I is a Herbrand interpretation of P.

The power set of the set of Herbrand interpretations of P (denoted by HB^P) forms a complete lattice under the $\subseteq$ ordering and thus the above defined fix-point theory can be utilized to study the semantics of logic programs. Furthermore, for *definite* logic programs, the T_P operator is continuos over the HB^P. So Theorem 3.2.1 above applies to such programs and so we are assured that the least fix point of T exists and can be computed in a countable number of steps. This provides a way of giving an operational semantics for definite logic programs as stated in the following theorem of Van Emden and Kowalski.

Theorem 3.2.2 *(vEK76)*
Let P be a definite logic program and let M_P be its unique minimal Herbrand model. Then

$$M_P = lfp(T_P) = T_P \uparrow \omega$$

The above theorem cannot be applied to normal logic programs since such programs need not have a least fix point or a unique minimal Herbrand model. We discuss the semantics of normal logic programs in the next subsection.

3.3 Semantics of Normal Logic Programs

Normal logic programs need not have a *unique* minimal Herbrand model. *The minimal model semantics* for normal logic program regards the set of all the minimal models of the program as the intended models of the program.

Example 3.3.1 *Let $P = \{a \leftarrow$ **not** $b\}$. The minimal models of P are $\{a\}$ and $\{b\}$.*

Minimal model semantics is widely regarded as unsatisfactory for normal logic programs. Thus, in the above example it is thought that $\{b\}$ should not be regarded as an intended model because **not** b should be regarded as true by default. Interpreting the negation **not** as default negation means that negative information should be regarded as true unless the program provides us a reason for thinking otherwise.

In the rest of this subsection we introduce three well-known semantics for normal logic programs.

3.3.1 Stable Model Semantics

In this section we introduce the stable model semantics for normal logic programs.

The stable model semantics is based on the Gelfond-Lifschitz transformation of a program ([GL88]).

Definition 3.3.1 Let P be a ground, normal logic program. Let M be a set of ground atoms. Then, the Gelfond-Lifschitz transformation of P is

$$P^M = \{a \leftarrow b_1, \ldots, b_k \mid a \leftarrow b_1, \ldots, b_k,\ \textbf{not}\ c_1, \ldots,\ \textbf{not}\ c_n \in P,\ c_1, \ldots, c_n \notin M\}$$

Note that P^M is always a definite program.

M is a *stable* model of a ground, normal logic program P if, and only if, M is the unique minimal model of the definite logic program P^M. The stable model semantics considers the stable models of a program as its intended models.

Note that this definition of stable models applies only to ground programs.

Example 3.3.2 *Let P be the ground program*

$$p \leftarrow a,\ \mathbf{not}\ q$$

$$p \leftarrow b,\ \mathbf{not}\ r$$

$$a \leftarrow\ \mathbf{not}\ b$$

$$b \leftarrow\ \mathbf{not}\ a$$

$$c \leftarrow$$

$$q \leftarrow r$$

$$r \leftarrow q$$

Then, $P^{\{p,a,c\}} = \{p \leftarrow a;\ p \leftarrow b;\ a \leftarrow;\ c \leftarrow;\ q \leftarrow r;\ r \leftarrow q\}$. The minimal model of $P^{\{p,a,c\}}$ is $\{p,a,c\}$. Hence $\{p,a,c\}$ is a stable model of P.

Similarly, $\{p,b,c\}$ is also a stable model of P. But, $\{p,a,b,c\}$ is not a stable model of P.

It is well known that not all normal logic programs have a two-valued stable model. Thus, $P = \{p \leftarrow\ \mathbf{not}\ p\}$ has no two-valued stable model. A related problem with the stable model semantics is the so-called "relevance problem" ([Dix95]). Let P be a program that has at least one stable model. Assume that $q \notin Atoms(P)$. In this sense q is not "relevant" to P. Then $P\ \cup\ \{q \leftarrow\ \mathbf{not}\ q\}$ has no stable models. That is, the addition of a rule irrelevant to P has robbed P of all its stable models.

3.3.2 Well Founded Semantics

According to the Well Founded Semantics, each normal logic program P has precisely one well founded model (henceforth referred to as $WFS(P)$). However, $WFS(P)$ can be a partial model in that it assigns neither true nor false to some atoms. We represent the well founded model of any program as a set of positive and negative literals.

The next four definitions are from [GRS91].

The well founded semantics is based on the idea of an unfounded set.

Definition 3.3.2 Let a program P, its associated Herbrand base HB_P, and a partial interpretation I be given. We say $A \subseteq HB_P$ is an *unfounded* set (of P) with respect to I if each atom $q \in A$ satisfies the following condition: For each instantiated clause C of P whose head is q, (at least) one of the following holds:

1. Some member (positive or negative) of $body(C)$ is false in I,

2. Some member of $posbody(C)$ occurs in A.

The union of all the unfounded sets with respect to I is itself an unfounded set, $U_P(I)$, and is called the greatest unfounded set of P with respect to I.

Definition 3.3.3 Let T_P be the operator defined in Subsection 3.2. Let U_P and W_P be transformations between sets of literals defined as follows:

- $U_P(I)$ is the greatest unfounded set of P with respect to I as defined above.

- $W_P(I) = T_P(I) \ \cup$ **not** $U_P(I)$

Definition 3.3.4 Let α range over all the countable ordinals. The sets I_α and I^∞ containing *literals* in HB_P are defined as:

- For a limit ordinal α,

$$I_\alpha = \bigcup_{\beta<\alpha} I_\beta$$

  Note that 0 is a limit ordinal and $I_0 = \emptyset$.

- For successor ordinal $\alpha + 1$,

$$I_{\alpha+1} = W_P(I_\alpha)$$

- Let

$$I^\infty = \bigcup_\alpha I_\alpha$$

Now we can define the well-founded semantics of P, $WFS(P)$, as

Definition 3.3.5 The well-founded semantics of P is the least fixed point of W_P, or the limit I^∞. Every positive literal denotes that its atom is true, every negative literal denotes that its atom is false, and missing atoms have no truth value assigned by the semantics.

Example 3.3.3 *As in Example 3.3.2, let P be*

$$\begin{aligned} p &\leftarrow a,\ \textbf{not}\ q \\ p &\leftarrow b,\ \textbf{not}\ r \\ a &\leftarrow\ \textbf{not}\ b \\ b &\leftarrow\ \textbf{not}\ a \\ c &\leftarrow \\ q &\leftarrow r \\ r &\leftarrow q \end{aligned}$$

Then, $I \uparrow 1 = W_P(\emptyset) = T_P(\emptyset) \cup\ \textbf{not}\ U_P((\emptyset) = \{c\} \cup\ \textbf{not}\ \{q, r\}$.

$I \uparrow 2 = I \uparrow 1$. *Hence,* $WFS(P) = \{c,\ \textbf{not}\ r,\ \textbf{not}\ q\}$

3.3.3 Well Supported Model Semantics

The idea of a well supported model is a refinement of the idea of a *supported* model ([ABW88]). A model M of a normal logic program P is considered supported if and only if for every $a \in M$ there exists a rule $R \in P$ such that $head(R) = a$ and $M \models body(R)$. The idea of a supported model is based on the intuition that an atom should be in a model only if there is adequate reason to include that atom in the model. However, the above stated definition of a supported model fails to fully capture this intuition. This can be seen by noting that $\{p\}$ is a supported model of the program $\{p \leftarrow p\}$, but intuitively this program fails to provide an adequate reason to include p in its model. The idea of well supported model was proposed to remedy this feature of supported models.

Definition 3.3.6 *([Fag91]) A model $M \subseteq HB_P$ of a normal logic program P is a two-valued well supported model if there exists a strict well-founded partial ordering $\ll$ on the atoms in HB_P such that for any $a \in M$, there exists a $R \in grd(P)$ such that $M \models body(R)$, where $head(R) = a$, and $b \ll a$ for every $b \in posbody(R)$.*

If we think of the body of a rule as providing evidence for attributing a certain truth-value to the head of the rule, then a well-supported model can be seen as assigning only that truth-value to any atom which can be justified in terms of the total evidence for it (with respect to that model), where the evidence must be independent of the truth-value assigned to that atom and must be finitely grounded in the facts. The well-founded ordering ensures that the truth-values assigned to an atom is not justified in terms of itself and the evidence is finitely grounded. Thus, for instance, no well-supported model of a program would assign true to p simply on the basis of the rule $p \leftarrow p$.

3.3.4 Relations Among the Semantics

The stable model semantics and the well supported model semantics turn out to be equivalent as stated in the following theorem.

Theorem 3.3.1 *([Fag91])*

M is a stable model of a normal logic program P if and only if it is a well supported model of P.

Well founded models are represented as a set of literals whereas stable models are represented as a set of atoms. However, a stable model can be represented as the set of *literals* that are true in that model. Representing stable models as a set of literals allows us to state the following two theorems which state the relation between stable model semantics and well founded semantics.

Theorem 3.3.2 *([GRS91])*

If a normal logic program has a well founded total model, then that model is the unique stable model of the program.

Theorem 3.3.3 *([GRS91])*

The well founded partial model of a normal logic program is a subset of every stable model of that program.

Chapter 4

Model theory for normal logic programs with contestations

4.1 Introduction

In this chapter we present a formal framework for reasoning with conflicting information. Most generally, two statements p and q are in conflict if the truth of p undermines the truth of q and vice versa. This may happen because p and q are each other's negation, i.e., they are logically inconsistent. But two statements can conflict even if there is no logical inconsistency among them. Thus, as discussed in Chapter 1, there can be semantic or evidential conflicts between statements.

Our focus in this chapter will be on non-logical conflicts. However, the framework we introduce here can be enriched to capture logical conflicts. This is done in Chapter 11. Furthermore, in this chapter we shall not assume that all conflicts are mutual. As discussed in Chapter 1, it can happen that accepting p as true can undermine the truth of q without accepting q as true undermining p's claim to be true. The framework we develop in this chapter will permit us to represent both mutual and non-mutual conflicts.

Even when the conflict between two statements p and q is mutual, the degree to which p undermines the truth of q may be different from the degree to which q undermines the truth of p. The framework we develop in this chapter also allows us to represent conflicts of different degrees of strength.

We develop this framework using **C4**, a new four valued logic. In the discussion section we compare this logic to other multi-truth valued approaches.

In Section 4.2 we introduce the four truth values $\mathcal{V} = \{F,\ CF,\ CT,\ T\}$ and provide a function for evaluating any closed sentence of the language of the program given an interpretation of the program based on $\mathcal{V}$. In this section we show how the four truth values of **C4** and the associated ordering between them can naturally be derived from the classical truth values T and F in the context of two players assigning the classical truth values to the same set of statements, where one player's assignment is allowed to dominate the other player's assignment without outright winning against the other player's assignment. In Section 4.3 we generalize the idea of a two-valued well-supported model of a program to a four-valued well-supported model, and we prove that every normal logic program has a four-valued well-supported model. In Section 4.4 we introduce contestations, which is a way of representing conflicts between statements. We define **C4**, a semantics for normal logic programs augmented with contestations of different degrees of strength. And in terms of this semantics we define two types of entailment: strong entailment and weak entailment. In Section 4.5 we compare the **C4** semantics with other multi truth-valued semantics for reasoning with conflicting information and compare it with some related work on argumentation.

4.2 Truth Values and Valuation

We propose a four-valued logic with the truth values, $\mathcal{V} = \{T,\ CT,\ CF,\ F\}$. This logic will be called **C4** and is defined precisely in Definition 4.4.8 below. It is called **C4** to suggest that it is a four valued logic of conflicts and contestations. Here, T and F have their usual meanings, but intuitively CF means "false only because successfully contested" and CT is the negation of CF.

We define a one-to-one mapping between members of $\mathcal{V}$ and members of $\mathcal{N} = \{1, 2/3, 1/3, 0\}$ thus: T maps to 1, CT maps to 2/3, CF maps to 1/3, and F maps to 0. We use the ordering among the members of $\mathcal{N}$ to induce the same ordering between members of $\mathcal{V}$. Thus, $F < CF < CT < T$.

We may regard the four truth values of **C4** and the associated ordering between them as arising naturally in the following fashion. Consider two players assigning truth values to the statements of a theory. Statements can be assigned the values T or F, and every statement of the underlying language of the theory must be assigned a truth value. Let us say player 1 has proposed the theory and player 1 gets to finally determine what truth value to assign to each statement taking into account the truth value he initially assigned to the statement and the truth value player 2 assigns to the statement. The two players may disagree on the assignment of truth values to some statements. On matters of disagreement, player 1 wants to let player 2 dominate, but not win outright. That is, if player 1 says p is true and player 2 says it is false, player 1 wants to relegate p to a status lower than true (i.e., let player 2 dominate), but does not want to assign it false (i.e., let player 2 win outright). He wants the truth value assigned to p to reflect the fact that he would regard p as true if it were not the fact that player 2 disagrees. As we shall see below, the truth values CF and CT are designed to play this sort of role.

For any given statement the four possible combinations of truth values are $\langle T, T\rangle$, $\langle T, F\rangle$, $\langle F, T\rangle$, $\langle F, F\rangle$, where the first component of each tuple is the value assigned by player 1 and the second component is the value assigned by player 2. Since $\langle T, T\rangle$ and $\langle F, F\rangle$ represent consensus, player 1 will finally assign T (F) to any statement which is assigned $\langle T, T\rangle$ (resp., $\langle F, F\rangle$). $\langle T, F\rangle$ is the value we call CF and $\langle F, T\rangle$ is the value we call CT. If player 2 is allowed to dominate but not win, then $F < \langle T, F\rangle < T$. Similarly, $F < \langle F, T\rangle < T$. We also hold that $\langle T, F\rangle < \langle F, T\rangle$ because player 1 allows player 2 to dominate. This produces the total ordering

$$F \ < \ \langle T, F\rangle \ < \ \langle F, T\rangle \ < \ T$$

which is the ordering we have adopted in this chapter where $\langle T, F\rangle$ is called CF and $\langle F, T\rangle$ is called CT.

Our terminology of domination can be defined precisely. To say that player 2 dominates player 1 is to say that the composite truth values are to be ordered first in terms of the second component of each composite (i.e., the truth value assigned by player 2) and if they are equal in terms of the second component then they are to be evaluated in terms of the first component. On the other hand, to say that player 1 dominates player 2 is to say that the composite truth values are to be ordered first in terms of the first component of each composite (i.e., the truth value assigned by player 1) and if they are equal in terms of the first component then they are to be evaluated in terms of the second component. To say that neither player dominates the other player is to say that one composite truth value t_1 is greater than or equal to another truth value t_2 if t_1 is greater than or equal to t_2 in both components and otherwise if neither is greater than or equal to the other in both components then the two truth values are incomparable. In this case $\langle T, F\rangle$ and $\langle F, T\rangle$ are incomparable.

T and CT are the *designated* values. That is, assigning T or CT to a sentence is taken as regarding that sentence as true. F is regarded as the *default* value in the sense that unless a sentence is assigned some other value it is taken as being assigned F. In the context of well supported models, it is assumed that the assignment of F to a sentence requires no justification.

An interpretation for a normal logic program P is a mapping from HB_P, the Herbrand base of P, to $\mathcal{V}$, or equivalently to $\mathcal{N}$. In the following we define a function $\mathcal{I}'$ which extends the mapping $\mathcal{I}$ to the (closed) sentences of the language.

Definition 4.2.1 *Let $\mathcal{I}$ be an interpretation. Then $\mathcal{I}'$ is a mapping from the sentences of the language to $\mathcal{N}$ recursively defined as:*

- *If S is a ground atom then $\mathcal{I}'(S) = \mathcal{I}(S)$.*
- *If S is a closed sentence then $\mathcal{I}'(\textbf{not } S) = 1 - \mathcal{I}'(S)$*
- *If S_1 and S_2 are (closed) sentences then*

$$\begin{aligned}
\mathcal{I}'(S_1 \wedge S_2) &= min(\mathcal{I}'(S_1), \mathcal{I}'(S_2)) \\
\mathcal{I}'(S_1 \vee S_2) &= max(\mathcal{I}'(S_1), \mathcal{I}'(S_2)) \\
\mathcal{I}'(S_1 \leftarrow S_2) &= \begin{cases} T & \text{if } \mathcal{I}'(S_1) \geq \mathcal{I}'(S_2) \\ CT & \text{if } \mathcal{I}'(S_1) = CF \text{ and } \mathcal{I}'(S_2) = CT \\ F & \text{otherwise} \end{cases}
\end{aligned}$$

- *For any sentence $p(X)$ with one unbound variable X,*

$$\mathcal{I}'(\forall X p(X)) = min\{\mathcal{I}'(p(t)) \mid t \in HU_P\}.$$

- *For any sentence $p(X)$ with one unbound variable X,*

$$\mathcal{I}'(\exists X p(X)) = max\{\mathcal{I}'(p(t)) \mid t \in HU_P\}.$$

Note that if we use only the classical truth values, T and F, then $\mathcal{I}'(S_1 \leftarrow S_2)$ reduces to the classical evaluation of implication which says that $S_1 \leftarrow S_2$ is evaluated as F if and only if S_2 is T and S_1 is F and otherwise it is evaluated as T. Thus, our evaluation function for implication is one way of generalizing the classical evaluation function to the multi-valued setting.

Note that on the interpretation of negation proposed above, **not not** $p = p$ and, for any interpretation $\mathcal{I}$ which assigns a truth value from $\mathcal{V}$ to p, $\mathcal{I}'($ **not** $p) =$ **not** $\mathcal{I}'(p)$. Furthermore, $p \leftarrow q$ is logically equivalent to **not** $q \leftarrow$ **not** p. However, $p \leftarrow q$ is not logically equivalent to $p \vee$ **not** q in terms of the above definition of implication. For instance, $CF \leftarrow CF = T$, but $CF \vee$ **not** $CF = CT$.

It should also be pointed out that although in many multi-valued logics implications can be assigned only the classical truth values (T and F), our evaluation function also permits CT to be assigned to implications. It will be seen below that this allows there to be a model for such pathological rules as $p \leftarrow$ **not** p. Thus, a model can be assigned to any normal logic program, which permits one to draw reasonable inferences from any normal logic program.

Given a set of literals $\{a_1, \ldots, a_n\}$, we use $\mathcal{I}'\{a_1, \ldots, a_n\}$ as shorthand for

$$\{\mathcal{I}'(a_1), \ldots, \mathcal{I}'(a_n)\}.$$

Given a rule $head \leftarrow body$, by $\mathcal{I}'$ $(body)$ we mean $min(\mathcal{I}'\{S \mid S \in body\})$.

When there is no possibility of confusion, in the following we use $\mathcal{I}$ to mean both the mapping from atoms to truth values (interpretation $\mathcal{I}$, properly speaking) as well as the evaluation function $\mathcal{I}'$ which is based on the interpretation $\mathcal{I}$ properly speaking.

4.3 Model Theoretic Preliminaries

For the purposes of the model theory of logic programs, we envisage the program P to be augmented as follows:

1. The atoms $true$, $CTrue$, $CFalse$ and $false$. It is assumed that $true$ evaluates to T, $CTrue$ evaluates to CT, $CFalse$ evaluates to CF and $false$ evaluates to F in any interpretation.

2. if P contains no constants, the dummy rule $p(\$a) \leftarrow p(\$a)$, where $\$a$ is a constant.

3. Any rule with an empty body is assumed to have $true$ as its body.

4. For each atom in HB_P, such that there is no rule in $grd(P)$ with that atom in the head, we add a rule with that atom as head and an atom denoting the default truth value as the body. Since we have chosen F as the default truth value, this atom will be $false$. Thus, if we have no information regarding an atom it will end up getting assigned the default truth value.

Augmenting the logic program in this manner allows us to state the model theory more elegantly than if we did not augment it thus. (More specifically, it helps with the definition of a well-supported model below.) It should be clear in the following that the augmentation does not change the actual semantics attributed to a logic program.

Definition 4.3.1 *We say that an interpretation $\mathcal{I}$ satisfies a rule R if $\mathcal{I}'(R) \in \{CT,\ T\}$.*

Our notion of satisfaction is one way of generalizing the classical notion which says that a rule is satisfied by an interpretation if it is true, or equivalently, if it has a designated truth value. In **C4** the designated truth values are CT and T.

4.3.1 Four-valued Well-Supported Models

Central to our semantics is the idea of a well-supported model ([Fag91]), which was defined in Chapter 3. In this subsection we show how to generalize it to four-valued well-supported models.

Recall that the idea behind the two-valued well supported model is that a well-supported model can be seen as assigning only that truth-value to any atom which can be justified in terms of the total evidence for it (with respect to that model), where the evidence must be independent of the truth-value assigned to that atom and must be finitely grounded in the facts. The well-founded ordering ensures that the truth-values assigned to an atom is not justified in terms of itself and the evidence is finitely grounded. Thus, for instance, no well-supported model of a program would assign true to p simply on the basis of the rule $p \leftarrow p$.

It is this idea which we wish to generalize to the four-valued context. The assignment of the *default truth value* is assumed to require no justification or evidence. The following definition assumes that F (or 0) is the default truth value. The assignment of any other truth value to an atom requires a non-circular justification that is finitely grounded in the facts. An atom which has no evidence for it must be assigned the default truth value. In this context having no evidence for an atom means

- having no information in support of that atom, or
- having only false information in support of that atom.

There is no information in support of an atom in case there are no rules in the original program with that atom in the head, i.e., the only rules in the augmented program with that atom in the head have the special atom denoting the default truth value in the body. To say that there is only false information in support of an atom (relative to an interpretation) is to say that the bodies of all rules with that atom in the head evaluate to F in that interpretation.

Definition 4.3.2 *A model $\mathcal{I}$ of P is a* well supported *model if there exists a strict well-founded partial ordering $\ll$ on the atoms in HB_P such that for any atom a in HB_P such that $F < \mathcal{I}(a)$, there exists a $R \in grd(LP)$ such that*

1. *$head(R) = a$, and*
2. *$\mathcal{I}(a) \leq \mathcal{I}'(body(R))$, and*
3. *$F < \mathcal{I}'(body(R))$, and*
4. *$b \ll a$ for every $b \in posbody(R)$.*

In the case of **C4**, condition 3 in the above definition can be subsumed by condition 2 and the requirement that $F < \mathcal{I}(a)$, but we state it explicitly to indicate that the attribution of a non-default truth value to an atom can be well supported in terms of a rule only if the body of the rule provides evidence for that atom.

We assume that in any such well-founded ordering the special atoms $true$, $CTrue$ and $CFalse$ are not ordered with respect to each other and are less than any other atom. Roughly speaking, a model is well supported if the truth value attributed to each atom is justified in terms of some rule of which that atom is the head. It is assumed that if the body of a rule supports the attribution of a certain truth value to an atom, then it supports the attribution of a lesser truth value to that atom. However, we regard it as epistemically unreasonable to attribute an atom a lesser truth value if a higher one can be well supported.

Example 4.3.1 *Let P be the ground program $\{a \leftarrow b;\ b \leftarrow a;\ c \leftarrow$ **not** $d;\ d \leftarrow$ **not** $d\}$. Then, in any well-supported model of P, a and b must be assigned F and d must be assigned CF. In addition, assigning CT to c would result in a well-supported model. But note that assigning CF to c would also result in a well-supported model.*

Note that when the only truth values used are T and F, the four-valued definition of a well-supported model reduces to the two-valued definition of a well-supported model.

Although not every normal program has a two-valued well-supported model in the sense of [Fag91], every normal program has a four-valued well-supported model.

Theorem 4.3.1 *Every normal logic program has at least one four-valued well-supported model.*

Proof: We show below how to construct a well-supported interpretation.

Let P be a normal logic program. Assign T (resp. F) to all atoms in HB_P which would be assigned T (resp. F) in $WFS(P)$, the well-founded semantics for P (see Chapter 3 above). Assign CF to all other atoms. We show that $\mathcal{I}$ constructed thus is a four-valued well-supported model of P.

$\mathcal{I}$ fails to be a model of P only if the head of a rule is assigned F or CF. But the head of a rule is assigned F only if $WFS(P)$ assigns it F and $WFS(P)$ is a model of such rules and so all such rules evaluate to T in $\mathcal{I}$. On the other hand, by construction, the head of a rule is assigned CF only if neither that atom nor its negation is in $WFS(P)$. This means that some member of the body of that rule does not evaluate to T. Since in our construction of the model we do not assign CT to any atom, this member of the body must evaluate to CF or F. In either case the rule evaluates to T. Hence, $\mathcal{I}$ must be a model of P.

It is clear that $\mathcal{I}$ is a *supported* model of P in the sense that for each $a \in HB_P$ there is a $R \in grd(P)$, such that $head(R) = a$ and $\mathcal{I}(a) \leq \mathcal{I}'(body(R))$. To show that it is well-supported we need to show that there is a well-founded ordering of the sort required in Definition 4.3.2.

An atom is assigned T in $WFS(P)$ if and only if that atom occurs in some iteration of the T_P operator. This ensures that a well-founded ordering can be constructed among those atoms that are assigned T by $\mathcal{I}$ in terms of when they first occur in an iteration of the T_P operator. Call it $\ll_T$. Note that no atoms are assigned CT by $\mathcal{I}$. We show that a well-founded ordering $\ll_{CF}$ can be constructed among the atoms that are assigned CF by $\mathcal{I}$. By way of contradiction assume otherwise. So there must be a $\subseteq$-minimal set S which consists of atoms that get assigned CF and whose members cannot be arranged in a well-founded ordering. But then all such atoms would belong to an unfounded set in the computation of $WFS(P)$. So they would all be assigned F in $WFS(P)$ and would thus be assigned F by $\mathcal{I}$. Thus a contradiction. And hence it must be possible to construct a well-founded ordering $\ll_{CF}$ on the atoms that get assigned CF.

The well-founded ordering $\ll$ that we require is any superset of $\ll_T \cup \ll_{CF}$

satisfying the constraint that no atom in $\ll_T$ is less than any atom in $\ll_{CF}$ in terms of $\ll$. There must be a well-founded ordering satisfying this constraint because any atom that gets assigned T (by $WFS(P)$ and thus by $\mathcal{I}$) must have a rule with that atom in the head such that no member of the *posbody* of that rule is undefined in $WFS(P)$ and thus assigned CF by $\mathcal{I}$.

This shows that $\mathcal{I}$ is a well-supported model of P. ∎

4.4 Semantics of Contestation

We write A contests b as $A \hookrightarrow b$, where A is a conjunction of ground literals and b is a ground atom.

Definition 4.4.1 *Let $A \hookrightarrow b$ be any contestations. Then*

- $Contestor(A \hookrightarrow b) = A$
- $Contested(A \hookrightarrow b) = b$

In this case we also call A the *contestor* of b and we say that b is contested by A. If b is contested by any A then we call b a *contested* atom. We say that a ground rule R is contested if $head(R)$ is a contested atom. We say that a non-ground rule R is contested if there is a ground substitution θ such that $head(R\theta)$ is a contested atom. When a logic program P is augmented with a set of contestations $\mathcal{C}$ we say P is *constrained* by $\mathcal{C}$ and write it as $P + \mathcal{C}$. As noted above, $A \hookrightarrow b$ can be understood as saying that the truth of A provides evidence against the truth of b. But this leaves open the question of whether this means that b is false. We can envisage contestations of different strengths. One type of contestation may be such that the truth of A guarantees the falsity of b, whereas a weaker type of contestation may be understood as saying that the truth of A merely ensures that b is not true. One can also imagine contestations where A must have the value T in order to block b from being true, whereas others in which A must be at least CT. Most generally $A \hookrightarrow b$ can be understood as saying that if A has a value of α then b can at most have a value of β (a *cap* of β).

Let *cap* be a mapping from $\mathcal{V}$ to $\mathcal{V}$. Clearly there can be many such mappings. In this context we call such mappings 'cap functions'. One such cap function cap_i can be associated with a contestation $A \hookrightarrow b$ in the sense that if A is assigned the truth value α then b must be assigned at most $cap_i(\alpha)$. This idea is captured in the following definition.

Definition 4.4.2 *Let A be a conjunction of literals in HB_P and b be any atom in HB_P. Let cap be a cap function associated with $A \hookrightarrow b$. Then $A \hookrightarrow b$ is satisfied by an interpretation $\mathcal{I}$ of P if, and only if, if $\mathcal{I}'(A)$ is α then $\mathcal{I}'(b)$ is at most $cap(\alpha)$.*

Note that $A \hookrightarrow b$ is trivially satisfied in $\mathcal{I}$ if A evaluates to an α in $\mathcal{I}'$ such that $cap(\alpha) = T$. In a situation where $cap(\alpha) = T$, $A \hookrightarrow b$ places no restriction on the truth value of b.

In the following we indicate the fact that a certain *cap* function cap_i is associated with a contestation $A \hookrightarrow b$ by writing the contestations as $A \hookrightarrow_i b$.

In the table below we define three cap functions.

If $A \hookrightarrow b$ is associated with cap_1, then it can be understood as saying that if A is assigned a value of *at least* CT in $\mathcal{I}$ then b must be assigned a value of *at most* CF if $A \hookrightarrow b$ is satisfied by $\mathcal{I}$. And if A is assigned any other value then $A \hookrightarrow b$ places no constraints on the value of b.

cap_1	cap_2	cap_3
$T \mapsto CF$	$T \mapsto CF$	$T \mapsto CF$
$CT \mapsto CF$	$CT \mapsto CF$	$CT \mapsto CT$
$CF \mapsto T$	$CF \mapsto CT$	$CF \mapsto T$
$F \mapsto T$	$F \mapsto T$	$F \mapsto T$

Table 4.1: The cap_1, cap_2 and cap_3 functions

If $A \hookrightarrow b$ is associated with cap_2, then it can be understood as saying that if A is assigned a value of *at least* CT in $\mathcal{I}$ then b must be assigned a value of *at most* CF and if A is assigned a value of CF then b must be assigned a value of *at most* CT if $A \hookrightarrow b$ is satisfied by $\mathcal{I}$. And if A is assigned any other value then $A \hookrightarrow b$ places no constraints on the value of b.

If $A \hookrightarrow b$ is associated with cap_3, then it can be understood as saying that if A is assigned a value of T in $\mathcal{I}$ then b must be assigned a value of *at most* CF and if A is assigned a value of CT then b must be assigned a value of *at most* CT if $A \hookrightarrow b$ is satisfied by $\mathcal{I}$. And if A is assigned any other value then $A \hookrightarrow b$ places no constraints on the value of b.

Recall that F is the default value. We regard all cap functions cap_j such that $cap_j(\alpha) < T$ when α is F as *ill defined* because assigning the default value to any contestor should place no restriction on the truth value of the atom contested by that contestor. All cap functions discussed in the following will be assumed to be not ill defined in the above sense.

Example 4.4.1 *Let P be the ground program $\{c \leftarrow$ **not** $a; a \leftarrow$ **not** $b; b \leftarrow$ **not** $a; d \leftarrow\}$. Let $\mathcal{C} = \{c \hookrightarrow_1 d\}$. That is, let the cap function associated with*

$\mathcal{C}$ be cap_1 as defined above. The table below displays some of the interpretations of P that satisfy $\mathcal{C}$.

	a	b	c	d
$\mathcal{I}_1$	T	F	F	T
$\mathcal{I}_2$	T	F	F	CT
$\mathcal{I}_3$	CT	CF	CF	T
$\mathcal{I}_4$	CT	CF	CF	CT
$\mathcal{I}_5$	CF	CT	CT	CF
$\mathcal{I}_6$	CF	CT	T	CF
$\mathcal{I}_7$	CF	CT	CF	T
$\mathcal{I}_8$	F	T	T	CF

Table 4.2: An example of interpretations that satisfy a contestation

In the above example $\mathcal{I}_1$, $\mathcal{I}_2$, $\mathcal{I}_3$, $\mathcal{I}_4$, and $\mathcal{I}_7$ trivially satisfy $c \hookrightarrow_1 d$ because in these interpretations c is assigned a truth value lower than CT.

In the above example $\mathcal{I}_5$, $\mathcal{I}_6$, and $\mathcal{I}_8$ are not models of P. However, as we shall see below, they turn out to be models of $P + \mathcal{C}$. $\mathcal{I}_2$ and $\mathcal{I}_4$ are not epistemically reasonable models because they assign d a value lower than would be supported by the evidence for d (i.e., T). Similarly, $\mathcal{I}_7$ is not an epistemically reasonable model because c is attributed a lower value than would be supported by the evidence. On the other hand, $\mathcal{I}_6$ is also not an epistemically reasonable model because the evidence for c can not support the assignment of T to c. Both $\mathcal{I}_5$ and $\mathcal{I}_7$ satisfy $c \hookrightarrow_1 d$, but it seems clear that the proper way to satisfy $c \hookrightarrow_1 d$ is by assigning CF to d because c provides evidence against d whereas there is no evidence against c.

Intuitively a model of $P + \mathcal{C}$ is an epistemically reasonable model if it assigns each atom the maximum truth value that it can be assigned taking into account the evidence *for* and *against* that atom relative to the assignment of truth value to the other atoms. In the following we capture the intuitive idea of epistemically reasonable models in terms of these three steps:

- Evidence for and evidence against must be *combined* together
- Attribution of truth values to an atom must be *justified* in terms of evidence for and against it
- An atom must be attributed the *maximal* justified truth value.

4.4.1 Combining Evidence For and Against

Since we understand the truth-value of the body of a rule (with respect to an interpretation) as providing evidence for attributing a certain truth-value to its head and we interpret a contestation as saying that the contestor provides evidence against the truth of the contested atom, we can combine these two ideas when the contested atom is also the head of a rule. Thus, we can say that the evidence provided by the body of a rule for the truth of the head of a rule must be constrained or capped by the evidence presented against the head of the rule by the truth of its contestors. This idea is captured below.

First we define a function cap' which takes an atom, a contestation and an interpretation as arguments and returns a special atom as a value.

Definition 4.4.3 *Let b be an atom, not necessarily ground. Let $\mathcal{C}_j$ be a contestation with an associated cap function cap_i. Then, $cap'_i(b, \mathcal{C}_j, \mathcal{I})$ returns the special* *atom which always evaluates to* $cap_i(\mathcal{I}(Contestor(C_j)))$ *if* $Contested(\mathcal{C}_j) = b\theta$*, for some substitution* θ *which can be the empty substitution, otherwise* $cap_i'(b, \mathcal{C}_j, \mathcal{I})$ *returns the special atom true.*

Example 4.4.2 *Let* $\mathcal{C}_j = a \hookrightarrow_i b$*. Let* cap_i *be* cap_2 *as defined above. If* $\mathcal{I}$ *assigns* T *or* CT *to* a *then* $cap_i'(b, \mathcal{C}_j, \mathcal{I})$ *returns the special atom* $CFalse$ *which always evaluates to* CF*. However, if* $\mathcal{I}$ *assigns* CF *to* a *then* $cap_i'(b, \mathcal{C}_j, \mathcal{I})$ *returns the special atom* $CTrue$ *which always evaluates to* CT*.*

We systematically abuse notation by extending the above definition of cap_i' to have a set of contestations as an argument instead of a single contestation.

Definition 4.4.4 *Let* b *be an atom and* $\mathcal{C}$ *be a set of contestations such that each member of* $\mathcal{C}$ *has the same associated cap function* cap_i*. Let* $\mathcal{I}$ *be an interpretation. Then,*

$$cap_i'(b, \mathcal{C}, \mathcal{I}) \;=\; min\{cap_i'(b, \mathcal{C}_j, \mathcal{I}) \mid \mathcal{C}_j \in \mathcal{C}\}$$

When a logic program P is augmented with a set of contestations $\mathcal{C}$, we say P is *constrained* by $\mathcal{C}$ and write it as $P + \mathcal{C}$. Each rule R in P is considered as constrained by $\mathcal{C}$. We write a rule R constrained by $\mathcal{C}$ as $head(R) \leftarrow_{\mathcal{C}} body(R)$. We call such rules 'constrained rules'. The function $\mathcal{I}'$ (Definition 4.2.1) needs to be modified to evaluate constrained rules. We define this function below.

Definition 4.4.5 *Let* P *be a normal logic program which is constrained by* $\mathcal{C}$*, a set of contestations. Let* cap_i *be the cap function associated with* $\mathcal{C}$*. Let* $\mathcal{I}$ *be an interpretation of* P*. Then,* $\mathcal{I}''$ *is a mapping from the rules of the language to* $\mathcal{N}$ *recursively defined as:*

- *If* E *is a literal or a conjunction or a disjunction of literals then* $\mathcal{I}''(E) = \mathcal{I}'(E)$*.*

- *If E is a rule (head $\leftarrow_{\mathcal{C}}$ body) then*

$$\mathcal{I}''(E) = \mathcal{I}'(head \ \leftarrow \ body, \ cap'_i(head, \ \mathcal{C}, \mathcal{I}))$$

In the above definition we assume that a unit rule, that is, a rule of the form $p \leftarrow$ is implicitly a rule of the form $p \leftarrow true$. Thus we distinguish between the atom p and the rule $p \leftarrow$. The atom p would be evaluated in terms of the first clause of the above definition, whereas the rule $p \leftarrow$ would be evaluated in terms of the second clause.

If a contestor of the head of a rule evaluates to at least α, then it provides evidence against the head being any greater than $cap(\alpha)$. Thus, in effect, the contestor puts an upper limit or a cap on how much evidence there can be for the head. This idea is captured by the second part of the above definition by inserting the special atom which always evaluates to $cap_i(\alpha)$ in the body of the rule. This brings out exactly how the rule is constrained by the contestations.

Note that $\mathcal{I}''$ reduces to $\mathcal{I}'$ when $\mathcal{C} = \emptyset$.

In the following we assume that the sentences of any program are evaluated according to $\mathcal{I}''$.

In the above definition we have assumed that the contestations are homogeneous in the sense there is only some one cap function associated with the entire class of contestations $\mathcal{C}$. But it is possible that $\mathcal{C}$ may contain many different types of contestations where each type has its own associated cap function. This allows our formalism to represent heterogeneous contestations. Thus, suppose we can exhaustively partition $\mathcal{C}$ into $\mathcal{C}_1, \ldots, \mathcal{C}_n$ in terms of their different associated cap functions. Then we can define n cap'_i functions where each $cap'_i(b, \mathcal{C}, \mathcal{I})$ returns the special atom which always evaluates to $cap_i(\alpha)$ if there exists a contestation

$A \hookrightarrow_i b \in \mathcal{C}_i$ such that $\mathcal{I}'(A) = \alpha$, where $\mathcal{C}_i$ is the subset of $\mathcal{C}$ with which cap_i is associated; otherwise $cap'_i(b, \mathcal{C}, \mathcal{I})$ returns the special atom $true$.

In case $\mathcal{C}$ contains heterogeneous contestations, $\mathcal{I}''$ can be understood as

$$\mathcal{I}''(head \;\leftarrow_{\mathcal{C}}\; body) = \mathcal{I}''(head \;\leftarrow\; body, cap'_1(head,\; \mathcal{C}, \mathcal{I}), \ldots, cap'_n(head,\; \mathcal{C}, \mathcal{I}))$$

4.4.2 Justified Attribution of Truth-values

In this subsection we carry out the second step in defining epistemically reasonable models.

The attribution of a truth value to an atom in a model of $P + \mathcal{C}$ is justified if that attribution is well-supported in terms of the rules of $P + \mathcal{C}$, where these rules are now understood as constrained rules. Thus we must extend the previously defined concept of well-supported models of P to $P + \mathcal{C}$. We do this by taking into account the evidence contrary to each atom which is attributed a value greater than the default truth value in determining whether the attribution of this value is well supported.

Definition 4.4.6 *Let P be a normal logic program. Let $\mathcal{C}$ be a set of contestations which can be partitioned into the sets $\{\mathcal{C}_1, \ldots, \mathcal{C}_n\}$ with each distinct $\mathcal{C}_i$ associated with a distinct cap function cap_i. Then model $\mathcal{I}$ of $P + \mathcal{C}$ is a* well supported *model if there exists a strict well-founded partial ordering $\ll$ on the atoms in HB_P such that for any atom a in HB_P such that $F < \mathcal{I}(a)$, there exists an $R \in grd(P) + \mathcal{C}$ such that*

1. *$head(R) = a$, and*

2. *$\mathcal{I}(a) \leq \; \mathcal{I}(body(R) \;\wedge\; cap'_1(a, \mathcal{C}_1, \mathcal{I}) \;\wedge \cdots \wedge\; cap'_n(a, \mathcal{C}_n, \mathcal{I}))$, and*

3. $b \ll a$ *for every* $b \in posbody(R)$.

Note that in case $\mathcal{C} = \emptyset$ the well-supported models of $P + \mathcal{C}$ become the well-supported models of P as defined in Definition 4.3.2. As in that definition, if the attribution of a truth-value to an atom is well-supported in a model then the attribution of a lower truth-value to that atom is also well-supported. However, we shall see in the next section that a well-supported model will not be regarded as an epistemically reasonable model if it does not attribute an atom the highest truth-value that would be well-supported in that model.

Theorem 4.3.1 above says that every normal logic program has at least one well-supported model. However, whether $P + \mathcal{C}$ has a well-supported model depends on the cap functions associated with $\mathcal{C}$. It can easily be shown that if cap_1 is the cap function associated with $\mathcal{C}$ then there can be no guarantee that $P + \mathcal{C}$ has a well-supported model. The example below illustrates this point. However Theorem 4.4.1 below says that $P + \mathcal{C}$ has a well-supported model if cap_2 is the cap function associated with $\mathcal{C}$.

Example 4.4.3 *Let* P *be the ground program* $\{p \leftarrow q;\ q \leftarrow\}$ *and let* $\mathcal{C} = \{p \hookrightarrow_1 q\}$. *That is, let* cap_1 *be the associated cap function. If an interpretation* $\mathcal{I}$ *were to assign* T *or* CT *to* p *then* $\mathcal{I}$ *would have to assign* CF *to* q *to satisfy* $\mathcal{C}$*, in which case the assignment of* T *or* CT *to* p *would not be well-supported. On the other hand if* CF *were assigned to* p *then* CT *would have to be assigned to* q *in order for* $\mathcal{I}$ *to be a model of* $p \leftarrow q$. *But in that case it would not be a model of the constrained rule* $q \leftarrow cap_1'(q, \mathcal{C}, \mathcal{I})$ *since* $cap_1'(q, \mathcal{C}, \mathcal{I})$ *would evaluate to* T. *Similarly* $\mathcal{I}$ *cannot assign* F *to* p *without failing to model one of the two clauses of* $P + \mathcal{C}$.

The following theorem says that every $P + \mathcal{C}$ has at least one well-supported model if cap_2 is the cap function associated with $\mathcal{C}$. Recall that the only difference between cap_1 and cap_2 is that $cap_1(CF) = T$ and $cap_2(CF) = CT$. Thus, in the case of the program and contestations in the above example, an interpretation which assigns CF to p and CT to q would be a well-supported model if cap_2 is associated with $p \hookrightarrow q$. This is because $cap_2'(q, \mathcal{C}, \mathcal{I})$ would evaluate to CT and, in **C4**, $CF \leftarrow CT$ evaluates to CT.

Theorem 4.4.1 *Let P be any normal logic program and let $\mathcal{C}$ be any set of contestations. Let cap_2 be the cap function associated with $\mathcal{C}$. Then $P+\mathcal{C}$ has at least one well-supported model.*

Proof: We show below how to construct a well-supported interpretation $\mathcal{I}$ of $P+\mathcal{C}$ assuming that cap_2 is the cap function associated with $\mathcal{C}$.

Let $\mathcal{J}$ be an interpretation of P such that $\mathcal{J}$ assigns F to any atom a such that **not** $a \in WFS(P)$, the well-founded semantics for P. Let $\mathcal{J}$ assign T to all atoms a such that $a \in WFS(P)$. Let $\mathcal{J}$ assign CF to all other atoms in HB_P. Modify this interpretation $\mathcal{J}$ so that all atoms a are assigned CF such that $B \hookrightarrow_2 a \ \in \mathcal{C}$ and $\mathcal{J}(B) = T$ and $\mathcal{J}(a) = T$. Call this interpretation $\mathcal{J}'$. We propagate this change in the status of a to all atoms whose presence in $WFS(P)$ depended on a being in $WFS(P)$ by deleting all rules R from $grd(P)$ such that $\mathcal{J}'(head(R)) = CF$. Let P' be the modified program. We modify $\mathcal{J}'$ so that all atoms a such that $\mathcal{J}'(a) = T$ but $a \notin WFS(P')$ are assigned CF. Similarly all atoms a such that $\mathcal{J}'(a) = F$ but **not** $a \notin WFS(P')$ are assigned CF. Call this interpretation $\mathcal{I}$. We show below that $\mathcal{I}$ is a well-supported model of $P + \mathcal{C}$.

Clearly, $\mathcal{I}$ fails to be a model of $P+\mathcal{C}$ only if $\mathcal{I}$ fails to be a model of $grd(P)+\mathcal{C}$. $\mathcal{I}$ fails to be a model of $grd(P) + \mathcal{C}$ only if the head of some rule is assigned F and the body evaluates to a value greater than F or the head is assigned CF and the body evaluates to T. But the head of a rule is assigned F by $\mathcal{I}$ only if the negation of the head is in $WFS(P')$, in which case some literal in the body of each rule of P' is false in $WFS(P')$ and thus evaluates to F in $\mathcal{I}$. So $\mathcal{I}$ is a model of any rule whose head is assigned F by $\mathcal{I}$. The head of a rule a is assigned CF only if neither a nor its negation is in $WFS(P)$ or a is in $WFS(P)$ but some contestor of a evaluates to T in $\mathcal{J}$ or a is in $WFS(P)$ but a is not in $WFS(P')$. In the first case no rule with a as head can have its body evaluate to T in $\mathcal{J}$ and thus not in $\mathcal{I}$. In the second case $cap_2'(a, \mathcal{C}, \mathcal{J}')$ evaluates to CF or CT and so the body of any such constrained rule cannot evaluate to T in $\mathcal{J}'$ and thus not in $\mathcal{I}$. In the last case the body of any such rule contains some literal that does not belong to $WFS(P')$ and so evaluates to CF or CT in $\mathcal{I}$. Thus $\mathcal{I}$ is a model of $grd(P) + \mathcal{C}$. Hence, $\mathcal{I}$ is a model of $P + \mathcal{C}$.

It is clear that for each $a \in HB_P$ there is a $a \in grd(P)$, such that $head(R) = a$ and $\mathcal{I}(a) \leq \mathcal{I}'(body(R) \wedge cap_2'(a, \mathcal{C}, \mathcal{I}))$. To show that $\mathcal{I}$ is well-supported we need to additionally show there is a well-founded ordering on HB_P of the sort required by the definition of a well-supported model. Such an ordering can be constructed exactly as in the proof of Theorem 4.3.1. ■

4.4.3 Maximally Justified

In the previous subsection we showed how to capture the idea that the attribution of a truth-value to any atom must be justified in epistemically reasonable models. In this section we show how to capture the idea that an epistemically reasonable model must attribute an atom the highest truth-value that would be well-supported in that model.

Recall that in any model $\mathcal{I}$ a rule is assigned CT only if it attributes CF to the head and CT to the body of the rule. In some cases it is impossible to have a well-supported model of the program which can assign CT to the head of the rule and at the same time have the body evaluate to CT (e.g. $p \leftarrow$ **not** p). But in other cases this is possible. So we can have two models of a program where one model assigns CF to the head of a rule and CT to the body and the other which assigns CT to head of that rule and CT to the body. In the first model the rule would evaluate to CT whereas in the second the rule would evaluate to T. In such cases the higher the value assigned to the head the higher the value assigned to the rule. Thus we can rank the well-supported models of $P + \mathcal{C}$ in terms of the truth-value they assign to the rules of $P + \mathcal{C}$. From the above discussion we see that those well-supported models would be ranked higher which assign a higher justified truth-value to the atoms. In order to capture this idea we introduce a *clausal* ordering between interpretations.

Let $\mathcal{I}_1$ and $\mathcal{I}_2$ be two interpretations of $P + \mathcal{C}$. Then, $\mathcal{I}_1 \leq_{P+\mathcal{C}} \mathcal{I}_2$ if, and only if, $\mathcal{I}_1''(R) \leq \mathcal{I}_2''(R)$ for every rule R in $P + \mathcal{C}$.

$\mathcal{I}_1 <_{P+\mathcal{C}} \mathcal{I}_2$ if, and only if, $\mathcal{I}_1 \leq_{P+\mathcal{C}} \mathcal{I}_2$ and it is not the case that $\mathcal{I}_2 \leq_{P+\mathcal{C}} \mathcal{I}_1$.

Given a set of interpretations ν, we say that an interpretation $\mathcal{I}_i$ is *maximal with respect to* $P+\mathcal{C}$ in ν if there is no interpretation $\mathcal{I}_j \in \nu$ such that $\mathcal{I}_i <_{P+\mathcal{C}} \mathcal{I}_j$.

When $\mathcal{C} = \emptyset$ the clausal ordering produces an ordering among the models of P. It is customary in this context to introduce a *pointwise* ordering among interpretations, either in terms of a truth ordering or in terms of a knowledge ordering among atoms. Thus, we could introduce: $\mathcal{I}_1 \leq^p \mathcal{I}_2$ iff for all atoms a in HB_P, $\mathcal{I}_1(a) \leq_P \mathcal{I}_2(a)$. But the two orderings do not produce the same result.

Let P be the ground program $\{p \leftarrow q, \textbf{not}\ r, \textbf{not}\ s;\ r \leftarrow \textbf{not}\ s;\ s \leftarrow \textbf{not}\ r, \textbf{not}\ p; q \leftarrow\}$. Consider the following two models of P.

	p	q	r	s
$\mathcal{I}_1$	CF	T	CF	CT
$\mathcal{I}_2$	CT	T	CF	CF

Table 4.3: Pointwise vs. clausal ordering among models.

In this case the two models are incomparable in terms of the pointwise ordering, but $\mathcal{I}_1$ is strictly greater than $\mathcal{I}_2$ in terms of the clausal ordering.

It seems to us that we should use the clausal ordering instead of the pointwise ordering because a model is supposed to be a model of the sentences of a theory; it is not required to be a model of the atoms of the theory. Hence, models that maximize the degree of truth of rules should be preferred.

In terms of the idea of maximal models in the clausal ordering we can define the *canonical* models of $P + \mathcal{C}$.

Definition 4.4.7 *The canonical models of $P+\mathcal{C}$ are the clausally maximal models among the well-supported models of $P + \mathcal{C}$.*

The idea of epistemically reasonable models is fully captured in terms of the above defined idea of canonical models.

Example 4.4.4 *As in Example 4.4.1 above, let*

$$P = \{c \leftarrow \textbf{not}\ a; a \leftarrow \textbf{not}\ b; b \leftarrow \textbf{not}\ a; d \leftarrow\}$$

and let

$$\mathcal{C} = \{c \hookrightarrow_1 d\}$$

Then the canonical models of $P + \mathcal{C}$ are

	a	b	c	d
$\mathcal{I}_1$	T	F	F	T
$\mathcal{I}_2$	CT	CF	CF	T
$\mathcal{I}_3$	CF	CT	CT	CF
$\mathcal{I}_4$	F	T	T	CF

Table 4.4: An example of the canonical models $P + \mathcal{C}$

In each of these models all the constrained rules evaluate to T and thus these models must be maximal in the clausal ordering. Note that the rule $d \leftarrow_{\mathcal{C}}$ evaluates to T in $\mathcal{I}_4$ even though the atom d is assigned CF because the contestor of d evaluates to T and thus $d \leftarrow_{\mathcal{C}}$ is evaluated as $d \leftarrow CFalse$. Thus, we see that the idea of epistemically reasonable models is captured in the canonical model theory.

Now we are in a position to formally define the semantics **C4**.

Definition 4.4.8 *By* **C4** *we mean the four truth-values with the associated ordering among them, the evaluation functions $\mathcal{I}'$ and $\mathcal{I}''$, the relation of satisfaction between interpretations and sentences, the selection function among the models of a program implicit in the definition of a well-supported model, the clausal ordering among interpretations and the selection function among models implicit in Definition 4.4.7 of canonical models.*

The following theorem follows directly from Theorem 4.4.1 above.

Theorem 4.4.2 *Let P be any normal logic program and let $\mathcal{C}$ be any set of contestations. Then $P + \mathcal{C}$ has at least one canonical model.*

We have defined the concept of a normal logic program P satisfying a set of contestations $\mathcal{C}$ and we have defined the canonical models of $P+\mathcal{C}$. The following theorem ties together these two concepts.

Theorem 4.4.3 *Every canonical model of $P+\mathcal{C}$ satisfies $\mathcal{C}$.*

Proof: Let $\mathcal{I}$ be a canonical model of $P+\mathcal{C}$. Assume by way of contradiction that there is a $\mathcal{C}_j \in \mathcal{C}$ such that $\mathcal{C}_j$ is not satisfied by $\mathcal{I}$. Let $Contestor(\mathcal{C}_j)$ evaluate to α in $\mathcal{I}$. Let $Contested(\mathcal{C}_j)$ evaluate to β in $\mathcal{I}$. Let the *cap* function associated with $\mathcal{C}_j$ be cap_i. Thus, if $\mathcal{I}$ violates $\mathcal{C}_j$ then $\beta > cap_i(\alpha)$.

Any rule $R \in grd(P) + \mathcal{C}$ such that $head(R) = Contested(\mathcal{C}_j)$, is evaluated by $\mathcal{I}''$ as if R has $cap'_i(head(R), \mathcal{C}_j, \mathcal{I}'')$ in its body. $cap'_i(head(R), \mathcal{C}_j, \mathcal{I}'')$ returns the special atom which evaluates to $cap_i(\alpha)$. Thus, the body of any R such that $head(R) = Contested(\mathcal{C}_j)$ can evaluate to at most $cap_i(\alpha)$ in $\mathcal{I}$. Hence since we assumed that $Contested(\mathcal{C}_j)$ evaluates to β and $\beta > cap_i(\alpha)$, it follows that $head(R)$ evaluates to a truth-value greater than the truth-value of $body(R)$ for any such R. Thus, $\mathcal{I}$ cannot be a well-supported model and, thus, cannot be a canonical model of $grd(P) + \mathcal{C}$. Hence, $\mathcal{I}$ cannot be canonical model of $P + \mathcal{C}$. Thus, we get a contradiction. ∎

The converse of the above theorem does not hold. That is, it is not true that every model of P that satisfies $\mathcal{C}$ is a canonical model of $P+\mathcal{C}$. This was illustrated in Examples 4.4.1 and 4.4.4.

Definition 4.4.9 $P + \mathcal{C}$ *strongly* entails a literal p under **C4** if, and only if, p evaluates to T in all the canonical models of $P + \mathcal{C}$.

$P + \mathcal{C}$ *weakly* entails a literal p under **C4** if, and only if, p evaluates to *at least* CT in all the canonical models of $P + \mathcal{C}$.

It is clear that if $P + \mathcal{C}$ strongly entails p then it weakly entails p.

Theorem 4.4.4 **C4** *as a semantics of normal logic programs augmented with a set of contestations is inferentially conflict-free with regard to the conflicts specified by the set of contestations.*

Proof: Let P be a normal logic program and let $\mathcal{C}$ be a set of contestations. Then we can establish that **C4** is inferentially conflict free by establishing that the set of strong and weak entailments of $P + \mathcal{C}$ satisfy $\mathcal{C}$. This follows trivially from Theorem 4.4.3 above. ∎

In the case where $\mathcal{C} = \emptyset$, the canonical models of $P + \mathcal{C} = P$ are simply the clausally maximal models among the well-supported models of P. Thus, **C4** provides a new semantics for normal logic programs. It is clear that the definitions of weak and strong entailment carry over to the case when $\mathcal{C} = \emptyset$. We explore this new semantics of normal logic programs in Chapter 5.

