Logic Programming and Non-Monotonic Reasoning

Frank J.M. Teusink

Logic Programming and Non-Monotonic Reasoning

Logic Programming and Non-Monotonic Reasoning

Academisch Proefschrift

ter verkrijging van de graad van doctor
aan de Universiteit van Amsterdam,
op gezag van de Rector Magnificus
prof.dr. P.W.M. de Meijer
ten overstaan van een door
het college van dekanen ingestelde commissie
in het openbaar te verdedigen in de
Aula der Universiteit
(Oude Lutherse Kerk, ingang Singel 411, hoek Spui)
op maandag 30 september 1996 te 12.00 uur

door

Franciscus Johannes Maria Teusink

geboren te Hengelo (Ov).

Promotor: Prof.dr. K.R. Apt
Faculteit Wiskunde en Informatica
Universiteit van Amsterdam
Plantage Muidergracht 24
1018 TV Amsterdam

These investigations were supported by the Stichting Informatica Onderzoek Nederland (SION), which is subsidized by the Netherlands Organization for Scientific Research (NWO).

Cover design by Frank Teusink.
Printed and bound by CWI.

Contents

Acknowledgments

This thesis, and the good time I had working on it, would not have existed in this form, if it weren't for ...

a lot of people.

First of all, if it weren't for Krzysztof Apt. Who accepted me as a PhD student after a phone call and one interview. Who made sure I started off in the right direction, and then gave me the freedom to find my own way. For Linda van der Gaag, who was my mentor while writing my masters thesis at the University of Utrecht and got me involved in computer science. And for Cees Witteveen, who was like a co-mentor in that period, and gave me the hint to apply for a PhD position with Krzysztof.

If it weren't for Elena Marchiori, my roommate at CWI, who was always in for a chat and a cup of coffee, and with whom it was a joy working together. If it weren't for friends like Annius, Wilco, and Andrea (Bracciali) with whom I could complain about the ordeals of writing a PhD thesis. If it weren't for all my colleagues at CWI. Especially, the people I lunched or drank coffee with regularly: Andrea (Schaerf), Bart, Femke, Gilles, Herbert, Inge, Jan (van Eyck), Jan (Rutten), Jan Willem, Maurizio, Sandro, Wilfried, Zena, and all the others. For the CWI itself, which is (still) a heaven for scientists. And for the (rest of) the 'Personeels Vereniging' organizing committee 1995, Annius, Annette, Frans, Hans, Jacques and Walter, who organized social events for the CWI.

If it weren't for the members of the Compulog II project, with whom I enjoyed stimulating discussions during their meetings in various beautiful places in Europe. And for the reading committee, Johan van Benthem, Roland Bol, Kees Doets, Peter van Emde-Boas, Elena Marchiori and John-Jules Meyer, for their willingness to review this thesis and for their many useful comments.

Finally, if it weren't for my family, who made sure my feet stayed on the ground, and always patiently bore with me whenever I insisted I was not one of those eternal students, but rather some lucky guy who got payed for doing whatever he liked, whenever he liked.

Amsterdam
July 1996.

Frank Teusink

Chapter 1

Introduction

1.1 Logic Programming

Logic programming is based on the idea that first-order predicate logic can be used as a programming language. This was first expressed in a paper by R.A. Kowalski [Kow74]. The idea is that logic is well-suited to express knowledge and therefore a programming language based on logic is well-suited in domains where most of the programming consists of writing down the knowledge about that domain. As such, a logic program P is seen as a theory expressed in some logical language, and execution of a logic program is the process of verifying whether a query, in the form of a formula ϕ, is provable in that theory; that is, whether $P \vdash \phi$. The algorithm that checks whether $P \vdash \phi$, is called a *proof procedure*. The essential difference between logic programming and (automated) theorem proving is, that with logic programming the proof procedure is supposed to be fully automatic and efficient, even if this implies that not always a proof is found, whereas theorem proving is more concerned with finding the proofs, even if this implies that the proof procedure gains complexity.

In order to be able to use efficient proof procedures, the language in which the logic program is written, and the logic which is used, are tailored. In the original paper by Kowalski [Kow74], a *logic program* consists of a number of rules of the form $A_0 \leftarrow A_1, \ldots, A_n.$, where $n \geq 0$ and, for $i \in [0..n]$, A_i is an atom. Such a rule should be read as "if A_1 and ... and A_n are all true, then A_0 is also true". Programs of this form are called *definite* logic programs. A *definite* query is a formula of the form $B_1, \ldots, B_n.$, where the B_i are atoms. It should be read as "(check whether) B_1 and ... and B_n are all true". For this class of logic programs and queries, there exists an effective proof procedure, called *SLD-resolution*.

Definite logic programs can be used for a wide range of problem domains. However, one thing it cannot deal with, is *negation*; the logical operator $\neg$

simply is not part of the language. Yet, when writing programs, negation is quite useful. The simplest example of this is probably the (railway) time table. Consider a time table represented as a logic program, via a number of statements of the form *train(from, to)*., for instance

$$\begin{array}{l} train(Amsterdam, Utrecht). \\ train(Utrecht, Amersfoort). \\ train(Amersfoort, Apeldoorn). \\ train(Amsterdam, Amersfoort). \end{array}$$

(note that we omit the $\leftarrow$ in clauses where $n = 0$).

In definite logic programming, the only way to express the fact that there is no railway connection between two places, is to add statements of the form *no_train(from, to)*., for *all* pairs of places between which there is no railway connection. This is laborious and undesirable. People who read a time table do not need this; they assume that, when a connection is not listed on the table, the connection does not exist. Translated to logic programming, one desires a logical operator $\neg$, such that $\neg train(place_1, place_2)$ holds whenever one cannot conclude from the program that $train(place_1, place_2)$ holds.

1.1.1 Normal Logic Programming

The above observation led to the definition of *normal* logic programs. A normal logic program consists of a number of rules of the form $A_0 \leftarrow L_1, \ldots, L_n.$, where $n \geq 0$, A_0 is an atom and, for $i \in [1..n]$, L_i is a literal, i.e. an atom A_i or a negated atom $\neg A_i$. A *normal* query is a formula of the form $L_1, \ldots, L_n$.

Introducing the negation operator $\neg$ in the language is not enough, however. There should be agreement on how to interpret this operator. It cannot be treated and used as classical negation. The reason is, that the conclusions of rules (the atoms left of the $\leftarrow$) all state positive facts. Therefore, one can only infer from these rules whether something is true according to the program. To obtain negative information from a logic program, one has to use the kind of negation that is used by the reader of a time table. In the case of normal logic programs, the negation that is used is *negation as (finite) failure*. This notion of negation can be stated as follows:

$\neg A$ succeeds if A fails (finitely)
$\neg A$ fails if A succeeds

This means that, to decide truth of $\neg A$, we first decide truth of A. If we conclude that A is true, we conclude that $\neg A$ is false, and vice versa.

1.2 Non-Monotonic Reasoning

In the field of artificial intelligence, those who try to model the reasoning behaviour of humans using a logical framework, have found that pure classical logic does not serve their need. The reason for this is that classical logic is *monotonic*. That is, if a formula ϕ is a logical consequence of a theory T then, given an arbitrary theory T', ϕ is also a logical consequence of the theory $T \cup T'$. In contrast, the reasoning employed by humans tends to be non-monotonic; one tends to reach conclusions from certain premises, which one would not reach in the presence of certain additional premises. As an example, when Mary tells John she sees a bird, John will certainly assume that this bird can fly, and also watch the sky in order to spot this bird. However, when Mary then continues to add that the bird is an ostrich, he will without any problem retract the assumption that the bird can fly, and start watching only the surrounding landscape.

With the observation that people apply some form of non-monotonic reasoning, people have developed various logical frameworks for this kind of reasoning. Among these are R. Reiters closed world assumption [Rei78] and default theory [Rei80], McCarthy's circumscription [McC80, McC86] and Moore's autoepistemic logic [Moo85]. In the remainder of this section, we give a short overview of some of these formalisms.

One of the first investigations on non-monotonic forms of negation was negation based on the *Closed World Assumption (CWA)* , proposed by R. Reiter in [Rei78]. This kind of negation can be formulated as the following meta rule:

$$\text{CWA : if } A \text{ cannot be proved, then infer } \neg A.$$

Note that this is the kind of reasoning humans apply when reading a time table. Also, the formulation is roughly similar to the formulation of negation as failure. Adding CWA introduces non-monotonicity. With CWA, we may infer $\neg p$ from a theory if we cannot prove p in that theory. However, if we extend that theory to an extended theory in which p *can* be proven, we can no longer use CWA to infer $\neg p$ from that extended theory.

In [Rei80], R. Reiter proposes default logic. It is based upon classical logic. In it, it is possible to have, besides a set of ordinary closed first-order sentences T, an additional set D of rules of inference, in the form of *default rules*:

$$\frac{\alpha : M\beta_1, \ldots, M\beta_n}{\gamma}$$

where α, γ and the β_i are first order formulae. Such a rule should be read as follows: if α holds, and each of the the β_i are possible (i.e. can be consistently assumed), then γ can be inferred. The theory T describes what is true, while D enables one to derive additional information, by reasoning *by default*. The semantics of such a default theory $\langle D, T\rangle$ is given by the notion of *extension*. In short, an extension Σ of $\langle D, T\rangle$ is a smallest set of first-order sentences that satisfies the following criteria:

1. T is contained in Σ,

2. Σ is closed under logical consequences,

3. if R is a rule in D such that α is in Σ and for every i, $\neg\beta_i$ is not in Σ, then γ is in Σ, and

4. Σ is an extension of $\langle D, \Sigma\rangle$.

A default theory can have no extension, a single extension, or multiple extensions. Each extension is a set of possible believes held by an agent. As an example, let us formalize the bird example in default logic: the (in)famous *Tweety* example. Let D consist of the single rule

$$\frac{\mathit{bird}(\mathit{tweety}) : M\neg \mathit{abnormal}(\mathit{tweety})}{\mathit{flies}(\mathit{tweety})}$$

and consider the following theories:

$$T_1 : \{\mathit{bird}(\mathit{tweety})\}$$
$$T_2 : \{\mathit{bird}(\mathit{tweety}), \mathit{abnormal}(\mathit{tweety})\}$$

Then, $\langle D, T_1\rangle$ has a single extension; one in which $\mathit{flies}(\mathit{tweety})$ holds. On the other hand, in the only extension for $\langle D, T_2\rangle$, $\mathit{flies}(\mathit{tweety})$ does not hold.

1.3 Non-Monotonicity via Logic Programming

The kind of negation used in normal logic programs, negation as finite failure, also introduces non-monotonicity. Consider for instance the program consisting of the single clause $p \leftarrow \neg q$. Because there are no clauses to prove q, using negation as finite failure we conclude that $\neg q$ is true. From this we conclude that p is true. However, if we add to the program the clause q. then q becomes true, therefore $\neg q$ becomes false, and therefore p becomes false. This observation has led to the idea of implementing various forms of non-monotonic reasoning using logic programming and negation as failure.

Also, the formulation of negation as failure is procedural. Yet one of the main reasons for the development of logic programming is the idea that a program should have, aside from an operational semantics, a declarative semantics.

1.4 Forms of Non-Monotonicity

Within the realm of artificial intelligence and knowledge representation, non-monotonic reasoning appears to be more of a standard than an exception. Let us now discuss some forms of non-monotonicity.

First, let us take a look at the example we saw already in the previous section, in the form of *default reasoning*. Default reasoning stems from the observation that reality is often not as well-structured as a programmer or knowledge engineer would like. For instance, nobody will object to the statement "birds can fly". However, when looking closer at the population of birds, one discovers that there exist (species of) birds that cannot fly, like penguins and ostriches. Thus, the statement "birds can fly" should be refined to something like, "all birds can fly, except for penguins and ostriches". This can be formulated in a normal logic program.

$$\begin{aligned}&\mathit{flies}(x) \leftarrow \mathit{bird}(x), \neg \mathit{abnormal}(x).\\&\mathit{abnormal}(x) \leftarrow \mathit{penguin}(x).\\&\mathit{bird}(x) \leftarrow \mathit{penguin}(x).\\&\mathit{bird}(x) \leftarrow \mathit{eagle}(x).\\&\mathit{penguin}(\mathit{tweety}).\\&\mathit{eagle}(\mathit{sam}).\end{aligned}$$

In this program, $\mathit{flies}(\mathit{tweety})$ does not hold, because *tweety* is a penguin, penguins are abnormal, and only normal birds fly. Also, $\mathit{flies}(\mathit{sam})$ holds, following the reasoning that, since $\mathit{abnormal}(\mathit{sam})$ does not hold, it should be assumed that $\neg\mathit{abnormal}(\mathit{sam})$ holds. Note that this form of reasoning is inherently non-monotonic; if one adds the clause $\mathit{abnormal}(\mathit{sam})$., $\mathit{flies}(\mathit{sam})$ no longer holds.

Another field in which non-monotonicity pops up naturally, is that of temporal logics. Suppose we want to reason with time, work with sentences like "at time t I switched the light off", and ask questions like "was the light on at time $t+4$?". This can be formalized in logic programming, using the predicates $\mathit{switch\_on}(t)$ and $\mathit{switch\_off}(t)$, which switch the light on or off at time t, and the predicate $\mathit{light\_on}(t)$ which is true only if the light is on at time t. Now, how do we ensure that, once the light is switched on, it remains on, until it is explicitly switched off again? This is called the temporal persistence problem, and it can be solved using non-monotonicity. In our case, using the following clauses:

$$\begin{aligned}&\mathit{light\_on}(t+1) \leftarrow \mathit{switch\_on}(t).\\&\mathit{light\_on}(t+1) \leftarrow \mathit{light\_on}(t), \neg\mathit{switch\_off}(t).\\&\mathit{switch\_on}(4).\end{aligned}$$

The first clause simply states that the light goes on when it is switched on. The second clause ensures temporal persistence, by stating that the light remains on, as long as it is not switched off. Again, the reasoning applied is inherently non-monotonic. For all t where $t > 4$ we have that $\mathit{light\_on}(t)$ holds. But if we add the clause $\mathit{switch\_off}(6)$., for all $t > 6$ $\mathit{light\_on}(t)$ no longer holds.

1.5 Semantics for Normal Logic Programs

The use of negation in logic programming enables us to write shorter and (hopefully) more readable programs. However, it has proven difficult to find a satisfactory way of interpreting negation in normal logic programs: although negation as failure provides a good idea of how negation can be implemented, it does not provide us directly with a (model theoretic) semantics.

One of the first ideas of providing a semantics was to add the Closed World Assumption to normal logic programs. However, such a semantics cannot explain certain programs, like programs consisting of the single clause $p \leftarrow \neg p$. Thus, people have started the quest for the 'right' semantics for normal logic programs, i.e. one that is best at representing the knowledge contained in such a program. This has resulted in a whole range of semantics for normal logic programs, from simple ones, for restricted classes of programs, to more complicated (three-valued) semantics that do not pose any restrictions on the programs they interpret. Aside from the problem of finding the 'right' semantics for normal logic programs, there is another problem, which is in some sense orthogonal to this one. That is, finding a semantics for which there exists an effective or even efficient proof procedure.

Together, answers to these two problems have created a whole landscape of semantics for normal logic programs. There does not exist one 'best' semantics. Instead, different semantics serve different purposes. One distinction which can be made, is that between using normal logic programs for knowledge representation and using them for programming. In the former, people are most interested in semantics which are best at extracting as much knowledge from the program as possible. Effectiveness and efficiency of proof procedures for the semantics used are of less concern. In the latter, the foremost concern is that of finding a semantics for which effective and efficient proof procedures exist, or even that of finding a semantics that is best at describing the answers computed by a given proof procedure.

The landscape of semantics grows even wider if one extends ones view beyond the realm of normal logic programs. Semantics have been developed (among others) for definite and normal *disjunctive* logic programs, where the head of a rule consists of a disjunction of atoms or literals, for *extended* logic programs, which is an extension of normal logic programs in which two distinct kinds of negation are used, and for *abductive* or *open* logic programs, which are based on abduction instead of deduction.

1.6 Synopsis

In this thesis, we investigate various semantics for normal logic programs, and their usefulness for non-monotonic reasoning. The chapters are presented in the chronological order of the papers they originate from (see Section 1.7). In general, there is a trend from more knowledge representation oriented semantics in the first chapters, to a more programming oriented approach on semantics in the final chapters.

To begin with, in Chapter 2, we take a look at the stable model semantics. We find a constructive (though transfinite) characterization of the stable models of a normal program. In the division between semantics for knowledge representation and for programming, stable models are definitely a semantics for knowledge representation. The reason is that there do not exist effective proof procedures for the stable model semantics. Moreover, even for function free programs, the complexity of finding these models is EXPTIME-hard.

When representing the knowledge about some problem domain in a logic program, the negation used in normal logic programs ($\neg$) is not always exactly what you want. A formula $\neg\phi$ is true when ϕ cannot be proven. In some situations, one wants to use a stronger form of negation ($\sim$) such that $\sim \phi$ is true only if there is an explicit proof for $\sim \phi$, and use it in programs where the head of a clause can contain (strongly) negated atoms. In Chapter 3, we consider such extended logic programs. By using $\sim$ in heads of clauses, a program can become *inconsistent*. This is a problem, because an inconsistent program does not give any useful information. Moreover, checking whether a program is consistent is expensive, and not always possible. In this chapter, we propose to take a *paraconsistent* approach to extended logic programs, to avoid these problems. That is, we extend our reasoning mechanism with rules that (try to) repair any inconsistencies that might appear in a program.

In (S)LDNF resolution, negation is treated as "negation as finite failure". In contrast, negation in Prolog is generally defined using meta-programming facilities and the cut operator. In Chapter 4, based on a paper written together with Krzysztof Apt, we investigate the precise behaviour of this negation in Prolog, and establish a formal result showing an equivalence in appropriate sense between these two uses of negation.

Another 'extension' of logic programming is abductive logic programming. Here, the language contains so-called *abducibles*. The idea is, that a query is now an observation in the real world, the program explains observables in terms of abducibles, and an answer to a query is an explanation of the observation in terms of abducibles. For instance, given a rule

$$shoes\_are\_wet \leftarrow it\_is\_raining$$

the observation *shoes_are_wet* is explained by *it_is_raining*. In Chapter 5 we show that, although the reasoning employed with abductive logic programs is based on abduction instead of deduction, the proof procedures and semantics for logic programs can be extended to deal with limited forms of abduction.

One problem with normal logic programs is, that they are not compositional. In Chapter 6, based on a paper written together with Sandro Etalle, we present a notion of modular logic programs, and a semantics for modular logic programs which is compositional, while maintaining some degree of non-monotonicity. This semantics is based on the unfolding of program definitions.

1.7 The Origins of Chapters

Chapter 2 is based on the article "A Characterization of Stable Models using a Non-Monotonic Operator", which appeared in the Journal of Methods of Logic in Computer Science, Volume 1, Number 4, 1994. A short version of this article was presented at the Second International Workshop on Logic Programming and Non-Monotonic Reasoning (1993) in Lissabon, Portugal.

Chapter 3 is based on the paper "A Proof Procedure for Extended Logic Programs", which was presented at the 1993 International Logic Programming Symposium in Vancouver, Canada.

Chapter 4 is based on Chapter 5 of the book Meta-Logics and Logic Programming, edited by Krzysztof Apt and Franco Turini, which was published in 1994 by the MIT Press. This chapter was written together with Krzysztof Apt.

Chapter 5 is based on the paper "Three-Valued Completion for Abductive Logic Programs", which is scheduled for publication in the Journal of Theoretical Computer Science. A short version of this article was presented at the 1994 International Conference on Algebraic and Logic Programming in Madrid, Spain.

Chapter 6 is based on the paper "A Compositional Semantics for Modular Logic Programs", which was written together with Sandro Etalle and has been presented at the 1996 Joint International Conference and Symposium on Logic Programming in Bonn, Germany.

Chapter 2

A Characterization of Stable Models using a Non-Monotonic Operator

Summary

Stable models seem to be a natural way to describe the beliefs of a rational agent. However, the definition of stable models itself is not constructive. It is therefore interesting to find a constructive characterization of stable models, using a fixpoint construction. The operator we define, is based on the work of -among others- F. Fages. For this operator, every total stable model of a general logic program coincides with the limit of some (infinite) sequence of interpretations generated by it. Moreover, the set of all stable models coincides with certain interpretations in these sequences. Furthermore, we characterize the least fixpoint of the Fitting operator and the well-founded model, using our operator.

2.1 Introduction

Stable models, as introduced by M. Gelfond and V. Lifschitz in [GL88] and extended to three-valued stable models by T. Przymusinski in [Prz90a], seem to be a natural candidate for providing normal logic programs with a meaning. However, their definition is not constructive. The aim of this chapter is to find a constructive characterization of (two- and three-valued) stable models for normal logic programs, using sequences of interpretations generated by iterating a non-deterministic non-monotonic operator. The non-deterministic behaviour of this operator is captured by using the notion of selection strategies. Our operator is based on the ideas of F. Fages [Fag91]. The main difference with the approach of Fages is, that our operator is less non-deterministic than his. As a result, our operator is more complex. However, it enables us to define a notion of (transfinite) fairness with which we can define a class of stabilizing strategies that characterize all total (i.e. two-valued) stable models. Moreover, the additional structure in our operator allows us to define various classes of strategies with nice properties. The difference of our operator with respect to the *backtracking fixpoint* introduced by D. Saccà and C. Zaniolo in [SZ90] is twofold: we find all (three-valued) stable models, instead of only all total stable models, and, if an inconsistency occurs, we use a non-deterministic choice over all possibilities for resolving that inconsistency, while their operator uses backtracking, which is just one particular possibility.

In the next section we give a short introduction on normal logic programs and interpretations, and introduce some notation that is used throughout the chapter. Section 2.3 contains an explanation of (three-valued) well-supported models and stable models, and a generalization of Fages' Lemma, which establishes the equivalence between a subset of the set of (three-valued) well-supported models and the set of (three-valued) stable models. In section 2.4 we introduce our operator $\mathcal{S}_P$, and prove that the sequences generated by this operator consist of well-supported interpretations. After this, we show in sections 2.5, 2.6, 2.7 and 2.8 how to find total stable models, (three-valued) stable models, the least fixpoint of the Fitting operator and the well-founded model, respectively, using our operator. In section 2.9, we take a short look at the complexity of the operator, and effective strategies for finding stable models.

2.2 Preliminaries and Notation

A *normal logic program* is a finite set of clauses $R : A \leftarrow L_1 \wedge \ldots \wedge L_k$, where A is an atom and L_i ($i \in [1..k]$) is a literal. A is called the *conclusion* of R, and $\{L_1, \ldots, L_k\}$ is called the *set of premises* of R. We write *concl*(R) and *prem*(R) to denote A and $\{L_1, \ldots, L_k\}$, respectively. For semantic purposes, a normal logic program is equivalent to the (possibly infinite) set of ground instances of its clauses. In the following, we only work with these infinite sets of ground clauses, and call them *programs*.

We use $\mathcal{B}_P$ to denote the Herbrand Base of a program P; A, A' and A_i represent typical elements of $\mathcal{B}_P$. Furthermore, $\mathcal{L}_P$ is the set of all literals of P; L, L' and L_i represent typical elements of $\mathcal{L}_P$. We use the following notations:

- for a literal L, $\neg L$ is the positive literal A, if $L \equiv \neg A$, and the negative literal $\neg A$, if $L \equiv A$, and
- for a set of literals S, we write
  - $\neg S$ to denote the set $\{\neg L \mid L \in S\}$,
  - $S^+ \overset{def}{=} \{A \mid A \in S\}$ to denote the set of all atoms that appear in positive literals of S,

- $S^- \overset{def}{=} \{A \mid \neg A \in S\}$ to denote the set of all atoms that appear in negative literals of S, and
- $S^\pm \overset{def}{=} S^+ \cup S^-$ to denote the set of all atoms that appear in literals of S.

A *two-valued* interpretation of a program P maps the elements of $\mathcal{B}_P$ on *true* or *false*. In this chapter, we use *three-valued* interpretations, in which an atom can also be mapped on *unknown*. They are defined as follows:

Definition 2.2.1 Let P be a program. An *interpretation* I of P is a set of elements from $\mathcal{L}_P$. An atom is *true* in I, if it is an element of I^+, it is *false* in I, if it is an element of I^-, and it is *unknown* in I, if it is not an element of $I^\pm$. If some atom is both *true* and *false* in I, then I is *inconsistent*. If all atoms in $\mathcal{B}_P$ are either *true* or *false* (or both) in I, then I is called *total*. □

Note, that a consistent total interpretation can be seen as a two-valued interpretation, because then no atom is both *true* and *false* and, because $I^\pm = \mathcal{B}_P$, no atom is unknown.

2.3 Well-Supported and Stable Models

In this section we recapture the notions of well-supported models and stable models. Our definition of well-supported models is an extension (to three-valued models) of the definition given in [Fag91]. Our definition of three-valued stable models follows the definition given in [Prz90a]. First, we introduce *well-supported models*, because they follow quite naturally from the intuitive idea of the meaning of a program. After this we give the definition of ***stable models***, which is quite elegant. In the remainder of this section we generalize of Fages' Lemma [Fag91], which states that the class of total stable models and the class of total well-supported models coincide, to three-valued models.

So, let's take a look at the intuitive idea of the meaning of a program. First of all, an interpretation should be consistent; it does not make sense to have atoms that are both true and false. Furthermore, one can see a clause in a program as a statement saying that the conclusion of that clause should be true if that clause is applicable.

Definition 2.3.1 Let P be a program, let I be an interpretation of P and let R be a clause in P. R is ***applicable*** in I, if $prem(R) \subseteq I$. R is ***inapplicable*** in I, if $\neg prem(R) \cap I \neq \emptyset$. We call $\neg prem(R) \cap I$ the ***blocking-set*** of R in I. □

Now, a *model* of a program P is a consistent interpretation I of P such that, for every clause in P that is applicable in I, the conclusion of that clause is true in I, and an atom is false in I only if all clauses with that atom as conclusion are inapplicable in I. Note, that we have to state explicitly that I has to be consistent, because in our definition an interpretation can be inconsistent.

In a model of P, atoms can be true, even if there is no reason for that atom being true. However, an atom should only be true, if there is some kind of "explanation" for the fact that that atom is true. This concept of "explanation" will be formalized using the notion of *support order*.

Definition 2.3.2 Let P be a program and let I be an interpretation of P. A (partial) pre-order $<$ on the elements of $\mathcal{L}_P$ is a *support order* on I, if, for all $A \in I^+$, there exists a clause R in P with conclusion A such that R is applicable in I and, for all $A' \in prem(R)^+$, $A' < A$. □

If, for some positive literal L that is true in M, we gather all literals L' such that $L' <^* L$ ($<^*$ is the transitive closure of $<$), then this set constitutes some kind of explanation for the fact that L is true in M.

Example 2.3.3 *Consider the program P_1:*

$$\begin{array}{l} p \leftarrow q \wedge r \\ q \leftarrow \\ r \leftarrow \neg s \end{array}$$

One of the models of P_1 is $\{p, q, r, \neg s\}$, and $\{q < p, r < p\}$ is a support order on this model. We can read this support order as follows: p is true because r and q are true, q is always true, r is true because s is false, and s is false because there is no reason why s should be true. ○

However, such an explanation can be rather awkward, either because it refers to the conclusion itself, or because it contains an infinite number of literals.

Example 2.3.4 *Consider the program P_2:*

$$\begin{array}{l} p \leftarrow q \\ q \leftarrow p \end{array}$$

One of the models of P_2 is $\{p, q\}$, and $\{p < q, q < p\}$ is a support order on this model. However, the explanation 'p is true because q is true and q is true because p is true', is not a reasonable explanation for the fact that p is true. ○

Example 2.3.5 *Consider the program P_3:*

$$p(x) \leftarrow p(s(x))$$

One of the models of P_3 is $\{p(s^i(0)) \mid i \geq 0\}$. For this model, the partial order $\{p(s^{i+1}(0)) < p(s^i(0)) \mid i \geq 0\}$ is a support order. However, any explanation for the fact that $p(0)$ is true in M_4, would be infinite. This seems to be rather counterintuitive. ○

Models for which every support order contains these cyclic or infinite explanations, should not be considered as giving a correct meaning to a program. This can be achieved by using the fact that a support order is well-founded if and only if it does not contain cyclic or infinite explanations. Now, we can give the definition of *well-supported models*.

Definition 2.3.6 Let P be a program, and let M be a model of P. M is a *well-supported model* of P, if there exists a support order on M which is a well-founded order. □

Example 2.3.7 *Consider following program P_4:*

$$\begin{aligned} p &\leftarrow \neg q \\ q &\leftarrow \neg p \\ r &\leftarrow r \end{aligned}$$

The interpretations $\{p, \neg q, \neg r\}$ and $\{\neg p, q, \neg r)\}$ are well-supported models of P_4. ∘

Another characterization of the meaning of a program is given by the definition of *stable models*. In the two-valued case [GL88], this definition uses the fact that the meaning of positive logic programs (in which the bodies of the clauses contain only positive literals) is well understood: it is given by the unique *two-valued minimal model* of the program. This definition of stable models has been generalized by T. Przymusinski to three-valued stable models [Prz90a]. In this definition, he uses the notion of (three-valued) truth-minimal models, and a program transformation.

Definition 2.3.8 Let P be a positive program and let M be a model of P. M is a *truth-minimal model* of P, if there does not exist a model M' (other than M) of P such that $M'^+ \subseteq M^+$ and $M'^- \supseteq M^-$. □

Definition 2.3.9 Let P be a program and let I be an interpretation of P. The program $\frac{P}{I}$ is obtained from P by replacing every negative literal L in the body of a clause in P that is true (resp. false; resp. unknown) in I by the proposition $\mathbf{t}$ (resp. $\mathbf{f}$; resp. $\mathbf{u}$). □

Now, we are able to give the definition of a *stable model.*

Definition 2.3.10 Let P be a program and let M be an interpretation of P. M is a *stable model* of P, if M is a truth-minimal model of $\frac{P}{M}$. □

Example 2.3.11 *Consider the program P_4 (Example 2.3.7), and the model $\{p, \neg q, \neg r\}$ of P_4. M is a stable model of P_4, because it is a truth-minimal model of the program $\frac{P_4}{M} = \{p \leftarrow \mathbf{t},\ q \leftarrow \mathbf{f},\ r \leftarrow r\}$.* ∘

The following lemma shows that the class of stable models coincides with a subclass of the well-supported models. This lemma is a generalization of the lemma by F. Fages [Fag91], which proves that two-valued stable models and two-valued well-supported models coincide. The proof we give, resembles the proof given by F. Fages. First, we have to introduce the notion of *(greatest) unfounded set*.

Definition 2.3.12 Let P be a program and let I be an interpretation of P. Let S be a subset of $\mathcal{B}_P - I^{\pm}$. S is an ***unfounded set*** of I, if all clauses R in P such that $concl(R) \in S$ are inapplicable in $I \cup \neg S$. The ***greatest unfounded set*** $U_P(\mathrm{I})$ of I is the union of all unfounded sets of I. $\square$

Note, that our definition of unfounded set differs from the definition used in [vGRS91]. However, we can define their operator as follows:

$$\mathbf{U}_P(I) = U_P(I) \cup I^-$$

Lemma 2.3.13 (Equivalence) *Let P be a program and let M be an interpretation of P. M is a stable model of P iff M is a well-supported model of P such that $U_P(M) = \emptyset$.*

Proof: By definition, M is a stable model of P iff M is a truth-minimal model of $\frac{P}{M}$. By Theorem 3.2 in [Prz90b] (page 451), the truth-minimal model can be characterized using the Fitting operator (see Definition 2.7.1); M is the truth-minimal model of $\frac{P}{M}$ iff $M = \Psi_{\frac{P}{M}} \uparrow^{\omega} (\emptyset)$. We write Ψ_α as a shorthand for $\Psi_{\frac{P}{M}} \uparrow^{\alpha} (\emptyset)$.
($\Leftarrow$) Let M be a well-supported model of P such that $U_P(M)$ is empty, and let $<_M$ be a well-founded support order on M. To prove that M is a stable model of P, it suffices to prove that $M = \Psi_\omega$.

1. We prove that $M^+ \subseteq \Psi^+_\omega$. In order to do this, we prove by induction on $<_M$ that $A \in M^+$ implies $A \in \Psi^+_\omega$. If A is a $<_M$-minimal element of M^+, then there exists a clause R in P with conclusion A that is applicable in M such that $prem(R)^+$ is empty. But then there exists a clause R' in $\frac{P}{M}$ with conclusion A that is applicable in M such that $prem(R)$ contains only propositional constants $\mathbf{t}$, and therefore by definition of Ψ, $A \in \Psi_\omega$. Assume that, for all $A' <_M A$, $A' \in M^+$ implies $A' \in \Psi_\omega$. Because $A \in M^+$, there exists a clause R in P with conclusion A that is applicable in M such that, for all $A' \in prem(R)^+$, $A' <_M A$. But then there exists a clause R' in $\frac{P}{M}$ with conclusion A that is applicable in M such that $A' \in prem(R)$ implies that A' is the propositional constant $\mathbf{t}$ or $A' <_M A$. By induction hypothesis, we have that $A' <_M A$ implies that $A' \in \Psi_\omega$. Therefore, R' is applicable in Ψ_ω and thus, by definition of Ψ_ω, $A \in \Psi_\omega$.

2. We prove by induction on α that $\Psi^+_\alpha \subseteq M^+$. For $\alpha = 0$, the lemma holds trivially. Assume that $\Psi^+_\alpha \subseteq M^+$. Suppose that $A \in \Psi^+_{\alpha+1}$. Then, there exists a clause R in $\frac{P}{M}$ with conclusion A that is applicable in Ψ_α. But then all elements of $prem(R)$ (excluding the propositional constant $\mathbf{t}$) are in Ψ_α and therefore in M. But then, there exists a corresponding clause R' in P with conclusion A that is applicable in M. But M is a model of P and therefore $A \in M$.

3. We prove that $M^- \subseteq \Psi^-_\omega$. Let $A \in M^-$. Because M is a model of P, every clause R in P with conclusion A is inapplicable in M. But then,

every clause R' in $\frac{P}{M}$ with conclusion A is inapplicable in M. But $\frac{P}{M}$ is a positive program, and therefore these clauses are also inapplicable in M^+. Also, we already have that $M^+ = \Psi_\omega^+$. So, because $\frac{P}{M}$ is positive, every clause R' in $\frac{P}{M}$ with conclusion A is inapplicable in Ψ_ω. Therefore, by definition of Ψ_ω, $A \in \Psi_\omega^-$.

4. We prove that $\Psi_\omega^- \subseteq M^-$. We already have that $M^- \subseteq \Psi_\omega^-$. Suppose that $S = \Psi_\omega^- - M^-$ is non-empty. Ψ_ω is a model of $\frac{P}{M}$. Therefore, for every $A \in S$, every clause R in $\frac{P}{M}$ with conclusion A is inapplicable in Ψ_ω. Because $M^+ = \Psi_\omega^+$, we know that $S \cap M^+ = \emptyset$. We also have that $M \cup \neg S = \Psi_\omega$. Therefore, for every $A \in S$, every clause R in $\frac{P}{M}$ with conclusion A is inapplicable in $M \cup \neg S$. But then by construction of $\frac{P}{M}$, for every $A \in S$, every clause R' in P with conclusion A is inapplicable in $M \cup \neg S$. So, S is an unfounded set of M. This is in contradiction with the fact that $U_P(M)$ is empty. Therefore S has to be empty.

($\Rightarrow$) Let $M = \Psi_\omega$. We have to prove that M is a well-supported model of P such that $U_P(M)$ is empty.

- We prove that $U_P(M)$ is empty. Suppose that $U_P(M)$ is non-empty. Consider the interpretation $M' = M \cup \neg U_P(M)$. Clearly, M' is smaller than M in the truth-ordering. But M' is also a model of P and $\frac{P}{M}$. This is in contradiction with the fact that $M = \Psi_\omega$ and that Ψ_ω is a truth-minimal model of $\frac{P}{M}$.

- We prove that there exists a well-founded support-order on M. We assign a *rank* to the elements of M^+: the rank $r(A)$ of an atom $A \in M^+$ is the least ordinal α such that $A \in \Psi_\alpha$. This rank is defined on all elements of M^+, because $M = \Psi_\omega$. We show that the partial ordering $<_r$ such that $A' <_r A$ iff $r(A') < r(A)$ is a well-founded support order on M. Clearly, $<_r$ is well-founded. Let A be an arbitrary element of M^+. We know that $A \in \Psi_{r(A)}$ iff there exists a clause R in $\frac{P}{M}$ that is applicable in $\Psi_{r(A)-1}$. But then, for all $A' \in prem(R)$, $r(A') < r(A)$ and therefore $A' <_r A$. By the construction of $\frac{P}{M}$ and the fact that M is a stable model of P, we have that there exists a clause R' in P with conclusion A that is applicable in M, such that, for all $A' \in prem(R)^+$, $A' <_r A$. Thus, $<_r$ is a well-founded support order on M. □

2.4 The Operator $\mathcal{S}_P$

In this section, we define the operator $\mathcal{S}_P$. This operator is inspired on the operator J_P^ρ of Fages, but there are some major differences.

The idea is, to generate all ***total*** stable models of a program, by starting from the empty interpretation. At each step, we try to extend an interpretation I to a new interpretation I', that brings us "nearer" to a total stable model. For this, we use the following strategies:

1. If there exists a clause R that is applicable in I and $concl(R)$ is not an element of I, then we add $concl(R)$ to I (after all, we are looking for a model).

2. If there exists an atom A such that all clauses R that have A as conclusion, are inapplicable in I, and $\neg A$ is not an element of I, then we add $\neg A$ to I (after all, we are working towards a total interpretation).

3. If the previous two strategies fail, we can do little more that blindly select an atom from $\mathcal{B}_P - I^{\pm}$, and add it or its negation, to I. However, in contrast with the two previous strategies, this strategy is flawed, in the sense that, even if I is a subset of some stable model, I' is not guaranteed to be a subset of a stable model. In fact, continuing the procedure with I' can lead to an inconsistent interpretation.

4. If I is inconsistent, then we should try to find a consistent interpretation I'. However, we do not want to throw away I completely. We know that the inconsistency was caused by some literal chosen by strategy 3. We will maintain "possible reasons for inconsistency" with our interpretation, in order to identify a literal in I that could be the reason for the inconsistency, and find a new consistent interpretation I' by removing from I all literals that were added to the interpretation due to the presence of this literal.

Note, that with all four strategies one could have more than one way to generate the next interpretation. For example, if there are two reasons for the inconsistency of an interpretation, there are two possibilities for resolving that inconsistency. As a result, our operator will be non-deterministic.

We have to maintain "reasons for inconsistency" with our interpretation. Moreover, we maintain a support order with our interpretation, to help us prove various properties. This leads to the following definition of ***j-interpretations***.

Definition 2.4.1 A *j-triple*, is a triple $\langle L, \tau, \psi \rangle$, such that L is an element of $\mathcal{L}_P$, and τ and ψ are subsets of $\mathcal{L}_P$. A ***j-interpretation*** J of P is a set of j-triples such that for every literal in $\mathcal{L}_P$, J contains at most one j-triple with that literal as the first element. We call τ the ***support-set*** of L and ψ the ***culprit-set*** of L. For a set S of j-triples, we write $\overline{S}$ to denote the set of literals $\{L \mid \langle L, \tau, \psi \rangle \in S\}$. □

The idea is, that a j-interpretation is an interpretation in which literals are annotated with two sets, a support set and a culprit set. The support set for a literal contains a justification for the fact that that literal is in the interpretation and, in inconsistent interpretations, for every conflicting pair $\{A, \neg A\}$ of atoms, the culprit sets of A and $\neg A$ contain possible causes for that inconsistency.

Note, that our support-set differs from the justification in a justified atom of Fages, because it can be infinite, and it is defined on literals instead of atoms. Moreover, our support-set is intended to contain a set of premises for a positive literal, and a set of elements of blocking-sets for negative literals, whereas the justifications of Fages contain a complete explanation for the fact that an atom is true. Using the support-sets in a j-interpretation J, we can define a partial order on the literals in $\overline{J}$.

Definition 2.4.2 Let J be a j-interpretation. We define $<_J$ to be the partial order such that $A' <_J A$ iff $\langle A, \tau, \psi\rangle \in J$ and $A' \in \tau^+$ (note, that A is a positive literal). □

In the interpretations on which $\mathcal{S}_P$ operates, the culprit-set contains the "possible reasons for inconsistency" and the partial order $<_J$ is a support order on $\overline{J}$.

In the definition of the operator $\mathcal{S}_P$, we use the *conflict-set*, *choice-set* and *culprit-set* of a j-interpretation J. The *conflict-set* of a j-interpretation J contains j-triples for every literal L for which there are one or more reasons for adding them to J, according to strategies 1 and 2.

Definition 2.4.3 Let P be a program and let J be a j-interpretation of P. The *conflict-set* $Conflict_P(J)$ of J is the set of j-triples $\langle L, \tau, \psi\rangle$ such that

- $L \notin \overline{J}$,
- if $L = A$, then there exists a clause R in P with conclusion A that is applicable in $\overline{J}$ such that $\tau = prem(R)$,
- if $L = \neg A$, then every clause R in P with conclusion A is inapplicable in $\overline{J}$, and for every clause R in P with conclusion A exists a literal L_R in the blocking-set of R in $\overline{J}$ such that $\tau = \{L_R \mid R \in P \wedge concl(R) = A\}$, and
- $\psi \overset{def}{=} \bigcup\{\psi' \mid \langle L', \tau', \psi'\rangle \in J \wedge L' \in \tau\}$. □

For a j-triple $\langle L, \tau, \psi\rangle$ in $Conflict_P(J)$, τ contains the reason for adding L to J, and ψ contains all literals that could be the cause of L being an element of $\overline{Conflict_P(J)}$, while $\neg L$ is an element of $\overline{J}$.

The *choice-set* of J contains j-triples that could be added to J on behalf of strategy 3. The support-sets and choice-sets of these j-triples reflect the fact that there is no real support for adding these literals to J.

Definition 2.4.4 Let P be a program and let J be a j-interpretation of P. The *choice-set* $Choice_P(J)$ of P is the set

$$\{\langle L, \emptyset, \{L\}\rangle \mid L \in \neg(\mathcal{B}_P - \overline{J}^{\pm})\}$$

□

The ***culprit-set*** of an inconsistent j-interpretation J, is the set of all "possible reasons for inconsistency"; that is, the set of literals that are common to the culprit-sets of all literals L in $\overline{J}$ whose negation $\neg L$ is also an element of $\overline{J}$.

Definition 2.4.5 Let P be a program and let J be a j-interpretation of P. The *culprit-set* $Culprit_P(J)$ of J is the set

$$\bigcap\{\psi \cup \psi' \mid \langle A, \tau, \psi\rangle \in J \wedge \langle \neg A, \tau', \psi'\rangle \in J\}$$

□

Note, that if $\overline{J}$ is consistent then $Culprit_P(J) = \emptyset$. We are now capable of defining our operator $\mathcal{S}_P$.

Definition 2.4.6 For a normal logic program P, we define the operator $\mathcal{S}_P$ as follows:

$$\mathcal{S}_P(J) \overset{\text{def}}{=} \begin{cases} J - \{\langle L, \tau, \psi\rangle \mid \rho_1 \in \psi\} & \text{, if } Culprit_P(J) \neq \emptyset \\ J \cup \{\rho_2\} & \text{, if } Conflict_P(J) \neq \emptyset \\ J \cup \{\rho_3\} & \text{, if } Choice_P(J) \neq \emptyset \\ J & \text{, otherwise} \end{cases}$$

where $\rho_1 \in Culprit_P(J)$
$\rho_2 \in Conflict_P(J)$
$\rho_3 \in Choice_P(J)$

□

Note, that in this definition the order of the conditions is relevant (i.e. a rule is only applied if its condition is satisfied *and* the conditions of all previous rules failed).

The operator as we defined it, is non-deterministic, in the sense that it non-deterministically chooses an element (ρ_1, ρ_2 or ρ_3) from a set of candidates. Because we want to manipulate this non-deterministic behaviour, we extend the operator with a *selection strategy*, that encapsulates this non-deterministic behaviour of $\mathcal{S}_P$.

Definition 2.4.7 Let P be a program. A *selection strategy* ρ for P is a non-deterministic function that, for a j-interpretation J of P, chooses ρ_1 among $Culprit_P(J)$, ρ_2 among $Conflict_P(J)$ and ρ_3 among $Choice_P(J)$. □

Note, that ρ can be deterministic if we consider more information. For instance, we could use a selection strategy that bases its choices for some j-interpretation J on the way in which J was generated (i.e. previous applications of $\mathcal{S}_P$). We use the notation $\mathcal{S}_P^\rho$ to indicate that we are using the operator on a program P with a selection strategy ρ for P.

As said before, we want to find a stable model of P by starting from the empty interpretation. In order to do this, we have to define the (ordinal) powers of $\mathcal{S}_P^\rho$.

Definition 2.4.8 Let P be a program and let ρ be a selection strategy for P. Let $\mathcal{S}_P^\rho$ be the operator as defined. We define the powers of $\mathcal{S}_P^\rho$ inductively:

$$\mathcal{S}_P^\rho \uparrow^\alpha = \begin{cases} \emptyset & \text{, if } \alpha = 0 \\ \mathcal{S}_P^\rho(\mathcal{S}_P^\rho \uparrow^{\alpha-1}) & \text{, if } \alpha \text{ is a successor ordinal} \\ \bigcup_{\beta<\alpha} \bigcap_{\beta\leq\gamma<\alpha} \mathcal{S}_P^\rho \uparrow^\gamma & \text{, if } \alpha \text{ is a limit ordinal} \end{cases}$$

□

The definition for zero and successor ordinals are standard. The definition for limit ordinal is the same as the one used by Fages; it states that at a limit ordinal α, we retain only the j-triples that where persistent in the preceding sequence of j-interpretations; that is, for every j-triple in $\mathcal{S}_P^\rho \uparrow^\alpha$, there exists an ordinal β smaller that α, such that, for all $\gamma \in [\beta..\alpha)$, this j-triple is an element of $\mathcal{S}_P^\rho \uparrow^\gamma$.

Using the powers of $\mathcal{S}_P^\rho$, we define the following infinite sequence of j-interpretations.

Definition 2.4.9 Let P be a program and let ρ be a selection strategy for P. The *sequence for P and ρ* is the transfinite sequence of j-interpretations

$$\Gamma_P^\rho \overset{def}{=} \mathcal{S}_P^\rho \uparrow 0, \mathcal{S}_P^\rho \uparrow 1, \ldots, \mathcal{S}_P^\rho \uparrow \alpha, \ldots$$

□

We now work towards a proof of the fact that certain fixpoints of $\mathcal{S}_P$ are stable models of P. First, we have to prove that the application of $\mathcal{S}_P$ on a j-interpretation results in a j-interpretation, and that every element of a sequence is a j-interpretation.

Lemma 2.4.10 *Let P be a program and let ρ be a selection strategy for P. If J is a j-interpretation, then $\mathcal{S}_P^\rho(J)$ is a j-interpretation.*

Proof: Suppose J is a j-interpretation. Then, we can obtain $\mathcal{S}_P^\rho(J)$ from J in two different ways:

- By adding a j-triple $\langle L, \tau, \psi\rangle$ to J. By definition of $\mathcal{S}_P$ (conflict-set and choice-set), we know that $L \notin \overline{J}$. It follows that $\mathcal{S}_P^\rho(J) = J \cup \{\langle L, \tau, \psi\rangle\}$ is a j-interpretation of P.

- By removing elements from J. Because any subset of a j-interpretation is itself a j-interpretation, we have that $\mathcal{S}_P^\rho(J)$ is a j-interpretation. □

Lemma 2.4.11 *Let Γ_P^ρ be a sequence for a program P and a selection strategy ρ. Every element J_α of Γ_P^ρ is a j-interpretation of P.*

Proof: For $J_0 = \emptyset$, the claim is trivially true. Assume that for all $\beta < \alpha$, J_β is a j-interpretation of P.

If α is a successor ordinal, $J_{\alpha-1}$ is a j-interpretation by induction hypothesis, and therefore, by Lemma 2.4.10, J_α is a j-interpretation.

If α is a limit ordinal, we know that it is a set of j-triples, because it is a subset of a union of j-interpretations. Furthermore, we have that if $\langle L, \tau, \psi \rangle \in J_\alpha$, then for some β such that $\beta < \alpha$ we have that, for all $\gamma \in [\beta..\alpha)$, $\langle L, \tau, \psi \rangle \in J_\gamma$. By induction hypothesis, for all $\gamma \in [\beta..\alpha)$, J_γ is a j-interpretation and therefore there is no j-triple other than $\langle L, \tau, \psi \rangle$ in J_γ with L on the first position. But then we have that there is no j-triple, other than $\langle L, \tau, \psi \rangle$, in J_α with L on the first position. Therefore, J_α is a j-interpretation. □

We now prove that for every j-interpretation J_α in a sequence Γ_P^ρ, the partial order $<_{J_\alpha}$ is a support order and a well-founded order. First, we have to prove the following auxiliary lemma.

Lemma 2.4.12 *Let Γ_P^ρ be a sequence. For all J_α in Γ_P^ρ, for all $\langle L, \tau, \psi \rangle \in J_\alpha$ and for all $L' \in \tau$, there exist a τ' and a $\psi' \subseteq \psi$ such that $\langle L', \tau', \psi' \rangle \in J_\alpha$.*

Proof: For $J_0 = \emptyset$, the claim is trivially true. Assume that, for all β smaller than α, we have that $\langle L, \tau, \psi \rangle \in J_\beta$ implies that, for all $L' \in \tau$, there exist a τ' and a $\psi' \subseteq \psi$ such that $\langle L', \tau', \psi' \rangle \in J_\beta$.

If α is a successor ordinal, J_α can be obtained from $J_{\alpha-1}$ in two ways:

- By adding a j-triple $\langle L, \tau, \psi \rangle$ to $J_{\alpha-1}$. Here, the claim follows directly from the definition of $\mathcal{S}_P$ (conflict-set and choice-set).
- By removing elements from $J_{\alpha-1}$. Suppose $\langle L, \tau, \psi \rangle \in J_{\alpha-1}$ and $L' \in \tau$. Then, we have by induction hypothesis that there exist a τ' and a $\psi' \subseteq \psi$ such that $\langle L', \tau', \psi' \rangle \in J_{\alpha-1}$. Because $\psi' \subseteq \psi$, we have by the definition of $\mathcal{S}_P$ that $L' \notin J_\alpha$ implies that $L \notin J_\alpha$.

If α is a limit ordinal, we have that if $\langle L, \tau, \psi \rangle \in J_\alpha$, then for some β such that $\beta < \alpha$ we have that, for all $\gamma \in [\beta..\alpha)$, $\langle L, \tau, \psi \rangle \in J_\gamma$. By induction hypothesis, we have that, for all $\gamma \in [\beta..\alpha)$ and for all $L' \in \tau$, there exist a τ' and a $\psi' \subseteq \psi$ such that $\langle L', \tau', \psi' \rangle \in J_\gamma$. Also, we have that if $\langle L', \tau', \psi' \rangle \in J_\gamma$ and $\langle L', \tau'', \psi'' \rangle \in J_{\gamma+1}$, then $\tau' = \tau''$ and $\psi' = \psi''$. Therefore, $\langle L', \tau', \psi' \rangle \in J_\alpha$. □

Theorem 2.4.13 (Supportedness) *Let Γ_P^ρ be a sequence for a program P. For every J_α in Γ_P^ρ, the partial order $<_{J_\alpha}$ is a support order on $\overline{J}_\alpha$.*

Proof: We have to prove that for all $A \in \overline{J}_\alpha^+$ there exists an applicable clause R in P with conclusion A such that for all $A' \in \mathit{prem}(R)^+$, $A' <_{J_\alpha} A$.

We proceed by induction on α. For $J_0 = \emptyset$, the claim holds trivially. Assume that for all β smaller than α and for all $A \in \overline{J}_\beta^+$ there exists a applicable clause R in P with conclusion A such that for all $A' \in \mathit{prem}(R)^+$, $A' <_{J_\beta} A$.

If α is a successor ordinal, then J_α can be obtained from $J_{\alpha-1}$ in two ways:

1. By adding a j-triple $\langle L, \tau, \psi \rangle$ to $J_{\alpha-1}$. If L is a negative literal, then $<_{J_\alpha} \equiv <_{J_{\alpha-1}}$ and the claim follows from the induction hypothesis and the fact that $J_\alpha \supseteq J_{\alpha-1}$. If L is positive, we have by the definition of $\mathcal{S}_P$ (conflict-set) that there exists a applicable clause R in P with conclusion L such that $\tau = prem(R)$. Therefore, $A \in prem(R)^+$ implies that $A \in \tau^+$, which, by definition of $<_{J_\alpha}$, implies that $A <_{J_\alpha} L$.

2. By removing a set of j-triples from $J_{\alpha-1}$. The claim then follows from Lemma 2.4.12 and the fact that $<_{J_{\alpha-1}}$ is a support order on $J_{\alpha-1}$.

If α is a limit ordinal, then $A \in \overline{J}_\alpha^+$ implies that there exists an β such that $\beta < \alpha$ and for some τ and ψ, for all $\gamma \in [\beta..\alpha)$, $\langle A, \tau, \psi \rangle \in J_\gamma$. By induction hypothesis we have that there exists a applicable clause $R \in P$ with conclusion A such that for all $A' \in prem(R)^+$, $A' <_{J_\gamma} A$ and therefore that $A' \in \tau$. By Lemma 2.4.12 we have that there exist τ' and ψ' such that $\langle A', \tau', \psi' \rangle \in J_\gamma$ and therefore $\langle A', \tau', \psi' \rangle \in J_\alpha$. From this we can conclude that if $A' \in prem(R)^+$ then $A' <_{J_\alpha} A$. □

Theorem 2.4.14 (Well-Foundedness) *Let Γ_P^ρ be a sequence for a program P. For every J_α in Γ_P^ρ, the partial order $<_{J_\alpha}$ is well-founded.*

Proof: Suppose that $<_{J_\alpha}$ is not well-founded. Then, there exists an infinite decreasing chain $\ldots <_{J_\alpha} A_2 <_{J_\alpha} A_1 <_{J_\alpha} A_0$. Because $A_i \in \overline{J}_\alpha^+$, there exists a least ordinal β_i such that $\beta_i \leq \alpha$ and for some τ_i and ψ_i, for all $\gamma \in [\beta_i..\alpha]$, $\langle A_i, \tau_i, \psi_i \rangle \in J_\gamma$. Also, because $A_{i-1} \in \overline{J}_\alpha^+$, there exists a least ordinal β_{i-1} such that $\beta_{i-1} \leq \alpha$ and for some τ_{i-1} and ψ_{i-1}, for all $\gamma \in [\beta_{i-1}..\alpha]$, we have that $\langle A_{i-1}, \tau_{i-1}, \psi_{i-1} \rangle \in J_\gamma$. Furthermore, we have that $A_i <_{J_\alpha} A_{i-1}$, which implies that $A_i \in \tau_{i-1}$, and therefore $\beta_i < \beta_{i-1}$. As a result, we have that $\ldots < \beta_2 < \beta_1 < \beta_0$ is an infinite decreasing chain. But the $<$ order on ordinals is well-founded. Thus, the assumption that $<_{J_\alpha}$ is not well-founded is in contradiction with the fact that the $<$ order on ordinals is well-founded. Therefore, we can conclude that $<_{J_\alpha}$ is well-founded. □

We now show that all fixpoints of $\mathcal{S}_P$ that appear in sequences are consistent. In order to prove this, we need a few auxiliary lemmas.

Lemma 2.4.15 *Let Γ_P^ρ be a sequence for a program P. Let α be the least ordinal such that $Conflict_P(J_\alpha) = \emptyset$. Then, for all $\beta \in [0..\alpha]$, $\overline{J}_\beta$ is consistent.*

Proof: We prove the claim with induction on β. For $\beta = 0$, we have that $J_\beta = \emptyset$, which is consistent. Assume that for all γ smaller than β, $\overline{J}_\gamma$ is consistent.

Suppose that β is a successor ordinal. If $\beta \notin [0..\alpha]$ or $\overline{J}_\beta$ is consistent, then the claim holds trivially. So, assume that $\beta \in [0..\alpha]$ and that $\overline{J}_\beta$ is inconsistent. Then, we have that $\overline{J}_\beta = \overline{J}_{\beta-1} \cup \{\neg L\}$, where $L \in \overline{J}_{\beta-1}$. First, note that by induction hypothesis, for all γ smaller than β, $\overline{J}_\gamma$ is consistent, and therefore $J_\gamma \subseteq J_{\gamma+1}$. As a result, every clause that is applicable (resp. inapplicable) in $\overline{J}_\gamma$, is applicable (resp. inapplicable) in $\overline{J}_{\gamma+1}$. There are two cases:

1. L is positive. Because $L \in \overline{J}_{\beta-1}$, there has to be at least one clause with conclusion L that is applicable in $J_{\beta-1}$. Also, by induction hypothesis, $J_{\beta-1}$ is consistent. Therefore, there exists at least one clause with conclusion L that is not inapplicable in $J_{\beta-1}$. But then, $\neg L \notin Conflict_P(J_{\beta-1})$. This contradicts the fact that $\beta \in [0..\alpha]$ and $\overline{J}_\beta = \overline{J}_{\beta-1} \cup \{\neg L\}$.

2. L is negative. Because $L \in \overline{J}_{\beta-1}$, all clauses with conclusion $\neg L$ have to be inapplicable in $J_{\beta-1}$. Also, by induction hypothesis, $J_{\beta-1}$ is consistent. Therefore, there does not exists a clause with conclusion $\neg L$ that is applicable in $J_{\beta-1}$. But then, $\neg L \notin Conflict_P(J_{\beta-1})$. This contradicts the fact that $\beta \in [0..\alpha]$ and $\overline{J}_\beta = \overline{J}_{\beta-1} \cup \{\neg L\}$.

Suppose that β is a limit ordinal. Then J_β is consistent, because it is the union of a monotone increasing chain of consistent interpretations. □

Lemma 2.4.16 *Let Γ_P^ρ be a sequence for a program P. Let α be the least ordinal such that $Conflict_P(J_\alpha) = \emptyset$. For all β greater than α and for all $\langle L, \tau, \psi \rangle \in J_\beta - J_\alpha$, the culprit-set ψ is non-empty.*

Proof: Suppose that for some γ greater than α and some $\langle L, \tau, \psi \rangle \in J_\gamma - J_\alpha$, the culprit-set ψ is empty. Let β be the least ordinal greater than α such that for some $\langle L, \tau, \psi \rangle \in J_\beta - J_\alpha$, ψ is empty. Because ψ is empty, the j-triple can only have been added on behalf of $Conflict_P(J_{\beta-1})$. There are two cases:

1. If L is a positive literal, then ψ is the union of the culprit-sets of the literals in *prem*(R), where R is a applicable clause with conclusion L. Clearly, *prem*(R) is non-empty, because otherwise $L \in \overline{J}_\alpha$. But if *prem*$(R)$ is non-empty and ψ is empty, then the culprit-sets of all the literals in *prem*(R) have to be empty But then all these literals are elements of J_α, and therefore $L \in \overline{Conflict_P(J_\alpha)}$. This contradicts the fact that $Conflict_P(J_\alpha) = \emptyset$.

2. If L is a negative literal, then ψ is the union of the culprit-sets of a set of literals that block all clauses with conclusion $\neg L$. This set is non-empty, because otherwise $L \in \overline{J}_\alpha$. But if this set is non-empty and ψ is empty, then the culprit-sets of all these literals have to be empty. But then, all these literals are elements of J_α, and therefore $L \in \overline{Conflict_P(J_\alpha)}$. This contradicts the fact that $Conflict_P(J_\alpha) = \emptyset$.

From these contradictions, we have that there cannot exist a least β greater than α such that for some $\langle L, \tau, \psi \rangle \in J_\beta - J_\alpha$, the culprit-set ψ is empty. □

Lemma 2.4.17 *Let Γ_P^ρ be a sequence for a program P. Let J_α be an element of Γ_P^ρ. If $\overline{J}_\alpha$ is inconsistent, then $\overline{J}_{\alpha+1}$ is consistent.*

Proof: We prove the claim by induction on α. The induction base holds trivially: $\overline{J}_0 = \emptyset$ is consistent. Assume that, for all ordinals β smaller than α, $\overline{J}_{\beta+1}$ is consistent if $\overline{J}_\beta$ is inconsistent.

Suppose that α is a successor ordinal and $\overline{J}_\alpha$ is inconsistent. It follows from Lemma 2.4.15 that α is greater than γ, where γ is the least ordinal such that $Conflict_P(J_\gamma) = \emptyset$. It is sufficient to prove that $Culprit_P(J_\alpha) \neq \emptyset$, because then is follows from the definition of $\mathcal{S}_P$ that $J_{\alpha+1}$ is consistent. First, observe that there is exactly one atom A such that both $\langle A, \tau, \psi \rangle$ and $\langle \neg A, \tau' \psi' \rangle$ are elements of J_α; at least one, because $\overline{J}_\alpha$ is inconsistent and at most one because by induction hypothesis $\overline{J}_{\alpha-1}$ is consistent. As a result, we have that $Culprit_P(J_\alpha) = \psi \cup \psi'$. We also know that at least one of these two j-triples is not an element of J_γ, because $\overline{J}_\gamma$ is consistent. Therefore, by Lemma 2.4.16 we have that at least one of ψ and ψ' is non-empty, and thus $\psi \cup \psi'$ is non-empty.

If α is a limit ordinal we have by induction hypothesis that, for all β smaller than α such that $\overline{J}_\beta$ is inconsistent, $\overline{J}_{\beta+1}$ is consistent. Therefore, for all β smaller than α such that $\overline{J}_\beta$ is inconsistent, $\bigcap_{\beta \leq \delta < \alpha} J_\delta \subseteq J_{\beta+1} \subset J_\beta$. From this we can conclude that J_α is consistent. □

Theorem 2.4.18 (Fixpoint Consistency) *Let Γ_P^ρ be a sequence for a program P. Let J_α be an element of Γ_P^ρ. If J_α is a fixpoint of $\mathcal{S}_P$, then $\overline{J}_\alpha$ is consistent.*

Proof: Suppose $\overline{J}_\alpha$ is inconsistent. Then, by Lemma 2.4.17, $J_{\alpha+1}$ is consistent. But then $J_\alpha \neq J_{\alpha+1}$. This is in contradiction with the fact that J_α is a fixpoint of $\mathcal{S}_P$. □

2.5 Total Stable Models as Limit Fixpoints of $\mathcal{S}_P$

We now take a look at the fixpoints of $\mathcal{S}_P$ that appear in the sequences of P (we call them *limit fixpoints*), and prove that they are the total stable models of P. First, we have to define the class of sequences that contain a fixpoint: ***stabilizing sequences***.

Definition 2.5.1 A sequence Γ_P^ρ is *stabilizing*, if there exists an ordinal α, such that, for all ordinals β greater than α, $J_\alpha = J_\beta$. The *closure ordinal* of Γ_P^ρ is the least ordinal α, such that, for all ordinals β greater than α, $J_\alpha = J_\beta$. □

Definition 2.5.2 Let P be a program. A j-interpretation J is a *limit fixpoint* of $\mathcal{S}_P$, if there exists a selection strategy ρ for P, such that the sequence Γ_P^ρ is stabilizing and $J = J_\alpha$, where α is the closure ordinal of Γ_P^ρ. □

Theorem 2.5.3 *Let P be a program. If J is a limit fixpoint of $\mathcal{S}_P$, then $\overline{J}$ is a total stable model of P.*

Proof: J is a limit fixpoint of $\mathcal{S}_P$. Therefore, there exists a selection strategy ρ such that Γ_P^ρ is stabilizing and $J = J_\alpha$, where α is the limit ordinal of Γ_P^ρ. By the Fixpoint Consistency Theorem (2.4.18), $\overline{J}_\alpha$ is consistent. By the construction of $\mathcal{S}_P$ and the fact that $J_\alpha = J_{\alpha+1}$, $\overline{J}_\alpha$ is a total model of P. Also, by the Supportedness Theorem (2.4.13) and the Well-Foundedness Theorem (2.4.14), $<_{J_\alpha}$ is a well-founded support order for $\overline{J}_\alpha$. Therefore, $\overline{J}$ is a total well-supported model of P. Because $\overline{J}$ is total, $U_P(\overline{J})$ is empty. From the Equivalence Lemma (2.3.13), we conclude that $\overline{J}$ is a total stable model of P. □

So, the limit fixpoints of $\mathcal{S}_P$ are total stable models of P. We now show the converse: every total stable model is a limit fixpoint of $\mathcal{S}_P$. We define, for every stable model M of P, a class of selection strategies ρ such that M is contained in Γ_P^ρ.

Definition 2.5.4 Let P be a program and let M be a stable model of P. A *selection strategy for M* is a selection strategy that, for all J such that $\overline{J} \subset M$, selects a j-triple $\langle L, \tau, \psi \rangle$ from $Conflict_P(J)$ or $Choice_P(J)$ such that $L \in M$. □

Lemma 2.5.5 *Let P be a program, let M be a stable model of P and let J be a j-interpretation such that $\overline{J} \subset M$. Then $\overline{Conflict_P(J)} \subseteq M$*

Proof: Suppose $A \in \overline{Conflict_P(J)}^+$. Then, there exists a clause with conclusion A that is applicable in $\overline{J}$. By construction of $\overline{J}$, this clause is also applicable in M, and therefore A has to be an element of M.

Suppose $A \in \overline{Conflict_P(J)}^-$. Then, all clauses with conclusion A are inapplicable in $\overline{J}$. By construction of $\overline{J}$, these clauses are also inapplicable in M. As a result, we have that every clause in $\frac{P}{M}$ with conclusion A is inapplicable. Because M is the truth-minimal model of $\frac{P}{M}$, we can conclude that $\neg A$ is an element of M. □

Lemma 2.5.6 *Let P be a program and let M be a stable model of P. Then, there exists a selection strategy ρ for M and for some J_α in Γ_P^ρ, $M = \overline{J}_\alpha$.*

Proof: First, we have to prove that there exists a selection strategy for M. Suppose that J is a j-interpretation such that $\overline{J} \subseteq M$.

1. Suppose ρ has to select from $Conflict_P(J)$.

   Then, by Lemma 2.5.5, any element select by a selection strategy is an element of M.

2. Suppose $Conflict_P(J) = \emptyset$ and $\overline{Choice_P(J)} \cap M \neq \emptyset$.

   Then we can select an element of M from $\overline{Choice_P(J)}$. Therefore, there exists a selection strategy that selects an element of M from $\overline{Choice_P(J)}$.

3. Suppose $Conflict_P(J) = \emptyset$ and $\overline{Choice_P(J)} \cap M = \emptyset$.

   Because $\overline{Choice_P(J)} = \neg(\mathcal{B}_P - \overline{J}^{\pm})$ and $\overline{J} \subseteq M$, we have that $\overline{J}^- = M^-$. Because $Conflict_P(J) = \emptyset$ and M is a supported model of P, we have that $\overline{J}^+ = M^+$. This is in contradiction with the fact that $\overline{J} \subset M$.

So, there exists a selection strategy ρ for M. Consider the sequence Γ_P^ρ.

Suppose that M is a total model. Then, Γ_P^ρ is stabilizing, and at its closure ordinal α, $\overline{J}_\alpha = M$.

Suppose M is not a total model. Consider any initial segment $J_0, \ldots, J_\alpha$ of Γ_P^ρ such that for all $\beta \leq \alpha$, $\overline{J}_\beta \subseteq M$. Then, because ρ is a selection strategy for M, for all $\beta < \alpha$, $\overline{J}_\beta \subseteq \overline{J}_{\beta+1}$. But then, because M is not total, there exists a least ordinal α such that $\overline{J}_\alpha \not\subset M$.

1. If $\alpha = 0$, then $\overline{J}_\alpha = \emptyset \subseteq M$. Because $J_\alpha \not\subset M$, it follows that $J_\alpha = M$.

2. If α is a successor ordinal, then, by definition of ρ, $\overline{J}_{\alpha-1} \subset M$. Also, by definition of ρ, $\overline{J}_\alpha = \overline{J}_{\alpha-1} \cup \{L\}$, where $L \in M$. Because $\overline{J}_\alpha \not\subset M$, we have that $\overline{J}_\alpha = M$.

3. If α is a limit ordinal, then we have that for all β smaller than α, $\overline{J}_\beta \subset M$. By definition of ρ, the prefix of Γ_P^ρ up to (not including) $\overline{J}_\alpha$ is a monotone increasing chain. Therefore, $\overline{J}_\alpha = \bigcup_{\beta<\alpha} \overline{J}_\beta \subseteq M$. Because $\overline{J}_\alpha \not\subset M$, we have that $\overline{J}_\alpha = M$.

So, there exists a J in Γ_P^ρ such that $\overline{J} = M$. □

Theorem 2.5.7 (Characterization) *Let P be a program. The limit fixpoints of $\mathcal{S}_P$, coincide with the total stable models of P.*

Proof: We have from Theorem 2.5.3 that all limit fixpoints of $\mathcal{S}_P$ contain stable models of P. Also, by Lemma 2.5.6, there exists for every (total) stable model M of P a selection strategy ρ such that M is contained in an element of Γ_P^ρ. Because M is total, it follows that M is a limit fixpoint of $\mathcal{S}_P$. □

2.6 A Characterization of Stable Models, using $\mathcal{S}_P$

In this section, we characterize the stable models of a program P, using our operator $\mathcal{S}_P$. As we have seen, the total stable models coincide with the limit fixpoints of $\mathcal{S}_P$. This means that we cannot characterize the set of all three-valued stable models as a set of fixpoints of $\mathcal{S}_P$. Instead, we identify the set of stable models of a program with some set of j-interpretations appearing in the sequences for that program.

Lemma 2.6.1 *Let P be a program and let M be an interpretation of P. M is a stable model of P iff there exists a j-interpretation J in a sequence for P, such that $M = \overline{J}$, $\overline{J}$ is consistent, $Conflict_P(J) = \emptyset$ and $U_P(\overline{J}) = \emptyset$.*

Proof:
($\Leftarrow$) Let J be an element of a sequence for P for which we have that $\overline{J}$ is consistent, $Conflict_P(J) = \emptyset$ and $U_P(\overline{J}) = \emptyset$. By the Supportedness Theorem (2.4.13) and the Well-Foundedness Theorem (2.4.14), $\overline{J}$ is a well-supported interpretation of P. Also, we know that $\overline{J}$ is consistent and that $U_P(\overline{J}) = \emptyset$. Because $Conflict_P(J) = \emptyset$, we know that for every clause R that is applicable in $\overline{J}$, $concl(R) \in \overline{J}$. Therefore, $\overline{J}$ is a model of P. Finally, by the Equivalence Lemma (2.3.13), $\overline{J}$ is a stable model of P.

($\Rightarrow$) Let M be a stable model of P. By Lemma 2.5.6, there exists a strategy ρ such that there exists an element J of Γ_P^ρ where $M = \overline{J}$. Clearly, M is consistent. So, we only have to prove that $Conflict_P(J) = \emptyset$ and that $U_P(\overline{J}) = \emptyset$.

- Suppose that $\langle L, \tau, \psi \rangle \in Conflict_P(J)$. If L is positive, then there exists a clause with conclusion L that is applicable in $\overline{J}$. But $\overline{J} = M$ and M is a model of P and therefore $L \in \overline{J}$. If L is negative, then all clauses with conclusion $\neg L$ are inapplicable in $\overline{J}$. The corresponding clauses in $\frac{P}{\overline{J}}$ will also be inapplicable. Because $\overline{J} = M$ and M is a stable model of P, M is a truth-minimal model of $\frac{P}{M}$ and therefore $L \in \overline{J}$. But the fact that $L \in \overline{J}$ is, by definition of $Conflict_P$, in contradiction with the fact that $\langle L, \tau, \psi \rangle \in Conflict_P(J)$.

- Suppose that $U_P(\overline{J}) \neq \emptyset$. Let $M' = M \cup \neg U_P(\overline{J})$. Clearly, M' is smaller than M in the truth-ordering. But M' is also a model of $\frac{P}{M}$. This is in contradiction with the fact that M is a stable model of P. □

2.7 Relating the Fixpoint of the Fitting Operator to the Sequences for P

In the operator $\mathcal{S}_P$, we have a preference for using elements of $Conflict_P$ to extend an interpretation. The definition of $Conflict_P$ bares resemblance to the sets T_P and F_P used by the Fitting operator [Fit85]. We can identify the least fixpoint of the Fitting operator Φ_P with a special j-interpretation that appears in every sequence for P (in fact, it is the last element of the maximal prefix shared by all sequences for P). First, we recall definition of the Fitting operator.

Definition 2.7.1 Let P be a program. The Fitting operator Φ_P is defined as follows:

$$\begin{aligned}
&\Phi_P(I) = T_P(I) \cup F_P(I) \\
&\text{where } \; T_P(I) = \{A \mid \exists_{R \in P} concl(R) = A \wedge prem(R) \subseteq I\} \\
&\qquad\quad\; F_P(I) = \{\neg A \mid \forall_{R \in P} concl(R) = A \rightarrow \neg prem(R) \cap I \neq \emptyset\}
\end{aligned}$$

□

The powers of the Fitting operator can be defined in the same way as we did for $\mathcal{S}_P$. Although the definition of Fitting differs in the case of limit ordinals, we can safely use our definition, because Φ_P is monotone, and for monotone operators both definitions coincide.

Lemma 2.7.2 *Let Γ_P^ρ be a sequence for a program P. Let α be the least ordinal such that $Conflict_P(J_\alpha) = \emptyset$. Then, $\overline{J_\alpha}$ is the least fixpoint of the Fitting operator Φ_P.*

Proof: Let M be the least fixpoint of Φ_P. We have that $M = \Phi\uparrow^\phi(\emptyset)$, where ϕ is the closure ordinal of Φ_P. We prove that $\overline{J_\alpha} \subseteq M$ and $\overline{J_\alpha} \supseteq M$.

1. We prove by induction on β that if $\beta \leq \alpha$ then $\overline{J_\beta} \subseteq M$. For $\overline{J_0} = \emptyset$, the claim holds trivially. Assume that for all $\gamma < \beta \leq \alpha$, $\overline{J_\gamma} \subseteq M$.

   If β is a successor ordinal, we have that $J_\beta = J_{\beta-1} \cup \{\langle L, \tau, \psi\rangle\}$. By induction hypothesis, we have that $\overline{J_{\beta-1}} \subseteq M$. Also, by the definition of $Conflict_P(J)$ and Φ_P, we have that $\overline{Conflict_P(J_{\beta-1})} \subseteq M$. Therefore, $\overline{J_\beta} \subseteq M$.

   If β is a limit ordinal, we have, because $\beta \leq \alpha$, that $J_\beta = \bigcup_{\gamma<\beta} J_\gamma$. By induction hypothesis, we have that $\overline{J_\gamma} \subseteq M$, for all $\gamma < \beta$. Therefore, $\overline{J_\beta} \subseteq M$.

2. We have to prove that $\overline{J_\alpha} \supseteq M$. It is enough to prove that $L \notin \overline{J_\alpha}$ implies that $L \notin M$. Suppose $L \notin \overline{J_\alpha}$. There are two cases:

   - L is positive.

By definition of $\mathcal{S}_P$ and the fact that $Conflict_P(J_\alpha) = \emptyset$, we know that all clauses with conclusion L are not applicable in $\overline{J}_\alpha$. Therefore, by the definition of Φ_P, $L \notin T_P(M)$. As a result, we have that $L \notin M$, because $M^+ = \Phi_P(M)^+ = T_P(M)$.

- L is negative.

  By definition of $\mathcal{S}_P$ and the fact that $Conflict_P(J_\alpha) = \emptyset$, we know that there exists a clause R in P with conclusion $\neg L$ such that $\neg prem(R) \cap \overline{J}_\alpha = \emptyset$. By this and the definition of Φ_P we have that $L \notin F_P(M)$, and therefore $L \notin M$. □

2.8 Finding the Well-Founded Model using $\mathcal{S}_P$

Although the well-founded model, as introduced in [vGRS91], is a stable model, and therefore can be found using the results in section 2.6, we want to give special consideration to this model, because it is one of the most interesting stable models (together with the total stable models). In this section, we show that the well-founded model of a program can be found using a special class of selection strategies, the *well-founded strategies.* First, we give a definition of the well-founded model (for a proper definition, we refer to [vGRS91]).

Definition 2.8.1 Let P be a program. The *well-founded model* of P is the smallest stable model of P (with respect to the knowledge ordering). □

This definition make use of the *knowledge ordering*. We omit its precise definition. Intuitively, for two models M and N, M is smaller or equal than N in the knowledge order, if all knowledge contained in M is also contained in N, i.e. all that is true in M is also true in N and all that is false in M is also false in N.

Now, we introduce the class of *well-founded strategies*.

Definition 2.8.2 Let P be a program. A selection strategy ρ for P is a *well-founded strategy*, if, for all J such that ρ has to select from $Choice_P(J)$ and $U_P(\overline{J})$ is non-empty, ρ selects a j-triple that contains a literal $\neg A$ such that $A \in U_P(\overline{J})$. □

Lemma 2.8.3 *Let P be a program and let M be a stable model of P. There exists a well-founded selection strategy for M.*

Proof: Let M be a stable model of P. By Lemma 2.5.6, there exist selection strategies for M. Therefore, it suffices to prove that, for a j-interpretation J such that $\overline{J} \subset M$, $Conflict_P(J)$ is empty, $Choice_P(J)$ is non-empty and $U_P(\overline{J})$ is non-empty, $U_P(\overline{J}) \cap M^-$ is non-empty. This follows from the stronger claim that, for $I \subseteq M$, $U_P(I) \subseteq M^-$. By Lemma 3.3 in [vGRS91], the operator $\mathbf{U}_P$ is monotone. We also have that $U_P(M) = \emptyset$. From these two facts we have that, for $I \subseteq M$,

$$U_P(I) \subseteq U_P(I) \cup I^- = \mathbf{U}_P(I) \subseteq \mathbf{U}_P(M) = U_P(M) \cup M^- = M^-$$

□

Lemma 2.8.4 *Let P be a program. Every well-founded selection strategy for P is a selection strategy for the well-founded model of P.*

Proof: Let M be the well-founded model of P and let ρ be a well-founded selection strategy for P. Let J be a j-interpretation such that $\overline{J} \subset M$. By Lemma 2.5.5, we know that $\overline{Conflict_P(J)} \subseteq M$. Therefore, we only have to consider the case in which we have to select from $Choice_P(J)$. There are two cases:

- Suppose that $U_P(\overline{J})$ is non-empty. Then, ρ will select a j-triple from $Choice_P(J)$ that contains a literal $\neg A$ such that $A \in U_P(\overline{J})$. Because $\overline{J} \subseteq M$, we have that $U_P(\overline{J}) \subseteq M^-$, and therefore that $A \in M^-$.

- Suppose that $U_P(\overline{J})$ is empty. Then, by Lemma 2.6.1, $\overline{J}$ is a stable model of P. But then, because $\overline{J} \subset M$, $\overline{J}$ is smaller than M in the knowledge-ordering, which is in contradiction with the fact that M is the well-founded model of P. □

Lemma 2.8.5 *Let P be a program. M is the well-founded model of P iff M is the first stable model in Γ_P^ρ, where ρ is a well-founded selection strategy for P.*

Proof: Let M be the well-founded model of P and let ρ be a well-founded selection strategy for P. By Lemma 2.8.4, ρ is a selection strategy for M. Therefore, there exists a least ordinal α, such that $\overline{J}_\alpha = M$ (for $J_\alpha \in \mathbf{P}\rho$). Moreover, the prefix of Γ_P^ρ ending at J_α is monotone increasing (in the knowledge order). Because M is the knowledge-minimal stable model of P, there does not exist an ordinal β smaller that α such that $\overline{J}_\beta$ is a stable model of P.
□

2.9 On the Complexity of $\mathcal{S}_P$

The fact that we can generate all stable models as limits of sequences of interpretations, does not mean that we are in general capable of finding them in finite time. M. Fitting has already shown in [Fit85] that the closure ordinal of his operator Φ_P could be as high as Church-Kleene ω_1, the first nonrecursive ordinal. Because our operator (in some sense) encapsulates the Fitting operator, we cannot hope to do better with our operator. It would be interesting to define classes of programs whose stable models can be generated in an "acceptable" amount of time.

The first class of programs that comes to mind, is the class of programs P whose Herbrand Base $\mathcal{B}_P$ is finite. The following result is similar to the results obtained in [Fag91] and [SZ90]. First, we have to define a class of selection strategies whose sequences are guaranteed to be stabilizing.

Definition 2.9.1 Let P be a program and let ρ be a selection strategy for P. We call ρ *fair* if, for all ordinals α and all ordinals β smaller than α, $J_\alpha = J_\beta$ implies that the selection made by ρ for J_α differs from the selection made by ρ for J_β. □

Lemma 2.9.2 *Let P be a program. If ρ is a fair strategy for P, then the sequence Γ_P^ρ is stabilizing.*

Proof: Suppose there exists a fair strategy ρ such that Γ_P^ρ is not stabilizing. Then, we have that, for all ordinals α, $J_\alpha \neq J_{\alpha+1}$. Because J_α is defined for all ordinals α, there exists at least one j-interpretation J, such that for any ordinal α, there exists an ordinal β such that $\beta > \alpha$ and $J_\beta = J$. This j-interpretation J has a set C associated with it, from which ρ makes a selection (C is one of $Culprit_P(J)$, $Conflict_P(J)$ and $Choice_P(J)$). This set C is non-empty, because otherwise we would have that $J = \mathcal{S}_P^\rho(J)$, and is countable (but possibly infinite), because $\mathcal{B}_P$ is countable. Because ρ is fair, we have that for any two j-interpretations J_α and J_β in Γ_P^ρ such that $J_\alpha = J_\beta$ and $\alpha \neq \beta$, the element selected by ρ for J_α differs from the element selected by ρ for J_β. Therefore, there exists an ordinal γ after which every element of C has been selected once for J. But we know that there exists an ordinal δ such that $\delta > \gamma$ and $J = J_\delta$. At that point, ρ cannot make a fair selection. This is in contradiction with the fact that ρ is a fair selection rule. Therefore, if ρ is fair then Γ_P^ρ is stabilizing. □

Lemma 2.9.3 *Let P be a program with a finite Herbrand base $\mathcal{B}_P$. Let ρ be a fair strategy for P. The closure ordinal of the sequence Γ_P^ρ is finite.*

Proof: First, note that by Lemma 2.9.2 Γ_P^ρ is stabilizing, and that therefore it has a closure ordinal. Because $\mathcal{B}_P$ is finite, the number of j-interpretations is finite. Moreover, for any j-interpretation J, the sets $Conflict_P(J)$, $Choice_P(J)$ and $Culprit_P(J)$ are finite. Because of this and the fact that ρ is fair, any j-interpretation J that is not the limit fixpoint of Γ_P^ρ will occur only finitely many times in Γ_P^ρ. As a result, we have that the closure ordinal of Γ_P^ρ is finite. $\square$

Note, that this result is not very surprising. If $\mathcal{B}_P$ is finite, the set of interpretations of P is finite, which means that one can simply enumerate the set of all interpretations of P and test which of them are stable models of P. Thus, any operator should be capable of finding a solution in finite time in this case.

There remains the question of what is the best method for finding stable models of programs in the case of finite Herbrand Bases: generating and testing all consistent interpretations of a program or using $\mathcal{S}_P$ with some carefully chosen family of selection strategies. We have good hope, that the second option will, in general, perform better than the first option. First of all, by inducing some order on the atoms in the Herbrand Base of a program, like Saccà and Zaniolo did with their backtracking operator in [SZ90], we can restrict ourselves to a family of 'ordered' selection strategies, in which the redundancy in partial interpretations being considered is greatly reduced (though not completely eliminated). Moreover, although in general the number of well-supported partial interpretations of a program can be greater than the number of consistent total interpretations of a program, we think that in the typical case the number of well-founded interpretations taken into consideration by $\mathcal{S}_P$ when using a family of ordered selection strategies will be much smaller.

In the remainder of this section, we formalize the idea of 'using $\mathcal{S}_P$ to find stable models' and present classes of families of strategies that reduce redundancy. First, we introduce the notion of a search-tree for a family of strategies.

Definition 2.9.4 Let P be a program and let $\mathcal{F}$ be a family of selection strategies for P. $T_\mathcal{F}$ is a tree, with j-interpretations as nodes, such that the branches of $T_\mathcal{F}$ are exactly the maximal prefixes of sequences Γ_P^ρ such that $\rho \in \mathcal{F}$ and, for any two j-interpretations J and J' in a branch, $J \neq J'$. $\square$

The idea is that –in order to find stable models– we have to traverse the tree $T_\mathcal{F}$ for some family $\mathcal{F}$ of strategies. Moreover, we think that building and traversing this tree should account for the exponential part in the costs of finding a stable model; the strategies in $\mathcal{F}$ should be relatively easy to find (i.e. we don't want to define $\mathcal{F}$ as the family of selection strategies that, for every stable model M of P, contains exactly one selection strategy for M). We now have to find some condition that allows us to conclude that the tree for some family of strategies contains stable models. The following lemma gives us such a condition.

Lemma 2.9.5 *Let P be a program and let $\mathcal{F}$ be a family of selection strategies for P. If, for some stable model M for P, $\mathcal{F}$ contains a selection strategy for M, then $T_\mathcal{F}$ has a node n containing a j-interpretation J such that $M = \overline{J}$. Moreover, if M is total, then n is a leaf.*

Proof: Suppose ρ is an element of $\mathcal{F}$ and suppose that ρ is a selection strategy for some stable model M for P. Let α be the least ordinal such that $\overline{J_\alpha} = M$ ($J_\alpha \in \Gamma_P^\rho$).

The prefix of Γ_P^ρ up to J_α increases strictly monotonically (inclusion order). Therefore this prefix is contained in a branch in $T_\mathcal{F}$. Moreover, if M is total, α is the closure ordinal of Γ_P^ρ, and therefore $J_\alpha = J_{\alpha+1}$. So, if M is total, the prefix of Γ_P^ρ up to J_α is the maximal prefix of Γ_P^ρ that does not contain twice the same j-interpretation, and therefore it coincides exactly with a branch in $T_\mathcal{F}$.

The last j-interpretation of the prefix of Γ_P^ρ, contains M. Therefore, there exists a branch in $T_\mathcal{F}$ with a node that contains M. Moreover, if M is total, there exists a branch that coincides exactly with this prefix, and therefore the leaf of this branch contains M. □

So, we have to find a family $\mathcal{F}$ of selection strategies such that $\mathcal{F}$ contains a selection strategy for every stable model in M (later on, we turn our attention to *total* stable models).

We present a number of restrictions on selection strategies, that define a class of so-called *families of <-order unfounded-set selection strategies.* Every family in this class will, for every stable model M, contain at least one selection strategy for M, but the size of the search-tree for these families (with respect to the search-tree for the family of all selection strategies) will be relatively small. We start by introducing <-ordered strategies.

Definition 2.9.6 Let P be a program and let $<$ be a total order on $\mathcal{L}_P$. We call a strategy ρ for P *<-ordered*, if, for all j-interpretations J of P such that ρ has to select from $Conflict_P(J)$, ρ selects a j-triple from $Conflict_P(J)$ containing a literal that is a <-minimal element of $\overline{Conflict_P(J)}$. □

The idea behind restricting ourselves to <-ordered strategies (for some order <) is, that we can define an equivalence relation on the selection strategies for P, in a way that every <-ordered strategy is a representative of an equivalence class.

Example 2.9.7 *Consider the program* P_5*:*

$$\begin{aligned} p &\leftarrow \\ q &\leftarrow \\ r &\leftarrow p \\ r &\leftarrow q \end{aligned}$$

We have that $Conflict_{P_5}(\emptyset)$ *consists of the j-triples* $\langle p, \emptyset, \emptyset\rangle$ *and* $\langle q, \emptyset, \emptyset\rangle$. *There exist two kinds of selection strategies for* P_5*: the ones that in a given situation select first p, then q or r and then the remaining one, and the ones that –in that given situation– select first q, then p or r and then the remaining one. But any two selection strategies of P that differ in this aspect only, are essentially* *equivalent, because they both end up with a j-interpretation containing the interpretation $\{p, q, r\}$ (note however, that the j-interpretations themselves may differ).* ∘

Lemma 2.9.8 *Let P be a program and let $<$ be a total order on $\mathcal{L}_P$. Then, for every stable model M of P, the family of $<$-ordered selection strategies contains a selection strategy for M.*

Proof: Let M be a stable model of P. By Lemma 2.5.6, there exist selection strategies for M. Therefore, it suffices to prove that, for a j-interpretation J such that $\overline{J} \subset M$ and $Conflict_P(J)$ is non-empty, $D \cap M \neq \emptyset$, where D is the set of $<$-minimal elements of $\overline{Conflict_P(J)}$. But this follows from the fact that, by definition of D and Lemma 2.5.5, $D \subseteq \overline{Conflict_P(J)} \subseteq M$. □

We can strengthen this result by combining it with the result on well-founded strategies.

Lemma 2.9.9 *Let P be a program and let $<$ be a total order on $\mathcal{L}_P$. Let $\mathcal{F}$ be the family of strategies that are both well-founded and $<$-ordered. Then, for every stable model M of P, $\mathcal{F}$ contains a selection strategy for M.*

Proof: The proof follows directly from lemma's 2.8.3 and 2.9.8, because the condition for $<$-orderedness is only relevant if an element of $Conflict_P$ is selected, while the condition for well-foundedness is only relevant if an element of $Choice_P$ is selected. □

A further strengthening is possible by using the order on $\mathcal{L}_P$ when selecting an element of U_P.

Definition 2.9.10 Let P be a program and let $<$ be a total order on $\mathcal{L}_P$. We call a strategy ρ for P an *$<$-order unfounded-set strategy*, if, for all j-interpretations J of P:

- if ρ has to select from $Conflict_P(J)$, it selects j-triple that contains a $<$-minimal literal of $\overline{Conflict_P(J)}$, and
- if ρ has to select from $Choice_P(J)$ and $U_P(\overline{J})$ is non-empty, it selects a j-triple that contains a $<$-minimal literal of $U_P(\overline{J})$. □

Lemma 2.9.11 *Let P be a program and let $<$ be a total order on $\mathcal{L}_P$. Let $\mathcal{F}$ be the family of $<$-order unfounded-set strategies. Then, for every stable model M of P, $\mathcal{F}$ contains a selection strategy for M.*

Proof: By definition, $\mathcal{F}$ is contained in the family of selection strategies that are both $<$-ordered and well-founded. Let M be a stable model of P and let J be a j-interpretation of P such that $\overline{J} \subset M$, $Conflict_P(J)$ is empty, $Choice_P(J)$ is non-empty and $U_P(\overline{J})$ is non-empty. We know that $U_P(\overline{J}) \subseteq M^-$ (see Lemma 2.8.3). But the $<$-minimal element of $U_P(\overline{J})$ is clearly an element of $U_P(\overline{J})$, and therefore an element of M^-. Therefore there exist $<$-order unfounded set strategies for M. □

We conclude this section by defining a class of families of selection strategies such that, for any family in this class and any *total* stable model M of P, the family contains a selection strategy for M. For this, we need to define a special dependency relation on the unknown atoms of an interpretation.

Definition 2.9.12 Let P be a program and let I be an interpretation for P. We define the *dependency relation* $<_{\mathcal{D}_I}$ *on* $\mathcal{B}_P - I^\pm$ as the transitive closure of the relation $\mathcal{D}_I$, which is defined as follows: $A' \mathcal{D}_I A$ iff there exists a rule R in P with conclusion A that is neither applicable nor inapplicable in I such that $A' \in prem(R)^\pm$. An element A of $\mathcal{B}_P - I^\pm$ is called $<_{\mathcal{D}_I}$*-minimal* if, for all A' such that $A' <_{\mathcal{D}_I} A$, $A <_{\mathcal{D}_I} A'$. □

Example 2.9.13 *Consider the program* P_6*:*

$$\begin{aligned} &p \leftarrow \neg p \\ &p \leftarrow q \\ &r \leftarrow s \\ &s \leftarrow r \\ &t \leftarrow s \\ &u \leftarrow \neg v \\ &v \leftarrow \neg u \end{aligned}$$

Let $I = \{\neg q\}$ *be an interpretation of* P_6*. Then we have that*

$$\mathcal{D}_I = \{\langle p,p\rangle, \langle r,s\rangle, \langle s,r\rangle, \langle s,t\rangle, \langle u,v\rangle, \langle v,u\rangle\}$$

So, $\{p,s,r,u,v\}$ *is the set of* $<_{\mathcal{D}_I}$*-minimal elements.* ○

Definition 2.9.14 Let P be a program. We call a strategy ρ for P $\mathcal{D}$*-ordered,* if, for all j-interpretations J of P such that ρ has to select from $Choice_P(J)$, ρ selects a j-triple containing a literal $\neg A$ such that A is $<_{\mathcal{D}_{\overline{J}}}$-minimal. □

Lemma 2.9.15 *Let* P *be a program and let* $<$ *be a total order on* $\mathcal{L}_P$*. Let* $\mathcal{F}$ *be the family of selection strategies that are both* $<$*-ordered and* $\mathcal{D}$*-ordered. For every total stable model of* P*,* $\mathcal{F}$ *contains a selection strategy for* M*.*

Proof: Let M be a total model of P. Because the conditions for $<$-orderedness and $\mathcal{D}$-orderedness do not interfere with eachother and because by Lemma 2.9.8 there exist $<$-ordered selection strategies for M, we only have to show that the $\mathcal{D}$-orderedness condition does not interfere with the condition for strategies for M. Let J be a j-interpretation such that $\overline{J} \subseteq M$, $Conflict_P(J)$ is empty and $Choice_P(J)$ is non-empty. Let D be the set of $<_{\mathcal{D}_{\overline{J}}}$-minimal elements of $\overline{J}$. It suffices to prove the $D \cap M^-$ is non-empty. First, note that D is non-empty because $\overline{J} \subset M$.