4.5 Discussion

In this chapter we have introduced the idea of contestations, which is a way of representing conflicts between statements. A contestation against a statement is also taken as evidence against the statement, whereas a normal logic rule with that statement in the head is understood as stating evidence in favor of that statement. There can be contestations of different degrees of strength. We have introduced **C4**, a semantics for normal logic programs augmented with contestations. This semantics is based on four truth values with an associated ordering between them. Our semantics is based on the idea of epistemically reasonable models which is captured in terms of the idea of well-supported models, a clausal ordering between well-supported models and the idea of combining evidence against a statement with the evidence for that statement. The canonical models of a normal logic program plus a set of contestations are the well-supported models which are maximal in the clausal ordering. Based on this model theory we have introduced two types of entailment: strong entailment and weak entailment. We have shown that every normal logic program augmented with contestations which are interpreted in a certain way has at least one canonical model.

In the following we compare the truth values of **C4** and the associated ordering with other multi-valued logics in terms of the ground program $P = \{a \leftarrow;\ b \leftarrow\}$ and the set of contestations $\mathcal{C} = \{b \hookrightarrow_1 a\}$. First, classical two-valued logic cannot provide a model for $P + \mathcal{C}$. The only model of P, which assigns true to both a and b, does not satisfy $b \hookrightarrow_1 a$. A three valued logic ([Kle50]) would provide as a model of $P + \mathcal{C}$ the interpretation which assigns $\mathbf{u}$ to a and T to b. As noted in Chapter 2, such a logic would have to consider $\mathbf{u}$ as a designated truth value. Thus, $P + \mathcal{C}$ would entail both a and the negation of a in terms of a three-valued logic. Hence, such a logic would not be inferentially conflict-free. A Belnap type of four values ([Bel77b]) would presumably assign T to b and $\{T, F\}$ to a. Thus, **not** a would also evaluate to $\{T, F\}$. Depending on the rules for entailment, this would have the consequence that $P + \mathcal{C}$ above would entail both a and **not** a or it would entail neither. Both of these seem to us undesirable consequences. In many contexts it would be useful to infer the negation of a successfully contested statement. $P + \mathcal{C}$ should entail **not** a without entailing a, as in **C4**.

In Section 4.2 we showed how the four truth values of **C4** and the ordering between them can naturally be derived from the classical truth values T and F in terms of two players assigning T or F to a set of sentences, where player 2's assignments are allowed to dominate player 1's assignment without winning outright. Instead of the two players we can also think of this in terms of evidence for a statement and evidence contrary to the statement. Thus, evidence for a statement plays the role of player 1 and evidence contrary to a statement plays the role of player 2. This will make clear why in ordering the truth values we have allowed player 2 to dominate player 1, the proponent of the theory in question. In this case player 2's assigning F to a statement means a contestor of the statement can be assigned T on the basis of available evidence and player 2's assigning T to a statement means a contestor of the negation of the statement can similarly be assigned T. With this interpretation it can be seen that the situation in which a statement is assigned $\langle F, T\rangle$ is to be preferred to a situation in which $\langle T, F\rangle$ is assigned to that statement because the former situation means there is no evidence against the statement but there is evidence against the negation of the statement whereas the latter situation means there is evidence against the statement even if there is evidence for the statement. Thus, the former situation is more cautious than the latter situation.

The work presented in this chapter has some connections with the work done on argumentation by Dung and his collaborators ([Dun93], [DKT96]). An argumentation framework is a pair $\langle AR, attacks\rangle$ where AR is a set of arguments and $attacks \subseteq AR \times AR$. A set S of arguments is said to be conflict-free if no two elements of it attack each other. A conflict-free set of arguments S is admissible if and only if for each argument B, if B attacks S then B is attacked by S. And a preferred extension of an argumentation framework AF is a maximal (with respect to set inclusion) admissible set of AF.

A contests b can be understood as saying that A attacks b (or that A is an argument against b) if A can be established. The set S of literals which evaluate to T or CT in a canonical model of $LP + \mathcal{C}$, where LP is a normal logic program and $\mathcal{C}$ is a set of contestations, can be interpreted as a preferred extension as defined above. The main difference between our work and the work on argumentation described above is that our work has the explicit semantic machinery to give a model theory. The cap functions associated with a contestation, the rules for semantically evaluating logic programming rules constrained by a set of contestations and the definition of a well-supported model ensures that both a contested atom and its contestor cannot have a designated truth value (T or CT) in any well-supported model of a normal logic program constrained by a set of contestations. In contrast, there is no such semantic machinery in the work on argumentation. The idea of one argument attacking another argument is introduced as a primitive. Therefore there is nothing in the semantics of 'attacks' which ensures that in a preferred extension there cannot be mutually attacking arguments except by explicitly defining preferred extensions so that only conflict-free sets are regarded as preferred extensions. If argument A attacks argument B and the conclusion of A is p and the conclusion of B is q, then what is needed is a semantic characterization of the relation between p and q which shows why establishing p disallows establishing q on the basis of B. When p and q are negations of each other the semantics of negation provides this semantic characterization. But when p and q are not negations of each other simply saying A attacks B provides no insight into why both p and q should not be accepted and provides no semantic machinery that precludes accepting q (on the basis of B) when p is accepted. Indeed, argumentation theory has only an operational semantics in terms of the fix-point of an operator. But there is nothing in the definition of this operator or its fix-point which precludes two mutually attacking arguments from belonging in the fix-point.

Another difference between our work and the work on argumentation theory is that we can accommodate contestations of different degrees of strength. The work on argumentation theory cannot accommodate this. Since it has only an operational semantics, either an argument is successfully attacked by another argument or not. It cannot allow for different degrees of successes of attacks.

4.6 Summary

In this chapter we have introduced the idea of contestations and provided the **C4** semantics for normal logic programs augmented with a set of contestations. More specifically, the research contributions of this chapter are summarized as follows.

- We introduce the four truth values $\mathcal{V} = \{F,\ CF,\ CT,\ T\}$ and provide a function for evaluating any closed sentence of the language of the program given an interpretation of the program based on $\mathcal{V}$ (Section 4.2).

- We show how the four truth values of **C4** and the associated ordering between them can naturally be derived from the classical truth values T and F in the context of two players assigning the classical truth values to the same set of statements, where one player's assignment is allowed to dominate the other player's assignment without outright winning against the other player's assignment (Section 4.2).

- We generalize the idea of a two-valued well-supported model of a program to a four-valued well-supported model, and we prove that every normal logic program has a four-valued well-supported model (Section 4.3).

- We introduce contestations, which is a way of representing conflicts between statements. A contestation $B \hookrightarrow_i a$, where B is a conjunction of ground literals and a is a ground atom, is understood as saying that if B has the truth-value α then a has at most the truth-value $cap_i(\alpha)$, where cap_i is a mapping from $\mathcal{V}$ to $\mathcal{V}$. Contestations of different degrees of strength are defined in terms of different *cap* functions (Subsection 4.4.1).

- We have defined **C4**, a semantics for normal logic programs augmented with contestations of different degrees of strength. This semantics is based on the four truth values of $\mathcal{V}$ with an associated ordering between them. The semantics is based on the idea of epistemically reasonable models which is captured in terms of the idea of well-supported models, a clausal ordering between well-supported models and the idea of combining evidence against a statement with the evidence for that statement. The canonical models of a normal logic program plus a set of contestations are defined as the well-supported models which are maximal in the clausal ordering (Subsection 4.4.3).

- We have shown that every normal logic program augmented with contestations which are defined in terms of a certain *cap* function has at least one canonical model (Subsection 4.4.3).

- Based on this model theory we have introduced two types of entailment: strong entailment and weak entailment. And we have proven that the inferences permitted in terms of these entailment relations are conflict-free with respect to the types of conflicts specified in terms of $\mathcal{C}$ (Subsection 4.4.3).

Chapter 5

C4 as a semantics of normal logic programs

5.1 Introduction

In the case where $\mathcal{C} = \emptyset$, the canonical models of $P+\mathcal{C} = P$ are simply the clausally maximal models among the well-supported models of P. Thus, **C4** provides a new semantics for normal logic programs. It is clear that the definitions of weak and strong entailment carry over to the case when $\mathcal{C} = \emptyset$.

In Section 5.2 we investigate **C4** as a semantics of normal logic programs. We prove that every definite logic program has a unique canonical **C4** model and that every normal logic program has at least one **C4** canonical model. In Section 5.3 we investigate the relation between the stable model semantics and **C4** as semantics of normal logic programs. We prove that a normal logic program which has any stable models entails a literal with respect to the stable models of that program if, and only if, that program weakly entails that literal under **C4**. In Section 5.4 we investigate the relation between the well founded semantics and **C4** as semantics of normal logic programs. We prove that a normal logic program entails a literal with respect to the well founded semantics if, and only if, that program strongly entails that literal under **C4**. In Section 5.5 we show how our formalism can be extended to express conjunctive queries one part of which must be answered in terms of strong entailment and another part of which may be answered in terms of weak entailment. In Section 5.6 we compare **C4** as a semantics of normal logic programs with the stable model semantics and the well founded semantics. In Section 5.7 we summarize the main research contributions of this chapter.

5.2 C4 as a Semantics of Normal Logic Programs

In this section we investigate the properties of **C4** as a semantics of normal logic programs.

Theorem 5.2.1 *Every normal logic program has at least one* **C4** *canonical model.*

Proof: Follows directly from Theorem 4.3.1 in Chapter 4. ∎

It can also be established that every definite logic program has a unique canonical model. This generalizes the theorem of Kowalski and van Emden ([vEK76]) which says that every definite logic program has a unique minimal Herbrand model. Our result generalizes the Kowalski and van Emden theorem because of the presence of the special atoms ($true$, $CTrue$, $CFalse$ and $false$) in the bodies of some rules, which can require the unique canonical model to assign truth values other than T and F to atoms. To prove this result we need to generalize the immediate consequence operator, T_P, of [vEK76] defined in Chapter 3. Our discussion here closely follows the three-valued generalization of this operator given in [Prz90b].

Definition 5.2.1 *Let P be a ground logic program, let $\mathcal{I}$ be a four-valued interpretation of P, and let $a \in HB_P$. Define $\Psi(\mathcal{I})$ to be the interpretation given by:*

1. $\Psi(\mathcal{I})(a) = 1$ *if there is a* $C \in P$ *such that* $head(C) = a$ *and* $\mathcal{I}(body(C)) = 1$;

2. $\Psi(\mathcal{I})(a) = 2/3$ *if* $\Psi(\mathcal{I})(a) \neq 1$ *and if there is a* $C \in P$ *such that* $head(C) = a$ *and* $\mathcal{I}(body(C)) = 2/3$;

3. $\Psi(\mathcal{I})(a) = 1/3$ *if* $\Psi(\mathcal{I})(a) \neq 1$, $\Psi(\mathcal{I})(a) \neq 2/3$ *and if there is a* $C \in P$ *such that* $head(C) = a$ *and* $\mathcal{I}(body(C)) = 1/3$;

4. $\Psi(\mathcal{I})(a) = 0$, *otherwise.*

In terms of the *pointwise* ordering (as opposed to the *clausal* ordering) among interpretations introduced above, the set of Herbrand interpretations of any definite logic program form a complete lattice with the bottom of the lattice being the interpretation which assigns F to all atoms and the top being the interpretation which assigns T to all atoms. Hence, we are assured by the Knaster-Tarski theorem ([Tar55]) that the operator Ψ has a least fixed point.

Example 5.2.1 *Let* $P = \{a \leftarrow b, c;\ b \leftarrow CTrue;\ c \leftarrow CFalse\}$. *Let* $\mathcal{I}$ *be such that* $\mathcal{I}$ *assigns* 0 *to* a, b, *and* c *and assigns the special atoms* $CTrue$ *and* $CFalse$ *their fixed values* 2/3 *and* 1/3 *respectively. Then* $\Psi(\mathcal{I}) = \mathcal{J}$ *assigns* 0 *to* a, 2/3 *to* b *and* 1/3 *to* c. *And* $\Psi(\mathcal{J})$ *assigns* 1/3 *to* a, 2/3 *to* b *and* 1/3 *to* c. *Any further application of the* Ψ *operator to* $\Psi(\mathcal{J})$ *yields the same result as* $\Psi(\mathcal{J})$.

Lemma 5.2.1 *If* P *is a definite logic program, then the operator* Ψ *has the least fixed point* M_P *such that* $\Psi(M_P) = M_P$. *The interpretation* M_P *is the least model of* P *in terms of the pointwise ordering.*

The sequence $\Psi \uparrow n$, $n = 0, 1, \ldots, \omega$, *of iterations of* Ψ *is monotonically increasing with respect to the pointwise ordering among interpretations (starting with the interpretation that assigns* F *to all atoms) and it has a fixed point* $\Psi \uparrow \omega = M_P$.

Proof: The proof is completely analogous to the proof in [vEK76] for two-valued interpretations. ∎

We are now in a position to prove the theorem that every definite logic program has a unique **C4** canonical model.

Theorem 5.2.2 *A definite logic program has a unique* **C4** *canonical model.*

Proof: We show that M_P, the least fix-point of Ψ for P, is the unique canonical model of P.

We know from the above lemma that M_P is a model of P. Furthermore, it is a well-supported model given the nature of the Ψ operator. Note that each clause of P evaluates to T in M_P. Hence, M_P is maximal in the clausal ordering. Thus, it is a canonical model.

Assume by way of contradiction that there is another canonical model $\mathcal{I}$ of P such that $M_P \neq \mathcal{I}$. Let S be the set of atoms on which M_P and $\mathcal{I}$ differ. Note that $\mathcal{I}$ cannot assign any atom a value higher than M_P does if it is a well-supported model.

Let us say that an atom *first appears* in an iteration of Ψ in the construction of M_P if it has its final value (that is, the value it has in M_P), which must be greater than F, in that iteration and has a strictly lower value in all other iterations before that. Thus we can stratify the atoms in S in terms of which iteration of Ψ they first appear in the construction of M_P.

Let $s \in S$ be an atom such that no atom in S first appears before s. Let $C \in P$ be such that $head(C) = s$ and no member of S is a member of $body(C)$ and $body(C)$ evaluates to the same truth value as $M_P(s)$. (There must be such a C given the bottom-up nature of constructing M_P and given that Ψ is a monotonic operator.)

Thus, $\mathcal{I}$ and M_P assign the same truth values to all members of $body(C)$. Hence, C will evaluate to a lesser value in $\mathcal{I}$ than in M_P. But since all clauses evaluate to T in M_P, it follows that $\mathcal{I}$ is strictly less than M_P in the clausal ordering. Thus, a contradiction.

Hence, M_P is the unique **C4** canonical model of P. ■

5.3 Relation to Stable Model Semantics

In this section we prove that P entails a literal q with respect to the stable models of P if, and only if, P weakly entails q (under **C4**).

It is well known that not all normal logic programs have a two-valued stable model. Thus, $P = \{p \leftarrow \textbf{ not } p\}$ has no two-valued stable model. However, this program has a canonical model according to **C4**, namely, the model which assigns CF to p.

Let $Truth(\mathcal{I})$ denote $\{a | \ a$ is an atom and $\mathcal{I}(a) \geq CT\}$.

Lemma 5.3.1 below says that every stable model of a program P is $Truth(\mathcal{I})$ for some canonical model $\mathcal{I}$ (under **C4**) of P.

Lemma 5.3.1 *Let P be $grd(LP)$. Then, for each stable model M of P, there exists a four-valued canonical model $\mathcal{I}$ of LP such that $M = Truth(\mathcal{I})$.*

Proof: Let M be a stable model of P. We show below how to construct a four-valued canonical model $\mathcal{I}$ such that $M = Truth(\mathcal{I})$.

Let $\mathcal{I}$ be such that it assigns T to all members of M and F to all other atoms. Clearly, by construction $M = Truth(\mathcal{I})$. We show below that $\mathcal{I}$ is a canonical model of P.

<u>$\mathcal{I}$ is a model of P</u>

For any $C \in P$, if $head(C) \in M$, then, by construction, $\mathcal{I}(head(C)) = T$. Thus, $\mathcal{I}$ models C.

If $head(C) \notin M$, then either some atom in $posbody(C)$ is not in M or some literal in $negbody(C)$ is false in M. In either case, by construction, that atom or literal evaluates to F in $\mathcal{I}$. So, once again, $\mathcal{I}$ models C.

<u>$\mathcal{I}$ is well-supported</u>

Since $\mathcal{I}$ assigns only T or F to atoms, $\mathcal{I}$ is identical to M if M is thought of as a two-valued mapping. Since M is a stable model of P, it is also a well-supported model of P ([Fag91]). So $\mathcal{I}$ is a well-supported model of P.

<u>$\mathcal{I}$ is maximal in the clausal ordering with respect to LP</u>

$\mathcal{I}$ is maximal in the clausal ordering with respect to LP only if it is maximal in the clausal ordering with respect to $grd(LP) = P$. To establish that $\mathcal{I}$ is maximal in the clausal ordering with respect to P it is enough to establish that all clauses in P evaluate to T according to $\mathcal{I}$.

Clearly, all clauses such that its head is assigned T by $\mathcal{I}$ evaluate to T. So, all that remains to be shown is that all clauses such that its head is assigned F by $\mathcal{I}$ also evaluate to T. But since we have already established that $\mathcal{I}$ is a model of P, the body of any clause whose head is assigned F must evaluate to F. Thus, any such clause evaluates to T in $\mathcal{I}$. Hence, all clauses in P evaluate to T under $\mathcal{I}$.

Hence, $\mathcal{I}$ is a canonical model of P and $M = Truth(\mathcal{I})$. ■

Corollary 1 *If a ground program P has a stable model, then P has a four-valued* *canonical model $\mathcal{I}$ such that each clause in P evaluates to T in $\mathcal{I}$.*

Proof: This was essentially proved in the proof of the previous lemma. ■

Corollary 2 *If a ground program P has a stable model, then every four-valued canonical model of P is such that each clause in P evaluates to T in it.*

Proof: By Corollary 1 we know that if P has a stable model then there is a canonical model $\mathcal{I}$ of P such that all clauses evaluate to T in $\mathcal{I}$. So for any model $\mathcal{J}$ of P such that $\mathcal{J}(C) < T$, it must be the case that $\mathcal{J} < \mathcal{I}$. Hence $\mathcal{J}$ cannot be cannonical. Thus, every cannonical model must be such that every clause of P evaluates to T in it. ■

Lemma 5.3.2 *Let P be $grd(LP)$. Every canonical four-valued model $\mathcal{I}$ of LP such that each clause of P evaluates to T in $\mathcal{I}$ is such that $Truth(\mathcal{I})$ is a stable model of P.*

Proof: Assume that $\mathcal{I}$ is a canonical model of LP such that each clause of LP evaluates to T in $\mathcal{I}$. Assume, by way of contradiction, that $Truth(\mathcal{I})$ is not a stable model of P. This implies that $MM(P^{Truth(\mathcal{I})}) \neq Truth(\mathcal{I})$, where $P^{Truth(\mathcal{I})}$ is the Gelfond-Lifschitz transformation (see Definition 3.3.1 of Chapter 3) of P with respect to $Truth(\mathcal{I})$.

This means that either there is an $a \in MM(P^{Truth(\mathcal{I})})$ such that $a \notin Truth(\mathcal{I})$, or there is an $a \in Truth(\mathcal{I})$ such that $a \notin MM(P^{Truth(\mathcal{I})})$.

<u>Case 1</u>: $a \in MM(P^{Truth(\mathcal{I})})$ and $a \notin Truth(\mathcal{I})$.

$P^{Truth(\mathcal{I})}$ is a definite program and, hence $MM(P^{Truth(\mathcal{I})})$ is the least fix-point of $T_Q \uparrow n$, where Q is $P^{Truth(\mathcal{I})}$. Thus, it is possible to stratify the members of $MM(P^{Truth(\mathcal{I})})$ in terms of the least n such that a member first occurs in $T_Q \uparrow n$.

Let a be of the lowest strata among those atoms in $MM(P^{Truth(\mathcal{I})})$ which are not in $Truth(\mathcal{I})$.

Since $a \in MM(P^{Truth(\mathcal{I})})$, there must be a clause in $P^{Truth(\mathcal{I})}$ of the form $a \leftarrow b_1, \ldots, b_m$ such that $\{b_1, \ldots, b_m\} \subseteq MM(P^{Truth(\mathcal{I})})$. But, by the assumption that a is of the lowest strata among those atoms in $MM(P^{Truth(\mathcal{I})})$ which are not in $Truth(\mathcal{I})$, it follows that $\{b_1, \ldots, b_m\} \subseteq Truth(\mathcal{I})$. Furthermore, since $a \leftarrow b_1, \ldots, b_m$ is in $P^{Truth(\mathcal{I})}$, there must be a clause C in P of the form $a \leftarrow b_1, \ldots, b_m,$ **not** $c_1, \ldots,$ **not** c_n such that $c_i \notin Truth(\mathcal{I})$, $i = 1, \ldots, n$. So, each member b_i of $posbody(C)$ is assigned at least CT by $\mathcal{I}$ (since each such b_i belongs to $Truth(\mathcal{I})$) and each member **not** c_j of $negbody(C)$ evaluates to at least CT (since each c_j is assigned at most CF by $\mathcal{I}$). Hence, $\mathcal{I}(body(C))$ is at least CT. By the assumption that C evaluates to T in $\mathcal{I}$, it follows that $\mathcal{I}(a)$ must be at least CT. Therefore, a must be in $Truth(\mathcal{I})$. Thus, a contradiction.

<u>Case 2</u>: $a \notin MM(P^{Truth(\mathcal{I})})$ and $a \in Truth(\mathcal{I})$.

Let $\ll$ be the well-founded ordering that makes $\mathcal{I}$ well supported. Among all the atoms x such that $x \notin MM(P^{Truth(\mathcal{I})})$ and $x \in Truth(\mathcal{I})$, let a be highest in the $\ll$. That is, let a be such that there does not exist a b such that $b \notin MM(P^{Truth(\mathcal{I})})$ and $b \in Truth(\mathcal{I})$ and $a \ll b$.

Since $a \in Truth(\mathcal{I})$, $\mathcal{I}(a)$ is at least CT and, hence, there must be a clause C in P of the form $a \leftarrow b_1, \ldots, b_m,$ **not** $c_1, \ldots,$ **not** c_n such that $body(C)$ must evaluate to at least CT under $\mathcal{I}$ (otherwise, $\mathcal{I}$ would not be well-supported). So each b_i in $posbody(C)$ must be assigned at least CT by $\mathcal{I}$. Thus, $\{b_1, \ldots, b_m\} \subseteq Truth(\mathcal{I})$. Furthermore, since each **not** c_j in $negbody(C)$ must evaluate to at least CT, each c_j must be assigned at most CF by $\mathcal{I}$. Thus, no c_j is in $Truth(\mathcal{I})$.

Hence, clearly, $a \leftarrow b_1, \ldots, b_m$ must be in $P^{Truth(\mathcal{I})}$.

By the nature of the well-founded ordering that makes $\mathcal{I}$ well supported, each of $b_1, \ldots, b_m$ must be lower than a in the well-founded ordering (otherwise a cannot be well-supported by C). By our assumption that a is the highest in the well-founded ordering, it follows that $\{b_1, \ldots, b_m\} \subseteq MM(P^{Truth(\mathcal{I})})$ since $\{b_1, \ldots, b_m\} \subseteq Truth(\mathcal{I})$. So a must belong to $MM(P^{Truth(\mathcal{I})})$. Thus, a contradiction. ■

Lemma 5.3.3 *Let P be $grd(LP)$. If P has any stable models then every canonical model $\mathcal{I}$ of LP is such that $Truth(\mathcal{I})$ is a stable model of P.*

Proof: Follows directly from Corollary 2 and Lemma 5.3.2. ■

Theorem 5.3.1 *If a ground normal logic program P has any stable models, then M is a stable model of P if, and only if, there exists a four valued canonical model of $\mathcal{I}$ of LP such that $M = Truth(\mathcal{I})$.*

Proof: Follows directly from Lemmas 5.3.1 and 5.3.3. ■

Since not all normal logic programs have stable models, an important question is what are the necessary and sufficient conditions for a normal logic program having a stable model. The following theorem gives an answer.

Theorem 5.3.2 *A ground normal logic program has a two-valued stable model if, and only if, every clause of the program evaluates to T in every canonical model of the program.*

Proof: The left-to-right direction is proven in Corollary 2. The right-to-left direction is proven in Lemma 5.3.2. ■

Theorem 5.3.2 above justifies the following definition.

Definition 5.3.1 *A* **C4** *model of a normal logic program is a C-Stable model if and only if all rules of the program evaluate to T in that model.*

We show below that any well-supported C-stable model of a normal logic program must be a canonical model of the program and if the program has a canonical C-stable model then all its canonical models must be C-stable.

Theorem 5.3.3 *Any well-supported C-stable model of a normal logic program must be a canonical model of the program and if the program has a canonical C-stable model then all its canonical models must be C-stable.*

Proof: Let P be a normal logic program which has well-supported C-stable model $\mathcal{I}$. Since $\mathcal{I}$ is well-supported and since every $R \in P$ evaluates to T in $\mathcal{I}$ there cannot be any other model of P which is strictly greater than $\mathcal{I}$ in the clausal ordering. Hence $\mathcal{I}$ must be a canonical model of P.

Given that $\mathcal{I}$ is a canonical C-stable model of P, it follows that any model $\mathcal{J}$ of P such that $\mathcal{J}$ is not C-stable would be strictly less than $\mathcal{I}$ in the clausal ordering. Thus, no such $\mathcal{J}$ could be a canonical model of P. Hence, it follows that if P has a canonical C-stable model, then all its canonical models must be C-stable. ∎

Let us say that P entails a sentence q *under the stable model semantics* if, and only if, every stable model of P is also a model of q.

Theorem 5.3.4 *If a ground normal logic program P has any stable models then it entails a sentence q under the stable model semantics if, and only if, P weakly entails q under* **C4**.

Proof: Follows directly from Theorem 5.3.1. ∎

Using the terminology of [Dix95], we state the following theorem.

Theorem 5.3.5 *If T and CT are collapsed into a single true value and CF and F are collapsed into a single false value,* **C4** *extends the stable model semantics both in the sense that*

- *For any program P,* **C4** *classifies at least as many atoms of P as true or false as does the stable model semantics.*
- **C4** *is defined for a class of programs that strictly includes the class of programs for which stable model semantics is defined and for all programs of this smaller class, the two semantics coincide.*

Proof: Since **C4** assigns a truth value to all atoms of P, it follows trivially that **C4** classifies at least as many atoms of P as true or false as does the stable model semantics, if T and CT are collapsed into true and CF and F are collapsed into false.

It follows from Lemma 5.3.1 that **C4** is defined for a class of programs that includes the class of programs for which stable model semantics is defined and for all programs of this smaller class, the two semantics coincide. So to prove the second part of the theorem all we need to do is produce a program which has no stable models, but for which **C4** has a model. The program $\{p \leftarrow$ **not** $p\}$ has no stable models, but it has a model under **C4**, namely, the model which assigns CF to p. ∎

Following [Prz90b], we define a four-valued stability operator $\Gamma^\star$ on normal, logic programs.

Definition 5.3.2 *Given a four-valued interpretation $\mathcal{I}$ of a normal, logic program P, let $LP^{\mathcal{I}}$ be the* definite *program obtained by transforming every clause C by* *replacing every member of negbody(C) which evaluates to T (resp. CT; resp. CF; resp. F) by the special atom true (resp. $Ctrue$; resp. $Cfalse$; resp. $false$) which evaluates to T (resp. CT; resp. CF; resp. F) in every interpretation. Let $\mathcal{J}$ be the unique canonical model of $LP^{\mathcal{I}}$. We define $\mathcal{J}$ to be the value of $\Gamma^{\star}(\mathcal{I})$.*

We say that $\mathcal{I}$ is a four-valued stable model of P if, and only if, $\Gamma^{\star}(\mathcal{I}) = \mathcal{I}$.

Not all normal, logic programs have a four-valued stable model. The program $\{p \leftarrow$ **not** $p\}$ has no four-valued stable model. However, it does have a four-valued well-supported model in which p is assigned CF. This shows that although the set of two-valued stable models of a program coincide with the set of two-valued well-supported models ([Fag91]), this equivalence does not hold for four-valued models.

Lemma 5.3.4 *$\mathcal{I}$ is a four-valued stable model of P if, and only if, for each $a \in HB_P$,*

$$\mathcal{I}(a) = \; max\{\mathcal{I}(body_1(a)), \ldots, \mathcal{I}(body_n(a))\}$$

where $body_1(a), \ldots, body_n(a)$ are the bodies of all the clauses in P which have a in the head.

Proof:

$\Rightarrow$

If $\mathcal{I}$ is a four-valued stable model then it must be the unique canonical model of $LP^{\mathcal{I}}$, which can be computed by iterating the Ψ operator. Given the bottom-up nature of this computation and given the monotonicity of the Ψ operator, it must be the case that for each $a \in HB_P$

$$\mathcal{I}(a) = \; max\{\mathcal{I}(body_1(a)), \ldots, \mathcal{I}(body_n(a))\}.$$

$\Leftarrow$

Let $\mathcal{I}$ be such that for each $a \in HB_P$,

$$\mathcal{I}(a) = \ max\{\mathcal{I}(body_1(a)), \ldots, \mathcal{I}(body_n(a))\}.$$

Let $\mathcal{J}$ be the unique canonical model of $LP^{\mathcal{I}}$. Given that $\mathcal{J}$ is the least fix-point of the Ψ operator, it is easy to see that $\mathcal{J}$ is a well-supported model. Thus, there is a well-founded order $\ll$ on the atoms of $HB_P = HB_{LP^{\mathcal{I}}}$. Based on this ordering we construct an inductive proof that for each $a \in HB_P$, $\mathcal{J}(a) = \mathcal{I}(a)$.

The ordering $\ll$ consists of a set of chains. We take the bottom of each chain to be in position 0, the next atom in the chain to be in position 1, and so on. Define the *rank* of each atom to be highest position it has in any chain in $\ll$ ([Fag91]). Inductive proof based on the *rank* of an atom.

Base Case: $rank = 0$

Only the special atoms ($true$, $false$) can be at the bottom of any chain since $LP^{\mathcal{I}}$ is a definite program. Necessarily, $\mathcal{I}$ and $\mathcal{J}$ assign the same value to all special atoms.

Inductive Step: Assume that $\mathcal{I}$ and $\mathcal{J}$ agree on all atoms of *rank* $j < n$. We show below that this is true for all atoms of *rank* n.

Let a be any atom of *rank* n. As noted in the left-to-right part above, $\mathcal{J}(a) = max\{\mathcal{J}(body_1(a)), \ldots, \mathcal{J}(body_n(a))\}$. Given the bottom-up computation of $\mathcal{J}$, there must be some i, $1 \leq i \leq n$, such that

$$\mathcal{J}(body_i(a)) = max\{\mathcal{J}(body_1(a)), \ldots, \mathcal{J}(body_n(a))\}$$

and every member of $body_i(a)$ is of lesser *rank* than a. But, by the inductive assumption, $\mathcal{I}$ and $\mathcal{J}$ agree on all members of $body_i(a)$. Hence, it follows that $\mathcal{I}$

and $\mathcal{J}$ agree on a.

It follows thus that $\mathcal{I}$ and $\mathcal{J}$ are identical. Hence, by the definition of a stable model, $\mathcal{I}$ is a stable model of P. ∎

We use Lemma 5.3.4 above in the proof of the following theorem which states precisely the relation between four-valued stable models and four-valued well-supported models.

Theorem 5.3.6 *If P has a four-valued stable model, then $\mathcal{I}$ is a four-valued stable model of P if, and only if, $\mathcal{I}$ is a clausally maximal four-valued well-supported model of P.*

Proof:

$\Rightarrow$

Let $\mathcal{I}$ be a four-valued stable model of P. Then, by Lemma 5.3.4, for each atom $a \in HB_P$, $\mathcal{I}(a) = max\{\mathcal{I}(body_1(a)), \ldots, \mathcal{I}(body_n(a))\}$. Hence, each clause evaluates to T in $\mathcal{I}$. Hence, $\mathcal{I}$ must be a clausally maximal, well-supported model of P.

$\Leftarrow$

Assume that P has a four-valued stable model $\mathcal{J}$.

Let $\mathcal{I}$ be a clausally maximal four-valued well-supported model of P. Hence, for each $a \in HB_P$,

$$\mathcal{I}(a) = max\{\mathcal{I}(body_1(a)), \ldots, \mathcal{I}(body_n(a))\},$$

otherwise $\mathcal{I}$ would be less than $\mathcal{J}$ in the clausal ordering. But then by Lemma 5.3.4, $\mathcal{I}$ must be a four-valued stable model of P. ∎

5.4 Relation to Well Founded Semantics

In this subsection we show that LP entails a ground literal p under the well founded semantics if, and only if, LP strongly entails p under **C4**. That is, we show that if p is a positive atom, $p \in WFS(LP)$ if, and only if, p is assigned T in all the four-valued canonical models of LP, and if p is a negative literal then $p \in WFS(LP)$ if, and only if, p evaluates to F in all the four-valued canonical models of LP.

Definition 5.4.1

$$\mathcal{T}(LP) = \{a \in HB_{LP} \mid a \in WFS(LP)\}$$

$$\mathcal{F}(LP) = \{a \in HB_{LP} \mid \textbf{not } a \in WFS(LP)\}$$

$$\perp(LP) = \{a \in HB_{LP} \mid a \notin \mathcal{T}(LP) \textit{ and } a \notin \mathcal{F}(LP)\}$$

Lemma 5.4.1 *If a positive (resp., negative) literal $a \in WFS(LP)$, then a is assigned T (resp., F) in all the four-valued canonical models of LP.*

Proof: $a \in WFS(LP)$ if, and only if, $a \in I^{\infty}$. We prove the lemma by proving inductively that for each ordinal α, that if a positive (resp., negative) literal $a \in I_{\alpha}$ then a is assigned T (resp., F) in all the four-valued canonical models of LP. Thus, it must be true for I^{∞}.

Base Case. $\alpha = 0$. The claim is trivially true since $I_0 = \emptyset$.

Inductive Step. Assume that the claim is true for all $\beta < \alpha$. We show that the claim is also true for α.

If α is a limit ordinal then

$$I_{\alpha} = \bigcup_{\beta < \alpha} I_{\beta}$$

Since the claim is true for all $\beta < \alpha$ the claim is also true for $\bigcup_{\beta<\alpha} I_{\beta}$.

If α is a successor ordinal then

$$I_\alpha = W_P(I_{\alpha-1}) = T_P(I_{\alpha-1}) \cup \textbf{not}\ U_P(I_{\alpha-1})$$

If $a \in I_\alpha$ then $a \in T_P(I_{\alpha-1})$. So there must be a rule

$$R \ = \ a \leftarrow b_1, \ldots, b_m, \textbf{not}\ c_1, \ldots, \textbf{not}\ c_n$$

such that $\{b_1, \ldots, b_m\} \subseteq I_{\alpha-1}$ and $\{\textbf{not}\ c_1, \ldots, \textbf{not}\ c_n\} \subseteq I_{\alpha-1}$. By the inductive hypothesis $b_1, \ldots, b_m$ are assigned T and $c_1, \ldots, c_n$ are assigned F in all the four valued canonical models. So $body(R)$ must evaluate to T in any canonical model of LP and, hence, in any such model $head(R)$ must be assigned T. Hence if $a \in I_\alpha$ then a must be assigned T in all the four valued canonical models of LP.

If **not** $a \in I_\alpha$ then a must be in G, the greatest unfounded set with respect to $I_{\alpha-1}$. We show below that every member of G gets assigned F in every canonical model.

It follows directly from the inductive assumption that any $b \in G$ gets assigned F in every canonical interpretation if every rule with b in the head evaluates to false with respect to $I_{\alpha-1}$. Let G' be the subset of G such that members of G' do not evaluate to false in this way. If G' is empty, the claim that all members of G are assigned F in all the canonical models stands proved. Hence, assume G' is not empty.

For each member of G', every rule with it in the head such that the body of the rule does not evaluate to false with respect to $I_{\alpha-1}$ contains some member of G' in the body. If any member of G' gets assigned a truth-value greater than F in any canonical model, then it must be well-supported in that canonical model by some rule. But any such rule must contain some member of G'. So that member of G' would have to be similarly well-supported by a rule containing a member of

G'. Thus, the members of G' would have to be well-supported in terms of each other. But this is not possible since there cannot be any cycles in the well-founded ordering which makes a canonical model a well-supported model. Thus, no member of G' can be assigned a value higher than F in any canonical model.

Hence, all members of G, the greatest unfounded set with respect to $I_{\alpha-1}$, are assigned F in every canonical model. Thus, if **not** $a \in I_\alpha$ then a is assigned F in all the canonical models. ∎

The following definitions are needed to prove the next lemma.

Definition 5.4.2 For any $C \in LP$, where LP is a ground program, $Residue(C)$ is the rule obtained by deleting all *literals* from $body(C)$ which are true in $WFS(LP)$.

Definition 5.4.3 Let $Residue(LP) = \{Residue(C) \mid C \in LP,\ head(C) \notin WFS(LP),$ **not** $head(C) \notin WFS(LP)$ and no member of $body(C)$ is false in $WFS(LP)\}$

That is, $Residue(LP)$ is the set of rules obtained by deleting all rules $C \in LP$ such that $head(C) \in WFS(LP)$ or **not** $head(C) \in WFS(LP)$ or whose body is false in $WFS(LP)$ and of the remaining rules, deleting all literals which are true in $WFS(LP)$ from the bodies of such rules. It is easy to see that $Residue(LP)$ is the part of LP that cannot be used in computing $WFS(LP)$.

Example 5.4.1 *Let P be as in Example 3.3.3. We saw in Example 3.3.3 that $WFS(P) = \{c, \mathbf{not}\ r, \mathbf{not}\ q\}$. Hence, $Residue(P) =$*

$$\begin{array}{ll} p \leftarrow a & p \leftarrow b \\ a \leftarrow \mathbf{not}\ b & b \leftarrow \mathbf{not}\ a \end{array}$$

Let $ResidueHeads(LP)$ be the set of heads of all rules in $Residue(LP)$. Then, given the nature of the U_P operator in the definition of the well-founded semantics, it is also easy to see that $a \in ResidueHeads(LP)$ if, and only if, every member of every rule in $Residue(LP)$ with a in the head is also in $ResidueHeads(LP)$. Let $Atoms(Residue(LP))$ be the set of those atoms which occur in some rule in $Residue(LP)$. Then $ResidueHeads(LP) = Atoms(Residue(LP))$.

The next two lemmas are needed to prove Theorem 5.4.1.

Lemma 5.4.2 *For any $a \in HB_{LP}$, $a \in Atoms(Residue(LP))$ if and only if $a \notin WFS(LP)$ and* **not** *$a \notin WFS(LP)$.*

Proof:

$\Leftarrow$

If $a \notin WFS(LP)$ and **not** $a \notin WFS(LP)$, then there must be a $C \in LP$ such that $head(C) = a$, otherwise, given the nature of the U_P operator, **not** $a \in WFS(LP)$. But for such a C, $Residue(C) \in Residue(LP)$. Thus, $head(C) = a \in Atoms(Residue(LP))$.

$\Rightarrow$

If $a \in Atoms(Residue(LP))$, then there is a $C \in Residue(LP)$ such that $head(C) = a$. So, by definition of $Residue(LP)$, $a \notin WFS(LP)$ and **not** $a \notin WFS(LP)$.

■

Thus, all members of $Atoms(Residue(LP))$ belong to $\perp(LP)$ and are not assigned a truth value by the well-founded semantics for LP.

Lemma 5.4.3 *For each $a \in Atoms(Residue(LP))$ there exists a canonical model $\mathcal{I}$ of LP such that $\mathcal{I}(a) = CT$ or $\mathcal{I}(a) = CF$.*

Proof: Suppose, by way of contradiction, that there is an $a \in Atoms(Residue(LP))$ such that every canonical model assigns either T or F to a.

Let $\mathcal{I}$ be any canonical model of LP. Clearly $\mathcal{I}$ must assign T or F to some atoms in $Atoms(Residue(LP))$. We construct an interpretation $\mathcal{J}$ such that for every $b \in Atoms(Residue(LP))$, if $\mathcal{I}(b) = T$ then $\mathcal{J}(b) = CT$ and if $\mathcal{I}(b) = F$ then $\mathcal{J}(b) = CF$, and for all other atoms in HB_{LP}, $\mathcal{I}$ and $\mathcal{J}$ assign the same truth value.

We show below that $\mathcal{J}$ is a canonical model, which contradicts the assumption that there is no such canonical model.

<u>$\mathcal{J}$ is a model</u>

Suppose by way of contradiction that $\mathcal{J}$ is not a model of some $C \in P$ where $P = grd(LP)$.

Either $head(C) \in Atoms(Residue(LP))$ or not.

Case 1: $head(C) \in Atoms(Residue(LP))$.

Case 1a: $\mathcal{I}(head(C)) = T$ or $\mathcal{I}(head(C)) = F$. In the first case $\mathcal{J}(head(C)) = CT$ and in the second case $\mathcal{J}(head(C)) = CF$. However, in either case $\mathcal{J}$ can fail to be a model of C only if $body(C)$ evaluates to T in $\mathcal{J}$. But since $head(C) \in Atoms(Residue(LP))$, at least one member l of $body(C)$ must also be in $Atoms(Residue(LP))$. If $\mathcal{I}(l) = T$ then $\mathcal{J}(l) = CT$. On the other hand if $\mathcal{I}(l) < T$ then $\mathcal{J}(l) < T$. So in either case $\mathcal{J}(body(C)) < T$. So $\mathcal{J}$ must be a model of C.

Case 1b: $\mathcal{I}(head(C)) = CT$ or $\mathcal{I}(head(C)) = CF$. So $\mathcal{J}(head(C)) = CT$ or $\mathcal{J}(head(C)) = CF$. But in either case since $\mathcal{I}$ is a model of LP, clearly, $\mathcal{I}(body(C)) < T$. So, it must be the case that $\mathcal{J}(body(C)) < T$. Hence, $\mathcal{J}$ must be a model of C.

Case 2: $head(C) \notin Atoms(Residue(LP))$. So, by Lemma 5.4.2, $head(C) \in WFS(LP)$ or **not** $head(C) \in WFS(LP)$. Since $\mathcal{I}$ is a canonical model this implies by Lemma 5.4.1 that $\mathcal{I}(head(C)) = T$ or $\mathcal{I}(head(C)) = F$. By construction $\mathcal{J}$ assigns the same truth value as $\mathcal{I}$ to all atoms a such that $a \notin Atoms(Residue(LP))$. So if $head(C) \in WFS(LP)$, then $\mathcal{J}(head(C)) = T$ and so, clearly, $\mathcal{J}$ is a model of C. If **not** $head(C) \in WFS(LP)$ then $head(C)$ is assigned F by both $\mathcal{I}$ and $\mathcal{J}$. However, **not** $head(C) \in WFS(LP)$ only if some member $l \in body(C)$ is false in $WFS(LP)$. Clearly, by Lemma 5.4.1, l evaluates to F in $\mathcal{I}$. By construction so does $\mathcal{J}$. Hence, $body(C)$ must evaluate to F in $\mathcal{J}$. Thus, again, $\mathcal{J}$ must be a model of C.

Hence, $\mathcal{J}$ must be a model of LP.

$\underline{\mathcal{I} \leq_{LP} \mathcal{J}}$

For each $C \in LP$, either $head(C) \in Atoms(Residue(LP))$ or $head(C) \notin Atoms(Residue(LP))$.

Case 1: $head(C) \in Atoms(Residue(LP))$. Note that for $head(C)$ to be in $Atoms(Residue(LP))$, $body(Residue(C))$ must be non-empty.

If $\mathcal{I}(head(C)) = T$ or $\mathcal{I}(head(C)) = CT$, then $\mathcal{J}(head(C)) = CT$. Since $\mathcal{J}$ is a model (proved above) in either case $\mathcal{J}(body(C))$ is at most CT. So $\mathcal{J}(C)$ is T. Hence, in either case $\mathcal{I}(C) \leq_{LP} \mathcal{J}(C)$.

If $\mathcal{I}(head(C)) = CF$, then $\mathcal{J}(head(C)) = CF$. In this case $\mathcal{I}(body(C))$ is CT or CF or F. By the nature of construction of $\mathcal{J}$, if $\mathcal{I}(body(C))$ is CT or CF, $\mathcal{J}(body(C))$ will have the same truth value. Hence, in either case $\mathcal{I}(C) \leq_{LP} \mathcal{J}(C)$. If $\mathcal{I}(body(C))$ is F, then $\mathcal{I}(body(C))$ is at most CF. So $\mathcal{J}(C)$ evaluates to T. Hence if $\mathcal{I}(head(C)) = CF$, $\mathcal{I}(C) \leq_{LP} \mathcal{J}(C)$.

If $\mathcal{I}(head(C)) = F$, then $\mathcal{J}(head(C)) = CF$. In this case $\mathcal{I}(body(C)) = F$. Hence, $\mathcal{J}(body(C))$ is at most CF. In this case, again, $\mathcal{J}(C) = T$.

So in case $head(C) \in Atoms(Residue(LP))$, $\mathcal{I}(C) \leq_{LP} \mathcal{J}(C)$.

Case 2: $head(C) \notin Atoms(Residue(LP))$. In this case, by Lemma 5.4.2, $\mathcal{I}(head(C)) = T$ or $\mathcal{I}(head(C)) = F$ and, by construction, $\mathcal{I}(head(C) = \mathcal{J}(head(C))$. If $\mathcal{I}(head(C)) = T$, then $\mathcal{I}(C) = \mathcal{J}(C) = T$. If $\mathcal{I}(head(C)) = F = \mathcal{J}(head(C))$, then, since both $\mathcal{I}$ and $\mathcal{J}$ are models, $\mathcal{I}(body(C)) = \mathcal{J}(body(C)) = F$. So, again, $\mathcal{I}(C) = \mathcal{J}(C)$.

Thus, we have shown that $\mathcal{I} \leq_{LP} \mathcal{J}$.

$\mathcal{J}$ is well supported.

Since $\mathcal{I}$ is well-supported, there exists a well-founded ordering $\ll_{\mathcal{I}}$ on atoms in HB_{LP} such that for any $a \in HB_{LP}$ such that $F < \mathcal{I}(a)$, there exists a $C \in LP$ such that $head(C) = a$ and $\mathcal{I}(a) \leq body(C)$ and for any $b \in posbody(C)$, $b \ll_{\mathcal{I}} a$.

We construct $\ll_{\mathcal{J}}$ as follows. If $a \ll_{\mathcal{I}} b$ then $a \ll_{\mathcal{J}} b$ for any $a, b \in HB_{LP}$. Let $S = \{a \in Residue(Atoms(LP)) \mid \mathcal{I}(a) = F\}$. By construction, members of S are assigned CF in $\mathcal{J}$. We let $c \ll_{\mathcal{J}} d$, where $c \in S$ and d is any atom such that $\mathcal{I}(d) \geq CF$. That is, all atoms whose truth value gets upgraded from F to CF in the construction of $\mathcal{J}$ are lesser in the ordering than all atoms which had at least CF in $\mathcal{I}$. Let $\ll_{\mathcal{J}'}$ denote the ordering created thus far. Furthermore since members of S do not belong to any unfounded set with respect to $WFS(LP)$ there must be a well-founded ordering among members of S. We construct one such ordering $\ll_S$ as follows. Let $Pos(Residue(LP))$ be $Residue(LP)$ with negative literal removed from rules of $Residue(LP)$. Since this is a definite logic program the T_P operator applied to it has a least fixed point. $\ll_S$ is constructed by letting $b \ll_S a$ where $a, b \in S$ and b occurs in an earlier iteration of the T_P operator than a. Then $\ll_{\mathcal{J}}$ is just $\ll_{\mathcal{J}'} \cup \ll_S$.

Clearly, $\ll_{\mathcal{J}}$ is a well-founded ordering. We show below that $\mathcal{J}$ is well-supported in terms of $\ll_{\mathcal{J}}$.

Any atom a that is assigned T or CT by $\mathcal{J}$ is well-supported in terms of the same rule $R \in LP$ which would make the assignment of T or CT to a by $\mathcal{I}$ well-supported. Similarly, any $b \notin S$ that is assigned CF by $\mathcal{J}$ is well-supported in terms of the same rule $R \in LP$ which makes the assignment of CF to b by $\mathcal{I}$ well-supported. Furthermore, the part of the $\ll_{\mathcal{J}}$ ordering that is relevant to this is exactly the same as the $\ll_{\mathcal{I}}$ ordering.

Every $c \in S$ is assigned CF by $\mathcal{J}$ but F by $\mathcal{I}$. We need to show that these assignments are also well-supported. Since $c \in Atoms(Residue(LP)$, clearly there must be at least one rule $R \in LP$ such that $head(R) = c$ and $body(Residue(R))$ is non-empty and $d \ll_{\mathcal{J}} c$ for any atom $d \in posbody(Residue(R))$. Any atom in $body(Residue(R))$ which is assigned F by $\mathcal{I}$ is assigned CF in $\mathcal{J}$ and all other members of $body(Residue(R))$ are assigned at least CF in $\mathcal{J}$. Furthermore, all members of $body(R) - body(Residue(R))$ belong to $WFS(LP)$ and thus evaluate to T in $\mathcal{I}$ and, hence, in $\mathcal{J}$. Thus, $body(R)$ must evaluate to at least CF in $\mathcal{J}$. Hence, R supports the attribution of CF to c in $\mathcal{J}$. Furthermore, $b \ll_{\mathcal{J}} c$ for all $b \in posbody(R)$ whether $b \in S$ or $b \notin S$. Hence the attribution of CF to $c \in S$ is well-supported in $\mathcal{J}$.

Thus, $\mathcal{J}$ is a well-supported model of LP.

We have shown that $\mathcal{J}$ is a well-supported model of LP such that $\mathcal{I} \leq_{LP} \mathcal{J}$. But since $\mathcal{I}$ is a canonical model it is not possible that $\mathcal{I} <_{LP} \mathcal{J}$. So it must be the case that $\mathcal{I} =_{LP} \mathcal{J}$. Hence $\mathcal{J}$ must be a canoncial model. Thus, a contradiction.

■

Theorem 5.4.1 *A positive (resp., negative) literal $a \in WFS(LP)$ if, and only if, a is assigned T (resp., F) in all the four-valued canonical models of LP.*

Proof: We have proved the left-to-right direction of the theorem in Lemma 5.4.1. The right-to-left direction can be proven by establishing that if $a \notin WFS(LP)$ and **not** $a \notin WFS(LP)$ then there is a canonical model which assigns neither T nor F to a. This follows directly from Lemma 5.4.2 and Lemma 5.4.3.

■

Theorem 5.4.2 *LP entails a ground literal p under the well founded semantics if, and only if, LP strongly entails p under* **C4**.

Proof: Follows directly from Theorem 5.4.1.

■

5.5 Hybrid Reasoning

Using **C4** we can define a skeptical and a credulous semantics for normal logic programs.

Definition 5.5.1 *The* skeptical *semantics for a normal logic program P are the set of literals* strongly *entailed by P under* **C4**.

In light of Theorem 5.4.1 we can identify the skeptical semantics with the Well-founded semantics.

Definition 5.5.2 *The* credulous *semantics for a normal logic program P are the set of literals* weakly *entailed by P under* **C4**.

In light of Theorem 5.3.4 we can assert that when a normal logic program P has any stable models, then the credulous semantics of P can be identified with the set of literals entailed by P under the stable model semantics.

The following theorem explains why the two semantics are labeled skeptical and credulous.

Theorem 5.5.1 *For any normal logic program P, the skeptical semantics of P is a subset of its credulous semantics.*

Proof: If any literal is strongly entailed by a normal logic program then it is also weakly entailed. Thus the theorem follows directly from the definitions of skeptical and credulous semantics. ■

Reasoning using skeptical (credulous) semantics can be called skeptical (resp., credulous) reasoning. We call reasoning *hybrid* if part of the reasoning is done using skeptical reasoning and part of the reasoning is done using credulous reasoning. Thus, we may want to know whether from a program P we can infer $\exists X(p(X) \wedge q(X))$ where we want only those instantiations t of X such that P strongly entails $p(t)$ but weakly entails $q(t)$.

We develop below a language for expressing such hybrid queries and a formalism for performing hybrid reasoning.

By an *annotated* literal ([BS89]) we mean an expression of the form $l : S$ where l is a literal and S is a non-empty subset of $\mathcal{V} = \{T, CT, CF, F\}$. We stipulate that $l : \emptyset$ is not a well-formed expression of our language. In any interpretation $\mathcal{I}$, $l : S$ evaluates to T if and only if $\mathcal{I}'(l) \in S$ and otherwise $l : S$ evaluates to

F. Thus annotated literals can have only the classical truth values. A program P entails $l : S$ if and if for all canonical models $\mathcal{I}'$ of P, $\mathcal{I}'(l) \in S$.

Since annotated literals have only the classical truth values, an annotated literal $l : S$ cannot be weakly entailed. However an annotated literal $l : \{CT, T\}$ can be entailed by a program P if and only if l is weakly entailed by P.

A query of the form $\exists X(p(X) \wedge q(X))$, where we want only those instantiations t of X such that the program strongly entails $p(t)$ but weakly entails $q(t)$, can be expressed as

$$\exists X(p(X) : \{T\} \ \wedge \ q(X) : \{CT, T\})$$

Thus our framework provides us a way to express hybrid queries and to engage in hybrid reasoning.

5.6 Discussion

We contrast **C4** as a semantics of normal logic programs with the stable model semantics and the well-founded semantics. As compared to the stable model semantics, **C4** provides at least one intended model for any normal logic program. Thus using **C4** it becomes possible to draw reasonable inferences from any normal logic program. Although one can make a case that some programs cannot describe the intended meaning of any reasoner and thus they should not have any meaning, in this work we take the position that it should be possible to assign at least one "reasonable" model to any logic program. This is a highly desirable feature in the context of information integration where information is drawn from different sources. In this context there is no one reasoner whose intended meaning is being expressed by the program or the pool of information. But it is still highly desirable that one should be able to reason in terms of information drawn from different sources regardless of what is contained in this pool.

A related problem with the stable model semantics is the so-called "relevance problem" ([Dix95]). Let P be a program that has at least one stable model. Assume that $q \notin Atoms(P)$. In this sense q is not "relevant" to P. Then $P \cup \{q \leftarrow$ **not** $q\}$ has no stable models. That is, the addition of a rule irrelevant to P has robbed P of all its stable models. Since **C4** provides an intended model for any normal logic program, **C4** does not face this problem. Again, in the information integration context it is necessary to have a semantics that is resistant to the relevance problem.

It has been widely observed that the well-founded semantics is cautious compared to the stable model semantics. Thus, reasoning under the well-founded semantics forces the reasoner to be uniformly cautious regarding all information. One aspect of **C4** wrt the well-founded semantics is that for strong entailment it is exactly as cautious as the well-founded semantics but for weak entailment it is less cautious than the well-founded semantics. Thus using **C4** a reasoner can engage in both kinds of reasoning.