Suppose that $D \cap M^-$ is empty. Then, because M is total, we have that $D \subseteq M^+ \subseteq M$. Now, let A be an element of D and let R be a clause with conclusion A that is applicable in M (there has to exist at least one such clause). Because $Conflict_P(J)$ is empty, $prem(R)^{\pm} \cap I^{\pm}$ is non-empty. Moreover, because A is $<_{\mathcal{D}_J}$-minimal, $prem(R)^{\pm} \cap D$ is non-empty. Finally, because R is consistent in M and $D \subseteq M$, we know that $prem(R)^- \cap D$ is empty. So, for all clauses R with conclusion in D that are applicable in M, $D \cap prem(R)^+$ is non-empty. But then, D is an unfounded set of M, and thus

$$D \subseteq U_P(M) \subseteq \mathbf{U}_P(M) = M^-$$

which contradicts the assumption that $D \cap M^-$ is empty. $\square$

2.10 Conclusions

In this chapter, we have presented an operator that generates sequences of interpretations. We have shown that the limits of these sequences are exactly all total stable models of a normal logic program. Moreover, the set of all stable models can be identified as a subset of the interpretations generated by the operator. Furthermore, we have shown that the least fixpoint of the Fitting operator appears in all sequences generated by our operator, and that we can find the well-founded model, using a special family of selection strategies.

It would be interesting to find classes of selection strategies that can be implemented efficiently, are complete (i.e. are capable of finding all (total) stable models), and have small closure ordinals. The families of selection strategies we presented here seems to be good candidates, and it might be possible that we are capable of restricting these classes further.

Chapter 3

A Proof Procedure for Extended Logic Programs

Summary

Recently, M. Gelfond and V. Lifschitz proposed to extend normal logic programs to so-called *extended logic programs*, by adding strong negation. They propose *answer sets* as a semantics for these programs. However, this semantics uses the notion of *global consistency*. The necessity of testing for global consistency makes finding a proof for a specific query with respect to a program as hard as finding a complete answer set for that program. In this chapter, we abandon the idea of preserving global consistency and propose a modified transformation from extended logic programs to normal logic programs, based on a semantics in which only *local consistency* is preserved. We use the notion of *conservative derivability*, as defined by G. Wagner, as a proof-theoretic semantics for extended logic programs, and show that the three-valued completion semantics of a transformed program is sound and complete with respect to conservative derivability in the original extended logic program. As a result, we can use any proof procedure for normal logic programs that is sound with respect to completion semantics, to answer queries with respect to extended logic programs. We illustrate our proof procedure by using it to prove queries with respect to an extended logic program discussed earlier by Gelfond and Lifschitz.

3.1 Introduction

Extended logic programs were introduced by M. Gelfond and V. Lifschitz in [GL90], to overcome some problems in dealing with incomplete information. In this chapter, we present a proof procedure for these extended logic programs. The reason for developing this proof procedure is, that we want to be able to compute answers to queries with respect to an extended logic program, without first having to compute some intended model of that program.

The proof procedure we present is based on a transformation from extended logic programs to normal logic programs (this transformation differs from the one defined by Gelfond and Lifschitz). We have chosen a transformational approach, because it enables us to profit from work done on proof procedures for normal logic programs. The transformation we propose implements the notion of conservative derivability as introduced by G. Wagner in [Wag91]. As a result, for an extended logic program without function symbols, the three-valued completion semantics of a transformed program is sound and complete with respect to the notion of conservative derivability in the original extended logic program.

As a semantics for extended logic programs, Gelfond and Lifschitz defined the so-called *answer sets* of an extended logic program. These sets are defined in terms of the stable models of a derived normal logic program, provided the extended logic program is consistent. The proof procedure we define, is neither sound nor complete with respect to the answer set semantics. The reason for our proof procedure not being complete is, that the problem of testing whether a normal logic program has a stable model is Σ_1^1-complete (see corollary 5.12 in [MNR92]). Consequently, no effective proof procedure can be complete with respect to answer set semantics. The reason for our proof procedure not being sound with respect to answer set semantics is, that conservative reasoning is a form of *paraconsistent* reasoning, i.e. it allows us to derive meaningful answers to queries with respect to inconsistent (extended) logic programs. In contrast, the answer set semantics collapses in the case of inconsistent extended logic programs: everything becomes true.

In the next section, we give a short introduction to extended logic programs and introduce some notation used throughout the chapter. In section 3.3 we explain the notion of conservative reasoning. In section 3.4, we define the transformation of an extended logic program P to a normal logic program P_{cr}, and prove that a query Q is conservatively derivable from P if and only if a query Q' (derived from Q by means of a transformation) is a logical consequence of $comp(P_{cr})$. In section 3.6, we use SLDNF-resolution to compute answers to queries with respect to an extended logic program discussed by Gelfond and Lifschitz in [GL90]. Finally in section 3.7, we relate our transformation to the one proposed by Gelfond and Lifschitz.

3.2 Preliminaries and Notation

A *normal logic program* is a finite set of *normal clauses* of the form

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

where, for $i \in [0..n]$, A_i is an atom. Formulae of the form A or $not\ A$, where A is an atom, are called *literals*. The negation used in normal logic programs is interpreted as negation as (finite) failure: *not* A is true whenever one fails to (finitely) derive A and *not* A is false if one can derive A (finitely). However, in some cases it is useful to have a stronger notion of negation (notation: $\sim$), in which $\sim A$ is true iff $\sim A$ can be derived. This is called ***strong negation.*** For this, Gelfond and Lifschitz introduced *extended logic programs.* In extended logic programs, we use both negation as failure (*not*) and strong negation ($\sim$). So, wherever one could write an atom in a normal logic program, one can write an atom or a strongly negated atom in an extended logic program. Thus, an *extended logic program* is a finite set of *extended clauses* of the form

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

where, for $i \in [0..n]$, L_i is a *literal* (i.e. a formula of the form A or $\sim A$, where A is an atom). Formulas of the form L or *not* L, where L is a literal, are called *extended literals.* Note, that in a normal logic program, a literal is of the form A or $\mathit{not}A$, while in an extended logic program, a literal is of the form A or $\sim A$. The $\leftarrow$ in extended logic programs should not be read as classical implication. Instead, clauses in an extended logic program should be seen as inference rules.

Let us now justify our choice of symbols for strong negation and negation as (finite) failure. The symbol $\neg$ is generally used for classical negation. Moreover, in normal logic programs, negation as failure is generally denoted by either '$\neg$' or '*not*'. In [GL90], '*not*' is used for negation as failure and '$\neg$' is used for strong negation. In [Prz90a], '$\sim$' is used for negation as failure and '$\neg$' is used for strong negation. (In both [GL90] and [Prz90a] they refer to the second form of negation as classical negation.) Finally, in [Wag93] '$-$' is used for negation as failure (or *weak negation*, as it is called there), '$\sim$' is used for strong negation and '$\neg$' is used for classical negation. To get some order in this confusion of symbols for negation, we use '$\neg$' for classical negation, '$\sim$' for strong negation and '*not* ' for negation as failure. The rationale behind our choice is, that the use of '$\neg$' for classical negation is standard. Moreover, the second form of negation used in extended logic programming differs from classical negation. Therefore, one should use a different symbol, so why not follow [Wag93] and use '$\sim$'. Finally, for negation as failure, the obvious choice is that between '*not*' and '$-$'. We chose '*not*', because it seems to be more standard than '$-$'.

In this chapter, we use $A, A', A_i, \ldots$ to denote atoms, $L, L', L_i, \ldots$ to denote (extended) literals and ϕ, ψ to denote formulas. We use boldface to denote finite sequences of objects. For instance, $\mathbf{A}$ denotes a sequence $A_1, \ldots, A_k$ of atoms. We identify a sequence $\mathbf{L}$ of (extended) literals with the conjunction $L_1 \wedge \ldots \wedge L_k$. Moreover, we sometimes identify a conjunction $L_1, \ldots, L_k$ of (extended) literals with the set of (extended) literals in $\{L_1, \ldots, L_k\}$. For the sake of simplicity, we treat both negations on (extended) literals as complement operators, i.e. $L \equiv \mathit{not}\ \mathit{not}\ L$ and $L \equiv \sim\sim L$. Note, that *not* and $\sim$ are not commutative, so we do not have that $\mathit{not}\ \sim \mathit{not}\ L \equiv \sim L$.

For a normal (resp. extended) logic program P, $\mathcal{B}_P$ denotes the Herbrand Base of P and $\mathcal{L}_P$ denotes the set of normal (resp. extended) literals build from atoms in $\mathcal{B}_P$. An *interpretation* for P is a subset of $\mathcal{L}_P$ (note that interpretations can be inconsistent). The set of ground instances of clauses in P is denoted by *ground*(P).

3.3 Conservative Reasoning

In [Wag91], G. Wagner introduces the notion of *conservative reasoning* as a means to reason with inconsistent programs (he also introduces other systems to deal with inconsistent programs, but in this chapter we are only interested in conservative reasoning). The system he proposes uses only strong negation. In [Wag93], he presents a system that incorporates negation as failure (he calls it *weak negation*, and uses '$-$' to denote it), but which is more restricted in other aspects. In this section, we present a combination of these two systems.

The language consists of the logical symbols $\wedge$ (and), $\vee$ (or), $\sim$ (strong negation), *not* (weak negation) and **t** (true), predicate symbols, constants and variables. We obtain this language by adding *not* to the language in [Wag91] or $\vee$ to the language in [Wag93]. Just like in [Wag91] and [Wag93], the language does not contain function symbols. This restriction is necessary, because we define the derivability relation $\vdash$ in terms of deduction rules; the restriction ensures that the number of premises in the deduction rules for ground literals are finite. As a consequence of this restriction, not every extended logic program can be represented as a program in this language.

The definition of a program is the same as the definition of an extended logic program. As a result, every extended logic program without function symbols is a program in this system. This definition of a program is more restricted than the definition in [Wag91], where the body of a clause is an arbitrary formula. However, we are only interested in extended logic programs, and therefore do not need arbitrary formulas in bodies of clauses.

The *conservative derivability relation* $\vdash$ is defined by a natural deduction system. The idea of conservative derivability is based on the idea of mutual neutralization, i.e. $\{A, \sim A\} \not\vdash A$. Intuitively, this means that if both A and $\sim A$ can be 'proven', we discard all 'proofs' for both A and $\sim A$. As a result, we not only loose conclusions, but also gain new ones, because *not* A and *not* $\sim A$ can be derived. Informally, $P \vdash \phi$ means that the existential closure of ϕ can be proven in P without using inconsistent knowledge in P. After introducing the deduction rules, we illuminate the idea of conservative derivability by an example.

The most important rules in this system are the rules for deriving *ground extended literals*:

$$(l) \quad \frac{\begin{array}{c}\text{there exists a } L \leftarrow \phi \text{ in } \mathit{ground}(P) \text{ such that } P \vdash \phi \text{ and} \\ \text{for all } \sim L \leftarrow \phi) \text{ in } \mathit{ground}(P) : P \vdash \mathit{not}\ \phi\end{array}}{P \vdash L}$$

$$(not\ l_1) \qquad \frac{\text{for all } L \leftarrow \phi \text{ in } ground(P) : P \vdash not\ \phi}{P \vdash not\ L}$$

$$(not\ l_2) \qquad \frac{\text{there exists a } \sim L \leftarrow \phi \text{ in } ground(P) \text{ such that } P \vdash \phi}{P \vdash not\ L}$$

The deduction rule (l) combines the notion of derivability by ground clauses with the notion of mutual neutralization: $P \vdash L$ if there exists a ground rule for L whose body is conservatively derivable, provided that $\sim L$ is not conservatively derivable. The deduction rules $(not\, l_1)$ and $(not\, l_2)$ state the converse, i.e. $P \vdash not\ L$ means that L is not conservatively derivable, either because there does not exist a ground clause for L whose body is conservatively derivable $(not\ l_1)$, or by mutual neutralization $(not\ l_2)$.

Furthermore, there are rules for deriving *complex ground formulas*:

$$(not\ not\,) \qquad \frac{P \vdash \phi}{P \vdash not\ not\ \phi}$$

$$(\,\wedge\,) \quad \frac{P \vdash \phi, \psi}{P \vdash \phi\ \wedge\ \psi} \qquad\qquad (not\ \wedge\,) \quad \frac{P \vdash not\ \phi}{P \vdash not\ (\phi\ \wedge\ \psi)}$$

$$(\,\vee\,) \quad \frac{P \vdash \phi}{P \vdash \phi\ \vee\ \psi} \qquad\qquad (not\ \vee\,) \quad \frac{P \vdash not\ \phi, not\ \psi}{P \vdash not\ (\phi\ \vee\ \psi)}$$

Note that these rules only hold for *ground* formulas.

Example 3.3.1 *Consider the program P_1:*

$$p(a) \leftarrow \mathbf{t}$$
$$q(b) \leftarrow \mathbf{t}$$

It is reasonable to deduce that $P_1 \vdash p(x), q(x)$ (i.e. $\exists x\ p(x)$ and $\exists x\ q(x)$), but to deduce $P_1 \vdash p(x) \wedge q(x)$ (i.e. $\exists x\ p(x) \wedge q(x)$) by deduction rule $(\wedge)$ is clearly wrong. ∘

Then, there is a rule for deriving *complex non-ground formulas*:

$$(\exists) \qquad \frac{P \vdash \phi\theta \text{ for some substitution } \theta}{P \vdash \phi}$$

And finally, there is of course the rule to derive verum:

$$(verum)\frac{}{P \vdash \mathbf{t}}$$

Example 3.3.2 *Consider following program P_2:*

$$r \leftarrow \mathbf{t}$$
$$p \leftarrow r$$
$$\sim p \leftarrow r$$
$$q \leftarrow not\ p$$
$$\sim q \leftarrow \sim r$$

We deduce $P_2 \vdash r$ by (l) using $\vdash$ t *and $P_2 \vdash$ not $\sim r$ by (not l_1). Moreover, we have by (not l_2) (mutual neutralization) $P_2 \vdash$ not p and $P_2 \vdash$ not $\sim p$. Finally, we deduce $P_2 \vdash$ not $\sim q$ by (not l_1) and $P_2 \vdash q$ by (l).* ∘

The derivability relation defined by these deduction rules differs from both the system in [Wag91] and the system in [Wag93]. In contrast with [Wag91] and in accordance with [Wag93], we can only derive $\sim\phi$, if ϕ is an atom. This is reasonable, because we can use *not* to negate complex formulas. Extending the derivability relation to strongly negated complex formulas is beyond the scope of this chapter. With this relation, we can derive non-ground formulas. This can be done with the system in [Wag91], but not with the system in [Wag93]. We need the derivability of non-ground formulas for the soundness and completeness results in section 3.4.

3.4 The *cr* Transformation

The idea of our proof procedure is, to find out whether a query is conservatively derivable from a program. If the query is conservatively derivable, the proof procedure should answer *yes*; otherwise, it should answer *no*. We define our proof procedure in terms of a derived normal logic program P_{cr}. The three-valued completion of P_{cr} is sound and complete with respect to conservative derivability in P (for extended logic programs without function symbols). As a result, we are free to use any proof procedure for normal logic programs that is sound with respect to the three-valued completion semantics, as a proof procedure for extended logic programs.

The idea of P_{cr} is, to split the declaration of a predicate in P into a positive and a negative part, just like Gelfond and Lifschitz did when transforming an extended logic program P into a normal logic program P'. The difference is, that we then combine these positive and negative declarations of a predicate into a declaration of the original predicate, in a way that ensures consistency of the derived program (with respect to strong negation; a normal or extended logic program is inherently consistent with respect to negation as finite failure).

First, we present the transformation used by Gelfond and Lifschitz (the transformed program $\widehat{P}$ we define, is the program Gelfond and Lifschitz refer to as P').

Definition 3.4.1 Let $\mathcal{L}$ be a language.

- The language $\widehat{\mathcal{L}}$ is the same as $\mathcal{L}$, but
  - without the logical connective $\sim$, and
  - with an additional predicate symbol $\sim p$, for every predicate symbol p in L.
- For a formula ϕ in L, $\widehat{\phi}$ is the formula in $\widehat{\mathcal{L}}$ that is obtained from ϕ by interpreting every combination $\sim p$ of the logical symbol $\sim$ and a predicate

symbol p as the predicate symbol $\widehat{\sim p}$. If $\sim$ appears in ϕ other than in front of an atom, $\widehat{\phi}$ is not defined.

- For a clause c of the form $L \leftarrow \phi$, $\widehat{c}$ is the clause $\widehat{L} \leftarrow \widehat{\phi}$.
- For a program P, $\widehat{P}$ is the program $\{\widehat{c} \mid c \in P\}$. □

Note that $\widehat{\phi}$ is not always defined. However, by construction of the derivability relation, the fact that $\widehat{\phi}$ is not defined implies that $P \vdash \phi$ does not hold.

Now we are able to define our $_{cr}$ transformation.

Definition 3.4.2 Let P be an extended logic program. P_{cr} is the normal logic program such that

- for every clause $A \leftarrow \phi$ (resp. $\sim A \leftarrow \phi$) in P, P_{cr} contains the clause $A^p \leftarrow \widehat{\phi}$ (resp. $A^n \leftarrow \widehat{\phi}$), and
- for every atom A in P, P_{cr} contains the clauses $\widehat{A} \leftarrow A^p, not\ A^n$ and $\widehat{\sim A} \leftarrow A^n, not\ A^p$. □

Note that $\mathcal{B}_{\widehat{P}} \subseteq \mathcal{B}_{P_{cr}}$.

3.5 Soundness and Completeness of P_{cr}

In this section, we prove that the $_{cr}$ transformation generates a normal logic program whose semantics corresponds with the semantics of the original extended logic program. We cannot prove soundness or completeness for arbitrary extended logic programs, simply because Wagner's definition of a program does not provide for function symbols. So, the soundness and completeness theorems are restricted to extended logic programs without function symbols.

The semantics we use for normal logic programs is the one proposed by K. Kunen in [Kun87]. It is based on the *completion comp(P)* of a normal logic program P. The definition of completion can be found in Chapter 5, Definition 5.4.1 and subsequent paragraph.

Definition 3.5.1 Let P be a normal logic program and let ϕ be a sentence. Then, *ϕ is true in P in Kunen Semantics*, if ϕ is true in all three-valued models of P (written $P \models_3 \phi$).

For a proper definition of three-valued model, we refer to Section 5.2.

One should note that the idea of (three-valued) completion semantics is, that negation as finite failure in a normal logic program P is characterized by classical negation in *comp*(P). Thus, the negation used in *comp*(P) is $\neg$ instead of *not* . In the following, we keep this conversion between negation as finite failure and classical negation implicit, and consistently use *not* in the context of normal logic programs and $\neg$ in the context of three-valued completion semantics.

In the remainder of this section, we prove that $comp(P_{cr})$ is sound and complete with respect to conservative derivability in P, i.e. $comp(P_{cr}) \models_3 \exists\widehat{\phi}$ iff $P \vdash \phi$. First, we need the following lemma, which proves that the least fixpoint of the Fitting operator $\Phi_{P_{cr}}$ (see [Fit85]) is 'sound' with respect to the conservative derivability relation. The definition of this operator can be found in Chapter 2, Definition 2.7.1.

Lemma 3.5.2 *Let P be an extended logic program without function symbols, and let L be a ground extended literal in $\mathcal{L}_P$. Then, for all natural numbers n, $\widehat{L} \in \Phi^n_{P_{cr}}$ implies $P \vdash L$.*

Proof: We prove the claim by induction on n. For $n = 0$, the claim holds trivially, because $\Phi^0_{P_{cr}} = \emptyset$. Assume that, for all m less than n, $\widehat{L} \in \Phi^m_{P_{cr}}$ implies $P \vdash L$.

First, we have the following observations:

1. $A^p \in \Phi^n_{P_{cr}}$, where A^p is ground, implies that there exists a $A \leftarrow \phi$ in $ground(P)$ such that $P \vdash \phi$.

   Suppose that $A^p \in \Phi^n_{P_{cr}}$. By construction of P_{cr} and $\Phi_{P_{cr}}$, there exists a formula ϕ such that $A^p \leftarrow \widehat{\phi}$ in $ground(P_{cr})$ and $\widehat{\phi} \subseteq \Phi^m_{P_{cr}}$, for some m less than n. By induction hypothesis, for all conjuncts L of ϕ, $P \vdash L$ and therefore, by deduction rule $(\wedge)$, $P \vdash \phi$. Moreover, by construction of P_{cr}, $A \leftarrow \phi \in ground(P)$.

2. $not\ A^p \in \Phi^n_{P_{cr}}$, where A^p is ground, implies, for all $A \leftarrow \phi$ in $ground(P)$, $P \vdash not\ \phi$.

   Suppose $not\ A^p \in \Phi^n_{P_{cr}}$. By construction of P_{cr} and $\Phi_{P_{cr}}$, for every formula ϕ such that $A^p \leftarrow \widehat{\phi}$ is in $ground(P_{cr})$, $not\ \widehat{\phi} \cap \Phi^m_{P_{cr}} \neq \emptyset$, for some m less than n. By induction hypothesis, for every $A^p \leftarrow \widehat{\phi}$ in $ground(P_{cr})$ there exists a conjunct L of ϕ such that $P \vdash not\ L$, and therefore by deduction rule $(not\ \vee)$, $P \vdash not\ \phi$. Moreover, by construction of P_{cr}, $A^p \leftarrow \widehat{\phi} \in ground(P_{cr})$ iff $A \leftarrow \phi \in ground(P)$. But then, for all $A \leftarrow \phi$ in $ground(P)$, $P \vdash not\ \phi$.

3. $A^n \in \Phi^n_{P_{cr}}$, where A^n is ground, implies that there exists a $\sim A \leftarrow \phi$ in $ground(P)$ such that $P \vdash \phi$.

   The proof of this is a variant of the proof in observation 1.

4. $not\ A^n \in \Phi^n_{P_{cr}}$, where A^n is ground, implies for all $\sim A \leftarrow \phi$ in $ground(P)$ $P \vdash not\ \phi$.

   The proof of this is a variant of the proof in observation 2.

Using these observations, we can now prove the lemma. Suppose that $\widehat{L} \in \Phi^n_{P_{cr}}$. There are two cases:

- $L \equiv A$ or $L \equiv \sim A$. By construction, P_{cr} contains exactly one clause with conclusion $\widehat{L}$.

  If $L \equiv A$, this clause is of the form $A \leftarrow A^p, \mathit{not}\ A^n$. Because $A \in \Phi^n_{P_{cr}}$, $A^p, \mathit{not}\ A^n \in \Phi^n_{P_{cr}}$. By observation 1, there exists a $A \leftarrow \phi$ in P such that $P \vdash \phi$. By observation 4, for all $\sim A \leftarrow \phi$ in P, $P \vdash \mathit{not}\ \phi$. By deduction rule (l), it follows that $P \vdash A$.

  The case where $L \equiv \sim A$ is symmetric.

- $L \equiv \mathit{not}\ A$ or $L \equiv \mathit{not}\ \sim A$. By construction, P_{cr} contains exactly one clause with conclusion $\mathit{not}\ \widehat{L}$.

  If $L \equiv \mathit{not}\ A$, this clause is of the form $A \leftarrow A^p, \mathit{not}\ A^n$. Because we have that $\mathit{not}\ A \in \Phi^n_{P_{cr}}$, $\mathit{not}\ A^p \in \Phi^n_{P_{cr}}$ or $A^n \in \Phi^n_{P_{cr}}$. Therefore, by observations 2 and 3, for all $A \leftarrow \phi$ in P, $P \vdash \mathit{not}\ \phi$ or there exists a $\sim A \leftarrow \phi$ in P such that $P \vdash \phi$. Therefore, either by deduction rule $(\mathit{not}\ l_1)$ or by deduction rule $(\mathit{not}\ l_2)$, we have that $P \vdash \mathit{not}\ A$.

  The case where $L \equiv \mathit{not}\ \sim A$ is symmetric.

□

We are now able to prove soundness and completeness of the *cr* transformation.

Theorem 3.5.3 (Soundness of the *cr* transformation)
Let P be an extended logic program and let ϕ be a formula in the language of P. Then, $comp(P_{cr}) \models_3 \exists\widehat{\phi}$ implies $P \vdash \phi$.

Proof: Suppose that $comp(P_{cr}) \models_3 \exists\widehat{\phi}$ for some formula ϕ in the language of P. We prove that $P \vdash \phi$ by induction on the complexity of ϕ.

Suppose that ϕ is a ground literal. Then $\exists\widehat{\phi}$ is also ground, and therefore $\exists\widehat{\phi} \cong \widehat{\phi}$. But then, $comp(P_{cr}) \models_3 \widehat{\phi}$. By Theorem 6.3 in [Kun87], $\widehat{\phi} \in \Phi^n_{P_{cr}}$, for some finite n. Because $\widehat{\phi}$ is a ground literal, we conclude by Lemma 3.5.2 that $P \vdash \phi$.

Suppose that ϕ is a ground formula. Then $\exists\widehat{\phi}$ is also ground, and therefore $\exists\widehat{\phi} \cong \widehat{\phi}$. We prove by induction on the structure of ϕ that $P \vdash \phi$. Suppose that $\phi \equiv \neg(\psi \vee \chi)$. Because $comp(P_{cr}) \models_3 \widehat{\phi}$, it follows that $comp(P_{cr}) \models_3 \neg\widehat{\psi}$ and $comp(P_{cr}) \models_3 \neg\widehat{\chi}$. By induction, it follows that $P \vdash \mathit{not}\ \psi$ and $P \vdash \mathit{not}\ \chi$. Thus by deduction rule $(\mathit{not} \vee)$, $P \vdash \mathit{not}\ (\psi \vee \chi)$. For ϕ equivalent to $\neg\neg\psi$, $\psi \wedge \chi$, $\neg(\psi \wedge \chi)$ or $\psi \vee \chi$, the proofs are similar.

Suppose ϕ is a non-ground formula. $comp(P_{cr}) \models_3 \exists\widehat{\phi}$ implies that, for some ground instantiation θ, $comp(P_{cr}) \models_3 \widehat{\phi}\theta$. By induction, it follows that $P \vdash \phi\theta$. Thus, by deduction rule $(\exists)$, $P \vdash \phi$. □

Theorem 3.5.4 (Completeness of the *cr* transformation)
Let P be an extended logic program and let ϕ be a formula in the language of P. If $P \vdash \phi$ then $comp(P_{cr}) \models_3 \exists\widehat{\phi}$.

Proof: $P \vdash \phi$ implies that there exists a finite sequence $\phi_1, \ldots, \phi_k \equiv \phi$ of formulae in the language of P such that, for all $i \in [1..k]$, ϕ_i is the result of applying one of the deduction rules for which, for every condition of the form $P \vdash \phi'$, $\phi' \equiv \phi_j$ for some j less than i. Therefore, in order to prove that $comp(P_{cr}) \models_3 \exists\widehat{\phi}$, it is sufficient to prove for each of the deduction rules that (in $comp(P_{cr})$) the conclusion is implied by the conditions.

The only deduction rules that are less than straightforward, are (l), $(not\, l_1)$ and $(not\, l_2)$: the rules for deriving ground extended literals.

- Consider deduction rule (l).

  Suppose there exists a clause $A \leftarrow \phi$ in $ground(P)$ for which we have that $comp(P_{cr}) \models_3 \widehat{\phi}$. Then there exists a clause $A^p \leftarrow \widehat{\phi}$ in $ground(P_{cr})$. But then, $comp(P_{cr}) \models_3 A^p$. Moreover, suppose that, for all clauses $\sim A \leftarrow \phi$ in $ground(P)$, $comp(P_{cr}) \models_3 \neg\widehat{\phi}$. Then, for all $A^n \leftarrow \widehat{\phi}$ in $ground(P_{cr})$, $comp(P_{cr}) \models_3 \neg\widehat{\phi}$. Thus, by construction of $comp(P_{cr})$, we have that $comp(P_{cr}) \models_3 \neg A^n$. Because $comp(P_{cr})$ models P_{cr} and $A \leftarrow A^p, not\ A^n$ is in $ground(P_{cr})$, $comp(P_{cr}) \models_3 A$.

  The case for deriving $\sim A$ using (l) is similar.

- Consider deduction rule $(not\, l_1)$.

  Suppose for all clauses $A \leftarrow \phi$ in $ground(P)$, $comp(P_{cr}) \models_3 \neg\widehat{\phi}$. Then, by construction of P_{cr} we have that, for all clauses $A^p \leftarrow \widehat{\phi}$ in $ground(P_{cr})$, $comp(P_{cr}) \models_3 \neg\widehat{\phi}$. By construction of $comp(P_{cr})$, $comp(P_{cr}) \models_3 \neg A^p$. Because $A \leftarrow A^p, not\ A^n$ is the only clause in $ground(P_{cr})$ with conclusion A, $comp(P_{cr}) \models_3 \neg A$.

  The case for deriving $not\ \sim A$ using $(not\, l_1)$ is similar.

- Consider deduction rule $(not\, l_2)$.

  Suppose there exists a clause $\sim A \leftarrow \phi$ in $ground(P)$ for which we have that $comp(P_{cr}) \models_3 \widehat{\phi}$. Then, there exists a clause $A^n \leftarrow \widehat{\phi}$ in $ground(P_{cr})$ such that $comp(P_{cr}) \models_3 \widehat{\phi}$, and therefore $comp(P_{cr}) \models_3 A^n$. But because $A \leftarrow A^p, not\ A^n$ is the only clause in $ground(P_{cr})$ with conclusion A, we have that $comp(P_{cr}) \models_3 \neg A$.

  The case for deriving $not\ \sim A$ using $(not\, l_2)$ is similar. □

Corollary 3.5.5 *Let P be an extended logic program and let ϕ be a conjunction of extended literals.*

(i) *If θ is an SLDNF computed answer substitution for $P_{cr} \cup \{\widehat{\phi}\}$, then, for every substitution σ, $P \vdash \phi\theta\sigma$.*

(ii) *If $P_{cr} \cup \{\widehat{\phi}\}$ has a finitely failed SLDNF-tree, then, for every substitution σ, $P \vdash$ not $\phi\sigma$.*

Proof:

(i) Suppose θ is an SLDNF computed answer substitution for $\widehat{\phi}$. Then, by three-valued soundness of SLDNF-resolution, $comp(P_{cr}) \models_3 \forall\widehat{\phi}\theta$. Therefore, for all substitutions σ, $comp(P_{cr}) \models_3 \exists\widehat{\phi}\theta\sigma$. Finally, by soundness of the $_{cr}$ transformation, we have that, for every substitution σ, $P \vdash \phi\theta\sigma$.

(ii) Suppose $P_{cr} \cup \{\widehat{\phi}\}$ has a finitely failed SLDNF-tree. Then, by soundness of SLDNF-resolution with respect to three-valued completion, we have that $comp(P_{cr}) \models_3 \neg\exists\widehat{\phi}$. Therefore, for all substitutions σ, $comp(P_{cr}) \models_3 \exists\neg\widehat{\phi}\sigma$. Finally, by soundness of the $_{cr}$ transformation, we have that, for every substitution σ, $P \vdash$ *not* $\phi\sigma$. □

3.6 An Example using SLDNF-Resolution

This section is dedicated to an example of using the transformation to answer queries. For this we use the program presented by Gelfond and Lifschitz in [GL90]. Consider the following program *School*:

$$\begin{array}{l}
eligible(x) \leftarrow highGPA(x) \\
eligible(x) \leftarrow minority(x), fairGPA(x) \\
\sim eligible(x) \leftarrow \sim fairGPA(x) \\
interview(x) \leftarrow not\ eligible(x), not\ \sim eligible(x) \\
fairGPA(Ann) \leftarrow \\
\sim highGPA(Ann) \leftarrow
\end{array}$$

The normal logic program $School_{cr}$ consists of the following clauses:

$$\begin{array}{l}
eligible^{p}(x) \leftarrow highGPA(x) \\
eligible^{p}(x) \leftarrow minority(x), fairGPA(x) \\
eligible^{n}(x) \leftarrow \sim fairGPA(x) \\
interview^{p}(x) \leftarrow not\ eligible(x), not\ \sim eligible(x) \\
fairGPA^{p}(Ann) \leftarrow \\
highGPA^{n}(Ann) \leftarrow
\end{array}$$

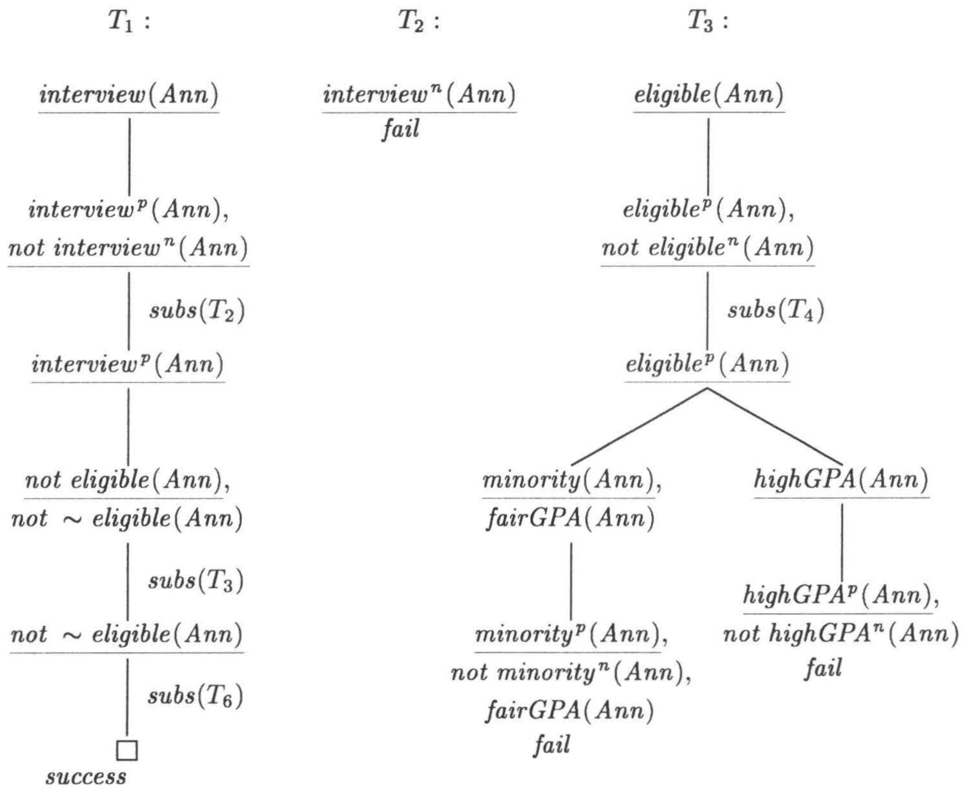


Figure 3.1: An SLDNF-tree for *interview*(*Ann*).

and

$$\begin{array}{l}
eligible(x) \leftarrow eligible^p(x), not\ eligible^n(x) \\
\sim eligible(x) \leftarrow eligible^n(x), not\ eligible^p(x) \\
fairGPA(x) \leftarrow fairGPA^p(x), not\ fairGPA^n(x) \\
\sim fairGPA(x) \leftarrow fairGPA^n(x), not\ fairGPA^p(x) \\
highGPA(x) \leftarrow highGPA^p(x), not\ highGPA^n(x) \\
\sim highGPA(x) \leftarrow highGPA^n(x), not\ highGPA^p(x) \\
interview(x) \leftarrow interview^p(x), not\ interview^n(x) \\
\sim interview(x) \leftarrow interview^n(x), not\ interview^p(x) \\
minority(x) \leftarrow minority^p(x), not\ minority^n(x) \\
\sim minority(x) \leftarrow minority^n(x), not\ minority^p(x)
\end{array}$$

Now, consider the query *interview*(*Ann*). One of the SLDNF-trees for $\{interview(Ann)\} \cup P_{cr}$ is the one given in Figures 3.1 and 3.2 (T_1 is the 'root' tree). These threes are based on the definition of SLDNF-tree given in [AD94]. In Section 4.3 one can find the definition of LDNF-tree, which is a SLDNF-tree in which the leftmost selection rule is used. Note, that in these SLDNF-trees,

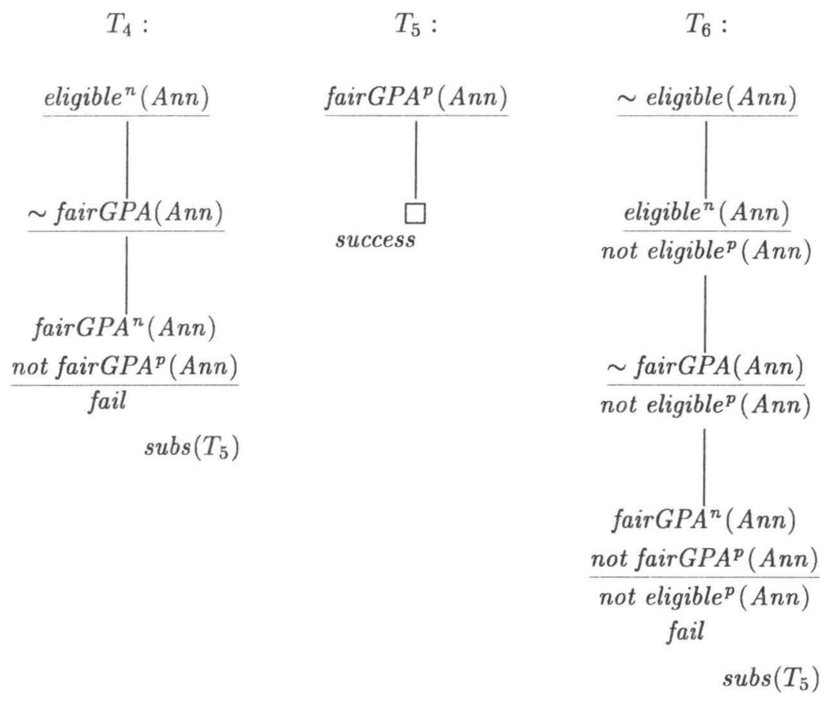


Figure 3.2: An SLDNF-tree for *interview*(*Ann*) (continued).

$subs(T_i)$ denotes a "pointer" to the subsidiary tree T_i.

As we see, we get the same answer as Gelfond and Lifschitz got with their answer set semantics. This is not very surprising. For a large class of consistent extended logic programs, completion semantics for $\widehat{P}$ and P_{cr} coincide. In the next section we go deeper into this relation between $\widehat{P}$ and P_{cr}.

3.7 On the Relation between $\widehat{P}$ and P_{cr}

If we know that an extended logic program is consistent, the most intuitive and simple translation to normal logic programs is the $\widehat{\ }$ transformation. Therefore, we would like the $_{cr}$ transformation to coincide with the $\widehat{\ }$ transformation, for extended logic programs that happen to be consistent.

First the good news: for consistent extended logic programs, the $_{cr}$ transformation is 'sound' with respect to the $\widehat{\ }$ transformation.

Theorem 3.7.1 *Let P be a consistent extended logic program and let ϕ be a formula in the language of P. Then, $comp(P_{cr}) \models_3 \widehat{\phi}$ implies $comp(\widehat{P}) \models_3 \widehat{\phi}$.*

Proof: Suppose $comp(P_{cr}) \models_3 A$. Then by construction of P_{cr} we have that $comp(P_{cr}) \models_3 A^p$ and therefore, by construction of P_{cr} and $\widehat{P}$, it follows that $comp(\widehat{P}) \models_3 A$.

Suppose $comp(P_{cr}) \models_3 \neg A$. Then, by construction of P_{cr}, we have that either $comp(P_{cr}) \models_3 \neg A^p$ or $comp(P_{cr}) \models_3 A^n$. If $comp(P_{cr}) \models_3 \neg A^p$, it follows by construction of P_{cr} and $\widehat{P}$ that $comp(\widehat{P}) \models_3 \neg A$. If $comp(P_{cr}) \models_3 A^n$, it follows by construction of P_{cr} and $\widehat{P}$ that $comp(\widehat{P}) \models_3 \sim A$, and by consistency of P that $comp(\widehat{P}) \models_3 \neg A$.

The case where $\phi \equiv \sim A$ is a variant of this proof. The case where ϕ is a complex formula is proven by induction on the structure of $\widehat{\phi}$. □

Note that this lemma holds also for two-valued completion. In fact, it seems reasonable to expect it to hold for any reasonable semantics for normal logic programs. A corollary to this lemma is that, for consistent extended logic programs, conservative derivability is sound with respect to answer-set semantics.

The converse of this lemma does not hold, as is shown in the following example.

Example 3.7.2 *Consider the extended logic program P_3:*

$$\begin{aligned} &\sim q \leftarrow not\ q \\ &q \leftarrow \end{aligned}$$

P_{3cr} is the normal logic program

$$\begin{aligned} &q^p \leftarrow \mathbf{t} \\ &q^n \leftarrow not\ q \\ &q \leftarrow q^p, not\ q^n \\ &\sim q \leftarrow q^n, not\ q^p \end{aligned}$$

For P_3 we have that comp($\widehat{P}$) $\models_3 q$. On the other hand, comp(P_{cr}) (after some simplification) is the theory

$$\begin{array}{l} q^p \leftrightarrow \mathbf{t} \\ q^n \leftrightarrow \neg q \\ q \leftrightarrow \neg q^n \\ \sim q \leftrightarrow \mathbf{f} \end{array}$$

and therefore comp(P_{cr}) $\not\models_3 q$.

Clearly, the behaviour of $\widehat{P}$ is more intuitive, and we would like P_{cr} to mimic it. ∘

This somewhat counterintuitive behaviour with respect to consistent programs also arises with the conservative derivability relation given in this chapter; we can derive neither $P_3 \vdash q$ nor $P_3 \vdash$ *not* q. The problem is, that in the conservative derivability relation as defined in this chapter (as well as in the relations defined by G. Wagner in [Wag91] and [Wag93]), *not* is defined as negation as *finite* failure. Because, in P_3, $\sim q$ does not fail finitely (there is a cyclic dependency between q and $\sim q$), in this system *not* $\sim q$ *should* not be derivable. A solution to this problem could be, to define a conservative derivability relation in which *not* stands for negation as (possibly infinite) failure. In such a case, we would get $P_3 \vdash q$ and $P_3 \vdash$ *not* $\sim q$. We are quite confident that such a modified system for conservative reasoning can be given, and that for such a system and for consistent extended logic programs P, we can prove soundness and completeness of conservative derivability with respect to three-valued completion of $\widehat{P}$.

With respect to such a modified conservative derivability relation, the $_{cr}$ transformation would no longer be complete. However, we can refine the transformation by omitting the consistency check for those predicates for which consistency can be proven.

Example 3.7.3 *Consider program P_3. It is clear that the definition of q is consistent. Therefore, a consistency check on q is superfluous. So, we refine P_{3cr} to*

$$\begin{array}{l} q^p \leftarrow \mathbf{t} \\ q^n \leftarrow \mathit{not}\ q \\ q \leftarrow q^p \\ \sim q \leftarrow q^n \end{array}$$

Clearly, q is a consequence of the completion of this program. ∘

So, we could improve the behaviour of the transformed program by analyzing the extended logic program and removing superfluous consistency checks in the transformed program.

As a final remark on this problem, we would like to stress that we do not advocate the use of the $_{cr}$ transformation for programs that are *known* to be consistent. Instead, we are concerned with extended logic programs for which it is not possible or practical to prove consistency beforehand.

Apart from a mismatch between the two translations with respect to three-valued completion semantics, there is also a problem related to floundering SLDNF-resolution.

Example 3.7.4 *Consider the extended logic program P_4:*

$$q(x) \leftarrow$$

P_{4cr} is the normal logic program

$$\begin{array}{l} q^p(x) \leftarrow \\ q(x) \leftarrow q^p(x), \mathit{not}\ q^n(x) \\ \sim q(x) \leftarrow q^n(x), \mathit{not}\ q^p(x) \end{array}$$

Now, consider the query $q(x)$. For P_4, this is a very simple query, which simply should be answered by yes. *But SLDNF-resolution on P_{4cr} flounders.* ∘

Note that, although in this example P_4 is a normal logic program, the problem also occurs in extended logic programs that are not normal logic programs.

A solution to this problem is to use a form of constructive negation, instead of SLDNF-resolution. For instance, W. Drabent presented SLDFA-resolution, which uses a form of constructive negation, in [Dra95] and proved that this proof procedure is sound and complete with respect to three-valued completion semantics. So, we can use the program transformation together with SLDFA-resolution as a sound and complete proof procedure for extended logic programs.

3.8 Conclusions

In this chapter we presented a transformation from extended logic programs to normal logic programs. For this transformation we have proven that, for extended logic programs without function symbols, the three-valued completion semantics of a transformed program is sound and complete with respect to conservative derivability in the original extended logic program. As a result, we can use arbitrary proof procedures for normal logic programs, as long as they are sound with respect to three-valued completion semantics. For instance, using the transformation together with SLDNF-resolution, we get a proof procedure for extended logic programs that is sound with respect to conservative derivability and using SLDFA-resolution we get a proof procedure which is sound and complete with respect to conservative derivability.

The advantage of using a transformation from extended logic programs to normal logic programs, is that it gives us access to all results concerning proof procedures for normal logic programs. For instance, we do not need to redo work on termination of queries.

The soundness and completeness result are restricted to extended logic programs without function symbols. The reason for this is, that the notion of conservative derivability is only defined for programs without function symbols. We believe that the notion of conservative derivability can be extended to programs with function symbols, and that with such an extended definition, we can generalize the soundness and completeness result to extended logic programs with function symbols.

Aside from extending the conservative derivability relation to programs with function symbols, it might be interesting to solve the second problem mentioned in section 3.7, i.e. define a notion conservative reasoning in which *not* stands for negation as (possibly infinite) failure. Once we have such a system, we could use consistency analysis on the extended logic program to optimize the transformed program by omitting superfluous consistency checks, without losing soundness of the optimized normal program with respect to conservative derivability on the original extended logic program.

Chapter 4

Comparing Negation in Logic Programming and in Prolog

Summary

We compare here two uses of negation – in logic programming and in Prolog. Because in Prolog negation is defined by means of meta-programming facilities and the cut operator, this requires a careful reexamination of the assumptions about the underlying syntax and a precise definition of the computational processes involved.

After taking care of these matters we establish a formal result showing an equivalence in appropriate sense between these two uses of negation. This result allows us to argue about correctness of various known Prolog programs which use negation, by reasoning about the corresponding normal logic programs.

4.1 Introduction

During the last 15 years, a lot of attention was devoted to the study of negation in logic programming. No less than seven survey articles on this subject were published. Just to mention two most recent ones: Dix [Dix93] and Apt and Bol [AB94].

The main reason for this interest is that in the logic programming setting negative literals can be used to model non-monotonic reasoning. The computation process of logic programming provides then a readily available computational interpretation. This is not the case with other approaches to non-monotonic reasoning. This computation process is called SLDNF-resolution and was proposed by Clark [Cla78]. Negation is interpreted in it using the "negation as finite failure" rule. Intuitively, this rule works as follows: for a ground atom A,

$$\neg A \text{ succeeds iff } A \text{ finitely fails,}$$
$$\neg A \text{ finitely fails iff } A \text{ succeeds,}$$

where "finitely fails" means that the corresponding evaluation tree is finite and all its leaves are marked as failed.

However, SLDNF-resolution is not a practical way of computing and usually one resorts to Prolog when seeking for a computational interpretation. But in Prolog negation is implemented in a different way, namely by the predicate (or synonymously relation symbol) **neg** defined internally by the following two clauses:

$$\mathbf{neg}(\mathtt{X}) \leftarrow \mathtt{X}, !, \mathtt{fail}. \qquad (4.1)$$
$$\mathbf{neg}(\mathtt{X}) \leftarrow . \qquad (4.2)$$

where "!" is the cut operator and `fail` is a Prolog built-in with the empty definition.

The intuition behind this definition is perhaps best revealed by first introducing the **if_then_else** predicate defined as follows:

$$\mathtt{if\_}\mathbf{then\_else}(\mathtt{P}, \mathtt{Q}, \mathtt{R}) \leftarrow \mathtt{P}, !, \mathtt{Q}.$$
$$\mathtt{if\_}\mathbf{then\_else}(\mathtt{P}, \mathtt{Q}, \mathtt{R}) \leftarrow \mathtt{R}.$$

if_then_else is intended to model within Prolog the customary **if** P **then** Q **else** R construct of imperative programming languages. Then **neg** can be equivalently defined by

$$\mathbf{neg}(\mathtt{X}) \leftarrow \mathtt{if\_}\mathbf{then\_else}(\mathtt{X}, \mathtt{fail}, \mathtt{true}).$$

where *true* is a predicate that immediately succeeds. So intuitively, **neg**(X) can be interpreted as "if X succeeds then fail else succeed".

It is usually tacitly assumed that logic programming and Prolog ways of dealing with negation are "equivalent", in the sense that SLDNF-resolution combined with the leftmost selection rule (henceforth called LDNF-resolution) properly reflects Prolog's way of handling negation. Upon closer scrutiny this assumption is far from being obvious. The above definition of the **neg** predicate and its use in programs calls upon a number of features which are present in Prolog, but absent in logic programming, and for which a formal treatment is lacking. These are:

- the use of *meta-variables*, that is variables which occur in an atom position, like X in the first clause,
- the use of *meta-programming* facilities that arise when applying this definition of **neg**, so in constructs of the form **neg**(A) where A is an atom, or a query in general.

Additionally, two better understood, though not necessarily simpler to handle, features of Prolog need to be taken care of, namely:

- the ordering of the program clauses,
- the use of the cut operator "!".

The aim of this chapter is to relate precisely these two uses of negation: in logic programming and in Prolog. To do this we appropriately tune the definition of the SLDNF-resolution given in Apt and Doets [AD94] to our present needs and formally define "Prolog trees" in the presence of the cut operator. Then we prove some results that show an appropriate equivalence between these two definitions of negation.

The outcome of this study is that we can now interpret various results about correctness of normal logic programs executed by means of the LDNF-resolution (see e.g. Apt [Apt94]) as correctness results about the corresponding Prolog programs that use negation.

4.2 Syntactic Matters

4.2.1 General Logic Programs

To relate normal logic programs to Prolog programs we have to be precise about the syntax. Fix a first-order language $\mathcal{L}$. To make this comparison possible we assume that

- a normal program is a *sequence* and not a *set* of normal clauses,
- the predicates !, *neg* and *fail* are not present in the language $\mathcal{L}$.

A *normal clause* is defined in the usual way (see e.g. Lloyd [Llo87]), so as a construct of the form $A \leftarrow L_1, \ldots, L_n$, where A is an atom and $L_1, \ldots, L_n$ are literals, i.e. atoms or their negations, all in the language $\mathcal{L}$. And a *query* is a finite sequence of literals. In the context of logic programming the negation connective is written as "$\neg$".

4.2.2 Prolog Programs

Prolog programs here considered are intended to be the programs that allow us to model the negation by means of the predicate **neg** defined by the clauses (4.1) and (4.2). However, the syntax of clause (4.1) creates a number of problems, even if we ignore the cut operator "!".

First of all, the use of the meta-variable `X` in clause (4.1) violates the syntax of the first-order logic. This use of `X` in the resolution process leads to further complications. Take an n-ary function symbol $\mathbf{p}$ in the language $\mathcal{L}$ and let $\mathbf{s}_1, \ldots, \mathbf{s}_n$ be some terms. Consider now the query $\mathbf{neg(p(s))}$ (note that we use boldface to denote finite sequences of elements, i.e. $\mathbf{s}$ is a finite sequence

$s_1, \ldots, s_n$ of terms). During the Prolog computation process it resolves using the clause (4.1) to the query **p(s)**, !, `fail`. Now in the first query **p** occurs in a position of a function symbol, whereas in the second one **p** occurs in a position of a relation symbol. So every function symbol needs also to be accepted as a relation symbol.

Also conversely: take an n-ary relation symbol p with some terms $s_1, \ldots, s_n$, and consider the normal clause $p(\mathbf{s}) \leftarrow \neg p(\mathbf{s})$. Its desired translation into a Prolog clause is **p(s)** $\leftarrow$ **neg(p(s))**. In the head of the latter clause **p** occurs in a position of a relation symbol, whereas in its body in the position of a function symbol.

As in both cases **p** was arbitrarily chosen, we conclude that to render the resolution process meaningful we need to accept that the classes of function symbols and of relation symbols in the underlying language coincide.

This is clearly in violation with the (usually tacit) assumption that in the first-order language, say $\mathcal{L}$, fixed above, the classes F_m and R_n of, respectively, its function symbols of arity m and its relation symbols of arity n are pairwise disjoint for $m, n \geq 0$. In short, the use of the clause (4.1) cannot be properly accounted for by just referring to the first-order logic.

A simple solution to the above mentioned two problems is to modify the syntax of the language $\mathcal{L}$ by allowing

- *meta-variables*, so variables that can occur in atoms positions, both in the queries and in the clause bodies,
- *ambivalent syntax*, so – in this case – by assuming that the classes of function and relation symbols coincide.

The latter can be achieved by extending $\mathcal{L}$ to a language in which for each $m \geq 0$ $F_m \cup R_m$ are the classes of both its function symbols and relation symbols. Thus in this language terms and atoms coincide.

Additionally, we assume that

- the predicates !, `neg` and `fail` are present in the underlying language,
- ! is a built-in 0-ary predicate (with a meaning to be explained later), and no clause uses it in its head,
- `neg` is a built-in predicate defined by the clauses (4.1) and (4.2), so no other clause uses it in its head,
- `fail` is a built-in 0-ary predicate with the empty definition, so no clause uses it in its head.

The last two assumptions ensure that `neg` and `fail` are indeed defined internally in the desired way. For the purposes of syntax the cut operator "!" is viewed here as a 0-ary predicate with the empty definition. This might suggest that its meaning coincides with that of `fail`. However, this is not the case. Its real, operational, "meaning" will be defined in Section 4.4 by means external to the resolution process.

So in the resulting language, apart of the customary atoms, also !, `fail` and meta-variables are admitted as atoms (henceforth called *special atoms*).

Now, a *Prolog program* is defined as a sequence of Prolog clauses preceded by the clauses (4.1) and (4.2). In turn a *Prolog clause* is a construct of the form $A \leftarrow B_1, \ldots, B_n$, where $A, B_1, \ldots, B_n$ are atoms in the language $\mathcal{L}$, and A is not a special atom. And a *Prolog query* is a finite sequence of atoms. For brevity, in the examples of Prolog programs, we drop the listing of the clauses (4.1) and (4.2).

Note that at this stage we use two notions of an atom – one within the language $\mathcal{L}$ and another in its ambivalent extension just defined. From the context it will be always clear to which of these two languages we refer.

4.2.3 Restricted Prolog Programs

The translation of a normal program to a Prolog program is now straightforward and as expected: we just replace everywhere a logic programming literal $\neg A$ by Prolog's atom **neg**(A) and prefix the resulting program with the clauses (4.1) and (4.2). In short, the logic programming negation connective "$\neg$" is traded for the built-in predicate **neg**. Similarly, a normal query is translated to a Prolog query by replacing everywhere $\neg A$ by **neg**(A).

This translation process maps every normal program (resp. normal query) onto a Prolog program. However, not every Prolog program (resp. Prolog query) is the result of translating a normal program (resp. normal query). Indeed, in general the cut operator "!" can be used in any Prolog clause, not only (4.1).

Let us now characterize the Prolog programs (resp. Prolog queries) which are the result of the above translation of normal programs (resp. normal queries). We call them *restricted Prolog programs* (resp. *restricted Prolog queries*). To this we translate "back" every Prolog program (resp. Prolog query) onto a normal program (resp. normal query) by replacing everywhere **neg**(A) by $\neg A$, and omitting the clauses (4.1) and (4.2) that define the **neg** predicate. Then a Prolog program (resp. Prolog query) is restricted if the outcome of this reverse translation is a syntactically legal normal program (resp. normal query). For example the Prolog query **neg(q)**, **q** is restricted because its reverse translation is $\neg q, q$, whereas neither **neg(q(neg(a)))** nor **p(q)**, **q** is restricted because their respective reverse translations violate the syntactic assumptions concerning normal programs.

Of course, it is possible to define the class of restricted Prolog programs and queries directly, though the resulting definition is rather tedious.

We now define a *resolvent* of a Prolog query as follows.

Definition 4.2.1 Consider a Prolog query $A, \mathbf{M}$ and a Prolog clause c. Let $H \leftarrow \mathbf{L}$ be a variant of c variable disjoint with $A, \mathbf{M}$ and let θ be an mgu of A and H. Then $(\mathbf{L}, \mathbf{M})\theta$ is called a *resolvent* of $A, \mathbf{M}$ and c *with an mgu* θ. □

The only unusual feature in the present setting is, that now the mgu's also bind the meta-variables. Also, note that the selected literal is always the leftmost literal.

It is worthwhile to mention that a resolvent of a restricted Prolog query with respect to a restricted Prolog program is not necessarily a restricted Prolog query. This is due to the use of clause (4.1), which introduces a cut atom. Thus, the Prolog queries generated in a computation of a restricted Prolog query are not necessarily restricted Prolog queries. However, the Prolog queries so generated do have one important property: they do not contain meta-variables.

Definition 4.2.2

- An atom A is called *unsafe* if one of the following holds:
  - A is a (meta) variable,
  - A is **neg**(`X`) where `X` is a variable,
  - A is **neg**(**neg**(`s`)) where `s` is a term.
- A Prolog query is called *meta-safe* if none of its atoms is unsafe. □

For example, `X`, `p(X)` is not meta-safe because its leftmost atom is a meta-variable, **neg**(`X`) is not meta-safe because the argument of **neg** is a meta-variable, and **neg**(**neg**(`p(X)`)) is not meta-safe because the argument of the outermost **neg** predicate is itself a **neg** predicate.

Note that restricted Prolog queries and bodies of the restricted Prolog clauses are meta-safe.

Lemma 4.2.3 *Let Q be a meta-safe Prolog query and P a restricted Prolog program. Then all resolvents of Q are meta-safe.*

Proof: Let Q be of the form $A, \mathbf{L}$, and let $(\mathbf{M}, \mathbf{L})\theta$ be a resolvent of Q, with an input clause c and mgu θ. As Q is meta-safe, we know that $\mathbf{L}\theta$ is meta-safe. We prove that $\mathbf{M}\theta$ is meta-safe as well. Three cases arise.

Case 1 : c is clause (4.1).

Then $\mathbf{M}\theta$ is of the form $B, !,$ `fail`, where A is of the form **neg**(B). But Q is meta-safe, so B is neither a meta-variable nor of the form **neg**(B'). So $\mathbf{M}\theta$ is meta-safe.

Case 2 : c is clause (4.2).

Then $\mathbf{M}\theta$ is the empty query, so obviously meta-safe.

Case 3 : c is different from clauses (4.1) and (4.2).

Then the body of c is meta-safe, and consequently so is $\mathbf{M}\theta$.

This proves that $(\mathbf{M}, \mathbf{L})\theta$ is meta-safe. □

Corollary 4.2.4 *All Prolog queries generated in a computation of a restricted Prolog query and a restricted Prolog program are meta-safe.* □

In Prolog, if the selected atom is a meta-variable, an *error* arises. The above result thus shows that no errors arise in Prolog computations for queries and programs that are obtained by a translation of a normal query and a normal program.

4.3 Computing with General Logic Programs

As the next step we define the LDNF-resolution that allows us to compute with normal logic programs. The definition of LDNF-resolution given here is derived in a straightforward way from that of the SLDNF-resolution given in Apt and Doets [AD94]. Apart of the fact that we view in this chapter a normal program as a finite sequence and not as a finite set of normal clauses, the differences are that:

- the leftmost selection rule is used,
- *floundering*, so –in this context– an abnormal termination due to selection of a non-ground literal is ignored.

In this way we bring the procedural interpretation of normal programs closer to that of the corresponding Prolog programs and make the subsequent comparison possible. Recall from Clark [Cla78] and Lloyd [Llo87] that floundering is a problem that arises only when dealing with the semantic aspects of the SLDNF-resolution, which are irrelevant here.

Before giving the definition of LDNF-resolution, we recall the definitions of *resolvent* and *pseudo-derivation*.

Definition 4.3.1 Consider a non-empty query $L, \mathbf{M}$ and a normal clause c.

- Suppose L is a positive literal.

  Let $H \leftarrow \mathbf{L}$ be a variant of c variable disjoint with $L, \mathbf{M}$ and let θ be an mgu of L and H. Then $(\mathbf{L}, \mathbf{M})\theta$ is called a *resolvent* of $L, \mathbf{M}$ and c *with respect to* L, *with an mgu* θ.

  We write then $L, \mathbf{M} \overset{\theta}{\underset{c}{\Longrightarrow}} (\mathbf{L}, \mathbf{M})\theta$, and call it a *positive derivation step*. We call $H \leftarrow \mathbf{L}$ the *input clause* of the derivation step.

- Suppose L is a negative literal. Then $\mathbf{M}$ is called a *resolvent* of $L, \mathbf{M}$ with the identity substitution ϵ *with respect to* L.

  We write then $L, \mathbf{M} \overset{\epsilon}{\underset{\emptyset}{\Longrightarrow}} \mathbf{M}$, and call it a *negative derivation step*.

- A normal clause c is called *applicable* to an atom if it has a variant the head of which unifies with the atom. □

Fix, until the end of this section, a normal program P.

Definition 4.3.2 A (finite or infinite) sequence

$$Q_0 \overset{\theta_1}{\underset{c_1}{\Longrightarrow}} Q_1 \cdots Q_n \overset{\theta_{n+1}}{\underset{c_{n+1}}{\Longrightarrow}} Q_{n+1} \cdots$$

of derivation steps is called a ***pseudo derivation of*** $P \cup \{Q_0\}$ if

- $Q_0, \ldots, Q_n, \ldots$ are normal queries,
- $\theta_1, \ldots, \theta_n, \ldots$ are substitutions,
- $c_1, \ldots, c_n, \ldots$ are normal clauses of P, or $\emptyset$,

and for every step involving selection of a positive literal the following condition holds:
Standardization apart: the input clause employed is variable disjoint from the initial normal query Q_0 and from the substitutions and input clauses used at earlier steps. □

Intuitively, an LDNF-derivation is a pseudo derivation in which the deletion of every negative literal is justified by means of a subsidiary (finitely failed LDNF-) tree. This brings us to consider *forests.*

Definition 4.3.3 A *forest* is a system $\mathcal{F} = (\mathcal{F}, T, subs)$ where

- $\mathcal{F}$ is a set of trees,
- T is an element of $\mathcal{F}$ called the *main tree*, and
- *subs* is a function assigning to some nodes of trees in $\mathcal{F}$ a ("subsidiary") tree from $\mathcal{F}$.

By a ***path*** in $\mathcal{F}$ we mean a sequence of nodes $N_0, \ldots, N_i, \ldots$ such that for all i, N_{i+1} is either an immediate descendant of N_i in some tree in $\mathcal{F}$, or the root of the tree $subs(N_i)$. The ***depth*** of $\mathcal{F}$ is the length of the longest path in $\mathcal{F}$. □

Thus a forest is a special directed graph with two types of edges – the "usual" ones stemming from the tree structures, and the ones connecting a node with the root of a subsidiary tree. An LDNF-tree is a special type of forest, built as a limit of certain finite forests: *pre-LDNF trees.*

Definition 4.3.4 A *pre-LDNF-tree* (relative to P) is a forest whose nodes are queries. Leaves can be unmarked, or can be marked as either *success* or *failure.* The class of pre-LDNF-trees is defined inductively:

- For every normal query Q, the forest consisting of the main tree which has the single unmarked node Q is a pre-LDNF-tree (an *initial* pre-LDNF-tree),
- If $\mathcal{T}$ is a pre-LDNF-tree, then any *extension* of $\mathcal{T}$ is a pre-LDNF-tree.

Before defining the notion of an *extension* of a pre-LDNF-tree, we need to define the notion of *successful* and *finitely failed* trees: for $T \in \mathcal{T}$,

- T is called *successful*, if one of its leaves is marked as *success*, and
- T is called *finitely failed*, if it is finite and all its leaves are marked as *failure*.

Now, an *extension* of a pre-LDNF-tree $\mathcal{T}$ is defined by performing the following actions for every non-empty normal query Q (with leftmost literal L) which is an unmarked leaf in some tree $T \in \mathcal{T}$:

- Suppose that L is a positive literal.
  - If Q has no resolvents with respect to L and a clause from P:
    Mark Q as *failure*.
  - If Q has such resolvents:
    For every clause c from P which is applicable to L, choose one resolvent Q' of Q with respect to L and c, with an mgu θ, and add this as an immediate descendant of Q in T. Choose the input clauses in such a way that all branches of T remain pseudo derivations.
- Suppose that L is a negative literal, say $\neg A$.
  - If $subs(Q)$ is undefined:
    Add a new tree T', consisting of the single node A, to $\mathcal{T}$, and let $subs(Q) = T'$.
  - If $subs(Q)$ is defined and successful:
    Mark Q as *failure*.
  - If $subs(Q)$ is defined and finitely failed:
    Add the resolvent $Q - \{L\}$ of Q as the only immediate descendant of Q in T.

Additionally, all empty queries are marked as *success*. □

Note that, if no tree in $\mathcal{T}$ has unmarked leaves, then trivially $\mathcal{T}$ is an extension of itself, and the extension process becomes stationary.

Next, we define LDNF-trees as the limit of sequences of pre-LDNF-trees. Every pre-LDNF-tree is a tree with two types of edges between possibly marked nodes, so the concepts of *inclusion* between such trees and of *limit* of a growing sequence of such trees have a clear meaning.

Definition 4.3.5

- An *LDNF-tree* is a limit of a sequence $\mathcal{T}_0, \ldots, \mathcal{T}_\alpha, \ldots i$ such that $\mathcal{T}_0$ is an initial pre-LDNF-tree, and for all i $\mathcal{T}_{i+1}$ is an extension of $\mathcal{T}_i$.
- An *LDNF-tree for Q* is an LDNF-tree in which Q is the root of the main tree.

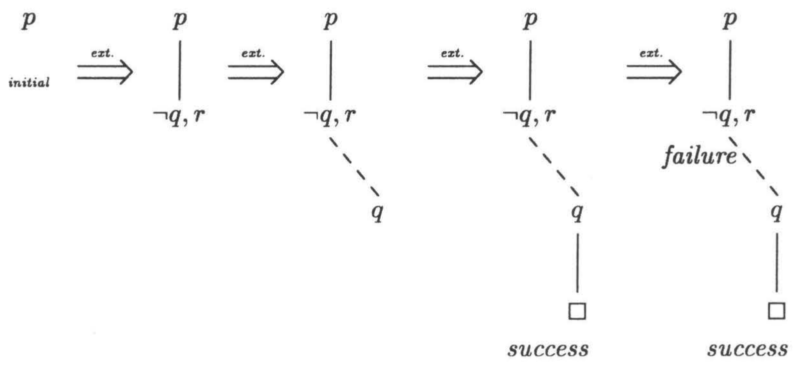


Figure 4.1: Step-by-step construction of an LDNF-tree for the query p with respect to the normal program $p \leftarrow \neg q, r \quad q \leftarrow$.

- A (pre-)LDNF-tree is called *successful* (resp. *finitely failed*) if the main tree is successful (resp. finitely failed).
- An LDNF-tree is called *finite* if no infinite path exists in it (cf. Definition 4.3.3). □

In Figure 4.1, we show how the notions of initial pre-LDNF-trees and extensions of pre-LDNF-trees are used to construct an LDNF-tree.

Finally, we recall the notion of a computed answer substitution.

Definition 4.3.6 Consider a branch in the main tree of a (pre-)LDNF-tree for Q which ends with the empty query. Let $\alpha_1, \dots, \alpha_n$ be the consecutive substitutions along this branch.

Then the restriction $(\alpha_1 \cdots \alpha_n)|Q$ of the composition $\alpha_1 \cdots \alpha_n$ to the variables of Q is called a ***computed answer substitution*** (*c.a.s.* for short) of Q. □

4.4 Computing with Prolog Programs

In this section, we model the computation process used in Prolog to find answers to queries by defining the notion of *P-resolution.* To this end we proceed in two steps.

First, we restrict LDNF-resolution to logic programs, so normal logic programs without negation, by simply disregarding the selection of a negative literal. We call the resulting computation process *LD-resolution.*

Then, we extend the LD-resolution to Prolog programs by allowing the choice of a meta-variable or of a cut atom as a selected atom. In the first case an error is reported, and in the second case the computation tree constructed so far is appropriately pruned.

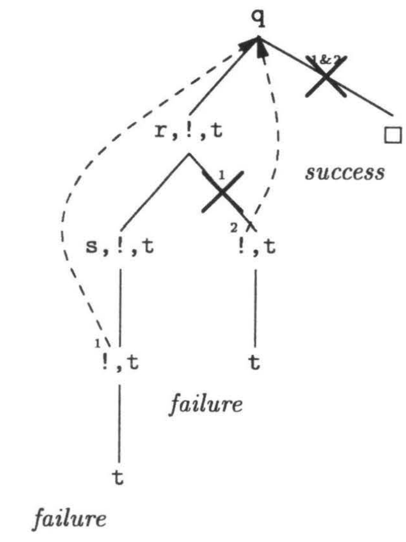


Figure 4.2: A computation tree for the query q

To better understand the issues involved in defining the effect of the cut operator, let us consider the definition of a predicate p:

$$
\begin{array}{l}
\mathtt{p}(\mathbf{s}_1) \leftarrow \mathbf{L}_1. \\
\dots \\
\mathtt{p}(\mathbf{s}_i) \leftarrow \mathbf{M}, !, \mathbf{N}. \\
\dots \\
\mathtt{p}(\mathbf{s}_k) \leftarrow \mathbf{L}_k.
\end{array}
$$

Here, the i-th clause contains a cut atom (there could be others, either in the same clause, or in other clauses). Now, suppose that during the computation of a query, some atom $\mathtt{p}(t)$ is resolved using (a variant of) the i-th clause, and that later on, the cut atom thus introduced becomes the leftmost atom. Then, according to the customary definition of the cut operator "!", once the indicated occurrence of ! is selected:

1. all other ways of resolving **M** are discarded, and
2. all derivations using (variants of) the i + 1-th to k-th clause for p are discarded.

Note that this operational definition of the behaviour of the cut operator depends on the leftmost selection rule, and on viewing a program as a sequence of clauses instead of a set of clauses.

To model this operational behaviour of the cut operator in P-resolution, we have to define it in terms of a pruning operator on LD-trees, but first, let us give an example of the behaviour of the cut operator. Consider the following Prolog program:

$$
\begin{array}{l}
\mathtt{q} \leftarrow \mathtt{r}, !, \mathtt{t}. \\
\mathtt{q} \leftarrow . \\
\mathtt{r} \leftarrow \mathtt{s}. \\
\mathtt{r} \leftarrow . \\
\mathtt{s} \leftarrow .
\end{array}
$$

In Figure 4.2, an LD-tree for the query q is shown. In this tree, there are two nodes with a cut atom as the leftmost atom. Both of these cut atoms are introduced by resolving q in the root node of the tree. We say that their *origin* is the root node. In the figure, we use dashed arrows to point from a selected cut atom to its origin. The two cut atoms that appear as leftmost atoms are marked as 1 and 2 respectively. Now, consider the cut atom marked as 1. Execution of this cut atom results in pruning: the middle branch has to be pruned according to rule 1, and the rightmost branch has to be pruned following rule 2. Execution of the cut atom marked as 2 also leads to a pruning of the rightmost branch (using rule 1 for the cut operator). In the figure, the pruned branches are marked with a cross. The label on the cross refers to the cut atoms that where responsible for the pruning of that branch.

Now, we can restate the behaviour of the cut operator as a pruning operator on LD-trees as follows:

> Consider an LD-tree T. Let Q be a node in T with a cut atom as the selected atom and let Q' be the origin of this cut atom (i.e. the node that introduced this cut atom). Then, execution of this cut atom results in pruning all branches that are to the right of Q, contain Q', and do not contain Q.

In the tree of Figure 4.2, the order in which selected cut atoms where processed, was not important. However, in general, the order *is* important. Consider the LD-tree for p in Figure 4.3, which is based on the Prolog program used in Figure 4.2, together with two additional clauses for p.

$$
\begin{array}{l}
\mathtt{p} \leftarrow \mathtt{q}, !, \mathtt{t}. \\
\mathtt{p} \leftarrow .
\end{array}
$$

Here, there are three nodes with a cut atom as leftmost atom, marked as 1, 2 and 3, respectively. Suppose we process them from right to left. First, the cut atom marked as 3 prunes the rightmost branch. Then, the cut atoms marked as 2 and 1 prune the third and the second branch from the left, respectively. The resulting tree consists of the leftmost branch only. On the other hand, when processed from left to right, the cut atom marked as 1 prunes the middle two branches. As a result, the cut atoms marked as 2 and 3 disappear, which prevents the rightmost branch from being pruned. Thus, the resulting tree consists of the leftmost branch *and* the rightmost branch.

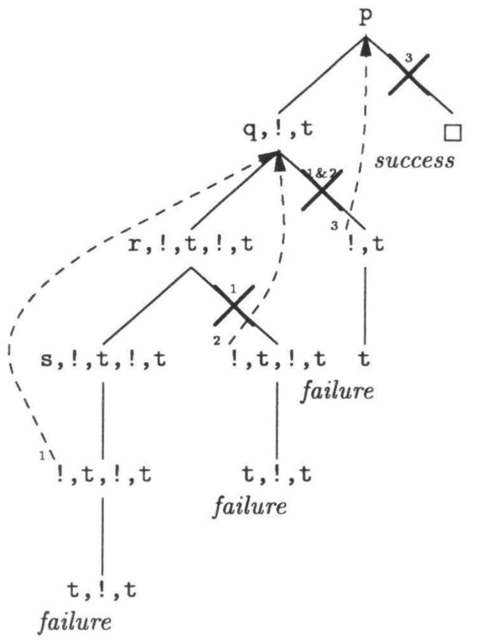


Figure 4.3: A computation tree for the query p

In Prolog, answers are computed using a left to right depth-first strategy. In particular, Prolog processes the cut atoms in the tree from left to right. On the other hand, LD-resolution is defined in a breadth-first manner: the process of extending a pre-tree consists of extending all unmarked leaves of that tree simultaneously. To solve this problem, we have to refine LD-resolution so that the depth-first strategy is used instead of the breadth-first strategy. At first sight it seems that to this end we have to implement the backtracking mechanism used by Prolog. Fortunately, it is not so. A simpler alternative is to generate at each stage all direct successors of the *leftmost* unmarked leaf only. In this way the backtracking process is taken care of automatically.

Having discussed the modifications of the LD-resolution we now model the computation process of Prolog, by providing a formal definition of P-resolution. The central notion in this definition is that of a *P-tree*. We define them as the limit of a sequence of *pre-P-trees*, which in turn are a subclass of a class of ordered trees called *semi-P-trees*.

Definition 4.4.1 A *semi-P-tree* (relative to P) is an ordered tree whose nodes contain queries, possibly marked with *success*, *failure*, or *error*. □

In an ordered tree, by definition, for every node there is a strict total order on its children. To define the behaviour of the cut operator, we use these total orders to define a partial order on the nodes of an ordered tree.

Definition 4.4.2 Let m, n be two nodes in an ordered tree. We say that n is ***to the right*** of m if for some predecessors m' and n' of m and n, respectively,

- m' and n' are siblings,
- m' is strictly smaller than n' in the total order on the children of a node. □

The first step in defining pre-P-trees is to define the effect of the cut operator.

Definition 4.4.3 Let $\mathcal{B}$ be a branch in a semi-P-tree, and let Q be a node in this branch with a cut atom as the leftmost atom. Then, the ***origin*** of this cut atom is the first predecessor of Q in $\mathcal{B}$ that contains less cut atoms than Q. □

To see that this definition properly captures the informal meaning of the origin note that, when following a branch from top to bottom, the cut atoms are introduced and removed in a First-In Last-Out manner.

Definition 4.4.4 Let $\mathcal{T}$ be a semi-P-tree, Q a query in $\mathcal{T}$ which has a cut atom as the leftmost atom, and Q' be the origin of this cut atom. Then, the operator *cut*$(\mathcal{T}, Q)$ removes from $\mathcal{T}$ all the nodes that

1. are descendants of Q', and
2. lie to the right of Q. □

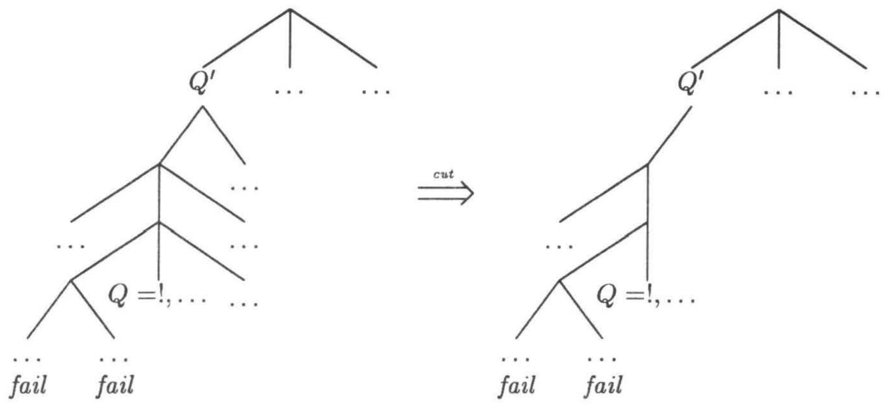


Figure 4.4: The effect of the operator *cut*($\mathcal{T}, Q$)

In Figure 4.4, we illustrate the effect of *cut*($\mathcal{T}, Q$).

Definition 4.4.5 The class of *pre-P-trees* is defined as follows:

- For every query Q, the tree consisting of the single unmarked node Q is a pre-P-tree (an *initial* pre-P-tree).
- If $\mathcal{T}$ is a pre-P-tree, then any *extension* of $\mathcal{T}$ is a pre-P-tree.

An *extension* of a pre-P-tree $\mathcal{T}$ is defined as follows:
Let Q be the leftmost unmarked leaf in $\mathcal{T}$. If Q is the empty query, mark Q as *successful.* Otherwise, let Q be of the form $A, \mathbf{M}$.

- Suppose A is an ordinary atom (i.e. not a special atom).
  - If Q has no resolvents with respect to a clause from P:
    Mark Q as *failure.*
  - If Q has such resolvents:
    For every clause c from P which are applicable to A, choose one resolvent Q' of Q with respect to c and add this as a child of Q in T. Choose the input clauses in such a way that all branches of T remain pseudo derivations. Order these children according to the the order in which their input-clauses appear in P.
- Suppose A is a cut atom.
  Apply the operation *cut*($\mathcal{T}, Q$).
  Provide Q with a single child $\mathbf{M}$.
- Suppose A is a meta-variable.
  Mark Q as *error.* □

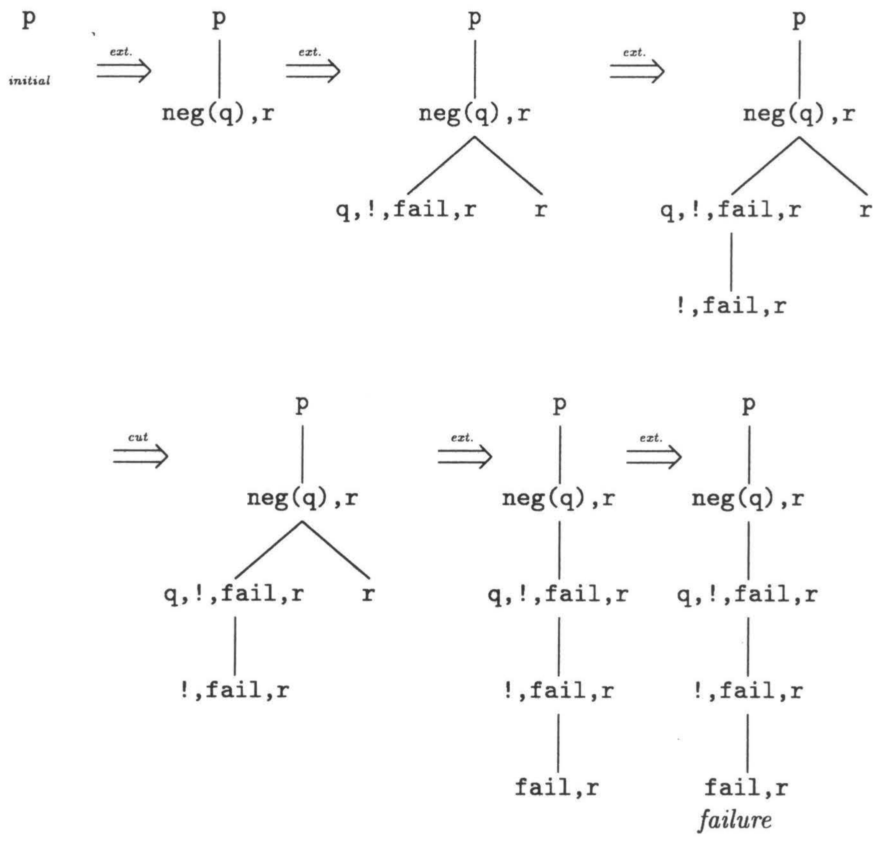


Figure 4.5: Step-by-step construction of a P-tree for the Prolog query p with respect to the Prolog program p ← neg(q), r. q ← . .

We now define P-trees as the limit of sequences of pre-P-trees. In Figure 4.5, we show how the notions of initial pre-P-trees and extensions of pre-P-trees can be used to construct a P-tree (the program used in the figure is the translation of the program used in Figure 4.1). Note that in this Figure, the result of the 'cut step' (that is, the fifth tree) is not itself part of the sequence of extensions; it was added to clarify the use of the cut operator in the construction of P-trees.

To be able to define the limit of a sequence of pre-P-trees, we have to define a notion of an *inclusion* between pre-P-trees, and of the *limit* of a growing sequence of pre-P-trees. For pre-LD-trees and pre-LDNF-trees, these notions were obvious. In the case of pre-P-trees, the pruning that takes place when extending a pre-P-tree, complicates the matter a bit.

Definition 4.4.6 Let $\mathcal{T}$ and $\mathcal{T}'$ be pre-P-trees. $\mathcal{T}$ is said to be *included* in $\mathcal{T}'$ if $\mathcal{T}'$ can be constructed from $\mathcal{T}$ by means of one of the following two operations:

1. adding some children to a leaf of $\mathcal{T}$.
2. removing a single subtree from $\mathcal{T}$, provided its root has siblings in $\mathcal{T}$.

We say that $\mathcal{T}$ is *properly included* in $\mathcal{T}'$, if $\mathcal{T}$ is included in $\mathcal{T}'$ and $\mathcal{T}'$ is not included in $\mathcal{T}$. We use $\subset$ to denote the transitive closure of the relation "$\mathcal{T}$ is properly included in $\mathcal{T}'$" and define $\mathcal{T} \subseteq \mathcal{T}'$ as $(\mathcal{T} \subset \mathcal{T}') \vee (\mathcal{T} = \mathcal{T}')$. □

Note that operation (2) never turns an internal node into a leaf.

Lemma 4.4.7 *The relation $\subset$ is a strict partial order on pre-P-trees.*

Proof: We have to prove that the conditions for a strict partial order hold.

1. $\mathcal{T} \not\subset \mathcal{T}$

   Suppose by contradiction that $\mathcal{T} \subset \mathcal{T}$. Then, there exists a $\mathcal{T}'$ such that $\mathcal{T}$ is properly included in $\mathcal{T}'$, and $\mathcal{T}' \subseteq \mathcal{T}$. There are two cases:

   - $\mathcal{T}'$ is constructed by adding children to a leaf of $\mathcal{T}$.
     But then, some node Q that is a leaf in $\mathcal{T}$, is an internal node in $\mathcal{T}'$. By definition of inclusion, and the fact that $\mathcal{T}' \subseteq \mathcal{T}$, Q is an internal node in $\mathcal{T}$. This is in contradiction with the fact that Q is a leaf in $\mathcal{T}$.
   - $\mathcal{T}'$ is constructed by pruning a single subtree from $\mathcal{T}$.
     By definition of inclusion, the parent of the pruned subtree has at least two children in $\mathcal{T}$, and therefore, it has at least one child in $\mathcal{T}'$. Moreover, new nodes can only "grow" from leaves. Thus subtrees pruned from $\mathcal{T}$ can never be "regenerated", to reconstruct $\mathcal{T}$ out of $\mathcal{T}'$. Therefore, $\mathcal{T}' \not\subseteq \mathcal{T}$, which leads to a contradiction.

2. $\mathcal{T} \subset \mathcal{T}'$ and $\mathcal{T}' \subset \mathcal{T}''$ imply $\mathcal{T} \subset \mathcal{T}''$.

   Straightforward by the definition of $\subset$. □

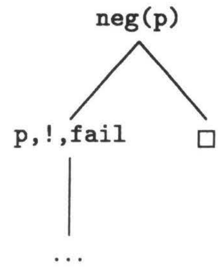


Figure 4.6: A P-tree for the query neg(p) with respect to p $\leftarrow$ p.

Corollary 4.4.8 *The relation $\subseteq$ is a partial order on pre-P-trees.* □

Clearly, with this notion of inclusion, we have that if $\mathcal{T}$ extends $\mathcal{T}'$ in the sense of Definition 4.4.5, then $\mathcal{T}' \subseteq \mathcal{T}$, so we can use this notion of extension to construct monotonously growing chains of pre-P-trees.

Definition 4.4.9

- A *P-tree* is a limit of a sequence $\mathcal{T}_0, \ldots, \mathcal{T}_i, \ldots$ such that $\mathcal{T}_0$ is an initial pre-P-tree, and for all i, $\mathcal{T}_{i+1}$ is an extension of $\mathcal{T}_i$.
- *A P-tree for Q* is a P-tree whose root is the query Q.
- A P-tree is called *finite* if no infinite branch exists in it. □

Formally, this definition is justified by the fact that every countable partial order with the least element (here the relation $\subseteq$ on pre-P-trees with the initial pre-P-tree as least element) can be canonically extended to a countable cpo (see e.g. Gierz [GHK$^+$80]).

Next, we define the concepts of *successful* and *finitely failed* P-trees.

Definition 4.4.10

- A P-tree is called *successful* if one of its leaves is marked as *success*.
- A (pre-)P-tree is called *finitely failed*, if it is finite, and all its leaves are marked as *failure*. □

Note that in P-trees, in contrast to LDNF-trees, some leaves can be unmarked. Whenever this is the case, the P-tree will contain exactly one infinite branch to the left of all these unmarked leaves. Such unmarked leaves represent the resolvents the Prolog computation process did not reach, because it got "trapped" in an infinite derivation (the infinite branch). For example, take the program p ← p., and the query **neg(p)**. Its P-tree is shown in Figure 4.6. This tree contains a branch ending with a leaf containing the empty query. However, this leaf is never reached by the Prolog computation process (and therefore never marked) because there is an infinite branch to the left of it.

Finally, it is clear how to define the notion of a computed answer substitution.

Definition 4.4.11 Consider a successful derivation in a pre-P-tree for Q. Let $\alpha_1, \ldots, \alpha_n$ be the consecutive substitutions along this branch.

Then the restriction $(\alpha_1 \cdots \alpha_n)|Q$ of the composition $\alpha_1 \cdots \alpha_n$ to the variables of Q is called a *computed answer substitution* (*c.a.s.* for short) of Q. □

4.5 Correspondence between LDNF-Trees and P-Trees: the Finite Case

In this section, we prove that there is a close correspondence between (computed answers of) LDNF-trees and P-trees. More precisely, we prove that termination results on normal programs with respect to LDNF-resolution translate directly into termination of their translated Prolog programs with respect to Prolog computation. For this purpose, we examine finite LDNF-trees, and their corresponding P-trees.

Theorem 4.5.1 *Let $\mathcal{T}_L$ be a finite LDNF-tree for a normal query Q. Then, there exists a finite P-tree $\mathcal{T}_P$ for Q such that $\mathcal{T}_L$ and $\mathcal{T}_P$ have the same set of computed answers.*

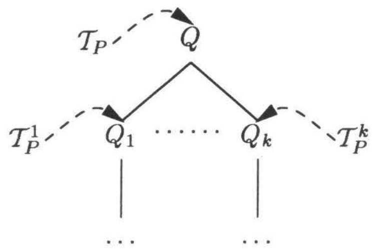
Proof: We prove the claim by induction on the depth of LDNF-trees (cf. Definition 4.3.3). Assume that the claim holds for all LDNF-trees of depth less than r. We have to prove the claim for LDNF-trees of depth r.

Let $\mathcal{T}_L$ be an LDNF-tree for Q of some finite depth r. In the remainder of this proof, we identify a normal query with its translation into a Prolog query. From the context it will always be clear whether we refer to a normal query, or a Prolog query. Two cases arise.

- Suppose that Q is of the form $A, \mathbf{L}$.

  Let $Q_1, \ldots, Q_k$ $(k \geq 0)$ be the children of Q in $\mathcal{T}_L$. Let, for $i \in [1..k]$, $\mathcal{T}_L^i$ denote the subtree of $\mathcal{T}_L$ starting at Q_i.

  As, for $i \in [1..k]$, $\mathcal{T}_L^i$ is finite and of depth less than r, by induction hypothesis there exists a P-tree $\mathcal{T}_P^i$ for Q_i such that $\mathcal{T}_P^i$ contains the same computed answers as $\mathcal{T}_L^i$. Now consider the semi-P-tree $\mathcal{T}_P$ with root Q, children $Q_1, \ldots, Q_k$ (ordered according to the order of their input clauses in P) and, for $i \in [1..k]$, $\mathcal{T}_P^i$ as the subtree starting at Q_i, as depicted by the following diagram:



To prove that $\mathcal{T}_P$ is a P-tree for Q, it is sufficient to show that all pruning caused by selection of cut atoms is guaranteed to be local to the respective subtrees $\mathcal{T}_P^i$ (for $i \in [1..k]$). Neither Q, nor its children $Q_1, \ldots, Q_k$ in $\mathcal{T}_P$, contain a cut atom, so no atom in $\mathcal{T}_P$ has Q as its origin. It follows from the definition of the cut operator that all pruning is indeed local to the respective subtrees $\mathcal{T}_P^i$. Thus $\mathcal{T}_P$ is a P-tree for Q. From its construction, it follows that it contains the same computed answers as $\mathcal{T}_L$. Moreover, it is finite.

- Suppose that Q is of the form $\neg A, \mathbf{L}$.

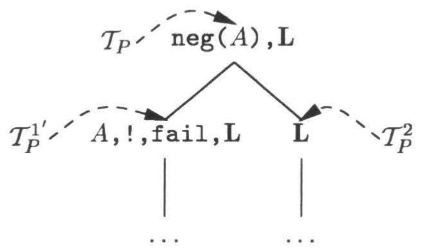
  Let $\mathcal{T}_L^1$ be the subtree of $\mathcal{T}_L$ starting at the root of *subs*(Q). As the LDNF-tree $\mathcal{T}_L^1$ for A is finite and of depth less than r, by induction hypothesis there exists a finite P-tree $\mathcal{T}_P^1$ for A that has the same computed answers as $\mathcal{T}_L^1$. There are two sub-cases.

  - Suppose that Q has a child in $\mathcal{T}_L$.

    Then, $\mathcal{T}_L^1$ is finitely failed, and therefore $\mathcal{T}_P^1$ is finitely failed as well. But then, we can construct a finitely failed P-tree $\mathcal{T}_P^{1'}$ for $A, !, \texttt{fail}, L$. In this P-tree, the cut atom introduced at the root will never be reached.

    Let $\mathcal{T}_L^2$ be the subtree of $\mathcal{T}_L$ starting at the single child $\mathbf{L}$ of Q. As the LDNF-tree $\mathcal{T}_L^2$ for $\mathbf{L}$ is finite and of depth less than r, by induction hypothesis there exists a finite P-tree $\mathcal{T}_P^2$ for $\mathbf{L}$ that has the same computed answers as $\mathcal{T}_L^2$.