Another aspect of **C4** is that for certain types of programs it produces the intuitively correct result, whereas both the stable model semantics and the well-founded semantics do not. Consider the following program.

$$P = \{q \leftarrow \textbf{not } p;\ p \leftarrow \textbf{not } p\}$$

Understood procedurally the first rule says q is provable if **not** p is provable. Assuming negation as failure, this means q is provable if p is not provable. Both stable model semantics and the well-founded semantics agree in holding that p

should not be provable from this program. Thus, q should be provable. But q is not provable from P using the stable model semantics or the well-founded semantics. However, q is weakly entailed by P under **C4**.

5.7 Summary

In this chapter we investigate **C4** as a semantics of normal logic programs. The main research contributions of this chapter are as follows.

- We have proven that every definite logic program has a unique **C4** canonical model (Section 5.2).
- We have proven that every normal logic program has at least one **C4** canonical model (Section 5.2).
- We have proven that a normal logic program which has any two-valued stable models entails a literal with respect to the stable models of that program if, and only if, that program weakly entails that literal under **C4** (Section 5.3).
- We have proven that a normal logic program entails a literal with respect to the well founded semantics if, and only if, that program strongly entails that literal under **C4** (Section 5.4).
- We have devised a formalism to express hybrid conjunctive queries one part of which must be answered in terms of strong entailment and another part of which may be answered in terms of weak entailment (Section 5.5).

Chapter 6

Proof Procedure for Weak Entailment

6.1 Introduction

In this chapter we describe a proof procedure for determining whether a query consisting of conjunctions or disjunction of ground literals to a finite, ground normal logic program is *weakly* entailed by the program.

The proof procedure consists in making assumptions and computing in a bottom-up fashion a model of the program in which the assumptions hold true. In the first phase **not** *query* is among the assumptions. If it finds a model in which this assumption holds then it returns NO to the query. Otherwise, in the second phase the procedure attempts to find a model in which *query* is among the assumptions. If it finds a model in which this assumption holds then it returns YES to the query. Otherwise it returns the message that the program has no C-Stable models. We prove that this procedure is sound and complete with respect to weak entailment in the **C4** semantics. In Chapter 7 this proof procedure is modified to answer whether a query is *strongly* entailed by a normal logic program. In Chapter 8 this procedure is extended to answer queries to ground normal logic programs augmented with contestations.

In Section 6.2 we develop the formal apparatus needed to state the proof procedure. In Section 6.3 we state the algorithms of the proof procedure. In Section 6.4 we prove the soundness and completeness of this proof procedure with respect to weak entailment in the **C4** semantics. In Section 6.6 we analyze the complexity of the proof procedure and compare it to a related proof procedure by Chen and Warren.

6.2 Preliminaries

First we reproduce some definitions and results from Chapter 5.

Recall that a **C4** model $\mathcal{I}$ of a normal logic program LP is said to be C-stable iff $\mathcal{I}(R) = T$ for all rules $R \in LP$. A well-supported C-stable model of LP is always canonical. Also recall that if LP has any canonical C-stable models then all its canonical models are C-stable (Theorem 5.3.3 of Chapter 5). Theorem 5.3.2 of Chapter 5 says that LP has any stable models iff it has any C-stable models.

Assume that a query, L, has been posed to a ground normal logic program P. We define below the concept of rules relevant to answering an atomic query.

Definition 6.2.1 *Let P be a ground normal logic program and let q be a query to P. A rule $R \in P$ is relevant to answering a query q iff*

- *$q \in Atoms(R)$, or*
- *there is an atom p such that p is relevant to answering q and $p \in Atoms(R)$, where any atom p is relevant to answering q if and only if $p \in Atoms(R_i)$ where R_i is relevant answering q.*

Although this definition of rules relevant to a query is defined only in terms of atomic queries, it is still useful for the case where a query L is not atomic because given a query L to the program P, the proof procedure starts by adding the rule $query \leftarrow L$, where $query$ is an atom that does not belong to HB_P. However, if needed we can easily extend the above definition of rules relevant to an atomic query to the case of a non-atomic query. Let L be a query to P. Then the rules relevant to answering L are $\{R \in P \mid R$ is relevant to answering p where $p \in Atoms(L)\}$.

For the sake of simplicity we assume that all the rules of P are relevant to answering query L, otherwise we can easily compute the relevant part of P. We also assume that for any atom $a \in HB_P$, there is at most one rule with that a in the head. If P contains n, $n \geq 1$, rules with a in the head, the n rules can be combined into the one rule $a \leftarrow body_1 \ \vee \ldots \vee \ body_n$, where $body_1, \ldots, body_n$ are the bodies of each of the n rules which contain a in the head. When all the rules in P with the same atom in the head are replaced by such a combined rule, we say that P is in *disjunctive form*. In the rest of this chapter we shall assume that all programs are in disjunctive form. Furthermore, we assume that unit rules contain $true$ in the body, and the program is augmented by adding a rule $b \leftarrow \mathbf{not}\ true$ for each $b \in HB_P$ such that there is no rule in P with b as its head. Programs which are augmented thus are said to be in *augmented* form. When a program is in both disjunctive and augmented form, for each $a \in HB_P$, there is exactly one rule with a as its head.

As noted above, given a query L to the program P, the proof procedure starts by adding the rule $query \leftarrow L$, where $query$ is an atom that does not belong to HB_P. The proof procedure is based on the following strategy.

- First we determine whether there exists a well-supported C-stable model $\mathcal{I}$ of $P \cup \{query \leftarrow L\}$ such that $\mathcal{I}(\textbf{not}\ query) \geq CT$. If there is such a model, we return NO to the query and terminate; otherwise we go to the next step.

- Second we determine whether there exists a well-supported C-stable model $\mathcal{I}$ of $P \cup \{query \leftarrow L\}$ such that $\mathcal{I}(query) \geq CT$. If there is such a model, we return YES to the query and terminate; otherwise we return a message saying "This program has no C-Stable models" and terminate.

We know that every well-supported C-stable model of a program is a canonical model of the program. Thus, if there exists a well-supported C-stable model $\mathcal{I}$ of $P \cup \{query \leftarrow L\}$ such that $\mathcal{I}(\textbf{not}\ query) \geq CT$, then this must be a canonical model of $P \cup \{query \leftarrow L\}$ and hence $P \cup \{query \leftarrow L\}$ cannot entail $query$. But this means there must be a canonical model of P in which **not** $L \geq CT$, and hence P cannot entail L. This justifies returning NO at step 1. On the other hand if there is no such well-supported C-stable model and there is a well-supported C-stable model such that $\mathcal{I}(query) \geq CT$, then it must be a canonical model. Furthermore, in that case, every canonical model $\mathcal{J}$ of $P \cup \{query \leftarrow L\}$ must be C-stable (by Theorem 5.3.3 of Chapter 5) and must be such that $\mathcal{J}(query) \geq CT$. But then every canonical model of P must be such that L evaluates to at least CT in those models. Thus, P must entail L. This justifies returning YES at step 2. However, if there exists no well-supported C-stable model $\mathcal{I}$ of $P \cup \{query \leftarrow L\}$ such that $\mathcal{I}(\textbf{not}\ query) \geq CT$ and there exists no well-supported C-stable model $\mathcal{J}$ such that $\mathcal{J}(query) \geq CT$, then $P \cup \{query \leftarrow L\}$ has no well-supported C-stable models. But then P has no well-supported C-stable models. This justifies returning the message that "This program has no C-stable models."

The proof procedure consists in making assumptions and in terms of these inferring superscripted literals. These assumptions and inferred literals are used to reduce the input program and to infer more superscripted literals in terms of the reduced program. The formalism of superscripted literals and the rules for inferring such literals is described in the next subsection.

Superscripted literals

The superscript S of a literal l is an expression consisting of a disjunction of conjunctions of literals. The expression l^S denotes that assigning a certain truth value to l can be justified on the basis of assigning a certain truth value to S. S can be the empty expression.

Superscripted literals are inferred as follows. Let R be the only rule in the program P with a in the head. If R is $a \leftarrow true^S$ then a^S can be inferred from R. On the other hand if R is $a \leftarrow false^S$ then **not** a^S can be inferred from R. In either case this permits the reduction of P by deleting R from P once a^S or **not** a^S has been inferred.

Rules with $true^S$ or $false^S$ in the body can be obtained by the process of *matching* a literal in the body of the rule with an appropriate assumed or inferred literal. Matching is formally described in the definition below. A literal, whether an assumption or an inference, can be matched only with atoms in the body of a rule, never with the head of a rule. A positive inference or a negative inference or a negative assumption can be matched with any matching atom in the body of a rule. However, a positive assumption can be matched only with a negative literal (or, more precisely with an atom in a negative literal), but never with a positive literal, in the body of a rule.

Assumptions are typographically distinguished from inferences by underlining the assumptions.

Definition 6.2.2 *Let R be the normal logic rule*

$$a \leftarrow b_1, \ldots, b_n, \mathbf{not}\ c_1, \ldots, \mathbf{not}\ c_m$$

Matching is defined in terms of the following rules.

1. *A negative assumption $\underline{\mathbf{not}\ l}$ matches with $\mathbf{not}\ l \in body(R)$ resulting in $true^{\mathbf{not}\ l}$, which replaces $\mathbf{not}\ l$ in the body of R.*

2. *A negative assumption $\underline{\mathbf{not}\ l}$ matches with $l \in body(R)$ resulting in $false^{\mathbf{not}\ l}$, which replaces l in the body of R.*

3. *A positive assumption $\underline{l}$ matches with $\mathbf{not}\ l \in body(R)$ resulting in $false^{l}$ which replaces $\mathbf{not}\ l$ in the body of R.*

4. *A positive inference l^S matches with $l \in body(R)$ (or, $\mathbf{not}\ l \in body(R)$) resulting in $true^S$ (resp., $false^S$), which replaces l (resp., $\mathbf{not}\ l^S$) in the body of R.*

5. *A negative inference $\mathbf{not}\ l^S$ matches with $l \in body(R)$ (or, $\mathbf{not}\ l \in body(R)$) resulting in $false^S$ (resp., $true^S$), which replaces l (resp., $\mathbf{not}\ l^S$) in the body of R.*

Intuitively, a literal l in the body of a rule can be replaced by $true^S$ ($false^S$) by the operation of matching because under the assumption S the literal l evaluates to $true$ (resp., $false$). This is why we do not allow a positive assumption to match with a positive literal in the body. This ensures that a positive assumption is not justified in terms of itself. Thus, given the rule $p \leftarrow p$ and the assumption $\underline{p}$, if $\underline{p}$ were allowed to match with p in the body of $p \leftarrow p$, we would get $p \leftarrow true^p$.

From this we would be able to infer p^p. But since our model theory is in terms of well-supported models, we do not want positive information to be supported or justified in terms of itself. However, since the negation **not** is default negation, the inference of negative information does not require any justification. Hence it is all right for positive and negative assumptions to match with a negative literal in the body of a rule.

We understand **not** $true^S$ to evaluate to $false^S$ and **not** $false^S$ to evaluate to $true^S$. We give below rules for evaluating expressions consisting of the superscripted literals $true^S$ and $false^S$ conjoined with conjunction ($\wedge$) and disjunction ($\vee$).

$\wedge$	$true^{S_1}$	$false^{S_1}$
$true^{S_2}$	$true^{S_1 \wedge S_2}$	$false^{S_1}$
$false^{S_2}$	$false^{S_2}$	$false^{S_1 \wedge S_2}$

Table 6.1: Rule for evaluating conjunction of superscripted literals.

$\vee$	$true^{S_1}$	$false^{S_1}$
$true^{S_2}$	$true^{S_1 \vee S_2}$	$true^{S_2}$
$false^{S_2}$	$true^{S_1}$	$false^{S_1 \vee S_2}$

Table 6.2: Rule for evaluating disjunction of superscripted literals.

$true^{true \ \vee \ S}$ evaluates to $true^{true}$, which we shall simplify to $true$. A rule $a \leftarrow true^{true \ \vee \ S}$ can thus be simplified to $a \leftarrow \ true$ from which can be inferred a

without any superscripts. Similarly, $true^{true \wedge S}$ evaluates to $true^S$, $false^{false \wedge S}$ evaluates to $false$, and $false^{false \vee S}$ evaluates to $false^S$.

Example 6.2.1 *Let* $P = \{a \leftarrow b, \textbf{not } c;\ b \leftarrow \textbf{not } d;\ c \leftarrow \textbf{not } d;\ d \leftarrow \textbf{not } c\}$*. The assumption* $\underline{\textbf{not } c}$ *matches with* **not** c *in the first rule resulting in* $true^{\textbf{not } c}$*, which makes the first rule into* $a \leftarrow b, true^{\textbf{not } c}$*.* $\underline{\textbf{not } c}$ *also matches with* **not** c *in the fourth rule resulting in* $true^{\textbf{not } c}$*, which makes the fourth rule into* $d \leftarrow true^{\textbf{not } c}$*. The assumption* $\underline{b}$ *does not match with any atom in either of the rules.*

Since the fourth rule is the only rule with d *in the head, from* $d \leftarrow true^{\textbf{not } c}$ *we can infer* $d^{\textbf{not } c}$*. This in turn matches with* **not** d *in the third rule resulting in* $false^{\textbf{not } c}$*. Thus, the third rule becomes* $c \leftarrow false^{\textbf{not } c}$*. Since this is the only rule with c in the head, we can infer* **not** $c^{\textbf{not } c}$ *and the program can be reduced by eliminating the third rule.* $d^{\textbf{not } c}$ *also matches with* **not** d *in the body of the second rule resulting in* $false^{\textbf{not } c}$*, which makes the second rule into* $b \leftarrow false^{\textbf{not } c}$*. This permits the inference* **not** $b^{\textbf{not } c}$ *and the elimination of the second rule. This inferred literal matches with* b *in the body of the first rule resulting in* $false^{\textbf{not } c}$*, which turns the first rule into* $a \leftarrow false^{\textbf{not } c},\ true^{\textbf{not } c}$*. By the rules of evaluation described above this rule becomes* $a \leftarrow false^{\textbf{not } c}$ *Since this is the only rule with* a *in the head, using this rule* **not** $a^{\textbf{not } c}$ *can be inferred and the program can be further reduced by eliminating the first rule.*

Thus, starting with the assumption $\underline{\textbf{not } c}$ *we can infer*

$$\{d^{\textbf{not } c},\ \textbf{not } c^{\textbf{not } c},$$
$$\textbf{not } b^{\textbf{not } c},\ \textbf{not } a^{\textbf{not } c}\}$$

Analogous to the T_P operator of Van Emden and Kowalski ([vEK76]), defined in Chapter 3, we define a T^P operator in terms of assumptions and matching.

Definition 6.2.3 *Let P be a ground normal logic program. Let I be a set of literals, consisting of assumptions, superscripted literals, and the special atom true. Then,*

$T^P(I) = I \cup \{a^S \mid a \leftarrow body \in P$, *and matching literals in body with literals in I results in* $a \leftarrow true^S\} \cup \{\textbf{not } a^S \mid a \leftarrow body \in P$, *and matching literals in body with literals in I results in* $a \leftarrow false^S\}$

Note that although the T_P operator of Van Emden and Kowalski as applied to normal logic programs is not monotonically increasing, the T^P operator defined above is monotonically increasing because for any I, $I \subseteq T^P(I)$.

Let $\mathcal{H}_P = \{\underline{l} \mid l \in HB_P \cup \textbf{not } HB_P\} \cup \{l^S \mid l \in HB_P \cup \textbf{not } HB_P\}$, where S is any expression in DNF, possibly empty, consisting of literals in $HB_P \cup \textbf{not } HB_P \cup \{\epsilon\}\}$. $\mathcal{H}_P$ as defined above is the set of all possible assumptions and all possible inferences. The power set of this set forms a complete lattice under the $\subseteq$ ordering. Thus, if I is a member of the power set of $\mathcal{H}_P$, the iterations of $T^P(I)$ must have a least fixed point, denoted as $lfp(T^P(I))$.

We use these ideas in formalizing the query answering procedures described below. First, in Section 6.3 we describe a procedure for answering a query with respect to programs having well-supported C-stable models. For programs without a well-supported C-stable models the procedure returns a message to that effect. We prove the correctness of this procedure in Section 6.4. As indicated in the introductory section this procedure is based on making assumptions and reducing the input program in terms of these assumptions and the inferences from these assumptions. The procedure to be described in Section 6.3 contains no rule for choosing which assumption to make next. In Section 6.5 below we augment this procedure with a selection rule for choosing which assumption to make next.

6.3 Algorithms

Assume that a ground, positive query, L, has been posed to a ground program LP. We assume that LP is in the canonical form and all the rules of LP are relevant to answering the query. We add a new rule $query \leftarrow L$ to LP, where $query \notin HB_{LP}$. Let P be LP augmented with $query \leftarrow L$. The proof procedure is based on the following strategy.

- First we determine whether there exists a well-supported C-stable model $\mathcal{I}$ of P such that $\mathcal{I}(\textbf{not } query) \geq CT$. If there is such a model, we return NO to the query and terminate; otherwise we go to the next step.

- Second we determine whether there exists a well-supported C-stable model $\mathcal{I}$ of P such that $\mathcal{I}(query) \geq CT$. If there is such a model, we return YES to the query and terminate; otherwise we return a message saying "This program has no C-stable models" and terminate.

The procedure for finding a C-stable model of the normal logic program P in which $query \geq CT$ (or in which $\textbf{not } query \geq CT$) consists of two steps.

1. The procedure does a depth-first search through an implicit graph for a node satisfying certain properties of consistency, verifiedness, and stability (defined below) in which the input program has been reduced to the empty program by making a certain sequence of assumptions and a sequence of inferences in terms of these assumptions and a sequence of reductions of the input program in terms of these assumptions and inferences in the manner described in the previous section.

2. The assumptions and inferences in step 1 are then transformed into a **C4** model using the procedure $Trans$, which is described below.

In the first step the procedure searches through an implicit graph. The nodes of the graph consist of tuples of the form $\langle P', A, Inf, H\rangle$ where P' is a subset of the set of normal logic rules that can be formed out of the Herbrand base of the input program P; A is a set of literals which have been so far assumed; Inf is the set of literals that have so far been inferred; and H is the set of literals that are assumable at this point. The starting node in generating the graph consists of P, the input program, as P'; $\{true\}$ as A; $\emptyset$ as Inf; and, $HB_P \cup \textbf{not } HB_P$ as H.

We define an operator Γ on a node which is used to generate the children of that node in the graph. We need the following definition to define the Γ operator.

Definition 6.3.1 *Given a set of superscripted literals* $S = \{l_1^{s_1}, \ldots, l_n^{s_n}\}$, $Atoms(S) = \{Atom(l_1), \ldots, Atom(l_n)\}$, *where* $Atom(a) = a$ *and* $Atom(\textbf{not } a) = a$.

Definition 6.3.2 *Let* $N = \langle P, A, Inf, H\rangle$. *Then* $\Gamma(N) = \langle P', A, Inf', H'\rangle$ *where* $Inf' = lfp(T^P(A \cup Inf)) - A$, $H' = H - Atoms(Inf') - \textbf{not } Atoms(Inf')$, *and* $P' = P - \{R \in P | head(R)^S \in (Inf' - Inf) \text{ or } \textbf{not } head(R)^S \in (Inf' - Inf)\}$

We define below the descendants of a node N using the projection operator Π. If T is a tuple then $\Pi_i(T)$ returns the i^{th} member of the tuple. We call $\Pi_1(N)$ the program part of N, $\Pi_2(N)$ the assumption part of N, $\Pi_3(N)$ the inference part of N, and $\Pi_4(N)$ the assumables part of N.

Definition 6.3.3 *Descendants(N)* $=$

$$\begin{cases} \Gamma(N) & \text{if } \Pi_1(\Gamma(N)) = \emptyset \\ \{\langle \Pi_1(\Gamma(N)),\ (\Pi_2(\Gamma(N)) \cup \{\underline{l}\}), & \\ \quad \Pi_3(\Gamma(N)),\ (\Pi_4(\Gamma(N)) - \{l, \textbf{not } l\})\rangle \mid l \in \Pi_4(\Gamma(N))\} & \text{otherwise} \end{cases}$$

Example 6.3.1 *Let* $P = \{a \leftarrow \textbf{not}\ b;\ b \leftarrow \textbf{not}\ a; p \leftarrow \textbf{not}\ p \vee \textbf{not}\ b\}$. *Let* $SN = \langle P,\ \{\underline{\textbf{not}\ a}\},\ \emptyset,\ \{b,\ \textbf{not}\ b,\ p,\ \textbf{not}\ p\}\rangle$.

In this case

$$
\begin{aligned}
T^P(\{\underline{\textbf{not}\ a}\}) &= \{\underline{\textbf{not}\ a},\ b^{\textbf{not}\ a}\} \\
lfp(T^P(\{\underline{\textbf{not}\ a}\})) &= T^P(T^P(\{\underline{\textbf{not}\ a}\})) = \\
&\{\underline{\textbf{not}\ a},\ b^{\textbf{not}\ a},\ \textbf{not}\ a^{\textbf{not}\ a}\}.
\end{aligned}
$$

$\Gamma(SN) = \langle P',\ A', Inf',\ H'\rangle$ *where* $A' = \{\underline{\textbf{not}\ a}\}$ *and*

$$
\begin{aligned}
P' &= \{p \leftarrow \textbf{not}\ p \vee false^{\textbf{not}\ a}\} \\
Inf' &= \{b^{\textbf{not}\ a},\ \textbf{not}\ a^{\textbf{not}\ a}\} \\
H' &= \{p, \textbf{not}\ p\}
\end{aligned}
$$

SN has two descendants which consist of the nodes obtained by augmenting A' *in* $\Gamma(SN)$ *with one of p and* **not** *p and replacing* H' *in* $\Gamma(SN)$ *with the empty set.*

This example will be continued in Example 6.3.2 by computing the descendants after we have defined the following properties

Before we can compute the descendants of a node, we have to define the following properties.

Definition 6.3.4 *A literal* l_1 *is said to be dependent on a literal* l_2 *relative to a node N iff* $l_1^S \in (\Pi_2(N)\ \cup\ \Pi_3(N))$ *and* $S \models l_2$, *that is, if* l_2 *is a member of every disjunct of S.*

Definition 6.3.5 *A node N is said to be inconsistent iff there exists two literals* l^{S_1}, $\textbf{not}\ l^{S^2} \in \Pi_2(N)\ \cup \Pi_3(N)$ *such that neither literal is dependent on the other.*

Definition 6.3.6 *A node N is said to be nonstable iff there exists a literal* $l^S \in \Pi_3(N)$, *the inferred part of N, such that l is dependent on* **not** *l or there exists a* *literal* **not** $l^S \in \Pi_3(N)$ *such that* **not** l *is dependent on* l. *Otherwise a node is said to be stable.*

Definition 6.3.7 *A positive assumption* $\underline{a}$ *is said to be verified relative to a node* N *iff there exists a literal* $a^S \in \Pi_3(N)$. *A node* N *is said to be verified iff all the positive assumptions in* $\Pi_2(N)$ *are verified relative to* N. *An assumption* $\underline{a}$ *is said to be unverifiable in a node* N *if* **not** $a^S \in (\Pi_2(N) \ \cup \ \Pi_3(N))$.

Example 6.3.2 *Let* P *and* SN *be as in Example 6.3.1 above. A descendant of* SN *in the implicit graph of* P *can be obtained by choosing* $\underline{p}$ *as the next assumption. Let* N_1 *be this node.* $N_1 = \langle P', \ A', Inf', \ H' \rangle$ *where* $H' = \emptyset$ *and*

$$\begin{aligned} P' &= \{p \leftarrow \textbf{not } p \vee false^{\textbf{not } a}\} \\ A' &= \{\underline{\textbf{not } a}, \ \underline{p}\} \\ Inf' &= \{b^{\textbf{not } a}, \ \textbf{not } a^{\textbf{not } a}\} \end{aligned}$$

It is easily seen that $\Gamma(N_1)$ *contains* **not** $p^{p \wedge \textbf{ not } a}$ *and thus the assumption* $\underline{p}$ *is unverifiable relative to* $\Gamma(N_1)$.

A second descendant of SN *is the node obtained by making* $\underline{\textbf{not } p}$ *as the next assumption instead of* $\underline{p}$. *Let* N_2 *be this node.* $N_2 = \langle P', \ A', Inf', \ H' \rangle$ *where* $H' = \emptyset$ *and*

$$\begin{aligned} P' &= \{p \leftarrow \textbf{not } p \vee false^{\textbf{not } a}\} \\ A' &= \{\underline{\textbf{not } a}, \ \underline{\textbf{not } p}\} \\ Inf' &= \{b^{\textbf{not } a}, \ \textbf{not } a^{\textbf{not } a}\} \end{aligned}$$

It is easily seen that $\Gamma(N_2)$ *contains* $p^{\textbf{not } p}$. *Thus,* $\Gamma(N_2)$ *is unstable.*

It is easy to see that the leaf nodes of the graph are nodes in which the program part of the node, i.e., $\Pi_1(N) = \emptyset$. To determine whether there exists a canonical

C-stable model of a normal logic program P in which **not** $query \geq CT$, the algorithm searches for a stable, consistent, verified leaf node N which can be reached from the starting node $\langle P, \{\underline{\textbf{not } query}\}, \emptyset, HB_P \ \cup \textbf{not } HB_P\rangle$ such that **not** $query^S \in \Pi_3(N)$, for some, possibly empty, superscript S. We adopt a similar strategy to determine whether there exists a canonical C-stable model of P in which $query \geq CT$. In case P does not have a canonical C-stable model in which $query \geq CT$ and does not have a canonical C-stable model in which **not** $query \geq CT$, we can conclude that P does not have a canonical C-stable model. In this case the algorithm returns a message to that effect. The following algorithm implements this strategy.

Main(LP, L)

1. $P \leftarrow LP \cup \{query \leftarrow L\}$

1. If *MasterStable*(P, **not** $query$) $\neq$ nil then Return NO

2. else if *MasterStable*(P, $query$) $\neq$ nil then Return YES

3. else Return "Program has no canonical C-Stable models"

In step 2 *MasterStable*(P, **not** $query$) is called to determine whether starting from the node

$$\langle P, \{\underline{\textbf{not } query}\}, \emptyset, ((HB_P \ \cup \ \textbf{not } HB_P) - \{query, \textbf{not } query\})\rangle$$

a stable, consistent, verified, leaf node N can be reached in which **not** $query \in (\Pi_2(N) \ \cup \Pi_3(N))$. If such a node cannot be reached, *MasterStable* returns *nil* otherwise it returns the node. Thus, if such a node can be reached this means there exists a canonical model $\mathcal{I}$ of P such that $\mathcal{I}(\textbf{not}\ query) \geq CT$. Similarly, in step 3 *MasterStable* is invoked to determine whether there exists a canonical C-stable model $\mathcal{J}$ of P such that $\mathcal{J}(query) \geq CT$. If *MasterStable* returns a value other than *nil* then such a model exists and *Main* returns YES to the query and terminates. Otherwise the program contains no C-stable models since in any model $\mathcal{I}$, for any literal l, either $\mathcal{I}(l) \geq CT$ or $\mathcal{I}(\textbf{not}\ l) \geq CT$. Thus, in that case *Main* returns the message that the program has no canonical C-stable models.

The algorithm *MasterStable* creates the starting node using *CreateNode* and invokes Proc, which does all the real work.

MasterStable$(P,\ lit)$

$SN \leftarrow$ *CreateNode*$(P,\ lit)$
Parent$(SN) \leftarrow$ nil
$Proc(SN)$

MasterStable creates a node SN which has *lit* as the starting assumption by invoking *CreateNode*$(P,\ lit)$ which returns

$$\langle \Pi_1(\Gamma(N_0)),\ \{true,\ lit\},\ \Pi_3(\Gamma(N_0)),\ \Pi_4(\Gamma(N_0)) - \{lit, \textbf{not}\ lit\})\rangle$$

where N_0 is the node $\langle P,\ \{true\},\ \emptyset,\ (HB_P\ \cup\ \textbf{not}\ HB_P)\rangle$.

Given a node N, *Proc* determines whether starting with N a consistent, verified and stable leaf node can be reached. If there is such a leaf node, *Proc* returns the leaf node; otherwise *Proc* returns *nil*. *Proc* does a depth-first search for such a leaf node by making recursive calls to itself.

Proc(N)

1. If $\Gamma(N)$ is unstable or inconsistent or unverifiable or $(\Pi_1(\Gamma(N)) \neq \emptyset$
and $(\Pi_4(\Gamma(N)) = \emptyset$ or N has no unvisited descendants))
then if Parent(N) = nil then RETURN nil else *Proc*(Parent(N))

2. else if $\Pi_1(\Gamma(N)) = \emptyset$ then RETURN $\Gamma(N)$

   3. else
   4.**begin**
   5. Create unvisited descendant N'
   6. $Status(N') \leftarrow \ visited$
   7. $Parent(N') \leftarrow N$
   8.$Proc(N')$
   9. **end**

Step 1 lists the conditions under which *Proc*(N) backtracks to Parent(N) if N has a parent, otherwise *Proc* cannot backtrack and returns *nil* indicating that starting with the node N it cannot reach a consistent, stable, and verified leaf node. *Proc* backtracks if the result of making all possible inferences $(\Gamma(N))$ using the assumptions and inferences of N leads to an inconsistent, unstable or unverifiable state. It also backtracks if a leaf node is not reached $(\Pi_1(\Gamma(N)) \neq \emptyset)$, but the current node has no children because there are no further assumptions to make $(\Pi_4(\Gamma(N)) = \emptyset)$ or all of the current node's children have been previously visited and found to lead to deadends. If *Proc* does not backtrack or terminate in step 1, then this means either $\Pi_1(\Gamma(N))$ is empty or $\Pi_1(\Gamma(N))$ is not empty and $\Pi_4(\Gamma(N))$ is not empty and N has some unvisited descendants. If $\Pi_1(\Gamma(N))$ is empty then *Proc* has reached a desirable leaf node and it returns $\Pi_1(\Gamma(N))$ and terminates at step 2. Otherwise in step 5 to step 7 it creates and initializes N', a descendant of N, and at step 8 recursively invokes *Proc* with N'.

Recall that if $\Gamma(N)$ is not a leaf node then the descendant of N is

$$\langle \Pi_1(\Gamma(N)),\ (\Pi_2(\Gamma(N)) \cup \{l\}),\ \Pi_3(\Gamma(N)),\ (\Pi_4(\Gamma(N)) - \{l, \mathbf{not}\ l\})\rangle$$

where $l \in \Pi_4(\Gamma(N))$. Thus, essentially the descendant of N is $\Gamma(N)$ with its assumption part augmented with the assumption $\underline{l}$. We assume that the algorithm has some way, not specified here, for keeping track of which nodes have so far been visited. This might, for instance, be a global list which is updated when the status of a node is marked as *visited* and which is passed to each recursive call of *Proc*. We also assume that such a list is stored in some data structure, such as a binary search tree or a heap, which allows for an efficient search for whether a node has already been visited.

In *Proc* as specified above we regard every member of $\Pi_4(\Gamma(N))$ as suitable for generating a descendant of N as any other member. However, in Section 6.5 below we introduce a selection rule which makes only a small subset of $\Pi_4(\Gamma(N))$ suitable for generating a descendant of N.

Example 6.3.3 *As in Example 6.3.1 and Example 6.3.2, let the input program be* $LP = \{a \leftarrow \mathbf{not}\ b;\ b \leftarrow \mathbf{not}\ a; p \leftarrow \mathbf{not}\ p \vee \mathbf{not}\ b\}$. *Let the query be* a.

Main(LP, a) in the first step creates the program P by augmenting LP with rule query $\leftarrow a$. *Then it invokes MasterStable(P,* **not** *query) which creates the starting node SN, which is the node*

$$\langle P,\ \{\underline{\mathbf{not}\ query}\},\ \emptyset,\ \{a,\ \mathbf{not}\ a,\ b,\ \mathbf{not}\ b,\ p,\ \mathbf{not}\ p\}\rangle$$

MasterStable then invokes Proc(SN), which computes $\Gamma(SN)$. *Since* $\Gamma(SN)$ *is* *consistent, stable and verifiable, Proc creates an unvisited descendant of SN by selecting an unchosen assumption from* $\Pi_4(\Gamma(SN))$.

Let us suppose that **not** a *is chosen as the next assumption. This results in the node* $N_0 = \langle P,\ \{\underline{\textbf{not}\ query}, \underline{\textbf{not}\ a}\},\ \emptyset,\ \{\ b,\ \textbf{not}\ b,\ p,\ \textbf{not}\ p\}\rangle$. *Proc then makes a recursive call to itself with* N_0 *as the input node. Proc next computes* $\Gamma(N_0) = \langle P',\ A', Inf',\ H'\rangle$ *where*

$$P' = \{p \leftarrow \textbf{not}\ p \vee false^{\textbf{not}\ a}\}$$

$$A' = \{\underline{\textbf{not}\ query},\ \underline{\textbf{not}\ a}\}$$

$$Inf' = \{\textbf{not}\ query^{\textbf{not}\ a},\ b^{\textbf{not}\ a},\ \textbf{not}\ a^{\textbf{not}\ a}\}$$

$$H' = \{p, \textbf{not}\ p\}$$

It can be easily seen that $\Gamma(N_0)$ *is consistent, stable, and not unverifiable. So again Proc creates an unvisited descendant* N_1 *of* N_0 *by selecting an unchosen assumption from* $\Pi_4(\Gamma(N_0)) = \{p, \textbf{not}\ p\}$.

Let us suppose that p *is chosen as the next assumption. Except for the occurrence of the new literal* **not** *query in the assumption part, the node* N_1 *is essentially the node* N_1 *of Example 6.3.2. In that example we saw that* $\Gamma(N_1)$ *is unverifiable and this holds in the current example as well. So Proc backtracks to* $N_)$. *The only unvisited descendant of* N_0 *is the node* N_2 *obtained by choosing* **not** p *as the next assumption instead of p which is essentially the node* N_2 *of Example 6.3.2. In that example we saw that* $\Gamma(N_2)$ *is unstable, and this holds in our current example too. So Proc backtracks all the way to* SN.

Proc might next create the node N_3 *obtained by adding the assumption* a *to* $\Gamma(SN)$. *It can be easily seen that* $\Gamma(N_3)$ *is an inconsistent node containing the assumption* **not** *query and the inference* $query^a$. *Thus Proc backtracks to* SN *and might next create the node* N_4 *obtained by adding the assumption* **not** b *to* $\Gamma(SN)$.

In this case too it can be easily seen that $\Gamma(N_4)$ *is inconsistent for the same reasons as* $\Gamma(N_3)$.

At this point the only unvisited descendant of SN *is the node* N_5 *obtained by adding the assumption* b *to* $\Gamma(SN)$. $\Gamma(N_5)$ *is the node*

$$\begin{aligned} P' &= \quad \{p \leftarrow \mathbf{not}\ p \vee false^b\} \\ A' &= \quad \{\underline{\mathbf{not}\ query},\ \underline{b}\} \\ Inf' &= \quad \{\mathbf{not}\ query^b,\ b^b,\ \mathbf{not}\ a^b\} \\ H' &= \quad \{p, \mathbf{not}\ p\} \end{aligned}$$

$\Gamma(N_5)$ *is consistent, stable, and not unverifiable. Proc can expand it by adding either the assumption* p *or the assumption* **not** p. *The former option leads to the node* N_6*, similar to* N_1*, which for the same reasons as* N_1 *results in an unverifiable node; the latter option leads to the node* N_7*, similar to* N_2*, which for the same reasons as* N_2 *results in an unstable state. So, after visiting both these nodes, Proc backtracks to* SN*. Since* SN *has no unvisited descendants and since Parent(SN) is nil, Proc returns nil and thus MasterStable(P,* **not** *query) returns nil.*

Main(P, a) next invokes MasterStable(P, query) which creates the starting node SN which in this case is $\langle P,\ \{\underline{query}\},\ \emptyset,\ \{a,\ \mathbf{not}\ a,\ b,\ \mathbf{not}\ b,\ p,\ \mathbf{not}\ p\}\rangle$.

Since $\Gamma(SN)$ *is verified, stable and consistent Proc next creates a descendant of* SN*. Let us suppose it creates the node* N_0 *by adding the assumption* a *to* $\Gamma(SN)$. *Proc next computes* $\Gamma(N_0) = \langle P',\ A', Inf',\ H'\rangle$ *where* $A' = \{\underline{a}\}$ *and*

$$\begin{aligned} P' &= \quad \{p \leftarrow \mathbf{not}\ p \vee true^a\} \\ A' &= \quad \{\underline{query},\ \underline{a}\} \\ Inf' &= \quad \{\mathbf{not}\ b^a,\ a^a,\ query^a\} \\ H' &= \quad \{p, \mathbf{not}\ p\} \end{aligned}$$

Since this is consistent, stable and not unverifiable, Proc creates a descendant of it. N_0 has two descendants which consist of the node N_1 obtained by augmenting A' in $\Gamma(N_0)$ with p and the node N_2 obtained by augmenting A' in $\Gamma(N_0)$ with **not** *p. In both these nodes, H' is the empty set.*

Suppose Proc next visits N_2. In this case $\Gamma(N_2)$ is the node

$$\langle \emptyset,\ \{\underline{query},\ \underline{a},\ \textbf{not}\ p\},\ \{\textbf{not}\ b^a,\ a^a,\ query^a,\ p^{\textbf{not}\ p\ \vee\ a}\},\ \emptyset \rangle$$

This is an inconsistent node because $\underline{\textbf{not}\ p} \in \Pi_2(\Gamma(N_2))$ and $p^{\textbf{not}\ p\ \vee\ a} \in \Pi_3(\Gamma(N_2))$. Note that in this case p does not depend on **not** *p, and thus is not an unstable node, because p can also be generated by assuming $\underline{a}$.*

Thus Proc now backtracks to N_0 which next generates N_1. $\Gamma(N_1)$ is the node

$$\langle \emptyset,\ \{\underline{query},\ \underline{a},\ \underline{p}\},\ \{\textbf{not}\ b^a,\ a^a,\ query^a,\ p^a\},\ \emptyset \rangle$$

This is a consistent, verified and stable node and the program part of it is empty. Hence Proc returns this node and thus MasterStable(P, a) returns this node and hence Main returns YES to the query.

Given a leaf node N, $Trans(N)$ transforms it into a model of the original program P.

$Trans(N)$

1. $\mathcal{I} \leftarrow \emptyset$
2. $Inf \leftarrow \Pi_3(N)$
3. $Assp \leftarrow \Pi_2(N)$
4. For each positive inference $a \in \Pi_3(N)$ with an empty superscript,

begin

$\mathcal{I} \leftarrow (\mathcal{I} \cup \{a \mapsto T\})$

$Inf \leftarrow Inf - \{a^S\}$, where S is any superscript including the empty superscript

end

5. For each negative inference **not** $a \in \Pi_3(N)$ with an empty superscript,

begin

$\mathcal{I} \leftarrow (\mathcal{I} \cup \{a \mapsto F\})$

$Inf \leftarrow Inf - \{\textbf{not } a^S\}$, where S is any superscript including the empty superscript

end

6. While Inf contains any literal l^S such that $\mathcal{I}(S)$ has a value, do

begin while

Choose an $l^S \in Inf$ such that $\mathcal{I}(S)$ has a value

If l is the atom a then $\mathcal{I} \leftarrow (\mathcal{I} \cup \{a \mapsto \mathcal{I}(S)\})$

else if l is the negative literal **not** a then $\mathcal{I} \leftarrow (\mathcal{I} \cup \{a \mapsto (1 - \mathcal{I}(S))\})$

Delete l^S from Inf

end while

7. $Assp \leftarrow Assp - \{\underline{a}, \underline{\textbf{not } a} \mid \mathcal{I}(a) \text{ is defined}\}$

8. For each positive assumption $\underline{a} \in Assp$,

$\mathcal{I} \leftarrow (\mathcal{I} \cup \{a \mapsto CT\})$

9. For each negative assumption $\underline{\textbf{not } a} \in Assp$,

$\mathcal{I} \leftarrow (\mathcal{I} \cup \{a \mapsto CF\})$

10. $Inf \leftarrow Inf - \{a^S, \textbf{not } a^S \in Inf \mid \underline{a} \in Assp \textit{ or } \underline{\textbf{not } a} \in Assp\})$

11. While Inf is not empty do

begin while

Choose an $l^S \in Inf$ such that $\mathcal{I}(S)$ has a value

If l is the atom a then $\mathcal{I} \leftarrow (\mathcal{I} \cup \{a \mapsto \mathcal{I}(S)\})$

else if l is the negative literal **not** a then $\mathcal{I} \leftarrow (\mathcal{I} \cup \{a \mapsto (1 - \mathcal{I}(S))\})$

Delete l^S from Inf

end while

$Trans(N)$ builds a model $\mathcal{I}$ of the program P by first assigning T to all inferred literals with empty superscripts (steps 4 and 5). In step 6 these values are propagated as far as possible. In step 7 those assumptions are removed from $Assp$ whose truth values have already been fixed and in steps 8 and 9 the value CT is assigned to all such assumptions. In step 11 the values of superscripts are assigned to the atoms that have not been assigned a value in $\mathcal{I}$ so far. Since a superscript consists of a disjunction of conjunctions of assumptions, clearly all members of $\Pi_3(N)$ will have a truth value in $\mathcal{I}$.

Example 6.3.4 *Let P be as in Example 6.3.3. As in Example 6.3.3 let $\Gamma(N_1)$ be the node*

$$\langle \emptyset,\ \{\underline{query},\ \underline{a},\ \underline{p}\},\ \{\mathbf{not}\ b^a,\ a^a,\ query^a,\ p^a\},\ \emptyset \rangle$$

*Trans converts this node into $\mathcal{I}$, a **C4** interpretation of P, in which $\mathcal{I}(a) = CT$, $\mathcal{I}(query) = CT$, and $\mathcal{I}(p) = CT$ (by Step 2 of Trans), and $\mathcal{I}(b) = CF$ (by Step 9 of Trans).*

6.4 Proofs

First, we show that if the implicit graph for a finite, ground normal logic program contains a consistent, stable, verified leaf node then *MasterStable* will reach it. The requirement that the program be ground and finite is to guarantee that *MasterStable* will terminate. Second, we show that the transformation of such a node is a canonical C-stable model of the input program. But we cannot assume that if the implicit graph does not contain a stable node therefore the program has no canonical C-stable models unless we can show that every canonical C-stable model of the program is represented by a node in the graph. So, third, we show that all canonical C-stable models of the program are represented by a stable, consistent and verified node in the implicit graph of the program.

Lemma 6.4.1 *If the implicit graph of a ground, finite, normal logic program contains a consistent, stable, verified leaf node then MasterStable will return that node.*

Proof: *MasterStable* does a depth-first search for a leaf node with the appropriate properties. Since the program is ground and finite, the implicit graph for the program contains only a finite number of nodes. But depth-first search is guaranteed to discover any node with any specified properties if there is such a node in a finite graph. ■

Lemma 6.4.2 *If the implicit graph for a normal logic program P contains a consistent, verified leaf node N then $Trans(N)$ is a well-supported model of P.*

Proof: Let N be a consistent, and verified leaf node in the graph for P.

First, we show $Trans(N)$ is a model of P. Assume by way of contradiction that $Trans(N)$ is not a model of P. So P must contain a rule

$$a \leftarrow \; body_1 \;\vee \cdots \vee\; body_m$$

such that

- Case 1: $Trans(N)(a) \;=\; F$ and $Trans(N)(body_1 \;\vee \cdots \vee\; body_m) > F$, or
- Case 2: $Trans(N)(a) \;=\; CT$ or CF and $Trans(N)(body_1 \vee \cdots \vee body_m) \;=\; T$.

Case 1: $Trans$ assigns F to a only if $\Pi_3(N)$ contains **not** a without any superscript or with a superscript S such that $Trans(N)(S) = F$. But this is possible only if each $body_i \in \{body_1, \ldots, body_m\}$ evaluates to $false$ without any superscript or has a superscript S_i such that $Trans(N)(S_i) = F$. In this case $Trans$ would assign F to at least one literal in each of $body_1, \ldots, body_m$. Thus, $Trans(N)(body_1 \lor \cdots \lor body_m) > F$ is not possible, and, hence, Case 1 is not possible.

Case 2: If $Trans(N)(body_1 \lor \cdots \lor body_m) = T$ then there must be an $i \in 1, \ldots, m$ such that $Trans(N)(body_i) = T$. So each literal $a_{i_j} \in body_i$ must be assigned T by $Trans(N)$. So each such literal must be in the inferential part of N without any superscripts or with a superscript which evaluates to T in $Trans(N)$. Hence the inferential part of N would also contain a without any superscript or with a superscript which evaluates to T in $Trans(N)$. Thus, $Trans$ would assign T to a. Hence Case 2 is also not possible.

Thus, $Trans(N)$ must be a model of P.

Next we show that $Trans(N)$ is a well-supported model of P. The well-founded ordering can be in terms of the first appearance of a positive literal in the inferential part of a node in the path from the starting node to the leaf node N. This ordering must be well-founded because the generation of the nodes and the inferred atoms in each node are by process of bottom-up inference which monotonically enlarges the inferential part of nodes. Furthermore, since the assignment of a truth value to any literal is not greater than the truth value assigned to its superscript, the truth value assigned to a literal must be supported. ∎

Lemma 6.4.3 *If the implicit graph for a normal logic program P contains a stable, consistent, verified leaf node N then $Trans(N)$ is a well-supported C-stable model* *of P.*

Proof: We have already shown that $Trans(N)$ is a well-supported model. Assume by way of contradiction that $Trans(N)$ is not C-stable. So there must be at least one rule

$$R \ = \ a \leftarrow body_1 \ \vee \cdots \vee \ body_n$$

such that $Trans(N)(R) < T$. Given that $Trans(N)$ is a model of P, as proved in the previous lemma, this means that $Trans(N)(a) = CF$ and $Trans(N)(body(R)) = CT$.

Since $Trans(N)$ assigns CF to a, **not** a must be in the assumption or inferential part of N. But since $body(R)$ evaluates to CT in $Trans(N)$, a disjunct in $body(R)$ must evaluate to CT. Each literal in that disjunct must be in the assumption or inferential part of N. Hence, a^S, for some S, will also be in the inferential part of N. Thus, N is inconsistent unless $S \models$ **not** a. But, since N is stable $S \not\models$ **not** a. So N is inconsistent which contradicts the assumption that N is consistent.

Thus, $Trans(N)$ is a C-stable model of P. ∎

The proof procedure presupposes that if *MasterStable* cannot find a consistent, verified and stable leaf node N such that $query$ (or, **not** $query$) is in the assumption or inference part of N then the program contains no canonical C-stable model $\mathcal{I}$ such that $\mathcal{I}(query) \geq CT$ (resp., $\mathcal{I}($**not** $query) \geq CT$). Lemma 6.4.1 tells us that if the implicit graph for P contains a leaf node of that sort then *MasterStable* will find it. But we can have no assurance that if *MasterStable* does not find a leaf node of that sort then the program has no C-stable canonical model unless we can show that every C-stable canonical model is represented in the implicit graph. Ideally, we would like to prove that for each C-stable canonical model $\mathcal{I}$ of P there exists a leaf node in the implicit graph for P such that $Trans(N) = \mathcal{I}$. However, this claim would not be true of a model $\mathcal{I}$ which assigns only T or F to atoms because $Trans$ may assign CT or CF to atoms. Nevertheless, we show below in Lemma 6.4.4 that every C-stable canonical model is represented in the implicit graph in the sense of 'representation' defined below in Definition 9.4.1.

Definition 6.4.1 *Let $\mathcal{I}$ and $\mathcal{J}$ be two models of a logic program P. $\mathcal{I}$ is congruent with $\mathcal{J}$ iff $\mathcal{I}$ is identical with $\mathcal{J}$ except that every atom that is assigned T (F) in $\mathcal{J}$ is assigned T or CT (resp., F or CF) in $\mathcal{I}$.*

Representation is defined below.

Definition 6.4.2 *A model $\mathcal{I}$ of a normal logic program P is represented by a node N in the implicit graph for P if $Trans(N)$ is congruent with $\mathcal{I}$.*

A model $\mathcal{I}$ and a model $\mathcal{J}$ which is congruent with $\mathcal{I}$ will be indistinguishable in terms of weak entailment. That is, if $\mathcal{I}$ weakly entails a literal l if and only if $\mathcal{J}$ weakly entails that literal. This justifies our definition of representation above.

Lemma 6.4.4 *Let P be a normal logic program. For each well-supported C-stable model $\mathcal{I}$ of P there exists a leaf node N in the implicit graph for P such that $Trans(N)$ is congruent with $\mathcal{I}$ and thus N represents $\mathcal{I}$.*

Proof: Since the graph is implicit, a node N exists in the graph only if there is a path from the starting node, $\langle P, \{true\}, \emptyset, (HB_P \cup \textbf{not } HB_P)\rangle$, to N. Recall that in the path from the starting node to a leaf node each new node (other than the starting node) is generated by adding a new assumption to the result of applying the Γ operator to the previous node along with some housekeeping operations. Let N be any leaf node in the graph such that the path from the starting node to N

satisfies the following property: For any node N_i in the path its child in the path must be obtained by adding an assumption l such that $\mathcal{I}(l) \geq CT$ to the result of applying the Γ operator to N_i. That is, the path is generated using the strategy of making a new assumption l only if $\mathcal{I}(l) \geq CT$.

We show below that a leaf node N reached by this strategy

1. is a stable, consistent and verified node, and
2. is such that $Trans(N)$ is congruent with $\mathcal{I}$.

We prove that N is a stable, consistent, and verified node and that $Trans(N)$ is congruent with $\mathcal{I}$ by inductively proving that each node N_i in the path to N (including N) is consistent, stable and not unverifiable and inductively proving that, for any literal l, if $l^S \in \Pi_2(N_i) \cup \Pi_3(N_i)$ then $\mathcal{I}(l) \geq CT$. The induction is done in terms of the order in which the nodes appear in the path $N_0, \ldots, N_i, \ldots, N_n$, where N_0 is the starting node, $\langle P, \{true\}, \emptyset, (HB_P \cup \textbf{not } HB_P)\rangle$, and N_n is N.

Base Case: i = 0. Clearly, the starting node, N_0, is stable, consistent, and not unverifiable. Similarly, since $\Pi_2(N_0) \cup \Pi_3(N_0) = \{true\}$ it is trivially true that if a literal $l^S \in \Pi_2(N_0) \cup \Pi_3(N_0)$ then $\mathcal{I}(l) \geq CT$.

Inductive Case: Assume that the claim is true for all N_k such that $k < i$. To show that the claim is true for N_i.

First, we show that if a literal $\underline{l} \in \Pi_2(N_i) \cup \Pi_3(N_i)$ then $\mathcal{I}(l) \geq CT$. If $l^S \in \Pi_2(N_i)$ (i.e., if l is an assumption) then by the strategy for selecting assumptions it follows that $\mathcal{I}(l) \geq CT$. Suppose, therefore, that $l^S \in \Pi_3(N_i)$ (i.e., l^S is an inference). If $l^S \in \Pi_3(N_k)$, where $k < i$, then the claim is true by the inductive assumption. Suppose therefore that $l^S \notin \Pi_3(N_k)$, for any $k < i$. So l^S must occur in some iteration of the T^P operator as applied to $\Pi_2(N_{i-1}) \cup \Pi_3(N_{i-1})$. Either

$l^S = a^S$ or $l^S = \textbf{not}\ a^S$, for some atom a.

Assume that $l^S = a^S$. By the definition of the T^P operator it follows that if any atoms a such that $a \in T^P(\Pi_2(N_{i-1}) \cup \Pi_3(N_{i-1}))$ there is a rule $R = a \leftarrow body_1 \vee \ldots \vee body_m$ such that each member of $body_j$, $1 \leq j \leq m$, is in $\Pi_2(N_{i-1}) \cup \Pi_3(N_{i-1})$. Thus, by the inductive assumption $\mathcal{I}(body_j) \geq CT$. But since $\mathcal{I}$ is C-stable, it follows that a must be CT or T.

Assume instead that $l^S = \textbf{not}\ a^S$. It also follows from the definition of the T^P operator that for any negative literal **not** $a \in T^P(\Pi_2(N_{i-1}) \cup \Pi_3(N_{i-1}))$ there is a rule $R = a \leftarrow body_1 \vee \ldots \vee body_m$ such that for each $body_j$, $1 \leq j \leq m$, there exists a literal p_j in $body_j$ such that the negation of p_j is in $\Pi_2(N_{i-1}) \cup \Pi_3(N_{i-1})$. Hence, by the inductive assumption, each such p_j is at most CF in $\mathcal{I}$. Hence, each $body_j$ evaluates to at most CF in $\mathcal{I}$. So $\mathcal{I}(a) \leq CF$ since $\mathcal{I}$ is a well-supported model. Hence $\mathcal{I}(\textbf{not}\ a) \geq CT$. By a similar argument it is easy to see that the same remarks apply to any literal that belongs to any iteration of the T^P operator as applied to $\Pi_2(N_{i-1}) \cup \Pi_3(N_{i-1})$.

Hence, we have shown that if $l^S \in \Pi_2(N_i) \cup \Pi_3(N_i)$ then $\mathcal{I}(l) \geq CT$.

Second, we show that N_i is not unverifiable. Let a be a positive assumption in N_i. So $\mathcal{I}(a) \geq CT$. But then **not** a^S cannot be in $\Pi_3(N_i)$ otherwise, as we have shown above, $\mathcal{I}(\textbf{not}\ a) \geq CT$. But both a and **not** a cannot be CT or greater in $\mathcal{I}$. Thus, N_i is not unverifiable.

Third, we show that N_i is consistent. Let **not** a be a negative assumption in $\mathcal{I}$. So $\mathcal{I}(\textbf{not}\ a) \geq CT$. But then a^S cannot be in $\Pi_3(N_i)$ otherwise, as we have shown above, $\mathcal{I}(a) \geq CT$. But both a and **not** a cannot be CT or greater in $\mathcal{I}$. Thus, N_i is consistent.

Fourth, we show that N_i is stable. Suppose by way of contradiction that N_i is not stable. So there exists an $a^S \in \Pi_3(N_i)$ such that **not** a is in every disjunct of S. As shown above if $a^S \in \Pi_3(N_i)$ then $\mathcal{I}(a) \geq CT$. However, since **not** a is in S, **not** a must be an assumption and hence $\mathcal{I}(\textbf{not } a) \geq CT$. But this is a contradiction. Hence N_i must be stable.

This completes the inductive step. Thus, we have shown by induction that the leaf node N is stable, consistent and not unverifiable, and such that if $l^S \in \Pi_2(N) \cup \Pi_3(N)$ then $\mathcal{I}(l) \geq CT$. It remains to be shown that N is verified.

Since $\Pi_1(N)$ (the program part) is empty, it follows that for any atom $a \in HB_P$, either a or **not** a belongs in $\Pi_2(N) \cup \Pi_3(N)$. Since N is not unverifiable it follows that for any positive assumption a in N, **not** a^S cannot be in $\Pi_3(N)$. Hence a^S must be in $\Pi_3(N)$. Thus N must be verified.

Any literal $l \in \Pi_2(N)$ is assigned at least CT in $Trans(N)$ and must be CT or greater in $\mathcal{I}$. Any literal l such that $l^S \in \Pi_3(N)$ is assigned at least CT in $Trans(N)$ and, as we have shown above, must be CT or greater in $\mathcal{I}$. Furthermore, $Trans(N)$ assigns a truth value to every atom in HB_P since, for any atom a, either a or **not** a belongs in $\Pi_2(N) \cup \Pi_3(N)$. Thus, for any atom that is assigned T (F) by $\mathcal{I}$, $Trans(N)$ assigns it at least CT (resp., at most CF) and otherwise $Trans(N)$ and $\mathcal{I}$ are identical. Hence, $Trans(N)$ is congruent to $\mathcal{I}$.

■

Theorem 6.4.1 *If the implicit graph of P contains no stable consistent, verified leaf node N such that a specified literal $lit^S \in \Pi_2(N) \cup \Pi_3(N)$ then P has no canonical C-stable model $\mathcal{I}$ such that $\mathcal{I}(lit) \geq CT$.*

Proof: The converse of the theorem follows trivially from the previous lemma. ■

Lemma 6.4.5 *Let P be a normal logic program. Let $P' = P \cup \{query \leftarrow L\}$, where query is an atom not in HB_P and L is a conjunction or disjunction of literals such that $Atoms(L) \subseteq HB_P$. Then, for any C-stable model $\mathcal{I}$ of P, $\mathcal{I}' = \mathcal{I} \cup \{query \mapsto \mathcal{I}(L)\}$ is C-stable canonical model of P'.*

Proof: Let $\mathcal{I}$ be a C-stable canonical model of P. Thus all rules of P evaluate to T in P. So all such rules must also evaluate to T in $\mathcal{I}'$. Furthermore, L must evaluate to the same truth value in both $\mathcal{I}$ and $\mathcal{I}'$. Hence, the new rule in P', $query \leftarrow L$ must evaluate to T in $\mathcal{I}'$. Thus, $\mathcal{I}'$ must be a C-stable model of P'. ■

Now we are in a position to prove the correctness of the main algorithm.

Theorem 6.4.2 *Let LP be a ground, finite, normal logic program and let L be a query to the program. If Main(LP, L) returns "NO" then LP does not weakly entail L, if Main(LP, L) returns "YES" then LP weakly entails query, and if Main(LP, L) returns "Program has no canonical C-stable models" then LP has no canonical stable models.*

Proof: Assume that *Main*(LP, L) returns "NO." Then *MasterStable*(P, **not** query) must return a node N, where $P = LP \cup \{query \leftarrow L\}$. From Lemma 6.4.1 we know that if *MasterStable*(P, **not** query) returns a node N as stable, verified and consistent and such that **not** *query* is in $\Pi_2(N)$ or $\Pi_3(N)$ then N is such a node in the implicit graph. From Lemma 6.4.3 we know that $Trans(N)$ is a C-stable canonical model of P. By the nature of the Trans transformation $Trans(N)(\textbf{not}\ query) \geq CT$. But since $Trans(N)$ is a C-stable model of P and it follows that every rule with *query* in the head must be such that its body must evaluate to CF or F in $Trans(N)$. Hence, $Trans(N)(L) < CT$. Clearly, the model $Trans(N) - \{query \mapsto Trans(N)(query)\}$ must be a C-stable well-supported model of the original program LP. So LP cannot entail L.

Assume that *Main*(LP, L) returns "YES." Then *MasterStable*(P, **not** query) must return *nil* and *MasterStable*(P, query) must return a node N. From Lemma 6.4.1 we know that if *MasterStable*(P, **not** query) fails to discover a leaf node with the appropriate properties in the implicit graph then there is no such node in the graph. From Theorem 6.4.1 we know that then there is no canonical C-stable model $\mathcal{I}$ of P such that $\mathcal{I}(query) \geq CT$. Similarly we know that if *MasterStable*(P, query) returns a node N as stable, verified and consistent and such that $query$ is in $\Pi_2(N)$ or $\Pi_3(N)$ then N is such a node in the implicit graph. From Lemma 6.4.3 we know that $Trans(N)$ is a C-stable canonical model of P. By the nature of the Trans transformation $Trans(N)(query) \geq CT$. Furthermore, since $Trans(N)$ is a canonical C-stable model, so all canonical models of P must be C-stable. None of them are such that **not** $query \geq CT$. So all of them are such that $query \geq CT$. Since all such models are well-supported then in all such models L must also evaluate to CT or T. But then in no C-stable model of the original program LP, L can evaluate to CF or F (by Lemma 6.4.5). Thus, in every C-stable model, and, hence, in every canonical model, of LP, L must evaluate to CT or T. Thus, LP weakly entails L.

Assume that *Main*(LP, L) returns "Program has no canonical C-stable models." So *MasterStable*(P, **not** query) must return *nil* and *MasterStable*(P, query) must return *nil*. So from the earlier two parts of the proof we know that P has no canonical C-stable models in which **not** $query \geq CT$ and none in which $query \geq$

CT. But then LP has no canonical C-stable models in which **not** $L \geq CT$ and none in which $L \geq CT$ But in any interpretation of LP either **not** $L \geq CT$ or $L \geq CT$. So LP has no canonical C-stable models.

■

6.4.1 Computing Stable Models

The algorithm of the previous section can easily be adapted to compute all the stable models of a program. The algorithm $Proc$ needs to be modified to keep a list of the stable, consistent and verified nodes found so far and returning this list instead of just returning the first stable, consistent and verified node. This modified algorithm is invoked with the empty query.

$Proc2(N, StabList)$

1. If $\Gamma(N)$ is unstable or inconsistent or unverifiable or $(\Pi_1(\Gamma(N)) \neq \emptyset$ and $(\Pi_4(\Gamma(N)) = \emptyset$ or N has no unvisited descendents))
then if $Parent(N) \ = \ nil$ then RETURN $StabList$
    else $Proc2(Parent(N),\ StabList)$

2. else if $\Pi_1(\Gamma(N)) = \emptyset$ then
    2a. **begin**
    2b. $StabList \leftarrow (StabList\ \cup\ \{\Gamma(N)\})$
    2c. $Proc2(Parent(N),\ StabList)$
    2d. **end**

    3. else
        3a.**begin**

3b. Create unvisited descendent N'

3c. $Status(N') \leftarrow\ visited$

3d. $Parent(N') \leftarrow N$

3e. $Proc2(N',\ StabList)$

3f. **end**

Proc2 is invoked by a driver procedure, $Master2$, which is stated below.

$Master2(P)$

$SN \leftarrow CreateNode(P)$

$Parent(SN) \leftarrow nil$

$StabList \leftarrow \emptyset$

$Proc2(SN,\ StabList)$

6.5 Selection Rule

A descendent of a non-leaf node N is generated by first computing $\Gamma(N)$ and then adding an assumption from $\Pi_4(\Gamma(N))$ to $\Pi_2(\Gamma(N))$. *Proc* puts no restrictions on which assumption from $\Pi_4(\Gamma(N))$ is used to generate the descendent. Thus, an assumption irrelevant or contrary to generating the desired leaf node may be made in generating the next node. In this section we introduce a selection rule which puts more restrictions on which assumption is made next. This will help the improved version of *Proc*, called *ProcSel*, avoid generating many unhelpful nodes. We offer the selection rule as a possible aid to an implementor, but do not here prove its correctness.

Suppose that *Proc* is trying to find if starting with $\langle P,\ \{lit\},\ \emptyset,\ (HB_P \cup$

not $HB_P) - \{lit, \textbf{not } lit\}\rangle$ it can reach a consistent, verifiable and stable leaf node N. Hence, lit^S, for some S, must be in the inference part of N. Thus, we can regard the assumption lit as discharged by proving lit. Hence, in selecting the next node *ProcSel* can choose an assumption which will advance the task of discharging the assumption lit. This will be either the assumption of a negative literal in the body of the rule with lit in the head or the assumption of a negative literal which will help establish a positive literal in the body of such a rule. Thus, assumptions can be only of negative literals. But this new assumption itself needs to be discharged. So this becomes the new sub-task. So *ProcSel* next makes another new assumption which will help discharge the earlier assumption. And so on. Thus, *ProcSel* can be seen as using the following selection rule: In generating the next node make an assumption which can help with discharging the most recently made assumption that needs to be discharged.

It is clear that lit cannot be in the inference part of a node N unless the rule with lit in the head evaluates to true given the assumptions and inferences in N. Suppose for now that each rule in the program (in its canonical form) contains only one disjunct in the body. Thus, given the rule

$$lit \leftarrow b_1 \wedge \cdots \wedge b_n$$

we can discharge the assumption of lit by making true each of $b_1, \ldots, b_n$. We can regard $b_1 \wedge \cdots \wedge b_n$ as a goal list–these are the literals that must be true in the final node N. The next literal that needs to be established is the first goal in the goal list, i.e., b_1. If b_1 is a positive literal then we resolve b_1 against the rule with b_1 in the head and add its body to the goal list. Thus, if the program contains the rule

$$b_1 \leftarrow c_1, \ldots, c_m$$

then the new goal list becomes

$$c_1 \wedge \cdots c_m, b_2 \wedge \cdots b_n$$

If b_1 is the negative literal **not** a then the next assumption made is **not** a and the next node is generated using that assumption. The assumption **not** a can be discharged by making false the body of the rule with a in the head. Suppose we have the rule

$$a \leftarrow a_1 \wedge \cdots \wedge a_m$$

$a_1 \wedge \cdots \wedge a_m$ can be made false by making false any of the literals $a_1, \ldots, a_m$. This can be seen as making true the disjunctive goal **not** $a_1 \vee \cdots \vee$ **not** a_m. This is a disjunctive goal which can be made true by making any of its disjuncts true. We add this disjunctive goal in place of b_1 in the goal list. Thus, only negative literals are assumed, but goals may be positive or negative literals. We formalize these ideas below.

We define an *expression* recursively as follows. An *expression* is a literal or the conjunction or disjunction of expressions. If an expression is a conjunction (disjunction) then it is called a conjunctive (resp., disjunctive) expression and each conjunct (resp., disjunct) is called a sub-expression of that expression. An expression which is not a literal we call a non-literal expression. A goal list consists of a conjunction of expressions. A conjunctive (disjunctive) sub-expression of the goal list is called a conjunctive (resp., disjunctive) goal. The first literal in a conjunction (disjunction) of expressions is recursively defined as the first conjunct (resp., disjunct) if the first conjunct (resp., disjunct) is a literal, otherwise it is the first literal in the first conjunct (resp., disjunct). The first literal in the goal list is called the first goal in the goal list. The goal list can be thought of as stored in an appropriate data structure such as an expression tree. The leaves of such a tree are literals. Then the first goal in the goal list is the leaf of the left-most branch of the tree.

Suppose the starting node SN in the implicit graph is

$$\langle P,\ \{lit\},\ \emptyset,\ (HB_P \cup \textbf{not}\ HB_P) - \{lit,\ \textbf{not}\ lit\}\rangle$$

If lit is a positive literal then the program must contain the rule

$$lit \leftarrow body_1 \wedge \cdots \wedge body_n$$

If lit is the negative literal **not** a then the program must contain the rule

$$a \leftarrow body_1 \wedge \cdots \wedge body_n$$

Suppose each $body_i = b_{i_1} \wedge \cdots \wedge b_{i_n}$. Then if lit is a positive literal, then the starting goal list consists of

$$(b_{1_1} \wedge \cdots \wedge b_{1_n}) \vee \cdots \vee (b_{n_1} \wedge \cdots \wedge b_{n_n})$$

In this case the first goal in the goal list is b_{1_1}. However, it could happen that some of the literals in the goal list evaluate to true (or false) in $\Gamma(SN)$. If b_{i_j} evaluates to true in $\Gamma(SN)$ then we remove it from the goal list. If b_{m_n} evaluates to false in $\Gamma(SN)$ then we remove all of $b_{m_1}, \ldots, b_{m_n}$ (i.e., we remove $body_m$) from the goal list. The modified goal list is associated with SN as its goal list. Thus, it could happen that the first goal in the modified list may not be b_{1_1}.

If lit is **not** a then the starting goal list consists of

$$(\textbf{not}\ b_{1_1} \vee \cdots \vee \textbf{not}\ b_{1_n}) \wedge \cdots \wedge (\textbf{not}\ b_{n_1} \vee \cdots \vee \textbf{not}\ b_{n_n})$$

In this case the first goal in the goal list is **not** b_{1_1}. Again, this goal list is modified in terms of $\Gamma(SN)$.

Slightly abusing terminology, by the *resolvent* of lit with the rule $lit \leftarrow body_1 \vee \cdots \vee body_n$ we mean

$$body_1 \vee \cdots \vee body_n$$

If lit is the negative literal **not** a, then by the *negresolvent* of lit with the rule $lit \leftarrow body_1 \vee \cdots \vee body_n$ we mean

$$\textbf{not } body_1 \wedge \cdots \wedge \textbf{not } body_n$$

Each **not** $body_i$ can be simplified to **not** $body_{i_1} \vee \cdots \vee$ **not** $body_{i_n}$.

A goal in the goal list is regarded as solved in a node N_k if it evaluates to true relative to the assumptions and inferences in that node. If the goal is positive (negative) then it is also regarded as solved if the body of the rule with the goal (resp., negation of the goal) in the head evaluates to true (resp., false) relative to the assumptions in the goal list associated with N_k. The idea here is that we assume that N_k is in the path from the starting node to the final node and hence all the expressions in the goal list will evaluate to true in N and thus the goal will be true in the final node. A disjunctive goal is solved by solving a disjunct in the goal and a conjunctive goal is solved by solving each conjunct in the goal. If a goal is solved, the goal is removed from the goal list. Otherwise, a positive goal is removed from the goal list by replacing it in the goal list with sub-list of goals such that solving this sub-list of goals will solve the positive goal. The goal list is regarded as solved if it is empty.

A disjunctive goal is solved by assuming one of the negative disjuncts as the next goal in generating the next node. It can happen that there is no path from that node to a consistent, verified and stable node. In that case we have to backtrack and generate another as yet untried node by assuming another as yet unchosen negative disjunct in the disjunctive goal. A conjunctive goal is solved by assuming one by one as many of the negative conjuncts that need to be assumed to solve each conjunct. Similarly, it can happen that there is no path from the node generated by assuming a particular conjunct in the conjunctive goal. In that case the conjunctive goal is unsolvable and in that case we have to backtrack to the disjunctive choice point which led to this conjunctive goal and choose another negative disjunct in the disjunctive goal. It can also happen that assuming one conjunct in a conjunctive goal results in making another conjunct false. In that case too we have to similarly backtrack.

The procedure *ProcSel* given below implements this selection strategy.

ProcSel(N)

1. If $\Gamma(N)$ is unstable or inconsistent or unverifiable or $(\Pi_1(\Gamma(N)) \neq \emptyset$ and $(\Pi_4(\Gamma(N)) = \emptyset$ or $UnchosenAssumption(N)$ = nil)
then if Parent(N) = nil then RETURN nil else *ProcSel*(Parent(N))

2. else if $\Pi_1(\Gamma(N)) = \emptyset$ then RETURN $\Gamma(N)$

3. else
4.**begin**
5. Assumption $\leftarrow$ $UnchosenAssumption(N,\ GoalList(N))$
6. In GoalList(N) set the status of Assumption as *chosen*
7. $N' \leftarrow CreateDescendent(N, Assumption)$
8. $Status(N') \leftarrow\ visited$
9. $Parent(N') \leftarrow N$
10. Set GoalList(N') to the goal list that results from matching the literals in GoalList(N') with the literals in $\Pi_2(\Gamma(N'))\ \cup \Pi_3(\Gamma(N'))$

and removing from GoalList(N') any disjunctive expressions that become true and any conjunctive expressions that become false.

11. Set GoalList(N') to the goal list that results from adding as the first conjunct to GoalList(N') the negresolvent of Assumption with the appropriate rule in $\Pi_2(\Gamma(N'))$.

12. $ProcSel(N')$

13. **end**

ProcSel uses the procedure *UnchosenAssumption* which is stated below. *UnchosenAssumption*(N, *GoalList*) returns an assumption from the goal list associated with N if there is an unchosen assumption in the goal list, otherwise it returns nil unless the goal list is empty in which case it returns any unchosen literal from $\Pi_4(\Gamma(N))$. The use of a goal list is the main difference between *Proc* and *ProcSel*. As in the case of *Proc*,step 1 lists the conditions under which *ProcSel* backtracks. In case there are no reasons to backtrack and the program part of $\Gamma(N)$ is empty, the procedure returns $\Gamma(N)$. Otherwise it creates a new node N' using an unchosen assumption (steps 7-9). In step 6 it marks as chosen the occurrence of the chosen assumption in the goal list of N. Thus, if the algorithm backtracks it will not try that assumption again at that point. In steps 10 and 11, the goal list of N' is created and associated with N'. In step 12 *ProcSel* is recursively called with N'.

Procedure *UnchosenAssumption* tries to find an unchosen negative goal form the goal list as the next assumption to make. If the first goal in the goal list is an unchosen negative literal then it returns that literal (Step 2); if the first goal is a positive literal then it resolves the literal against the appropriate rule, replaces the first goal with the body of the rule in the goal list associated with the node N and recursively calls *UnchosenAssumption* with N and the goal list associated with N (Step 3). The goal list of N is modified in terms of the resolvent rather the parameter GoalList because if later *ProcSel* has to backtrack from a descendent of N to N, it is saved the task of making all the resolutions all over again. If the first goal is a negative literal that had been chosen earlier, then the procedure does not try that literal again and makes a recursive call to *UnchosenAssumption* with that literal removed from the GoalList (Step 4). The auxiliary function Tail(GoalList) returns GoalList with the first goal deleted.

$UnchosenAssumption(N,\ GoalList)$

1. If GoalList is empty go to step 5.

2. If first goal of GoalList is an unchosen negative literal l
then RETURN l.

3. If first goal of GoalList is a positive literal l then
   begin
   GoalList(N) $\leftarrow$ *Substitute*(Resolvent(l), GoalList(N))
   $UnchosenAssumption(N,\ GoalList(N))$
   end

4. else if first goal is a chosen negative literal
then $UnchosenAssumption(N,\ Tail(GoalList))$

5. If $\Pi_4(\Gamma(N))$ has an unchosen literal then RETURN any such unchosen literal else RETURN nil.

ProcSel is invoked by a driver procedure, *MasterSel*, which is stated below.

MasterSel(P, lit)

1. $SN \leftarrow CreateNode(P, lit)$

2. $Parent(SN) \leftarrow nil$

3. If *lit* is positive then set $GoalList(SN)$ to the resolvent of *lit* with the appropriate rule, else set $GoalList(SN)$ to the negresolvent of *lit* with the appropriate rule.

4. Set $GoalList(SN)$ to the goal list that results from matching the literals in GoalList(SN) with the literals in $\Pi_3(\Gamma(SN))$ and removing from $GoalList(SN)$ any disjunctive expressions that become true and any conjunctive expressions that become false.

5. $ProcSel(SN)$

Example 6.5.1 *Let* $P = \{a \leftarrow b, c;\ b \leftarrow \textbf{not}\ d, \textbf{not}\ a;\ c \leftarrow \textbf{not}\ e \vee b;$ $e \leftarrow \textbf{not}\ d; d \leftarrow \textbf{not}\ e\}$. *Let the query be a.*

Procedure Main invokes procedure MasterSel with program $P' = P \cup \{query \leftarrow a\}$ *and* **not** *query as the parameters. MasterSel creates the starting node* SN *with* **not** *query as the initial assumption and* $\{\textbf{not}\ a\}$ *as the initial goal list, and then invokes procedure ProcSel with* SN *as its parameter.*

ProcSel makes **not** *a as the Assumption and invokes CreateDescendent.* **not** *a is negresolved with* $a \leftarrow b, c$ *to produce* $\{\textbf{not}\ b \vee \textbf{not}\ c\}$ *as the goal list. The node* N_0 *is created with* $\{\textbf{not}\ query,\ \textbf{not}\ a\}$ *as the assumption part and* $\{\textbf{not}\ b \vee \textbf{not}\ c\}$ *as the goal list. Then procedure ProcSel is invoked recursively with* N_0 *as its parameter.*

ProcSel makes **not** *b as the Assumption and invokes CreateDescendent to cre-*

ate node N_1 *with* $\{\textbf{not}\ a,\ \textbf{not}\ b\}$ *as the assumption part. The goal list* $\{\textbf{not}\ b \vee \textbf{not}\ c\}$ *is solved in terms of the assumption* **not** b *and is empty (step 10 of ProcSel). At step 11* **not** b *is negresolved with the rule* $b \leftarrow \textbf{not}\ d,\ true^{\textbf{not}\ a}$ *to get* $\{d \vee false^{\textbf{not}\ a}\}$ *as the goal list, which can be simplified to* $\{d\}$. *ProcSel recursively calls ProcSel with* N_1 *as the parameter.*

ProcSel(N_1) invokes procedure UnchosenAssumption with $\{d\}$ *as the goal list. UnchosenAssumption resolves* d *against* $d \leftarrow \textbf{not}\ e$ *to make* $\{\textbf{not}\ e\}$ *as the goal list, and thus returns* **not** e *as the next assumption. The assumptions*

$$\{\textbf{not}\ query,\ \textbf{not}\ a,\ \textbf{not}\ b,\ \textbf{not}\ e\}$$

together result in the following inferences:

$$\{\textbf{not}\ query^{\textbf{not}\ a},\ d^{\textbf{not}\ e},\ \textbf{not}\ e^{\textbf{not}\ e},\ c^{\textbf{not}\ e},\ \textbf{not}\ b^{\textbf{not}\ e},\ \textbf{not}\ a^{\textbf{not}\ e}\}$$

The program part and the goal list is empty at this point. So ProcSel and MasterSel returns $\Gamma(N_1)$. *Hence, Main returns NO to the query.*

6.6 Discussion

The complexity of the proof procedure will be analyzed in terms of the operation of matching as the unit of computation. We will do a worst-case analysis of the number of matching operations performed as a function of the Herbrand base of the input program.

Assume that a query L has been posed to a program P. Let the cardinality of HB_P be n.

First, we analyze in the worst-case the number of matching operations that have to be performed in expanding any node. A node N in the implicit graph of a program P is expanded by computing $\Gamma(N)$, which is where all the matching operations take place. To compute $\Gamma(N)$ the procedure computes the least fix-point of the T^P operator on the program part of N. In the worst-case the program part of N contains a rule for each atom in HB_P. (Recall that the program is in disjunctive form.) Although each rule can in the worst-case contain at most factorial of $2n$ literals in its body, clearly there can be at most $2n$ *distinct* literals in the body of any rule since the cardinality of HB_P is n. We assume that when a literal is matched with an assumption or an inference, then all occurrences of that literal in the body of that rule are replaced in the body by the result. Thus, in computing the least fix-point of T^P the procedure needs to do at most $2n$ matching operations for each rule. Since there are at most n rules, at most $2n^2$ matching operations are required for computing the least fix-point of the T^P operator. This means at most $2n^2$ matching operations are required for expanding any node. Thus, it takes $O(n^2)$ operations in the worst case for expanding any node.

In the worst case there will be n nodes in each path from the starting node to a leaf node. That is, in the worst case each leaf node will contain n assumptions, either a or **not** a, for each $a \in HB_P$. Thus, we can count the number of leaf nodes in the worst case by adding the number of leaf nodes containing the assumption a and the number of leaf nodes containing the assumption **not** a, for any $a \in HB_P$, since every leaf node must contain either an atom or its negation as an assumption. Thus, we can enumerate all the leaf nodes containing an assumption l, by enumerating all the possible sets of assumptions containing l. To count the number of assumption sets containing a given assumption, l, imagine that we have put the atoms in HB_P in some ordering, say lexicographical ordering, with $Atom(l)$ as the first in the ordering. Then we can represent all such sets by a binary tree containing l as its root and the next atom in the ordering as its left child and the negation of that atom as its right child, and so on. Such a tree will contain 2^{n-1} paths, where the set of assumptions along a path represents an assumption set. Thus, there are 2^n assumption sets containing a given atom or its negation. Thus, in the worst case the implicit graph will contain 2^n leaf nodes.

By similar reasoning we can see that the graph will contain 2^{n-1} nodes containing $n-1$ assumptions, and so on. Thus, the total number of nodes in the graph can be expressed as

$$T(n) = 2^n \; + \; 2^{n-1} \; + \cdots + 2^0 \; = \; 2^{n+1} - 1 \; = \; O(2^n)$$

Clearly, $MasterStable$ expands each node only once. Thus in the worst case $MasterStable$ will expand $O(2^n)$ nodes. Hence in the worst case $MasterStable$ will perform $O(n^2 \times 2^n)$ matching operation. Hence, the complexity of $Main$ in terms of the number of matching operations performed is $O(n^2 \times 2^n)$.

This result matches well with well-known results. For instance, [MT91] and [MM93] have shown that determining whether an atom belongs to all the stable models of a program is a co-NP problem.

The proof procedure of this chapter is similar to the proof procedure described in [CW97]. They describe a procedure for finding all the stable models of a normal logic program by assuming literals step-by step and inferring other literals on the basis of the assumed literals and reducing the original program step-by-step in terms of the assumed literals and inferred literals. Their procedure is restricted to programs that have stable models, and for programs without any stable models their procedure returns the empty set. One difference between our procedure and the procedure in [CW97] is that we use superscripts to keep track of the assumptions on which an inference has been based. This feature will be seen to be very useful in Chapter 8 when we develop a proof procedure for normal logic programs augmented with contestations.

6.7 Summary

In this chapter we have devised a proof procedure for determining whether a query consisting of conjunctions or disjunction of ground literals to a finite, ground normal logic program which has at least one C-stable model is *weakly* entailed by the program. In case the program has no C-stable models the procedure terminates gracefully by sending a message to that effect. The main research contributions of this chapter are as follows.

- We have developed the formal apparatus and algorithms for computing a canonical model of a program in which a specified literal is true by making assumptions and inferring literals on the basis of these assumptions and the input program.
- We have devised a procedure which utilizes this apparatus and algorithms for determining whether a query is weakly entailed by the input program (Section 6.3).
- We have proven the soundness and completeness of this proof procedure (Section 6.4).
- We have modified this proof procedure to compute all the two-valued stable models of a finite and ground normal logic program (Section 6.3).

- We have provided a tool for optimizing the performance of the proof procedure in the form of a selection rule for determining which assumption to make next at a given stage of constructing a canonical model of the input program (Section 6.5).

- We have proven that the worst-case complexity of this procedure is $O(n^2 \times 2^n)$, where n is the cardinality of the Herbrand base of the program (Section 6.6).

Chapter 7

Proof Procedure for Strong Entailment

7.1 Introduction

In this chapter we extend the proof procedure for weak entailment to cover strong entailment. Since the set of strong entailments of a normal logic program has been shown to be equivalent to the well-founded semantics (Theorem 5.4.1 of Chapter 5), the resulting proof procedure will also be a proof procedure for the well-founded semantics. The procedure for strong entailment is restricted to queries, which can be a conjunction or disjunction of literals, to finite, ground normal logic programs.

In the case of weak entailment the difference between T and CT is not of any significance in the sense that a model which assigns CT to a literal just as much weakly entails that literal as a model which assigns T to it. In the same way the difference between CF and F is of no significance for weak entailment. But in the case of strong entailment these differences matter because an atom p is strongly entailed by a program P iff p is assigned T in all the canonical models of P and **not** p is strongly entailed by P iff p is assigned F in all the canonical models. For this reason, since in a well-supported model an atom which has no non-circular support must be assigned F, and not CF, we need to keep track of such atoms in the case of strong entailment. In our proof procedure for weak entailment we allowed a negative assumption to match with an atom in the body of a rule because we were indifferent to the difference between the atom in the head of that rule being false on the basis of an assumption and the atom being false because it had no non-circular support. And this was because for weak entailment the difference between assigning CF and assigning F to an atom was of no significance. But strictly speaking, given our definition of a well supported interpretation, an atom that is false on the basis of an assumption should be assigned CF if the assumption evaluates to CT in that interpretation, whereas an atom that is false because it has no non-circular support must be assigned F in any well-supported model. For strong entailment this difference is critical. Hence in our proof procedure for strong entailment we will *not* allow a negative assumption to match a positive atom so as to give the proof procedure a chance to discover whether the atom should be judged false because it has no non-circular support.

For weak entailment the difference between CF and CT is important. But for strong entailment whether a literal is assigned CF or CT in a model, it is equally not strongly entailed in that model. We will exploit this feature of strong entailment to simplify the proof procedure of the last chapter by eliminating the consistency and verifiedness checks in a manner to be explained in Section 7.3.

Since strong entailment has been proven to be equivalent to the well founded semantics, and since the well-founded semantics is defined for programs with no stable models, and, hence, for programs with no C-stable models, our proof procedure for strong entailment and for well-founded semantics must be designed to work for programs without any C-stable models. This is another respect in which the proof procedure will have to be different from the proof procedure for weak entailment.

As in Chapter 6, we assume that the input program is in the disjunctive and augmented form.

In Section 7.2 we redefine some of the formal apparatus of Chapter 6 to accommodate the above described differences between the procedure for weak entailment and the procedure for strong entailment. In Section 7.3 we describe the algorithms of the proof procedure for strong entailment, and in Section 7.4 we prove that this procedure is sound and complete with respect to strong entailment in **C4**. In Section 7.5 we discuss the worst-case complexity of this proof procedure. In Section 7.6 we summarize the main research contributions of this chapter.

7.2 Preliminaries

To accommodate the above described differences between weak entailment and strong entailment we need to redefine some of the apparatus developed for the proof procedure for weak entailment. In this section we accomplish this redefinition. In particular the rules for matching literals and inferring superscripted literals will be modified as well as the definition of the T^P operator, the Γ operator, and the definition of the descendants of a node.

As noted in the previous section, the new rules of matching, which will be specified below, will not allow a negative assumption to match a positive literal so as to give the proof procedure a chance to discover whether an atom should be judged false because it has no non-circular support. In case it is discovered at a node N that an atom p has no non-circular support, the procedure should be allowed to infer **not** p in the generation of the descendant of N. This is done in terms of the $Falsify$ operation described later in this section.

These two features of the new proof procedure create the possibility that a literal l^a is inferred on the basis of an assumption which is later shown to be false. This can lead to making wrong inferences unless we redefine the result of matching a literal **not** a with the assumption $\underline{a}$ to be $true^{\textbf{not}\ a}$ instead of $false^a$. The example below makes this point.

Example 7.2.1 *Let P be the ground program $\{b \leftarrow$ **not** a, **not** c; $a \leftarrow a$; $c \leftarrow$ **not** $c\}$. Suppose the query is b. Clearly, P does not strongly entail b. Suppose the proof procedure begins by assuming $\underline{\textbf{not}\ c}$. Then the first rule becomes $b \leftarrow$ **not** a, $true^{\textbf{not}\ c}$. Suppose the next assumption is $\underline{a}$. If the result of matching $\underline{a}$ with **not** a in the body of the first rule were to be $false^a$, then the first rule would reduce to $b \leftarrow false^a$, $true^{\textbf{not}\ c}$. This simplifies to $b \leftarrow false^a$, thereby throwing out $true^{\textbf{not}\ c}$. Thus, the procedure would wrongly infer **not** b^a. Since there is no non-circular support for a, the procedure at this step by using the $Falsify$ operation, which is described below, should infer **not** a. It would also infer $c^{\textbf{not}\ c}$ on the basis of the assumption $\underline{\textbf{not}\ c}$. In translating this set of assumptions and inferences into an interpretation, clearly a should be assigned F. This would result in assigning F to **not** b since its superscript, a, is assigned F. Thus, b would wrongly be assigned T. But in no well-supported model of P can b be assigned T.*

*However, if the result of matching **not** a with a were to be $true^{\textbf{not}\ a}$ then the first rule would reduce to $b \leftarrow true^{\textbf{not}\ a}$, $true^{\textbf{not}\ c}$. In this case the procedure would infer $b^{\textbf{not}\ a \wedge \textbf{not}\ c}$. Thus, the information that the reduction of the first rule* *is partially on the basis of* $\underline{\textbf{not } c}$ *is not lost. Now* b *would be assigned* CT *since* a *would be assigned* F *and* **not** c *would be assigned* CT*. This is the correct result.*

In light of the above example, we take the result of matching **not** a with a to be $true^{\textbf{not } a}$ instead of $false^{a}$.

For any atom a we understand $Neg(a)$ to be **not** a and $Neg(\textbf{not } a)$ to be a.

Then, matching is redefined as follows.

Definition 7.2.1 *Let* R *be the normal logic rule*

$$a \leftarrow b_1, \ldots, b_n, \textbf{not } c_1, \ldots, \textbf{not } c_m$$

Matching is defined in terms of the following rules.

1. *A negative assumption* $\underline{\textbf{not } l}$ *matches with* **not** $l \in body(R)$ *resulting in* $true^{\textbf{not } l}$*, which replaces* **not** l *in the body of* R*.*

2. *A positive assumption* $\underline{l}$ *matches with* **not** $l \in body(R)$ *resulting in* $true^{\textbf{not } l}$ *which replaces* **not** l *in the body of* R*.*

3. *A positive inference* l^S*, where* S *is non-empty, matches with* $l \in body(R)$ *(or* **not** $l \in body(R)$*) resulting in* $true^S$ *(resp.,* $true^{\textbf{not } S}$*), which replaces* l *(resp.,* **not** l*) in the body of* R*.*

4. *A negative inference* **not** l^S*, where* S *is non-empty, matches with* $l \in body(R)$ *(or* **not** $l \in body(R)$*) resulting in* $true^{\textbf{not } S}$ *(resp.,* $true^S$*), which replaces* l *(resp.,* **not** l*) in the body of* R*.*

5. *A positive or negative inference* l*, without any superscript or the empty superscript, matches with* $l \in body(R)$ *(or* $Neg(l) \in body(R)$*) resulting in* $true$ *(resp.,* $false$*), which replaces* l *(resp.,* $Neg(l)$*) in the body of* R*.*

The rules for inferring literals are given as follows:

a can be inferred from $a \leftarrow\ true$. **not** a can be inferred from $a \leftarrow\ false$. a^S can be inferred from $a \leftarrow\ true^S$.

Example 7.2.2 *Let P be the ground program $\{b \leftarrow\ e;\ c \leftarrow\ c, d;\ d \leftarrow \textbf{not}\ c;\ e \leftarrow \textbf{not}\ e\}$. The assumption $\underline{\textbf{not}\ e}$ cannot match with e in the first rule according to the redefined rules of matching. However, the assumption $\underline{\textbf{not}\ e}$ matches with* **not** *e in the fourth rule, which turns it into e $\leftarrow\ true^{\textbf{not}\ e}$. From this e$^{\textbf{not}\ e}$ can be inferred. The redefined rules of matching permit the inference e$^{\textbf{not}\ e}$ to match with the e in the first rule, which turns it into $b \leftarrow\ true^{\textbf{not}\ e}$, from which can be inferred $b^{\textbf{not}\ e}$. The assumption $\underline{\textbf{not}\ c}$ matches with* **not** *c in the third rule, which makes the third rule into $d \leftarrow\ true^{\textbf{not}\ c}$. However, the redefined rules of matching do not permit the assumption $\underline{\textbf{not}\ c}$ to match with c in the second rule.*

In light of the changes in the rules for inferring literals we also need to change the definition of the T^P operator defined in the weak entailment section. We redefine this concept in the next definition. This definition is identical to the definition of the T^P operator of the previous chapter except that the rules of matching refer to the rules of matching defined above in this chapter.

Definition 7.2.2 *Let P be a ground normal logic program. Let I be a set of literals, consisting of assumptions and superscripted literals. Then,*

$T'^P(I) = I \cup \{a^S \mid a \leftarrow body\ \in P$, and matching literals in body with literals in I results in $a \leftarrow true^S\} \cup \{\textbf{not}\ a^S \mid a \leftarrow body\ \in P$, and matching literals in body with literals in I results in $a \leftarrow false\}$.

In this definition we assume that S can be possibly empty.

Example 7.2.3 *As in Example 7.2.2 above, let* $P = \{b \leftarrow\ e;\ c \leftarrow\ c, d;\ d \leftarrow$ $\mathbf{not}\ c;\ e \leftarrow \mathbf{not}\ e\}$.

In this case

$$T'^P(\{\underline{\mathbf{not}\ c},\ \underline{\mathbf{not}\ e}\}) \ = \ \{\underline{\mathbf{not}\ c},\ d^{\mathbf{not}\ c},\ \underline{\mathbf{not}\ e},\ e^{\mathbf{not}\ e}\}$$
$$lfp(T'^P(\{\underline{\mathbf{not}\ c},\ \underline{\mathbf{not}\ e}\})) \ = \ T'^P(T'^P(\{\underline{\mathbf{not}\ c},\ \underline{\mathbf{not}\ e}\})) \ =$$
$$\{\underline{\mathbf{not}\ c},\ \underline{\mathbf{not}\ e},\ d^{\mathbf{not}\ c},\ e^{\mathbf{not}\ e},\ b^{\mathbf{not}\ e}\}.$$

The program reduces to $\{c \leftarrow c,\ true^{\mathbf{not}\ c}\}$.

The new definition of the T'^P operator requires a corresponding redefinition of the Γ operator from the section on weak entailment. The new operator will be called Γ'. The redefinition consists in substituting all occurrences of the T^P operator with the T'^P operator.

Definition 7.2.3 *Let* $N = \langle P, A, Inf, H\rangle$. *Then* $\Gamma'(N) = \langle P', A, Inf', H'\rangle$ *where* $Inf' = lfp(T'^P(A \cup Inf)) - A$, $H' = H - Atoms(Inf') - \mathbf{not}\ Atoms(Inf')$, *and* $P' = P - \{R \in P | head(R)^S \in (Inf' - Inf)$ *or* $\mathbf{not}\ head(R)^S \in (Inf' - Inf)\}$

Example 7.2.4 *As in Examples 7.2.3 and 7.2.2, let* $P = \{b \leftarrow\ e;\ c \leftarrow\ c, d;\ d \leftarrow$ $\mathbf{not}\ c;\ e \leftarrow \mathbf{not}\ e\}$. *Let* $SN \ = \ \langle P,\ \{\underline{\mathbf{not}\ c},\ \underline{\mathbf{not}\ e}\},\ \emptyset,\ \{b,\ \mathbf{not}\ b,\ d,\ \mathbf{not}\ d\}\rangle$.

As seen in Example 7.2.3, the least fix-point of the T'^P *operator as applied to* $\{\underline{\mathbf{not}\ c},\ \underline{\mathbf{not}\ e}\}$, *is* $\{\underline{\mathbf{not}\ c},\ \underline{\mathbf{not}\ e},\ d^{\mathbf{not}\ c},\ e^{\mathbf{not}\ e},\ b^{\mathbf{not}\ e}\}$.

Hence $\Gamma'(SN) = \langle P',\ A', Inf',\ H'\rangle$ *where* $A' = \{\underline{\mathbf{not}\ c},\ \underline{\mathbf{not}\ e}\}$ *and*

$$P' \ = \ \{c \leftarrow c,\ true^{\mathbf{not}\ c}\}$$
$$Inf' \ = \ \{d^{\mathbf{not}\ c},\ e^{\mathbf{not}\ e},\ b^{\mathbf{not}\ e}\}$$
$$H' \ = \ \emptyset$$

In the proof procedure for weak entailment the main part is the procedure *Proc*. The main part of the proof procedure for strong entailment will be the procedure *ProcStrong*, which is based on *Proc*. One key difference between the two procedure results from the fact that in the case of the procedure for strong entailment we do not allow a negative assumption to match with a positive literal in the body of a rule. However, as in the case of weak entailment we do not allow a positive assumption to match with a positive literal in the body of a rule either. Thus, if we were to use procedure *Proc* for strong entailment it can result in the procedure *Proc* reaching a node in which the program part is not empty and there are no more assumptions to make. At this point all the remaining rules will have only positive atoms in the bodies (in addition to $true^S$ or $false^S$, for some S). The atoms in the head of these rules have no non-circular support. Thus their negation can be inferred and the rules with these atoms in the head can be deleted from the program part. The procedure *Proc* needs to be modified to take this step. To do this we define the $Falsify$ operation which when applied to a program P puts the special atom $false$ in the body of each rule of P which has only positive atoms (including the special atoms, which may be superscripted) in its body. The $Falsify$ operation is applied only when there are no more assumptions left to be made.

Definition 7.2.4 *Let P be the ground program $\{p \leftarrow q;\ q \leftarrow \textbf{not } r\}$. Then*

$$Falsify(P) \ = \ \{p \leftarrow q,\ false;\ q \leftarrow \textbf{not } r\}.$$

Note that $false$ is inserted in the body of the first rule only.

We redefine the descendants of a node using the $Falsify$ operation.

Definition 7.2.5 *descendants(N)* $=$

$$
\begin{cases}
\Gamma'(N) & \text{if } \Pi_1(\Gamma'(N)) = \emptyset \\[1em]
\Gamma'(\langle\ Falsify(\Pi_1(\Gamma'(N))),\ \Pi_2(\Gamma'(N)),\ \Pi_3(\Gamma'(N)),\ \Pi_4(\Gamma'(N))\rangle) & \begin{array}{l}\text{if } \Pi_1(\Gamma'(N)) \neq \emptyset \\ \text{and } \Pi_4(\Gamma'(N)) = \emptyset\end{array} \\[1em]
\begin{array}{l}\{\langle \Pi_1(\Gamma'(N)),\ (\Pi_2(\Gamma'(N))\ \cup \{\underline{l}\}), \\ \ \Pi_3(\Gamma'(N)),\ (\Pi_4(\Gamma'(N)) - \{l, \mathbf{not}\ l\})\rangle \mid l \in \Pi_4(\Gamma'(N))\}\end{array} & \text{otherwise}
\end{cases}
$$

This key difference between the procedure *Proc* and *ProcStrong* is encoded in the second clause of the above definition of *descendants*.

Example 7.2.5 *As in Example 7.2.4 above, let* $P = \{b \leftarrow\ e;\ c \leftarrow\ c, d;\ d \leftarrow \mathbf{not}\ c;\ e \leftarrow \mathbf{not}\ e\}$ *and let* $SN = \langle P,\ \{\underline{\mathbf{not}\ c},\ \underline{\mathbf{not}\ e}\},\ \emptyset,\ \{b,\ \mathbf{not}\ b,\ d,\ \mathbf{not}\ d\}\rangle$. *Example 7.2.4 above computed* $\Gamma'(SN)$ *to be* $\langle P',\ A', Inf',\ H'\rangle$ *where* $A' = \{\underline{\mathbf{not}\ c},\ \underline{\mathbf{not}\ e}\}$ *and*

$$
\begin{aligned}
P' &= \{c \leftarrow c,\ true^{\mathbf{not}\ c}\} \\
Inf' &= \{d^{\mathbf{not}\ c},\ e^{\mathbf{not}\ e},\ b^{\mathbf{not}\ e}\} \\
H' &= \emptyset
\end{aligned}
$$

Since the $\Pi_1(\Gamma'(SN)) \neq \emptyset$ *and* $\Pi_4(\Gamma'(SN)) = \emptyset$, *descendants*$(SN)$ *must be computed in terms of the second clause of the definition. Thus, the* $Falsify$ *operation must be applied to* $c \leftarrow c,\ true^{\mathbf{not}\ c}$, *which turns it into* $c \leftarrow c,\ true^{\mathbf{not}\ c},\ false$. *Thus,* *descendants*$(SN) = \Gamma'(\langle\ \{c \leftarrow c,\ true^{\mathbf{not}\ c},\ false\},\ \{\underline{\mathbf{not}\ c},\ \underline{\mathbf{not}\ e}\},\ Inf',\ \emptyset\rangle)$, *where* Inf' *is as above in the specification of* $\Gamma'(SN)$.

As noted above, from the point of view of strong entailment the difference between CT and CF is of no significance. We will exploit this feature of strong entailment in the proof procedure. The following apparatus is required to do this.

We introduce a new truth value X, which can stand indifferently for either CT or CF. We call $\mathcal{X} = \{F,\ X,\ T\}$ an abstraction of $\mathcal{V} = \{F,\ CF,\ CT,\ T\}$. We assume the ordering $F < X < T$. We map T to 1, X to 1/2, and F to 0.

Definition 7.2.6 *A mapping from the Herbrand base of a logic program P to $\mathcal{X}$ is an abstract interpretation of P.*

We assume that **not** $X = X$. As in the case of **C4**, **not** $T = F$ and **not** $F = T$. Also we assume, as in the earlier chapters, that given $v_1,\ v_2 \in \mathcal{X}$, $v_1 \vee\ v_2 = max\{v_1,\ v_2\}$ and $v_1 \wedge\ v_2 = min\{v_1,\ v_2\}$. Given a rule $a \leftarrow B$, where a is a ground atom and B is a conjunction of ground literals, then given an abstract interpretation $\mathcal{J}$, $a \leftarrow B$ evaluates to T if $\mathcal{J}(a) \geq \mathcal{J}(B)$ and F otherwise.

Definition 7.2.7 *An abstract interpretation $\mathcal{J}$ of a normal logic program is an abstract model of P iff every rule $R \in P$ evaluates to T in $\mathcal{J}$.*

An abstract model $\mathcal{J}$ of a normal logic program is an abstract well-supported model of P iff for every atom that is assigned a value greater than F by $\mathcal{J}$ there is a rule that supports the attribution of this value in a non-circular way in exactly the way specified for non-abstract models in Definition 4.3.2 in Chapter 4

7.3 Algorithms

Just as in computing whether a query is *weakly* entailed by a program we need to consider only the relevant rules of the program, in computing whether a query is *strongly* entailed by a program we need to consider only the *related* rules of the program. This concept is defined below.

Definition 7.3.1 *A ground atom a is related to a ground atomic query q in a normal logic program P if a is q or $a \in Atoms(body(R))$, where $head(R)$ is related to q. A rule R is related to a query q if its head is related to q.*

Although a query L can be a conjunction or disjunction of literals, the definition of a related rule above in terms of an atomic query will still serve our purposes because we assume that we add the rule $query \leftarrow L$ to the program and answer the original query by answering the query *query*.

In determining whether a query q is *strongly* entailed by a program we need to consider only those rules in P that are related to q. This is different from the case of weak entailment where we have to look at rules that are relevant to the query in the special sense of the term as defined in the previous section. The key difference between the two concepts is that a rule can be relevant to answering a query if the *body* of the rule contains some atom that is *related* to the query, even if the head of that rule is not related to answering the query; whereas a rule is related to answering a query only if its head is related to answering a query. Roughly speaking the difference consists in whether in determining if query is entailed by the program we need to look at the consequences of query in the program. For strong entailment (and thus for the well-founded semantics) the consequences of query are irrelevant to determining whether query is entailed by the program, whereas for weak entailment the consequences of a query can in some cases prevent the program from weakly entailing the query.

Example 7.3.1 *Let P be the ground program*

$$\{a \leftarrow \textbf{ not } b;\ b \leftarrow \textbf{ not } a;\ p \leftarrow \textbf{ not } p \ \vee \ \textbf{not } b\}.$$

Then $a \leftarrow$ **not** b *and* $b \leftarrow$ **not** a *are* related *to* a*, whereas, additionally,* $p \leftarrow$ **not** $p \vee$ **not** b *is also* relevant *to* a*.*

The following lemma justifies restricting the procedure to only related rules.

Lemma 7.3.1 *A ground normal logic program* P *strongly entails a ground literal* q *iff* $P' \subseteq P$ *strongly entails* q*, where* P' *consists of all rules in* P *related to* q*.*

Proof:

$\Rightarrow$

We will prove that if a normal logic program P strongly entails a literal q then $P' \subseteq P$ strongly entails q by proving the converse of this claim. So assume that P' does not strongly entail q, where P' consists of all the rules related to q. Thus $q \notin WFS(P')$. The well-founded semantics of a normal logic program can be computed in a bottom-up manner in terms of the least fixed point of the operator W_P as described in Chapter 3. In this definition of the well-founded semantics, a literal l belongs to $WFS(P)$ only if l is in some iteration β of W_P. Either $l = a$ or $l =$ **not** a, for some atom a. If $l = a$, then $l = a$ is in some iteration n of W_P only if there is a rule $R = l \leftarrow body$ such that each member of $body$ is in some iteration $m < n$ of W_P. If $l =$ **not** a, then for each rule R with a in the head there must be some literal such that its negation is in some iteration $m < n$ of W_P. Thus, any rule R that can play a role in the generation of a literal l in some iteration of W_P must be related to l. By definition all such rules are in P'. Thus if $q \notin WFS(P')$ then $q \notin WFS(P)$. Hence, if P' does not strongly entail q then P does not strongly entail q.

$\Leftarrow$

Assume that P' strongly entails q. Then $q \in WFS(P')$. Then clearly $q \in WFS(P)$

since all rules in $P - P'$ are unrelated to q and thus cannot alter the status of q in $WFS(P)$. Thus, if $q \in WFS(P')$ then $q \in WFS(P)$ and P strongly entails q. $\blacksquare$

In light of the above lemma, in determining strong entailment, we will assume that the program contains only rules that are related to answering the query. It is easy to determine this dynamically, but in the interest of keeping the proof procedure simple we shall assume that all rules in the program are related to answering the query.

Given a node N, *ProcStrong* finds a leaf node that can be reached from N. *ProcStrong* does a depth-first search for such a leaf node by making recursive calls to itself.

$ProcStrong(N)$

1. If $\Pi_1(\Gamma'(N)) = \emptyset$ then RETURN $\Gamma'(N)$
2. else **begin**
   3. Create unvisited descendent N'
   4. $Status(N') \leftarrow \ visited$
   5. $Parent(N') \leftarrow N$
   6. $ProcStrong(N')$
   7. **end**

If in step 1, it has reached a leaf node it returns $\Gamma'(N)$ and terminates at step 1. Otherwise in steps 3 to steps 5 it creates and initializes N', a descendent of N, and at step 6 recursively invokes *ProcStrong* with N'. Note that at step 3 an unvisited descendent is created by choosing a new assumption if there are any assumptions left to be made; otherwise, if there are no more assumptions left, a new node is created by applying the $Falsify$ operation.

MainStrong takes a finite and ground normal logic program LP and a ground query L as arguments. It adds the rule $query \leftarrow L$ to LP, where $query$ is an atom that does not occur in HB_{LP}. It creates the starting node which has the special atom *true* as the starting assumption. Then it invokes *ProcStrong*, which is described above. It returns YES if $query$ is assigned T by the $TransStrong$ operation, which is described below, applied to the node returned by *ProcStrong*.

$MainStrong(LP,\ L)$

$P \leftarrow LP \cup \{query \leftarrow L\}$

$SN \leftarrow CreateNode(P,\ \{true\})$

$N \leftarrow ProcStrong(SN)$

If $TransStrong(N)$ assigns T to $query$ then return YES, else return NO

Here $MainStrong$ uses the $CreateNode$ procedure described in the previous chapter on proof procedure for *weak* entailment. The $TransStrong$ operation, which converts a node into an abstract interpretation, is given below.

$TransStrong(N)$

1. $\mathcal{I} \leftarrow \emptyset$
2. $Inf \leftarrow\ \Pi_3(N)$
3. $Assp \leftarrow\ \ \Pi_2(N)$
4. For each positive inference $a \in \Pi_3(N)$ with an empty superscript,

   begin

   $\mathcal{I} \leftarrow (\mathcal{I}\ \cup\ \{a \mapsto T\})$

   $Inf \leftarrow\ Inf - \{a^S\}$, where S is any superscript including the empty superscript

   end

5. For each negative inference **not** $a \in \Pi_3(N)$ with an empty superscript,

begin

$\mathcal{I} \leftarrow (\mathcal{I} \cup \{a \mapsto F\})$

$Inf \leftarrow Inf - \{\textbf{not } a^S\}$, where S is any superscript including the empty superscript

end

6. While Inf contains any literal l^S such that $\mathcal{I}(S)$ has a value, do

begin while

Choose an $l^S \in Inf$ such that $\mathcal{I}(S)$ has a value

If l is the atom a then $\mathcal{I} \leftarrow (\mathcal{I} \cup \{a \mapsto \mathcal{I}(S)\})$

else if l is the negative literal **not** a then $\mathcal{I} \leftarrow (\mathcal{I} \cup \{a \mapsto (1 - \mathcal{I}(S))\})$

Delete l^S from Inf

end while

7. $Assp \leftarrow Assp - \{\underline{a}, \underline{\textbf{not } a} \mid \mathcal{I}(a) \text{ is defined}\}$

8. For each positive assumption $\underline{a} \in Assp$,

$\mathcal{I} \leftarrow (\mathcal{I} \cup \{a \mapsto X\})$

9. For each negative assumption $\underline{\textbf{not } a} \in Assp$,

$\mathcal{I} \leftarrow (\mathcal{I} \cup \{a \mapsto X\})$

10. $Inf \leftarrow Inf - \{a^S, \textbf{not } a^S \in Inf \mid \underline{a} \in Assp \textit{ or } \underline{\textbf{not } a} \in Assp\})$

11. While Inf is not empty do

begin while

Choose an $l^S \in Inf$ such that $\mathcal{I}(S)$ has a value

If l is the atom a then $\mathcal{I} \leftarrow (\mathcal{I} \cup \{a \mapsto \mathcal{I}(S)\})$

else if l is the negative literal **not** a then $\mathcal{I} \leftarrow (\mathcal{I} \cup \{a \mapsto (1 - \mathcal{I}(S))\})$

Delete l^S from Inf

end while

$TransStrong$ is the same procedure as $Trans$ of previous chapter, except that where $Trans$ would assign CT or CF to an atom, $TransStrong$ assigns X. Thus, given a node N, $TransStrong$ translates that node into an abstract interpretation.

Since the starting node with which *ProcStrong* is invoked by *MainStrong* contains no assumptions other than *true*, and since, as will be proven below in Lemma 7.4.3, the implicit graph of every normal logic program contains at least one leaf node, it follows that if *ProcStrong* is correct, it must return at least one non-nil node. This is different from *Proc* where the starting node contains an assumption, and, thus, there can be no guarantee that the implicit graph contains a leaf node consistent with the starting assumption.

7.4 Proofs

In this section we prove the soundness and completeness of $MainStrong$ with respect to strong entailment.

To prove soundness we need to prove that if $MainStrong$ returns YES to a query L for a normal logic program LP, then L is strongly entailed by LP. $MainStrong$ returns YES to query L only if $query$ evaluates to T in $TransStrong(N)$, where N is the node returned by procedure $ProcStrong$, which is invoked by $MainStrong$. $ProcStrong$ returns N only if N is a leaf node. Thus, to prove soundness of $MainStrong$ we need to prove that

- if $query$ is assigned T by $TransStrong(N)$, where N is any leaf node in the implicit graph of $P = LP \cup \{query \leftarrow L\}$, then $query$ is strongly entailed by P, and

- if *query* is strongly entailed by P then L is strongly entailed by LP

We need to introduce the following definitions to prove that if a literal l is assigned T in $TransStrong(N)$, where N is a leaf node in the implicit graph of a normal logic program P, then l is strongly entailed by P.

Definition 7.4.1 *A literal $l^S \in \Pi_3(N)$, where N is a leaf node, is assumption free if and only if S is the empty superscript or each conjunct in some disjunct of S is logically equivalent to an assumption free literal or is logically equivalent to the negation of an assumption free literal.*

In this context we understand **not not** p to be logically equivalent to p. Although we do not recognize expressions such as **not not** p to be well-formed expressions in the language of normal logic programs, such expressions do occur in superscripts, given our new rules of matching.