    Using $\mathcal{T}_P^{1'}$ and $\mathcal{T}_P^2$ we can construct a finite P-tree $\mathcal{T}_P$ for Q that has the same computed answers as $\mathcal{T}_L$. This tree has the following form:



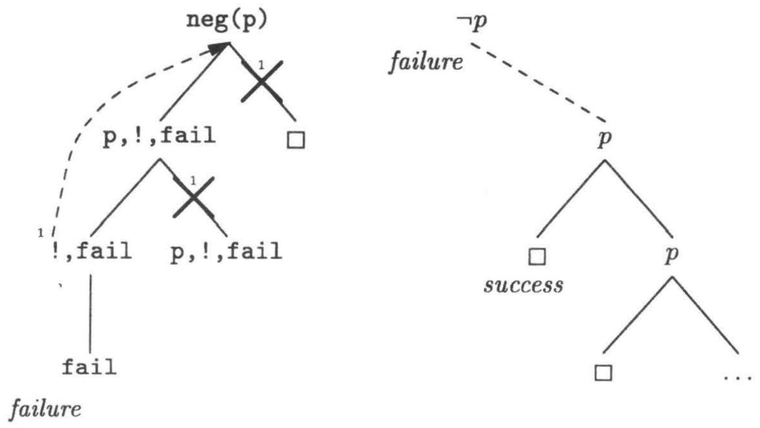


Figure 4.7: A P-tree and an LDNF-tree for neg(p)

- Suppose that Q has no children in $\mathcal{T}_L$.

  Then, $\mathcal{T}_L^1$ is successful, and therefore $\mathcal{T}_P^1$ is successful as well. But then we can construct a finitely failed P-tree $\mathcal{T}_P^{1'}$ for $A, !, \texttt{fail}, L$, in which the cut atom present in its root is selected at some point.

  Let $\mathcal{T}_P$ be the semi-P-tree such that its root is Q, and the subtree starting at the single child $A, !, \texttt{fail}, L$ of Q is $\mathcal{T}_P^{1'}$. In this tree, the origin of the cut atom that appears in the single child of Q, is Q. This cut atom is the selected atom in some node within $\mathcal{T}_P^{1'}$. Thus $\mathcal{T}_P$ is a P-tree for Q, because the potential second child of Q, that would contain the query $\mathbf{L}$ has been pruned at some stage. Thus $\mathcal{T}_P$ is finitely failed, just as $\mathcal{T}_L$ is. □

Thus if we have a normal query Q that terminates with respect to a normal program P, we know that Prolog computation on that query and that program will terminate, and give the same computed answers as LDNF-resolution.

Now what if we have a finite P-tree for a restricted Prolog query Q and a restricted Prolog program P? Consider the following restricted Prolog program

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

and the restricted Prolog query **neg(p)**. The P-tree and LDNF-tree for this query and this program are shown in Figure 4.7 (note that the pruned branches are not really part of the P-tree for **neg(p)**, but existed at some point during the construction of this P-tree). In this example, the P-tree is finite, because the potentially infinite branch caused by the clause **p** ← **p** is pruned. However, in the LDNF-tree, this branch has been constructed in full, and therefore this LDNF-tree is infinite.

4.6 Correspondence between LDNF-Trees and P-Trees: the Infinite Case

In this section, we prove a correspondence between P-trees and LDNF-trees that are possibly infinite. For this, a simple induction on the depth of P-trees is not sufficient. Instead, we define an *entailment function* from the nodes of a P-tree to the nodes of a corresponding LDNF-tree, such that a branch in the P-tree is mapped onto the 'equivalent' branch in the LDNF-tree.

We restrict ourselves in this section to propositional programs. This allows us to evade a number of complications related to the handling of substitutions. A direct result of this is, that there are no computed answer substitutions, and that therefore we are only interested whether the marks on the leaves of the P-trees and LDNF-trees correspond.

From the previous section, we know that P-trees and LDNF-trees do not correspond with respect to finite failure. As it turns out, there is a notion of 'failure' on which P-trees and LDNF-trees do correspond.

Definition 4.6.1

- A P-tree is *unsuccessful* if all leaves are marked as *failure*.
- An LDNF-tree is *unsuccessful* if no leaf of the main tree is marked as *success*. □

Informally, a tree should be considered *unsuccessful*, when it proves that there does not exist an (SLDNF) computed answer for the query in the root node. Note the difference of formulation for P-trees and LDNF-trees. A P-tree should not be considered *unsuccessful*, if it has unmarked leaves. This is because unmarked leaves represent (possibly successful) branches that were not reached by the Prolog computation process, but that are explored by the breadth-first mechanism used when constructing LDNF-trees. On the other hand, an LDNF-tree is already considered *unsuccessful* when all leaves in the main tree are either marked as *failure* or unmarked. The reason for this is, that a leaf in the main tree is only unmarked, if its subsidiary tree is infinite. But then, the branch of the main tree ending in this leaf cannot be successful, i.e. it is unsuccessful.

Theorem 4.6.2 (Infinite Correspondence) *Let Q be a restricted query and P be a restricted program.*

1. *If there exists a successful P-tree for Q with respect to P, then there exists a successful LDNF-tree for Q with respect to P.*
2. *If there exists an unsuccessful P-tree for Q with respect to P, then there exists an unsuccessful LDNF-tree for Q with respect to P.*

To prove the theorem, we define a so-called ***entailment function*** that maps nodes in a P-tree to 'similar' nodes in an LDNF-tree. In defining this notion of 'similar', we use the following ***strip*** operation.

Definition 4.6.3 Let Q be a Prolog query. We define $\boldsymbol{strip}(Q)$ to be the largest prefix of Q that is a restricted Prolog query. □

Definition 4.6.4 Let $\mathcal{T}_P$ be a P-tree and let $\mathcal{T}_L$ be an LDNF-tree. An *entailment function ent* from $\mathcal{T}_P$ to $\mathcal{T}_L$ is a partial bijective function from nodes in $\mathcal{T}_P$ to nodes in $\mathcal{T}_L$ that satisfies the following conditions:

1. For the root Q of $\mathcal{T}_P$, if $ent(Q)$ is defined, then it is the root of the main tree in $\mathcal{T}_L$.
2. If Q and Q' are nodes in $\mathcal{T}_P$ such that Q' is a child of Q and $ent(Q')$ is defined, then $ent(Q)$ is defined, and either $ent(Q')$ is a child of $ent(Q)$ or $ent(Q')$ is $subs(ent(Q))$.
3. If Q is a node in $\mathcal{T}_P$, such that $ent(Q)$ is defined, then $ent(Q)$ is of the form $strip(Q)$.

For two entailment functions *ent* and *ent'* from $\mathcal{T}_P$ to $\mathcal{T}_L$ we say that *ent contains ent'* if

- $Dom(ent') \subseteq Dom(ent)$, and
- for all $Q \in Dom(ent')$, $ent'(Q) = ent(Q)$.

An entailment function *ent* from $\mathcal{T}_P$ to $\mathcal{T}_L$ is *maximal*, if there exists no entailment function *ent'* (different from *ent*) such that *ent* is contained in *ent'*. □

Thus, informally, an entailment function maps a 'cap' of a P-tree onto a 'cap' of an LDNF-tree, such that both 'caps' have the same structure, and nodes in the P-tree are mapped on to 'similar' nodes in the LDNF-tree (note that stripped atoms are either special atoms or atoms that are never selected).

First, we prove existence of maximal entailment functions.

Lemma 4.6.5 *Let Q be a query and P be a program. Let $\mathcal{T}_P$ be a P-tree for Q with respect to P and let $\mathcal{T}_L$ be an LDNF-tree for Q with respect to P. Then there exists a maximal entailment function from $\mathcal{T}_P$ to $\mathcal{T}_L$.*

Proof: We construct a sequence of entailment functions from $\mathcal{T}_P$ to $\mathcal{T}_L$, and prove that the least fixpoint of this sequence is a maximal entailment function from $\mathcal{T}_P$ to $\mathcal{T}_L$.

Let the *initial* function ent_0 be the function that maps the root of $\mathcal{T}_P$ onto the root of $\mathcal{T}_L$, and is undefined on all other nodes of $\mathcal{T}_P$. Clearly this is an entailment function from $\mathcal{T}_P$ to $\mathcal{T}_L$.

Let *ent* be an entailment function from $\mathcal{T}_P$ to $\mathcal{T}_L$. Construct an *extension ent'* of *ent* by mapping $ent(Q)$ onto Q', for all Q and Q' that satisfy the following conditions:

1. Q is a node in $\mathcal{T}_P$ such that *ent*(Q) is undefined,
2. Q has a parent Q'' such that *ent*(Q'') is defined,
3. Q' is either a child of Q'' or $Q' = subs(Q'')$, and
4. $Q' = strip(Q)$.

Thus, we can construct a (possibly infinite) sequence $ent_0, \ldots, ent_\alpha, \ldots$ of entailment functions from $\mathcal{T}_P$ to $\mathcal{T}_L$. Moreover, because the extension operator is monotonous, we know that the sequence has a least fixpoint (in fact, this least fixpoint will be reached in at most ω steps, because the length of the branches is at most ω). Clearly, the least fixpoint of $ent_0, \ldots, ent_\alpha, \ldots$ is a maximal entailment function from $\mathcal{T}_P$ to $\mathcal{T}_L$. □

In the following lemma we show that, with restricted programs and queries, the domain of a maximal entailment functions is 'big enough', in the sense that

1. it contains all successful branches in P-trees, and
2. it contains all unsuccessful branches, except for their leaves.

Lemma 4.6.6 *Let Q be a restricted query and let P be a restricted program. Let $\mathcal{T}_P$ be a P-tree with respect to P and let $\mathcal{T}_L$ be an LDNF-tree for Q with respect to P. Let ent be a maximal entailment function from $\mathcal{T}_P$ to $\mathcal{T}_L$. If Q is a node in $\mathcal{T}_P$ such that ent(Q) is undefined, then either*

- *Q is an unmarked leaf, or*
- *the selected atom in Q is* `fail`.

Proof: We prove the Lemma by contradiction. Let Q be a node in $\mathcal{T}_P$ such that *ent*(Q) is undefined, and Q is neither an unmarked leaf, nor has it `fail` as its selected atom. Let Q' be the first predecessor of Q in $\mathcal{T}_P$ such that *ent*(Q') is defined, and let Q'' be the child of Q' that is either a predecessor of Q, or Q itself. Let Q' be of the form $A, \mathbf{L}$. Three cases arise:

- Suppose that A is neither a special atom, nor of the form $\texttt{neg}(B)$.

  We know that *ent*(Q') is of the form *strip*(Q'). Thus, the leftmost literals in Q' and *ent*(Q') coincide. But then, by the definitions of P-trees and LDNF-trees, *ent*(Q') has a child of the form *strip*(Q''). As a result, we can extend *ent* to an entailment function *ent'* by mapping Q'' onto this child of *ent*(Q'). But this is in contradiction with the fact that *ent* is a maximal entailment function from $\mathcal{T}_P$ to $\mathcal{T}_L$.

- Suppose A is of the form $\texttt{neg}(B)$.

  Then, Q' has only children of the form $B, !, \texttt{fail}, L$ and of the form $\mathbf{L}$.

- Suppose Q'' is of the form $B, !, \texttt{fail}, L$.

  We know that *subs*(*ent*(Q')) is of the form B, i.e. that it is equivalent to *strip*(Q''). As a result, we can extend *ent* to an entailment function *ent'* by mapping Q'' onto *subs*(*ent*(Q')). But this is in contradiction with the fact that *ent* is a maximal entailment function from $\mathcal{T}_P$ to $\mathcal{T}_L$.

- Suppose Q'' is of the form $\mathbf{L}$.

  We know that the subtree starting at the first child of Q' is finite, because Q is a descendant of Q'' (the second child of Q'), and because Q is not an unmarked leaf, it must have been extended at some point. Moreover, because Q'' is not pruned, we know that in this subtree the cut atom never became leftmost atom. Thus, there exists a finitely failed P-tree for B. But then, by Lemma 4.5.1, there exists a finitely failed LDNF-tree for B, and therefore *ent*(Q') must have a child of the form *strip*($\mathbf{L}$). As a result, we can extend *ent* to an entailment function *ent'* by mapping Q'' onto this (single) child of *ent*(Q'). But this is in contradiction with the fact that *ent* is a maximal entailment function from $\mathcal{T}_P$ to $\mathcal{T}_L$.

- Suppose A is a special atom.

  Then A is either a cut atom or a `fail` atom. A cannot be a `fail` atom, because then Q' would have no children.

  If A is a !, then Q'' has `fail` as leftmost atom. But then Q'' has no children and therefore Q'' and Q must be the same. But that is in contradiction with the fact that Q does not have `fail` as selected atom. □

We also want to prove that, in the case of restricted programs and queries, also the range of maximal entailment functions is 'big enough'. For our purposes, the range is 'big enough' if, when the P-tree contains no unmarked leaves, the range contains all marked leaves of the main tree of the LDNF-tree.

Lemma 4.6.7 *Let Q be a restricted query and let P be a restricted program. Let $\mathcal{T}_P$ be a P-tree with respect to P and let $\mathcal{T}_L$ be an LDNF-tree for Q with respect to P. Let ent be a maximal entailment function from $\mathcal{T}_P$ to $\mathcal{T}_L$. If $\mathcal{T}_P$ has no unmarked leaves, then all leaves of the main tree of $\mathcal{T}_L$ are in the range of ent.*

Proof: Let Q be a leaf of the main tree of $\mathcal{T}_L$ such that Q is not in the range of *ent*. Let Q'_L be the first predecessor of Q that is in the range of *ent* and let Q''_L be the child of Q'_L that is a predecessor of Q. Let Q'_L be of the form $L, \mathbf{L}$. We know that for some Q'_P in $\mathcal{T}_P$, $Q'_L = ent(Q'_P)$, where, for some $\mathbf{M}$, Q'_P is of the form $L, \mathbf{L}, \mathbf{M}$. Two cases arise:

- Suppose that L is positive.

  We know that Q'_P cannot be an unmarked leaf, because $\mathcal{T}_P$ has no unmarked leaves. Moreover, because Q'_L has children in $\mathcal{T}_L$, Q'_P cannot be marked as either *failure* or *success*. But then, because *ent* is a bijection, and Q'_P and Q'_L have the same number of children, Q'_P must have a child Q''_P such that *ent*(Q''_P) is undefined and $Q''_L = strip(Q''_P)$. As a result, we can extend *ent* to an entailment function *ent'* by mapping Q''_P onto Q''_L. But this is in contradiction with the fact that *ent* is a maximal entailment function from $\mathcal{T}_P$ to $\mathcal{T}_L$. Thus assuming that L is positive leads to a contradiction.

- Suppose that L is negative.

  Then Q''_L is of the form $\mathbf{L}$. Because Q'_L has a child, we know that the LDNF-tree for L is finitely failed. But then, by Lemma 4.5.1, there exists a finitely failed P-tree for L, and therefore there exists a finitely failed P-tree for $L, !, \texttt{fail}, L, M$ in which the cut atom originating from its root is never reached. But then, Q'_P has a child Q''_P of the form $\mathbf{L}, \mathbf{M}$. Because *ent* is a bijection, Q'_P has 2 children, and Q'_L has one child and *subs*(Q'_L) is defined, we know that *ent*(Q''_P) is undefined. As a result, we can extend *ent* to an entailment function *ent'* by mapping Q''_P onto Q''_L. But this is in contradiction with the fact that *ent* is a maximal entailment function from $\mathcal{T}_P$ to $\mathcal{T}_L$. Thus assuming that L is negative leads to a contradiction.
  □

Proof:(of Theorem 4.6.2)

1. Suppose that $\mathcal{T}_P$ is successful. Then $\mathcal{T}_P$ contains a branch whose leaf is marked as *success*. To prove that $\mathcal{T}_L$ is successful, it is sufficient to prove that $\mathcal{T}_L$ contains a branch whose leaf is marked as *success*.

   Let $C_1, \ldots, C_n$ be a branch in $\mathcal{T}_P$ whose leaf is marked as *success*. By Lemma 4.6.5, there exists a maximal entailment function *ent* from $\mathcal{T}_P$ to $\mathcal{T}_L$. Moreover, by Lemma 4.6.6, *ent*(Q_n) is defined, and therefore by definition of entailment functions, *ent*(Q_i) is defined for all $i \in [1..n]$. Let, for $i \in [1..n]$, Q'_i denote *ent*(Q_i). Clearly, $Q'_1, \ldots, Q'_n$ is a branch in $\mathcal{T}_L$. Moreover, Q_n is the empty query, and therefore, by the definition of *strip*, Q'_n is also the empty query. But then, by definition of LDNF-trees, Q'_n is marked as *success*. Thus $Q'_1, \ldots, Q'_n$ is a branch in $\mathcal{T}_L$ whose leaf is marked as *success*.

2. Suppose that $\mathcal{T}_P$ is unsuccessful. Then, all leaves of $\mathcal{T}_P$ are marked as failure. But then, by Lemma 4.6.7, for all leaves Q_L in the main tree of $\mathcal{T}_L$, there exists a node Q_P in $\mathcal{T}_P$ such that *ent*$(Q_P) = Q_L$. As $\mathcal{T}_P$ is

unsuccessful, it does not contain empty queries. But then, no leaf of the main tree of $\mathcal{T}_L$ can contain the empty query, and therefore no leaf of the main tree of $\mathcal{T}_L$ can be marked as *success*. □

The converse of Theorem 4.6.2 does not hold. Consider the following Prolog program:

```
p ← p.
p ← .
```

The P-tree for the query p has no branches marked with *success*. On the other hand, an LDNF-tree for the query p and the corresponding logic program *has* a successful branch. If we omit the second clause of this program, and take again the query p we have an example where the P-tree is not unsuccessful, but the corresponding LDNF-tree is.

4.7 Applications

Due to the presence of cut in the definition of the predicate **neg** it is difficult to reason in a declarative way about Prolog programs that use negation. In other words, it is not clear how to prove correctness of such programs using their declarative interpretation.

We now show how this is possible using the results of this chapter. The key observation is that Theorem 4.5.1 provides a crucial relationship between the computational behaviour of Prolog programs and their translations into normal logic programs.

In the subsequent discussion we assume that the variables in the input clauses and the mgu's are chosen in a fixed way. We can then assume that for every Prolog program P and Prolog query Q there exists exactly one P-tree, and similarly for normal logic programs, normal queries and LDNF-trees.

So consider a restricted Prolog program P with a restricted query Q and their translation P_L and Q_L onto a normal logic program and a normal logic query, respectively. To reason about correctness of P with Q it is sufficient to reason about P_L and Q_L. Indeed, suppose that we proved already that all LDNF-derivations of P and Q are finite. Then by Theorem 4.5.1 the P-tree for P_L and Q_L is finite, and P_L with Q_L and P with Q have the same set of computed answers.

As an example, let us consider the following well-known Prolog program **TRANS**, about which one claims that it computes the transitive closure of a binary relation e:

```
trans(X,Y,E,Avoids) ← member([X,Y],E).
trans(X,Z,E,Avoids) ← member([X,Y],E),
                      neg(member(Y,Avoids)),
                      trans(Y,Z,E,[Y|Avoids]).
member(X,[X|Xs]) ← .
member(X,[Y|Xs]) ← member(X,Xs).
```

In Apt [Apt94] the following facts about its translation $\mathtt{TRANS}_L$ to a normal logic program and a binary relation e were established:

- all LDNF-derivations of trans(X, Y, e, [ ]) are finite,
- the computed answer substitutions of trans(X, Y, e, [ ]) determine all pairs of elements which form the transitive closure of e.

Now, by Theorem 4.5.1 the same conclusions can be drawn about the original program TRANS.

The fact that the above approach to correctness is limited to restricted Prolog programs is in our opinion not serious. After all, these methods were designed to verify and analyze declarative programs. So it is natural that they can be applied only to Prolog programs that are use only the 'declarative' part of the Prolog language.

4.8 Related Work

We conclude by briefly discussing related work.

There is an enormous literature on the subject of negation in logic programming, see, e.g., the references in the surveys cited in the introduction. However, to our knowledge, no work has been done on negation in Prolog.

The use of the ambivalent syntax was first advocated in mathematical logic by Richards [Ric74], in the logic programming setting by Kalsbeek [Kal93] and Jiang [Jia94], and in the programming languages area by Chen, Kifer and Warren[CKW89] in their language proposal HiLog. In each of these references different versions of ambivalence are assumed. For example, in Kalsbeek [Kal93] atoms can appear as terms and in Jiang [Jia94] formulas can appear as terms. Here we use an alternative version of ambivalence, which amounts to identification of atoms and terms, though we also allow meta-variables.

The definition of the LDNF-resolution given in Section 3 is derived from the definition of the SLDNF-resolution provided in Apt and Doets [AD94]. An alternative definition of SLDNF-resolution was given earlier by Martelli and Tricomi [MT92]. Both definitions overcome problems encountered in the original definition of Clark [Cla78].

In our operational semantics of Prolog programs, given in Section 4, we also provided a meaning to the cut operator. The problem of formalizing the meaning of cut has been studied in a number of publications during the last 10 years. Jones and Mycroft [JM84] defined various semantics for Prolog with cut. This work was pursued by Arbab and Berry [AB87], Debray and Mishra [DM88], and more recently by Lilly and Bryant [LB92].

In the literature several alternatives to the cut operator have been proposed – see e.g. Moss [Mos86] and more recent Hill, Lloyd and Shepherdson [HLS90].

Chapter 5

Three-Valued Completion for Abductive Logic Programs

Summary

In this chapter, we propose a three-valued completion semantics for abductive logic programs, which solves some problems associated with Console et al's two-valued completion semantics. The semantics is a generalization of Kunen's completion semantics for normal logic programs, which is known to correspond very well to a class of effective proof procedures for normal logic programs. Secondly, we propose a proof procedure for abductive logic programs, which is a generalization of a proof procedure for normal logic programs based on constructive negation. This proof procedure is sound and complete with respect to the proposed semantics. By generalizing a number of results on normal logic programs to the class of abductive logic programs, we present further evidence for the idea that limited forms of abduction can be added quite naturally to normal logic programs.

5.1 Introduction

Abduction is a form of inference where one, given some rules and an observation, tries to find an explanation of that observation using these rules. For instance, given a rule

$$shoes\_are\_wet \leftarrow it\_is\_raining$$

the observation *shoes_are_wet* is explained by *it_is_raining*. As such, abduction is quite the reverse of deduction, where facts and rules are used to deduce conclusions. In the above example, from the fact *shoes_are_wet* one cannot deduce anything. But from the fact *it_is_raining* one can deduce *shoes_are_wet*. One can find abduction in many fields within the realm of artificial intelligence and knowledge engineering, including diagnosis, planning, computer vision, natural language understanding, default reasoning, and knowledge assimilation.

Abductive logic programming (first proposed in [EK89]) is a crossover between logic programming and abduction. The idea is to represent the rules as a logic program and the observation as a query. Then, abduction is used to infer an explanation using program and query. The best known semantics for abductive logic programs are those based upon (generalized) stable models [Dun91, SI92, KM90] and argumentation semantics [Dun91, KM91]. Proof procedures for these semantics were proposed by K. Eshghi and R. Kowalski [EK89] and extended by K. Satoh and N. Iwayama [SI92] and T. Kakas and P. Mancarella [KM90]. In [CDT91], L. Console, D.T. Dupre and P. Torasso propose a different kind of semantics, based on the two-valued completion of a program. The aim of their paper was to investigate the relation between abduction and deduction. In [DS92], M. Denecker and D. DeSchreye propose a proof procedure for such a two-valued completion semantics, which is based on SLDNF-resolution. For a thorough overview on abductive logic programming and their semantics, we refer to the excellent survey by A.C. Kakas, R.A. Kowalski and F. Toni [KKT93].

In normal logic programming, completion semantics was developed as a semantics for describing what can be computed using SLDNF-resolution. By giving a completion semantics for abductive logic programming, Console et al showed that abduction is closely related to deduction. Denecker and DeSchreye added to this by proposing SLDNFA-resolution, a proof procedure for abductive logic programming based on SLDNF-resolution. However, a disadvantage of the two-valued completion approach is, that it is not defined for arbitrary programs: for many interesting programs there do not exist two-valued models of their completion. In normal logic programming, it has been shown that three-valued semantics are better suited to characterize proof procedures based on SLDNF-resolution than two-valued semantics. In [Fit85], M. Fitting proposes a three-valued immediate consequence operator, on which he bases a semantics (*Fitting semantics*). Basically, it states that a formula is true in a program iff it is true in all three-valued Herbrand models of the completion of that program. In [Kun87], K. Kunen proposes an alternative to this semantics (*Kunen semantics*), in which a formula is true in a program iff it is true in all three-valued models of the completion of that program.

In this chapter, we generalize Fitting semantics and Kunen semantics to abductive logic programs. In the process, we also propose a three-valued immediate consequence operator, and truth- and falseness formulae as presented by J.C. Shepherdson in [She88b], for abductive logic programs. With this, we provide abduction with a semantics which gives a good characterization of the answers that can be actually computed by effective proof procedures. This in contrast with semantics based on well founded semantics, whose proof procedures involve expensive loop checking, and those based on stable model semantics, which become intractable as soon as function symbols are used. Such complex semantics are interesting from the point of view of knowledge representation, and are definitely of use in specific problem domains, but are not viable candidate semantics for general purpose (abductive) logic programming systems. This in contrast to Kunen semantics, which is commonly used in the verification of normal logic programs. By generalizing Kunen semantics to abductive logic programs, we present further evidence for the idea that limited forms of abduction can be added quite naturally to normal logic programs.

The obtained results are also interesting within the context of *modular logic programming*, where one reasons with predicates (called *open* predicates) which are (partially) undefined, or defined in other modules, and of *constraint logic programming*, where some of the predicates represent constraints that are to be handled by a constraint solver. In each of these cases there is a distinction between the 'logic programming part' of the program and some other part, which is either abducible, handled by an other (unknown) logic program, or handled by a constraint solver. In these contexts, it is interesting to see if we can find a semantics of the logic programming part, which is parametric with respect to the 'abducible', 'open' or 'constraint' part.

Moreover, we present an alternative proof procedure, based upon SLDFA-resolution: a proof procedure proposed by W. Drabent in [Dra95]. This proof procedure solves some problems associated with SLDNFA-resolution. First of all, by using constructive negation instead of negation as failure, we remove the problem of *floundering*. Secondly, instead of skolemizing non-ground queries, which introduces some undesired technicalities, we use equality in our language, which allows a natural treatment of non-ground queries.

The chapter is organized in three more or less separate parts. In the first part, we give an introduction to abductive logic programming (Section 5.3), and present two- and three-valued completion semantics (Section 5.4). Then, in the second part, which starts with Section 5.5, we present the immediate consequence operator (Section 5.6), and use it to characterize Fitting semantics (Section 5.7) and Kunen semantics (Section 5.8) for abductive logic programs. In the third part, we generalize SLDFA-resolution to the case of abductive logic programs (Section 5.9), and present some soundness and completeness results on SLDFA-resolution in Sections 5.10 and 5.11.

5.2 Preliminaries and Notation

In this chapter, we use k, l, m and n to denote natural numbers, f, g and h to denote functions (constants are treated as 0-ary functions), x, y and z to denote variables, s, t and u to denote terms, p, q and r to denote predicate symbols, A, B and C to denote atoms, L, M and N to denote literals, Q and R to denote queries, θ, δ, σ, τ and ρ to denote abducible formulae (they will be defined later) and ϕ and ψ to denote formulae.

We use boldface to denote finite sequences of objects. Thus, $\mathbf{L}$ denotes a sequence $L_1, \ldots, L_n$ of literals and **s** denotes a sequence $s_1, \ldots, s_n$ of terms. Moreover, in formulae we identify comma with conjunction. Thus, **L** (also) denotes a conjunction $L_1 \land \ldots \land L_n$. Finally, for two sequences $s_1, \ldots, s_k$ and $t_1, \ldots, t_k$ of terms, we use $(\mathbf{s} = \mathbf{t})$ to denote the formula $(s_1 = t_1) \land \ldots \land (s_k = t_k)$.

In the remainder of this section, we introduce some basic notions concerning algebras and models. For a more thorough treatment of these notions, we refer to [Doe93]. To begin with, an *algebra* (or *pre-interpretation*, as it is called in [Llo87]), is the part of a model that interprets the terms of the language.

Definition 5.2.1 Let $\mathcal{L}$ be a language and let $\mathcal{F}$ be the set of function symbols in $\mathcal{L}$. An *$\mathcal{L}$-algebra* is a complex $J = \langle D, \mathbf{f}, \ldots \rangle_{f \in \mathcal{F}}$ where D is a non-empty set, the *domain* (or *universe*) of J, and for every n-ary function symbol $f \in \mathcal{F}$, **f** is an n-ary function $\mathbf{f} : D^n \to D$. □

Note that constant symbols are treated as 0-ary functions. Interpretation of terms of $\mathcal{L}$ in a $\mathcal{L}$-algebra J is defined as usual.

We now define the notion of two- and three-valued *models*.

Definition 5.2.2 Let $\mathcal{L}$ be a language. Let $\mathcal{F}$ be the set of function symbols in $\mathcal{L}$ and let $\mathcal{R}$ be the set of predicate symbols in $\mathcal{L}$. A *two-valued $\mathcal{L}$-model* is a complex $M = \langle D, \mathbf{f}, \ldots \mathbf{r}, \ldots \rangle_{f \in \mathcal{F}, r \in \mathcal{R}}$ where $\langle D, \mathbf{f}, \ldots \rangle_{f \in \mathcal{F}}$ is an $\mathcal{L}$-algebra, for every n-ary predicate symbol $r \in \mathcal{R}$, **r** is a subset of D^n, and equality (if present) is interpreted as identity. □

Definition 5.2.3 Let $\mathcal{L}$ be a language. Let $\mathcal{F}$ be the set of function symbols in $\mathcal{L}$ and let $\mathcal{R}$ be the set of predicate symbols in $\mathcal{L}$. A *three-valued $\mathcal{L}$-model* is a complex $M = \langle D, \mathbf{f}, \ldots \mathbf{r}, \ldots \rangle_{f \in \mathcal{F}, r \in \mathcal{R}}$ where $\langle D, \mathbf{f}, \ldots \rangle_{f \in \mathcal{F}}$ is an $\mathcal{L}$-algebra, for every n-ary predicate symbol $r \in \mathcal{R}$, **r** is an n-ary function $\mathbf{r} : D^n \to \{\mathbf{t}, \mathbf{f}, \perp\}$, and equality (if present) is interpreted as two-valued identity. □

Following [Doe93], we treat equality as a special predicate with a fixed (two-valued) interpretation.

For two-valued models, the interpretation of (complex) formulae is defined as usual. For three-valued models, the interpretation of (complex) formulae is defined by the use of Kleene's truth-tables for three-valued logic. We use $\models$ to denote ordinary two-valued logical consequences, while $\models_3$ is used for three-valued logical consequences ($T \models_3 \phi$ iff ϕ is true in all three-valued models of T).

In this chapter, we always use equality in the context of Clark's Equality Theory (*CET*), which consists of the following *Free Equality Axioms*:

(i) $f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n) \to (x_1 = y_1) \land \ldots \land (x_n = y_n)$ $(\forall f)$
(ii) $f(x_1, \ldots, x_n) \neq g(y_1, \ldots, y_m)$ ($\forall$ distinct f and g)
(iii) $x \neq t$ (for all x and t where x is a proper sub-term of t)

Note, that the fixed interpretation of equality replaces the usual equality axioms, which are normally part of *CET*.

Whenever we work with two-valued models we always assume the language $\mathcal{L}$ to contain the propositional variables **t** and **f**, with a fixed interpretation (**t** is always true, **f** is always false). Additionally, when working with three-valued models, we assume $\mathcal{L}$ to contain the propositional variable $\perp$, which is always undefined, i.e. mapped onto $\perp$.

One important algebra is the *Herbrand Algebra HA*. It is the algebra that has the set of all closed terms as domain, and maps each closed term on 'itself'. Given an algebra J, a *J-model* is a model with algebra J. For instance, the set of all HA-models is the set of all Herbrand models. A *CET-algebra* is an algebra that satisfies CET. Note that every CET-algebra extends HA.

For a formula ϕ, $FreeVar(\phi)$ denotes the set of free variables in ϕ. A *sentence* is a closed formula (i.e. $FreeVar(\phi)$ is empty). A *ground* formula is a quantifier-free sentence. A *ground instance* of a formula ϕ is a formula ϕ' such that ϕ' is the result of substituting all variables in ϕ (free *and* local ones) by ground terms.

When working with some language $\mathcal{L}$ and models over some domain D, it is sometimes useful to work with the domain elements of D as if they were constants. This can be done using the following definitions. Given a language $\mathcal{L}$ and a domain D, the *D-language* $\mathcal{L}_D$ is obtained by extending $\mathcal{L}$ with a fresh constant for every domain element in D. When working in some language $\mathcal{L}$ and referring to D-sentences or D-formulae, we intend sentences or formulae in the language $\mathcal{L}_D$. We can extend an $\mathcal{L}$-algebra J to an $\mathcal{L}_D$-algebra J_D by interpreting each new constant in $\mathcal{L}_D$ 'as itself', and extend a J-model M to a J_D-model M_D by replacing the algebra J by the algebra J_D. Given a domain D, a language $\mathcal{L}$ and a formula ϕ, a *D-ground* instance of ϕ is a ground instance of ϕ in the language $\mathcal{L}_D$.

Lemma 5.2.4 *Let J be an algebra with domain D and let M be a J-model. Let ϕ be a quantifier-free formula. Then, $M \models \phi$ iff for all D-ground instances ϕ' of ϕ $M_D \models \phi'$.*

In the following, given a model M with domain D and a D-ground formula ϕ, we write $M \models \phi$ whenever we intend $M_D \models \phi$. Also, given an algebra J with domain D, we sometimes refer to D-ground formulae as J-ground formulae. For a logic program P, we use *J-ground*(P) to denote the (possibly infinite) set of all J-ground instances of clauses in P.

In the remainder of this chapter, we will not always specify the language. When no language is given, we assume a fixed 'universal' language $\mathcal{L}_{\mathcal{U}}$, which has a (countably) infinite number of constant and function symbols of all arities. The advantage of using such a universal language is, among others, that for that language CET is complete.

5.3 Abductive Logic Programming

Abduction is the process of generating an explanation E, given a theory T and an observation Ψ. More formally, E is an explanation for an abductive problem $\langle T, \Psi \rangle$, if $T \cup E$ is consistent, Ψ is a consequence of $T \cup E$, and E satisfies "some properties that make it interesting".

In this chapter, we limit ourselves to the context of abductive logic programs, in which T is an *abductive logic program*, Ψ is a formula and E is an *abducible formula.*

Definition 5.3.1 An *abductive logic program* P is a triple $\langle \mathcal{A}_P, \mathcal{R}_P, \mathcal{I}_P \rangle$, where

- $\mathcal{A}_P$ is a set of *abducible predicates*,
- $\mathcal{R}_P$ is finite set of clauses $A \leftarrow \theta, \mathbf{L}$, where A is a non-abducible atom, θ is an abducible formula and $\mathbf{L}$ is a sequence of non-abducible literals, and
- $\mathcal{I}_P$ is a finite set of first-order integrity constraints.

An *abducible formula* (with respect to a program P) is a first-order formula build out of the equality predicate '$=$' and the abducible predicates. □

Example 5.3.2 *Here is an example abductive logic program P_{Tweety}.*

- $\mathcal{A}_{P_{Tweety}} = \{penguin, ostrich\}$
- $$\mathcal{R}_{P_{Tweety}} = \left\{ \begin{array}{l} flies(x) \leftarrow bird(x) \ \wedge \ \neg ab(x) \\ ab(x) \leftarrow penguin(x) \\ ab(x) \leftarrow ostrich(x) \\ bird(tweety). \end{array} \right\}$$
- $\mathcal{I}_{P_{Tweety}} = \emptyset$ ○

In the remainder of this chapter, no integrity constraints are used, i.e. $\mathcal{I}_P$ is always empty. Integrity constraints are used to restrict the explanations for a given observation to a smaller class of 'legal' explanations. As such, they can be seen completely separate from the 'program part' of the abductive logic program, which specifies the explanations for a given observation. In this chapter, we concentrate on the 'program part' of abductive logic programs. That is, we give a semantics for it, and develop a proof procedure for it. On a more practical level, a reason for omitting integrity constraints is, that the proof procedure we propose has no way of dealing with them in full generality. One should note however, that there exist techniques that, under certain conditions, can translate integrity constrains to some set $\mathcal{IR}_P$ of program rules with head *False* (a propositional variable). Instead of testing whether a candidate explanation δ for a problem $\langle P, \phi \rangle$ satisfies the integrity constraints, one can find an explanation for the problem $\langle P', \phi \ \wedge \ \neg False \rangle$, where P' is the program $\langle \mathcal{A}_P, \mathcal{R}_P \cup \mathcal{IR}_P, \emptyset \rangle$.

If we compare our definition of abductive logic programs with the definitions given in [KKT93], the main difference is, that we add equality to our abducible formulae. Of course, equality is not abducible, in the sense that one can assume two terms to be equal, in order to explain an observation; we use equality in context of *CET*, which completely defines the meaning of '=' (*CET* is a complete theory when a universal language is used). However, when one thinks of the class of abducible formulae as the class of formulae that can be used to explain a given observation, it makes perfect sense to include equality. Note that also K. Eshghi in [Esh88] uses a kind of equality in its abducible formulae. However, it is a restricted notion of equality, consisting of only the identity and transitivity axioms, and inequality between distinct skolem constants.

5.4 Completion Semantics for Abductive Logic Programs

In [Cla78], K. L. Clark introduces the notion of *completion* of a normal logic program, and proposes the (two-valued) completion semantics for normal logic programs. The central notion in the definition of the completion of a program is the notion of the *completed definition* of a predicate.

Definition 5.4.1 Let P be a program and let p be a predicate symbol in the language of P. Let n be the arity of p and let $x_1, \ldots, x_n$ be a sequence of fresh variables. Let $p(\mathbf{s}_1) \leftarrow \theta_1, \mathbf{L}_1 \ \ldots \ p(\mathbf{s}_m) \leftarrow \theta_m, \mathbf{L}_m$ be the clauses in P with head p, and let, for $i \in [1..m]$, $\mathbf{y_i} = \mathit{FreeVar}(\theta_i, \mathbf{L}_i) - \mathit{FreeVar}(p(\mathbf{s}_i))$. The *completed definition of p (with respect to P)* is the formula

$$p(\mathbf{x}) \cong \bigvee_{i \in [1..m]} \exists_{\mathbf{y}_i}((\mathbf{x} = \mathbf{s}_i), \theta_i, \mathbf{L}_i)$$

□

Intuitively, the completed definition of a predicate states that "*p* is true *iff* there exists a rule for *p* whose body is true".

The *completion* (*comp*(P)) of a normal logic program consists of the completed definitions of its predicates, plus *CET* to interpret equality correctly. In the (two-valued) completion semantics for normal logic programs, a formula is true in a program iff it is true in all (two-valued) models of the completion of that program.

In [CDT91], Console et al propose a two-valued completion semantics for abductive logic programs. The idea is, that the completion of an abductive logic program only contains completed definitions of non-abducible predicates. As a result, the theory *comp*(P) contains no information on the abducible predicates (i.e. the abducible predicates can be freely interpreted).

Definition 5.4.2 Let P be an abductive logic program. The *completion of* P (denoted by *comp*(P)) is the theory that consists of *CET* and, for every non-abducible predicate p in P, the completed definition of p. □

Example 5.4.3 *Given the program P_{Tweety} of Example 5.3.2, the completed program $comp(P_{Tweety})$ consists of the following formulae*

$$\begin{aligned} \mathit{flies}(x) &\cong \mathit{bird}(x) \wedge \neg \mathit{ab}(x) \\ \mathit{ab}(x) &\cong \mathit{penguin}(x) \vee \mathit{ostrich}(x) \\ \mathit{bird}(\mathit{tweety}) &\cong \mathbf{t}. \end{aligned}$$

plus CET. ○

Using this notion of completion for abductive logic programs, Console et al give an object level characterization of the explanation of an abductive problem $\langle P, \phi \rangle$. Intuitively, it is the formula (unique up to logical equivalence) that represents all possible ways of explaining the observation in that abductive problem. Before we can give its definition, we have to introduce the notion of *most specific* abducible formula.

Definition 5.4.4 For abducible formulae θ and σ, θ is *more specific* than σ if $CET \models \forall\theta \rightarrow \forall\sigma$. θ is *most specific* if there does not exist a σ (different from θ, modulo logical equivalence) such that σ is more specific than θ. □

We now give the definition of explanation, as proposed by Console et al (i.e. the object level characterization of Definition 2 in [CDT91]). As we want to reserve the term 'explanation' for an alternative notion of explanation we define later on, we use the term 'full explanation' here.

Definition 5.4.5 Let $\langle P, \phi \rangle$ be an abductive problem. Let δ be an abducible formula. Then, δ is *the full explanation of* $\langle P, \phi \rangle$, if δ is the most specific abducible formula such that $comp(P) \cup \{\phi\} \models \delta$, and $comp(P) \cup \{\delta\}$ is consistent. □

Note, that in this definition ϕ and δ switched positions with respect to the ordinary characterization of abduction. The advantage of this definition is, that for a given abductive problem, the full explanation is unique (up to logical equivalence).

Example 5.4.6 *Consider the program P_{Tweety} and observation $\neg \mathit{flies}(\mathit{tweety})$. The full explanation for this observation is $\mathit{penguin}(\mathit{tweety}) \vee \mathit{ostrich}(\mathit{tweety})$. With this simple program, this can be easily checked by using the equivalence formulae in $comp(P_{Tweety})$:*

$$\begin{aligned} \neg \mathit{flies}(\mathit{tweety}) &\cong \neg(\mathit{bird}(\mathit{tweety}) \wedge \neg \mathit{ab}(\mathit{tweety})) \\ &\cong \neg \mathit{bird}(\mathit{tweety}) \vee \mathit{ab}(\mathit{tweety})) \\ &\cong \neg \mathbf{t} \vee \mathit{penguin}(\mathit{tweety}) \vee \mathit{ostrich}(\mathit{tweety}) \\ &\cong \mathit{penguin}(\mathit{tweety}) \vee \mathit{ostrich}(\mathit{tweety}) \end{aligned}$$

○

In their paper, Console et al restrict their abductive logic programs to the class of ***hierarchical programs***. As a reason for this, they argue that 'it is useless to explain a fact in terms of itself'. Practical reasons for this restriction are twofold: it ensures consistency of $comp(P)$, and soundness and completeness of their 'abstract' proof procedure ABDUCE. Although we agree that, as is the case with normal logic programs, a large class of naturally arising programs turn out to be hierarchical, we do not want to restrict ourselves to hierarchical programs. Moreover, the problem of checking whether a given program is hierarchical is not always easy (see [AB91] for some techniques). Thus, instead of restricting ourselves to hierarchical programs, in the definition of full explanation, we added the condition that $comp(P) \cup \{\delta\}$ has to be consistent.

We now define an alternative notion of 'explanation'. This second definition is more in line with the normal characterization of abduction. However, it is also weaker, in the sense that there can exist more than one explanation for a given abductive problem.

Definition 5.4.7 Let $\langle P, \phi\rangle$ be an abductive problem. An abducible formula δ is *a (two-valued) explanation for* $\langle P, \phi\rangle$, if $comp(P) \cup \{\delta\} \models \phi$ and $comp(P) \cup \{\delta\}$ is consistent. □

Example 5.4.8 *Consider again the problem* $\langle P_{Tweety}, \neg flies(tweety)\rangle$. *Then, penguin(tweety) is an explanation, and so is ostrich(tweety).* ∘

Note, that in both δ and ϕ the free variables are implicitly universally quantified. Thus, there is no 'communication' between free variables in δ and ϕ. As a result, the observation $flies(x)$ does not stand for the generic "given a hypothetical individual x, what can you tell me (about x) when I observe that x flies". Instead, it just states that you observe that "all x fly".

The following lemma shows that the full explanation of a given abductive problem is less specific than any explanation for that abductive problem.

Lemma 5.4.9 *Let* $\langle P, \phi\rangle$ *be an abductive problem, let* δ *be the full explanation of* $\langle P, \phi\rangle$, *and let* θ *be an explanation for* $\langle P, \phi\rangle$. *Then,* $CET \models \forall\theta \rightarrow \forall\delta$.

Proof: δ is the full explanation of $\langle P, \phi\rangle$, and therefore $comp(P) \cup \{\phi\} \models \delta$, which implies $comp(P) \models \forall\phi \rightarrow \forall\delta$. Moreover, θ is an explanation for $\langle P, \phi\rangle$, and therefore $comp(P) \cup \{\theta\} \models \phi$, which implies $comp(P) \models \forall\theta \rightarrow \forall\phi$. But then, it follows that $comp(P) \models \forall\theta \rightarrow \forall\delta$. But $\forall\theta \rightarrow \forall\delta$ is an abducible formula and therefore $CET \models \forall\theta \rightarrow \forall\delta$. □

Thus, the difference between the two kinds of explanations is, that the full explanation incorporates all possible ways of explaining a given observation, while an (ordinary) explanation is a formula that is just sufficient to explain that given observation.

In the above, we used two-valued completion as a semantics. In normal logic programming, there also exists a three-valued completion semantics. In this semantics, the third truth-value models the fact that effective proof procedures

$\leftrightarrow$	**t**	**f**	$\perp$
t	**t**	**f**	$\perp$
f	**f**	**t**	$\perp$
$\perp$	$\perp$	$\perp$	$\perp$

$\cong$	**t**	**f**	$\perp$
t	**t**	**f**	**f**
f	**f**	**t**	**f**
$\perp$	**f**	**f**	**t**

Figure 5.1: Kleene equivalence and strong equivalence

cannot determine truth or falsity for all formulae. Thus, the third truth-value ($\perp$) stands for 'truth-value undetermined'. In the following example, we show how this third truth-value can be useful.

Example 5.4.10 *Let us construct the program $P_{Tweety'}$ by adding to P_{Tweety} the seemingly irrelevant clause*

$$p \leftarrow \neg p$$

The completion $comp(P_{Tweety'})$ has no two-valued models. As a result, the observation $\neg flies(tweety)$ has no two-valued explanations. This problem can be solved by assigning to p the third truth-value $\perp$, i.e. by switching to a three-valued logic. ○

In Section 5.5, we characterize Fitting semantics and Kunen semantics for abductive logic programs, using a three-valued immediate consequence operator. In the remainder of this section, we present these semantics using a model-theoretic approach.