Example 7.4.1 *Let $P = \{q \leftarrow \textbf{ not } p;\ p \leftarrow p\}$. Consider the leaf node*

$$\langle \emptyset, \{\underline{\textbf{not } p}\},\ \{q^{\textbf{not } p},\ \textbf{not } p\},\ \emptyset \rangle.$$

Here the inference $q^{\textbf{not } p}$ is assumption free because although **not** *p was assumed,* **not** *p was also inferred without making any assumptions. Hence $q^{\textbf{not } p}$ should be regarded assumption free since the inference* **not** *p is assumption free. In other words, q could have been inferred without making the assumption* **not** *p.*

Thus, assumption free literals can be inferred without making any assumptions. As the above example shows, even though a literal p might have been assumed in inferring l^S in N, l^S can still be regarded as assumption free if p is itself assumption free.

Lemma 7.4.1 *Let P be a finite and ground normal logic program, and let N be a node returned by procedure $ProcStrong$ for P. An atom $a \in HB_P$ is assigned T (F) in $TransStrong(N)$ if and only if a^S (resp.,* **not** *a^S), for some S, is assumption free in N.*

Proof: It is evident from steps 4, 5 and 6 of procedure $TransStrong$ that each assumption free literal l^S is assigned T by $TransStrong$. It is evident from steps 7-11 of $Trans$ that no literal that is not assumption-free is assigned either T or F by $Trans$. These two observations together establish the lemma. ∎

Atoms that are assigned T or F by $TransStrong(N)$ can be stratified as follows.

Definition 7.4.2 *Let P be a normal logic program and let N be a leaf node in the implicit graph of P. Let S be the set of atoms that are assigned T or F by $TransStrong(N)$. S is stratified as follows.*

- *Strata 0: The special atoms $true$ and $false$,*
- *Strata 1: Any atom a such that P contains a rule $a \leftarrow true$ or $a \leftarrow false$ or $a \leftarrow$* **not** *$true$. Any atom a that is assigned F as a result of applying the $Falsify$ operation in the computation of N.*
- *Strata $n > 1$: a does not already belong to a stratum $k < n$ and*
  - *$TransStrong(N)(a) = T$ and there is a rule R_a s.t. $head(R_a) = a$ and $body(R_a)$ evaluates to T in $TransStrong(N)$ and each member of $Atoms(body(R_a))$ is of stratum less than n, or*
  - *$TransStrong(N)(a) = F$ and each rule R_a which contains a in the head is such that $body(R_a)$ contains a literal l that evaluates to F in*

$TransStrong(N)$ and is such that the atom in l is of stratum less than n.

In connection with this definition recall that all unit clauses in P are understood as having $true$ in the body and a rule $a \leftarrow false$ is inserted in a program only if the program contains no rules with a in the head.

All atoms that are assigned T or F belong to this stratification because by the above lemma they are all assumption free. Thus regarding such atoms we cannot have a situation where an atom is inferred on the basis of an assumption l and the atom is then used to infer l or $Atom(l)$. Hence, the stratification described above is possible and includes all atoms that are assigned T or F by $Trans$.

Now we are in a position to prove that if a literal l is assigned T in $TransStrong(N)$, where N is a leaf node in the implicit graph of a normal logic program P, then l is strongly entailed by P.

Lemma 7.4.2 *Let P be a finite and ground normal logic program, and let N be a leaf node in the implicit graph of P. Then, if a literal l evaluates to T in $TransStrong(N)$ then l is strongly entailed by P.*

Proof: Assume that N is a leaf node in the implicit graph of P. Assume that l evaluates to T in $TransStrong(N)$. Let S be the set of atoms in HB_P that are assigned either T or F by $TransStrong(N)$. Members of S can be stratified in the manner described above.

Assume by way of contradiction that l is not strongly entailed by P. So there must be a canonical model $\mathcal{I}$ of P such that $\mathcal{I}(l) < T$. So there is a non-empty $S' \subseteq S$ such that S' consists of those members of S to which $\mathcal{I}$ assigns a different truth value than $TransStrong(N)$. Let a be any atom in S' such that a is in the lowest stratum in terms of the stratification of S. a is assigned either T or F by $TransStrong(N)$.

Case 1: Assume that a is assigned T by $TransStrong(N)$. So there must be a rule R_a such that $head(R_a) = a$ and $body(R_a)$ evaluates to T in $TransStrong(N)$. Furthermore, all members of $Atoms(body(R_a))$ must be of a lower stratum than a.

By assumption $\mathcal{I}(a) < T$. Furthermore, by the assumption, that a is of the lowest stratum among members of S', $\mathcal{I}$ must assign the same truth value to all members of $Atoms(body(R_a))$ as $TransStrong(N)$. Hence, $body(R_a)$ must evaluate to T in $\mathcal{I}$. But since $\mathcal{I}(head(R_a)) < T$, $\mathcal{I}$ is not a model of R_a, which contradicts the assumption that $\mathcal{I}$ is a model of P.

Case 2: Assume instead that a is assigned F by $TransStrong(N)$. So for each rule R_a such that $head(R_a) = a$, $body(R_a)$ must contain a literal l that evaluates to F in $TransStrong(N)$ and such that the atom in l must be of a lower stratum than a.

By assumption $\mathcal{I}(a) > F$ because its value is different from the value assigned to it by $TransStrong(N)$, which assigns it F. Furthermore, by the assumption, that a is of the lowest stratum among members of S', $\mathcal{I}$ must assign the same truth value to the l in each $Atoms(body(R_a))$ as $TransStrong(N)$. Hence, each $body(R_a)$ must evaluate to F in $\mathcal{I}$. But since $\mathcal{I}(head(R_a)) > F$, $\mathcal{I}$ cannot be a well-supported model of P. This contradicts the assumption that $\mathcal{I}$ is a canonical model of P. ■

The following theorem proves the soundness of $MainStrong$.

Theorem 7.4.1 *If the procedure MainStrong returns YES for query L and a normal logic program $LP' \subseteq LP$ which consists of the set of rules in LP related to L* *then LP strongly entails L.*

Proof: Assume that the procedure returns YES for program LP' and query L. Let $P' = LP' \cup \{query \leftarrow L\}$. So procedure *ProcStrong* returns a leaf node N such that $TransStrong(N)(query) = T$. Thus, it follows from Lemma 7.4.2 that *query* is strongly entailed by P'. Hence, P' must strongly entail L since the only rule with *query* in the head is $query \leftarrow L$. Thus LP' must entail L since LP' is identical to P' except that $query \leftarrow L$ is in P'. It follows from Lemma 7.3.1 that P strongly entails L. ∎

Next we prove the completeness of $MainStrong$. First we show that for any finite and ground normal logic program, P, *ProcStrong* returns a leaf node N which can be translated by $TransStrong$ into an abstract well-supported model of P. Second, we show that $TransStrong(N)$, where N is a leaf node returned by $ProcStrong$ with P as the input program is equivalent to the well-founded semantics of P, $WFS(P)$. Since a normal logic program P entails a literal l iff $l \in WFS(P)$ (Theorem 5.4.1 in Chapter 5), it follows that a normal logic program P entails a literal l iff l evaluates to T in $TransStrong(N)$. But in that case $MainStrong$ would return YES to the query l. This establishes completeness of $MainStrong$.

The following lemma establishes that for any finite and ground normal logic program, $ProcStrong$ returns a leaf node.

Lemma 7.4.3 *Let P be a finite and ground normal logic program. Then $ProcStrong$ returns at least one leaf node for P as the input program.*

Proof: A node in the implicit graph of P is a leaf node if the program part of the node is empty. By step 1 of $ProcStrong$ any node that it returns must be a leaf node. So *ProcStrong* can fail to return a leaf node only if it fails to terminate. But *ProcStrong* can fail to terminate only if at some node N, which is not a leaf node, the program part of N contains a rule R that cannot be reduced any further regardless of which additional assumptions are made. If body of R contains any negative literals, then clearly those negative literals can be assumed and R can be reduced further. On the other hand if *body*(R) contains no negative literal, then after exhausting all the remaining assumptions, R can be reduced further by applying the $Falsify$ operation. So R can always be reduced further. Thus, *ProcStrong* must terminate by returning a leaf node. ∎

The next lemma shows that the leaf node returned by *ProcStrong* for a program P can be translated into an abstract well-supported model of P.

Lemma 7.4.4 *Let P be a finite and ground normal logic program. Let N be a leaf node in the implicit graph of P returned by ProcStrong. Then $TransStrong(N)$ is an abstract well-supported model of P.*

Proof: First we show that $TransStrong(N)$ is an abstract model of P.

For any rule $R \in P$, if *body*(R) evaluates to T in $TransStrong(N)$ then each literal l in the body must evaluate to T in $TransStrong(N)$. But then for each such literal l, the inference part of N must contain a literal l^S, where S is the empty superscript or S must evaluate to T in $TransStrong(N)$, or the inference part must contain the literal $Neg(l)^S$ where S must evaluate to F in $TransStrong(N)$. But then the head of R would be inferred with an empty superscript or with a superscript that evaluates to T in $TransStrong(N)$. So $TransStrong(N)$ would be an abstract model of all such rules.

For any rule $R \in P$, if *body*(R) evaluates to X in $TransStrong(N)$ then it contains at least one literal l which evaluates to X and no literal that evaluates to F

in $TransStrong(N)$. But then for each such literal l, the inference part of N must contain a literal l^S or a literal $Neg(l)^S$, where S evaluates to X in $TransStrong(N)$. But then the head of R would be inferred either with a superscript which evaluates to X in $TransStrong(N)$ or which evaluates to T in $TransStrong(N)$. In either case the head would not be assigned F in $TransStrong(N)$. So $TransStrong(N)$ would be an abstract model of all such rules.

For any rule $R \in P$, if $body(R)$ evaluates to F in $TransStrong(N)$ then trivially $TransStrong(N)$ is an abstract model of R.

Thus, it follows that $TransStrong(N)$ must be an abstract model of P.

Next we show that $TransStrong(N)$ is a well-supported model of P. The well-founded ordering can be in terms of the first appearance of a literal in the inferential part of a node in the path from the starting node to the leaf node N. Here we need only consider a literal l such that $TransStrong(N)$ assigns at least X to $Atom(l)$. This ordering must be well-founded because the generation of the nodes and the inferred literals in each node are by process of bottom-up inference which monotonically enlarges the inferential part of nodes. Furthermore, since the assignment of a truth value to any literal is not greater than the truth value assigned to its superscript, the truth value assigned to a literal must be supported.

■

Lemma 7.4.3 and Lemma 7.4.4 together establish that for any finite and ground normal logic program P, $ProcStrong$ returns an abstract well-supported model of P in the sense that it returns a leaf node N which can be translated into such a model by $TransStrong$.

We have represented $TransStrong(N)$ as a mapping from the atoms in N to the truth values $\{F,\ X,\ T\}$. But $TransStrong(N)$ can also be represented as the set of literals which evaluate to T in $TransStrong(N)$, with the understanding that any other literal in N such that neither it or its negation is in $TransStrong(N)$ evaluates to X. We take $WFS(P)$ also to be represented by the set of literals that are true in the well-founded semantics of P. The next lemma states the equivalence of $TransStrong(N)$ and $WFS(P)$.

Lemma 7.4.5 *Let P be a finite, ground normal logic program. Let N be the node returned by ProcStrong operating on P. Let $WFS(P)$ be the well-founded semantics of P. Then $TransStrong(N) = WFS(P)$, where both $WFS(P)$ and $TransStrong(N)$ are represented as a set of literals.*

Proof: It follows straightforwardly from Theorem 7.4.1 above that $TransStrong(N) \subseteq \{l \mid P \text{ strongly entails } l\}$. It follows from Theorem 5.4.1 of Chapter 5 that $\{l \mid P \text{ strongly entails } l\} = WFS(P)$. Thus it follows that $TransStrong(N) \subseteq WFS(P)$. We show next that $WFS(P) = TransStrong(N)$.

Assume by way of contradiction that $TransStrong(N) \subset WFS(P)$. So there must be a non-empty set of literals $S \subseteq WFS(P)$ such that no member of S is in $TransStrong(N)$. The literals of $WFS(P)$ can be stratified in terms of the smallest iteration of the W_P operator in the definition of the well-founded semantics in which the literal first appears. Let $l \in S$ be such that no other literal in S occurs in a lower level of this stratification. Either $l = a$ or $l = \textbf{not}\ a$ for some atom a.

Case 1: $l = a$. So there must be a rule $R \in P$ such that $head(R) = a$ and $body(R)$ is true in $WFS(P)$ and every member of $body(R)$ occurs in a lower level of stratification than a. Clearly, since every member of $body(R)$ occurs in a lower level of stratification than a, all members of $body(R)$ must be in $TransStrong(N)$. Hence, $body(R)$ must evaluate to T in $TransStrong(N)$. Thus, since $TransStrong(N)$ is an abstract model of P, $TransStrong(N)$ must assign T to a. But then $l \in TransStrong(N)$, which contradicts the assumption that $l \in S$.

Furthermore, it is clear from the reasoning above that every atom $p \in S$ which belongs to the lowest strata among members of S must also be in $TransStrong(N)$.

Case 2: $l =$ **not** a. So for each rule $R \in P$ such that $head(R) = a$, $body(R)$ must be false in $WFS(P)$. For each $body(R)$, there must be a literal p in $body(R)$ such that p is false in $WFS(P)$ (that is, $Neg(p) \in WFS(P)$) and $Neg(p)$ belongs to a lower strata or the same strata as **not** a. (Recall that $Neg(b) =$ **not** b and $Neg($**not** $b) = b$ for any atom b.) If $Neg(p)$ belongs to a lower strata than **not** a, then by the reasoning of Case 1, $Neg(p) \in TransStrong(N)$. Hence, $body(R)$ would evaluate to F in $TransStrong(N)$.

On the other hand suppose that $Neg(p)$ is of the same stratum as **not** a. Then either $p = b$ or $p =$ **not** b, for some atom b. Thus, $Neg(p) =$ **not** b or $Neg(p) = b$. If $p = b$, and **not** b and **not** a are of the same stratum, and p occurs in the body of a rule with a in the head, then p and a must mutually support each other and thus must belong by virtue of this mutual support to the unfounded set computed in that iteration of the W_P operator. But in this case the $Falsify$ operation embedded in $ProcStrong$ would produce both **not** a and $Neg(p)$ as an inference in N. Hence both these literals would belong to $TransStrong(N)$. However, if $p =$ **not** b then $Neg(p) = b$. If **not** a and $Neg(p)$, i.e. b, are of the same stratum, then b is of the lowest stratum in S. Since, we have already shown that any atom which is of the lowest stratum in S must also belong to $TransStrong(N)$, it follows that $Neg(p)$ would be in $TransStrong(N)$. Hence, in either case $Neg(p)$ belongs to $TransStrong(N)$. And, thus, since p belongs to $body(R)$, it would evaluate to F

in $TransStrong(N)$. Hence, whether $Neg(p)$ is of the strata or a lower strata than **not** a, $body(R)$ evaluates to F in $TransStrong(N)$. Since this is true of each rule with a in the head, it follows that **not** a must be a member of $TransStrong(N)$ since $TransStrong(N)$ is well-supported. Hence, this contradicts the assumption that l, that is, **not** a, is in S.

Thus, it follows that there cannot be a literal l which belongs to $WFS(P)$ but not to $TransStrong(N)$. Hence $TransStrong(N) = WFS(P)$. ■

Clearly, the above lemma shows that the procedure *ProcStrong* run on a ground and finite normal logic program P computes its well-founded semantics in the sense that it returns a node N which is translated into the well-founded semantics of P by $TransStrong$.

Now we are in a position to prove the completeness of $MainStrong$.

Theorem 7.4.2 *Let LP be a normal logic program. If the procedure MainStrong returns NO for query L to a normal logic program $LP' \subseteq LP$ which consists of the set of rules in LP related to L then LP does not strongly entail L.*

Proof: Let $P = LP' \cup \{query \leftarrow L\}$.

Assume that *MainStrong* returns NO for query L to a normal logic program $LP' \subseteq LP$ which consists of the set of rules in LP related to L. Then $ProcStrong$ run on P must return a node N such that *query* evaluates to T in $TransStrong(N)$. Since the only rule with *query* in the head is $query \leftarrow L$ and since $TransStrong(N)$ is a well-supported model of P (Lemma 7.4.4), it follows that L must evaluate to T in $TransStrong(N)$. Thus, by Lemma 7.4.5, it follows that L is true in $WFS(P)$. Hence, P strongly entails L (by Theorem 5.4.1 of Chapter 5). Then LP' strongly entails L since LP' is identical to P except that it does not contain the rule $query \leftarrow L$. It follows then that LP strongly entails L (by Lemma 7.3.1). ∎

7.5 Discussion

The proof procedure for strong entailment exploits the feature that from the perspective of strong entailment there is no difference between CF and CT. In particular, the proof procedure makes no attempt to distinguish the case where a literal is inferred on the basis of evidence that evaluates to CF as opposed to the case where it is inferred on the basis of evidence that evaluates to CT. This feature allows the proof procedure to eliminate the verifiedness and consistency checks made by the proof procedure for weak entailment.

A consistency check is not required because even if a node N contains an assumption $\underline{l}$ and an inference $Neg(l)^S$, if S evaluates to T or F in $TransStrong(N)$ then l is assigned a value in terms of the value of S; whereas if S evaluates to X in $TransStrong(N)$ then $Neg(l)$ is assigned X and the assumption l is assigned X and this assignment is consistent because the negation of X is X. Thus, the translation of any node is always consistent because we do not distinguish between CT and CF.

In the chapter on weak entailment, a verifiedness check is required to ensure that we do not assign CT to a positive assumption a when in fact there is not enough evidence to assign CT to a, even on the assumption $\underline{a}$. This can happen only when the evidence for a justifies assigning at most CF. In case the evidence justifies assigning at most F then a is assigned F by both $Trans$ of previous chapter and $TransStrong$. But since in this chapter $TransStrong$ would assign X to a

(instead of CT) and X to the evidence for a (instead of CF), it can never happen that the assignment of X to any atom is not well-supported. Thus, a verifiedness check is not required.

Thus, we see that not distinguishing between CF and CT allows us to dispense with the consistency and verifiedness checks, which makes it possible for $ProcStrong$ to terminate without ever having to backtrack. This results in a polynomial time worst-case complexity for $MainStrong$.

$ProcStrong$ expands at most n nodes, where n is the cardinality of the Herbrand base of the input program. As we saw in the last chapter, the worst-case complexity of expanding any node requires $O(n^2)$ matching operations. Thus, the worst-case complexity of answering whether a given program strongly entails a query is $O(n^3)$, where n is the cardinality of the Herbrand base of the program. And thus on our algorithm the worst-case complexity for answering a query with respect to the well-founded semantics of a finite and ground program is $O(n^3)$. This compares well with the standard results for the worst-case analysis for answering a query with respect to the well-founded semantics. For instance, [BDK97] state that the worst-case complexity for answering a query to a finite and ground normal logic program with respect to the well-founded semantics is $O(nm)$, where n is the cardinality of the Herbrand base of the program and m is the length of the program.

7.6 Summary

In this section we summarize the main research contributions of this chapter.

- We have developed a proof procedure for answering whether a ground query consisting of a conjunction or a disjunction of ground literals is *strongly* entailed by a finite and ground normal logic program without any contestations. This proof procedure consists in constructing a well-supported model of the input program in a bottom-up fashion by making assumptions and inferring literals in terms of these assumptions and the input program (Section 7.3).

- We have proven that this proof procedure is sound and complete with respect to the **C4** semantics for normal logic programs (Section 7.4).

- We have proven that the worst-case complexity of this procedure is $O(n^3)$, where n is the cardinality of the Herbrand base of the program (Section 7.5).

- We have proven that this proof procedure also computes the well-founded semantics of a normal logic program (Section 7.4).

Chapter 8

Proof Procedure for Normal Logic Programs with Contestations

8.1 Introduction

In this chapter we describe a proof procedure for determining whether a query is *weakly* entailed by a normal logic program augmented with heterogeneous contestations and a procedure for determining whether a query is *strongly* entailed by a normal logic program augmented with heterogeneous contestations. For both proof procedures we assume that the query is a conjunction or disjunction of ground literals and the program is a finite, ground normal logic program. We also assume that the contestations are ground. These proof procedures will be extensions of the proof procedures we have developed in previous chapters for normal logic programs without contestations.

In Section 8.2 we develop the formal apparatus for stating the proof procedure. In Section 8.3 we state the algorithms for the two proof procedures, and in Section 8.4 we prove that these procedures are sound and complete with respect to **C4**. In Section 8.6 we summarize the main research contributions of this chapter.

8.2 Preliminaries

Recall from Chapter 4 that $P + \mathcal{C}$, where P is a logic program and $\mathcal{C}$ is a set of contestations, is to be understood as the rules of P constrained by the contestations in $\mathcal{C}$. To say that a ground rule $a \leftarrow\ body$ is constrained by a contestation $B \hookrightarrow_i a$ is to say that the rule is really understood as being $a \leftarrow\ body,\ cap'_i(a, B \hookrightarrow_i a, \mathcal{I})$, where $cap'_i(a, B \hookrightarrow_i a, \mathcal{I})$ returns a special atom. Which special atom is returned in this context depends on the nature of the *cap* function and the truth value assigned to B in $\mathcal{I}$. Thus, it is natural to compile the contestations into the bodies of rules in order to extend our proof procedure for normal programs into a proof procedure for normal programs with contestations.

Before we do this we will simplify our notation. We will write $cap'_i(a, B \hookrightarrow_i a, \mathcal{I})$ as $cap'_i(B, a)$. First we can drop the reference to $\mathcal{I}$ since in the context of this proof procedure the cap' function is always evaluated in a certain node which translates into a partial interpretation. Thus, an explicit reference to an interpretation is superfluous. Secondly, $cap'_i(B, a)$ tells us that the contestation is based on a cap_i function and the Contestor part of it is B and the Contested part is a, so the contestation $B \hookrightarrow_i a$ does not have to occur as an explicit argument to the cap'_i function. Thus, in our current context the expression $cap'_i(a, B \hookrightarrow_i a, \mathcal{I})$ can be simplified to $cap'_i(B, a)$ without any loss of information or generality.

Example 8.2.1 *Let P be the ground program $\{a \leftarrow b;\ b \leftarrow;\ c \leftarrow\}$, let $\mathcal{C} = \{c \hookrightarrow_3 a,\ b \hookrightarrow_1 c\}$. Thus, $P + \mathcal{C}$ is understood as $\{a \leftarrow b,\ cap'_3(c, a);\ b \leftarrow;\ c \leftarrow cap'_1(b, c)\}$.*

Let $\mathcal{I}$ be an interpretation of $P + \mathcal{C}$ which assigns T to b and CF to c. In this interpretation $cap'_3(c, a)$ returns $CTrue$ because $cap_3(CF) = CT$ and $cap'_1(b, c)$ returns $CFalse$ because $cap_1(T) = CF$. Hence, relative to this assignment of truth values to b and c, $P + \mathcal{C}$ can be understood as $\{a \leftarrow b,\ CTrue;\ b \leftarrow;\ c \leftarrow CFalse\}$.

Our proof procedure is restricted to $P + \mathcal{C}$ which have C-stable models. This restriction will apply both to the proof procedure for strong entailment as well as the proof procedure for weak entailment. Recall that a model of $P + \mathcal{C}$ is said to have a C-stable model if all the constrained rules of $P + \mathcal{C}$ evaluate to T in that model. It is easy to see that even if P has C-stable models, $P + \mathcal{C}$ may have no C-stable models, as the following example illustrates.

Example 8.2.2 *Let $P = \{p \leftarrow q;\ q \leftarrow\}$. Let $\mathcal{C} = \{p \hookrightarrow_2 q\}$. P has the unique C-stable model which assigns T to both p and q. However, $P + \mathcal{C}$ has the unique well-supported model which assigns CF to p and CT to q, which is not a C-stable model.*

The contestations are compiled into the programs as follows. Assume that the rules are in the disjunctive form. For each rule $a \leftarrow body_1 \vee \cdots \vee body_m$ and for each contestation $B_1 \hookrightarrow_1\ a,\ B_2 \hookrightarrow_2\ a, \ldots, B_n \hookrightarrow_n\ a$, we transform the rule into the rule

$$\begin{aligned} a \leftarrow\ & body_1,\ cap'_1(B_1, a),\ cap'_2(B_2, a), \ldots, cap'_n(B_n, a) \vee \cdots \vee \\ & body_m,\ cap'_1(B_1, a),\ cap'_2(B_2, a), \ldots, cap'_n(B_n, a). \end{aligned}$$

In the above rule each B_i is a conjunction of literals.

Example 8.2.3 *Let P be the ground program*

$$\{a \leftarrow b \vee \textbf{not}\ c;\ b \leftarrow \textbf{not}\ d, \textbf{not}\ e;\ d \leftarrow\ true; e \leftarrow c;\ c \leftarrow \textbf{not}\ b;\ f \leftarrow\ true\}.$$

Let $\mathcal{C} = \{d \wedge \textbf{not}\ f \hookrightarrow_1 a;\ e \hookrightarrow_2 a; c \hookrightarrow_3 d\}$. Then $P + \mathcal{C}$ is

$a \leftarrow b, cap'_1(d \wedge \textbf{not}\ f,\ a),\ cap'_2(e,\ a)\ \vee\ \textbf{not}\ c, cap'_1(d \wedge \textbf{not}\ f,\ a),\ cap'_2(e,\ a);$

$b \leftarrow \textbf{not}\ d, \textbf{not}\ e; \quad d \leftarrow\ true,\ cap'_3(c,\ d);$

$e \leftarrow c; \quad c \leftarrow \textbf{not}\ b; \quad f \leftarrow\ true$

Compiling contestations this way into the bodies of rules allows us to extend naturally the proof procedures developed for logic programs without contestations. This is because when contestations are compiled into the rules of a logic program by means of *cap* functions, the body of the rule is augmented with functions that return special atoms. So in essence a rule with contestations compiled into it will be just like any other logic program rule except that it will have some special atoms in the body. However, which special atom will be in the body of a compiled rule depends on the context consisting of the assumptions and inferences in which a *cap* function is evaluated. A special atom is different from any atom only in that it evaluates to a certain fixed truth value in every interpretation. Hence, a logic program with contestations compiled in the rules is identical to a logic program some of whose atoms have fixed truth values; however, which logic program it is identical to depends on the context in which the *cap* functions are evaluated.

In the following we won't strictly observe the distinction between cap'_i returning a special atom and cap'_i returning the truth value to which that special atom evaluates. Recall that a cap_i function takes a truth value as an argument and returns a truth value. A cap'_i function takes a conjunction of literals and an atom as arguments and which truth value it returns depends on the underlying cap_i function. However, in the interest of notational simplicity we will write cap'_i as cap_i. The context should make it clear which function is intended.

Definition 8.2.1 *A literal l is known in a node N in the implicit graph of a program P if either l or the negation of l is in the assumption part or the inference part of N. More precisely, l is known in N if and only if either $\underline{l'} \in \Pi_2(N)$ or $l'^S \in \Pi_3(N)$, where S is a possibly empty superscript and l' is l or the negation of l. In this context the negation of a negative literal,* **not** *a, is understood to be the* *atom a.*

A cap expression $cap_i(B, a)$ is known in a node N if all the literals in B are known in that node.

A normal logic program P with a set of contestations $\mathcal{C}$ compiled into it can have an implicit graph for it in exactly the same manner as for normal logic programs without contestations. Given a node N in the implicit graph, from a rule of the form $a \leftarrow true^S, cap_i(B_i, p)$ in $\Pi_1(N)$, the literal $a^{S \,\wedge\, cap_i(B_i,p)}$ can be inferred if all members of B_i are known in N.

Example 8.2.4 *Let P+$\mathcal{C}$ be $\{a \leftarrow b \vee \mathbf{not}\ c;\ b \leftarrow cap_1(d,\ b);\ c \leftarrow cap_4(e,\ c);\ d \leftarrow true;\ e \leftarrow true\}$.*

From the last two rules we can infer $\{d^{true},\ e^{true}\}$. Thus, d and e are known. So we can infer $b^{cap_1(d,\ b)}$ and $c^{cap_4(e,\ c)}$. Thus, $P + \mathcal{C}$ can be reduced to the rule $a \leftarrow b \vee \mathbf{not}\ c$.

Note that the above rule of inference does not check whether $cap_i(B_i, p)$ is true before inferring $a^{S \,\wedge\, cap_i(B_i,p)}$. Thus, when the assumptions and inferences in a node are translated into truth values we have no guarantee that a will be assigned CT or T. This raises the issue of what should be the result of matching $a^{S \,\wedge\, cap_i(B_i,p)}$ with **not** a in the body of a rule. If we were to follow the matching rules of Chapter 6, the result would be $false^{S \,\wedge\, cap_i(B_i,p)}$. But this can lead to wrong results in a case where both S and $cap_i(B_i, p)$ evaluate to one of the designated truth values, CT or T. The example below makes this point.

Example 8.2.5 *Let $P + \mathcal{C}$ be as in Example 8.2.4 above. We saw in that example that by the rules for inferring superscripted literals we can infer*

$$\{d^{true},\ e^{true},\ b^{cap_1(d,\ b)},\ c^{cap_4(e,\ c)}\}$$

and $P + \mathcal{C}$ *can be reduced to* $\{a \leftarrow b \vee \textbf{not}\ c\}$.

Matching $b^{cap_1(d,\ b)}$ *with* b *in the body of the rule* $a \leftarrow b \vee \textbf{not}\ c$ *results in* $true^{cap_1(d,\ b)}$. *But if matching* $c^{cap_4(e,\ c)}$ *with* **not** c *in the body of that rule were to result in* $false^{cap_4(e,\ c)}$, *then the rule would become* $a \leftarrow true^{cap_1(d,\ b)} \vee false^{cap_4(e,\ c)}$. *In general,* $true^{S_1} \vee false^{S_2}$ *evaluates to* $true^{S_1}$, *and so the body of the rule* $a \leftarrow b \vee \textbf{not}\ c$ *would be simplified to* $a \leftarrow true^{cap_1(d,\ b)}$ *from which* $a^{cap_1(d,\ b)}$ *would be inferred. But this would not produce a model of* $P + \mathcal{C}$ *when the inferences are translated into a* **C4** *model. Clearly,* d *and* e *should be assigned* T. *Thus,* $cap_1(d,\ b)$ *should evaluate to* CF *and* $cap_4(e,\ c)$ *should evaluate to* F. *Hence,* b *should be assigned* CF *and* c *would be assigned* F. *So* a *would be assigned* CF *because* $cap_1(d,\ b)$ *evaluates to* CF *and because we have inferred* $a^{cap_1(d,\ b)}$. *But clearly this is incorrect because if* c *is assigned* F *then* a *should be assigned* T.

As in Chapter 7, this problem can be avoided if matching $c^{cap_4(e,\ c)}$ *with* **not** c *results in* $true^{\textbf{not}\ cap_4(e,\ c)}$ *instead of* $false^{cap_4(e,\ c)}$. *Now the body of the rule becomes* $true^{cap_1(d,\ b) \wedge \textbf{not}\ cap_4(e,\ c)}$. *Thus,* $a^{cap_1(d,\ b) \wedge \textbf{not}\ cap_4(e,\ c)}$ *would be inferred instead of* $a^{cap_1(d,\ b)}$. *In this case* a *would be correctly assigned* T *when the inferences are translated into a* **C4** *model.*

In light of the above example, as in Chapter 7, we revise the rules of matching as follows. A superscripted literal a^S, where S can contain *cap* expressions, matches with a in the body of a rule resulting in $true^S$ and matches with **not** a in the body resulting in $true^{\textbf{not}\ S}$. When S is empty we shall regard it as implicitly consisting of the special atom *true*. Thus, the result of matchings can only produce $true^{S_i}$, for some superscript S_i which can be the negation of a disjunction of conjunctions. So by the rule of inference stated above, only the superscipted *atoms* can be inferred. But this does not mean that when the inferences are translated into an interpretation all atoms will be assigned CT or T. If a superscript evaluates to CF or F the superscripted atom will be assigned one of these truth values. Thus, the inference of a superscripted *atom* in a node may turn out to be the inference of an (unsuperscripted) negative literal when the node is translated into a **C4** interpretation.

The above described changes in the rules for matching and for inferring superscripted literals as well as the presence of *cap* expressions in the bodies of rules requires us to revise the definition of the T^P operator given in Chapter 6 and Chapter 7.

Definition 8.2.2 *Let P be a ground normal logic program and let $\mathcal{C}$ be a set of contestations. Let $\mathcal{I}$ be a set of literals consisting of assumptions and superscripted literals. Assume that the rules of $P + \mathcal{C}$ are written as*

$$a \leftarrow \; b_{1_1}, \cdots, b_{1_n}, c_1, \cdots, c_m \vee \cdots \vee b_{k_1}, \cdots, b_{k_n}, c_1, \cdots, c_m$$

where $c_1, \cdots, c_m$ are all cap expressions. Then,

$T^{P\mathcal{C}}(\mathcal{I}) = \mathcal{I} \cup \{a^S \mid a \leftarrow$ *body* $\in P{+}\mathcal{C}$ *and matching literals in body with literals in* $\mathcal{I}$ *results in* $a \leftarrow true^S\}$

where body is of the form $b_{1_1}, \cdots, b_{1_n}, c_1, \cdots, c_m \vee \cdots \vee b_{k_1}, \cdots, b_{k_n}, c_1, \cdots, c_m$ *and* $S = S_{1_1} \wedge \cdots \wedge S_{1_n} \wedge c_1 \wedge \cdots \wedge c_m \vee \cdots \vee S_{k_1} \wedge \cdots \wedge S_{k_n} \wedge c_1 \wedge \cdots \wedge c_m$ *and where each of* $c_1, \cdots, c_m$ *are known in* $\mathcal{I}$ *and matching each* b_{i_k} *with literals in* $\mathcal{I}$ *results in* $true^{S_{i_k}}$.

Using this definition of the $T^{P\mathcal{C}}$ operator, we can define the least fixed point of the $T^{P\mathcal{C}}$ operator ($lfp(T^{P\mathcal{C}})$) in a manner completely analogous to that definition in Chapter 6.

Example 8.2.6 *As in Example 8.2.3 above let* $P + \mathcal{C}$ *be*

$a \leftarrow b, cap_1(d \wedge \textbf{not } f,\ a),\ cap_2(e,\ a)\ \vee\ \textbf{not } c, cap_1(d \wedge \textbf{not } f,\ a),\ cap_2(e,\ a)$

$b \leftarrow \textbf{not } d, \textbf{not } e$

$d \leftarrow\ true,\ cap_3(c,\ d)$

$e \leftarrow c$

$c \leftarrow \textbf{not } b$

$f \leftarrow\ true$

Let $\mathcal{I} = \{true,\ \underline{\textbf{not } b}\}$. *Then*

$$T^{P\mathcal{C}}(\mathcal{I}) = \mathcal{I}\ \cup\ \{f^{true},\ c^{\textbf{not } b}\}.$$

$$T^{P\mathcal{C}}(T^{P\mathcal{C}}(\mathcal{I})) = T^{P\mathcal{C}}(\mathcal{I})\ \cup\ \{e^{\textbf{not } b},\ d^{cap_3(c,\ d)}\}.$$

$$T^{P\mathcal{C}}(T^{P\mathcal{C}}(T^{P\mathcal{C}}(\mathcal{I}))) = T^{P\mathcal{C}}(T^{P\mathcal{C}}(\mathcal{I}))\ \cup\ \{b^{\textbf{not } cap_3(c,\ d)\ \wedge\ b}\}$$

$$lfp(T^{P\mathcal{C}}(\mathcal{I})) = T^{P\mathcal{C}}(T^{P\mathcal{C}}(T^{P\mathcal{C}}(T^{P\mathcal{C}}(\mathcal{I})))) = T^{P\mathcal{C}}(T^{P\mathcal{C}}(T^{P\mathcal{C}}(\mathcal{I})))\ \cup$$

$$\{a^{\textbf{not } cap_3(c,\ d) \wedge b \wedge cap_1(d \wedge \textbf{not } f,\ a) \wedge cap_2(e,\ a)\ \vee\ b \wedge cap_1(d \wedge \textbf{not } f,\ a) \wedge cap_2(e,\ a)}\}$$

Using this definition of $T^{P\mathcal{C}}$ and $lfp(T^{P\mathcal{C}})$ we can define the Γ operator and *Descendants* of a node N in a manner completely analogous to those definitions in Chapter 6.

The algorithm for weak entailment will be exactly same as the algorithm for weak entailment without contestations. However, in checking for consistency, stability and verifiedness of a node we cannot as in the case of logic programs without contestations rely on a purely syntactic test. In that case if the node contained p^S and **not** p is part of every disjunct in p^S then the node can be regarded as unstable, and if a node contained p^{S_1} and **not** p^{S_2} then the node can be regarded as inconsistent so long as **not** p is not part of every disjunct in S_1. But in the case of logic programs with contestations we cannot assume that all superscripts will evaluate to CT or T. What truth value a superscript will evaluate to depends on what the *cap* functions in the superscript evaluate to, and there is no way of determining this purely syntactically. Hence, the consistency, stability and verifiedness tests have to be done by translating each node into an interpretation, which may be a partial interpretation. The translation algorithm, $Trans$, of Chapter 6 can be used for this purpose as it can handle partial interpretations. However, that algorithm was written with the assumption that the node to be translated is a consistent and verified node. Hence the algorithm has to be modified to detect unstable, inconsistent or non-verifiable nodes. The algorithm $TransCon$, given below, is designed to do that. But first we need to redefine the concepts of consistency, verifiedness and stability as earlier these concepts were defined purely syntactically.

Recall that given the new rules for inferring literals in a node, all inferences are of superscripted *positive literals*. However, an assumption can be a positive or a negative literal.

Definition 8.2.3 *A node N is unstable if and only if the inference part of N contains a literal a^S such that $S \models$ **not** a and $TransCon(N)(S)$ evaluates to at least CT.*

Definition 8.2.4 *A node N is inconsistent if and only if the assumption part of N contains the negative assumption $\underline{\textbf{not}\ a}$ and the inference part contains a^S such that $TransCon(N)(S)$ evaluates to at least CT and $S \not\models$ **not** a.*

The requirement that $S \not\models$ **not** a is to distinguish inconsistency from unstability in a node.

Definition 8.2.5 *A positive assumption $\underline{a}$ is said to be verified relative to a node N if and only if there exists a literal a^S in the inference part of N such that*

$TransCon(N)(S)$ evaluates to at least CT. A node N is said to be verified if all its positive assumptions are verified relative to N. A positive assumption $\underline{a}$ is said to be unverifiable in a node N if and only if there exists a literal a^S in the inference part of N such that $TransCon(N)(S)$ evaluates to at most CF.

8.3 Algorithms

The above definitions suggest straightforward tests for determining whether a node is consistent, stable, and not unverifiable by translating the node into a partial interpretation. The algorithm for translating a node into a partial interpretation is given below. The tests for stability, consistency and verifiability are built into the $TransCon$ algorithm. The algorithm uses the idea of a superscript expression *simplifying* to *true* or to **not** *true*, which is defined below.

Definition 8.3.1 *An expression $S_1 \wedge \cdots \wedge S_n$ simplifies to true if each S_i, $i \in \{1, \ldots, n\}$, is the expression true. An expression $S_1 \vee \cdots \vee S_n$ simplifies to true if any S_i simplifies to true. An expression $S_1 \wedge \cdots \wedge S_n$ simplifies to* **not** *true if at least one S_i, $i \in \{1, \ldots, n\}$, is the expression* **not** *true. An expression $S_1 \vee \cdots \vee S_n$ simplifies to* **not** *true if each S_i simplifies to* **not** *true.*

$TransCon(N)$

1. $\mathcal{I} \leftarrow \emptyset$
2. $Inf \leftarrow \Pi_3(N)$
3. $Assp \leftarrow \Pi_2(N)$
4. For each inference $a^S \in \Pi_3(N)$ s.t. S simplifies to $true$

begin

$\mathcal{I} \leftarrow (\mathcal{I} \cup \{a \mapsto T\})$

Delete a^S from Inf

end

5. For each inference $a^S \in \Pi_3(N)$ s.t. S simplifies to **not** $true$

   begin

   $\mathcal{I} \leftarrow (\mathcal{I} \cup \{a \mapsto F\})$

   Delete a^S from Inf

   end

6. While Inf contains any literal a^S such that $\mathcal{I}(S)$ has a value, do

   begin while

   Choose an $a^S \in Inf$ such that $\mathcal{I}(S)$ has a value

   $\mathcal{I} \leftarrow (\mathcal{I} \cup \{a \mapsto \mathcal{I}(S)\})$

   Delete a^S from Inf

   end while

7. For each positive assumption $\underline{a}$ s.t. $\mathcal{I}(a)$ is defined

   if $\mathcal{I}(a) = F$ then RETURN "Node not verifiable" and TERMINATE

   else $Assp \leftarrow Assp - \{\underline{a}\}$

8. For each negative assumption $\underline{\textbf{not}\ a}$ s.t. $\mathcal{I}(a)$ is defined

   if $\mathcal{I}(a) = T$ then RETURN "Node not consistent" and TERMINATE

   else $Assp \leftarrow Assp - \{\underline{\textbf{not}\ a}\}$

Comment: Up to this point $\mathcal{I}$ assigns only T or F to atoms.

9. For each positive assumption $\underline{a} \in Assp$,

   $\mathcal{I} \leftarrow (\mathcal{I} \cup \{a \mapsto CT\})$

10. For each negative assumption $\underline{\textbf{not}\ a} \in Assp$,

    $\mathcal{I} \leftarrow (\mathcal{I} \cup \{a \mapsto CF\})$

11. While Inf is not empty do

begin while

Choose an $a^S \in Inf$ such that $\mathcal{I}(S)$ is defined

If $\mathcal{I}(a)$ is defined **Comment: $\underline{a}$ or $\underline{\textbf{not } a}$ has been assumed**

then if $\mathcal{I}(S) < \mathcal{I}(a)$ then RETURN "Node not verifiable"

and TERMINATE

else if $\mathcal{I}(S) \in \{CT, T\}$ and $\mathcal{I}(a) \in \{CF, F\}$ then RETURN "Node not consistent or not stable" and TERMINATE

else $\mathcal{I} \leftarrow (\mathcal{I} \cup \{a \mapsto \mathcal{I}(S)\})$

Delete a^S from Inf

end while

Example 8.3.1 *Let $P + \mathcal{C}$ be as in Example 8.2.3. In Example 8.2.6 we see that for this $P+\mathcal{C}$ starting with the assumption $\{true,\ \underline{\textbf{not } b}\}$ we arrive at the inference set which is given by $lfp(T^{P\mathcal{C}}(\{true,\ \underline{\textbf{not } b}\})$. This is the set*

$$\{f^{true},\ c^{\textbf{not } b},\ e^{\textbf{not } b},\ d^{cap_3(c,\ d)},\ b^{\textbf{not } cap_3(c,\ d)\wedge b},$$
$$a^{\textbf{not } cap_3(c,\ d)\wedge b\wedge cap_1(d\wedge \textbf{not } f,\ a)\wedge cap_2(e,\ a)\ \vee\ b\wedge cap_1(d\wedge \textbf{not } f,\ a)\wedge cap_2(e,\ a)}\}$$

$TransCon$ would translate the node containing this assumption set and this inference set as follows:

$f \mapsto T$ (by step 4)

$b \mapsto CF$ (by step 10)

$c \mapsto CT$ and $e \mapsto CT$ (by step 11)

$cap_3(c,\ d)$ evaluates to CT, so $d \mapsto CT$ (by step 11)

not *$cap_3(c,\ d) \wedge b$ evaluates to CF, hence the initial assignment of CF to b is stable and consistent.*

*$cap_1(d \wedge$ **not** $f,\ a) \wedge cap_2(e,\ a)\}$ evaluates to CF and so* **not** *$cap_3(c,\ d) \wedge b \wedge$*

$cap_1(d \wedge \textbf{not}\ f,\ a) \wedge cap_2(e,\ a)\ \ \vee\ \ b \wedge cap_1(d \wedge \textbf{not}\ f,\ a) \wedge cap_2(e,\ a)$ evaluates to CF, and thus $a \mapsto CF$.

As in the case of the weak entailment proof procedure for logic programs, we will assume that all the rules in $P + \mathcal{C}$ are *relevant* to the *query* posed to the proof procedure. The definition of a rule relevant to a query that was given in Chapter 6 needs to be modified to take contestations into account. Before we do that we need to modify the definition of $Atoms(R)$ to take into account rules with *cap* functions in their bodies. This is done in the next definition.

Definition 8.3.2 *Let R be the rule $a \leftarrow b_1, \ldots, b_n, cap_i(B, a)$. Then $Atoms(R) = \{a, b_1, \ldots, b_n\}\ \cup\ Atoms(B)$.*

Now we are in a position to redefine the idea of a rule *relevant* to a query.

Definition 8.3.3 *A rule $R \in P + \mathcal{C}$ is relevant to answering a query l, where l is an atom, iff*

- *$l \in Atoms(R)$, or*
- *there is an atom p such that p is relevant to answering l and $p \in Atoms(R)$, where any atom p is relevant to answering any atom l if and only if $p \in Atoms(R_i)$ where R_i is relevant to answering l.*

Although a query L can be a conjunction or disjunction of literals, the definition of a relevant rule above in terms of an atomic query will still serve our purpose because we assume that we add the rule $query \leftarrow L$ to the program and answer the original query L by answering the query $query$.

The procedure $Proc$ stated below is the same procedure as the weak entailment proof procedure for logic programs without contestations. We assume that the checks for consistency, stability and verifiedness are made using the $TransCon$ procedure given above.

$Proc(N)$

1. If $\Gamma(N)$ is unstable or inconsistent or unverifiable or $(\Pi_1(\Gamma(N)) \neq \emptyset$ and $(\Pi_4(\Gamma(N)) = \emptyset$ or N has no unvisited descendants$))$ then if Parent(N) = nil then RETURN nil else *Proc*(Parent(N))

2. else if $\Pi_1(\Gamma(N)) = \emptyset$ then RETURN $\Gamma(N)$

3. else

4.**begin**

5. Create unvisited descendant N'

6. $Status(N') \leftarrow\ visited$

7. $Parent(N') \leftarrow N$

8.$Proc(N')$

9. **end**

The procedure *Proc* is invoked by the procedure *MasterStable*, which is the same procedure as the procedure of that name in Chapter 6.

The algorithm *MasterStable* creates the starting node using *CreateNode* and invokes $Proc$, which does all the real work.

$MasterStable(P,\ lit)$

$SN \leftarrow CreateNode(P,\ lit)$

Parent$(SN) \leftarrow$ nil

$Proc(SN)$

As in the chapter on the weak entailment proof procedure for logic programs without contestations, the procedure *MasterStable* is invoked by the procedure *MainCW*. In this procedure we assume that the contestations are compiled into the program.

$MainCW(P, \mathcal{C}, L)$

1. $P\mathcal{C} \leftarrow P$ with $\mathcal{C}$ compiled into it.

2. $P\mathcal{C} \leftarrow P\mathcal{C} \cup \{query \leftarrow L\}$

3. If $MasterStable(P\mathcal{C}, \mathbf{not}\ query) \neq$ nil then Return NO

4. else if $MasterStable(P\mathcal{C}, query) \neq$ nil then Return YES

5. else Return "Program has no canonical C-Stable models"

Example 8.3.2 *Let* $P + \mathcal{C}$ *be as in Example 8.2.3. Let query be* $a \vee \mathbf{not}\ b$. *Procedure* $MainCW$ *begins by adding the rule* $query \leftarrow a \vee \mathbf{not}\ b$ *to* $P + \mathcal{C}$. *It next executes* $MasterStable(P\mathcal{C}, \mathbf{not}\ query)$, *which returns nil. So next* $MasterStable(P\mathcal{C}, query)$ *is executed which returns a stable, consistent and verified node containing the assumptions* $\{\underline{query}, \underline{\mathbf{not}\ b}\}$ *and the inference set consisting of*

$\{query^{\mathbf{not}\ b\ \vee\ \mathbf{not}\ cap_3(c,\ d) \wedge b \wedge cap_1(d \wedge \mathbf{not}\ f,\ a) \wedge cap_2(e,\ a)\ \vee\ b \wedge cap_1(d \wedge \mathbf{not}\ f,\ a) \wedge cap_2(e,\ a)}$,
f^{true}, $c^{\mathbf{not}\ b}$, $e^{\mathbf{not}\ b}$, $d^{cap_3(c,\ d)}$, $b^{\mathbf{not}\ cap_3(c,\ d) \wedge b}$,
$a^{\mathbf{not}\ cap_3(c,\ d) \wedge b \wedge cap_1(d \wedge \mathbf{not}\ f,\ a) \wedge cap_2(e,\ a)\ \vee\ b \wedge cap_1(d \wedge \mathbf{not}\ f,\ a) \wedge cap_2(e,\ a)}\}$. *Thus,* $MainCW$ *returns YES to the original query.*

Strong Entailment

The proof procedure for strong entailment will be very similar to the proof procedure of the previous section. Unfortunately, this proof procedure cannot be as simple as the strong entailment proof procedure of Chapter 7 for normal logic programs without contestations. That proof procedure depended on a translation algorithm, $TransStrong$, which assigned T or F only to those inferences in a leaf node which could be inferred independently of any assumptions. This ensured that $TransStrong$ assigns T or F to only those atoms that would be assigned T or F in every well-supported model. Thus, the strong entailments of the program can be determined just in terms of the model constructed by $TransStrong$. However, in the case of logic programs with contestations an atom may be inferred on the basis of its contestor having a certain truth value, and it may have that truth value on the basis of an assumption. Thus, the translation algorithm may be forced to assign T or F to an atom which is not assumption free. Thus, the model computed by the translation algorithm can assign T or F to an atom, which may have a different truth value in other well-supported models of the program. Furthermore, the simplicity of the proof procedure of Chapter 7 depended on abstracting away the difference between CF and CT. But in the case of contestations we cannot abstract away the difference between CF and CT because the underlying *cap* function on which the contestation is based may return different values for CF and CT. The following example illustrates these points.

Example 8.3.3 *Let* P *be the ground program* $\{a \leftarrow;\ b \leftarrow \mathbf{not}\ c;\ c \leftarrow \mathbf{not}\ b\}$. *Let* $\mathcal{C} = \{b \hookrightarrow_1 a\}$. *Then given the assumption* **not** b *we can infer*

$$\{c^{\textbf{not}\ b},\ b^{\textbf{not}\ \textbf{not}\ b},\ a^{cap_1(b,a)}\}$$

Clearly, in any translation algorithm the value assigned to a should depend on the value assigned to b. But the value assigned to b depends on **not** *b being an assumption. For instance, $TransCon$ would assign CF to b and would thus assign T to a. Thus, the assignment of T to a cannot be independent of any assumptions. Furthermore, the value assigned to a would have to be different, given the definition of the cap_1 function, if b had been assigned CT. This illustrates that in this context we cannot abstract away from the difference between CF and CT.*

The strong entailment proof procedure for logic programs with contestations will first look for a canonical model in which *query* is F or CF. It will return NO if it finds such a model, else it looks for a model in which *query* is CT. If it finds such a model then it returns NO, else it looks for a model in which *query* is T. If it finds such a model then it returns YES, else it returns the message "Program has no C-stable models."

The procedure looks for a model in which *query* is F or CF by running procedure $MasterStable(P', \textbf{not}\ query)$, where P is the original program P with the contestations in $\mathcal{C}$ compiled into the rules of P. The procedure looks for a model in which *query* is CT by running procedure $MasterStableC(P', query)$, which is the same as the procedure $MasterStable$ except that instead of invoking the procedure $Proc$ it invokes procedure $ProcCheck$ defined below, which is the same as the procedure $Proc$ with an additional check which checks each node N_i to see if in $TransCon(N_i)$ *query* is T. If N_i has this property then $ProcCheck$ backtracks. This ensures that if $ProcCheck$ returns a node N, then in the model $TransCon(N)$

$query$ is CT. The main procedure then looks for a model in which $query$ is T by running $MasterStable(P', query)$. If a node N is returned by $MasterStable$ at this point, we can infer that in $TransCon(N)$ $query$ must be T as we have already ruled out the possibility of a canonical model in which $query$ is CT.

Procedure $MainCS(P, \mathcal{C}, L)$

1. $P\mathcal{C} \leftarrow P$ with $\mathcal{C}$ compiled into it

2. $P\mathcal{C} \leftarrow P\mathcal{C} \cup \{query \leftarrow L\}$

3. If $MasterStable(P\mathcal{C}$, **not** $query) \neq$ nil then Return NO

4. else if $MasterStableC(P\mathcal{C}, query) \neq$ nil then Return NO

5. else if $MasterStable(P\mathcal{C}, query) \neq$ nil then Return YES

6. else Return "Program has no canonical C-Stable models"

Procedure $MainCS$ uses procedure $MasterStableC$ which creates the starting node and invokes procedure $ProcCheck$. Procedures $MasterStable$ and $ProcCheck$ are described below.

$MasterStableC(P, lit)$

1. $SN \leftarrow CreateNode(P, lit)$
2. Parent$(SN) \leftarrow$ nil
3. $ProcCheck(SN)$

Procedure $MasterStableC$ uses procedure $ProcCheck$ which is described below.

$ProcCheck(N)$

1. If $\Gamma(N)$ is unstable or inconsistent or unverifiable or $(\Pi_1(\Gamma(N)) \neq \emptyset$ and $(\Pi_4(\Gamma(N)) = \emptyset$ or N has no unvisited descendants)) or $TransCon(\Gamma(N))(query) = T$ then if Parent(N) = nil then RETURN nil else *Proc*(Parent(N))

2. else if $\Pi_1(\Gamma(N)) = \emptyset$ then RETURN $\Gamma(N)$

 3. else

 4.**begin**

 5. Create unvisited descendant N'

 6. $Status(N') \leftarrow\ visited$

 7. $Parent(N') \leftarrow N$

 8.$Proc(N')$

 9. **end**

Example 8.3.4 *Let* $P+\mathcal{C}$ *be as in Example 8.2.3. As in Example 8.3.2, let query be* $a\ \vee$ **not** b*. But now we are trying to determine whether the query is strongly entailed by* $P+\mathcal{C}$*. Procedure* $MainCS$ *begin by adding the rule* $query \leftarrow\ a \vee \textbf{not}\ b$ *to* $P + \mathcal{C}$*. It next executes* $MasterStable(P\mathcal{C}, \textbf{not}\ query)$*, which returns nil. So next* $MasterStableC(P\mathcal{C},\ query)$ *is executed which returns a stable, consistent and verified node containing the assumptions* $\{\underline{query},\ \underline{\textbf{not}\ b}\}$ *and the inference set consisting of*

$\{query^{\textbf{not}\ b\ \vee\ \textbf{not}\ cap_3(c,\ d) \wedge b \wedge cap_1(d \wedge \textbf{not}\ f,\ a) \wedge cap_2(e,\ a)\ \vee\ b \wedge cap_1(d \wedge \textbf{not}\ f,\ a) \wedge cap_2(e,\ a)}$, f^{true}, $c^{\textbf{not}\ b}$, $e^{\textbf{not}\ b}$, $d^{cap_3(c,\ d)}$, $b^{\textbf{not}\ cap_3(c,\ d) \wedge b}$,

$a^{\textbf{not } cap_3(c,\ d) \wedge b \wedge cap_1(d \wedge \textbf{not } f,\ a) \wedge cap_2(e,\ a)\ \vee\ b \wedge cap_1(d \wedge \textbf{not } f,\ a) \wedge cap_2(e,\ a)}\}$. *From Example 8.3.1 we know that* $TransCon$ *assigns* CF *to both* a *and* b. *Thus, query is assigned* CT. *Hence,* $MainCS$ *returns NO to the original query.*

8.4 Proofs

In this section we prove the correctness of the weak entailment proof procedure and the strong entailment proof procedure for logic programs with contestations. Given the close similarity of the weak entailment proof procedure for logic programs with contestations and logic programs without contestations, the proof of correctness of the weak entailment proof procedure will be very similar to the proofs of correctness of the weak proof procedure for logic programs without contestations. Many of the lemmas and theorems that appeared in those proofs will also appear here. Some of the proofs of these lemmas and theorems will have to be slightly modified because in this chapter we use slightly different rules of matching and for inferring superscripted literals.

First we prove the correctness of the weak entailment proof procedure for logic programs with contestations and then we prove the correctness of the strong entailment proof procedure for logic programs with contestations.

Let P be a ground, finite, normal logic program and let $\mathcal{C}$ be a set of ground contestations. First, we show that if the implicit graph of $P + \mathcal{C}$ contains a consistent, stable, verified leaf node then *MasterStable* will reach it. Second, we show that the transformation of such a node is a canonical C-stable model of $P + \mathcal{C}$. Third, we show that all canonical C-stable models of $P + \mathcal{C}$ are represented, as defined in Chapter 4, by a stable, consistent and verified node in the implicit graph.

Lemma 8.4.1 *Let P be a ground, finite, normal logic program and let $\mathcal{C}$ be a set of ground contestations. If the implicit graph of $P+\mathcal{C}$ contains a consistent, stable, verified leaf node then MasterStable will return that node.*

Proof: $P+\mathcal{C}$ is a finite grounded program. Thus, the implicit graph of $P+\mathcal{C}$ contains only a finite number of nodes. *MasterStable* does a depth-first search for a leaf node with the appropriate properties. But for a finite graph depth-first search is guaranteed to discover any node with any specified properties if there is such a node in the graph, provided the depth-first search procedure has the correct mechanisms for detecting the specified properties. In our case *MasterStable* invokes *Proc* to do the depth-first search, and *Proc* invokes *TransCon* to test for instability, inconsistency, and nonverifiability. Since the tests used to determine whether a node has these properties are directly based on the definition of these concepts, obviously *TransCon* is a correct mechanism for detecting these properties. Hence, if the implicit graph of $P+\mathcal{C}$ contains a consistent, stable, verified leaf node then *MasterStable* will return that node. ■

Lemma 8.4.2 *Let P be a ground, finite, normal logic program and let $\mathcal{C}$ be a set of ground contestations. If the implicit graph for $P+\mathcal{C}$ contains a consistent, verified leaf node N then $TransCon(N)$ is a well-supported model of $P+\mathcal{C}$.*

Proof: Let N be a consistent, and verified leaf node in the graph for $P+\mathcal{C}$.

First, we show $TransCon(N)$ is a model of $P+\mathcal{C}$. Assume by way of contradiction that $TransCon(N)$ is not a model of $P+\mathcal{C}$. So $P+\mathcal{C}$ must contain a rule

$$a \leftarrow \ body_1 \ \vee \cdots \vee \ body_m$$

such that

- Case 1: $TransCon(N)(a) = F$ and $TransCon(N)(body_1 \vee \cdots \vee body_m) > F$, or

- Case 2: $TransCon(N)(a) = CT$ or CF and $TransCon(N)(body_1 \vee \cdots \vee body_m) = T$.

Case 1: $TransCon$ assigns F to a only if $\Pi_3(N)$ contains $a^{\textbf{not}\ true}$. But this is possible only if each of $body_1, \ldots, body_m$ evaluates to $true^{\textbf{not}\ true}$ when matched with the assumptions and inferences of N and when all the *cap* functions in the bodies are evaluated. In this case $Trans$ would assign F to at least one literal in each of $body_1, \ldots, body_m$. Thus, $TransCon(N)(body_1 \vee \cdots \vee body_m) > F$ is not possible, and, hence, Case 1 is not possible.

Case 2: If $TransCon(N)(body_1 \vee \cdots \vee body_m) = T$ then there must be an $i \in 1, \ldots, m$ such that $TransCon(N)(body_i) = T$. So each literal $b_{i_j} \in body_i$ must be assigned T by $TransCon(N)$. So for each such literal b_{i_j}, if it is a *cap* function then it must evaluate to $true$ and if it is not a cap function then there must be a literal $b_{i_j}^{true}$ in the inferential part of N. Hence the inferential part of N would also contain a^{true}. Thus, $TransCon$ would assign T to a. Hence Case 2 is also not possible.

Thus, $TransCon(N)$ must be a model of $P + \mathcal{C}$.

Next we show that $TransCon(N)$ is a well-supported model of $P+\mathcal{C}$. The well-founded ordering on the atoms of $P+\mathcal{C}$ can be in terms of the earliest node N_i in the path from the starting node to the leaf node N in which an atom in the inferential part of N_i is *first* assigned a value greater than F by $TransCon$. This ordering must be well-founded because the generation of the nodes and the inferred atoms in each node are by process of bottom-up inference which monotonically enlarges the inferential part of nodes. Furthermore, since the assignment of a truth value to any literal is not greater than the truth value assigned to its superscript, the truth value assigned to a literal must be supported. ■

Lemma 8.4.3 *Let P be a ground normal logic program and $\mathcal{C}$ be a set of contestations. If the implicit graph for $P + \mathcal{C}$ contains a stable, consistent, verified leaf node N then $TransCon(N)$ is a well-supported C-stable model of $P + \mathcal{C}$.*

Proof: Assume that N is a stable, consistent, and verified leaf node. We have already shown that $TransCon(N)$ is a well-supported model. Since N is stable, the inferential part of N contains no literal a^S such that $S \models \textbf{not}\ a$ and S evaluates to at least CT in $TransCon(N)$. Thus, for every literal a^S $TransCon$ would assign a the truth value of S. Let S be the disjunction $S_1 \vee \cdots \vee S_n$. So there must be a rule R in $P + \mathcal{C}$

$$R \;=\; a \leftarrow \mathit{body}_1 \;\vee \cdots \vee\; \mathit{body}_n$$

such that the assumption and inferences of N are matched with the literals in the body of R the result is

$$a \leftarrow true^{S_1} \;\vee \cdots \vee\; true^{S_n}.$$

Clearly then each body_i would evaluate to the truth value that S_i evaluates to in $TransCon(N)$. Thus, $\mathit{body}(R)$ would evaluate to the maximum of

$$\{TransCon(S_1), \ldots, TransCon(S_n)\}.$$

But this is what S would evaluate to. Since a is given the truth value S evaluates to, a and $\mathit{body}(R)$ would have the same truth value in $TransCon(N)$. Thus, every

R in $P + \mathcal{C}$ would evaluate to T. Hence $TransCon(N)$ is C-stable. ■

As in the weak entailment proof procedure for logic programs without contestations, the proof procedure in this chapter presupposes that if *MasterStable* cannot find a consistent, verified and stable leaf node N such that $query$ (or, **not** $query$) is in the assumption or inference part of N then the program contains no canonical C-stable model $\mathcal{I}$ such that $\mathcal{I}(query) \geq CT$ (resp., $\mathcal{I}(\textbf{not}\ query) \geq CT$). Lemma 8.4.1 tells us that if the implicit graph for P contains a leaf node of that sort then *MasterStable* will find it. But we can have no assurance that if *MasterStable* does not find a leaf node of that sort then the program has no C-stable canonical model unless we can show that every C-stable canonical model is represented in the implicit graph. Ideally, we would like to prove that for each C-stable canonical model $\mathcal{I}$ of $P + \mathcal{C}$ there exists a leaf node N in the implicit graph for $P + \mathcal{C}$ such that $TransCon(N) = \mathcal{I}$. However, this claim would not be true of a model $\mathcal{I}$ which assigns only T or F to atoms because $TransCon$ also assigns CT or CF to atoms. Nevertheless, we show below in Lemma 8.4.4 that every C-stable canonical model is represented in the implicit graph in the sense of 'representation' defined in Definition 9.4.1 in Chapter 6.

Lemma 8.4.4 *Let P be a normal logic program and let $\mathcal{C}$ be a set of contestations. For each well-supported C-stable model $\mathcal{I}$ of $P + \mathcal{C}$ there exists a leaf node N in the implicit graph for $P + \mathcal{C}$ such that $TransCon(N)$ is congruent with $\mathcal{I}$ and thus N represents $\mathcal{I}$.*

Proof: Since the graph is implicit, a node N exists in the graph only if there is a path from the starting node, $\langle P, \{true\}, \emptyset, (HB_P \cup \textbf{not}\ HB_P)\rangle$, to N. Recall that in the path from the starting node to a leaf node each new node is generated by adding a new assumption to the result of applying the Γ' operator to the previous node and by performing some housekeeping operations. Let N be any leaf node in the graph such that the path from the starting node to N satisfies the following property: For any node N_i in the path, if its child in the path is obtained by adding an assumption l to the result of applying the Γ' operator to N_i, then l must be such that $\mathcal{I}(l) \geq CT$. That is, the path is generated using the strategy of making a new assumption l only if $\mathcal{I}(l) \geq CT$ unless, of course, there are no assumptions left to be made, in which case the new node is generated by applying the $Falsify$ operation to the program part of N_i.

We show below that a leaf node N reached by this strategy

1. is a stable, consistent and verified node, and
2. is such that $TransCon(N)$ is congruent with $\mathcal{I}$.

We prove that N is a stable, consistent and verified node and that $TransCon(N)$ is congruent with $\mathcal{I}$ by inductively proving that each node N_i in the path to N (including N) is consistent,stable and not unverifiable, and inductively proving that, for any atom a, if $a^S \in \Pi_2(N_i) \cup \Pi_3(N_i)$ and $TransCon(N_i)(S) \geq CT$ iff $\mathcal{I}(a) \geq CT$. Thus, N must be verified as well as being consistent and stable and N must represent $\mathcal{I}$. The induction is done in terms of the order in which the nodes appear in the path $N_0, \ldots, N_i, \ldots, N_n$, where N_0 is the starting node, $\langle P, \{true\}, \emptyset, (HB_P \cup \textbf{not } HB_P)\rangle$, and N_n is N.

Base Case: i = 0. Clearly, the starting node, N_0, is stable, consistent and not unverifiable. Similarly, since $\Pi_2(N_0) \cup \Pi_3(N_0) = \{true\}$ it is trivially true that if an atom $a^S \in \Pi_2(N_0) \cup \Pi_3(N_0)$ then $\mathcal{I}(a) \geq CT$ iff $TransCon(N_0)(S) \geq CT$.

Inductive Case: Assume that the claim is true for all N_k such that $k < i$. To show that the claim is true for N_i.

First, we show that if an atom $a^S \in \Pi_2(N_i) \cup \Pi_3(N_i)$ then $TransCon(N_i)(S) \geq CT$ iff $\mathcal{I}(l) \geq CT$. If $\underline{a} \in \Pi_2(N_i)$ (i.e., if a is an assumption) then by the strategy for selecting assumptions it follows that $\mathcal{I}(a) \geq CT$ and since $\underline{a}$ is understood to have the superscript $true$, the claim follows trivially.

Suppose, therefore, that $a^S \in \Pi_3(N_i)$ (i.e., a^S is an inference). If $a^S \in \Pi_3(N_k)$, where $k < i$, then the claim is true by the inductive assumption. Suppose therefore that $a^S \notin \Pi_3(N_k)$, for any $k < i$. So a^S must occur in some iteration of the T'^P operator as applied to $\Pi_2(N_{i-1}) \cup \Pi_3(N_{i-1})$. Let $S = S_1 \vee \cdots \vee S_n$.

Either $TransCon(N_i)(S) \geq CT$ or not. If $TransCon(N_i)(S) \geq CT$ then there is a disjunct S_k in S such that $TransCon(N_i)(S_k) \geq CT$. So $P + \mathcal{C}$ must contain a rule

$$a \leftarrow \mathit{body}_1 \vee \cdots \vee \mathit{body}_k \vee \cdots \vee \mathit{body}_n$$

such that evaluating the cap functions in body_k and matching the literals in body_k with the assumptions and inferences of N_{i-1} results in $true^{S_k}$. So corresponding to each a_j in $\mathit{Atoms}(\mathit{body}_k)$ there must be a literal $a_j^{S_{k_j}}$ in an earlier iteration of $T'^P(\Pi_2(N_{i-1}) \cup \Pi_3(N_{i-1}))$ and each such S_{k_j} must evaluate to at least CT in $TransCon(N_i)$. Thus, by the inductive assumption $\mathcal{I}(a_j) \geq CT$ for all such a_j. Thus, $\mathcal{I}(\mathit{body}_k) \geq CT$. Since $\mathcal{I}$ is C-stable, it follows therefore that $\mathcal{I}(a) \geq CT$.

If it is not the case that $TransCon(N_i)(S) \geq CT$, then $TransCon(N_i)(S) \leq CF$. In that case each disjunct S_k in S is such that $TransCon(N_i)(S_k) \leq CF$. So at least one member S_{k_j} of each S_k must evaluate to CF or less in $TransCon(N_i)$ and at least one literal $l_{k_j}^S$ from each body_k must be in an earlier iteration of $T'^P(\Pi_2(N_{i-1}) \cup \Pi_3(N_{i-1}))$. Thus, by the inductive assumption $\mathcal{I}(l) \leq CF$. So

$\mathcal{I}(body_k) \leq CF$ for all k from 1 to n. Since $\mathcal{I}$ is well-supported, then $\mathcal{I}(a) \leq CF$.

By a similar argument it is easy to see that the same remarks apply to any literal that belongs to any iteration of the T^P operator as applied to $\Pi_2(N_{i-1}) \cup \Pi_3(N_{i-1})$.

Hence, we have shown that if $a^S \in \Pi_2(N_i) \cup \Pi_3(N_i)$ then $TransCon(N)(S) \geq CT$ iff $\mathcal{I}(a) \geq CT$.

Second, we show that N_i is not unverifiable. Let $\underline{a}$ be a positive assumption in N_i. So $\mathcal{I}(a) \geq CT$. But then if a^S is in $\Pi_3(N_i)$ then S cannot evaluate to less than CT in $TransCon(N_i)$ otherwise, as we have shown above, $\mathcal{I}(a) \leq CF$. But a cannot both be greater and lesser than CT in $\mathcal{I}$. Thus, N_i is not unverifiable.

Third, we show that N_i is consistent. Let **not** a be a negative assumption in $\mathcal{I}$. So $\mathcal{I}(\textbf{not } a) \geq CT$. Assume by way of contradiction that $a^S \in \Pi_3(N_i)$ and $TransCon(N_i)(S) \geq CT$ and $S \not\models \textbf{not } a$. But then, as we have shown by the inductive proof above, $\mathcal{I}(a) \geq CT$. But both a and **not** a cannot be CT or greater in $\mathcal{I}$. Thus, N_i is consistent.

Fourth, we show that N_i is stable. Suppose by way of contradiction that N_i is not stable. So there exists an $a^S \in \Pi_3(N_i)$ such that $S \models \textbf{not } a$ and $TransCon(N_i)(S) \geq CT$. By the inductive proof above in that case $\mathcal{I}(a) \geq CT$. However, if $S \models \textbf{not } a$ then **not** a is in S, and so **not** a must be an assumption and hence $\mathcal{I}(\textbf{not } a) \geq CT$. But this is a contradiction. Hence N_i must be stable.

This completes the inductive step. Thus, we have shown by induction that the leaf node N is stable, consistent and not unverifiable, and such that if $a^S \in \Pi_2(N) \cup \Pi_3(N)$ and $TransCon(N)(S) \geq CT$ iff $\mathcal{I}(a) \geq CT$. It remains to be shown that N is verified.

We know that N is not unverifiable. This means that for any positive assumption $\underline{a}$, there is no inference a^S in N such that $TransCon(N)(S) \leq CF$. But since $\Pi_1(N)$ (the program part) is empty, it follows that for any atom $a \in HB_P$, a^S, where S is some superscript, belongs in $\Pi_3(N)$. Either $TransCon(N)(S) \leq CF$ or $TransCon(N)(S) \geq CT$. But as remarked above it cannot be the case that $TransCon(N)(S) \leq CF$ if $\underline{a}$ is an assumption and a^S is in $\Pi_3(N)$. So in that case $TransCon(N)(S) \geq CT$. So each positive assumption is verified in N and thus N is verified.

We show that N represents $\mathcal{I}$. Since N is stable, consistent and verified, for any $a^S \in \Pi_3(N)$, $TransCon(N)(a) = TransCon(N)(S)$. As shown in the inductive proof above, $Transcon(N)(S) \geq CT$ iff $\mathcal{I}(a) \geq CT$. This implies that $TranCon(N)(a) \geq CT$ iff $\mathcal{I}(a) \geq CT$. Which implies that $TranCon(N)(a) \leq CF$ iff $\mathcal{I}(a) \leq CF$. So $TransCon(N)$ is congruent to $\mathcal{I}$, and thus N represents $\mathcal{I}$.

■

Now we are in a position to prove the correctness of the main algorithm.

Theorem 8.4.1 *Let P be a normal logic program, $\mathcal{C}$ be a set of contestations, and let L be a query to $P + \mathcal{C}$. If MainCW(P, $\mathcal{C}$, L) returns "NO" then $P + \mathcal{C}$ does not weakly entail L, if MainCW(P, $\mathcal{C}$, L) returns "YES" then $P + \mathcal{C}$ weakly entails L, and if MainCW(P, $\mathcal{C}$, L) returns "Program has no canonical C-stable models" then $P + \mathcal{C}$ has no canonical stable models.*

The proof of the above theorem is entirely analogous to the proof of the corresponding theorem about the correctness of the weak entailment proof procedure for logic programs without contestations (Theorem 6.4.2).

Next we prove the correctness of the strong entailment proof procedure for normal logic programs with contestations.

Theorem 8.4.2 *Let P be a normal logic program, $\mathcal{C}$ be a set of contestations, and let L be a query to $P + \mathcal{C}$. If MainCS(P, $\mathcal{C}$, L) returns "NO" then $P + \mathcal{C}$ does not strongly entail L, if MainCS(P, $\mathcal{C}$, L) returns "YES" then $P + \mathcal{C}$ strongly entails L, and if MainCS(P, $\mathcal{C}$, L) returns "Program has no canonical C-stable models" then $P + \mathcal{C}$ has no canonical stable models.*

Proof: Assume *MainCS*(P, $\mathcal{C}$, L) returns "NO". This implies that either $MasterStable(P\mathcal{C},$ **not** $query) \neq nil$ or $MasterStableC(P\mathcal{C},\ query) \neq nil$. If the former then $Proc$ has returned a stable, consistent and verified node N such that $TransCon(N)(query) \leq CF$, and by Lemma 8.4.1 we know that the implicit graph of $P + \mathcal{C}$ contains such a node. Since $TransCon(N)$ is a model of $P + \mathcal{C}$, clearly $TransCon(N)(L) \leq CF$. But since $TransCon(N)$ is a canonical model of $P + \mathcal{C}$ (by Lemma 8.4.3), it follows that $P + \mathcal{C}$ cannot strongly entail L.

On the other hand if it is the case that $MasterStable(P\mathcal{C},$ **not** $query)$ returns nil and $MasterStableC(P\mathcal{C},\ query)$ returns a non-nil node, then we know that $ProcCheck$ has returned a stable, consistent and verified node N such that $TransCon(N)(query) = CT$. By reasoning similar to the proof of Lemma 8.4.1 we know that the implicit graph of $P + \mathcal{C}$ must contain such a node. This is because procedure $ProcCheck$ is just like procedure $Proc$ except that it contains an additional check to determine for any node N, which need not be a leaf node, whether $TransCon(N)(query) = T$. Thus, if procedure $Proc$ correctly returns a node with specified properties then by similar reasoning procedure $ProcCheck$ also operates correctly. Thus, we can assume that if $ProcCheck$ has returned a stable, consistent and verified node N such that $TransCon(N)(query) = CT$ then the implicit graph of $P + \mathcal{C}$ contains such a node. Since $TransCon(N)$ is a model of $P+\mathcal{C}$, clearly $TransCon(N)(L) = CT$. But since $TransCon(N)$ is a canonical model of $P + \mathcal{C}$ (by Lemma 8.4.3), it follows that $P + \mathcal{C}$ cannot strongly entail L since L is less than T in a canonical model.

Assume instead that *MainCS*(P, $\mathcal{C}$, L) returns "YES". This implies that *MasterStable*($P\mathcal{C}$, *query*) returns a non-nil node at step 5 of $MainCS$. *MainCS* reaches step 5 only if at step 3 it fails to find a consistent, verified and stable node N such that $TransCon(N)(query) \leq CF$ and such that at step 4 it fails to find a stable, consistent and verified node N such that $TransCon(N)(query) = CT$. As proven in the first part of this proof this means that the implicit graph of $P+\mathcal{C}$ contains no consistent, verified and stable node N such that $TransCon(N)(query) \leq CF$ or in which $TransCon(N)(query) = CT$. By the converse of Lemma 8.4.4, it follows therefore that $P + \mathcal{C}$ contains no C-stable models in which *query* is F or CF or CT. Thus either $P + \mathcal{C}$ has no C-stable models or *query* is T in all its C-stable models. But since at step 5 *MasterStable*($P\mathcal{C}$, *query*) returns a stable, consistent and verified node N, it follows from Lemma 8.4.3 that $TransCon(N)$ is a C-stable model of $P + \mathcal{C}$. Hence, *query* must evaluate to T in all the C-stable models of $P + \mathcal{C}$. But since all such models are well-supported, it follows that L must be T in all the C-stable models of $P+\mathcal{C}$. However, if $P+\mathcal{C}$ has any C-stable models, then all its canonical models are C-stable. Thus it follows that $P + \mathcal{C}$ strongly entails L.

Assume instead that $MainCS$ returns "Program has no C-stable models". This implies that at steps 3, 4, and 5 the program failed to find consistent, verified, and a stable node N such that $TransCon(query)$ is F or CF or CT or T. This means that $P + \mathcal{C}$ has no C-stable models in which *query* has one of these truth values.

But since these are all the possible truth values, it follows that $P + \mathcal{C}$ has no C-stable models. ■

8.5 Discussion

In this chapter we have described two proof procedures. The first proof procedure is for answering whether a query is *weakly* entailed by a finite and ground normal logic program augmented with a set of ground contestations. The second proof procedure is for answering whether a query is *strongly* entailed by a finite and ground normal logic program augmented with a set of ground contestations. We have proven that these proof procedures for sound and complete with respect to the **C4** semantics for normal logic programs augmented with contestations. These proof procedures are restricted to programs augmented with contestations which have at least one well-supported C-stable model. For a program augmented with contestations which lacks a well-supported C-stable model, the proof procedures terminate by sending a message saying that the augmented program lacks a C-stable model.

As in Chapter 6, we will analyze the worst case complexity of the proof procedures in terms of the number of matching operations. Note that determining whether a *cap* function is known does not require the proof procedure to perform any matching operations. Recall that a *cap* function such as $cap_i(B, a)$ is considered known in a node when each literal in B or its negation occurs in the inference or the assumption part of that node. Thus, contestations compiled into rules do not require any additional matching operations for expanding a node. Hence, the worst-case complexity for answering whether a query is weakly entailed by a finite and ground normal logic programs with contestations is exactly the same as for finite and ground normal logic programs without any contestations. In Chapter 6 we determined that this complexity is $O(n^2 \times 2^n)$, where n is the cardinality of the Herbrand base of the input program.

The worst-case complexity for the procedure for determining whether a query is *strongly* entailed by finite and ground normal logic program augmented with contestations will be exactly be the same as the proof procedure for weak entailment since the strong entailment procedure consists in running three times essentially the same procedure as is the case of weak entailment. Thus, the worst case cost of answering whether a query is strongly entailed by a program augmented with contestations is much more expensive than answering whether the same query is strongly entailed by a program *without* contestations, which we determined in Chapter 7 to be $O(n^3)$.

8.6 Summary

In this chapter we have provided a proof procedure for answering ground queries, which can be a disjunction or conjunction of literals, to a ground and finite normal logic program augmented with a set of heterogeneous ground contestations. The research contributions of this chapter are summarized in this section.

- We introduce a way of compiling contestations into the bodies of the rules of a program.

- We have developed the formal apparatus and algorithms for computing a canonical model of a program with contestations compiled into it in which a specified literal is true by making assumptions and inferring literals on the basis of these assumptions and the input program (Section 8.3).

- We have devised a procedure which utilizes this apparatus and algorithms for determining whether a ground query is weakly entailed or strongly entailed by the input program (Section 8.3).

- We have proved the soundness and completeness of this procedure with respect to the **C4** semantics for normal logic programs augmented with contestations (Section 8.4).

- We have proved that the worst-case complexity of this procedure is $O(n^2 \times 2^n)$ for both weak and strong entailment, where n is the cardinality of the Herbrand base of the program (Section 8.5).

Chapter 9

Integrity Constraints and Contestations

9.1 Introduction

In this chapter we use our semantics of logic programs with contestations to develop a semantics for deductive databases that violate their integrity constraints. Standardly, databases are supposed to satisfy their integrity constraints. Violation of an integrity constraint by a database is regarded as a system failure. The mechanism for ensuring that a database satisfies its integrity constraints is a layer of software in the database management system ($DBMS$) that blocks updates to the database that would result in the violation of any constraints. This layer of software is commonly called the transaction manager of the DBMS. This model of transaction management is carried over to the multi-database setting ([BGMS92]). The *global* transaction manager is granted the authority to block *global* updates which would violate global constraints as well as *local* updates which would violate *global* constraints.

However, there are contexts in which the transaction manager may not have this authority ([AKWS95]). Thus, there can be loosely coupled multi-database systems in which the global transaction manager does not have the authority to block a local transaction at one of the participating databases which would result in the violation of the global integrity constraints without any violation of the local constraints. In this case the global database state (the naive union of all the local database states) would violate the global integrity constraints. Furthermore, in integrating information from different sources an agent or a mediator may have the capacity to draw information from different sources without having any transaction management facilities. Thus, in this context the information in the integrator may conflict with the integrator's integrity constraints. That is, the state of the integrator's database may violate the integrity constraints. But in this case the integrator should be able to reason using the information at hand even though the information contains conflicts. This again creates a need for a semantics of databases that violate their integrity constraints.

Thus, it is necessary to reconsider the relation of integrity constraints to databases. If in the sorts of contexts described above integrity constraints do not play the role of constraining the state of the database, then what role can they play? We propose that integrity constraints be viewed as constraints on what can be inferred from the database as opposed to constraints on the state of the database. We propose that even if the state of a database violates its integrity constraints, nevertheless we can constrain what can be inferred from the database so that the inferred information always satisfies the integrity constraints. Ensuring that the state of the database satisfies the constraints is just one way of ensuring that what can be inferred from the database satisfies the constraints. We show below how **C4** can be used to give an account of integrity constraint satisfaction in which the information inferred from the database can satisfy the constraints even when the state of the database does not.

In Section 9.2 we describe an entailment relation such that the set of entailments of a database in terms of this entailment relation satisfies the integrity constraints. In Section 9.3 we show how to represent a wide range of integrity constraints in terms of contestations. Thus, a deductive database with integrity constraints can be viewed as a logic program augmented with contesations. In Section 9.4 we show how the semantics we have developed in Chapter 4 for normal logic programs augmented with contestations can provide a semantics for deductive databases augmented with a wide range of integrity constraints. In terms of this semantics we define the entailment relation described in Section 9.2 and prove that the entailment relation thus defined can provide a satisfactory account of integrity constraint satisfaction in which the information inferred from the database can satisfy the constraints even when the state of the database does not. In Section 9.5 we discuss the merits of our approach and compare it to related work. In Section 9.6 we summarize the main research contributions of this chapter.

9.2 Preliminaries

A deductive database (DB) consists of two parts: a set of facts and a set of rules. The set of facts in a database is called the *extensional* database (EDB) and the set of rules in a database is called the *intensional* database (IDB). Rules in deductive databases allow implicit facts to be derived. Thus, the extensional and the intensional parts of the DB together explicitly and implicitly specify all the information contained in the DB. The facts are always ground atoms. If the rules of a DB contain no negative literal in the bodies, then the DB is called a Horn database if the head of the rule has at most one atom. Furthermore, Horn databases that contain no function symbols in the facts or the rules are called

Datalog databases. In this chapter we assume that databases are function-free, but the rules may contain negative literals in the bodies. Thus, the databases we consider in this chapter are function-free normal logic programs.

In addition to the information explicitly and implicitly contained in the DB, it is common for databases to have integrity constraints (ICs). The role of ICs is to further specify what information is contained in the DB. This is done not by adding further facts or rules which can be used to specify more information, but by further characterizing the information already specified in the DB. This can be done by specifying the relation between certain predicates, or by specifying the range of values that a certain variable can take, or by specifying which combinations of information cannot occur together or must occur together, or what cannot count as legitimate information from the point of view of the database. Some examples of integrity constraints associated with a company's database might be "All managers must be employees," or "Salary cannot be less than 0," or "No employee can be a contractor." Thus, integrity constraints delimit or constrain the possible ways in which the information in the database can be interpreted. This has led some writers to view ICs as specifying the semantics of the database. However, the term "semantics" is also used to describe the model theory of the facts and the rules in the database, which determines what information can be correctly viewed as implicitly contained in the database. Thus, in the context of deductive databases it is unwise to characterize ICs as specifying the semantics of the database as it obscures the difference between the issue of what information is implicitly (and explicitly) contained in the database and the issue of how that information is to be interpreted.

Clearly, it is desirable for a DB to satisfy its integrity constraints since the

ICs are meant to confer meaningfulness on the information contained in the DB. This raises the issue of what it means for a DB to satisfy ICs.

Traditionally, it is the *state* of the DB that is supposed to satisfy the ICs. In the case of relational databases it is relatively easy to specify what counts as the state of a database: it is the set of tuples contained in all the tables in the database. Because some of the information in a deductive database is contained implicitly, which is to be made explicit by making all possible inferences from the extensional and intensional parts of the database, specifying what counts as the state of a deductive database is a more complex matter. We do that below.

The extensional and intensional part of a DB together constitute a normal logic program. We associate a specific semantics SEM to DB, where SEM can be any semantics for normal logic programs such as the stable model semantics, or the well-founded semantics, or **C4**. Let $\models_{SEM}$ be the entailment relation defined in terms of the chosen semantics SEM. In terms of $\models_{SEM}$ we define the state of a deductive database as follows.

Definition 9.2.1 *Let DB be a deductive database and let SEM be its chosen semantics. Then the state of DB, relative to SEM, is $CONT_{SEM}(DB)$ which is defined as $CONT_{SEM}(DB) = \{l \mid DB \models_{SEM} l\}$, where l is a literal.*

Thus, $CONT_{SEM}(DB)$ is the set of literals that can be inferred from the database relative to a chosen semantics SEM. The state of DB has to be specified relative to a chosen semantics, and since different semantics for normal logic programs are not equivalent in terms of the consequences they legitimize from a program, it follows that the state of DB can vary depending on the chosen semantics. This does not happen in the case of Datalog databases because such databases are essentially definite logic programs and the different semantics for definite logic programs coincide in terms of the set of consequences they legitimize.

Following the traditional perspective, deductive database theorists have also held that it must be state of the database that satisfies its integrity constraints. There are two well-known theories of integrity constraint satisfaction in the deductive database literature. The *entailment* theory of integrity constraint satisfaction ([Rei84]) holds that a database DB satisfies an integrity constraint IC just in case $DB \models IC$. The *consistency* theory of integrity constraint satisfaction ([Kow78]) holds that a database DB satisfies an integrity constraint IC just in case $DB \cup IC$ is consistent.

Example 9.2.1 *Let* $DB = \{a \leftarrow \textbf{not } b;\ b \leftarrow \textbf{not } a\}$*. Let* $IC = \{\textbf{not } a\}$*. Let* SEM *be stable model semantics. Then on the entailment theory of integrity constraint satisfaction* DB *does not satisfy* IC*. However, on the consistency theory of integrity constraint satisfaction* DB *satisfies* IC*.*

It is clear that if a DB satisfies its ICs on the entailment theory then it satisfies those ICs on the consistency theory. But the converse does not hold. Hence, the demands that a set of ICs make on a DB are more stringent on the entailment theory than on the consistency theory of IC satisfaction. In the rest of this chapter we assume the entailment theory of integrity constraint satisfaction.

In the sorts of contexts described in the introductory section of this chapter the *state* of a database may violate its integrity constraints. Nevertheless we want the integrity constraints to play a role in interpreting the information contained in the database since we recognize that integrity constraints can encode valuable information about the domain of the database. In this chapter we seek an account of integrity constraint satisfaction that views such constraints not on the state of the database but on what can be inferred from the database. But since any inferential powers of a database must be sound with respect to an entailment relation, we must formulate an entailment relation that supports such a revised account of integrity constraint satisfaction.

Formally speaking, what is required is an entailment relation $\approx\!\!\!\!\mid$ such that the set of consequences of DB, relative to a set of IC, in terms of $\mid\!\approx$, are guaranteed to satisfy IC. Let $CONS_{IC}(DB) = \{l \mid DB \mid\!\approx l\}$, where l is a literal. Then the requirement that the information inferred from a database DB, with the associated integrity constraints IC, should satisfy IC can be reformulated as the requirement that $CONS_{IC}(DB)$ should satisfy IC. Since $CONS_{IC}(DB)$ is a set of literals we can say that $CONS_{IC}(DB)$ satisfies IC if it entails IC in the sense that each member of IC is true in $CONS_{IC}(DB)$. Note that the entailment relation $\mid\!\approx$ is so defined that the set of such entailments of a DB must satisfy IC, but this is not a requirement on the entailment relation $\models_{SEM}$, and thus not a requirement on $CONT_{SEM}(DB)$, for any choice of SEM. Clearly, any semantics on which $\mid\!\approx$ is based must be inferentially conflict-free with respect to the types of conflicts expressed by integrity constraints.

Example 9.2.2 *Let* $DB = \{p,\ q\}$. *Let* $IC = \{\textbf{not}\ p \lor \textbf{not}\ q\}$. *Then* $CONT_{SEM}(DB) = \{p,\ q\}$ *on any reasonable choice of* SEM. *Thus, the state of* DB *does not satisfy* IC. *However if a set of inferences from* DB *are to satisfy* IC *then either* $p \notin CONS_{IC}(DB)$ *or* $q \notin CONS_{IC}(DB)$. *So the entailment relation* $\mid\!\approx$ *must be such that either* $DB \not\mid\!\approx p$ *or* $DB \not\mid\!\approx q$.

Let us call any entailment relation *non-reflexive* if it is such that $p \in S$ but S does not entail p. The above example shows that $\mid\!\approx$ must be a non-reflexive entailment relation.

Since $\approx$ is an entailment relation, it must be based on a model theory. In the following we propose **C4** for normal logic programs with contestations as such a model theory. But this requires that integrity constraints should be represented in the language of contestations. We show how to do that in the next section. We restrict our account to ground constraints, and hence to the propositional case. However, the database need not be ground.

9.3 Representing integrity constraints

An integrity constraint such as:

No one is both a male and a female

is represented in deductive databases as $\leftarrow male(X), female(X)$ with the intended meaning that $male(X)$ and $female(X)$ cannot be simultaneously true of the same entity. However, since we are considering only propositional constraints, this constraint must be instantiated with respect to a specific entity. Thus, regarding some individual Pat the constraint says that both $male(Pat)$ and $female(Pat)$ cannot be true at the same time. So the constraint can naturally be divided into two parts:

- If $male(Pat)$ is true then $female(Pat)$ cannot be true, and
- If $female(Pat)$ is true them $male(Pat)$ cannot be true.

This suggests that the constraint can be represented by the set of contestations $\{male(Pat) \hookrightarrow_1 female(Pat);\ female(Pat) \hookrightarrow_1 male(Pat)\}$. More generally, we can represent a denial integrity constraint $\leftarrow a_1, \ldots, a_n$, where each a_i is a ground atom, by the following set of contestations:

$$\{a_1 \wedge \cdots \wedge a_{i-1} \wedge a_{i+1} \wedge \cdots \wedge a_n \hookrightarrow_2 a_i \mid i \in \{1, \ldots, n\}\}$$

As a limiting case when $n = 1$, i.e., when the constraint is $\leftarrow a_1$, we represent this as $true \hookrightarrow_1 a_1$, where $true$ is a special atom that always evaluates to T in all interpretations.

So far we have represented constraints containing only atoms. But of course constraints can also contain negative literals. Thus, there can be a constraint $\leftarrow p$, **not** q. This constraint can be represented as **not** $q \hookrightarrow_1 p$. However, it would be incorrect to represent this constraint by $p \hookrightarrow_1$ **not** q because contestations with a negative literal in the right hand side should be regarded as ill-formed. For a model to satisfy $p \hookrightarrow_1$ **not** q, the truth of p in that model must block the truth of **not** q in that model, which means that q must be true. But of course a contestation cannot by itself make some atom true in a model–at most, it can provide a cap on the truth value assigned to that atom in that model. That is, in a well-supported model an atom can be assigned true only if there is evidence *for* that atom which supports assigning it true– evidence *against* the negation of that atom cannot be construed as evidence for that atom. For this reason we regard a contestation of the form $p \hookrightarrow_1$ **not** q as ill-formed. Hence we suggest that a constraint of the form $\leftarrow p$, **not** q should be represented by the single contestation **not** $q \hookrightarrow_1 p$.

More generally, we can represent constraints of the form $\leftarrow l_1, \ldots, l_n$, where at least one l_i, $i \in \{1, \ldots, n\}$, is a positive literal, as follows. For the sake of simplicity of representation assume that all the positive and negative literals in the constraint are grouped separately with the negative literals starting at i. Then $\leftarrow l_1, \ldots, l_n$ can be represented by the following set of contestations:

$$\{l_1 \wedge \cdots \wedge l_{j-1} \wedge l_{j+1} \wedge \cdots \wedge l_n \hookrightarrow_1 l_j \mid j \in \{1, \ldots, n\}\}$$

This representation can be shown to be correct by the following lemma.

Lemma 9.3.1 *Let $\mathcal{M}$ be a set of* **C4** *interpretations. Let IC_1 be the ground constraint $\leftarrow l_1, \ldots, l_n$, where at least one l_i, $i \in \{1, \ldots, n\}$, is a positive literal. Let all the positive and negative literals in IC_1 be grouped separately with the negative literals starting at i. Then for any $\mathcal{I} \in \mathcal{M}$, IC_1 evaluates to at least CT in $\mathcal{I}$ if and only if $\mathcal{I}$ satisfies each contestation in*

$$\mathcal{C} \;=\; \{l_1 \wedge \cdots \wedge l_{j-1} \wedge l_{j+1} \wedge \cdots \wedge l_n \hookrightarrow_1 l_j \mid j \in \{1, \ldots, i\}\}$$

Proof: $\Rightarrow$

Let $\mathcal{I}$ be any member of $\mathcal{M}$. Assume that $\leftarrow l_1, \ldots, l_n$ evaluates to at least CT in $\mathcal{I}$. So there exists an l_k, $k \in \{1, \ldots, n\}$, such that $\mathcal{I}(l_k) \leq CF$. If $i \leq k \leq n$ then l_k is a negative literal, which occurs in the left hand side of every contestation in $\mathcal{C}$, and thus the left hand side of each member of $\mathcal{C}$ is at most CF in $\mathcal{I}$. In this case every member of $\mathcal{C}$ is trivially satisfied in $\mathcal{I}$. On the other hand, if $1 \leq k \leq i-1$ (that is, l_k is a positive literal), then $\mathcal{C}$ contains one contestation with l_k on its right-hand side and the other members of $\mathcal{C}$ have l_k on the left-hand side. The contestations with l_k on the left hand side are trivially satisfied in $\mathcal{I}$ because the left-hand side of each such contestation evaluates to at most CF in $\mathcal{I}$. The contestation with l_k on the right-hand side is also satisfied in $\mathcal{I}$ because its right-hand side is at most CF.

$\Leftarrow$

Assume that each member of $\mathcal{C}$ is satisfied in $\mathcal{I}$. Assume, by way of contradiction, that $\leftarrow l_1, \ldots, l_n$ does not evaluate to at least CT in $\mathcal{I}$. So each literal in $\{l_1, \ldots, l_n\}$ must evaluate to at least CT in $\mathcal{I}$. But this means, for instance, $l_2 \wedge \cdots \wedge l_n \hookrightarrow_1 l_1$ cannot be satisfied by $\mathcal{I}$. This contradicts the assumption that every member of $\mathcal{C}$ is satisfied in $\mathcal{I}$. Thus, $\leftarrow l_1, \ldots, l_n$ must evaluate to at least CT in $\mathcal{I}$. ■

This representation of integrity constraints can be generalized to non-denial integrity constraints such as $p \rightarrow q$, where both p and q are atoms. We understand $\rightarrow$ to be material implication. We understand the constraint as saying that $p \rightarrow q$ should be entailed by the database. This means that in any canonical model of the database p should be true only if q is true. So the constraint $p \rightarrow q$ can be represented by the contestation **not** $q \hookrightarrow_1 p$ because the contestation can be satisfied by any model if and only if in that model if **not** q is at least CT (and thus q is at most CF) then p is at most CF. So in any canonical model of the database p is true (T or CT) only if q is true (T or CT).

Following this line of thinking a constraint of the form $p \wedge q \rightarrow r$ can be understood as **not** $r \hookrightarrow_1 p \wedge q$. Although it is easy to give a semantics for contestations with a conjunction as the contested part, such contestations cause problems for the proof procedure given for normal logic programs with contestations. However, we can represent such contestations by the set $\{p \wedge \textbf{not } r \hookrightarrow_1 q;\ q \wedge \textbf{not } r \hookrightarrow_1 p\}$. It is easy to see that any **C4** interpretation satisfies the contestation **not** $r \hookrightarrow_1 p \wedge q$ if and only if it also satisfies the set $\{p \wedge \textbf{not } r \hookrightarrow_1 q;\ q \wedge \textbf{not } r \hookrightarrow_1 p\}$.

More generally, a constraint of the form $a_1 \wedge \cdots \wedge a_n \rightarrow a_{n+1}$, where all the a_i are atoms, can be represented by

$$\{a_1 \wedge \cdots \wedge a_{i-1} \wedge a_{i+1} \wedge \cdots \wedge a_n \wedge \textbf{ not } a_{n+1} \hookrightarrow_1 a_i \mid i \in \{1, \ldots, n\}\}$$

Generalizing this even further, a constraint of the form $a_1 \wedge \cdots \wedge\ a_n \rightarrow b_1 \vee \cdots \vee\ b_m$, where all the literals are atoms, can be understood as **not** $b_1 \wedge \cdots \wedge$ **not** $b_m\ \hookrightarrow_1\ a_1 \wedge \cdots \wedge\ a_n$. As seen above, this can be represented by the following set of contestation:

$$\{a_1\ \wedge \cdots \wedge a_{i-1}\ \wedge a_{i+1}\ \wedge \cdots \wedge a_n \wedge\ \textbf{not}\ b_1 \wedge \cdots \wedge\ \textbf{not}\ b_m\ \hookrightarrow_1\ a_i \mid i \in \{1, \ldots, n\}\}$$

The following lemma demonstrates the correctness of this representation of the above sorts of constraints.

Lemma 9.3.2 *Let $\mathcal{M}$ be a set of* **C4** *interpretations. Let IC_1 be the ground constraint $a_1 \wedge \cdots \wedge\ a_n \rightarrow b_1 \vee \cdots \vee\ b_m$, where all the literals are atoms. Then for any $\mathcal{I} \in \mathcal{M}$, IC_1 evaluates to at least CT in $\mathcal{I}$ if and only if $\mathcal{I}$ satisfies each contestation in*

$$\mathcal{C}\ = \{a_1\ \wedge \cdots \wedge a_{i-1}\ \wedge a_{i+1}\ \wedge \cdots \wedge a_n \wedge\ \textbf{not}\ b_1 \wedge \cdots \wedge\ \textbf{not}\ b_m\ \hookrightarrow_1\ a_i \mid i \in \{1, \ldots, n\}\}$$

Proof: $\Rightarrow$

Let $\mathcal{I}$ be any member of $\mathcal{M}$. Assume that IC_1 evaluates to at least CT in $\mathcal{I}$. Either $a_1 \wedge \cdots \wedge\ a_n$ is at least CT in $\mathcal{I}$ or it is at most CF in $\mathcal{I}$. If $a_1 \wedge \cdots \wedge\ a_n$ is at least CT in $\mathcal{I}$, then $b_1 \vee \cdots \vee\ b_m$ is at least CT in $\mathcal{I}$. So there exists a $j \in \{1, \ldots, m\}$ such that b_j is at least CT and so **not** b_j is at most CF in $\mathcal{I}$. But since **not** b_j occurs in the left-hand side of every member of $\mathcal{C}$, it follows that in this case the left-hand side of every member of $\mathcal{C}$ evaluates to at most CF, and thus every member of $\mathcal{C}$ is trivially satisfied in $\mathcal{I}$.

On the other hand if the left-hand side of IC_1 is at most CF then there exits an a_i, $i \in \{1, \ldots, n\}$, such that a_i is at most CF in $\mathcal{I}$. One member of $\mathcal{C}$ will have a_i in its right-hand side and the other members of $\mathcal{C}$ have a_i on the left-hand side. The contestations with a_i on the left hand side are trivially satisfied in $\mathcal{I}$

because the left-hand side of each such contestation evaluates to at most CF in $\mathcal{I}$. The contestation with a_i on the right-hand side is also satisfied in $\mathcal{I}$ because its right-hand side is at most CF.

$\Leftarrow$

Assume that each member of $\mathcal{C}$ is satisfied in $\mathcal{I}$. Assume, by way of contradiction, that IC_1 does not evaluate to at least CT in $\mathcal{I}$. So the left-hand side of IC_1 must evaluate to at least CT and the right-hand side of IC_1 must evaluate to at most CF in $\mathcal{I}$. But this means, for instance

$$a_2 \wedge \cdots \wedge a_n \wedge \textbf{not } b_1 \wedge \cdots \wedge \textbf{ not } b_m \hookrightarrow_1 a_1$$

cannot be satisfied by $\mathcal{I}$. This contradicts the assumption that every member of $\mathcal{C}$ is satisfied in $\mathcal{I}$. Thus, IC_1 must evaluate to at least CT in $\mathcal{I}$. ■

A constraint of the form $\textbf{not } p \rightarrow \textbf{ not } r$ can be represented by the contestation $\textbf{not } p \hookrightarrow_1 r$. Clearly, if r is blocked from being true in a model in which $\textbf{not } p$ is true, then $\textbf{not } p \rightarrow \textbf{ not } r$ is true in that model. More generally, a constraint of the form $l_1 \wedge \cdots \wedge l_n \rightarrow \textbf{ not } p$ can be represented by the contestation $l_1 \wedge \cdots \wedge l_n \hookrightarrow_1 p$.

The following lemma proves the correctness of this representation of such constraints.

Lemma 9.3.3 *Let $\mathcal{M}$ be a set of* **C4** *interpretations. Let IC_1 be the ground constraint $l_1 \wedge \cdots \wedge l_n \rightarrow$* **not** *$p$. Then for any $\mathcal{I} \in \mathcal{M}$, IC_1 evaluates to at least CT in $\mathcal{I}$ if and only of $\mathcal{I}$ satisfies $l_1 \wedge \cdots \wedge l_n \hookrightarrow_1 p$.*

Proof: $\Rightarrow$

Let $\mathcal{I}$ be any member of $\mathcal{M}$. Assume that IC_1 evaluates to at least CT in $\mathcal{I}$. Then either $l_1 \wedge \cdots \wedge l_n$ evaluates to at most CF in $\mathcal{I}$ in which case the contestation is trivially satisfied by $\mathcal{I}$, or $l_1 \wedge \cdots \wedge l_n$ and **not** p evaluates to at least CT and p evaluates to at most CF. In the latter case too the contestation is satisfied by $\mathcal{I}$.

$\Leftarrow$

Assume that $\mathcal{I}$ satisfies $l_1 \wedge \cdots \wedge l_n \hookrightarrow_1\ p$. Either $l_1 \wedge \cdots \wedge l_n$ evaluates to at least CT or it evaluates to at most CF in $\mathcal{I}$. In the former case p must be CF in $\mathcal{I}$ and so IC_1 must be true in $\mathcal{I}$. In the latter case IC_1 is true in $\mathcal{I}$ regardless of the truth value of p. ∎

So far we have represented conditional constraints in which the right-hand side is a negative literal and conditional constraints in which the left-hand side consists entirely of positive literals. Can we represent conditional constraints in which the left-hand side contains negative literals and the right-hand side is a positive literal? As will be discussed below, we cannot represent in terms of contestations a conditional constraint in which the left-hand side consists *entirely* of negative literals and the right-hand side is a positive literal. However, it is possible to represent a constraint of the form **not** $p,\ q \rightarrow\ r$ as **not** $p \wedge$ **not** $r \hookrightarrow_1\ q$. This says that in any model if both p and r fail to be true then q cannot be true, which captures the intuition behind the constraint **not** $p,\ q \rightarrow\ r$. But note that this representation of the constraint in terms of the contestation is possible only because the constraint contains an atom in its left-hand side which can be a contested atom in the corresponding contestation. More generally, a constraint of the form **not** $a_1 \wedge \cdots \wedge$ **not** $a_m \wedge\ b_1 \cdots \wedge\ b_n \rightarrow\ c$, where c is an atom and each b_j, $1 \leq\ j \leq\ n$, is an atom, can be represented by the following set of contestations:

$\{\textbf{not } a_1 \wedge \cdots \wedge \textbf{ not } a_m \wedge\ b_1 \cdots \wedge\ b_{j-1} \wedge\ b_{j+1} \wedge \cdots \wedge\ b_n \wedge \textbf{ not } c \hookrightarrow_1\ b_j \mid j \in \{1, \ldots, n\}\}$

The following establishes the correctness of this representation.

Lemma 9.3.4 *Let $\mathcal{M}$ be a set of* **C4** *interpretations. Let IC_1 be the ground constraint* **not** $a_1 \wedge \cdots \wedge$ **not** $a_m \wedge\ b_1 \cdots \wedge\ b_n \rightarrow\ c$*, where c is an atom and each b_j, $1 \leq\ j \leq\ n$, is an atom. Then for any $\mathcal{I} \in \mathcal{M}$, IC_1 evaluates to at least CT in $\mathcal{I}$ if and only of $\mathcal{I}$ satisfies each contestation in*

$\{\textbf{not } a_1 \wedge \cdots \wedge \textbf{ not } a_m \wedge\ b_1 \cdots \wedge\ b_{j-1} \wedge\ b_{j+1} \wedge \cdots \wedge\ b_n \wedge \textbf{ not } c \hookrightarrow_1\ b_j \mid j \in \{1, \ldots, n\}\}$

Proof: $\Rightarrow$

Let $\mathcal{I}$ be any member of $\mathcal{M}$. Assume that IC_1 evaluates to at least CT in $\mathcal{I}$. Either c evaluates to at least CT or at most CF in $\mathcal{I}$. In the former case since **not** c is on the left-hand side of each member of $\mathcal{C}$, it follows that the left-hand side of each contestation in $\mathcal{C}$ evaluates to at most CF in $\mathcal{I}$, and, thus, each contestation is trivially satisfied in $\mathcal{I}$. On the other hand, if c is at most CF, then the left-hand side of IC_1 must be at most CF. So at least literal in the left-hand side of IC_1 must be at most CF. If it is a negative literal then, since every negative literal in the left-hand side of IC_1 occurs in the left-hand side of every contestation in $\mathcal{C}$, the left-hand side of every contestation must be at most CF and thus every contestation must be trivially satisfied in $\mathcal{I}$. On the other hand, if a positive literal b_j in the left-hand side of IC_1 is at most CF then every contestation with b_j in the left-hand side is trivially satisfied in $\mathcal{I}$. And the only contestation in $\mathcal{C}$ with b_j in the right-hand side is also satisfied in $\mathcal{I}$ since b_j is CF in $\mathcal{I}$.

$\Leftarrow$

Assume that each member of $\mathcal{C}$ is satisfied in $\mathcal{I}$. Assume, by way of contradiction, that IC_1 does not evaluate to at least CT in $\mathcal{I}$. So the left-hand side of

IC_1 must evaluate to at least CT and the right-hand side of IC_1 must evaluate to at most CF in $\mathcal{I}$. But this means, for instance, **not** $a_1 \wedge \cdots \wedge$ **not** $a_m \wedge b_2 \cdots \wedge b_n \wedge$ **not** $c \hookrightarrow_1 b_1$ cannot be satisfied by $\mathcal{I}$ since both the left-hand side and the right-hand side of this contestation evaluates to at least CT in $\mathcal{I}$. Thus, a contradiction. Hence, IC_1 must evaluate to at least CT in $\mathcal{I}$. ■

Summarizing the discussion so far, the following types of ground constraints can be represented in terms of contestations:

1. $\leftarrow l_1 \wedge \cdots \wedge l_n$, where at least one literal is positive. This can be represented by the following set of contestations

$$\{l_1 \wedge \cdots \wedge l_{j-1} \wedge l_{j+1} \wedge \cdots \wedge l_n \hookrightarrow_1 l_j \mid j \in \{1, \ldots, n\}\}$$

2. $a_1 \wedge \cdots \wedge a_n \rightarrow b_1 \vee \cdots \vee b_m$, where all the literals are atoms. This can be represented by the following set of contestations

$$\{a_1 \wedge \cdots \wedge a_{i-1} \wedge a_{i+1} \wedge \cdots \wedge a_n \wedge \textbf{not } b_1 \wedge \cdots \wedge \textbf{not } b_m \hookrightarrow_1 a_i \mid i \in \{1, \ldots, n\}\}$$

3. $l_1 \wedge \cdots \wedge l_n \rightarrow$ **not** p. This can be represented by the contestation $l_1 \wedge \cdots \wedge l_n \hookrightarrow_1 p$.

4. **not** $a_1 \wedge \cdots \wedge$ **not** $a_m \wedge b_1 \cdots \wedge b_n \rightarrow c$, where each b_j, $1 \leq j \leq n$, and c are atoms. This can be represented by the following set of contestations:

$$\{\textbf{not } a_1 \wedge \cdots \wedge \textbf{not } a_m \wedge b_1 \cdots \wedge b_{j-1} \wedge b_{j+1} \wedge \cdots \wedge b_n \wedge \textbf{not } c \hookrightarrow_1 b_j \mid j \in \{1, \ldots, n\}\}$$

Are there any constraints expressible in the language of propositional logic that cannot be represented in terms of contestations? Most generally, from our perspective we cannot allow constraints that require that some positive information be provable. A constraint cannot guarantee the provability of something; at most, it guarantees the non-provability of something. Constraints are regarded as constraints on what can be inferred from a database. Something can be inferred from a database only if there is evidence for it in the database. A mere demand in the form of a constraint that some thing be provable does not make it provable. However, a demand that some literal not be provable from the database can be satisfied by blocking the inference of that literal. For this reason we hold that constraints of the form $a_1 \vee \cdots \vee a_n$, where each a_i is an atom, should not be permitted. They cannot be expressed in terms of contestations.