Fitting semantics and Kunen semantics use the same notion of completion as the one used in two-valued semantics. However, they use it in the setting of three-valued models. In this three-valued setting, special care must be taken to interpret the equivalence operator, used in the completed definition of a predicate, correctly. Intuitively, this equivalence should enforce that the left-hand side and the right-hand side of the completed definition have the same truth-value. However, Kleene's three-valued equivalence ($\leftrightarrow$) stands for something like "the truth-values of left and right hand sides are equal and neither one is unknown". Therefore, instead of $\leftrightarrow$, another notion of equivalence ($\cong$) is used, which has the required truth-table (see figure 5.1). The operator $\cong$ cannot be constructed using Kleene's operators, and therefore has to be introduced separately. Its use is be restricted: it is only used in the completed definition of a predicate. Note, that $\leftrightarrow$ and $\cong$ are equivalent when restricted to the truth-values **t** and **f**.

Using a model-theoretic approach, Fitting semantics and Kunen semantics can be stated very succinctly.

Definition 5.4.11 Let $\langle P, \phi \rangle$ be an abductive problem. A consistent abducible formula δ is a *(model theoretic) explanation for $\langle P, \phi \rangle$ (in Fitting semantics)*, if ϕ is true in all three-valued Herbrand models of $comp(P) \cup \{\delta\}$. □

Definition 5.4.12 Let $\langle P, \phi\rangle$ be an abductive problem. A consistent abducible formula δ is a *(model theoretic) explanation for* $\langle P, \phi\rangle$ *(in Kunen semantics)*, if $comp(P) \cup \{\delta\} \models_3 \phi$. □

Note, that in these definitions only consistency of δ (with respect to CET) is required. The reason is, that in three-valued completion the completed definitions of the program-rules are always consistent. In these two definitions, we added the prefix 'model theoretic', to distinguish them from the proof theoretic Definitions 5.7.1 and 5.8.1. In the following, when we refer to a three-valued explanation, we refer to a model theoretic explanation in Kunen semantics.

From these definitions, it is easy to see that any model theoretic explanation in Kunen semantics is also a model theoretic explanation in Fitting semantics. The converse, however, does not hold. To get an idea of the difference, consider the following example, involving the *universal query problem* in (abductive) logic programming.

Example 5.4.13 *Let P be the program:*

$$\langle p(c) \leftarrow q, \{q\}, \emptyset\rangle$$

Let ϕ be the formula $\forall_x p(x)$. Now, consider the abductive problem $\langle P, \phi\rangle$. In Fitting semantics (over the language $\mathcal{L}_P$), q is a model theoretic explanation for this problem. The reason is, that in Herbrand models, domain elements are isomorphic to terms of the language. On the other hand, if we allow arbitrary (three-valued) models, we can choose richer models. For instance, consider the model M with domain $\{c, d\}$, in which c is mapped onto itself, in which q and p(c) are true, but p(d) is false. Clearly, M is a model of comp(P) $\cup \{q\}$. However, ϕ is not true in M, and therefore q is not an explanation for ϕ in Kunen semantics. ○

There is a large difference in the handling of inconsistencies between two- and three-valued completion. In the following example, we show how inconsistencies 'disappear' in three-valued completion semantics.

Example 5.4.14 *Consider the abductive logic program P, with a single abducible predicate a, and the following two clauses:*

$$\begin{array}{l} p \leftarrow \neg p, a \\ q \leftarrow a \end{array}$$

Then, comp(P) $\cup \{a\}$ is obviously inconsistent in two-valued completion, because when a is true, the completed definition of p reduces to $p \cong \neg p$. Thus, among others, a is not a (two-valued) explanation for $\langle P, q\rangle$. However, by assigning $\perp$ to p, we can construct three-valued models of comp(P), and therefore a is *a three-valued explanation for $\langle P, q\rangle$.* ○

Thus, there is a choice to be made between two-valued and three-valued semantics. In our opinion, the choice of semantics depends on your view on abductive logic programs, and the relation between abducible and non-abducible predicates. If one assumes that a program, i.e. the definition of the non-abducible predicates, can contain implicit information on the abducible predicates, in the form of potential inconsistencies, one should use two-valued completion. On the other hand, if one thinks of abducible predicates as *completely* undefined (apart from integrity constraints), or thinks that only integrity constraints should be used for constraining the abducible predicates, one can use three-valued completion, because then inconsistencies can be seen as flaws in the program. But even if one thinks that the two-valued semantics is the proper one, Kunen's three-valued semantics remains interesting, because it describes the explanations that can actually be computed using a SLD-like proof procedure. However, one still would have to prune those explanations that are inconsistent with respect to the program.

5.5 Three-Valued Completion Semantics

In definition 5.4.12 of the previous section, we generalized Kunen semantics to abductive logic programs. The definition as given there is, however, very succinct. For one thing, it does not express the intention behind both Fitting and Kunen semantics. That is, that the third truth-value stands for something like 'truth-value not determined'.

In [Fit85], M. Fitting proposes the use of three-valued semantics for normal logic programs, using the third truth-value ($\perp$) to represent the fact that for some formulae, the truth-value cannot be determined. For this purpose, Fitting introduced a three-valued immediate consequence operator Φ_P, to characterize the meaning of a normal logic program (cf. Definition 2.7.1). He proves that the fixpoints of this operator are three-valued Herbrand models of the completed program, and takes the least fixpoint of this operator as the meaning of that program (*Fitting semantics*). However, as Fitting points out, in general this semantics is highly non-constructive: the closure ordinal for the least fixpoint can be as high as ω_1; the first non-recursive ordinal.

In [Kun87], K. Kunen proposes a semantics in which the iteration of Fitting's immediate consequence operator is cut-off at ordinal ω. Moreover, he proves that a sentence ϕ is true in his semantics iff ϕ is true in all three-valued models of $comp(P)$.

In the following sections, we define an immediate consequence operator for abductive logic programs, and use it to characterize Fitting semantics and Kunen semantics for abductive logic programs. In the process, we also generalize Shepherdson's truth- and falseness formulae (see [She88b]).

5.6 The Immediate Consequence Operator

Let us now define a three-valued immediate consequence operator for abductive logic programs. For normal logic programs, the immediate consequence operator Φ_P operates on models, and $\Phi_P(M)$ denotes the one-step consequences of M, given a program P (see Definition 2.7.1 in Chapter 2). If we use this operator on an abductive logic program, this operator generates all observations that need no explanation (i.e. are explained by the formula $\mathbf{t}$). We however, want to build an operator that generates all observations ϕ that are explained by some explanation δ. Therefore, we define an operator $\Phi_{P,\delta}$, such that $\Phi_{P,\delta}(\mathcal{M})$ denotes the one-step consequences of $\mathcal{M}$, given an abductive logic program P and an explanation δ. So, we compute immediate consequences in P, under the assumption that δ holds. One problem is, that for an arbitrary abducible formula δ, δ cannot be characterized by a single model. For instance, if δ is of the form $p(a) \lor p(b)$, it has two minimal models. Therefore, $\Phi_{P,\delta}$ operates on sets of models. In [Fit85] and [Kun87], Φ_P operates on Herbrand models. We however follow K. Doets [Doe93], and define the operators on arbitrary J-models, given an algebra J. If we annotate the operators with an algebra J, they operate on J-models. When no algebra is given, they operate on Herbrand models (HA is our 'default' algebra).

Thus, the operator $\Phi_{P,\delta}$ operates on sets of models. To facilitate its definition and various proofs, we define the operator $\Phi_{P,\delta}$ in two steps. First, we define an operator $\Phi_{P,\Delta}$, which operates on models. Then, in the second step, we define $\Phi_{P,\delta}$ in terms of $\Phi_{P,\Delta}$. In $\Phi_{P,\Delta}$, a model Δ models the abducible predicates of P. The idea is that, because Δ is a model instead of an abducible formula, the set of immediate consequences of a model M in P under assumption Δ can be characterized by a single model. Because we want Δ to model the abducible predicates only, we first have to introduce the notion of ***abducible models***.

Definition 5.6.1 Let P be a program. A model M is an ***abducible model*** (with respect to P), if all non-abducible atoms in P are mapped to $\bot$ in M. □

Example 5.6.2 *Given the program P_{Tweety}, Let us define the Herbrand model M_{Tweety} as follows: penguin(tweety) is* $\mathbf{t}$ *in M_{Tweety}, all other ground abducible atoms are* $\mathbf{f}$ *in M_{Tweety}, and all ground non-abducible atoms are $\bot$ in M_{Tweety}. Then M_{Tweety} is an abducible model (with respect to P_{Tweety}).* ∘

Now, the definition of $\Phi_{P,\Delta}$ is a straightforward generalization of the operator Φ_P for normal logic programs. For non-abducible atoms, the definition stays the same. However, for an abducible atom A, A is true (resp. false) in $\Phi_{P,\Delta}(M)$ iff it is true (resp. false) in Δ.

Definition 5.6.3 Let P be a program. Let J be an algebra and let Δ be a abducible J-model. The three-valued immediate consequence $\Phi^J_{P,\Delta}$ is defined as follows:

$$\Phi^J_{P,\Delta}(M)(A) = \begin{cases} \mathbf{t} & , \Delta \models_3 A \text{ or} \\ & \text{there exists a } A \leftarrow \theta, \mathbf{L} \text{ in } J\textit{-ground}(P) \\ & \text{such that } \Delta \models_3 \theta \text{ and } M \models_3 \mathbf{L} \\ \mathbf{f} & , \text{if } \Delta \models_3 \neg A \text{ or} \\ & \text{for all } A \leftarrow \theta, \mathbf{L} \text{ in } J\textit{-ground}(P) \\ & \text{either } \Delta \models_3 \neg\theta \text{ or } M \models_3 \neg\mathbf{L} \\ \bot & , \text{otherwise} \end{cases}$$

The powers of $\Phi^J_{P,\Delta}$ are defined as follows:

$$\Phi^J_{P,\Delta} \uparrow \alpha = \begin{cases} \Delta & , \text{if } \alpha = 0 \\ \Phi^J_{P,\Delta}(\Phi^J_{P,\Delta} \uparrow n-1) & , \text{if } \alpha \text{ is a successor ordinal} \\ \bigcup_{\beta<\alpha} \Phi^J_{P,\Delta} \uparrow \beta & , \text{if } \alpha \text{ is a limit ordinal} \end{cases}$$

□

Note that this definition is not standard for $\alpha = 0$. We could define $\Phi^J_{P,\delta} \uparrow 0$ to be the empty set, but at the cost of having a special treatment of the base case in some of the lemmas.

Example 5.6.4 *Given the program P_{Tweety} and abducible model M_{Tweety}, we can observe that:*

- *penguin(tweety) is* **t** *in $\Phi_{P_{Tweety},M_{Tweety}} \uparrow 0$, therefore*
- *ab(tweety) becomes* **t** *in $\Phi_{P_{Tweety},M_{Tweety}} \uparrow 1$, and therefore*
- *flies(tweety) becomes* **f** *in $\Phi_{P_{Tweety},M_{Tweety}} \uparrow 2$.* ○

Now, we can define $\Phi_{P,\delta}$. We do not define $\Phi_{P,\delta}(\mathcal{M})$ for arbitrary sets of models $\mathcal{M}$. Instead, we only define $\Phi_{P,\delta} \uparrow \alpha$, for arbitrary ordinals α.

Definition 5.6.5 Let P be a program and let δ be a consistent abducible formula. Let J be an algebra and let $\mathcal{M}$ be the set of abducible J-models of $\{\delta\}$. Then,

$$\Phi^J_{P,\delta} \uparrow \alpha = \{\Phi^J_{P,\Delta} \uparrow \alpha \mid \Delta \in \mathcal{M}\}$$

□

In [She88b], J.C. Shepherdson defines the notion of truth- and falseness formulae. These formulae give an elegant alternative characterization of what is computed by the immediate consequence operator. We generalize these formulae to abductive logic programs.

Definition 5.6.6 Let P be a program. For a natural number n and a formula ϕ, we define the formulae $T_n(\phi)$ and $F_n(\phi)$ as follows:

- If ϕ is an abducible formula, then for all n

$$T_n(\phi) \stackrel{def}{=} \phi \qquad F_n(\phi) \stackrel{def}{=} \neg\phi$$

- If ϕ is an atom of the form $p(\mathbf{s})$, where p is a non-abducible predicate, then $comp(P)$ contains a definition $p(\mathbf{x}) \cong \psi$, where $FreeVars(\psi) = \mathbf{x}$. We define

$$T_0(\phi) \stackrel{def}{=} \mathbf{f} \qquad F_0(\phi) \stackrel{def}{=} \mathbf{f}$$

and

$$T_n(\phi) \stackrel{def}{=} T_{n-1}(\mathbf{x} = \mathbf{s} \wedge \psi) \qquad F_n(\phi) \stackrel{def}{=} F_{n-1}(\mathbf{x} = \mathbf{s} \wedge \psi)$$

- If ϕ is a complex formula, we define

$$\begin{array}{rlrl}
T_n(\neg\phi) & \stackrel{def}{=} F_n(\phi) & F_n(\neg\phi) & \stackrel{def}{=} T_n(\phi) \\
T_n(\phi \wedge \psi) & \stackrel{def}{=} T_n(\phi) \wedge T_n(\psi) & F_n(\phi \wedge \psi) & \stackrel{def}{=} F_n(\phi) \vee F_n(\psi) \\
T_n(\phi \vee \psi) & \stackrel{def}{=} T_n(\phi) \vee T_n(\psi) & F_n(\phi \vee \psi) & \stackrel{def}{=} F_n(\phi) \wedge F_n(\psi) \\
T_n(\forall \mathbf{x}\phi) & \stackrel{def}{=} \forall \mathbf{x} T_n(\phi) & F_n(\forall \mathbf{x}\phi) & \stackrel{def}{=} \exists \mathbf{x} F_n(\phi) \\
T_n(\exists \mathbf{x}\phi) & \stackrel{def}{=} \exists \mathbf{x} T_n(\phi) & F_n(\exists \mathbf{x}\phi) & \stackrel{def}{=} \forall \mathbf{x} F_n(\phi)
\end{array}$$

□

Example 5.6.7 *Given the program P_{Tweety}, we have that*

$$\begin{aligned}
T_2(\neg \mathit{flies}(tweety)) &= F_2(\mathit{flies}(tweety)) \\
&= F_1(bird(tweety) \wedge \neg ab(tweety)) \\
&= F_1(bird(tweety)) \vee F_1(\neg ab(tweety)) \\
&= F_1(bird(tweety)) \vee T_1(ab(tweety)) \\
&= F_0(\mathbf{t}) \vee T_0(penguin(tweety) \vee ostrich(tweety)) \\
&= penguin(tweety) \vee ostrich(tweety)
\end{aligned}$$

○

The following lemma is a generalization of Lemma 4.1 in [She88b] to abductive logic programs.

Lemma 5.6.8 *Let P be a program. Let J be an algebra with domain D, let Δ be an abducible J-model and let ϕ be a D-sentence. Then, for all natural numbers n,*

1. $\Phi^J_{P,\Delta} \uparrow n \models_3 \phi$ *iff* $\Delta \models_3 T_n(\phi)$

2. $\Phi^J_{P,\Delta} \uparrow n \models_3 \neg\phi$ *iff* $\Delta \models_3 F_n(\phi)$

Proof: Suppose that ϕ is an abducible formula. Then we have that $T_n(\phi) = \phi$ and $F_n(\phi) = \neg\phi$. So, we only have to prove that $\Phi^J_{P,\Delta} \uparrow n \models_3 \phi$ iff $\Delta \models_3 \phi$. This follows directly from the construction of $\Phi^J_{P,\Delta}$. Thus, for abducible formulae, the claim holds.

Suppose that ϕ is not an abducible formula. For such ϕ, we prove the claim by induction on n and formula induction on ϕ. Consider the case where $n = 0$ and ϕ is a non-abducible atom $p(\mathbf{s})$. Then, by definition, $p(\mathbf{s})$ is $\bot$ in $\Phi^J_{P,\Delta} \uparrow 0$ and $T_0(p(\mathbf{s})) = F_0(p(\mathbf{s})) = \mathbf{f}$. Therefore, the claims hold.

Assume that the claim holds for all $m < n$. We prove that the claim also holds for n. First, consider the case where ϕ is the atom $p(\mathbf{s})$. Because ϕ is a D-sentence, $p(\mathbf{s})$ is J-ground. Because p is a non-abducible predicate, $comp(P)$ contains a definition $p(\mathbf{x}) \cong \psi$. Now,

$$\begin{array}{lll}
 & \Delta \models_3 T_n(p(\mathbf{s})) & \text{by definition of } T_n(p(\mathbf{s})) \\
\text{iff} & \Delta \models_3 T_{n-1}(\mathbf{x} = \mathbf{s} \wedge \psi) & \text{by induction hypothesis} \\
\text{iff} & \Phi^J_{P,\Delta} \uparrow n-1 \models_3 \mathbf{x} = \mathbf{s} \wedge \psi & \text{by construction of } \psi \\
\text{iff} & \exists\, p(\mathbf{s}) \leftarrow \mathbf{L} \in J\text{-}ground(P) : \Phi^J_{P,\Delta} \uparrow n-1 \models_3 \mathbf{L} & \\
 & & \text{by construction of } \Phi^J_{P,\Delta} \\
\text{iff} & \Phi^J_{P,\Delta} \uparrow n \models_3 p(\mathbf{s}) &
\end{array}$$

The reasoning for $F_n(p(\mathbf{s}))$ is similar.

Now, consider the case where ϕ is a complex formula. If ϕ is of the form $\neg\psi$, $\psi \wedge \eta$, or $\psi \vee \eta$, the claim follows from the construction of $T_n(\phi)$ and $F_n(\phi)$.

Suppose ϕ is of the form $\exists x\psi$. Then, $\Phi^J_{P,\Delta} \uparrow n \models_3 \exists x\psi$ iff, for some element a of the domain of J, $\Phi^J_{P,\Delta} \uparrow n \models_3 \psi(a)$. Because $\psi(a)$ is a D-sentence, we have by induction that $\Phi^J_{P,\Delta} \uparrow n \models_3 \psi(a)$ iff $\Delta \models_3 T_n(\psi(a))$. Finally, we have that $\Delta \models_3 T_n(\psi(a))$ iff $\Delta \models_3 T_n(\exists x\psi)$. The other cases with quantifiers are similar. □

Corollary 5.6.9 *Let P be a program and let δ be an abducible formula. Let J be an algebra with domain D and let ϕ be a D-sentence. Then,*

1. $\Phi^J_{P,\delta} \uparrow n \models_3 \phi$ *iff* $J \cup \{\delta\} \models_3 T_n(\phi)$

2. $\Phi^J_{P,\delta} \uparrow n \models_3 \neg\phi$ *iff* $J \cup \{\delta\} \models_3 F_n(\phi)$

Proof: The proof follows immediately from the fact that, for arbitrary ψ, $J \cup \{\delta\} \models_3 \psi$ iff ψ is true in all abducible J-models of $\{\delta\}$. □

5.7 Fitting Semantics for Abductive Logic Programs

In this section, we use the three-valued consequence operator defined in the previous section to generalize Fitting semantics to abductive logic programs.

Definition 5.7.1 Let $\langle P, \phi \rangle$ be an abductive problem. Let δ be a consistent abducible formula. Let $\mathcal{M}$ be the least fixpoint of $\Phi^{HA}_{P,\delta}$. Then, δ is a *proof theoretic explanation* for $\langle P, \phi \rangle$ *in Fitting semantics*, if $\mathcal{M} \models_3 \phi$. □

With Fitting semantics for normal logic programs, a formula is true in the Fitting semantics iff it is true in all three-valued Herbrand models. The same holds for Fitting semantics for abductive logic programs. In order to prove this, we first present two lemmas. First of all, the following lemma shows that the fixpoints of $\Phi_{P,\Delta}$ are indeed three-valued models of $comp(P) \cup \{\delta\}$.

Lemma 5.7.2 *Let P be a program and let δ be a consistent abducible formula. Let J be a CET-algebra, let Δ be an abducible J-model of $\{\delta\}$ and let M be a J-model. Then,*

$$\Phi^J_{P,\Delta}(M) = M \textit{ implies } M \models_3 comp(P) \cup \{\delta\}$$

Proof: Suppose that $\Phi^J_{P,\Delta}(M) = M$. The fact that M is a model of $\{\delta\}$ follows trivially from the definition of $\Phi^J_{P,\Delta}$. We have to prove that $M \models_3 comp(P)$. Because M is a J-model and J is a CET-algebra, we know that $M \models CET$.

Now, let $p(\mathbf{x}) \cong \psi$ be a formula in $comp(P)$. Let $p(\mathbf{a})$ be a J-ground atom. Then,

$$\begin{array}{lll} & M \models_3 \psi(\mathbf{a}) & \text{by definition of } \psi \\ \text{iff} & \exists\, p(\mathbf{a}) \leftarrow \mathbf{L} \in J\text{-}ground(P) : M \models_3 \mathbf{L} & \text{by definition of } \Phi^J_{P,\Delta} \\ \text{iff} & \Phi^J_{P,\Delta}(M) \models_3 p(\mathbf{a}) & \text{because } \Phi^J_{P,\Delta}(M) = M \\ \text{iff} & M \models_3 p(\mathbf{a}) & \end{array}$$

The proof for $\neg\psi(\mathbf{a})$ is similar. □

Corollary 5.7.3 *Let P be a program and let δ be a consistent abducible formula. Let J be a CET-algebra. If $\mathcal{M}$ is a fixpoint of $\Phi^J_{P,\delta}$, then*

$$\mathcal{M} \models_3 comp(P) \cup \{\delta\}$$

In the second lemma, we prove the converse. For this, we need the following definition.

Definition 5.7.4 Let P be a program and let M be a model. The *abducible projection* of M is the abducible model Δ such that

- $\Delta(A) = M(A)$, if A is an abducible atom, and
- $\Delta(A) = \bot$, otherwise. □

Lemma 5.7.5 *Let P be a program and let δ be an abducible formula. Let J be an algebra and let M be a J-model such that $M \models_3 comp(P) \cup \{\delta\}$. Let Δ be the abducible projection of M. Then, M is a fixpoint of $\Phi^J_{P,\Delta}$.*

Proof: Suppose that $M \models_3 comp(P) \cup \{\delta\}$.
We have to prove that $\Phi^J_{P,\Delta}(M) = M$.

- If L is an abducible J-ground literal, $\Delta \models_3 L$ iff $M \models_3 L$, and therefore, by definition of $\Phi^J_{P,\Delta}$, $\Phi^J_{P,\Delta}(M) \models_3 L$ iff $M \models_3 L$.
- If $p(\mathbf{a})$ is a non-abducible J-ground atom there exists a J-ground instance $p(\mathbf{a}) \cong \psi$ of a formula in $comp(P)$ such that

$$\begin{array}{lll} & \Phi^J_{P,\Delta}(M) \models_3 p(\mathbf{a}) & \text{by definition of } \Phi^J_{P,\Delta} \\ \text{iff} & \exists\, p(\mathbf{a}) \leftarrow \mathbf{L} \in J\text{-}ground(P) : M \models_3 \mathbf{L} & \text{by definition of completion} \\ \text{iff} & M \models_3 \psi & \text{because } M \models_3 comp(P) \\ \text{iff} & M \models_3 p(\mathbf{a}) & \end{array}$$

The case for $\neg p(\mathbf{a})$ is similar. □

Theorem 5.7.6 *Let $\langle P, \phi \rangle$ be an abductive problem. A consistent abducible formula δ is a proof theoretic explanation for $\langle P, \phi \rangle$ in the Fitting semantics iff ϕ is a model theoretic explanation in Fitting semantics.*

Proof: Let $\mathcal{M}$ be the least fixpoint of $\Phi^{HA}_{P,\delta}$.
($\Leftarrow$) This follows directly from Lemma 5.7.2: take J to be HA, and because HA is a CET-algebra, we have that the fixpoints of $\Phi^{HA}_{P,\delta}$ are subsets of the set of Herbrand models of $comp(P) \cup \{\delta\}$.
($\Rightarrow$) Let M' be an arbitrary Herbrand model of $comp(P) \cup \{\delta\}$, and let Δ be its abducible projection. By Lemma 5.7.5, M' is a fixpoint of $\Phi^{HA}_{P,\Delta}$. Moreover, Δ is an abducible HA-model of $\{\delta\}$. As a result, for some fixpoint $\mathcal{M}'$ of $\Phi^{HA}_{P,\delta}$, $M' \in \mathcal{M}'$. Because $\mathcal{M}$ is the least fixpoint of $\Phi^{HA}_{P,\delta}$, there exists a $M \in \mathcal{M}$ such that $M' \models_3 M$. But then, if ϕ is true in $\mathcal{M}$, it is true in M, and therefore in M', which is what we started with; an arbitrary Herbrand model of $comp(P) \cup \{\delta\}$.
□

5.8 Kunen Semantics for Abductive Logic Programs

In this section, we generalize Kunen semantics to abductive logic programs. In [Kun87], Kunen proposes to cut off iteration of the immediate consequence operator at ordinal ω, instead of continuing until the least fixpoint is reached. Generalizing this idea to abductive logic programming, we get the following semantics.

Definition 5.8.1 Let $\langle P, \phi\rangle$ be an abductive problem. Let δ be a consistent abducible formula. Then, δ is a *proof theoretic explanation* for $\langle P, \phi\rangle$ *in Kunen semantics* if, for some natural number n, $\Phi^{HA}_{P,\delta} \uparrow n \models_3 \phi$. $\square$

Note, that this definition differs from definition 5.4.12. The remainder of this section is dedicated to proving that these two definitions give rise to the same semantics (Theorem 5.8.10). In his proof of Theorem 6.3 in [Kun87], Kunen makes heavy use of ultra-products. We base our proofs on an alternative proof given by K. Doets in [Doe93].

The larger part of the work is done in the proof of Theorem 5.8.2, which proves one direction of the desired result for the operator $\Phi_{P,\Delta}$ (the other direction is not very difficult to prove). Basically, with this result on $\Phi_{P,\Delta}$, we have proven the result for $\Phi_{P,\delta}$, for the case where δ is a conjunction of abducible literals (i.e. it has a minimal model over any algebra). The remainder of the proof of Theorem 5.8.10 is concerned with extending this result to the case where δ is an arbitrary abducible sentence, and proving the other direction of the desired result.

Theorem 5.8.2 *Let P be a program and let ϕ be a sentence. Let δ be a consistent abducible formula and let Δ be an abducible HA-model of $\{\delta\}$. Then, if comp$(P) \cup \{\delta\} \models_3 \phi$, for some natural number n, $\Phi^{HA}_{P,\Delta} \uparrow n \models_3 \phi$.*

The proof of this theorem closely resembles the proof of Corollary 8.37 in [Doe93]. It is organized as follows. In Lemma 5.8.3 we show that we can replace HA with an elementary extension of HA. Then, in Lemma 5.8.7 we show that for certain elementary extensions J of HA, $\Phi^{J}_{P,\Delta}$ is *continuous.* In Lemma 5.8.8 we show that for certain elementary extensions J of HA, $\Phi^{J}_{P,\Delta} \uparrow \omega$ is a least fixpoint. From these lemmas, and from the fact that, by properties 5.8.5 and 5.8.6 stated below (see [CK73]), we know these desired elementary extensions of HA exist, we can prove Theorem 5.8.2.

Lemma 5.8.3 *Let P be a program. Let J be an elementary extension of HA, let Δ be an abducible HA-model and let Δ' be an elementary J-extension of Δ. For every sentence ϕ and natural number n, $\Phi^{HA}_{P,\Delta} \uparrow n \models_3 \phi$ iff $\Phi^{J}_{P,\Delta'} \uparrow n \models_3 \phi$.*

Proof: By Lemma 5.6.8, $\Phi^{HA}_{P,\Delta} \uparrow n \models_3 \phi$ iff $\Delta \models_3 T_n(\phi)$. Because Δ' is an elementary extension of Δ, and $T_n(\phi)$ is a sentence, $\Delta \models_3 T_n(\phi)$ iff $\Delta' \models_3 T_n(\phi)$.

Again by Lemma 5.6.8, $\Delta' \models_3 T_n(\phi)$ iff $\Phi^J_{P,\Delta'} \uparrow n \models_3 \phi$. □

For Lemmas 5.8.7 and 5.8.8, we need the following definitions and results from model theory, concerning recursively saturated models.

Definition 5.8.4 Let $\Psi = \{\psi^i \mid i \in \mathbb{N}\}$ be a sequence of formulae ψ^i in finitely many free variables $x_1, \ldots, x_k, y_1, \ldots, y_m$ and let M be a two-valued model. M is called *Ψ-saturated* if, for every sequence $d_1, \ldots, d_m$ of domain elements, either

- $\{\psi^i\{\mathbf{y}/\mathbf{d}\} \mid i \in \mathbb{N}\}$ is satisfiable in M, or
- there exist a natural number N such that $\{\psi^i\{\mathbf{y}/\mathbf{d}\} \mid i < N\}$ is not satisfiable in M.

M is called ***saturated*** if it is Ψ-saturated for every sequence Ψ. M is called ***recursively saturated*** if it is Ψ-saturated for every computable sequence Ψ. □

Property 5.8.5 *Every countable model has a countable recursively saturated elementary extension.*

Property 5.8.6 *Let $\Psi = \{\psi^i \mid i \in \mathbb{N}\}$ be a sequence of sentences with free variable x. Let M be a recursively saturated model and let D be the domain of M. Then, $\forall_{d \in D} \exists_n M \models \psi^i(d)$ implies $\exists_n \forall_{d \in D} M \models \psi^i(d)$.*

Lemma 5.8.7 *Let P be a program. Let J be a recursively saturated algebra with domain D and let Δ be an abducible J-model. Let ϕ be a D-sentence. If ϕ is* **t** *(resp.* **f***) in $\Phi^J_{P,\Delta} \uparrow \omega$, then, for some natural number n, ϕ is* **t** *(resp.* **f***) in $\Phi^J_{P,\Delta} \uparrow n$.*

Proof: The proof is by induction on the complexity of ϕ. Only when ϕ is of the form $\forall y \psi$ or $\exists y \psi$, the proof is non-trivial, and we can write $\exists y \psi$ as $\neg \forall y \neg \psi$.

Assume that $\forall y \psi$ is **t** in $\Phi^J_{P,\Delta} \uparrow \omega$. Then, for all $d \in D$, $\psi(d)$ is **t** in $\Phi^J_{P,\Delta} \uparrow \omega$. By induction hypothesis, for all $d \in D$, there exists an n such that $\psi(d)$ is **t** in $\Phi^J_{P,\Delta} \uparrow n$. But then, by Lemma 5.6.8, for all $d \in D$, there exists an n such that $T_n(\psi)(d)$ is **t** in Δ. Because J is recursively saturated, by Lemma 5.8.6 there exists an n such that for all $d \in D$ $T_n(\psi)(d)$ is **t** in Δ. But then, $T_n(\forall y \psi)$ is **t** in Δ and therefore by Lemma 5.6.8, $\forall y \psi$ is **t** in $\Phi^J_{P,\Delta} \uparrow n$.

Assume that $\forall y \psi$ is **f** in $\Phi^J_{P,\Delta} \uparrow \omega$. Then, for some $d \in D$, $\psi(d)$ is **f** in $\Phi^J_{P,\Delta} \uparrow \omega$. By induction hypothesis, for some $d \in D$, there exists an n such that $\psi(d)$ is **f** in $\Phi^J_{P,\Delta} \uparrow n$. But then, $\forall y \psi$ is **f** in $\Phi^J_{P,\Delta} \uparrow n$. □

Lemma 5.8.8 *Let P be a program. Let J be a recursively saturated CET-algebra and let Δ be an abducible J-model. Then, $lfp(\Phi^J_{P,\Delta}) = \Phi^J_{P,\Delta} \uparrow \omega$.*

Proof: We have to prove for an arbitrary J-ground atom A that, whenever $\Phi^J_{P,\Delta} \uparrow \omega + 1(A) = \mathbf{t}$, then $\Phi^J_{P,\Delta} \uparrow \omega(A) = \mathbf{t}$, and if $\Phi^J_{P,\Delta} \uparrow \omega + 1(A) = \mathbf{f}$, then $\Phi^J_{P,\Delta} \uparrow \omega(A) = \mathbf{f}$.

For abducible atoms, the claims hold trivially, because then $\Phi^J_{P,\Delta} \uparrow \alpha(A) = \mathbf{t}$ (resp. $\mathbf{f}$) iff $\Delta \models_3 A$ (resp. $\Delta \models_3 \neg A$).

Suppose $p(\mathbf{s})$ is a non-abducible J-ground atom.

- Suppose $p(\mathbf{s})$ is $\mathbf{t}$ in $\Phi^J_{P,\Delta} \uparrow \omega + 1$. Then, there exists a J-ground instance $p(\mathbf{s}) \leftarrow \mathbf{L}$ of a clause in P such that $\Phi^J_{P,\Delta} \uparrow \omega \models_3 \mathbf{L}$. But then by Lemma 5.8.7, there exists a natural number n such that $\Phi^J_{P,\Delta} \uparrow n \models_3 \mathbf{L}$, and therefore $p(\mathbf{s})$ is $\mathbf{t}$ in $\Phi^J_{P,\Delta} \uparrow n + 1$. Thus, $p(\mathbf{s})$ is $\mathbf{t}$ in $\Phi^J_{P,\Delta} \uparrow \omega$.

- Suppose $p(\mathbf{s})$ is $\mathbf{f}$ in $\Phi^J_{P,\Delta} \uparrow \omega + 1$. Let $p(\mathbf{t}_1) \leftarrow \mathbf{L}_1 \ldots p(\mathbf{t}_k) \leftarrow \mathbf{L}_k$ be the clauses defining p. Then, for all $i \in [1..k]$, $\Phi^J_{P,\Delta} \uparrow \omega \models_3 \neg(\mathbf{s} = \mathbf{t}_i \ \wedge \ \mathbf{L}_i)$. Because $\neg(\mathbf{s} = \mathbf{t}_i \ \wedge \ \mathbf{L}_i)$ is quantifier-free, it is equivalent to its universal closure. But for all $i \in [1..k]$, $\forall\neg(\mathbf{s} = \mathbf{t}_i \ \wedge \ \mathbf{L}_i)$ is a D-sentence (where D is the domain of J), and therefore by Lemma 5.8.7 there exists an n_i such that $\Phi^J_{P,\Delta} \uparrow n_i \models_3 \forall\neg(\mathbf{s} = \mathbf{t}_i \ \wedge \ \mathbf{L}_i)$. Because k is finite, there exists an n such that, for all $i \in [1..k]$, we have that $\Phi^J_{P,\Delta} \uparrow n \models \neg(\mathbf{s} = \mathbf{t}_i \ \wedge \ \mathbf{L}_i)$. By construction of $\Phi^J_{P,\Delta}$, we have that $p(\mathbf{s})$ is $\mathbf{f}$ in $\Phi^J_{P,\Delta} \uparrow n + 1$ and therefore, $p(\mathbf{s})$ is $\mathbf{f}$ in $\Phi^J_{P,\Delta} \uparrow \omega$. □

Before proving Theorem 5.8.2, we combine the preceding two lemmas in the following corollary.

Corollary 5.8.9 *Let P be a program and let δ be a consistent abducible formula. Let J be a recursively saturated CET-algebra and let Δ be an abducible J-model of $\{\delta\}$. Let ϕ be a sentence. If comp$(P) \cup \{\delta\} \models_3 \phi$, then for some n $\Phi^J_{P,\Delta} \uparrow n \models_3 \phi$.*

Proof: By Lemma 5.7.2 and Lemma 5.8.8, $\Phi^J_{P,\Delta} \uparrow \omega$ is a three-valued model of *comp*$(P) \cup \{\delta\}$, and therefore $\Phi^J_{P,\Delta} \uparrow \omega \models_3 \phi$. Therefore, by Lemma 5.8.7 there exists a finite n such that $\Phi^J_{P,\Delta} \uparrow n \models_3 \phi$. □

Proof: (of Theorem 5.8.2)
Suppose that *comp*$(P) \cup \{\delta\} \models_3 \phi$. By property 5.8.5, there exists a recursively saturated elementary extension J of *HA*. Because J is an extension of *HA*, it is a *CET*-algebra. Again, by property 5.8.5, there exists an elementary J-extension Δ' of Δ. By Corollary 5.8.9 there exists a finite n such that $\Phi^J_{P,\Delta'} \uparrow n \models_3 \phi$. Finally, by Lemma 5.8.3, $\Phi^{HA}_{P,\Delta} \uparrow n \models_3 \phi$. □

Thus, for $\Phi_{P,\Delta}$, we have proven one direction of the desired result (the other direction is not very difficult to prove, and will therefore be proven directly for $\Phi_{P,\delta}$).

In the following theorem, we prove that the desired correspondence holds for $\Phi_{P,\delta}$.

Theorem 5.8.10 *Let P be a program and let δ be a consistent abducible sentence. Let ϕ be a sentence. Then, δ is a model theoretic explanation for $\langle P, \phi\rangle$ in Kunen semantics iff δ is a proof theoretic explanation for $\langle P, \phi\rangle$ in Kunen semantics.*

Before proving the theorem, we first need to prove two lemmas. The first one states that, in some sense, the operator $\Phi_{P,\delta}$ behaves 'monotonically' with respect to the assumption δ.

Lemma 5.8.11 *Let P be a program and let δ and σ be consistent abducible formulae. Let J be an algebra. If $J \models_3 \forall\delta \rightarrow \forall\sigma$ then, for all natural numbers n, $\Phi^J_{P,\delta} \uparrow n \models_3 \Phi^J_{P,\sigma} \uparrow n$.*

Proof: It suffices to prove that, for all natural numbers n, $M \in \Phi^J_{P,\delta} \uparrow n$ implies $M \in \Phi^J_{P,\sigma} \uparrow n$.

Suppose that $M \in \Phi^J_{P,\delta} \uparrow n$. Then, for some abducible J-model Δ of $\{\delta\}$, $M = \Phi^J_{P,\Delta} \uparrow n$. But because $J \models_3 \forall\delta \rightarrow \forall\sigma$, Δ is also an abducible J-model of $\{\sigma\}$. Therefore, $M \in \Phi^J_{P,\sigma} \uparrow n$. □

Lemma 5.8.12 *Let P be a program and let δ be a consistent abducible formula. Let ϕ be a sentence and let J be a recursively saturated CET-algebra. Then, comp$(P) \cup \{\delta\} \models_3 \phi$ implies that, for some finite n, $\Phi^J_{P,\delta} \uparrow n \models_3 \phi$.*

Proof: *comp*$(P) \cup \{\delta\} \models_3 \phi$ implies that *comp*$(P) \models_3 \forall\delta \rightarrow \forall\phi$. Let σ be an abducible formula which is a tautology, and let Δ be the least abducible J model of σ. By Corollary 5.8.9, there exists a finite n for which we have that $\Phi^J_{P,\Delta} \uparrow n \models_3 \forall\delta \rightarrow \forall\phi$. Because Δ is the least abducible J-model of $\{\sigma\}$, $\Phi^J_{P,\sigma} \uparrow n \models_3 \forall\delta \rightarrow \forall\phi$ iff $\Phi^J_{P,\Delta} \uparrow n \models_3 \forall\delta \rightarrow \forall\phi$. Moreover, because $J \models_3 \delta \rightarrow \sigma$, by Lemma 5.8.11 it follows that $\Phi^J_{P,\delta} \uparrow n \models_3 \forall\delta \rightarrow \forall\phi$. Finally, because we know that $\Phi^J_{P,\delta} \uparrow n \models_3 \delta$, it follows that $\Phi^J_{P,\delta} \uparrow n \models_3 \phi$. □

Proof: (of Theorem 5.8.10)
We have to prove that *comp*$(P) \cup \{\delta\} \models_3 \phi$ iff, for some finite n, $\Phi^{HA}_{P,\delta} \uparrow n \models_3 \phi$.
($\Rightarrow$) Suppose that *comp*$(P) \cup \{\delta\} \models_3 \phi$. By property 5.8.5, there exists a recursively saturated elementary extension J of *HA*. Because J is an extension of *HA*, it is a *CET*-algebra. By Lemma 5.8.12 there exists an n such that $\Phi^J_{P,\delta} \uparrow n \models_3 \phi$. Let Δ be an arbitrary abducible *HA*-model of $\{\delta\}$. By property 5.8.5, there exists an elementary J-extension Δ' of Δ. Because Δ' is an elementary extension of Δ, δ is a sentence and $\Delta \models_3 \delta$, it follows that $\Delta' \models_3 \delta$. Therefore, it follows from $\Phi^J_{P,\delta} \uparrow n \models_3 \phi$ that $\Phi^J_{P,\Delta'} \uparrow n \models_3 \phi$. But then, by

Lemma 5.8.3, $\Phi^{HA}_{P,\Delta} \uparrow n \models_3 \phi$. Thus, for arbitrary Herbrand models Δ of δ, we have that $\Phi^{HA}_{P,\Delta} \uparrow n \models_3 \phi$. But then, also $\Phi^{HA}_{P,\delta} \uparrow n \models_3 \phi$.

($\Leftarrow$) Suppose that ϕ is an abducible formula. Then, by definition of $\Phi^{HA}_{P,\Delta}$, $\Phi^{HA}_{P,\delta} \uparrow n \models_3 \phi$ iff $HA \cup \{\delta\} \models_3 \phi$. Because δ and ϕ are sentences and every model of *CET* is an extension of a Herbrand model, $CET \cup \{\delta\} \models_3 \phi$ and therefore $comp(P) \cup \{\delta\} \models_3 \phi$.

Suppose that ϕ is not an abducible formula. We prove the claim by by induction on n and formula induction on ϕ. For $n = 0$ and ϕ a J-ground literal, the claim holds trivially, because $\Phi^{HA}_{P,\delta} \uparrow 0 \not\models_3 \phi$.

Assume that the claim holds for all $m < n$. Let $p(\mathbf{s})$ be a non-abducible J-ground atom. Then,

$$\begin{array}{ll}
 & \Phi^{J}_{P,\delta} \uparrow n \models_3 p(\mathbf{s}) \\
 & \text{by definition of } \Phi^{J}_{P,\delta} \\
\text{iff} & \exists\, p(\mathbf{s}) \leftarrow \mathbf{L} \in J\text{-}ground(P) : \Phi^{J}_{P,\delta} \uparrow n-1 \models_3 \mathbf{L} \\
 & \text{by induction hypothesis} \\
\text{then} & \exists\, p(\mathbf{s}) \leftarrow \mathbf{L} \in J\text{-}ground(P) : comp(P) \cup \{\delta\} \models_3 \mathbf{L} \\
 & \text{by definition of completion} \\
\text{iff} & comp(P) \cup \{\delta\} \models_3 p(\mathbf{s})
\end{array}$$

The case for $\neg p(\mathbf{s})$ is similar.

For complex sentences, the proof is by formula induction. □

Thus, we have proven that definitions 5.4.12 and 5.8.1 give rise to the same semantics. In the following section, we present a proof procedure for this semantics.

5.9 Generalizing SLDFA-Resolution

In [DS92], M. Denecker and D. DeSchreye propose SLDNFA-resolution, a proof procedure for abductive logic programs based on SLDNF-resolution. This proof procedure is sound with respect to the two-valued completion semantics of [CDT91]. In this chapter, we propose an alternative proof procedure, which is based upon SLDFA-resolution; a proof procedure for normal logic programs proposed by W. Drabent [Dra95]. This proof procedure solves some problems associated with SLDNFA-resolution. First of all, by using constructive negation instead of negation as failure, we remove the problem of *floundering*. Secondly, instead of skolemizing non-ground queries, which introduces some technical problems, we use equality in our language, which allows a natural treatment of non-ground queries.

In the last few years, various forms of constructive negation have been proposed (see for instance [Cha88, Stu91, Dra95, Dra93b, Fag]), to deal with the problem of *floundering* in SLDNF-resolution. In [Dra95], W. Drabent introduces SLDFA-resolution, a proof procedure for normal logic programs based on SLD-resolution and constructive negation, proves that it is sound and complete with respect to Kunen's three-valued completion semantics, and sound with respect to two-valued completion semantics.

In this section we generalize SLDFA-resolution to abductive logic programs. The main difference with the definition given in [Dra95] is, that the answers we compute are abducible formulae instead of constraints. As a result, most definitions in this section are direct copies of definitions in [Dra95]. Apart from the fact that we use *queries* instead of *goals*, only the definition of *query* is slightly different from the definition of *goal* in [Dra95].

The basic idea of using constructive negation in proof procedures for normal logic programming is, that computed answers to normal queries are *equality constraints*, i.e. first-order formulae build out of the equality predicate '='. This notion of computed answer generalizes the notion of computed answer substitutions, because a substitution can be written as a conjunction of primitive equalities. Instead of using equality constraints as computed answers, we use abducible formulae. If we only look at their definition, we see that abducible formulae are a generalization of equality formulae. However, there is a difference in the meaning of an abducible formula when it is used as a computed answer. When using an equality constraint θ as computed answers, one requires it to be *satisfiable* in CET, i.e. $CET \models \exists\theta$. However, when the computed answer is an abducible formula, there is no theory with respect to which one can require it to be satisfiable. The only requirement for such a computed answer is, that it is consistent. Therefore, we require consistency instead of satisfiability. As our abducible formulae can contain equality predicates, we require our computed answers to be consistent with respect to CET. This consistency requirement for abducible formulae generalizes the satisfiability requirement for equality constraints, whenever a universal language is used.

Lemma 5.9.1 *Let θ be an equality constraint. Then, θ is satisfiable in $CET_{\mathcal{L}_\mathcal{U}}$ iff $CET_{\mathcal{L}_\mathcal{U}} \cup \{\theta\}$ is consistent.*

Proof: The lemma follows directly from the fact that $CET_{\mathcal{L}_\mathcal{U}}$ is a complete theory. □

We do not concern ourselves with reducing abducible formulae to normal forms. We simply assume the existence of normalization procedures that transform a given abducible formula into a format that is intelligible to humans.

SLDFA-resolution is defined by two basic notions: *SLDFA-refutations* and *(finitely failed) SLDFA-trees*. An *SLDFA-refutation* is a sequence of queries, ending in a query without non-abducible atoms, such that each query in the sequence is obtained from the previous query by a *positive* or *negative derivation step*. A *positive derivation step* is the usual one used in SLD-resolution, with the difference that the resolved atom has to be a non-abducible atom. A *negative derivation step* is the replacement of a negative non-abducible literal $\neg A$ in the query by an abducible formula σ such that σ, A is guaranteed to fail finitely. A *finitely failed SLDFA-tree* for a query Q is a proof for the fact that Q fails finitely; it is an approximation that is 'safe' with respect to finite failure; if a finitely failed SLDFA-tree for Q exists, it is guaranteed that Q fails finitely, but the fact that there exists an SLDFA-tree for Q that is not finitely failed, does not imply that Q is not finitely failed.

Before we can define SLDFA-resolution, we have to define the notion of a *query*.

Definition 5.9.2 Let P be a program. A *query* (with respect to P) is a formula of the form $\theta, L_1, \ldots, L_k$, such that

- θ is a consistent abducible formula, and
- L_i (for $i \in [1..k]$) is a non-abducible literal.