For the same reason we cannot allow a constraint of the form $l_1 \wedge \cdots \wedge l_n \rightarrow a_1 \vee \cdots \vee a_m$, where each l_i is a negative literal and a_j is a positive literal. Since $\rightarrow$ is material implication, this can be thought of as equivalent to a disjunction of positive literals. However, as we have seen before, a constraint cannot guarantee the truth (or provability) of any atom, but only the falsity (or non-provability) of an atom.

Similarly, a denial constraint of the form $\leftarrow l_1, \ldots, l_n$ where each l_i is a negative literal should be regarded as ill-formed from our perspective. This constraint is essentially equivalent to a disjunction of positive literals.

The above discussion of representing integrity constraints in terms of contestations shows that a large class of integrity constraints expressed in the language of propositional logic can be represented in terms of contestations. However, the language of contestations allows us to formulate constraints that do not correspond to any constraint that can be expressed in propositional logic (or predicate logic). Thus, suppose a bank wants the constraint that someone should be judged credit worthy only if there is no question of the individual being a loan defaulter. The policy is so strict that if evidence has been presented that an individual is a loan defaulter, then even if the evidence has been successfully contested, but not decisively refuted, the bank will not deem the individual credit worthy. This can be done by defining a *cap* function cap_5 such that $cap_5(v) = CF$, where $v \in \{T,\ CT,\ CF\}$, and $cap_5(F) = T$. In terms of cap_5, we can express the contestation $LoanDefaulter \hookrightarrow_5 \ CreditWorthy$. This contestation is satisfied by a model of the database only if in that model $LoanDefaulter$ is at least CF then $CreditWorthy$ is at most CF. Note that this is not the same as the *constraint* $LoanDefaulter \rightarrow$ **not** $CreditWorthy$, which can be satisfied even if in a canonical model of the database $LoanDefaulter$ is CF and $CreditWorthy$ is T.

Furthermore, since we allow heterogeneous contestations that allow evidence of different degrees of strength to count as evidence against some statement, representing constraints in terms of contestations permits us even more flexibility.

9.4 Semantics for integrity constraints

In this section we provide a formal semantics for databases with integrity constraints such that the *state* of the database may violate those constraints.

Definition 9.4.1 *A ground integrity constraint is of the allowed type if it is of the following type*

1. $\leftarrow l_1 \wedge \cdots \wedge l_n$*, where at least one literal is positive.*
2. $a_1 \wedge \cdots \wedge \ a_n \rightarrow b_1 \vee \cdots \vee \ b_m$*, where all the literals are atoms*
3. $l_1 \wedge \cdots \wedge l_n \rightarrow$ **not** p
4. **not** $a_1 \wedge \cdots \wedge$ **not** $a_m \wedge \ b_1 \cdots \wedge \ b_n \rightarrow \ c$*, where each* b_j*,* $1 \leq \ j \leq \ n$*, and* c *are atoms*

Furthermore, all contestations with an atom on the right-hand side are considered of the allowed type.

Since we have broadened the idea of integrity constraints to include contestations, in representing a set of integrity constraints in terms of contestations a contestation which is a member of that set is regarded as representing itself.

The following theorem says that our representation of integrity constraints of the allowed type in terms of contestations is correct. Thus, it merely summarizes the results of Lemma 9.3.1, Lemma 9.3.2, Lemma 9.3.3, and Lemma 9.3.4.

Theorem 9.4.1 *Let IC be a set of constraints of the allowed type. Let $\mathcal{C}$ be the representation of the constraints in IC in terms of contestations. Let $\mathcal{I}$ be any* **C4** *interpretation. Then IC is true in $\mathcal{I}$ if and only if $\mathcal{I}$ satisfies every contestation in $\mathcal{C}$.*

Proof: This follows straightforwardly from Lemma 9.3.1, Lemma 9.3.2, Lemma 9.3.3, and Lemma 9.3.4. ∎

Definition 9.4.2 *Let DB be a deductive database in the form of a normal logic program and let IC be a set of integrity constraints of the allowed type on DB. We define the canonical models of $DB \cup IC$ to be the* **C4** *canonical models of $DB + \mathcal{C}$, where $\mathcal{C}$ is the set of contestations representing IC.*

In terms of this definition of the canonical models of $DB \cup IC$, we can easily define the entailment relation $\approx$ as follows.

Definition 9.4.3 *Let DB be a deductive database in the form of a normal logic program and let IC be a set of integrity constraints of the allowed type on DB.*

Then, $DB \approx\!\!\!| \, l$*, with respect to* IC*, if and only if* $DB + \mathcal{C} \models_{\mathbf{C4}} l$*, where* $\mathcal{C}$ *represents* IC*.*

Example 9.4.1 *Let* $DB = \{p \leftarrow a;\ p \leftarrow b;\ a;\ b;\ q\}$*. Let* $IC = \{\leftarrow a, b\}$*. In this situation neither* a *nor* b *should be inferable from the database, but* $a \vee b$ *should be inferable and thus* p *should be inferable.*

IC is represented by $\mathcal{C} = \{a \hookrightarrow_1 b;\ b \hookrightarrow_1 a\}$*. The canonical models of* $DB + \mathcal{C}$ *are*

	a	b	p	q
$\mathcal{I}_1$	T	CF	T	T
$\mathcal{I}_2$	CF	T	T	T

Table 9.1: Models of a database that is inconsistent with its integrity constraints.

Since p *is* T *in both the canonical models of* $DB + \mathcal{C}$*,* $DB \mid\!\approx p$*. However,* $DB \not\mid\!\approx a$ *and* $DB \not\mid\!\approx b$*. Clearly,* $CONS_{IC}(DB) = \{p,\ q\}$ *and, thus,* $\leftarrow a, b$ *is true in* $CONS_{IC}(DB)$*. Thus, in our sense of integrity constraint satisfaction, this integrity constraint is satisfied by* DB *even though the* state *of* DB *does not satisfy* IC*.*

Note that although $a, b \in DB$*,* $DB \not\mid\!\approx a$ *and* $DB \not\mid\!\approx b$*. Thus, the entailment relation* $\mid\!\approx$ *is non-reflexive.*

The following theorem demonstrates that this is the correct definition of $\mid\!\approx$. Recall that the only formal requirement imposed on $\mid\!\approx l$ was that $CONS_{IC}(DB)$, which was defined to be the set $\{l \mid DB \mid\!\approx l\}$, should satisfy IC in the sense that each member of IC should be true in $CONS_{IC}(DB)$.

Theorem 9.4.2 *Let DB be a deductive database in the form of a normal logic program and let IC be a set of integrity constraints of the allowed type on DB. Then $CONS_{IC}(DB)$ satisfies IC.*

Proof: Recall that $CONS_{IC}(DB) = \{l \mid DB \approx l\}$ where l is a literal. By definition $DB \approx l$ if and only if $DB + \mathcal{C} \models_{\mathbf{C4}} l$, where $\mathcal{C}$ is the representation of IC. From Theorem 4.4.3 in Chapter 4 we know that all the **C4** canonical models of $P + \mathcal{C}$, for any normal logic program P and any set of contestations $\mathcal{C}$, satisfy all members of $\mathcal{C}$. Thus, all **C4** canonical models of $DB + \mathcal{C}$ must satisfy all members of $\mathcal{C}$. Hence, the intersection of all these models must also satisfy $\mathcal{C}$. But then by Theorem 9.4.1 above IC must be true in the intersection of all these models. But the intersection of all these models can be understood as the set $\{l \mid DB + \mathcal{C} \models_{\mathbf{C4}} l$, which is just $CONS_{IC}(DB)$. Thus $CONS_{IC}(DB)$ satisfies IC. ■

It is clear that the above theorem also establishes that the **C4** semantics for a database with its associated integrity constraints is inferentially conflict-free with regard to the types of conflicts expressed in terms of integrity constraints of the type discussed in this chapter.

9.5 Discussion

In Chapter 6 we describe a sound and complete procedure for answering queries to finite and ground normal logic programs augmented with a set of ground contestations. Clearly, this procedure then can be a sound and complete procedure for answering queries to a finite and ground normal logic databases augmented with a set of ground integrity constraints. To use the procedure all we have to do is represent the integrity constraints in terms of contestations in the manner described in this chapter.

Our approach to integrity constraint satisfaction has several useful features. First, it gives a role for integrity constraints even when the *state* of the database violates the constraints. For instance, the knowledge encoded in the constraints can still be used for semantic query optimization ([CGM90]). Second, our approach makes it possible to return meaningful answers to queries to a database even when the state of the database violates its integrity constraints. Thus, in Example 9.4.1 above, the answer to the query a would be NO, but an answer to the query p would be YES. Thus, our approach allows a database to return NO to a query even when the information is in the *state* of the database, if the information is involved in the violation of an integrity constraint. This makes the query answering procedure non-reflexive. Third, our approach allows for *lazy* updates. Thus, on the standard approach a database that contains the atom a and has a constraint $\leftarrow a, b$, would reject an update that tries to insert b in the database. However, our approach would permit this insertion, but it would have the effect that a can no longer be derived from the database and neither can b be derived. Thus, in effect, our approach allows the DBMS to perform a lazy deletion and a lazy insertion. This has the feature that if later a were withdrawn from the database, then b can be derived. Thus, the update b is not lost. In this respect our approach differs from work in belief revision ([GR95]). In belief revision an update that conflicts with existing information in the database is allowed to eliminate that conflicting piece of information. Thus, the database is always kept consistent. As noted above, our approach allows for lazy updates.

Yet another feature of our approach is that it does consistency checking on a need only basis. On the standard approach consistency checking is done with each update. However, some of this updated information may never be involved in answering any queries. From our perspective then the effort expended on consistency checking of this information is wasted. However, if consistency checking is done only for the information that is involved in answering queries, then consistency checking is done on a need only basis. If there are many more updates than queries to the database, this can result in a significant gain.

Although there are several approaches to reasoning with databases which violate their integrity constraints, as far as we know none of them have suggested a redefinition of what it means for integrity constraints to be satisfied by a database.

In [ABC99] an approach to answering queries to possibly inconsistent databases is described in terms of a query rewriting which is based on the notion of residues ([CGM90]). They describe a semantics for such databases in terms of the set of minimal modifications to an inconsistent databases which would result in a consistent version of the database. They prove the query procedure is sound and complete with respect to this semantics. However, their work seems to be restricted to databases without rules. Furthermore, this approach allows an inconsistent database to be modified into a consistent one by inserting new information. Thus, if a database with the constraint $p \rightarrow q$ does not entail q and it has p inserted in it, then this approach allows q to be inserted as well. This seems to us wrong. A constraint should not by itself generate new information.

[AKWS95] introduced the idea of *flexible relation*, a non-1NF relation that contains tuples with sets of values with the set standing for one of its values. So if there is a constraint that says there cannot be two tuples in a relation instance that differ only on that value and if a relation instance were to contain two such tuples, then these tuples can be combined into one tuple where in the relevant field there is a set containing the conflicting values. In effect, then the set can be considered a disjunction of the conflicting values. [AKWS95] is restricted to primary key functional dependencies, but this approach is generalized to other key functional dependencies in [Dun96]. These approaches rely on the construction of a single disjunctive instance and the deletion of conflicting tuples. [BKM91] also adopts this approach. In essence, this is also the approach utilized in [PMS95] and [PM96]. Our current approach is different in that conflicting information is not deleted or modified in any way. Instead, the inference procedure or the query answering procedure incorporates mechanisms that block the inference of conflicting information. Thus, there is no need to make any modifications to the database in the event of conflicts.

9.6 Summary

In this chapter we have provided an account of propositional integrity constraint satisfaction for function-free deductive databases that may be inconsistent with their own integrity constraints in terms of the **C4** semantics for normal logic programs augmented with a set of contestations. The specific research contributions of this chapter are as follows.

- We propose that integrity constraints be viewed as constraints on what can be proven from a database rather than constraints on the state of a database. We propose a new account of integrity constraint satisfaction in terms of this reinterpretation of the role of integrity constraints. More specifically, we have defined an entailment relation $\models\!\!\!\approx$ such that the set of entailments of the database in terms of this entailment relation satisfy the integrity constraints.

We have redefined the concept of integrity constraint satisfaction so that it is not the *state* of the database that must satisfy the constraints, but instead the set of inferences in terms of $\approx$ which must satisfy the constraints (Section 9.2).

- We show how to translate a wide range of propositional integrity constraints in terms of contestations and prove that this translation is correct (Section 9.3).

- We show that the **C4** semantics for normal logic programs augmented with a set of contestations can be used as a semantics for deductive databases augmented with a set of propositional integrity constraints (Section 9.4).

Chapter 10

Extending C4: Semantics of Preferences

10.1 Introduction

To solve a problem one may need to draw on the knowledge of several different experts. It can happen that some of the claims of one or more experts may be in conflict with the claims of other experts. We assume that the knowledge of experts can be encoded in the form of normal logic programs. Such normal logic programs can also be considered as databases. Thus, pooling together the knowledge of different experts can be regarded as combining databases. Conflicts among statements can be represented in the form of contestations. In the previous chapters we have developed a semantics and a proof procedure for normal logic programs augmented with contestations.

In this chapter we introduce preferences among statements as a way of reducing conflicts among statements. We envisage these preferences as provided by a user of the combined database or by the integrator of the different pools of information. These statements of preference are intended to express arbitrary preferences of a given user.

Consider a motivating example. Suppose the personnel officer of a large company has to determine whether there are any medical reasons for not hiring a certain applicant for a high stress job. It is the standard practice of the company to consult both a cardiologist who examines the applicant and the patient's personal physician. The cardiologist's report says, among other things, that the applicant suffers from a certain heart irregularity that leads to a heart attack under great stress, and therefore the applicant may well suffer a heart attack due to the stress of the job. It also says that the applicant's diet, if continued over a long period of time, will worsen the heart irregularity. The personal physician testifies that over the many years that the applicant has been his patient he has remained very healthy even in times of great stress. And the patient's generally robust health and healthy habits will enable him to handle very stressful situations in spite of his heart irregularity. The physician further adds that the applicant has been following a diet over the years prescribed by the physician which will reduce the heart irregularity. The physician notes that indeed the heart irregularity has somewhat diminished over the years. The rest of the physician's report is not in conflict with the cardiologist's report.

Clearly, these two reports are in conflict. Furthermore, they are in conflict over two points: 1) whether the patient's heart irregularity will make him unable to handle great stress, and 2) the effect of his diet on his heart irregularity. In making his decision the personnel manager of the company may prefer one statement over another in case they conflict. Thus, in our example, the personnel manager can give preference to the physician's claim (x) that the applicant can handle the stress of the job over the cardiologist's claim (y) that the stress of the job will cause a heart attack in the applicant. The personnel manager may give preference to the claim x of the physician because he thinks it is more reliable, or because it is the company's policy to give an applicant the benefit of the doubt in these matters, or because the personnel manager favors the applicant, or whatever. That is, in our formal treatment of preferences, we shall not make any assumptions about the user's reasons for preferring one statement to another.

To say that the manager prefers x over y does not mean that x will be finally be accepted by the manager, but only that in the conflict between x and y, x is chosen over y. It could happen that x is in conflict with some other statement, z, in the combined database which is preferred over x and, thus, x may not end up being accepted. Or it could happen that there is not enough evidence for the claim x, so regardless of the preference it cannot be accepted.

The preferences of an ideally rational agent are transitive. But real agents (or users) are not always this rational in their preference structures ([TK81]). So we shall not assume that the user-supplied preferences are transitive. But, if for a given user they happen to be transitive, our approach applies to such preferences without any modification.

In Section 10.2 we develop the formal preliminaries for stating semantics of normal logic programs augmented with a set of contestations and a set of preferences. In Section 10.3 we state two semantics of normal logic programs augmented with a set of contestations and a set of preferences and prove their equivalence. In Section 10.4 we discuss related work. In Section 10.5 we summarize the main research contributions of this chapter.

10.2 Preliminaries

We write x is preferred to y as $x \succ y$.

The preference $x \succ y_1 \wedge y_2 \wedge \cdots \wedge y_m$, where x and each y_i are atoms, is understood to mean that x is preferred to the conjunction of y_i. The preference $x \succ y_1 \vee \cdots \vee y_m$, where x and each y_i are atoms, is understood to mean that x is preferred to each y_i and that x is preferred to the conjunction of atoms in any subset of $\{y_1, \ldots, y_m\}$. Let $Y = \{y_1, y_2, \ldots, y_m\}$. Then we understand $x \succ Y$ to mean $x \succ y_1 \vee \cdots \vee y_m$. Let $X = \{x_1, x_2, \ldots, x_n\}$ be sets of atoms. Then the preference $X \succ Y$ is understood as an abbreviation of $\{x_1 \succ Y, x_2 \succ Y, \ldots, x_n \succ Y\}$.

We call a set of preference $\{x \succ Y_1, x \succ Y_2, \ldots, x \succ Y_m\}$ *additive* if they imply $x \succ Y_1 \cup \ldots \cup Y_m$. *In general, we assume that preferences are not additive.* This is because, in general, it is not the case that if a statement (or a choice) is preferred to several other statements (or choices) *taken individually* that statement (or choice) is preferred to those other statements (or choices) *taken jointly.*

Intuitively, we understand a reasoner's preference $x \succ y$ over a logic program LP and a set of contestations $\mathcal{C}$ to mean that the reasoner prefers $LP + \mathcal{C}$ to entail x over entailing y. Thus, *other things being equal*, the preference should result in $LP + \mathcal{C}$ entailing x if it entails y. The semantics of preferences given below are guided by this intuition.

We suggest two different ways of explicating the idea that $x \succ y$ should mean that the reasoner prefers $LP + \mathcal{C}$ to entail x over entailing y.

- **Preference Ordering**. Let $\mathcal{P}$ be a set of preferences. We use the preferences in $\mathcal{P}$ to induce an ordering among the models and choose only the maximal members of this ordering as candidates for the canonical models of $LP+\mathcal{C}+\mathcal{P}$.

- **Satisfaction**. Analogous to the idea of the satisfaction of a clause or a contestation by an interpretation, we develop the concept of the satisfaction of a preference $P \in \mathcal{P}$ by an interpretation, and choose only those models which satisfy all the preferences as candidates for the canonical models of $LP + \mathcal{C} + \mathcal{P}$.

Recall that given a four-valued interpretation $\mathcal{I}$, by $Truth(\mathcal{I})$ we mean $\{a \mid a$ is an atom and $\mathcal{I}(a) \geq CT\}$

We need the following definitions to formalize the above stated ways of understanding preferences.

Definition 10.2.1 *Let $\mathcal{I}$ be a well-supported model of $LP+\mathcal{C}$ and let Y be a set of ground atoms. Then, $Dep_{\mathcal{I}}(Y)$ denotes all the members of $Truth(\mathcal{I})$ which become unsupported in any mapping $\mathcal{I}'$ such that the only difference between $\mathcal{I}$ and $\mathcal{I}'$ is that $\mathcal{I}'$ assigns at most CF to members of Y.*

Intuitively, $Dep_{\mathcal{I}}(Y)$ are those atoms whose status as members of $Truth(\mathcal{I})$ depends on the status of Y in $\mathcal{I}$.

Definition 10.2.2 *Let $\mathcal{I}$ be a well-supported model of $LP + \mathcal{C}$ and let Y be a set of ground atoms. Then,*

$$Effect_{\mathcal{I}}(Y) \;=\; Y \cup Dep_{\mathcal{I}}(Y)$$

Thus, by $Effect_{\mathcal{I}}(Y)$ we mean all those atoms that will be demoted from the status of the Truth in $\mathcal{I}$ if Y is demoted from the status of Truth.

We say that $\mathcal{I} \models a$ if $\mathcal{I}(a) \geq CT$. Clearly, $\mathcal{I} \models a$ if, and only if, $a \in Truth(\mathcal{I})$.

10.3 Semantics of Preferences

Preference Ordering Semantics

When $LP + \mathcal{C}$ is augmented with a preference, $P = x \succ Y$, where x and all members of Y belong to HB_{LP}, we take this preference as inducing an ordering among the models of $LP + \mathcal{C}$. *Other things being equal*, the well-supported models of LP which satisfy all the members of $\mathcal{C}$ in which x is at least CT are preferred over the well-supported models of LP which satisfy all members of $\mathcal{C}$ in which any member of Y is at least CT. The following definitions are needed to formalize this idea.

Definition 10.3.1 *Let P be the preference $x \succ Y$. Let LP be a normal logic program and let $\mathcal{C}$ be a set of contestations. Let $\mathcal{I}_1$ and $\mathcal{I}_2$ be two models of $LP+\mathcal{C}$. We say that $\mathcal{I}_1 \sqsubseteq_P \mathcal{I}_2$ if*

1. *$\mathcal{I}_1 \not\models x$ and $\mathcal{I}_2 \models x$, and*
2. *$\mathcal{I}_1 \models y$ for some $y \in Y$, and*
3. *$Truth(\mathcal{I}_1) - Effect_{\mathcal{I}_1}(Y) \subseteq Truth(\mathcal{I}_2)$.*

Conditions 1 and 2 together say that $x \succ Y$ makes $\mathcal{I}_2$ preferred to $\mathcal{I}_1$ if $\mathcal{I}_1 \not\models x$ but $\mathcal{I}_1 \models y$ whereas $\mathcal{I}_2 \models x$ *so long as other things are equal.* Condition 3 captures this qualification. Other things are not equal if there is a z such that independently of Y it prevents $\mathcal{I}_1$ from entailing x. If there were such a z it cannot belong to $Truth(\mathcal{I}_2)$ since $\mathcal{I}_2$ does entail x, but such a z would belong to $Truth(\mathcal{I}_1) - Effect_{\mathcal{I}_1}(Y)$. Thus, Condition 3 says that there is no z such that $z \in Truth(\mathcal{I}_1) - Effect_{\mathcal{I}_1}(Y)$ but $z \notin Truth(\mathcal{I}_2)$. That is, other things are equal.

We say that $\mathcal{I}_1 \sqsubseteq_{\mathcal{P}} \mathcal{I}_2$ if there is a $P \in \mathcal{P}$ such that $\mathcal{I}_1 \sqsubseteq_P \mathcal{I}_2$.

Given a set of interpretations ν of $LP+\mathcal{C}$, we say that $\mathcal{I}_1$ is a *preferred* member of ν with respect to a set of preferences $\mathcal{P}$ if and only if there is no interpretation $\mathcal{I}_2$ in ν such that $\mathcal{I}_1 \sqsubseteq_{\mathcal{P}} \mathcal{I}_2$.

$\mathcal{I}_1$ is a model of $LP+\mathcal{C}+\mathcal{P}$ if it is a preferred member, with respect to $\mathcal{P}$, of the set of models of $LP+\mathcal{C}$.

Definition 10.3.2 *Let LP be a normal logic program, let $\mathcal{C}$ be a set of contestations and let $\mathcal{P}$ be a set of preferences. Then according to the* preference semantics *the* canonical *models of $LP+\mathcal{C}+\mathcal{P}$ are the clausally maximal members among the preferred members, with respect to $\mathcal{P}$, of the set of well-supported models of $LP+\mathcal{C}$.*

Satisfaction Semantics of Preferences

A preference $x \succ Y$ can be understood as making a demand of a model $\mathcal{I}$ that *other things being equal* if $\mathcal{I}$ entails any member of Y then it should entail x. Other things are not equal with respect to $x \succ Y$ if some factor independent of Y makes it impossible for $\mathcal{I}$ to entail x.

Definition 10.3.3 *Let LP be a normal logic program and let $\mathcal{C}$ be a set of contestations. Let $x \succ Y$ be a preference. An interpretation $\mathcal{I}_1$ satisfies $x \succ Y$ if*

1. *$\mathcal{I}_1 \models x$, or*
2. *For all $y \in Y$, $\mathcal{I}_1 \not\models y$, or*
3. *There is no well-supported model $\mathcal{I}_2$ of $LP+\mathcal{C}$ such that*
   - *$\mathcal{I}_2 \models x$, and*

- $(Truth(\mathcal{I}_1) - Effect_{\mathcal{I}_1}Y) \subseteq Truth(\mathcal{I}_2)$

As in the definition of Preference Ordering Semantics, Condition 3 captures the qualification that other things are equal.

$\mathcal{I}$ satisfies a set of preferences $\mathcal{P}$ if, and only if, it satisfies all members of $\mathcal{P}$.

Definition 10.3.4 *Let LP be a normal logic program, let $\mathcal{C}$ be a set of contestations, and let $\mathcal{P}$ be a set of preferences. Then according to the* satisfaction semantics *the* canonical *models of $LP + \mathcal{C} + \mathcal{P}$ are the clausally maximal models among all the well-supported model of $LP + \mathcal{C}$ which satisfy all the preferences in $\mathcal{P}$.*

The following theorem shows that the two semantics of preferences are equivalent.

Theorem 10.3.1 *A well-supported model $\mathcal{I}$ of $LP+\mathcal{C}$ satisfies a set of preferences $\mathcal{P}$ if, and only if, $\mathcal{I}$ is a preferred model, with respect to $\mathcal{P}$, of $LP + \mathcal{C}$.*

Proof:

$\Rightarrow$

Assume that $\mathcal{I}_1$ is a well-supported model of $LP + \mathcal{C}$ which satisfies all the preferences in $\mathcal{P}$. Assume by way of contradiction that $\mathcal{I}_1$ is not a preferred model. So there must be another well-supported model $\mathcal{I}_2$ of $LP + \mathcal{C}$ and a preference $x \succ Y$ such that $\mathcal{I}_1 \sqsubseteq_{x \succ Y} \mathcal{I}_2$. That is, it must be the case that

1. $\mathcal{I}_1 \not\models x$ and $\mathcal{I}_2 \models x$, and
2. $\mathcal{I}_1 \models y$ for some $y \in Y$, and
3. $Truth(\mathcal{I}_1) - Effect_{\mathcal{I}_1}(Y) \subseteq Truth(\mathcal{I}_2)$.

However, $\mathcal{I}_1$ satisfies $x \succ Y$. So it must be the case that

1. $\mathcal{I}_1 \models x$, or

2. For all $y \in Y$, $\mathcal{I}_1 \not\models y$, or

3. There is no well-supported model $\mathcal{I}_2$ of $LP + \mathcal{C}$ such that

   - $\mathcal{I}_2 \models x$, and
   - $(Truth(\mathcal{I}_1) - Effect_{\mathcal{I}_1}(Y)) \subseteq Truth(\mathcal{I}_2)$

If $\mathcal{I}_1$ satisfies $x \succ Y$ by clause 1 of the definition of satisfaction then the first condition for its being the case that $\mathcal{I}_1 \sqsubseteq_{x \succ Y} \mathcal{I}_2$ is violated.

If $\mathcal{I}_1$ satisfies $x \succ Y$ by clause 2 of the definition of satisfaction then the second condition for its being the case that $\mathcal{I}_1 \sqsubseteq_{x \succ Y} \mathcal{I}_2$ is violated.

If $\mathcal{I}_1$ satisfies $x \succ Y$ by clause 3 of the definition of satisfaction then the third condition for its being the case that $\mathcal{I}_1 \sqsubseteq_{x \succ Y} \mathcal{I}_2$ is violated.

$\Leftarrow$

Assume that $\mathcal{I}_1$ is a preferred model of $LP+\mathcal{C}+\mathcal{P}$. Assume by way of contradiction that $\mathcal{I}_1$ does not satisfy all the preferences. Let $x \succ Y$ be one such preference. This implies that

1. $\mathcal{I}_1 \not\models x$, and

2. For some $y \in Y$, $\mathcal{I}_1 \models y$, and

3. There is a well-supported model $\mathcal{I}_2$ of $LP + \mathcal{C}$ such that

   - $\mathcal{I}_2 \models x$, and
   - $(Truth(\mathcal{I}_1) - Effect_{\mathcal{I}_1}(Y)) \subseteq Truth(\mathcal{I}_2)$

It is apparent that such a $\mathcal{I}_1$ and $\mathcal{I}_2$ must satisfy all the conditions for it being the case that $\mathcal{I}_1 \sqsubseteq_{x \succ Y} \mathcal{I}_2$.

But this contradicts the assumption that $\mathcal{I}_1$ is a preferred model. Thus $\mathcal{I}_1$ must satisfy all the preferences. ■

Example 10.3.1 *Let $LP = \{a \leftarrow; b \leftarrow\}$ and let $\mathcal{C} = \{a \hookrightarrow_1 b;\ b \hookrightarrow_1 a\}$. Furthermore, let $\mathcal{P} = \{a \succ b\}$. Then the well-supported models of $LP + \mathcal{C}$ are $\mathcal{I}_1$ which assigns T to a and CF to b, and $\mathcal{I}_2$ which assigns CF to a and T to b. But the only preferred model and, thus, the only canonical model of $LP + \mathcal{C} + \mathcal{P}$ is $\mathcal{I}_1$.*

The following defintion extends the definitions of strong and weak entailment to the case of $LP + \mathcal{C} + \mathcal{P}$.

Definition 10.3.5 $LP + \mathcal{C} + \mathcal{P}$ *strongly* entails a literal p under **C4** if, and only if, p evaluates to T in all the canonical models of $LP + \mathcal{C} + \mathcal{P}$.

$LP + \mathcal{C} + \mathcal{P}$ *weakly* entails a literal p under **C4** if, and only if, p evaluates to *at least* CT in all the canonical models of $LP + \mathcal{C} + \mathcal{P}$.

10.4 Discussion

In this chapter we have provided two equivalent semantics for normal logic programs augmented with a set of contestations and a set of preferences. We do not present a procedure for answering queries to normal logic programs augmented with contestations and preferences. Our results in this chapter are rather limited. In [PM96] we showed that there exists at least one canonical model for definite logic programs augmented with denial integrity constraints and preferences having a non-cyclic structure. For the class of general logic programs and contestations which need not represent denial constraints, it is not the case that such programs must have at least one canonical model when they are augmented with preferences, regardless of what restrictions one puts on the structure of the preferences. The following example illustrates this point.

Example 10.4.1 *Let* $LP = \{a \leftarrow \textbf{not } b;\ b \leftarrow \textbf{not } c;\ c \leftarrow \textbf{not } d; d \leftarrow \textbf{not } a\}$. *Let* $\mathcal{C} = \emptyset$ *and* $\mathcal{P} = \{b \succ a,\ c \succ d\}$. $LP + \mathcal{C}$ *has the following four well-supported models* $b \succ a$ *is not satisfied by* $\mathcal{I}_1$ *and* $\mathcal{I}_3$ *and* $c \succ d$ *is not satisfied by* $\mathcal{I}_2$ *and*

	a	b	c	d
$\mathcal{I}_1$	T	F	T	F
$\mathcal{I}_2$	F	T	F	T
$\mathcal{I}_3$	CT	CF	CT	CF
$\mathcal{I}_4$	CF	CT	CT	CF

Table 10.1: A logic program augmented with contestations and preferences that has no canonical models.

$\mathcal{I}_4$. *Thus,* $LP + \mathcal{C} + \mathcal{P}$ *has no well-supported models even though* $\mathcal{P}$ *has only two preferences without any apparent relation between them.*

In terms of our work on preferences among arbitrary statements, we can express preferences among theories. Thus, $T_1 > T_2$, understood to mean $T_1 = \{a_1, a_2, \ldots, a_n\}$ has preference over T_2, can be expressed as $\{a_1 > T_2, a_2 > T_2, \ldots, a_n > T_2\}$.

Our framework can also be used to express preferences among theories in terms of topics. Thus, let $T_1 >_t T_2$ mean that theory T_1 is to be preferred over theory T_2 on topic t. In our framework this can be expressed as, $\{a_1 > \{b_1, b_2, \ldots, b_j\}, a_2 >$

$\{b_1, b_2, \ldots, b_j\}, \ldots, a_k > \{b_1, b_2, \ldots, b_j\}\}$, where $\{a_1, a_2, \ldots, a_k\}$ are the statements in the Herbrand base of T_1 on topic t and $\{b_1, b_2, \ldots, b_j\}$ are the statements in the Herbrand base of T_2 on topic t.

In the following we discuss related work. [BKM91] gives methods of combining theories each of which satisfies a set of integrity constraints, where the naive union of the theories fails to satisfy the integrity constraints. They do not however consider adding preferences among arbitrary statements or even preferences among theories. [BKMS92] gives methods for combining first order theories with preferences among theories. [FUV83] present an account of updates with preferences among sets of statements. However, none of these papers consider the problem of combining databases with preferences among arbitrary statements.

Ryan's work on Ordered Theory Presentations [Rya91, Rya92a, Rya92b] gives a semantics for first order sentences with arbitrary preferences among statements. It is based on the idea of ordering all possible interpretations of the sets of sentences in terms of which interpretations most nearly satisfy the set of sentences and satisfy the preferences. Interpretations maximal in the ordering are taken to be the models of the set of sentences with the preferences. Clearly, Ryan's approach is closely related to what we have called *Preference Ordering Semantics*. Our approach is different than Ryan's in several respects. First, Ryan's preferences are required to be transitive; we do not require preferences to be transitive. Second, in our system preferences are not additive. That is, if $a > b$ and $a > c$, then it does not follow that $a > \{b, c\}$. Third, Ryan's treatment of preferences restricts itself to preferences among sentences in the theory, but does not consider preferences among any two sentences in the Herbrand base, regardless of whether they are part of the theory. But our approach allows this.

In [PMS95] we gave several equivalent semantics for logic programs consisting entirely of ground atoms augmented with a set of denial integrity constraints and a set of preferences. This work was extended in [PM96] in which we developed two equivalent non-cautious semantics and a cautious semantics for definite logic programs augmented with a set of denial integrity constraints and a set of preferences. The work presented in this chapter generalizes this work to the class of general logic programs augmented with a set of contestations and a set of preferences. As noted above, some of the key results of [PMS95] and [PM96] cannot be extended to this more general class of programs and constraints.

An alternative approach to combining multiple databases has been developed by Subrahmanian [Sub94] who develops a language for expressing supervisory databases. Intuitively, a supervisory database contains conflict resolution information. What [Sub94] lacks is an explicit articulation of what preference means, and this is provided by our semantics of preferences.

10.5 Summary

The main research contributions of this chapter are summarized as follows.

- We provide a language for expressing preferences among statements.
- We extend **C4** to provide two semantics for a normal logic program, LP, augmented with a set of contestations, $\mathcal{C}$, and a set of preferences, $\mathcal{P}$.
  - The first semantics is based on using the preferences of $\mathcal{P}$ to induce an ordering among the well-supported models of $LP + \mathcal{C}$.
  - The second semantics is based on the idea of a well-supported model of $LP + \mathcal{C}$ satisfying the preferences of $\mathcal{P}$.

- Although these two semantics are based on different ways of factoring in the role of preferences, we prove that these two semantics are equivalent.

Chapter 11

Extended Logic Programs

11.1 Introduction

The case for logic programs with two types of negation, one which may be called default negation and the other which may be called non-default has been made by several authors ([GL90], [KS90]). A bus driver may use the rule :

Cross railway tracks if train is not coming.

As [GL90] note it would be folly if this rule were interpreted to mean

Cross railway tracks if you cannot prove that a train is coming

This interpretation is based on interpreting ‘not’ as default negation.

Rather, the rule is intended to mean

Cross railway tracks if you can prove that a train is not coming

This interpretation is based on interpreting ‘not’ as non-default negation.

On the other hand, the use of ‘not’ should be understood as default negation in a rule such as

Continue driving if a stop request has not been made

Thus there can be use for logic programs containing the use of default as well as non-default negation.

There can be different types of non-default negation such as classical, strong, and explicit negation. They differ in terms of how closely and in what respects they approximate classical negation. [AP92a] contains a systematic study of several types of non-default negation.

Various semantics have been proposed for logic programs containing default and non-default negations ([GL90], [AP92a], [Prz90a], [DR91], [ADP93]). Although these semantics have employed different versions of non-default negation, these semantics have not always been based on clearly identifying the semantic differences between default negation and their chosen version of non-default negation. These semantic differences can be displayed most clearly by associating non-default negation with a mapping from tuples of truth values to truth values and associating a different such mapping with default negation, where these mappings completely characterize the semantics of each type of negation. This also allows us to treat both kinds of negation as logical operators.

In this chapter we extend **C4** to **C5**, a five-valued semantic framework. In terms of **C5** we propose a family of semantics for logic programs containing both default and non-default negation. Using **C5** we give a semantic account of the difference between default and non-default negation by associating a different mapping with each type of negation.

In Section 11.2 we introduce the five truth values of **C5** and several types of ordering between them. We define the functions for evaluating arbitrary extended logic programming sentences in terms of a mapping from atomic sentences to these truth values. In Section 11.3 we develop the **C5** semantics for extended logic programs. In Section 11.4 we compare the **C5** semantics with the answer set semantics for extended logic programs ([GL90]). We prove that an extended logic program LP with a consistent answer set entails a literal p with respect to the answer sets of LP if, and only if, LP weakly entails p (under **C5**). In Section 11.5 we show how the five truth values of **C5** and the three types of ordering between them can be derived from the more basic set of truth values $\{T, U, F\}$ and the standard truth and information ordering among them. In Section 11.6 we summarize the main research contributions of this chapter.

11.2 Preliminaries

We extend the language of normal logic programs by adding a new negation symbol $\neg$.

By an *objective literal* we mean either an atom a or $\neg a$. We call $\neg a$ the *non-default negation* of a. By a *default literal* we mean an expression of the form **not** l, where l is an objective literal. We call **not** l the *default negation* of l. We stipulate that $\neg$**not** l is not a well-formed expression of our language.

An extended rule R is of the form

$$l \leftarrow a_1, \ldots, a_m, \mathbf{not}\ b_1, \ldots, \mathbf{not}\ b_n$$

where l, each a_i and each b_j are objective literals. An extended logic program is a set of such rules.

By EHB_{LP}, the extended Herbrand base of LP, we mean $\{\neg a \mid a \in HB_{LP}\} \cup HB_{LP}$, where HB_{LP} is the Herbrand base of LP.

Given the extended rule R above

1. $body(R) = \{a_1, \ldots, a_m, \textbf{not } b_1, \ldots, \textbf{not } b_n\}$

2. $objbody(R) = \{l \in body(R) \mid l \text{ is an objective literal}\}$

3. $posbody(R) = \{a \in objbody(R) \mid a \text{ is an atom}\}$

4. $negbody(R) = \{\textbf{not } b_1, \ldots, \textbf{not } b_n\}$

We understand $\neg\neg l$ to mean l. That is, we ignore the double negation. The function $Atom(l)$, where l is a literal, returns the atom that occurs in l. Thus If l is an atom then $Atom(l)$ returns l. Otherwise if l is of the form $\neg a$, or $\neg\neg a$, or **not** a, or **not** $\neg a$, then $Atom(l)$ returns the atom a.

We extend **C4** to **C5**. Let $\mathcal{V} = \{T, CT, CF, F, U\}$. The new truth value U intuitively means unknown. We introduce a truth ordering, $<_t$, and an information ordering, $<_i$, among the members of$\mathcal{V}$. According to the truth ordering $F <_t U <_t T$ and $F <_t CF <_t CT <_t T$. Thus, in the truth ordering U and CF and U and CT are incomparable. According to the information ordering U is lower than the other members of $\mathcal{V}$ which are themselves incomparable with each other in the information ordering.

In terms of the truth ordering and information ordering, we construct a supported ordering, $<_s$. Given $\nu_1,\ \nu_2 \in \mathcal{V}$, $\nu_1 <_s \nu_2$ iff $\nu_1 <_i \nu_2$ or $\nu_1 <_t \nu_2$ if ν_1 and ν_2 are incomparable in terms of $<_i$. Thus the supported ordering gives us $U <_s F <_s CF <_s CT <_s T$. We use the supported ordering to induce an ordering among the models of an extended logic program, whereas we use the truth ordering to define the truth values of nonatomic sentences.

In the discussion section of this chapter we explain how we derive the truth values of $\mathcal{V}$ and the truth and information orderings among the members of $\mathcal{V}$ in terms of a more basic set of truth values $\{T, U, F\}$ and the truth and information ordering among them.

As in **C4**, T and CT are regarded as the designated true values. Whereas in **C4** the default truth value is F, in **C5** the default truth value is U in the sense that any atom in the Herbrand base of a program which is not the head of a clause, or whose non-default negation is not the head of a clause, is assigned U.

In our semantics $\neg$ denotes the mapping **NEG** and **not** denotes the mapping **NOT**. We state the mappings in terms of the following truth-tables.

NEG	T	CT	CF	F	U
	F	CF	CT	T	U

Table 11.1: The **NEG** function

Note that on this interpretation of non-default negation, $\mathbf{NEG}(\mathbf{NEG}(\mathbf{v})) = \mathbf{v}$ for any $\mathbf{v} \in \mathcal{V}$, and for any interpretation $\mathcal{I}$, $\mathcal{I}(\neg p) = \mathbf{NEG}(\mathcal{I}(p))$.

NOT	T	CT	CF	F	U
	F	CF	CT	T	T

Table 11.2: The **NOT** function

The differences between these two mappings clearly bring out the differences between $\neg$ and **not** . It can be seen that **NEG** and **NOT** coincide on all truth-values except U, the default truth-value. It is precisely because $\mathbf{NOT}(U)$ evaluates to T that we call **not** *default* negation.

In our semantics $\wedge$ denotes the mapping **AND** and $\vee$ denotes the mapping **OR**. These mappings are given below, where we assume that $\nu_1, \nu_2 \in \mathcal{V}$.

$$\mathbf{AND}(\nu_1, \nu_2) = \begin{cases} min(\nu_1, \nu_2) & \text{if } \nu_1, \nu_2 \neq U \\ U & \text{otherwise} \end{cases}$$

$$\mathbf{OR}(\nu_1, \nu_2) = \begin{cases} max(\nu_1, \nu_2) & \text{if } \nu_1, \nu_2 \neq U \\ \nu_2 & \text{if } \nu_1 = U \\ \nu_1 & \text{otherwise} \end{cases}$$

An interpretation for an extended logic programming LP is a mapping from HB_{LP}, the Herbrand base of LP, to $\mathcal{V}$. In the following we define the function $\mathcal{I}'$, which extends this mapping to the (closed) sentences of the language.

Definition 11.2.1 *Let $\mathcal{I}$ be an interpretation. Then $\mathcal{I}'$ is a mapping from the sentences of the language to $\mathcal{V}$ recursively defined as:*

- *If S is a ground atom then $\mathcal{I}'(S) = \mathcal{I}(S)$.*
- *If S is a closed sentence then*

$$\mathcal{I}'(\neg S) = \mathbf{NOT}(\mathcal{I}'(S))$$
$$\mathcal{I}'(\mathbf{not}\ S) = \mathbf{NEG}(\mathcal{I}'(S))$$

- *If S_1 and S_2 are (closed) sentences then*

$$\mathcal{I}'(S_1 \wedge S_2) = \mathbf{AND}(\mathcal{I}'(S_1), \mathcal{I}'(S_2))$$
$$\mathcal{I}'(S_1 \vee S_2) = \mathbf{OR}(\mathcal{I}'(S_1), \mathcal{I}'(S_2))$$
$$\mathcal{I}'(S_1 \leftarrow S_2) = \begin{cases} T & \textit{if } \mathcal{I}'(S_1) \geq_t \mathcal{I}'(S_2) \\ U & \textit{if } \mathcal{I}'(S_1) \textit{ and } \mathcal{I}'(S_2) \textit{ are not comparable} \\ CT & \textit{if } \mathcal{I}'(S_1) = CF \textit{ and } \mathcal{I}'(S_2) = CT \\ F & \textit{otherwise} \end{cases}$$

- *For any sentence $p(X)$ with one unbound variable X,*

$$\mathcal{I}'(\forall X p(X)) = min\{\mathcal{I}'(p(t)) \mid t \in HU_P\}.$$

- *For any sentence $p(X)$ with one unbound variable X,*

$$\mathcal{I}'(\exists X p(X)) = max\{\mathcal{I}'(p(t)) \mid t \in HU_P\}.$$

11.3 Model Theory

For the purposes of the model theory of logic programs, we envisage the extended logic program LP to be augmented as follows:

- As in the case of **C4**, we add

  1. The special atoms $true$, $CTrue$, $CFalse$, $false$. It is assumed that $true$ (resp. $CTrue$, $CFalse$, and $false$) evaluates to T (resp., CT, CF, and F) in any interpretation.
  2. if LP contains no constants, the dummy clause $p(\$a) \leftarrow p(\$a)$, where $\$a$ is a constant.
  3. Any clause with an empty body is assumed to have $true$ as its body.

- Additionally for **C5** we add

  1. The special atom $unknown$ which is assumed to evaluate to U in any interpretation.
  2. For each literal l in EHB_{LP}, such that there is no clause in $grd(LP)$ with l in the head or with $Atom(l)$ in the head, we add a clause with $Atom(l)$ as head and $unknown$ as body.

Augmenting the logic program in this manner allows us to state the model theory more elegantly than if we did not augment it thus. (More specifically, it helps with the definition of a well-supported model below.) It should be clear in the following that the augmentation makes no difference to the actual semantics attributed to a logic program.

As in the case of **C4**, we say that an interpretation $\mathcal{I}$ satisfies a *ground* clause C if $\mathcal{I}'(C) \in \{T, CT\}$. Recall that T and CT are the designated truth values.

As usual an interpretation is a model of LP if it satisfies all the rules of LP.

Given an interpretation of LP, which is a mapping from the atoms of HB_{LP} to $\mathcal{V}$, the truth value of all the ground objective literals is determined by **NEG** and the truth values of all the ground default literals is determined by **NOT**. This fact and the fact that for any interpretation $\mathcal{I}$, if $\mathbf{NEG}(\mathcal{I}(p))$ then $\mathbf{NOT}(\mathcal{I}(p))$ ensures that every interpretation of any extended logic program satisfies the so-called *coherence principle*

$$\neg p \ \rightarrow \ \mathbf{not}\ p.$$

Note that unlike in [AP92a] the coherence principle does not have to be enforced by adding any special sentences to a program.

Furthermore, since **NEG** is a one-to-one and onto mapping it follows that for any objective literal l every interpretation I obeys the following *structural principle*

$$\mathcal{I}(l) = \mathbf{NEG}(\mathcal{I}(\neg l))$$

The above structural principle implies that the truth values of a literal determine the truth value of its non-default negation and vice versa. This implies that

- The truth value assigned to $\neg l$ by an interpretation can justify assigning l a *higher* truth value than would be justified in terms of rules with l in the head. Thus, the definition of a well supported model has to be extended to allow that the truth value of $\neg l$ can indirectly support the assignment of a certain truth value to l. This is accomplished in the next sub-section.

- The truth value assigned to $\neg l$ can force l to be assigned a *lower* truth value than would be justified in terms of rules with l in the head. This is analogous to a contestation of an atom forcing the atom to have a lower truth value than would be justified in terms of rules with l in the head. This property of non-default negation can be captured by extending the apparatus of contestations to non-default negation. This is accomplished in the sub-section following the next subsection.

Well-Supported Interpretations

Central to **C4** is the idea of a well-supported interpretation as expounded in Chapter 4. We recall below the intuition behind the idea of well-supported interpretations and extend it to **C5**.

If we think of the body of a sentence as providing evidence for attributing a certain truth-value to the head of the sentence, then a well-supported interpretation can be seen as assigning only that truth-value to any atom which can be justified in terms of the total evidence for it (with respect to that interpretation), where the evidence must be independent of the truth-value assigned to that atom and must be finitely grounded in the facts. The well-founded ordering ensures that the truth-values assigned to an atom is not justified in terms of itself and the evidence is finitely grounded. Thus, for instance, no well-supported interpretation of a program would assign true to p simply on the basis of the sentence $p \leftarrow p$.

It is this idea which we wish to generalize. However, in the case of extended logic programs we can have the evidence for the ground literal p provide indirect evidence for the ground literal $\neg p$ and vice versa. In many cases this is quite legitimate. Thus, consider the ground program:

$$P1: \quad \neg p \leftarrow q; \; q \leftarrow true; \; p \leftarrow unknown$$

Here we are clearly justified in assigning T to $\neg p$ and thus indirectly justified in assigning F to p even though the only sentence with p in the head has *unknown* in the body.

Contrast this with the ground program $\{p \leftarrow unknown; \; \neg p \leftarrow unknown\}$. Here assigning anything other than U to p or to $\neg p$ is unjustified. So the interpretation which assigns F to p and T to $\neg p$ should be unsupported even though the assignment of T to $\neg p$ is indirectly supported by the assignment of F to p, and vice versa.

Thus, in our generalization of the definition of well-supported interpretation we should allow for indirect support to a literal (by its negation), but we should not allow a literal and its negation to indirectly support each other.

As in **C4**, the attribution of a non-default truth value to a non-default literal must be based on evidence for that literal. In **C4** the default truth value is F and so in **C4** F can be assigned to any non-default literal on the basis of no evidence for that non-default literal. But in **C5** the default truth value is U and hence the attribution of F to a non-default literal must be based on evidence. Recall that having no evidence for a non-default literal means either having no information in support of that literal or having only false information in support of that literal. Recall that we treat all information in support of a non-default literal l as false (relative to an interpretation) in case the body of each rule with l or $Atom(l)$ in the head evaluates to F in that interpretation. Thus, in that case l cannot be assigned any non-default truth value in **C5**, not even F, by a well-supported interpretation.

Definition 11.3.1 *An interpretation $\mathcal{I}$ of an extended logic program LP is* well supported *if there exists a strict well-founded partial ordering $\ll$ on the atoms in HB_{LP} such that for any literal l in EHB_{LP} such that $U <_s \mathcal{I}'(l)$, there exists an $R \in grd(LP)$ such that*

- *$head(R) = l$ and $\mathcal{I}'(l) \leq_s \mathcal{I}'(body(R))$, or*
- *$head(R) = \neg l$ and $\mathcal{I}'(\neg l) \leq_s \mathcal{I}'(body(R))$, and*
- *$F <_s \mathcal{I}'(body(R))$, and*
- *$b \ll Atom(l)$ for every $b \in Atoms(objbody(R))$.*

In this case we say that the truth value of l is supported by R in $\mathcal{I}$. We say that the truth value of l is directly supported in $\mathcal{I}$ if the truth value of l is supported by a R such that $head(R) = l$. Otherwise, we say the truth value of l is indirectly supported in $\mathcal{I}$ by $\neg l$.

Thus the assignment of any truth value other than the default one (U) to an objective literal requires a non-circular, finite justification, but the assignment can be justified in terms of a rule whose body is assigned a higher truth value (in the supported ordering) than the literal. Note that although an interpretation is a mapping from *atoms* to truth values and although the well-founded ordering is over the *atoms* in HB_{LP}, the 'assignment'of a truth value to each *objective literal* must be justified. This is required in order to ensure that the assignment of truth vlaues to an objective literal and its negation do not justify each other. We assume that the special atoms ($true$, $Ctrue$, etc.) are not ordered with respect to each other and are less than any other atoms in the ordering.

This definition is a five-valued generalization of the definition of a four-valued well-supported interpretation in Chapter 4.

Note that if $\mathcal{I}$ is a well-supported interpretation of LP then the truth value of every literal in the EHB_{LP} is either directly or indirectly well-supported in $\mathcal{I}$. Furthermore, for any literal l and its negation, if the truth value of one of them is indirectly supported in $\mathcal{I}$ then the truth value of other must be directly supported. However, the truth value of both can be directly well-supported in $\mathcal{I}$ as in the ground program

$$P2: \ \{p \leftarrow \ true; \ \neg p \leftarrow \ true\}$$

It should be noted that in the program $P2$ the assignment of T to p and the assignment of T to $\neg p$ would both be directly supported by the program; however, this does not describe any interpretation of the program. Since an interpretation is a mapping from the *atoms* in HB_{LP} to the truth values, assigning T to p perforce assigns F to $\neg p$.

Negation as Contestation

The previous section introduced the idea of well-supported interpretations of extended logic programs. Now we consider the issue of when a well-supported interpretation is a model of an extended logic program.

Consider the following ground program.

$$P3: \ \{\neg b \leftarrow \ c; \ c \leftarrow; \ b \leftarrow\}$$

An interpretation which assigns T to b and c is a well-supported interpretation, but is it a model of P3? If the evaluation $\mathcal{I}'$ is used to evaluate the first clause, then it evaluates to F and thus this interpretation cannot be a model of P3. Indeed, it can be easily seen that if $\mathcal{I}'$ is used to evaluate the clauses of P3, then P3 has no models. Note that in P3 there is direct evidence for assigning T to both b and $\neg b$, but assigning T to one of these literals puts a restriction on the truth value that can be assigned to the other literal. Thus, to come up with a suitable model theory for extended logic programs we need to revise the evaluation function $\mathcal{I}'$ to take into account the restriction that the assignment of a truth value to a literal places on the assignment of a truth value to its non-default negation. We do this below using the apparatus of contestations developed in the previous chapters.

Chapter 4 introduced the idea of contestations. A contestation $A \hookrightarrow b$ says that A provides evidence against b. Chapter 4 provides a semantics for a normal logic program LP constrained by a set of contestations $\mathcal{C}$. Each rule in LP is evaluated under these constraints. Thus, suppose we have the ground program $\{ b \leftarrow c;\ c \leftarrow;\ a \leftarrow \}$ and the contestation $a \hookrightarrow b$ and an interpretation $\mathcal{I}$ which assigns T to a and T to c. So although relative to that interpretation there is enough evidence for assigning T to b, the evidence provided by a against b puts a *cap* on how much overall evidence can be said to exist for b.

In a similar fashion the assignment of a certain truth value to l can put a *cap* on the truth value that can be assigned to $\neg l$. Consider the program P3. Consider the interpretation which assigns T to b and c. Although relative to that interpretation there is enough evidence to assign T to $\neg b$, the evidence provided by b against $\neg b$ puts a *cap* on how much overall evidence there can be said to exist for $\neg b$.

This aspect of non-default negation can be captured in terms of the apparatus of contestations in the following manner.

First, the extended logic program LP is augmented with the following special set of contestations

$$\mathcal{C}_\neg \;=\; \{p \hookrightarrow_\neg \; \neg p\} \;\cup\; \{\neg p \hookrightarrow_\neg p\}$$

for each $p \in HB_{LP}$.

Associated with each such contestation is the following *cap* function:

$$cap_\neg(\alpha) \;=\; \begin{cases} \mathbf{NEG}(\mathcal{I}'(\neg l)) & \text{if } \alpha \neq U \\ T & \text{if } \alpha = U \end{cases}$$

Corresponding to Definition 4.4.3 of Chapter 4, we define a function cap' which takes an objective literal, a contestation and an interpretation as arguments and returns a special atom as a value.