An *s-query* is a query in which one of the literals is marked as *selected.* $\square$

We begin the definition of SLDFA-resolution with the definition of *positively derived queries.*

Definition 5.9.3 Let P be a program, let Q be the s-query $\theta, \mathbf{N}, p(\mathbf{t}), \mathbf{M}$ (with $p(\mathbf{t})$ selected) and let $p(\mathbf{s}) \leftarrow \sigma, \mathbf{L}$ be a variant of a clause in P. A query Q' is *positively derived* from Q using $p(\mathbf{s}) \leftarrow \sigma, \mathbf{L}$ if

- $\mathit{FreeVar}(Q) \cap \mathit{FreeVar}(p(\mathbf{s}) \leftarrow \sigma, \mathbf{L}) = \emptyset$ and
- Q' is of the form $\theta, (\mathbf{t} = \mathbf{s}), \sigma, \mathbf{N}, \mathbf{L}, \mathbf{M}$.

If Q' is positively derived from Q using a variant of a clause c, we call c *applicable* to Q. $\square$

Note that the abducible formula in Q' is (by definition) consistent because Q' is (by definition) a query, and by definition the abducible formula in a query is consistent.

Let us now give the definitions of *negatively derived queries*, *finitely failed queries*, *(finitely failed) SLDFA-trees*, and *SLDFA-refutations*. These definitions are mutually recursive. Therefore, we define them inductively, using the notion of *rank*.

Definition 5.9.4 Let P be a program and let Q be the s-query $\theta, \mathbf{N}, \neg A, \mathbf{M}$ (with $\neg A$ selected). Let the notion of *rank k finitely failed queries* be defined. A query Q' is *rank k negatively derived* from Q if

- Q' is of the form $\theta, \sigma, \mathbf{N}, \mathbf{M}$,
- θ, σ, A is a rank k finitely failed query, and
- $\mathit{FreeVar}(\sigma) \subseteq \mathit{FreeVar}(A)$.

We call θ, σ a *(rank k) fail answer* for θ, A. $\square$

Definition 5.9.5 Let P be a program and let Q be a goal. Let the notion of ***rank k finitely failed SLDFA-tree*** be defined. Q is a ***rank k finitely failed query*** if there exists a rank k finitely failed SLDFA-tree for Q. □

Definition 5.9.6 Let P be a program and let Q be a query. Let the notion of *rank k SLDFA-refutation* be defined. A *rank k SLDFA-tree* for Q is a tree such that

1. each node of the tree is an s-query and the query part of the root node is Q,
2. the tree is finite,
3. if $R: \theta, \mathbf{L}_1, A, \mathbf{L}_2$ (with A selected) is a node in the tree then, for every clause c in P applicable to R, there exists exactly one son of R that is positively derived from R using a variant of c, and
4. if $R: \theta, \mathbf{L}_1, \neg A, \mathbf{L}_2$ (with $\neg A$ selected) is a node in the tree, then it has sons
$$\sigma_1, \mathbf{L}_1, \mathbf{L}_2\ ,\ \ldots\ ,\ \sigma_m, \mathbf{L}_1, \mathbf{L}_2$$
provided there exist $\delta_1, \ldots, \delta_n$ that are SLDFA-computed answers obtained by rank k SLDFA-refutations of θ, A, such that
$$CET \models \theta \rightarrow \delta_1 \vee \ldots \vee \delta_n \vee \sigma_1 \vee \ldots \vee \sigma_m$$

If no node in an SLDFA-tree is of the form θ, then that tree is called *finitely failed.* □

Definition 5.9.7 Let P be a program and let Q be a query. Let the notion of *rank $k-1$ negatively derived s-query* be defined. A *rank k SLDFA-refutation* of Q is a sequence of s-queries $Q_0, \ldots, Q_n$ such that Q is the query part of Q_0, Q_n is of the form θ and, for $i \in [1..n]$,

- Q_i is positively derived from Q_{i-1} using a variant c of a clause in P such that
$\mathit{FreeVar}(c) \cap \mathit{FreeVar}(Q_0, \ldots, Q_{i-1}) = \emptyset$, or
- Q_i is rank $k-1$ negatively derived from Q_{i-1}.

The abducible formula $\exists \mathbf{y}\theta$, where $\mathbf{y} = \mathit{FreeVar}(\theta) - \mathit{FreeVar}(Q)$, is a *SLDFA-computed answer* for Q. □

To get some insight in the construction of SLDFA-refutations, let us conclude with an example

Example 5.9.8 *Consider program P_{Tweety} and the observation $\neg$flies(tweety). In Figure 5.2 we show the SLDFA-refutation for this query in R_1. Let us see*

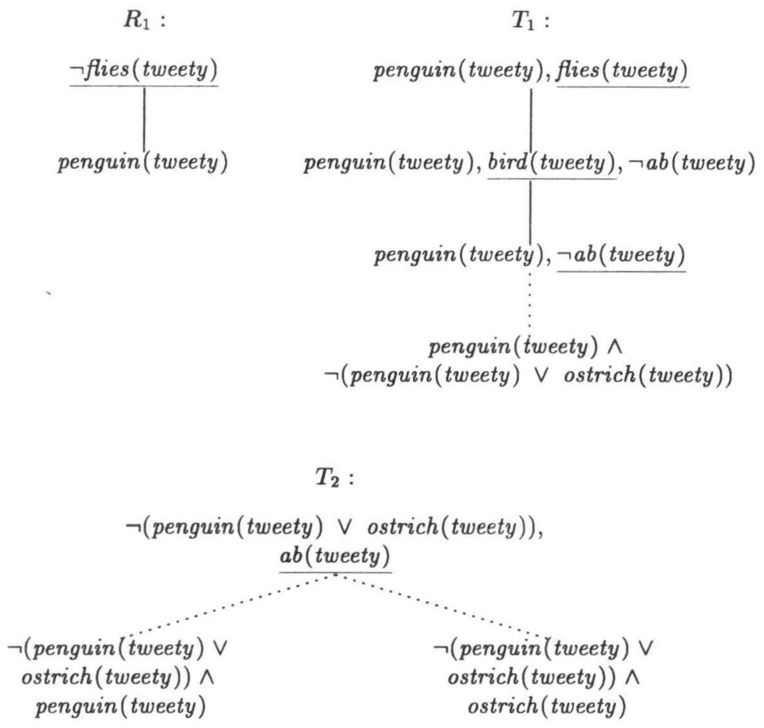

R₁ :
¬flies(tweety)
penguin(tweety)
T₁ :
penguin(tweety), flies(tweety)
penguin(tweety), bird(tweety), ¬ab(tweety)
penguin(tweety), ¬ab(tweety)
penguin(tweety) ∧ ¬(penguin(tweety) ∨ ostrich(tweety))
T₂ :
¬(penguin(tweety) ∨ ostrich(tweety)), ab(tweety)
¬(penguin(tweety) ∨ ostrich(tweety)) ∧ penguin(tweety)
¬(penguin(tweety) ∨ ostrich(tweety)) ∧ ostrich(tweety)

Figure 5.2: An SLDFA-refutation for $\neg \mathit{flies}(\mathit{tweety})$

how this refutation is constructed. First of all, consider T_2 which is a finitely failed SLDFA-tree for

$$\neg(penguin(tweety) \lor ostrich(tweety)), ab(tweety)$$

The two dotted lines in the tree lead to two 'queries' which are not valid resolvents, because their constraint part is inconsistent. Thus, the root of T_2 is a finitely failed query. Secondly, we have T_1 is a finitely failed SLDFA-tree for

$$penguin(tweety), flies(tweety)$$

This tree is finitely failed, because applying the finite fail answer obtained by T_2 in a (negative) resolution step with $\neg ab(tweety)$ results in a 'query' with an inconsistent constraint (see the dotted line). Because, the finitely failed query in T_2 is most general, it follows that the third query in T_1 has no resolvents. Thus, the root of T_1 is a finitely failed query. This fact is used in the construction of the SLDFA-refutation R_1. ∘

5.10 Soundness of SLDFA-Resolution

In this section we present some soundness results on SLDFA-resolution for abductive logic programs. We start by proving soundness with respect to three-valued completion semantics for abductive logic programs.

Theorem 5.10.1 *Let P be a program and let Q be the query $\theta, \mathbf{L}$.*

1. *If δ is an SLDFA-computed answer for Q then $comp(P) \models_3 \delta \rightarrow \theta \land \mathbf{L}$.*
2. *If Q finitely fails then $comp(P) \models_3 \theta \rightarrow \neg\mathbf{L}$.*

The proof of this theorem closely resembles the proof of Theorem 4.2 in in [Dra95]. The differences between the two proofs are, that here we prove soundness with respect to three-valued completion semantics, while Drabent's proof proves soundness with respect to two-valued completion, and that we work with abductive formulae instead of constraints.

Before giving the proof of the theorem, we first prove the following lemma.

Lemma 5.10.2 *Let P be a program and let Q be a query with a positive literal selected. Let ${Q'}_1, \ldots, {Q'}_n$ $(n \geq 0)$ be the set of all queries positively derived from Q in P and let, for $i \in [1..n]$, $\mathbf{x'}_i = FreeVar(Q_i) - FreeVar(Q)$. Then,*

$$comp(P) \models_3 Q \cong \exists_{\mathbf{x'}_1} Q'_1 \ \lor \ \ldots \ \lor \ \exists_{\mathbf{x'}_n} Q'_n$$

Proof: Let Q be of the form $\theta, L_1, \ldots, L_k$. Let us assume, without loss of generality, that the left-most literal (i.e. L_1) is selected. Let L_1 be the atom $p(\mathbf{s})$ and let $\mathbf{L}'$ denote the sequence $L_2, \ldots, L_k$. Let

$$c_1 : p(\mathbf{t}_1) \leftarrow \sigma_1, \mathbf{M}_1 \ \ \ldots \ \ c_m : p(\mathbf{t}_m) \leftarrow \sigma_m, \mathbf{M}_m$$

contain a variant for each clause in P with head p. Assume that these clauses are standardized apart from each other and from $\mathbf{L}$. Finally, let, for $i \in [1..m]$, $\mathbf{y}_i = \mathit{FreeVar}(\sigma_i, \mathbf{M}_i) - \mathit{FreeVar}(p(\mathbf{t}_i))$.

By definition of $comp(P)$, we have that

$$comp(P) \models_3 p(\mathbf{z}) \cong \bigvee_{i \in [1..m]} \exists_{\mathbf{y}_i}((\mathbf{z} = \mathbf{t}_i), \sigma_i, \mathbf{M}_i)$$

(where the $\mathbf{z}$ do not appear in $c_1, \ldots, c_m, \theta, \mathbf{L}$). But then, we also have that

$$comp(P) \models_3 \theta, \mathbf{L} \cong \bigvee_{i \in [1..m]} \exists_{\mathbf{y}_i}(\theta, (\mathbf{s} = \mathbf{t}_i), \sigma_i, \mathbf{M}_i, \mathbf{L}')$$

By definition of positively derived query, for all $i \in [1..n]$ there exists an $j \in [1..m]$ such that Q'_i is a variant of $\theta, (\mathbf{s} = \mathbf{t}_j), \sigma_j, \mathbf{M}_j, \mathbf{L}'$ under renaming $\{\mathbf{x}'_i/\mathbf{y}_j\}$. Moreover, for each c_i for which there is not a query Q'_j such that Q'_j is derived from Q using a variant of c_i, $\theta, (\mathbf{s} = \mathbf{t}_i), \sigma_i$ is inconsistent (with respect to CET), and therefore

$$comp(P) \models_3 \exists_{\mathbf{y}_i}(\theta, (\mathbf{s} = \mathbf{t}_i), \sigma_i, \mathbf{M}_i, \mathbf{L}') \cong \mathbf{f}$$

But then,

$$comp(P) \models_3 Q \cong \exists_{\mathbf{x}_1} Q'_1 \lor \ldots \lor \exists_{\mathbf{x}_n} Q'_n$$

□

Proof: (of Theorem 5.10.1)
We prove by complete induction over k that for all natural numbers k and all queries Q of the form $\theta, \mathbf{L}$ that

1. if δ is a computed answer of a rank k SLDFA-refutation for Q, then $comp(P) \models_3 \delta \rightarrow \theta \land \mathbf{L}$, and
2. if Q is a rank k finitely failed query, then $comp(P) \models_3 \theta \rightarrow \neg\mathbf{L}$.

Assume that 1. and 2. hold for all ranks smaller than k. We first prove 1. for rank k, using the induction hypothesis, and then prove 2. for rank k, using the fact that we already have proven 1. for rank k.
(1.) Suppose that Q has a rank k SLDFA-refutation $Q'_0, \ldots, Q'_n$ with computed answer δ. Let, for $i \in [1..n]$, $\mathbf{x}'_i = \mathit{FreeVar}(Q'_i) - \mathit{FreeVar}(Q'_{i-1})$, and let $\mathbf{z} = \mathit{FreeVar}(Q'_n) - \mathit{FreeVar}(Q'_0)$. For a moment, let us assume that, for all $i \in [1..n]$, $comp(P) \models_3 Q'_{i-1} \leftarrow \exists_{\mathbf{x}'_i} Q'_i$. Then, it follows by straightforward induction that

$$comp(P) \models_3 Q'_0 \leftarrow \exists_{\mathbf{z}} Q'_n$$

Because $Q'_0, \ldots, Q'_n$ is a refutation of Q, we have that Q'_0 is of the form $\theta, \mathbf{L}$ and Q'_n is of the form τ. But then, it follows that

$$comp(P) \models_3 \exists_{\mathbf{z}}\tau \rightarrow \theta, \mathbf{L}$$

Now, δ is a variant of $\exists \mathbf{z}\tau$, and therefore

$$comp(P) \models_3 \delta \rightarrow \theta, \mathbf{L}$$

So, to prove 1. for rank k, it is sufficient to prove that, for all $i \in [1..n]$, $comp(P) \models_3 Q'_{i-1} \leftarrow \exists \mathbf{x'}_i Q'_i$. Because every Q'_i is either positively or negatively derived from Q'_{i-1}, there are two cases:

- Suppose that Q'_i is positively derived from Q'_{i-1}. Then, by Lemma 5.10.2,

  $$comp(P) \models_3 Q'_i \cong \exists_{\mathbf{x''}_1} Q''_1 \ \vee \ \dots \ \vee \ \exists_{\mathbf{x''}_m} Q''_m$$

  where $Q''_1, \dots, Q''_m$ contains all queries that are positively derivable from Q'_i and $\mathbf{x''}_i = \mathit{FreeVar}(Q''_i) - \mathit{FreeVar}(Q'_i)$. Because Q'_i is positively derivable from Q'_{i-1}, for some $j \in [1..m]$, $\exists_{\mathbf{x'}_i} Q'_i$ is a variant of $\exists_{\mathbf{x''}_j} Q''_j$, and therefore

  $$comp(P) \models_3 Q'_{i-1} \leftarrow \exists_{\mathbf{x'}_i} Q'_i$$

- Suppose that Q'_i is negatively derivable from Q'_{i-1}. Then, Q'_{i-1} is of the form $\rho, \neg A, \mathbf{L'}$ (we assume without loss of generality that the left-most literal is selected) and Q'_i is of the form $\rho, \sigma, \mathbf{L'}$, such that ρ, σ, A has a rank $k-1$ finitely failed SLDFA-tree. By induction hypothesis (part 2.), it follows that

  $$comp(P) \models_3 \rho \ \wedge \ \sigma \rightarrow \neg A$$

  But then, we also have that

  $$comp(P) \models_3 \rho \ \wedge \ \sigma \ \wedge \ \mathbf{L'} \rightarrow \rho \ \wedge \ \neg A \ \wedge \ \mathbf{L'}$$

  which is equivalent to

  $$comp(P) \models_3 Q'_{i-1} \leftarrow \exists_{\mathbf{x'}_i} Q'_i$$

Thus, we have proven 1. for arbitrary queries with rank k SLDFA-computed answers.
(2.) Suppose that $\theta, \mathbf{L}$ is the root of a rank k finitely failed SLDFA-tree. We have to prove that $comp(P) \models_3 \theta \rightarrow \neg \mathbf{L}$. We prove this by complete induction over the depth l of rank k finitely failed SLDFA-trees. Assume that the claim holds for all rank k finitely failed SLDFA-trees with a depth smaller than l. Now, suppose we have a rank k finitely failed SLDFA-tree of depth l, where the root Q has the form $\leftarrow \theta, L_1, \dots, L_k$. Let us assume, without loss of generality, that the left-most literal (i.e. L_1) is selected. Let the sons of Q be

$$Q_1 : \sigma_1, \mathbf{M}_1 \ \dots \ Q_n : \sigma_n, \mathbf{M}_n$$

Each Q_i is the root of a rank k finitely failed SLDFA-tree of depth $l-1$. Therefore, by induction hypothesis, for all $i \in [1..n]$,

$$comp(P) \models_3 \sigma_i \rightarrow \neg \mathbf{M}_i$$

which can be rewritten as

$$comp(P) \models_3 \neg Q_i$$

Now, there are two cases:

- Suppose that L_1 is the positive literal A. Then, by definition, for every clause c in P that is applicable to A, Q has exactly one son which is positively derived from Q using a variant of c. By Lemma 5.10.2 it follows that

  $$comp(P) \models_3 Q \cong \exists_{\mathbf{x}_1} Q_1 \ \vee \ \ldots \ \vee \ \exists_{\mathbf{x}_n} Q_n$$

  Because for all $i \in [1..n]$, $comp(P) \models_3 Q_i$, we have that $comp(P) \models_3 \neg Q$. Which can be rewritten as $comp(P) \models_3 \theta \rightarrow \neg \mathbf{L}$.

- Suppose that L_1 is the negative literal $\neg A$. Then, by definition, there exist rank k SLDFA-computed answers $\delta_1, \ldots, \delta_m$ for $\leftarrow \theta, A$ such that

  $$CET \models_3 \theta \rightarrow \delta_1 \ \vee \ \ldots \ \vee \ \delta_m \ \vee \ \sigma_1 \ \vee \ \ldots \ \vee \ \sigma_n$$

  We can make the following two observations.

  - We have proven 1 for rank k. It follows therefore that, for all $i \in [1..m]$, $comp(P) \models_3 \delta_i \rightarrow A$ and therefore $comp(P) \models_3 \delta_i \rightarrow \neg \mathbf{L}$.
  - For all $i \in [1..n]$, $comp(P) \models_3 \sigma_i \rightarrow \neg \mathbf{M}_i$, and, by definition, $\mathbf{M}_i$ is of the form $L_2, \ldots, L_k$. Therefore, $comp(P) \models_3 \sigma_i \rightarrow \neg \mathbf{L}$.

  But then, it follows that $comp(P) \models_3 \theta \rightarrow \neg \mathbf{L}$. □

The following corollary proves soundness of SLDFA-resolution with respect to the three-valued completion semantics for abductive logic programs, as stated in definition 5.4.12.

Corollary 5.10.3 (Three-valued Soundness) *Let P be a program and let Q be the query $\theta, \mathbf{L}$. If δ is an SLDFA-computed answer for Q, then δ is a three-valued explanation for $\langle P, Q \rangle$.*

Proof: Because δ is an SLDFA-computed answer for Q, by Theorem 5.10.1, $comp(P) \models_3 \delta \rightarrow \theta \ \wedge \ \mathbf{L}$. Moreover, δ has a 3-valued model, which implies that $comp(P) \cup \{\delta\}$ is consistent. But then, $comp(P) \cup \{\delta\} \models_3 \theta \ \wedge \ \mathbf{L}$. Thus, δ is a three-valued explanation for $\langle P, \theta \ \wedge \ \mathbf{L} \rangle$. □

Now that we have proven soundness with respect to three-valued completion semantics, the following result is straightforward.

Theorem 5.10.4 *Let P be a program and let $Q : \ \theta, \mathbf{L}$ be a query.*

1. *If δ is an SLDFA-computed answer for Q then $comp(P) \models \delta \rightarrow \theta \wedge \mathbf{L}$.*

2. *If Q finitely fails then $comp(P) \models \theta \rightarrow \neg\mathbf{L}$.*

Proof:

1. Suppose that δ is an SLDFA-computed answer for Q. Then, by Theorem 5.10.1, $comp(P) \models_3 \delta \rightarrow \theta \wedge \mathbf{L}$. But we know that every two-valued model for $comp(P)$ is also a three-valued model for $comp(P)$, and therefore $comp(P) \models \delta \rightarrow \theta \wedge \mathbf{L}$.

2. Suppose that Q finitely fails. Then, by Theorem 5.10.1, we have that $comp(P) \models_3 \theta \rightarrow \neg\mathbf{L}$. But every two-valued model for $comp(P)$ is also a three-valued model for $comp(P)$, and therefore $comp(P) \models \theta \rightarrow \neg\mathbf{L}$. □

Using this theorem, we can prove the following soundness result with respect to two-valued completion semantics.

Corollary 5.10.5 (Two-valued Soundness) *Let P be a program and let Q be the query $\theta, \mathbf{L}$. If δ is an SLDFA-computed answer for Q and $comp(P) \cup \{\delta\}$ is consistent, then δ is a three-valued explanation for $\langle P, \theta \wedge \mathbf{L}\rangle$.*

Proof: Because δ is an SLDFA-computed answer for Q, by Theorem 5.10.4, $comp(P) \models \delta \rightarrow \theta \wedge \mathbf{L}$. But then, because $comp(P) \cup \{\delta\}$ is consistent, we have that $comp(P) \cup \{\delta\} \models \theta \wedge \mathbf{L}$. Thus, δ is a three-valued explanation for $\langle P, \theta \wedge \mathbf{L}\rangle$. □

5.11 Completeness of SLDFA-Resolution

In this section, we prove completeness of the generalized SLDFA-resolution with respect to three-valued completion semantics.

Theorem 5.11.1 *Let P be a program and let $Q : \theta, \mathbf{L}$ be a query. Let δ be an abducible sentence. Then, for an arbitrary fair selection rule,*

1. *if $comp(P) \cup \{\delta\} \models_3 \theta \wedge \mathbf{L}$, then there exist computed answers $\delta_1, \dots, \delta_n$ for Q such that $CET \models_3 \delta \rightarrow \delta_1 \vee \dots \vee \delta_n$, and*

2. *if $comp(P) \models_3 \theta \rightarrow \neg\mathbf{L}$ then Q fails finitely.*

As was the case with Theorem 5.10.1, the proof of this theorem is (almost) identical to the proof of the corresponding theorem in [Dra95] (Theorem 5.1). The only difference is, that we use results from Section 5.5, where Drabent used results from [Kun87].

Before giving the proof of the theorem, we present three (technical) lemmas.

Lemma 5.11.2 *Let P be a program, let $\theta, \mathbf{L}$ be a rank k finitely failed query and let σ be a consistent abducible formula. If $CET \models_3 \sigma \rightarrow \theta$, then $\leftarrow \sigma, \mathbf{L}$ is a rank k finitely failed query.*

Proof: Suppose that $\theta, \mathbf{L}$ has a rank k finitely failed SLDFA-tree. Then, there exists a rank k finitely failed SLDFA-tree for $\theta, \mathbf{L}$ such that, for all variables x occurring in σ but not in $\theta, \mathbf{L}$, x does not occur in that SLDFA-tree. From this SLDFA-tree, we can construct a rank k finitely failed SLDFA-tree for $\theta, \sigma, \mathbf{L}$, by adding σ to every node in the tree and then pruning subtrees whose roots contain an inconsistent abducible formula. Because $CET \models_3 \sigma \rightarrow \theta$, the resulting tree is also a rank k finitely failed SLDFA-tree for $\leftarrow \sigma, \mathbf{L}$. □

Lemma 5.11.3 *Let P be a program. Let δ be a computed answer for $\theta, \mathbf{L}$. Then, for any abducible formula σ such that σ, δ is consistent, σ, δ is (equivalent to) an SLDFA-computed answer for $\sigma, \theta, \mathbf{L}$.*

Proof: Suppose that $\theta, \mathbf{L}$ has an SLDFA-refutation. Then, it also has an SLDFA-refutation ${Q'}_0, \ldots, {Q'}_n$ such that, for all variables x occurring in σ but not in $\theta, \mathbf{L}$, x does not occur in${Q'}_0, \ldots, {Q'}_n$. Now let, for $i \in [0..n]$, Q'_i be of the form $\theta_i, \mathbf{L}_i$ and Q''_i of the form $\sigma, \theta_i, \mathbf{L}_i$. Let $\mathbf{y}' = \mathit{FreeVar}(\theta_n) - \mathit{FreeVar}(\theta, \mathbf{L})$. Then, δ is of the form $\exists_{\mathbf{y}'} \theta_n$. We prove that ${Q''}_0, \ldots, {Q''}_n$ is an SLDFA-refutation of $\sigma, \theta, \mathbf{L}$, and that its computed answer is equivalent to σ, δ.

To prove that ${Q''}_0, \ldots, {Q''}_n$ is an SLDFA-refutation, it suffices to prove that, for all $i \in [0..n]$, σ, θ_i is consistent. We know that $\sigma \wedge \exists_{\mathbf{y}'} \theta_n$ is consistent and that θ_n is consistent. Because $\mathbf{y}'$ only quantifies variables that do not occur in σ, it follows that σ, θ_n is consistent. Now, assume that σ, θ_i is consistent. Because Q'_i is (positively or negatively) derived from Q'_{i-1}, we have by the definition of positively and negatively derived queries that $CET \models_3 \theta_i \rightarrow \theta_{i-1}$. From this, and the fact that σ, θ_i is consistent, it follows that σ, θ_{i-1} is consistent. Thus, ${Q''}_0, \ldots, {Q''}_n$ is an SLDFA-refutation.

Now, Q''_n is of the form σ, θ_n. Let $\mathbf{y}'' = \mathit{FreeVar}(\sigma, \theta_n) - \mathit{FreeVar}(\sigma, \theta, \mathbf{L})$. Then, $\exists_{\mathbf{y}''}(\sigma, \theta_n)$ is a computed answer for $\sigma, \theta, \mathbf{L}$. Because σ occurs in $\sigma, \theta, \mathbf{L}$, the variables in $\mathbf{y}''$ do not occur in σ, and therefore

$$\exists_{\mathbf{y}''}(\sigma, \theta_n) \cong \sigma \wedge \exists_{\mathbf{y}''} \theta_n \cong \sigma, \delta$$

□

The following lemma forms the core of the proof of Theorem 5.11.1. In the lemma, and in the proof of the theorem, we use the following notation.

Definition 5.11.4 Let $L_1, \ldots, L_m$ be a sequence of literals, and let $n_1, \ldots, n_m$ be a sequence of natural numbers. Then

$$T_{n_1, \ldots, n_m}(L_1, \ldots, L_m) \overset{def}{=} T_{n_1}(L_1) \wedge \ldots \wedge T_{n_m}(L_m)$$

□

Note, that by definition of T_n and F_n, for a sequence $L_1, \ldots, L_k$ of literals, $T_n(\mathbf{L}) \cong T_{\mathbf{n}}(\mathbf{L})$ and $F_n(\mathbf{L}) \cong F_{\mathbf{n}}(\mathbf{L})$, where $\mathbf{n}$ is the sequence $n, \ldots, n$ of length k.

Lemma 5.11.5 *Let $L_1, \ldots, L_k$ be a sequence of non-abducible literals and let $n_1, \ldots, n_k$ be a sequence of natural numbers. Then, for arbitrary fair selection rules,*

1. *There exist computed answers $\delta_1, \ldots, \delta_l$ for $\leftarrow \mathbf{L}$ such that*

$$CET \models_3 T_{\mathbf{n}}(\mathbf{L}) \rightarrow \delta_1 \ \vee \ \ldots \ \vee \ \delta_l$$

2. *For any sequence $\mathbf{M}$ of literals, either $F_{\mathbf{n}}(\mathbf{L}), \mathbf{L}, \mathbf{M}$ fails or $F_{\mathbf{n}}(\mathbf{L})$ is inconsistent.*

Proof: The proof of the two claims is by induction on $n_1, \ldots, n_k$, using the multiset ordering. For the base case, where $\mathbf{n} = 0$ ($k = 1$), the two claims are trivially true, because $T_0(L) = F_0(L) = \mathbf{f}$ holds for arbitrary non-abducible literals L.

Assume that we have proven the two claims for all $\mathbf{L}'$ and $\mathbf{n}'$ such that $\mathbf{n}'$ is smaller than $\mathbf{n}$ (in the multiset order). We prove the two claims for $\mathbf{n}$ and $\mathbf{L}$.

(1.) We have to prove that there exist computed answers $\delta_1, \ldots, \delta_l$ for $\mathbf{L}$ such that

$$CET \models_3 T_{\mathbf{n}}(\mathbf{L}) \rightarrow \delta_1 \ \vee \ \ldots \ \vee \ \delta_l$$

Suppose that $n_1 = 0$. Then, $T_{n_1}(L_1) = \mathbf{f}$ and therefore the claim is trivially true. So, assume that $n_1 > 0$. Without loss of generality, let us assume that the selected literal in $L_1, \ldots, L_k$ is L_1. Let $\mathbf{L}'$ be the sequence $L_2, \ldots, L_k$ and let $\mathbf{n}'$ be the sequence $n_2, \ldots, n_k$. There are two cases:

1. L_1 is of the form $p(\mathbf{s})$. Let

$$p(\mathbf{t}^1) \leftarrow \sigma^1, \mathbf{M}^1 \ \ \ldots \ \ p(\mathbf{t}^m) \leftarrow \sigma^m, \mathbf{M}^m$$

contain a variant for each clause in P with head p. Assume that these clauses are standardized apart from each other and from $\mathbf{L}$. consider the following queries:

$$(\mathbf{s} = \mathbf{t}^1), \sigma^1, \mathbf{M}^1, \mathbf{L}' \ \ \ldots \ \ (\mathbf{s} = \mathbf{t}^m), \sigma^m, \mathbf{M}^m, \mathbf{L}'$$

Let, for $i \in [1..m]$, $\mathbf{y}^i = \mathit{FreeVar}(\sigma^i, \mathbf{M}^i) - \mathit{FreeVar}(p(\mathbf{t}^i))$, and let $\mathbf{n}^{\mathbf{i}}$ be the sequence $(n_1 - 1), \ldots, (n_1 - 1)$, where the length of $\mathbf{n}^{\mathbf{i}}$ is equal to the length of $\mathbf{M}^{\mathbf{i}}$. By applying the induction hypothesis for each $i \in [1..m]$, we have that there exist computed answers ${\delta^i}_1, \ldots, {\delta^i}_{v^i}$ for $\mathbf{M}^i, \mathbf{L}'$ such that

$$CET \models_3 T_{\mathbf{n}^{\mathbf{i}}, \mathbf{n}'}(\mathbf{M}^{\mathbf{i}}, \mathbf{L}') \rightarrow \delta^i_1 \ \vee \ \ldots \ \vee \ \delta^i_{v^i}$$

We proceed by first showing that $CET \models_3 T_{\mathbf{n}}(\mathbf{L}) \rightarrow \phi$, where ϕ is a disjunction of abducible formulae, and then proving that each disjunct of ϕ is either inconsistent (with respect to CET) or (equivalent to) a computed answer for $\mathbf{L}$.

To begin with, we have by definition of T_n that

$$\begin{array}{ll} & T_{n_1}(p(\mathbf{s})) \\ \cong & T_{n_1-1}\left(\bigvee_{i\in[1..m]} \exists_{\mathbf{y}^i}((\mathbf{s}=\mathbf{t}^i),\sigma^i,\mathbf{M}^i)\right) \\ \cong & \bigvee_{i\in[1..m]} \exists_{\mathbf{y}^i}((\mathbf{s}=\mathbf{t}^i),\sigma^i,T_{n_1-1}(\mathbf{M}^i)) \\ \cong & \bigvee_{i\in[1..m]} \exists_{\mathbf{y}^i}((\mathbf{s}=\mathbf{t}^i),\sigma^i,T_{\mathbf{n}^i}(\mathbf{M}^i)) \end{array}$$

But then,

$$\begin{array}{ll} & T_{\mathbf{n}}(\mathbf{L}) \\ \cong & (\bigvee_{i\in[1..m]} \exists_{\mathbf{y}^i}((\mathbf{s}=\mathbf{t}^i),\sigma^i,T_{\mathbf{n}^i}(\mathbf{M}^i))) \ \wedge \ T_{\mathbf{n}'}(\mathbf{L}') \\ \cong & \bigvee_{i\in[1..n]} \exists_{\mathbf{y}^i}((\mathbf{s}=\mathbf{t}^i),\sigma^i,T_{\mathbf{n}^i,\mathbf{n}'}(\mathbf{M}^i,\mathbf{L}')) \end{array}$$

and therefore

$$CET \models_3 T_{\mathbf{n}}(\mathbf{L}) \rightarrow \bigvee_{i\in[1..m]} \exists_{\mathbf{y}^i}((\mathbf{s}=\mathbf{t}^i),\sigma^i,(\delta_1^i \vee \ldots \vee \delta_{v^i}^i))$$

This formula can be rewritten as

$$CET \models_3 T_{\mathbf{n}}(\mathbf{L}) \rightarrow \bigvee_{i\in[1..m]} \bigvee_{j\in[1..v^i]} \exists_{\mathbf{y}^i}((\mathbf{s}=\mathbf{t}^i),\sigma^i,\delta_j^i)$$

For this formula we prove that each disjunct on the right-hand side of the implication is either inconsistent with CET or equivalent to a computed answer for $\mathbf{L}$.

Recall that, for $i \in [1..m]$ and $j \in [1..v^i]$, δ_j^i is a computed answer for $\mathbf{M}^i, \mathbf{L}'$. But then, by Lemma 5.11.3, $(\mathbf{s}=\mathbf{t}^i),\sigma^i,\delta_j^i$ is either inconsistent (with respect to CET) or a computed answer for $(\mathbf{s}=\mathbf{t}^i),\sigma^i,\mathbf{M}^i,\mathbf{L}'$. If $(\mathbf{s}=\mathbf{t}^i),\sigma^i,\delta_j^i$ is a computed answer, we may assume that it is obtained from a refutation that does not use variables that occur in $\mathbf{L}$. From this refutation, we can construct a refutation of $\mathbf{L}$, by putting the query $\mathbf{L}$ in front. But then, for all $i \in [1..m]$ and $j \in [1..v^i]$, $\exists_{\mathbf{y}^i}((\mathbf{s}=\mathbf{t}^i)\sigma^i,\delta_j^i)$ is either inconsistent or (equivalent to) a computed answer for $\mathbf{L}$.

2. Suppose that L_1 is the negative literal $\neg A$. We have that

$$T_{n_1}(\neg A) \overset{def}{=} F_{n_1}(A)$$

But then,

$$T_{\mathbf{n}}(\mathbf{L}) \cong F_{n_1}(A) \ \wedge \ T_{\mathbf{n}'}(\mathbf{L}')$$

By the inductive hypothesis for A and n_1, $F_{n_1}(A), \mathbf{L}'$ is negatively derived from $\mathbf{L}$. By the inductive assumption 2. for $\mathbf{L}'$ and $\mathbf{n}'$, we obtain computed answers for $\mathbf{L}$.

(2.) If $n_1 = 0$, then $F_{\mathbf{n}}(\mathbf{L}) \cong F_{\mathbf{n}'}(\mathbf{L}')$ and $F_{\mathbf{n}'}(\mathbf{L}'), \mathbf{L}'$ fails by induction hypothesis, which implies that $F_{\mathbf{n}'}(\mathbf{L}'), \mathbf{L}', \mathbf{M}$ fails finitely. So, assume that $n_1 > 0$.

Consider the tree $\mathcal{T}$, which is constructed as follows:

- The root node of $\mathcal{T}$ is $F_{\mathbf{n}}(\mathbf{L}), \mathbf{L}, \mathbf{M}$.
- All nodes in $\mathcal{T}$ are of the form $F_{\mathbf{n}}(\mathbf{L}), \rho, \mathbf{L}, \mathbf{N}$ (for some ρ and $\mathbf{N}$).
- For all nodes $F_{\mathbf{n}}(\mathbf{L}), \rho, \mathbf{L}, \mathbf{N}$ in $\mathcal{T}$,
  - if the selected literal is a member of $\mathbf{L}$, then the node is a leaf, and
  - if the selected literal is a member of $\mathbf{N}$, then the children of the node are defined according to definition 5.9.6.

Clearly, $\mathcal{T}$ is the 'top part' of some SLDFA-tree for $F_{\mathbf{n}}(\mathbf{L}), \mathbf{L}, \mathbf{M}$.

We prove that $\mathcal{T}$ can be extended to a finitely failed SLDFA-tree for the query $F_{\mathbf{n}}(\mathbf{L}), \mathbf{L}, \mathbf{M}$. For this, we have to prove that we can extend all leaves $F_{\mathbf{n}}(\mathbf{L}), \rho, \mathbf{L}, \mathbf{N}$ in which a literal from $\mathbf{L}$ is selected. It is sufficient to prove that, for all ρ and $\mathbf{N}$, $F_{\mathbf{n}}(\mathbf{L}), \rho, \mathbf{L}, \mathbf{N}$ (where the selected literal is a member from $\mathbf{L}$) fails finitely. Without loss of generality, let us assume that the selected literal from $\mathbf{L}$ is L_1. Let $\mathbf{L}'$ be the sequence $L_2, \ldots, L_k$ and let $\mathbf{n}'$ be the sequence $n_2, \ldots, n_k$. There are two cases:

- L_1 is of the form $p(\mathbf{s})$. Let

$$p(\mathbf{t}^1) \leftarrow \sigma^1, \mathbf{M}^1 \quad \ldots \quad p(\mathbf{t}^m) \leftarrow \sigma^m, \mathbf{M}^m$$

contain a variant for each clause with head p. Assume that these clauses are standardized apart from each other and from $\mathbf{L}$. Let, for $i \in [1..m]$, $\mathbf{n^i}$ be the sequence $(n_1 - 1), \ldots, (n_1 - 1)$, where the length of $\mathbf{n^i}$ is equal to the length of $\mathbf{M^i}$ and let $\mathbf{y}^i = \mathit{FreeVar}(\sigma^i, \mathbf{M}^i) - \mathit{FreeVar}(p(\mathbf{t}^i))$.

We have, by definition of F_n, that

$$\mathit{comp}(P) \models_3 F_{n_1}(p(\mathbf{s})) \cong \bigwedge_{i \in [1..m]} \forall_{\mathbf{y}^i}((\mathbf{s} = \mathbf{t}^i), \sigma^i \rightarrow F_{\mathbf{n}^i}(\mathbf{M}^i))$$

But then, for all $i \in [1..m]$,

$$\mathit{comp}(P) \models_3 F_{n_1}(p(\mathbf{s})), (\mathbf{s} = \mathbf{t}^i), \sigma^i \rightarrow F_{\mathbf{n}^i}(\mathbf{M}^i)$$

and therefore

$$\mathit{comp}(P) \models_3 F_{\mathbf{n}}(\mathbf{L}), (\mathbf{s} = \mathbf{t}^i), \sigma^i \rightarrow F_{\mathbf{n}^i}(\mathbf{M}^i) \;\vee\; F_{\mathbf{n}'}(\mathbf{L}')$$

which can be rewritten as

$$\mathit{comp}(P) \models_3 F_{\mathbf{n}}(\mathbf{L}), (\mathbf{s} = \mathbf{t}^i), \sigma^i \rightarrow F_{\mathbf{n}^i,\mathbf{n}'}(\mathbf{M}^i, \mathbf{L}')$$

By induction hypothesis, for all $i \in [1..m]$, $F_{\mathbf{n}^i,\mathbf{n}'}(\mathbf{M}^i, \mathbf{L}')$ is inconsistent or $F_{\mathbf{n}^i,\mathbf{n}'}(\mathbf{M}^i, \mathbf{L}'), \mathbf{M}^i, \mathbf{L}, \mathbf{N}$ fails finitely for any fair computation rule. But then, by Lemma 5.11.2, for all $i \in [1..m]$, $F_{\mathbf{n}}(\mathbf{L}), \rho, (\mathbf{s} = \mathbf{t}^i), \sigma^i$ is inconsistent or $F_{\mathbf{n}}(\mathbf{L}), \rho, (\mathbf{s} = \mathbf{t}^i), \sigma^i, \mathbf{M}^i, \mathbf{L}', \mathbf{N}$ fails finitely for any fair computation rule. But then, there exists a finitely failed SLDFA-tree for $F_{\mathbf{n}}(\mathbf{L}), \rho, \mathbf{L}, \mathbf{N}$

- Suppose that L_1 is of the form $\neg A$. By definition, $F_{n_1}(\neg A)$ is equivalent to $T_{n_1}(A)$. But then,

$$F_{\mathbf{n}}(\mathbf{L}) \cong T_{n_1}(A) \ \vee \ F_{\mathbf{n}'}(\mathbf{L}')$$

By induction hypothesis, it follows that $F_{\mathbf{n}'}(\mathbf{L}'), \mathbf{L}', \mathbf{N}$ has a finitely failed SLDFA-tree. If we extend this tree by making the node $F_{\mathbf{n}}(\mathbf{L}), \mathbf{L}, \mathbf{N}$ the parent of $F_{\mathbf{n}'}(\mathbf{L}'), \mathbf{L}', \mathbf{N}$, we have build a finitely failed SLDFA-tree for $F_{\mathbf{n}}(\mathbf{L}), \mathbf{L}, \mathbf{N}$. But then, by Lemma 5.11.2, there exists a finitely failed SLDFA-tree for $F_{\mathbf{n}}(\mathbf{L}), \rho, \mathbf{L}, \mathbf{N}$ □

Proof: (of Theorem 5.11.1)
We have to prove that

1. If $comp(P) \cup \{\delta\} \models_3 \theta, \mathbf{L}$, then there exist computed answers $\delta_1, \ldots, \delta_n$ for $\theta, \mathbf{L}$ such that $CET \models_3 \delta \rightarrow \delta_1 \ \vee \ \ldots \ \vee \ \delta_n$.
2. If $comp(P) \models_3 comp(P) \models_3 \theta \rightarrow \neg\mathbf{L}$ then Q fails finitely.

(1.) From $comp(P) \cup \{\delta\} \models_3 \theta, \mathbf{L}$, $comp(P) \models_3 \forall\delta \rightarrow \forall(\theta, \mathbf{L})$. By Corollary 5.6.9 and Theorem 5.8.10, for some n, $CET \models_3 T_n(\forall\delta \rightarrow \forall(\theta \ \wedge \ \mathbf{L}))$, which, by definition of T_n, is equivalent to $CET \models_3 \forall\delta \rightarrow \forall(\theta \ \wedge \ T_n(\mathbf{L}))$. Let $\mathbf{n}$ be the sequence $n, \ldots, n$, whose length is the same as the length of $\mathbf{L}$. By Lemma 5.11.5, there exist SLDFA-computed answers $\delta'_1, \ldots, \delta'_l$ for $\mathbf{L}$ such that

$$CET \models_3 T_{\mathbf{n}}(\mathbf{L}) \rightarrow \delta'_1 \ \vee \ \ldots \ \vee \ \delta'_l$$

But then, we also have that

$$CET \models_3 \delta \rightarrow (\theta, \delta'_1) \ \vee \ \ldots \ \vee \ (\theta, \delta'_l)$$

Moreover by Lemma 5.11.3, for each $i \in [1..l]$, θ, δ'_i is either inconsistent, or a SLDFA-computed answer for $\theta, \mathbf{L}$. Let $\delta_1, \ldots, \delta_k$ contain all θ, δ'_i that are consistent. Then, because for those $i \in [1..l]$, for which θ, δ'_i is inconsistent, we have that $CET \not\models_3 \theta, \delta'_i$, it follows that

$$CET \models_3 \delta \rightarrow \delta_1 \ \vee \ \ldots \ \vee \ \delta_k$$

(2.) Suppose that $comp(P) \models_3 \theta \rightarrow \neg\mathbf{L}$. Then, by Corollary 5.6.9 and Theorem 5.8.10, for some n, $CET \models_3 T_n(\forall\theta \rightarrow \neg\mathbf{L}))$ and therefore, by definition of

T_n and F_n, $CET \models_3 \theta \rightarrow F_n(\mathbf{L})$. Let $\mathbf{n}$ be the sequence $n, \ldots, n$, whose length is the same as the length of $\mathbf{L}$. By Lemma 5.11.5, $F_{\mathbf{n}}(\mathbf{L}), \mathbf{L}$ fails finitely. But then, because $CET \models_3 \theta \rightarrow F_{\mathbf{n}}(\mathbf{L})$, by Lemma 5.11.2, $\theta, \mathbf{L}$ fails finitely. □

Corollary 5.11.6 (Three-valued Completeness) *Let P be a program, let Q be the query $\theta, \mathbf{L}$ and let δ be an abducible sentence. If δ is a three-valued explanation for $\langle P, \theta \wedge \mathbf{L} \rangle$, then there exist SLDFA-computed answers $\delta_1, \ldots, \delta_k$ for Q such that $CET \models_3 \delta \rightarrow \delta_1 \vee \ldots \vee \delta_k$.*

Proof: By definition, δ is a three-valued explanation for $\langle P, \theta \wedge \mathbf{L} \rangle$ if and only if $comp(P) \cup \{\delta\} \models_3 \theta \wedge \mathbf{L}$. But then, by Theorem 5.11.1, there exist SLDFA-computed answers $\delta_1, \ldots, \delta_k$ for $\theta, \mathbf{L}$ such that $CET \models_3 \delta \rightarrow \delta_1 \vee \ldots \vee \delta_k$. □

5.12 Conclusions

In this chapter we generalize Kunen semantics and Fitting semantics to the setting of abductive logic programming. This is, we think, the main contribution of this chapter. We think that, as is the case with logic programming, also with abductive logic programming these semantics are of interest, especially when considering SLD like proof procedures, as an alternative to the more informative but also computationally more expensive semantics like the argumentation semantics. Also, by providing these semantics, we underline the fact that deduction and (limited forms of) abduction are closely related.

Also, we show that it is not necessary to restrict explanations to ground formulae, as is often done when presenting semantics or proof procedures for abductive logic programs. However, by allowing variables in explanations, we have to take care with free variables in observations and explanations. In our definition of explanation, we chose to implicitly universally quantify the free variables in both observation and explanation. By doing so, we do not allow any 'communication' between observation and explanation. As a result, we cannot handle situations where the observation and explanation both are to be seen as 'generic' in some set of free variables i.e. where, given observation ϕ and explanation δ, both with free variables $\mathbf{x}$, and a substitution θ with domain $\{\mathbf{x}\}$, it is understood that $\delta\theta$ is an explanation for $\phi\theta$. We could define the notion of explanation differently, by having $comp(P) \models_3 \delta \rightarrow \phi$ in its definition, instead of $comp(P) \cup \{\delta\} \models_3 \phi$. With such a definition, there would be 'communication' between free variables in δ and ϕ. Our reasons for not doing so are mostly of a technical nature, concerning the definition of the immediate consequence operator. We think that for this alternative notion of explanation, also a Kunen semantics can be established, and that the proof procedure would also be sound with respect to this alternative notion of explanation.