Definition 11.3.2 *Let l be an objective literal, not necessarily ground, $\mathcal{C}_j$ be a contestation with the associated cap function $cap_\neg$. Then, $cap'_\neg(l, \mathcal{C}_j, \mathcal{I})$ returns the special atom which always evaluates to $cap_\neg(\mathcal{I}(Contestor(C_j)))$ if $Contested(\mathcal{C}_j) = l\theta$, for some substitution θ which can be the empty substitution, otherwise $cap'_\neg(l, \mathcal{C}_j, \mathcal{I})$ returns the special atom true.*

Note that $\mathcal{C}_\neg$ contains only contestations of the form $l \hookrightarrow_\neg \; \neg l$. Hence in the above definition if $Contested(\mathcal{C}_j) = l\theta$, for some substitution θ, $Contestor(\mathcal{C}_j)$ must be $\neg l\theta$. In light of this, the above cap' function can be simplified as

$$cap'_\neg(l, \mathcal{C}_j, \mathcal{I}) \;=\; \begin{cases} \mathbf{NEG}(\mathcal{I}'(\neg l\theta)) & \text{if } Contested(\mathcal{C}_j) = l\theta \text{ and } \mathcal{I}'(\neg l\theta) \neq U \\ T & \text{otherwise} \end{cases}$$

Recall that in a well-supported model the assignment of a truth value to a literal l can be either directly supported in terms of a sentence with l in the head or indirectly supported in terms of the truth value of $\neg l$. If an interpretation $\mathcal{I}$ assigns a truth-value $\mathbf{v} \in \mathcal{V}$ to l then, by the structural principle, the truth value of $\neg l$ must be $\mathbf{NEG(v)}$. But if the assignment of $\mathbf{v}$ to l is directly supported in $\mathcal{I}$ then we can think of it as providing evidence against $\neg l$ and thus providing a *cap* on what truth-value can be assigned to $\neg l$. On the other hand, if the assignment of $\mathbf{v}$ to l is *indirectly* supported then it provides no independent evidence against $\neg l$ and thus cannot be seen as providing a *cap* on what truth value can be assigned to $\neg l$. Rather, the situation is reversed. It is the assignment of $\mathbf{NEG(v)}$ to $\neg l$ that provides evidence against l. This is because if the assignment of $\mathbf{v}$ is indirectly supported then the assignment of $\mathbf{NEG(v)}$ to $\neg l$ is directly supported. It could happen that the assignment of $\mathbf{v}$ to l and the assignment of $\mathbf{NEG(v)}$ to $\neg l$ are both directly supported in $\mathcal{I}$ (as in the program $P2$ above) and in this case we view them as providing evidence against each other.

This motivates the following modification of $\mathcal{I}'$.

Definition 11.3.3 *Given a well-supported interpretation $\mathcal{I}$ of an extended logic program LP, $\mathcal{I}'''$ is a function for evaluating any closed sentence in the language of LP. $\mathcal{I}'''$ is just like $\mathcal{I}'$ for all operators except $\leftarrow$.*
If $S1$ and $S2$ are (closed) sentences and θ is any substitution, then $\mathcal{I}'''(S1 \leftarrow S2)\theta$ is just $\mathcal{I}'(S1 \leftarrow S2)\theta$ if $\neg S1\theta$ is not directly supported in $\mathcal{I}$; otherwise $\mathcal{I}'''(S1 \leftarrow S2)\theta$ is $\mathcal{I}'((S1 \leftarrow S2)\theta,\ cap'_{\neg}(S1\theta, \mathcal{C}_{\neg}, \mathcal{I}))$.

$\mathcal{I}'''$ provides a way of taking into account the justified reductions of the assignment of a truth value to the head of a sentence in evaluating the truth value of that sentence. Note that the definition of $\mathcal{I}'''$ makes reference to the notion of well-supportedness, which is defined using $\mathcal{I}'$ for evaluating the sentences of the program. Thus, in determining whether an interpretation is well-supported we make use of $\mathcal{I}'$ rather than $\mathcal{I}'''$, otherwise our definition would be circular.

Thus, a well-supported interpretation $\mathcal{I}$ is a model of an extended logic program LP if and only if all the rules of LP evaluate to either CT or T under $\mathcal{I}'''$. Such models are the well-supported models of LP.

Semantics of Extended Logic Programs

The clausal ordering among interpretations defined in Chapter 4 assumed the $\mathcal{I}''$ function for evaluating the sentences of a program relative to an interpretation. We define below a form of clausal ordering among well-supported models using $\mathcal{I}'''$ for evaluating sentences.

Definition 11.3.4 *Let $\mathcal{I}_1$ and $\mathcal{I}_2$ be two well-supported models of LP. Then, $\mathcal{I}_1 \leq_{LP}^{mod} \mathcal{I}_2$ if, and only if, $\mathcal{I}_1(C) \leq_s \mathcal{I}_2(C)$ for every sentence C in LP where the sentences are evaluated using $\mathcal{I}'''$.*

We call this ordering the *mclausal* ordering among well-supported models.

As before, we say that an interpretation $\mathcal{I}_i$ is *maximal with respect to LP* in a set of interpretations ν if there is no interpretation $\mathcal{I}_j \in \nu$ such that $\mathcal{I}_i <_{LP}^{mod} \mathcal{I}_j$.

Definition 11.3.5 *The canonical models of an extended logic program LP under* **C5** *are the maximal models in terms of the mclausal ordering among the well-supported models.*

Example 11.3.1 *Let LP be the following program from [DR91].*

$$\begin{array}{llll} C_1: & \neg fly(x) & \leftarrow & \textbf{not } bird(x) \\ C_2: & fly(x) & \leftarrow & bat(x) \\ C_3: & bat(tom) & \leftarrow & \end{array}$$

We give below the well-supported models of the program and the evaluation of each sentence in a well-supported model using $\mathcal{I}'''$.

	fly(tom)	bird(tom)	bat(tom)	C_1	C_2	C_3
$\mathcal{I}_1$	T	U	T	T	T	T
$\mathcal{I}_2$	CT	U	T	T	T	T
$\mathcal{I}_3$	CF	U	CT	T	T	T
$\mathcal{I}_4$	F	U	CT	T	T	T

Table 11.3: The **C5** well-supported models of an extended logic program.

Note that C_1 and C_2 evaluate to T in these interpretations because in the evaluation of those rules we view the program as being implicitly augmented with $\mathcal{C}_\neg = \{fly(tom) \hookrightarrow_\neg \neg\, fly(tom),\ \neg\, fly(tom) \hookrightarrow_\neg\ fly(tom)\}$.

All of these interpretations are the well-supported models of the program and all of them are canonical.

Similar to the skeptical and credulous versions of **C4**, we can define skeptical and credulous versions of **C5**. A *skeptical* version of **C5** identifies the meaning of an extended logic program LP with the literals that evaluate to T in all the canonical models of LP under **C5**, whereas a *credulous* version of **C5** identifies the meaning of an extended logic program LP with the literals that evaluate to CT under **C5**.

The following theorem establishes that the **C5** semantics is inferentially conflict-free with respect to the types of conflicts that can be expressed in terms of the $\neg$ operator. Recall that in Chapter 1 we had noted that almost all paraconsistent logics permit the inference of logically inconsistent sentences. To the extent that $\neg p$ and p, for any sentence p, expresses a logical inconsistency, the following theorem establishes that **C5** is inferentially free of logical inconsistencies.

Theorem 11.3.1 *The* **C5** *semantics for extended logic programs is inferentially consistent.*

Proof: To prove this we need to prove that for any extended logic program P and any ground atom p, P does not entail both p and $\neg p$ under the **C5** semantics. This follows from the fact that in no **C5** interpretation can both p and $\neg p$ evaluate to at least CT, since a **C5** interpretation is a mapping from *atoms* to truth values. ■

11.4 Relation to Answer Set Semantics

In this section we introduce the answer set semantics of Gelfond and Lifschitz for extended logic programs ([GL90]). We prove that an extended logic program LP with a consistent answer set entails a literal p with respect to the answer sets of LP if, and only if, LP weakly entails p (under **C5**).

The answer set semantics is a generalization of the stable model semantics introduced in Chapter 3. Gelfond and Lifschitz define an answer set in two steps: the generalized Gelfond-Lifschitz transformation of a program and the α operator. These are explained below.

Definition 11.4.1 Let P be a ground, extended logic program. Let M be a set of ground objective literals. Then, the generalized Gelfond-Lifschitz transformation ([GL90]) of P is

$$P^M = \{a \leftarrow b_1, \ldots, b_k \mid a \leftarrow b_1, \ldots, b_k, \mathbf{not}\ c_1, \ldots, \mathbf{not}\ c_n \in P,\ c_1, \ldots, c_n \notin M\}$$

Note that P^M contains no default literals.

Given an extended logic program rule $R = a \leftarrow b_1, \ldots, b_k, \mathbf{not}\ c_1, \ldots, \mathbf{not}\ c_n$ and a set of ground objective literals M, let

$$R^M = \begin{cases} a \leftarrow b_1, \ldots, b_k & \text{if } c_1, \ldots, c_n \notin M \\ a \leftarrow \ false & \text{otherwise} \end{cases}$$

Let α be an operator such that for a program P containing no occurrence of default literals, $\alpha(P)$ is $S \subseteq EHB_P$ where S is the smallest set such that

1. for any rule $a \leftarrow b_1, \ldots, b_m$ in P^M, if $b_1, \ldots, b_m \in S$ then $a \in S$

2. if S contains l and $\neg l$ for any literal l, then $S = EHB_P$

M is an answer set of P if and only if $M = \alpha(P^M)$.

Example 11.4.1 *Let P be*

$$\begin{aligned} cross\_track &\leftarrow \neg train\_approaching,\ \mathbf{not}\ stop\_request \\ \neg stop &\leftarrow \mathbf{not}\ stop\_request \\ stop &\leftarrow stop\_request \\ \neg train\_approaching &\leftarrow \end{aligned}$$

Let $M = \{cross\_track,\ \neg train\_approaching, \neg stop\}$. Then P^M is

$$\begin{aligned} cross\_track &\leftarrow \neg train\_approaching \\ \neg stop &\leftarrow \\ stop &\leftarrow stop\_request \\ \neg train\_approaching &\leftarrow \end{aligned}$$

$\alpha(P^M) = \{cross\_track,\ \neg train\_approaching, \neg stop\}$. Thus, $M = \alpha(P^M)$ and M is an answer set of P.

When P is a normal logic program, the answer set semantics reduces to the stable model semantics. In this case P^M is a definite program, $\alpha(P^M)$ is the unique minimal model of the program and if $M = \alpha(P^M)$, then M has to be a set of atoms.([GL90]).

Gelfond and Lifschitz show how an extended logic program P can be reduced to a normal logic program P^+ by replacing each rule of R by its *positive* form. The positive form of

$$l_0 \leftarrow l_1, \ldots, l_m, \mathbf{not}\ l_{m+1}, \ldots, \mathbf{not}\ l_n$$

is a rule of the form

$$l_0^+ \leftarrow l_1^+, \ldots, l_m^+, \mathbf{not}\ l_{m+1}^+, \ldots, \mathbf{not}\ l_n^+$$

where l_i^+ is l_i if l_i is an atom, otherwise if l_i is a negative objective literal then l_i^+ is the new atom $(l_i)'$. They prove that

Theorem 11.4.1 *[GL90]*
M is a consistent answer set of an extended logic program P if and only if M^+ is an answer set of P^+

The following lemma uses the above theorem to set up a connection between answer sets and two-valued well-supported models. We use this lemma in the proof Lemma 11.4.3 below.

Lemma 11.4.1 *M is a consistent answer set of an extended logic program P if and only if M^+ is a two-valued well-supported model of P^+*

Proof: Since P^+ is a normal logic program, M^+ is also a stable model of P^+. By Theorem 3.3.1 in Chapter 3, M^+ must also be a two-valued well-supported model of P^+. ∎

Definition 11.4.2 *Given an interpretation $\mathcal{I}$, let $ETruth(\mathcal{I})$ denote $\{l|$ l is an objective literal and $\mathcal{I}(l) \geq CT\}$.*

The following lemma shows that $ETruth(\mathcal{I})$ is always consistent for any interpretation $\mathcal{I}$.

Lemma 11.4.2 *For any interpretation $\mathcal{I}$, $ETruth(\mathcal{I})$ is a consistent set, i.e., it is not the case that both a and $\neg a$ belongs to $ETruth(\mathcal{I})$ for any atom a.*

Proof: Since $\mathcal{I}$ is a mapping from *atoms* to truth values, $\mathcal{I}(\neg a) \geq CT$ if and only if $\mathcal{I}(a) < CT$. Hence $ETruth(\mathcal{I})$ must be consistent. ∎

Lemma 11.4.3 below says that every consistent answer set of an extended program P is $ETruth(\mathcal{I})$ for some canonical model $\mathcal{I}$ (under **C5**) of P.

Lemma 11.4.3 *Let LP be an extended logic program and let P be $grd(LP)$. Then, for each consistent answer set M of P, there exists a five-valued canonical model $\mathcal{I}$ of LP such that $M = ETruth(\mathcal{I})$.*

Proof: Let M be a consistent answer set of P. We show below how to construct a five-valued canonical model $\mathcal{I}$ such that $M = ETruth(\mathcal{I})$.

Let $\mathcal{I}$ be such that it assigns T to all members of M and F to the atoms of all the negative literals in M and U to all other atoms in HB_{LP}. Clearly, by construction $M = ETruth(\mathcal{I})$. We show below that $\mathcal{I}$ is a five-valued canonical model of P.

$\mathcal{I}$ is well-supported

Since M is a consistent answer set of P, it follows that M^+ is a two-valued well-supported model of P^+ (by Lemma 11.4.1 above). So there must be well-founded ordering $\ll^+$ by which M^+ is well-supported. For each l_i^+ that occurs in this ordering, l_i is an atom if $l_i^+ = l_i$ and otherwise if l_i^+ is l_i', then l_i is a negative objective literal. We derive a well-founded ordering $\ll$ from $\ll^+$ by substituting $Atom(l_i)$ for each l_i^+ such that $l_i^+ = l_i'$. We show that $\mathcal{I}$ is a well-supported model in terms of this ordering of the atoms in HB_P.

Since the only non-default truth values assigned to any atom by $\mathcal{I}$ are T and F, we need only show that $\mathcal{I}$ is well-supported regarding these truth values.

If $\mathcal{I}$ assigns T to an atom a then $a \in M$. So there must be a rule $R^M \in P^M$ such that $head(R^M) = a$ and $body(R^M) \subseteq M$. Thus, by construction, $\mathcal{I}(body(R^M))$ must evaluate to T. Hence, there must be a rule $R \in P$ such that $R = a \leftarrow body(R^M), \textbf{not } c_1, \ldots, \textbf{not } c_n$, such that $c_1, \ldots, c_n \notin M$. So $\mathcal{I}$ assigns U to each of $c_1, \ldots, c_n$. Thus, $body(R)$ must evaluate to T in $\mathcal{I}$. Hence the attribution of T to a by $\mathcal{I}$ is directly supported through R. Furthermore, the attribution of T to a must be well-supported since if $a \in M$ then $a \in M^+$. M^+ can be understood as attributing T to a. It is easy to see that since the attribution of T to a by M^+ is well-supported in terms of $\ll^+$, the attribution of T to a by $\mathcal{I}$ must be well-supported in terms of $\ll$.

If $\mathcal{I}$ assigns F to an atom a then $\neg a \in M$. So there must be a rule $R^M \in P^M$ such that $head(R^M) = \neg a$ and $body(R^M) \subseteq M$. By reasoning similar to the previous case it follows that $body(R)$ must evaluate to T in $\mathcal{I}$. Hence the attribution of F to a by $\mathcal{I}$ is indirectly supported through R. Furthermore, the attribution of F to a must be well-supported since if $\neg a \in M$ then $(\neg a)' \in M^+$. M^+ can be understood as attributing T to $(\neg a)'$. It is easy to see that since the attribution of T to $(\neg a)'$ by M^+ is well-supported in terms of $\ll^+$, the evaluation of T to $\neg a$ by $\mathcal{I}$ must be well-supported in terms of $\ll$. Thus, the attribution of F to a by $\mathcal{I}$

must be *indirectly* well-supported in terms of $\ll$.

$\mathcal{I}$ is a model of P

Assume by way of contradiction that $\mathcal{I}$ is not a model of P. Since $\mathcal{I}$ does not attribute CF or CT to any atoms, $\mathcal{I}$ fails to be a model of P only if there is a $R \in P$ such that $\mathcal{I}'''(R)$ is F or U.

There can be no $R \in P$ such that $\mathcal{I}'''(R) = U$ since an R is assigned U only if the truth values of $head(R)$ and $body(R)$ are incomparable. But, since $\mathcal{I}$ assigns only T, F and U, $head(R)$ and $body(R)$ cannot have incomparable values.

So assume $\mathcal{I}'''(R)$ evaluates to F. So either $body(R)$ evaluates to T and $head(R)$ evaluates to F or U, or $body(R)$ evaluates to U and $head(R)$ evaluates to F. But $body(R)$ evaluates to T only if $body(R^M) \subseteq M$. In that case $head(R) = head(R^M) \in M$ and so $head(R)$ would be T in $\mathcal{I}'''$ and thus $\mathcal{I}'''(R) = T$. Hence we need consider only the case where $head(R)$ evaluates to F and $body(R)$ evaluates to U.

Either $head(R) = a$ or $head(R) = \neg a$ for some atom a. If $head(R) = a$ and a is assigned F, then, by the way $\mathcal{I}$ is constructed, $\neg a \in M$. So there must be a rule $R_1 \in P$ such that $head(R_1) = \neg a$ and $\mathcal{I}(\neg a) = T$. Since $\mathcal{I}$ is well-supported as shown above, the attribution of T to $\neg a$ must be directly supported in $\mathcal{I}$. So by the evaluation rule $\mathcal{I}'''$, in evaluating any rule with a in the head, the expression $cap_{\neg}(a, \mathcal{C}_{\neg}, \mathcal{I})$ must be inserted in the body of any rule which has a as its head. This cap expression evaluates to F in $\mathcal{I}$. So $body(R)$ must evaluate to F in $\mathcal{I}$ according to the **AND** function. So $\mathcal{I}'''(R) = T$.

If $head(R) = \neg a$ and $\neg a$ is assigned F, then, by the way $\mathcal{I}$ is constructed, $a \in M$. So there must be a rule $R_1 \in P$ such that $head(R_1) = a$ and $\mathcal{I}(a) = T$. Since $\mathcal{I}$ is well-supported as shown above, the attribution of T to a must be directly supported in $\mathcal{I}$. So by the evaluation rule $\mathcal{I}'''$, in evaluating any rule with $\neg a$ in the head, the expression $cap_\neg(\neg a, \mathcal{C}_\neg, \mathcal{I})$ must be inserted in the body of any rule which has $\neg a$ as its head. This cap expression evaluates to F in $\mathcal{I}$. So $body(R)$ must evaluate to F in $\mathcal{I}$ according to the **AND** function. So again $\mathcal{I}'''(R) = T$.

Hence, given the way $\mathcal{I}$ has been constructed, there cannot be any rules in P such that they evaluate to F or U using the evaluation function $\mathcal{I}'''$. Thus, $\mathcal{I}$ must be a model of P.

$\mathcal{I}$ is maximal in the mclausal ordering with respect to LP

$\mathcal{I}$ is maximal in the mclausal ordering with respect to LP only if it is maximal in the mclausal ordering with respect to $grd(LP) = P$. To establish that $\mathcal{I}$ is maximal in the clausal ordering with respect to P it is enough to establish that all rules in P evaluate to T according to $\mathcal{I}'''$.

Clearly, all rules such that its head is assigned T by $\mathcal{I}$ evaluate to T. So, all that remains to be shown is that all rules such that its head is assigned F or U by $\mathcal{I}$ also evaluate to T. But since we have already established that $\mathcal{I}$ is a model of P, the body of any rule whose head is assigned F must evaluate to F. Thus, any such rule evaluates to T in $\mathcal{I}$. Similarly, since $\mathcal{I}$ is a model of P, any rule whose head is assigned U in $\mathcal{I}$ cannot have its body evaluate to T. So any such rule must also evaluate to T.

Hence, all rules in P evaluate to T under $\mathcal{I}$.

Hence, $\mathcal{I}$ is a canonical model of P and $M = ETruth(\mathcal{I})$. ■

Corollary 3 *If a ground program P has a consistent answer set, then P has a five-valued canonical model $\mathcal{I}$ such that each rule in P evaluates to T in $\mathcal{I}$.*

Proof: This was essentially proved in the proof of the previous lemma. ■

Corollary 4 *If a ground program P has a consistent answer set, then every five-valued canonical model of P is such that each rule in P evaluates to T in it.*

Proof: By Corollary 3 we know that if P has a consistent answer set then there is a canonical model $\mathcal{I}$ of P such that all rules evaluate to T in $\mathcal{I}$. So for any model $\mathcal{J}$ of P such that for any rule R, $\mathcal{J}(R) < T$, it must be the case that $\mathcal{J} < \mathcal{I}$. Hence $\mathcal{J}$ cannot be canonical. Thus, every canonical model must be such that every rule of P evaluates to T in it. ∎

Lemma 11.4.4 *Let P be $grd(LP)$. Every canonical five-valued model $\mathcal{I}$ of LP such that each rule of P evaluates to T in $\mathcal{I}$ is such that $ETruth(\mathcal{I})$ is an answer set of P.*

Proof: Assume that $\mathcal{I}$ is a canonical model of LP such that each rule of P evaluates to T in $\mathcal{I}$.

We prove the lemma in three steps.

1. We show that if each rule of P evaluates to T in $\mathcal{I}$, then $\alpha(P^{ETruth(\mathcal{I})})$ is a consistent set.

2. We show that if $\alpha(P^{ETruth(\mathcal{I})})$ is consistent then every member of $\alpha(P^{ETruth(\mathcal{I})})$ is also a member of $ETruth(\mathcal{I})$.

3. We show that every member of $ETruth(\mathcal{I})$ is also a member of $\alpha(P^{ETruth(\mathcal{I})})$.

Step 1: Assume by way of contradiction that $\alpha(P^{ETruth(\mathcal{I})})$ is an inconsistent set. So there must be literals l and $\neg l$ in $\alpha(P^{ETruth(\mathcal{I})})$ and rules R_1 and R_2 in $P^{ETruth(\mathcal{I})}$ such that

- $head(R_1) = l$

- $head(R_2) = \neg l$
- $body(R_1) \subseteq \alpha(P^{ETruth(\mathcal{I})})$
- $body(R_2) \subseteq \alpha(P^{ETruth(\mathcal{I})})$.

Thus, there must be rules R_1' and R_2' in P such that

- $head(R_1) = l = head(R_1')$
- $head(R_2) = \neg l = head(R_2')$
- $body(R_1) = objbody(R_1')$
- $body(R_2) = objbody(R_2')$.

Hence all the default literals and the non-default literals in $body(R_1')$ and $body(R_2')$ must evaluate to at least CT in $\mathcal{I}$. However, both l and $\neg l$ cannot evaluate to at least CT in $\mathcal{I}$ by Lemma 11.4.2. Hence, both R_1' and R_2' cannot evaluate to T in $\mathcal{I}$, which contradicts the assumption that all rules of P evaluate to T in $\mathcal{I}$. Thus, $\alpha(P^{ETruth(\mathcal{I})})$ must be a consistent set if all rules of P evaluate to T in $\mathcal{I}$.

Step 2: We show that if $\alpha(P^{ETruth(\mathcal{I})})$ is a consistent set, then every member of $\alpha(P^{ETruth(\mathcal{I})})$ is a member of $Etruth(\mathcal{I})$.

From Step 1 we know that $S = \alpha(P^{ETruth(\mathcal{I})})$ is a consistent set under the assumption that all rules in P evaluate to T in $\mathcal{I}$. Clearly, S is a consistent answer set of $P^{ETruth(\mathcal{I})}$. Thus, by Theorem 11.4.1, S^+ is an answer set of $(P^{ETruth(\mathcal{I})})^+$. We show below that for any objective literal l, if $l^+ \in S^+$ then $l \in Etruth(\mathcal{I})$. Since $l^+ \in S^+$ if and only if $l \in S = \alpha(P^{ETruth(\mathcal{I})})$, this establishes that every member of $\alpha(P^{ETruth(\mathcal{I})})$ is a member of $Etruth(\mathcal{I})$.

$(P^{ETruth(\mathcal{I})})^+$ is a definite logic program and, thus, $S^+ = \alpha((P^{ETruth(\mathcal{I})})^+)$ is the unique minimal model of $(P^{ETruth(\mathcal{I})})^+$. Thus, by Theorem 3.2.2 (the van Emden-Kowlaski Theorem), S^+ is the least fix-point of $T_Q \uparrow n$, where Q is $(P^{ETruth(\mathcal{I})})^+$. Thus, it is possible to stratify the members of S^+ in terms of the least n such that a member first occurs in $T_Q \uparrow n$.

In the following let us call an objective literal l the *original form* of l^+. Recall that l^+ is called the *positive form* of l.

Let l^+ be of the lowest strata among those objective literals in S^+ such that their original form is not in $ETruth(\mathcal{I})$.

Since $l^+ \in S^+$, there must be a rule in $(P^{ETruth(\mathcal{I})})^+$ of the form $l^+ \leftarrow (b_1)^+, \ldots, (b_m)^+$ such that $\{(b_1)^+, \ldots, (b_m)^+\} \subseteq S^+$. But, by the assumption that l^+ is of the lowest strata among those literals in S^+ whose original forms are not in $ETruth(\mathcal{I})$, it follows that $\{b_1, \ldots, b_m\} \subseteq Truth(\mathcal{I})$. Furthermore, since $l^+ \leftarrow (b_1)^+, \ldots, (b_m)^+$ is in $(P^{ETruth(\mathcal{I})})^+$, there must be a rule R in P of the form $l \leftarrow b_1, \ldots, b_m, \mathbf{not}\ c_1, \ldots, \mathbf{not}\ c_n$ such that $c_i \notin ETruth(\mathcal{I})$, $i = 1, \ldots, n$. So, each member b_i of $objbody(R)$ is assigned at least CT by $\mathcal{I}$ (since each such b_i belongs to $ETruth(\mathcal{I})$) and each member $\mathbf{not}\ c_j$ of $negbody(R)$ evaluates to at least CT (since each c_j is assigned at most CF by $\mathcal{I}$). Hence, $\mathcal{I}(body(R))$ is at least CT. By the assumption that R evaluates to T in $\mathcal{I}$, it follows that $\mathcal{I}(l)$ must be at least CT. Therefore, l must be in $ETruth(\mathcal{I})$. Thus, for every objective literal l, if $l^+ \in S^+$, then $l \in Etruth(\mathcal{I})$. And, hence, every member of $S = \alpha(P^{ETruth(\mathcal{I})})$ is a member of $ETruth(\mathcal{I})$.

Step 3: Assume by way of contradiction that there is $a \in ETruth(\mathcal{I})$ and $a \notin \alpha(P^{ETruth(\mathcal{I})})$.

Let $\ll$ be the well-founded ordering that makes $\mathcal{I}$ well supported. Among all the atoms x such that $x \notin \alpha(P^{ETruth(\mathcal{I})})$ and $x \in ETruth(\mathcal{I})$, let a be highest in terms of $\ll$. That is, let a be such that there does not exist a b such that $b \notin \alpha(P^{ETruth(\mathcal{I})})$ and $b \in ETruth(\mathcal{I})$ and $a \ll b$.

Since $a \in ETruth(\mathcal{I})$, $\mathcal{I}(a)$ is at least CT and, hence, there must be a rule R in P of the form $a \leftarrow b_1, \ldots, b_m, \textbf{not } c_1, \ldots, \textbf{not } c_n$ such that $body(R)$ must evaluate to at least CT under $\mathcal{I}$ (otherwise, $\mathcal{I}$ would not be well-supported). So each b_i in $objbody(R)$ must be assigned at least CT by $\mathcal{I}$. Thus, $\{b_1, \ldots, b_m\} \subseteq ETruth(\mathcal{I})$. Furthermore, since each **not** c_j in $body(R)$ must evaluate to at least CT, each c_j must be assigned at most CF by $\mathcal{I}$. Thus, no c_j is in $ETruth(\mathcal{I})$.

Hence, clearly, $a \leftarrow b_1, \ldots, b_m$ must be in $P^{ETruth(\mathcal{I})}$.

By the nature of the well-founded ordering that makes $\mathcal{I}$ well supported, each of $b_1, \ldots, b_m$ must be lower than a in the well-founded ordering (otherwise a cannot be well-supported by R). By our assumption that a is the highest in the well-founded ordering, it follows that $\{b_1, \ldots, b_m\} \subseteq \alpha(P^{ETruth(\mathcal{I})})$ since $\{b_1, \ldots, b_m\} \subseteq ETruth(\mathcal{I})$. So a must belong to $\alpha(P^{ETruth(\mathcal{I})})$. Thus, a contradiction. Hence, every member of $ETruth(\mathcal{I})$ must be a member of $\alpha(P^{ETruth(\mathcal{I})})$.

Steps 1 and 2 together show that if each rule of P evaluates to T in $\mathcal{I}$, then every member of $\alpha(P^{ETruth(\mathcal{I})})$ is also a member of $ETruth(\mathcal{I})$. This together with Step 3 proves the lemma. ■

Lemma 11.4.5 *Let P be $grd(LP)$. If P has a consistent answer set then every canonical model $\mathcal{I}$ of LP is such that $ETruth(\mathcal{I})$ is an answer set of P.*

Proof: Follows directly from Corollary 4 and Lemma 11.4.4. ■

Theorem 11.4.2 *If a ground extended logic program P has a consistent answer set, then M is an answer set of P if, and only if, there exists a five valued canonical* *model of $\mathcal{I}$ of P such that $M = ETruth(\mathcal{I})$.*

Proof: Follows directly from Lemmas 11.4.3 and 11.4.5. ∎

Since not all extended logic programs have consistent answer sets, an important question is what are the necessary and sufficient conditions for an extended logic program having a consistent answer set. The following theorem gives an answer.

Theorem 11.4.3 *A ground extended logic program has a consistent answer set if, and only if, every rule of the program evaluates to T in every canonical model of the program.*

Proof: The left-to-right direction is proven in Corollary 4. The right-to-left direction is proven in Lemma 11.4.4. ∎

Let us say that P entails a sentence q *under the answer set semantics* if, and only if, q is a member of every answer set of P.

Theorem 11.4.4 *If a ground extended logic program P has any consistent answer sets then it entails a sentence q under the answer set semantics if, and only if, P weakly entails q under* **C5**.

Proof: Follows directly from Theorem 11.4.2. ∎

11.5 Discussion

As in the case of **C4**, the truth values of **C5** and the orderings among them can be seen as composed out of a more basic set of truth values and the orderings between them. In the case of **C4** the basic set of truth values is $\{T, F\}$ with only a truth ordering among them, whereas in the case of **C5** the basic set of truth values is $\{T, U, F\}$, which are themselves ordered along both the truth and information dimension thus: $F <_t U <_t T$ and $U <_i F, T$. As in the case of **C4**, we imagine two players assigning one of the basic truth values to a set of sentence. Player 1 has the final say in which truth value is assigned to a sentence. This gives rise to the following tuples of truth values, where the first member of each tuple is the truth value assigned by player 1 and the second member is the truth value assigned by player 2: $\langle T, T\rangle$, $\langle T, U\rangle$, $\langle T, F\rangle$, $\langle U, T\rangle$, $\langle U, U\rangle$, $\langle U, F\rangle$, $\langle F, T\rangle$, $\langle F, U\rangle$, $\langle F, F\rangle$.

As in case of **C4**, player 1 determines what truth value to assign to a sentence taking into account the truth values assigned by both players. In doing this player 1 adopts the policy that in case either player assigns a definite truth value (T or F) to a sentence and the other player assigns U, then the definite assignment should be allowed to win, but otherwise if both players assign a definite truth value then the assignment by player 2 should dominate the assignment by player 1 without winning outright. Call this 'the assignment policy'. This assignment policy means player 1 will finally assign T to a sentence in case of $\langle T, U\rangle$ or $\langle U, T\rangle$ and F in case of $\langle U, F\rangle$ or $\langle F, U\rangle$. It also means that $\langle T, F\rangle$ and $\langle F, T\rangle$ are *not* simplified to F and T respectively (which would be allowing the assignment by player 2 to win outright), but are instead are retained with the ordering $\langle T, F\rangle \ <_t \langle F, T\rangle$, which reflects the idea that the assignment by player 2 dominates the assignment by player 1. As in **C4**, $\langle T, F\rangle$ is represented by CF and $\langle F, T\rangle$ is represented by CT. If there is consensus in the truth values assigned to a sentence, player 1 will finally assign that truth value to the sentence. This gives us the truth values of **C5**.

In deriving the ordering among the truth values of **C5** we use the truth and information orderings in the basic set, $\{T, U, F\}$, as well as the assignment policy adopted by player 1. Given the truth ordering among the values of the basic set, clearly, the ordering $F <_t U <_t T$ among the truth values of **C5**is justified. Also, as in **C4**, $\langle T, F\rangle \ <_t \langle F, T\rangle$ since we allow player 2 to dominate player 1. That is, $CF <_t CT$. It is obvious also that $F <_t CF$ and $CT <_t T$.

How should $\langle U, U\rangle$ be ranked with respect to $\langle T, F\rangle$ and $\langle F, T\rangle$? Note that commonly U, which is regarded as the unknown truth value, is considered less than T and greater than F in the truth ordering because even if more information were provided about a sentence that is now regarded as unknown, that sentence will never be assigned a value greater than T and less than F. Using this reasoning we hold that $\langle U, U\rangle$ is incomparable in the truth ordering with respect to $\langle T, F\rangle$ and $\langle F, T\rangle$ because if more information were provided to both players regarding a sentence that they now regard as unknown in truth value, then in the worst case they both might regard it as false and in the best case they both might regard it as true. Thus, we have no way of locating $\langle U, U\rangle$ with respect to $\langle T, F\rangle$ and $\langle F, T\rangle$ in the truth ordering. That is, U is incomparable with respect to CT and CF.

Putting all this together we get the following truth ordering among the values of **C5**: $F <_t U <_t T$, $F <_t CF <_t CT <_t T$.

The information ordering among the truth values of **C5** is straight forward. Clearly $U \ <_i \ T, CT, CF, F$. And, furthermore, T, CT, CF, F are incomparable among themselves in terms of the information ordering.

In Chapter 5 we showed that **C4** subsumes and extends the stable model semantics for those normal logic programs which have any stable models. The answer set semantics generalizes the stable model semantics and **C5** generalizes **C4**. It is satisfying therefore that the results regarding the relation between stable model semantics for normal logic programs and the **C4** semantics for this class of programs holds in the more generalized setting of answer set semantics for extended logic programs and the **C5** semantics for this class of programs.

The answer set semantics inherits the problems of stable model semantics discussed in Chapter 5: there are no answer sets for some extended logic programs and the addition of an "irrelevant" clause to a program which has an answer set can result in a program which has no answer sets. The **C5** semantics for extended logic programs overcomes both these problems.

The answer set semantics also has the drawback that certain programs, the "inconsistent" programs, have only the trivial answer set which consists of all the objective literals in the extended Herbrand base of the program. This is clearly a drawback of the answer set semantics in that no meaningful inferences can be drawn from such programs on the basis of the answer set semantics. The **C5** semantics for extended logic programs does not suffer from this drawback.

In Chapter 5 we have also shown that **C4** subsumes the well-founded semantics ([GRS91]). It would therefore be desirable to show that **C5** subsumes well-founded semantics for extended logic programs. However, there is no agreement on what would count as *the* well-founded semantics for extended logic programs ([DR91], [AP92b], [ADP93], [Sak92]). Hence we have not attempted to show that **C5** semantics for extended logic programs subsumes well-founded semantics for extended logic programs.

Based on the idea of well-supported models and a five valued logic, **C5**, we have characterized the semantic difference between default negation and what we call non-default negation in terms of the associated mappings. This is in contrast to the practice among some authors ([GL90], for instance) of characterizing the difference between the two types of negation in terms of the difference between the treatment of sentences containing default negation and the treatment of sentences containing non-default negation in the procedure for determining the semantics of a program containing both types of negation. Rather, what we have done is first give the semantics of the two types of negation and on the basis of this, and the semantics of the other operators, given the semantics of a logic program.

The apparatus of this chapter can be extended in a straight-forward manner to develop a semantics for extended logic programs with heterogeneous contestations. The various *cap* functions will have to be redefined in terms of the truth values of **C5**. In evaluating the truth value of a rule constrained by contestations, the evaluation function $\mathcal{I}'''$ should be used. This semantics otherwise will be analogous to the semantics of normal logic programs with contestations.

We have not developed a proof procedure for extended logic programs. This will be done in future work.

11.6 Summary

In this chapter we have developed the **C5** semantics for extended logic programs, which contain both a default and a non-default negation. The specific research contributions of this chapter are summarized below.

- We have developed a five-valued semantics **C5** which is an extension of **C4** (Section 11.3).

- We have proven that every extended logic program has at least one (consistent) canonical model under **C5** (Section 11.3).

- We have shown how to capture part of the logical force of non-default negation in terms of contestations. If non-default negation is viewed as an approximation of classical negation, then logical conflict in a logic program can be represented in terms of the derivability of a literal and its non-default negation from the program. Thus, logical conflicts as well as non-logical conflicts can be represented in terms of contestations. Thus we have established that contestations provide a flexible framework for expressing and reasoning with a wide variety of conflicts among statements (Section 11.3).

- We have proven that **C5** is inferentially conflict-free with respect to the approximation of logical conflicts in terms of non-default negation (Section 11.3).

- We have proven that for any extended logic program P which has a consistent answer set, a literal l is strongly entailed by P under the answer set semantics ([GL90]) if and only if l is weakly entailed under **C5** (Section 11.4).

- We have shown how the five truth values of **C5** and orderings associated among these truth values can be derived from the truth values $\{F, U, T\}$ of Kleene ([Kle50]) and the truth and knowledge orderings among these truth values in the context of two players assigning these truth values to the same set of statements, where one player's assignment is allowed to dominate the other person's assignment without winning outright (Section 11.5).

Chapter 12

Conclusions and Future Work

In this chapter we summarize the research described in this thesis and outline the direction of the future development of the research accomplishments described in this thesis.

12.1 Summary

In this dissertation we have presented a framework for expressing different types of conflicts among statements and for reasoning with information containing these types of conflicts. The conflicts are expressed using a construct called contestations. Contestations are symbolically expressed as $A \hookrightarrow_i b$, where A is a conjunction or a disjunction of ground literals and b is a ground atom. The contestation $A \hookrightarrow_i b$ says that if A attains a certain truth value, v_1, then b can attain at most a certain other truth value, v_2, which depends on the truth value v_1 and the truth function cap_i on which $\hookrightarrow_i$ is based. Different types of contestations can be based on different *cap* functions (Chapter 4).

We have provided a semantics, **C4**, for normal logic programs augmented with a set of contestations. These contestations can be heterogeneous in the sense that they are based on different *cap* functions. **C4** is based on a set of truth values, $\{F,\ CF,\ CT,\ T\}$, and the ordering $F < CF < CT < T$ among these truth values; the idea of a well-supported model, which is intended to capture the idea of an evidentially reasonable model; a clausal ordering among the well-supported models; and the idea of canonical models as the models that are maximal in this ordering among the well-supported models. In terms of this semantics, we define a strong entailment relation and a weak entailment relation. The strong entailments of $P+\mathcal{C}$, where P is a normal logic program and $\mathcal{C}$ is a set of contestations, are the literals which are T in all the canonical models of $P+\mathcal{C}$. The weak entailments of $P+\mathcal{C}$ are those literals which are at *at least* CT in all the canonical models of $P+\mathcal{C}$. We have shown that the **C4** semantics is inferentially conflict-free with respect to the types of conflicts that can be represented in terms of contestations (Chapter 4).

We have shown that for any normal logic program without contestations, the **C4** semantics provides at least one well-supported model. Although it is a highly desirable that for any normal logic program, P, and any set of contestations, $\mathcal{C}$, **C4** provides at least one well-supported model for $P+\mathcal{C}$, this claim was shown to be false. However, we show that **C4** provides at least one well-supported model for any normal logic program augmented with a set of contestations based on any truth function cap_i such that $cap_i(CF) = CT$ (Chapter 4).

In Chapter 5 we have investigated the properties of **C4** as a semantics of normal logic programs (without contestations). We have proven that every definite logic program has a unique **C4** canonical model and we have proven that every normal logic program has at least one **C4** canonical model (Section 5.2). We have proven that the **C4** semantics of normal logic programs subsumes both the stable model semantics and the well-founded semantics. **C4** can provide models for programs for which there are no stable models. Furthermore, for any program which has a stable model, a literal l is true in all the stable models of the program if and only if l is weakly entailed by the program under **C4** (Section 5.3). We have also proven that l is true in the well-founded semantics of a normal logic program if and only if l is strongly entailed by the program under **C4** (Section 5.4). We also provide a framework for hybrid reasoning in which part of a query is to be answered under weak entailment and the remaining under strong entailment. Since strong entailment is more cautious that weak entailment this provides a way of being cautious regarding part of a query and non-cautious regarding the rest of the query (Section 5.5).

In Chapter 6 we have developed a bottom-up assumption based proof procedure for answering whether a ground query, consisting of a conjunction or a disjunction of literals, is *weakly* entailed by a finite and ground normal logic program. This proof procedure is restricted to programs having C-stable canonical models, and for programs without any C-stable canonical models the proof procedure terminates gracefully by informing the user about this (Section 6.3). We have proven that this proof procedure is sound and complete with respect to the **C4** semantics for normal logic programs (Section 6.4). We have computed the worst case complexity of the weak entailment proof procedure to be $O(n^2 \times 2^n)$, where n is the cardinality of the Herbrand base of the program (Section 6.6).

We have also developed a bottom-up assumption based proof procedure for answering whether a ground query, consisting of a conjunction or a disjunction of literals, is *strongly* entailed by a finite and ground normal logic program (Section 7.3). This proof procedure works for all finite and ground normal logic programs. We have proven that this procedures is sound and complete with respect to the **C4** semantics for normal logic programs (Section 7.4). We have computed the worst case complexity of the strong entailment proof procedure to be $O(n^3)$, where n is the cardinality of the Herbrand base of the input program (Section 7.5).

In Chapter 8 we have extended the proof procedures of Chapter 6 and Chapter 7 to a proof procedure for answering whether a ground query is weakly or strongly entailed by a finite and ground normal logic program augmented with a heterogeneous set of ground contestations. These proof procedures are also restricted to programs with a C-stable canonical models (Section 8.3). We have proven that these proof procedures are sound and complete with respect to the **C4** semantics for normal logic programs augmented with contestations (Section 8.4). We have computed the complexity of both the weak and strong entailment proof procedures to be $O(n^2 \times 2^n)$ (Section 8.5).

In Chapter 9 we have shown **C4** can be used to reason with a deductive database that is inconsistent with its own integrity constraints. We have shown how to capture a wide variety of propositional integrity constraints for a deductive database in terms of contestations. This enables us to use **C4** as a semantics for deductive databases that are inconsistent with their own integrity constraints. This account is restricted to propositional integrity constraints since in this dissertation we have only considered propositonal contestations. We have also introduced a new theory of integrity constraint satisfaction according to which a database satisfies its integrity constraints only if the integrity constraints are true in what can be inferred from it in. Thus, integrity constraints on a database are best viewed as constraints on what can be inferred from the database rather than on the state of the database.

In Chapter 10 we have shown how to extend the **C4** semantics for a normal logic program, LP, augmented with contestations, $\mathcal{C}$, by adding a set of preferences, $\mathcal{P}$, among statements. The first semantics is based on using the preferences of $\mathcal{P}$ to induce an ordering among the well-supported models of $LP + \mathcal{C}$. The second semantics is based on the idea of a well-supported model of $LP + \mathcal{C}$ satisfying the preferences of $\mathcal{P}$. Although these two semantics are based on different ways of factoring in the role of preferences, we proved that these two semantics are equivalent.

In Chapter 11 we have extended **C4** to **C5**, which is based on an additional truth value U and three types of ordering between the five truth values of **C5**. We used **C5** to provide a semantics for extended logic programs, which contain both a default and a non-default negation. We proved that every extended logic program has a well-supported model. We prove that the **C5** semantics for extended logic programs subsumes the answer set semantics for extended logic programs in the sense that for any extended logic program which has a consistent answer set semantics, a sentence is true in all the answer sets of a program if and only if it is weakly entailed by the program under **C5**. We have shown that to the extent that non-default negation approximates classical negation, to that extent **C5** can be viewed as a framework for reasoning with logical conflicts. Thus, **C5** provides a framework for reasoning with logical as well as non-logical conflicts. We have shown that the **C5** for extended logic programs is inferentially conflict-free in the sense that no extended logic program entails, weakly or strongly, p and $\neg p$ for any atom p.

12.2 Future Research

The **C4** semantics for normal logic programs augmented with a set of contestations is restricted to ground contestations. It would be useful to extend this account to non-ground contestations. This would allow us to represent non-propositional integrity constraints in terms of contestations. Thus, our method of reasoning with deductive databases that are inconsistent with their own integrity constraints can be extended to a wider set of integrity constraints.

The prooof procedures described in this work are all restricted to finite and ground programs. These proof procedures need to be extended to non-ground programs. Furthermore, all the proof procedures except for the strong entailment proof procedure for normal logic programs without any contestations are restricted to programs (possibly augmented with contesations) having C-stable canonical models. It would be desirable to extend these procedures to programs having no canonical C-stable models.

This work is restricted to normal logic programs. It would be useful to extend this work to a wider class of programs, such as disjunctive logic programs.

BIBLIOGRAPHY

[ABC99] M. Arenas, L. Bertossi, and J. Chomicki. Consistent query answers in inconsistent databases. In *Proceedings of the 18th Symposium on Principles of Database Systems*, pages 68–79, 1999.

[ABW88] K.R. Apt, H.A. 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 Pub., Washington, D.C., 1988.

[ADP93] J.J. Alferes, P.M. Dung, and L.M. Pereira. Scenario semantics of extended logic programs. In L.M. Pereira and A. Nerode, editors, *Proc. 2nd International Conference on Logic Programming and Nonmontonic Reasoning*, pages 334–348, 1993.

[AKWS95] S. Agarwal, A.M. Keller, G. Wiederhold, and K. Saraswat. Flexible relations: An approach for integrating data from multiple, possibly inconsistent databases. In *IEEE International Conference on Data Engineering*, pages 495–504, 1995.

[AP92a] J.J. Alferes and L.M. Pereira. On logic program semantics with two kinds of negation. In K. Apt, editor, *Proceedings of the Joint In-*

ternational Conference and Symposium on Logic Programming, pages 574–588, 1992.

[AP92b] J.J. Alferes and L.M. Pereira. Well founded semantics for logic programs with explicit negation. In B. Neumann, editor, *10th ECAI*, 1992.

[Arr79] A.I. Arruda. A survey of paraconsistent logic. In A.I. Arruda, R. Chuaqui, and N.C.A. da Costa, editors, *Mathematical Logic in Latin America*, pages 1–41. D. Reidel, 1979.

[BDK97] G. Brewka, J. Dix, and K. Konoloige, editors. *Nonmonotonic Reasoning: An Overview*. CSLI Lecture Notes, 1997.

[Bel77a] N. Belnap. How a computer should think. In G. Ryle, editor, *Contemporary Aspects of Philosophy*. Oriel Press, 1977.

[Bel77b] N. Belnap. A useful four-valued logic. In G.Epstein and J.M.Dunn, editors, *Modern Uses of Multiple-Valued Logic*. D. Reidel, 1977.

[BGMS92] Y. Breitbart, H. Garcia-Molina, and A. Silberschatz. Overview of multidatabase transaction management. *VLDB Journal*, 1(2), 1992.

[BKM91] C. Baral, S. Kraus, and J. Minker. Combining multiple knowledge bases. *IEEE Transactions on Data and Knowledge Engineering*, 3:208–220, 1991.

[BKMS92] C. Baral, S. Kraus, J. Minker, and V.S. Subrahmanian. Combining knowledge bases consisting of first order theories. *Computational Intelligence*, 8:45–71, 1992.

[BS89] H. Blair and V.S. Subrahmanian. Paraconsistent logic programming. *Theoretical Computer Science*, 68:135–154, 1989.

[CGM90] U.S. Chakravarty, J. Grant, and J. Minker. Logic-based approach to semantic query optimization. *ACM Transactions on Database Systems*, 15(2):162–207, 1990.

[CKRP73] A. Colmerauer, H. Kanoui, P. Roussel, and R. Pasero. *Un System de Communication Homme-Machine en Francais*. Groupe de Recherche en Inetelligence Artificielle, Universite d'Aix-Marseille, 1973.

[Cos74] N. Costa. On the theory of inconsistent formal systems. *Notre Dame Journal of Formal Logic*, 15:497–510, 1974.

[CW97] W. Chen and D. S. Warren. Computation of stable models and its integration with logical query processing. *IEEE Transactions on Knowledge and Data Engineering*, 1997.

[Dix95] J. Dix. A classification theory of semantics of normal logic programs: I. strong properties. *Fundamenta Informaticae*, 22(3):227–255, 1995.

[DKT96] P.M. Dung, R.A. Kowalski, and F. Toni. Argumentation-theoretic proof procedure for non-monotonic reasoning. In J. Gallagher, editor, *Proceedings of the 6th LOgic Programming Synthesis and TRansformation*, 1996.

[Doy80] J. Doyle. Truth Maintenance System. *Artificial Intelligence*, 13, 1980.

[DR91] P.M. Dung and R. Ruamviboonsuk. Well founded reasoning with classical negation. In A. Nerode, W. Marek, and V.S. Subrahmanian, editors, *Proceedings of the First International Conference on Logic Programming and Nonmonotonic Reasoning*, pages 120–132. MIT Press, 1991.

[Dun93] P.M. Dung. The acceptability of arguments and its fundamental role in non-monotonic reasoning and logic programming. In R. Bajcsy, editor, *Proceedings of the 13th International Joint Conference on Artificial Intelligence*, pages 852–857. Morgan Kaufmann, 1993.

[Dun96] P.M. Dung. Integrating data from possibly inconsistent databases. In *Proceedings of International Conference on Cooperative Information Systems*, 1996.

[Fag91] F. Fages. A new fixpoint semantics for general logic programs compared with the well-founded and stable model semantics. *New Generation Computing*, 9:425–443, 1991.

[FH85] R. Fagin and J.Y. Halpern. Belief, awareness and limited reasoning. *Proc. 9th International Conference on Artificial Intelligence*, pages 491–501, 1985.

[Fit85] M. Fitting. A kripke-kleene semantics of logic programs. *Journal of Logic Programming*, 2(4):295–312, 1985.

[FUV83] R. Fagin, J.D. Ullman, and M.Y. Vardi. On the semantics of updates in databases. In *Proc. 7th ACM SIGACT/SIGMOD Symposium on Principles of Database Systems*, pages 352–365, 1983.

[GL88] M. Gelfond and V. Lifschitz. The Stable Model Semantics for Logic Programming. In R.A. Kowalski and K.A. Bowen, editors, *Proc.* 5^{th} *In-*

ternational Conference and Symposium on Logic Programming, pages 1070–1080, Seattle, Washington, August 15-19 1988.

[GL90] M. Gelfond and V. Lifschitz. Logic programs with classical negation. In D. Warren and P. Szerdi, editors, *Proc. 7^{th} International Conference and Symposium on Logic Programming*, pages 579–597, 1990.

[GR95] P. Gaerdenfors and H. Rott. Belief revision. In D.M. Gabbay, J. Hogger, and J.A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, pages 35–132. Oxford University Press, 1995.

[Gra74] J. Grant. Incomplete models. *Notre Dame J. of Formal Logic*, XV(4):601–607, 1974.

[Gra75] J. Grant. Inconsistent and incomplete logics. *Mathematics Magazine*, 48(3):154–159, 1975.

[Gra78] J. Grant. Classifications for inconsistent theories. *Notre Dame J. of Formal Logic*, XIX(3):435–444, 1978.

[GRS88] A. Van Gelder, K. Ross, and J.S. Schlipf. Unfounded sets and well-founded semantics for General Logic Programs. In *Proc. 7^{th} Symposium on Principles of Database Systems*, pages 221–230, 1988.

[GRS91] A. Van Gelder, K. Ross, and J.S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650, 1991.

[KL92] M. Kifer and E.L. Lozinskii. A logic for reasoning with inconsistency. *Journal of Automated Reasoning*, 9(2):179–215, 1992.

[Kle50] S.C. Kleene. *Introduction to Metamathematics.* D. Van Nostrand, 1950.

[Kow74] Robert Kowalski. Predicate logic as a programming language. *Information Processing*, (74):569–574, 1974.

[Kow78] Robert Kowalski. Logic for data description. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 77–102. Plenum Press, New York, 1978.

[KS90] Robert A. Kowalski and Fariba Sadri. Logic programs with exceptions. In Warren and Szeredi, editors, *Proceedings of the* 7^{th} *International Logic Programming Conference*, pages 598–613, Jerusalem, 1990. MIT Press.

[Lin96] J. Lin. A semantics for reasoning correctly in the presence of inconsistency. *Artificial Intelligence*, 86(1-2):75–95, 1996.

[Llo87] J. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, second edition edition, 1987.

[Luk20] J. Lukasiewicz. On three-valued logic. In L. Borkowski, editor, *Selected Works*, pages 87–88. North-Holland, Amsterdam, 1920.

[MM93] M.Cadoli and M.Schaerf. A survey of complexity results for non-monotonic logics. *Journal of Logic Programming*, 17:127–160, 1993.

[MT91] V. Marek and M. Truszczynski. Computing intersection of autoepistemic expansions. *Proceedings of the First International Workshop on Logic Programming and Non Monotonic reasoning*, pages 37–50, 1991.

[PM96] S. Pradhan and J. Minker. Using priorities to combine knowledge bases. *International Journal of Intelligent and Cooperative Information Systems*, 5(2-3):333–364, 1996.

[PMS95] S. Pradhan, J. Minker, and V.S. Subrahmanian. Combining databases using priorities. *Journal of Intelligent Information Systems*, 4:231–260, 1995.

[Prz90a] T. Przymusinski. Extended stable semantics for normal and disjunctive programs. In *Proc. ICLP'90*, pages 459–477, 1990.

[Prz90b] T. Przymusinski. Well-founded semantics coincides with three-valued stable semantics. *Fundamenta Informaticae*, 13:445–463, 1990.

[Rei84] R. Reiter. Towards A Logical Reconstruction of Relational Database Theory. In M.L. Brodie, J.L. Mylopoulos, and J.W. Schmit, editors, *On Conceptual Modelling*, pages 163–189. Springer–Verlag Pub., New York, 1984.

[Rya91] M. Ryan. Belief revision and ordered theory presentation. In P. Dekker and M. Stokhof, editors, *Proc. Eighth Amsterdam Colloquium on Logic*, 1991.

[Rya92a] M. Ryan. *Ordered Presentation of Theories: Default Reasoning and Belief Revision*. PhD thesis, Department of Computing, Imperial College, 1992. Ph.D. thesis.

[Rya92b] M. Ryan. Representing defaults as sentences with reduced priority. In B. Nebel and W. Swartout, editors, *Proc. KR '92*. Morgan Kaufmann, 1992.

[Sak92] C. Sakama. Extended well-founded semantics for paraconsistent logic programs. In *Proc. of the International Conference on Fifth Generation Computer Systems, ICOT*, pages 592–599, 1992.

[Sub94] V.S. Subrahmanian. Amalgamating knowledge bases. *ACM Trans. on Database Systems*, 19:291–331, 1994.

[Tar55] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.

[TK81] A. Tversky and D. Kahneman. The framing of decisions and the psychology of choice. *Science*, 211:453–458, 1981.

[vEK76] M.H. van Emden and R.A. Kowalski. The Semantics of Predicate Logic as a Programming Language. *J.ACM*, 23(4):733–742, 1976.