In the second part of this chapter we present a generalization of Drabent's SLDFA-resolution, and use it as a proof procedure for abductive logic programming. We show that the proof procedure is sound with respect to two-valued completion semantics –provided the union of completed program and answer is consistent– and that it is sound and complete with respect to three-valued completion semantics. There is quite a difference between SLDFA-resolution for abductive logic programming, and Denecker and De Schreye's SLDNFA-resolution. For one thing, Denecker and De Schreye want the explanations to be ground conjunctions of atoms. For this, they skolemize non-ground queries, and use 'skolemizing substitutions' in the resolution steps. Instead, we allow our explanations to be arbitrary non-ground abducible formulae. These differences would make a close comparison between the two proof procedures a rather technical exercise. However, we are quite confident that, for any answer given by SLDNFA-resolution, there is an 'equivalent' SLDFA-computed answer. We expect this not to hold the other way around, simply because our proof procedure is based on constructive negation, while SLDNFA-resolution is based on negation as failure.

The great similarity between SLDFA-resolution and SLDNFA-resolution is, that they both use deduction, and both do not concern themselves with the consistency of the obtained answers with respect to the completed program. As a result, they cannot be compared with ordinary proof procedures for abductive logic programming, whose main concern *is* consistency of the obtained answers. In this context, choice between two- and three-valued completion semantics is an important one; if we use two-valued completion semantics, in addition to SLDFA-resolution we do need a procedure to check whether the obtained SLDFA-computed answer is consistent with respect to the completed program. We think that this implies a considerable increase in computation costs. On the other hand, if we use three-valued completion semantics, the need for this consistency check disappears. However, one can argue that this is a 'fake' solution: in some sense we just disregard inconsistencies, by weakening the notion of a model. In our opinion, the choice of semantics depends on your view on abductive logic programs, and the relation between abducible and non-abducible predicates. A second reason why it is interesting to look at proof procedures for abductive logic programming that do not check for consistency, is the case where you can guarantee that the union of computed answer and completed program is consistent. An example of this is the translation proposed by Denecker and De Schreye in [DS93]. The abductive logic programs resulting from this translation are acyclic (proposition 3.1), which implies that the union of their completion with a consistent abducible formula is consistent (a corollary of Proposition C.2 in [Den93]). There might be more of these examples, and it might be interesting to define classes of programs for which this property holds (among others, the above conjecture on acyclic programs should be proven).

Chapter 6

A Compositional Semantics for Modular Logic Programs

Summary

Modular programs are built as a combination of separate modules, which may evolve separately, and be verified separately. Therefore, in order to reason over such programs, *compositionality* plays a crucial role: the semantics of the whole program must be obtainable as a simple function from the semantics of its individual modules. In this chapter we propose such a compositional semantics for first-order logic programs. This semantics is correct with respect to the set of logical consequences of the program. Moreover, – in contrast with other approaches – it is always computable. Furthermore, we show how our results on first-order programs may be applied in a straightforward way to normal logic programs, in which case our semantics might be regarded as a compositional counterpart of Kunen's semantics, and discuss how these results have to be modified in order to be applied to normal constraint logic programs.

6.1 Introduction

Modularity is a crucial feature of most modern programming languages. It allows one to construct a program out of a number of separate *modules*, which can be developed, optimized and verified separately. Indeed, incremental and modular design is by now a well established software-engineering methodology which helps to verify and maintain large applications.

In the logic programming field, modularity has received a considerable attention (see for instance [BLM94]), and has generated two distinct approaches: the first one is inspired by the work of O'Keefe [O'K85] and is based on the consideration that module composition is basically a *metalinguistic* operation, in which the modular construct should be independent from the logic language being used; the second one originated with the work of Miller [Mil86, Mil89], and is obtained by using a logical system richer than Horn clauses, thus providing a *linguistic* approach. In this chapter we follow the first approach. Viewing modularity in terms of *meta-linguistic* operations on programs has several advantages. It leads to the definition of a simple and powerful methodology for structuring programs which does not require to extend the underlying language's syntax. This is essential if we want to compose modules written in different languages. Furthermore, the typical mechanisms of the object-oriented paradigm, such as encapsulation and information hiding can be easily realized within this framework [BBG$^+$93].

In order to deal with modular programs, it is crucial that the semantics we refer to is *compositional*, i.e. that the semantics of the whole program is a (simple) function of the semantics of its modules. The need for a compositional semantics becomes even more pressing if one wants to build applications in which logic modules are combined with modules that are not logic programs themselves, such as constraint solvers, imperative programs, neural networks, etc. In such a situation, compositionality enables one to reason about the logic module in isolation, while the reference to knowledge provided by other modules is maintained intact.

In logic programming, this need for a compositional semantics has been long recognized. For *definite* (i.e. negation-free) logic programs a number of semantics have been proposed. To the best of our knowledge, the first papers to discuss various forms of compositional semantic characterizations of definite logic programs were the ones of Lassez and Maher [LM84, Mah88]. Further work has been done by Mancarella and Pedreschi [MP88] and Brogi et.al. [BLM92]. In [GS89] Gaifman and Shapiro proposed a compositional semantics, which was further extended in [BGLM94] and – for CLP programs – in [GDL95].

However, in the development of semantics for *normal* logic programs, (which employ the negation operator) compositionality has been widely disregarded. Notable exception to this are the papers by Maher [Mah93] and Ferrand and Lallouet [FL95]. The reason for this disattention is that, because of the presence of the negation-as-failure mechanism, the semantics of normal logic programs is typically non-monotonic. But compositionality and non-monotonicity are (almost) irreconcilable aspects. Compositionality implies that the 'old knowledge' is maintained when new knowledge is added. Non-monotonicity is defined as exactly the opposite. Thus, it seems that one can have either compositionality or non-monotonicity, but not both. Still, we need both aspects. On the one hand, the non-monotonicity that arises from the use of negation as failure is something we want in our logic programming language, because it enables us to define relations in a natural and succinct manner. On the other hand, modularity, and therefore compositionality of the declarative semantics, is essential when one wants to use a logic programming language in real life applications.

In this chapter we propose a semantics for modular logic programs. This semantics extends the semantics for first-order programs given by Sato in [Sat92], which in turn can be regarded as an extension of Kunen semantics for normal logic programs [Kun87]. Our semantics is compositional while remaining non-monotonic to a certain extent. In essence, the semantics is compositional and monotonic on the level of composition of modules, while addition of clauses to modules remains a non-monotonic operation.

We carry out our task as follows. After some preliminaries in Section 6.2, we restate Sato's results on first-order programs in Section 6.3 in terms of modular first-order programs. Then, in Section 6.4, we provide a compositional semantics for *first-order* programs. Finally, in Sections 6.5 and 6.6, we show how this can be naturally used to provide a compositional semantics for normal logic programs and normal constraint logic programs.

6.2 Preliminaries and Notation

We assume that the reader is familiar with the basic concepts of logic programming. Throughout the chapter we use the standard terminology of [Apt90, Llo87]. We use boldface to denote finite sequences of objects. For instance, $\mathbf{x}$ denotes a sequence $x_1, \ldots, x_n$ of variables, and $\mathbf{x} = \mathbf{y}$ stands for the formula $x_1 = y_1 \wedge \ldots \wedge x_n = y_n$. In formulae, we sometimes identify a sequence of literals with its conjunction, i.e. $L_1, \ldots, L_k$ ($\mathbf{L}$ for short) also denotes the conjunction $L_1 \wedge \ldots \wedge L_k$.

Throughout the chapter we work with three valued logic, with truth values $\mathbf{t}$, $\mathbf{f}$ and $\perp$, which stand, for *true*, *false* and *undefined*, respectively. We adopt Kleene's regular three valued truth tables of [Kle52], which can be summarized as follows: the usual logical connectives have truth value $\mathbf{t}$ (or $\mathbf{f}$) when they have that truth value in ordinary two valued logic for all possible replacements of $\perp$ by $\mathbf{t}$ or $\mathbf{f}$. Otherwise they have truth value $\perp$. Three valued logic allows us to define connectives that do not exist in two valued logic. In particular in the sequel we use the symbol $\cong$ corresponding to the operator of "having the same truth value": $a \cong b$ has truth value $\mathbf{t}$ if a and b both have truth value $\mathbf{t}$, both have truth value $\mathbf{f}$ or both have truth value $\perp$; in any other case $a \cong b$ has truth value $\mathbf{f}$. As opposed to it, $\leftrightarrow$ has truth value $\perp$ when one of its arguments has truth value $\perp$ (see also Figure 5.1). Note that $\cong$ and $\leftrightarrow$ only differ when one of the operands is $\perp$; their truth tables coincide when restricted to the truth values $\mathbf{t}$ and $\mathbf{f}$.

The operator $\cong$ cannot be constructed using Kleene's regular three valued truth tables.

Definition 6.2.1 A logic connective $\diamond$ is *allowed* iff the following property holds: when $a \diamond b$ is $\mathbf{t}$ or $\mathbf{f}$ then its truth value does not change if the interpretation of one of its argument is changed from $\perp$ to $\mathbf{t}$ or $\mathbf{f}$.

A first order formula is *allowed* iff it contains only allowed connectives. $\square$

A formula containing $\cong$ is not allowed. Therefore, use of $\cong$ will be restricted:

it will only appear in predicate definitions, to relate the 'head' of a definition to its 'body'.

6.3 First Order Modules

In this section, we present the notion of first order modules, and their unfolding semantics. The definitions and results in this section are slight adaptions of the ones in [Sat92]. The difference is, that in this section we prepare the way for the composition of modules. For this we have (among others) to add the notion of 'open' predicates. That is, predicates that are used in a module, but are not defined in that module. Also, what T. Sato calls a program, we call a module.

A first order program consists of a number of modules, each of which consists of a number of predicate definitions.

Definition 6.3.1 A *(predicate) definition* is a formula of the form

$$d : p(\mathbf{x}) \cong \phi$$

where $\mathbf{x}$ is a tuple of distinct variables and ϕ is an allowed formula such that $\mathit{FreeVar}(\phi) = \mathbf{x}$.

We define $\mathit{Head}(d) \overset{def}{=} p(\mathbf{x})$ and $\mathit{Body}(d) \overset{def}{=} \phi$. □

Example 6.3.2 *Here are the definitions of the predicates even/1 and odd/1.*

$$\begin{aligned} \mathit{odd}(x) &\cong (x = s(0)) \ \lor \ \exists_y(x = s(y) \ \land \ \mathit{even}(y)) \\ \mathit{even}(x) &\cong (x = s(s(0))) \ \lor \ \exists_y(x = s(y) \ \land \ \mathit{odd}(y)) \end{aligned}$$

∘

Modules are defined within the context of a fixed *base language* $\mathcal{L}_B$, which contains all the constants and function symbols which can occur in the module itself, and a number of predicate symbols of relations which have some predefined meaning. We assume that $\mathcal{L}_B$ always contains the equality symbol '=' and (with a harmless overload of notation), three predicative constants $\mathbf{t}$, $\mathbf{f}$ and $\bot$, corresponding to the truth values $\mathbf{t}$, $\mathbf{f}$ and $\bot$. The predefined relations in $\mathcal{L}_B \setminus \{\mathbf{t}, \mathbf{f}, \bot\}$ are assumed to be defined in a fixed first-order consistent *base theory* Δ. Typical choices for Δ are for example the set of equality axioms together with Clark's equality theory, the domain closure axiom, or axioms defining arithmetic primitives. In the remainder of this chapter, we *always* implicitely assume that all modules are given on the same fixed base language $\mathcal{L}_B$, and that the meaning of the predicates and functions in $\mathcal{L}_B$ is provided by a fixed base theory Δ.

Before we define the notion of module, we first introduce some notation

Definition 6.3.3 Let S be a set of predicate definition. Then,

- $\mathit{Pred}(S) \overset{def}{=}$ the set of predicates that appear in S minus the predefined predicates in $\mathcal{L}_B$.
- $\mathit{Def}(S) \overset{def}{=} \{p \mid \exists_{d \in S} \mathit{Head}(d) = p(\mathbf{t})\}$
- $\mathit{Open}(S) \overset{def}{=} \mathit{Pred}(S) \setminus \mathit{Def}(S)$ □

Then, the notion of ***module*** is defined as follows.

Definition 6.3.4 A *module* M is a collection of predicate definitions (in $\mathcal{L}_B$) such that

- there do not exist distinct definitions d_1, d_2 in M such that $\mathit{Head}(d_1)$ is a variant of $\mathit{Head}(d_2)$, and
- $\mathit{Def}(M) \cap \mathcal{L}_B = \emptyset$. □

Predicates in $\mathit{Open}(M)$ are supposed to be *imported*, i.e. defined in some other – maybe unspecified – module M'. Those predicates are also referred to as the *open* predicates of M. If $\mathit{Open}(M)$ is empty then the module is said to be *closed.* A closed module corresponds to a classical first-order program.

6.3.1 The Unfolding Operator

The semantics we propose is based on the unfolding semantics for first-order logic programs proposed by T. Sato in [Sat92]. The only difference is, that we define the semantics on modules instead of programs, and therefore we have to take special care of the open predicates.

Let us start with recalling the definition of unfolding.

Definition 6.3.5 (Unfolding) Let $c : p(\mathbf{x}) \cong \phi$ and $d : q(\mathbf{y}) \cong \psi$ be two predicate definitions (which we assume to be standardized apart). Let $q(\mathbf{t})$ be an atomic subformula of ϕ. Then the *unfolding* of $q(\mathbf{t})$ in c (*via* d) consists of substituting $q(\mathbf{t})$ with $\phi\{\mathbf{y}/\mathbf{t}\}$. In this case c is called the *unfolded* definition while d is the *unfolding* one. □

Definition 6.3.6 Let M and N be modules. Then, $M \circ N$ is the module that is obtained by applying the unfolding operation (in parallel) to all the atoms in the bodies of the definitions of M which are defined in N, using clauses of N as unfolding clauses. □

As usual, we associate the $\circ$ operator to the left. Thus, $M \circ N \circ O$ should be read as $(M \circ N) \circ O$.

Example 6.3.7 *Consider the definitions of even and odd of Example 6.3.2. Unfolding odd using the definition of even we get the definition*

$$\begin{aligned} \mathit{odd}(x) \quad \cong \quad & x = s(0) \lor \\ & \exists_y(x = s(y) \ \land \ (y = s(s(0)) \ \lor \ \exists_z(z = s(y) \ \land \ \mathit{odd}(z)))) \end{aligned}$$

$\circ$

Now, for a module M, we adopt the following notation:

$$M^n \overset{def}{=} \begin{cases} \{p(\mathbf{x}) \cong p(\mathbf{x}) \mid p \in Def(M)\} & \text{, if } n = 0 \\ M^{n-1} \circ M & \text{, otherwise} \end{cases}$$

So, intuitively, M^n is obtained from M by unfolding n times all its atoms (using the definitions of M as unfolding definitions). Notice that $M \equiv M^1 \equiv M \circ M^0 \equiv M^0 \circ M$.

The unfolding operation, when applied to a closed module is *correct*, in the sense that it maintains the set of logical consequences. This is the content of the following Lemma, which is due to Sato [Sat92].

Lemma 6.3.8 (Correctness of the Unfolding Operation)
Let M be a closed module. Suppose that M' is obtained from M by (repeatedly) applying the unfolding operation, using the definitions of M as unfolding definitions. Then, for any allowed formula ϕ,

$$M \cup \Delta \models \phi \textit{ iff } M' \cup \Delta \models \phi$$

6.3.2 Unfolding Semantics

In [Kun87], K. Kunen proposed a three-valued completion semantics for normal logic programs. In short, his semantics states that, given a formula ϕ and a program P, ϕ is **t** *iff* ϕ is **t** in all three-valued models of the *completion* of P. He then continues to show that a formula ϕ is true in his semantics iff, for some natural number n, $\Phi^n(P) \models \phi$, where Φ is Fitting's three-valued immediate consequence operator [Fit85]. This approach – as opposed to virtually all others available for normal programs – has the advantage of leading to a semantics which is always *computable*, and thus had a great impact in the logic programming community. In [Sat92], Sato provides an extension of this semantics to first-order programs, based upon the unfolding operator.

Let us now restate Sato's results for first-order modules. Because we have to deal with open predicates, there are some slight differences. We start by defining the *skeleton* of a module.

Definition 6.3.9 Let M be a module. We denote

$$Dummy(M) \overset{def}{=} \{p(\mathbf{x}) \cong \perp \mid p \in Def(M)\}$$

Then, the *skeleton* of M (written $[M]$) is defined as follows:

$$[M] \overset{def}{=} M \circ Dummy(M)$$

□

Example 6.3.10 *Consider the module M of in Example 6.3.2. The skeleton $[M]$ of this module is*

$$\begin{aligned} odd(x) &\cong (x = s(0)) \ \vee \ \exists_y (x = s(y) \ \wedge \ \perp) \\ even(x) &\cong (x = s(s(0))) \ \vee \ \exists_y (x = s(y) \ \wedge \ \perp) \end{aligned}$$

The skeleton $[M^2]$ *of* M^2 *(after some rewriting) is*

$$\begin{array}{l} \mathit{odd}(x) \cong (x = s(0)) \ \vee \ (x = s(s(s(0)))) \ \vee \ \bot \\ \mathit{even}(x) \cong (x = s(s(0))) \ \vee \ \bot \end{array}$$

∘

Intuitively, the skeleton of a module represents all knowledge that is 'directly accessible', i.e. that can be obtained without using unfolding on the predicates defined in the module itself. This is expressed in the following lemma, which states that truth of a formula in a skeleton can be established simply by unfolding that formula once, using that skeleton.

Lemma 6.3.11 *Let* M *be a module and let* ϕ *be an allowed formula. Then,*

$$[M] \cup \Delta \models \phi \textit{ iff } \Delta \models \phi \circ [M]$$

Proof: We prove the claim by structural induction on ϕ. Suppose ϕ is an atom of the form $p(\mathbf{t})$. There are two cases.

- Suppose $p \notin \mathit{Def}(M)$.

  Then $[M] \cup \Delta \models p(\mathbf{t})$ iff $\Delta \models p(\mathbf{t})$. Moreover, $p(\mathbf{t}) \equiv p(\mathbf{t}) \circ [M]$. Therefore, the claim holds.

- Suppose $p \in \mathit{Def}(M)$.

  Then, $[M]$ must contain a definition $d : p(\mathbf{x}) \cong \psi$. Because $[M]$ is a skeleton, $\mathit{Def}(M) \cap \mathit{Pred}(\psi) = \emptyset$. Therefore,

$$\begin{array}{lll} & [M] \cup \Delta \models p(\mathbf{t}) & \\ \text{iff} & [M] \cup \Delta \models (\mathbf{x} = \mathbf{t}) \ \wedge \ \psi & \text{since } \mathit{Def}(M) \cap \mathit{Pred}(\psi) = \emptyset \\ \text{iff} & \Delta \models (\mathbf{x} = \mathbf{t}) \ \wedge \ \psi & \\ \text{iff} & \Delta \models p(\mathbf{t}) \circ [M] & \end{array}$$

The inductive steps for the logical operators are straightforward. □

Using the skeleton and the unfolding operator, we can generate an infinite chain of approximations of the meaning of a module, with $[M^0], [M^1], [M^2], \ldots$ (note that $[M^n]$ is equivalent to $M^n \circ \mathit{Dummy}(M)$). This sequence is essentially the same as the one generated by Sato in [Sat92]. In fact Theorem **3.3** in [Sat92] may be reformulated as follows.

Theorem 6.3.12 *Let* M *be a module. Then, for any allowed formula* ϕ*,*

$$M \cup \Delta \models \phi \textit{ iff, for some } n, \ [M^n] \cup \Delta \models \phi$$

Proof: We have that

$$\begin{array}{lll} & M \cup \Delta \models \phi & \\ & \text{by Theorem 3.3 in [Sat92]} & \\ \text{iff} & \exists_n \text{ such that } \Delta \models \phi \circ [M^n] & \text{by Lemma 6.3.11} \\ \text{iff} & \exists_n \text{ such that } [M^n] \cup \Delta \models \phi & \end{array}$$

□

6.3.3 An Example Modular Program

Let us now work out a small example. The following program verifies, given a directed graph, whether a certain node is *critical*, i.e. whether by removing that node from the graph, some other nodes in the network become disconnected. We assume that the graph is represented in a module M_g. This module defines only the predicate $arc/2$ in such a way that $arc(x,y)$ is t in M_g iff there is a (direct) link from x to y in the graph. Further, we have a module M_p which, referring to $arc/2$ as an open predicate, defines the predicate $path/3$ as follows

$$\begin{array}{rl} path(x,z,a) \cong & arc(x,z) \vee \\ & \exists_y arc(x,y) \ \wedge \ \neg member(y,a) \ \wedge \ path(y,z,[y|a]) \end{array}$$

Thus, $path(x,y,a)$ is true iff there exists an acyclic path from x to z that avoids all the nodes in a. The predicate $member/2$ is defined in a separate module M_m:

$$member(x,y) \cong \exists_{y_1,y_s}(y = [y_1|y_s] \ \wedge \ (x = y_1 \ \vee \ member(x,y_s)))$$

Finally, we have a module M_c that defines the predicate $critical/1$: it contains the single definition

$$critical(x) \cong \exists_{y,z}(y \neq x \ \wedge \ z \neq x \ \wedge \ path(y,z,[\,]) \ \wedge \ \neg path(y,z,[x]))$$

which states that x is critical if we can find a path from some node y to some node z, both different from x, but we cannot find a path from y to z that avoids x. If we want to compute critical nodes of different graphs, we compose this module with different graph modules.

Now, let us see how these modules behave under unfolding. Consider module M_p. Figure 6.1 shows the body of the definition of $path/3$ in M_p^0, in $M_p^1 (\equiv M_p)$ and in M_p^2. The definition of $path/3$ in $[M_p^0]$, in $[M_p^1]$ and in $[M_p^2]$ can simply be obtained by replacing with the constant $\perp$ all the atoms in the above table which have *path* as predicate symbol. This is due to the fact that *path* is the only non-open predicate symbol occurring in M_p.

Finally, it is worth noticing that, since the body of the definition of $critical/1$ does not contain any non-open predicate, for all n, $M_c \equiv M_c^n \equiv [M_c^n]$.

n	body of *path*/3 in M_p^n
0	$\perp$
1	$arc(x,z) \vee$ $\exists_y\ arc(x,y) \wedge \neg member(y,a) \wedge path(y,z,[y\vert a])$
2	$arc(x,z) \vee$ $\exists_y(\ arc(x,y) \wedge \neg member(y,a) \wedge$ $(\ arc(y,z) \vee$ $\exists_{y'}\ arc(y,y') \wedge \neg member(y',[y\vert a]) \wedge path(y',z,[y'[y\vert a]])\ )\ )$

Figure 6.1: Unfolding the definition of *path*/3.

6.4 A Compositional Semantics

Following the original paper of R. O'Keefe [O'K85], the approach to modular programming we consider here is based on a *meta-linguistic* programs composition mechanism. In this framework, logic programs are seen as elements of an algebra and the composition operation is modelled by an operator on the algebra.

Viewing modularity in terms of *meta-linguistic* operations on programs has several advantages. For one, it leads to the definition of a simple and powerful methodology for structuring programs which does not require to extend the underlying language's syntax. This is not the case if one tries to extend programs by *linguistic* mechanisms, an approach which originated with the work of Miller [Mil86, Mil89]. Moreover, *meta-linguistic* operations are quite powerful. For instance, the compositional systems of Mancarella and Pedreschi [MP88], Gaifman and Shapiro [GS89], Bossi et.al. [BBG$^+$93] and Brogi et.al. [BLM92, BMPT90] can be seen as different instances of this idea. Furthermore, the typical mechanisms of the object-oriented paradigm, such as encapsulation and information hiding, as well as more complex form of composition mechanisms – in which we may distinguish between imported, exported, and local (hidden) predicates – can be easily realized within this framework. These mechanisms are implemented (for instance) in the language Gödel [HL94], in Quintus Prolog [Qui86] and in SICStus Prolog [Car88]. For a more detailed analysis we refer to the survey of Bugliesi et.al. [BLM94].

6.4.1 Module Composition

To compose first-order modules we follow the same approach as [BGLM94] and use a simple program union operator.

Definition 6.4.1 (Module Composition) Let M_1 and M_2 be modules. We define

$$M_1 \oplus M_2 \overset{def}{=} M_1 \cup M_2$$

provided that $\mathit{Def}(M_1) \cap \mathit{Def}(M_2) = \emptyset$. Otherwise $M_1 \oplus M_2$ is undefined. □

This definition extends in a straightforward way to the case of several modules: $M_1 \oplus \ldots \oplus M_k$ is defined naturally as $(M_1 \oplus \ldots \oplus M_{k-1}) \oplus M_k$.

In the definition we use, we require $\mathit{Def}(M_i) \cap \mathit{Def}(M_j) = \emptyset$, for all distinct i and j. This condition allows us to circumvent a number of unnecessary technicalities, and, in particular, to keep module composition a *monotonic* operation. At first, the condition seems rather restrictive, in that it prevents one from refining a predicate p in a module M, by composing it with some module M' also containing a definition for p. Strictly speaking, this is true. However, part of the problem can be easily solved by the use of some renaming and an additional 'interface' module. Consider a predicate p, defined in both N_1 and N_2. Then, $N_1 \oplus N_2$ is not defined. However, we can circumvent this problem as follows. We rename p to p_1 (resp. p_2) in the head of the definition of p in N_1 (resp. N_2), resulting in a module N_1' (resp. N_2') (we assume that p_1 and p_2 are "new" predicate symbols). Additionally, we define an interface module I

$$I = \{p(\mathbf{x}) \cong p_1(\mathbf{x}) \ \vee \ p_2(\mathbf{x})\}$$

Now observe that $I \oplus N_1' \oplus N_2'$ is well-defined (provided there are no other name clashes) and behaves exactly the way we would expect $N_1 \oplus N_2$ to. Thus, the extra condition we add is not a real restriction. Finally, it is worth noticing that mutual recursion among modules *is allowed*.

The most important question with respect to a composition operator, like our $\oplus$ is, whether the semantics one has in mind is compositional with respect to this operator. In our case, we have to decide whether, for modules $M_1, \ldots, M_k$ on a common base language $\mathcal{L}_B$ such that $M_1 \oplus \ldots \oplus M_k$ is defined, we have that

$$M_1 \oplus \ldots \oplus M_k \cup \Delta \models \phi \text{ iff } \exists_n \ [M_1^n] \oplus \ldots \oplus [M_k^n] \cup \Delta \models \phi$$

In Theorem 6.4.11, we prove that this is the case.

6.4.2 Expressiveness of Modules

Before we look at compositionality of $\oplus$, we first take a little detour, in order to define two kinds of 'expressiveness' orders on modules. When defining the abstract concept of *expressiveness* of modules, we have to take into account the fact that modules are meant to be composed together. This we do in the following (semantical) notion of expressiveness.

Definition 6.4.2 Let M and N be two modules such that $\mathit{Def}(M) = \mathit{Def}(N)$. Then, M is *(compositionally) more expressive* than N (written $M \succeq N$) if, for any other module Q such that $M \oplus Q$ and $N \oplus Q$ are defined, for any allowed formula ϕ, $N \oplus Q \cup \Delta \models \phi$ implies $M \oplus Q \cup \Delta \models \phi$.

Two modules M and N are *(compositionally) equivalent* (written $M \sim N$) if $M \succeq N \succeq M$. □

In other words, we say that two first-order modules are compositionally equivalent if they have the same set of logical consequences in every possible 'context'. Therefore, – according to the notation of [BLM94] – $\sim$ is actually a congruence relation.

The relation $\succeq$ is clearly and order relation. This order is based on the expressiveness of the modules in three valued logic, and is therefore purely semantical. The following lemma states an obvious yet important property of this order relation.

Lemma 6.4.3 *Let M, N and Q be modules such that $M \oplus Q$ is defined. If $M \succeq N$ then $M \oplus Q \succeq N \oplus Q$.*

Thus, $\succeq$ combines naturally with the $\oplus$ operator. Regrettably, $\succeq$ does not combine as well with the other operators we have defined, i.e. the $\circ$ and the $[\,]$ operator. First of all, $M \succeq N$ does not imply $[M] \succeq [N]$.

Example 6.4.4 *Consider the following modules and their skeletons:*

$$\begin{array}{ll} M & : \{p \cong q, q \cong \mathbf{t}\} \\ N & : \{p \cong \mathbf{t}, q \cong \mathbf{t}\} \\ & \vdots \\ [M] & : \{p \cong \bot, q \cong \mathbf{t}\} \\ [N] & : \{p \cong \mathbf{t}, q \cong \mathbf{t}\} \end{array}$$

Clearly, $M \succeq N$. On the other hand, $[N] \models p$ while $[M] \not\models p$. Thus, $[M] \not\succeq [N]$. ∘

Secondly, $M \succeq N$, does not imply, for arbitrary Q, $Q \circ M \not\succeq Q \circ N$.

Example 6.4.5 *Consider the modules of Example 6.4.4, together with the following modules:*

$$\begin{array}{ll} Q & : \{r \cong p\} \\ Q \circ M & : \{r \cong q\} \\ Q \circ N & : \{r \cong \mathbf{t}\} \end{array}$$

Although $M \succeq N$, we have that $Q \circ N \models r$, while $Q \circ M \not\models r$. Therefore, we do not have that $Q \circ M \succeq Q \circ N$. ∘

Because of these difficulties with $\succeq$, we now define a second order on modules, which is syntactical rather than semantical.

Definition 6.4.6 Let $p(\mathbf{x}) \cong \phi$ and $p(\mathbf{x}) \cong \psi$ be predicate definitions. Then,

$$p(\mathbf{x}) \cong \phi \hookrightarrow p(\mathbf{x}) \cong \psi$$

if ψ can be obtained from ϕ by substituting some (or none) of its subformulae with the propositional constant $\bot$.

Let M and N be two modules such that $Def(M) = Def(N)$. Then, $M \hookrightarrow N$ if, for each definition $d \in M$, there exists a definition $d' \in N$ such that $d \hookrightarrow d'$. □

Of course, $\hookrightarrow Q$ is transitive. Therefore $\hookrightarrow$ induces an order relation on modules, and it will be used in that sense. As it turns out, $\hookrightarrow$ behaves better with respect to $\circ$ and $[\ ]$. This is not that surprising, when one notices that these two operators themselves are syntactical rather than semantical. Some (simple) properties of $\hookrightarrow$ that are going to be needed in the sequel are the following.

Remark 6.4.7 *For any module M, we have*

- $M \hookrightarrow [M]$
- $[M \circ M] \hookrightarrow [M]$

Let M and N be modules. If $M \hookrightarrow N$ then

- $[M] \hookrightarrow [N]$

Let M and N and Q be modules. If $M \hookrightarrow N$ then

- $M \oplus Q \hookrightarrow N \oplus Q$
- $M \circ Q \hookrightarrow N \circ Q$
- $Q \circ M \hookrightarrow Q \circ N$

Let us now relate $\hookrightarrow$ and $\succeq$.

Lemma 6.4.8 *Let M and N be modules. Then, $M \hookrightarrow N$ implies $M \succeq N$.*

Proof: Assume that $M \hookrightarrow N$. To prove the claim, we have to prove that, for any allowed formula ϕ, $N \cup \Delta \models \phi$ implies $M \cup \Delta \models \phi$. From Theorem 6.3.12 it suffices to prove that, for all n, $[N^n] \cup \Delta \models \phi$ implies $[M^n] \cup \Delta \models \phi$.

Assume that $[N^n] \cup \Delta \models \phi$. By Lemma 6.3.11, $\Delta \models \phi \circ [N^n]$. By Remark 6.4.7, $[M^n] \hookrightarrow [N^n]$, and therefore $\phi \circ [N^n]$ is obtained from $\phi \circ [M^n]$ by replacing some subformulae with the predicative constant $\perp$. Therefore, $\Delta \models \phi \circ [M^n]$. Again, by Lemma 6.3.11, it follows that $[M^n] \cup \Delta \models \phi$. □

It is easy to check that the converse of this lemma does not hold. Thus, $\hookrightarrow$ is a stronger order relation than $\succeq$.

6.4.3 A Compositional Semantics for First-Order Modules

In this section we prove, in Theorem 6.4.11, that the semantics we propose is compositional. Before we can prove this, we need some lemmata.

Lemma 6.4.9 *Let M and N be modules such that $M \oplus N$ is defined. Then $[M^{n+1}] \circ [(M \oplus N)^n] \hookrightarrow M \circ [(M \oplus N)^n]$.*

Proof: We prove the claim by induction on n. For $n = 0$ the claim holds trivially, because

$$[M^1] \circ [(M \oplus N)^0] \equiv M \circ [(M \oplus N)^0]$$

Assume we proved the claim for n. We have to prove that

$$[M^{n+2}] \circ [(M \oplus N)^{n+1}] \hookrightarrow M \circ [(M \oplus N)^{n+1}]$$

To begin with, we make two observations:

1. $$\begin{aligned} & (M \circ [M^{n+1}]) \circ [(M \oplus N)^{n+1}] \\ \equiv\ & M \circ (([M^{n+1}] \circ [(M \oplus N)^{n+1}]) \oplus |[(M \oplus N)^{n+1}]|_{Def(N)}) \end{aligned}$$

   where we use the notation $|M|_S$ to denote the restriction of M to those definitions whose head appears in S. In order to prove this identity, let us focus on the leftmost occurrence of the module M in the left hand side of the above equivalence, and consider an atom A in the body of a definition of M. If A is defined in M then A will be unfolded via $[M^{n+1}]$ and successively via $[(M \oplus N)^{n+1}]$. Otherwise, if A is not defined in M then A will be left unchanged by the application of the unfolding via $[M^{n+1}]$. It might successively be modified by the unfolding via $[(M \oplus N)^{n+1}]$. This is exactly what would happen if we unfolded A via

   $$(([M^{n+1}] \circ [(M \oplus N)^{n+1}]) \oplus |[(M \oplus N)^{n+1}]|_{Def(N)})$$

   And this is what we do (to A) on the right hand side of the above identity.

2. $$\begin{aligned} & |[(M \oplus N)^{n+1}]|_{Def(N)} \\ \equiv\ & |M \oplus N|_{Def(N)} \circ [(M \oplus N)^n] \\ \equiv\ & N \circ [(M \oplus N)^n] \end{aligned}$$

We are now able to prove the claim.

$$\begin{aligned}
& [M^{n+2}] \circ [(M \oplus N)^{n+1}] \\
\equiv\ & (M \circ [M^{n+1}]) \circ [(M \oplus N)^{n+1}] \\
& \text{by observation 1} \\
\equiv\ & M \circ (([M^{n+1}] \circ [(M \oplus N)^{n+1}]) \oplus |[(M \oplus N)^{n+1}]|_{Def(N)}) \\
& \text{by observation 2} \\
\equiv\ & M \circ (([M^{n+1}] \circ [(M \oplus N)^{n+1}]) \oplus (N \circ [(M \oplus N)^n])) \\
& \text{by Remark 6.4.7} \\
\hookrightarrow\ & M \circ (([M^{n+1}] \circ [(M \oplus N)^n]) \oplus (N \circ [(M \oplus N)^n])) \\
& \text{by the induction and Remark 6.4.7} \\
\hookrightarrow\ & M \circ ((M \circ [(M \oplus N)^n]) \oplus (N \circ [(M \oplus N)^n])) \\
\equiv\ & M \circ ((M \oplus N) \circ [(M \oplus N)^n]) \\
\equiv\ & M \circ [(M \oplus N)^{n+1}]
\end{aligned}$$

□

Lemma 6.4.10 *Let M and N be modules such that $M \oplus N$ is defined. Then $[([M^n] \oplus [N^n])^n] \hookrightarrow [(M \oplus N)^n]$.*

Proof: We proceed by induction on n. For the base case, where $n = 1$, the claim holds trivially, because $[([M^1] \oplus [N^1])^1] \equiv [(M \oplus N)^1]$.

Assume the claim holds for n. Then

$$\begin{array}{ll}
 & [([M^{n+1}] \oplus [N^{n+1}])^{n+1}] \\
\equiv & [([M^{n+1}] \oplus [N^{n+1}]) \circ ([M^{n+1}] \oplus [N^{n+1}])^n] \\
 & \text{by Remark 6.4.7} \\
\hookrightarrow & [([M^{n+1}] \oplus [N^{n+1}]) \circ ([M^n] \oplus [N^n])^n] \\
 & \text{by induction and Remark 6.4.7} \\
\hookrightarrow & [([M^{n+1}] \oplus [N^{n+1}]) \circ [(M \oplus N)^n]] \\
\equiv & [([M^{n+1}] \circ [(M \oplus N)^n]) \oplus ([N^{n+1}] \circ [(M \oplus N)^n])] \\
 & \text{by Lemma 6.4.9} \\
\hookrightarrow & [(M \circ [(M \oplus N)^n]) \oplus (N \circ [(M \oplus N)^n])] \\
\equiv & [(M \oplus N) \circ [(M \oplus N)^n]] \\
\equiv & [(M \oplus N)^{n+1}]
\end{array}$$

Hence the claim holds for $n + 1$. □

Now, we are finally able to prove our main theorem.

Theorem 6.4.11 (Main) *Let $M_1, \ldots, M_k$ be first-order modules such that $M_1 \oplus \ldots \oplus M_k$ is defined. Then, for all allowed formulae ϕ,*

$$M_1 \oplus \ldots \oplus M_k \cup \Delta \models \phi \textit{ iff } \exists_n\ [M_1^n] \oplus \ldots \oplus [M_k^n] \cup \Delta \models \phi$$

Proof: We prove the claim by induction on k. To begin with, let us consider the base case, where $k = 2$.
($\Leftarrow$) From Remark 6.4.7 we know that $M_1^n \hookrightarrow [M_1^n]$ and therefore (via the same Remark) that $M_1^n \oplus M_2^n \hookrightarrow [M_1^n] \oplus [M_2^n]$. Therefore, we have by Lemma 6.4.8 that, if $[M_1^n] \oplus [M_2^n] \cup \Delta \models \phi$ then $M_1^n \oplus M_2^n \cup \Delta \models \phi$. Therefore, by the correctness of the unfolding operation, Lemma 6.3.8 and Lemma 6.4.3, it follows that $M_1 \oplus M_2 \cup \Delta \models \phi$.
($\Rightarrow$) Assume that $M_1 \oplus M_2 \models \phi$. By Theorem 6.3.12 we have that there exists an integer n such that

$$[(M_1 \oplus M_2)^n] \cup \Delta \models \phi$$

Now,

$$\begin{array}{lll}
 & [(M_1 \oplus M_2)^n] & \text{by Lemmata 6.4.10 and 6.4.8} \\
\preceq & [([M_1^n] \oplus [M_2^n])^n] & \text{by Remark 6.4.7} \\
\preceq & ([M_1^n] \oplus [M_2^n])^n & \text{by Lemma 6.3.8} \\
\sim & [M_1^n] \oplus [M_2^n] &
\end{array}$$

But then, it follows that $[M_1^n] \oplus [M_2^n] \cup \Delta \models \phi$.

Now, assume we have proven the claim for k or less modules. We prove the claim for $k+1$ modules. Then,

$$
\begin{array}{lll}
 & M_1 \oplus \ldots \oplus M_{k+1} \cup \Delta \models \phi & \text{Let } N \equiv M_1 \oplus \ldots \oplus M_k \\
\text{iff} & N \oplus M_{k+1} \cup \Delta \models \phi & \text{induction hypothesis} \\
\text{iff} & \exists_n : [N^n] \oplus [M_{k+1}^n] \cup \Delta \models \phi & \text{Lemma 6.3.11} \\
\text{iff} & \exists_n : \Delta \models \phi \circ ([N^n] \oplus [M_{k+1}^n]) & \\
\text{iff} & \exists_n : \Delta \models \phi \circ [M_{k+1}^n \circ [N^0]] \circ [N^n] & \text{Let } \psi \equiv \phi \circ [M_{k+1}^n \circ [N^0]] \\
\text{iff} & \exists_n : \Delta \models \psi \circ [N^n] & \text{Lemma 6.3.11} \\
\text{iff} & \exists_n : [N^n] \cup \Delta \models \psi & \text{Theorem 6.3.12} \\
\text{iff} & \exists_n : N \cup \Delta \models \psi & \text{induction hypothesis} \\
\text{iff} & \exists_{n,m} : [M_1^m] \oplus \ldots \oplus [M_k^m] \cup \Delta \models \psi & \text{Let } N' \equiv ([M_1^m] \oplus \ldots \oplus [M_k^m]) \\
\text{iff} & \exists_{n,m} : N' \cup \Delta \models \psi & \text{Lemma 6.3.11} \\
\text{iff} & \exists_{n,m} : \Delta \models \psi \circ N' & \\
\text{iff} & \exists_{n,m} : \Delta \models \phi \circ [M_{k+1}^n \circ [N^0]] \circ N' & \\
\text{iff} & \exists_{n,m} : \Delta \models \phi \circ ([M_{k+1}^n] \oplus N') & \text{Lemma 6.3.11} \\
\text{iff} & \exists_{n,m} : [M_{k+1}^n] \oplus N' \cup \Delta \models \phi & \\
\text{iff} & \exists_{n,m} : [M_1^m] \oplus \ldots \oplus [M_k^m] \oplus [M_{k+1}^n] \cup \Delta \models \phi & \\
\text{iff} & \exists_n : [M_1^n] \oplus \ldots \oplus [M_{k+1}^n] \cup \Delta \models \phi &
\end{array}
$$

□

Notice that, if M is a module, then $[M^n]$ is a collection of formulae of the form $p(\mathbf{x}) \cong \phi$, where ϕ contains *only* open or base predicates (for instance, in $[M^n]$, recursion is impossible). In a way, we could say that each $[M^n]$ is an *elementary* module. With this in mind, the above theorem states that the semantics of a module M is given by the increasing sequence of elementary modules $[M^0], [M^1], [M^2], \ldots$.

In the introduction, we referred to the fact that we have both compositionality and non-monotonicity. One on the level of modules and the other within modules. Let us now see how this is achieved. Consider a module M, with $\mathcal{L}_B$ consisting of equality and the constants a and b. Suppose that M only defines a predicate q, in such a way that only $q(a)$ holds. Now, suppose extend our knowledge on q. That is, add the fact that $q(b)$ holds. There are two ways of achieving this. One that is non-monotonic and one that is compositional; one that adds knowledge by changing modules and one that adds knowledge by adding modules.

First, the non-monotonic method, in which we directly change M. In that case, let the definition in M simply be $q(x) \cong x = a$, and the definition in the replacement module, M' be $q(x) \cong x = a \vee x = b$. Then, going from M to M', we have non-monotonic behaviour, because $M \cup \Delta \models \neg q(b)$ while it is not the case that $M' \cup \Delta \models \neg q(b)$. This method of adding knowledge is not compositional, because we cannot ensure that, for some N and ϕ, $M \oplus N \cup \Delta \models \phi$ implies $M' \oplus N \cup \Delta \models \phi$.

On the other hand, we can choose to add knowledge in a compositional way, using open predicates. Therefore, let M be the module containing the definition $q(x) \cong x = a \ \vee \ q'(x)$. The predicate $q'(x)$ is an open predicate that can be used to extend our knowledge on q. We do this by adding a module N containing the definition $q'(x) \cong x = b$. Now, a move from M to $M \oplus N$ is monotonic, because we have that $M \not\models \neg q(b)$ and also $M \oplus N \not\models \neg q(b)$. This is caused by the fact that q' is an open predicate and therefore $q'(b)$ is $\perp$ in M, which causes $b = a \ \vee \ q'(b)$ to be $\perp$ in M.

Thus, compositionality is achieved by use of open predicates. In this respect it is now clear why, for achieving compositionality, it it essential that a predicate is defined in only one module. If we allow predicates to be defined in more than one module, any new module could extend already defined predicates and induce non-monotonic behaviour.

6.5 Normal Logic Programs

In this section we show how the results provided in the previous section may be used in a straightforward way in order to provide a compositional semantics to normal logic programs (i.e. logic programs with negation). Normal modules are finite collections of *normal clauses*, $A \leftarrow L_1, \dots, L_m$. where A is an atom and each L_i is a literal (i.e. an atom or a negated atom).

Since negative information cannot follow from a set of clauses, in order to provide a sound semantics to a normal module we follow [Cla78] and refer to the module's completion.

Definition 6.5.1 Let M be a normal module and $p(\mathbf{t}_1) \leftarrow \mathbf{L}_1, \dots, p(\mathbf{t}_r) \leftarrow \mathbf{L}_r$ be all the clauses which define the predicate symbol p in M. The *completed definition* of p is

$$p(\mathbf{x}) \cong \bigvee_{i=1}^{r} \exists_{\mathbf{y}_i} (\mathbf{x} = \mathbf{t}_i) \ \wedge \ \mathbf{L}_i$$

where the $\mathbf{x}$ are fresh variables and $\mathbf{y}_i = \mathit{FreeVar}(\mathbf{L}_i) \backslash \mathit{FreeVar}(\mathbf{t}_i)$.

The *completion* of M, $\mathit{Comp}(M)$ consists in the conjunction of the completed definition of all the predicates *defined* in M. □

It is important to notice that here we depart from [Cla78] in the fact that we do *not* close those definitions which are *not explicitly given* in M. In a modular context, these predicates need to remain open.

The completed definition of a predicate is a first order formula that contains the equality symbol; hence, in order to interpret '=' correctly, we also need an equality theory.

Definition 6.5.2 $CET_{\mathcal{L}}$, *Clark's Equality Theory* for the language $\mathcal{L}$, consists of the axioms:

- $f(\mathbf{x}) \neq g(\mathbf{y})$ for all distinct f, g in $\mathcal{L}$,
- $f(\mathbf{x}) = f(\mathbf{y}) \rightarrow \mathbf{x} = \mathbf{y}$ for all f in $\mathcal{L}$, and

- $x \neq t$ for all terms t distinct from x in which x occurs;

(where $\mathbf{x}$ and $\mathbf{y}$ are sequences of fresh variables of proper arity) together with the usual *equality axioms*, that are needed in order to interpret '=' correctly: *reflexivity, symmetry, transitivity,* and $(\mathbf{x} = \mathbf{y}) \rightarrow (f(\mathbf{x}) = f(\mathbf{y}))$ for all functions f in $\mathcal{L}$. □

Notice that that '=' is always interpreted as two valued.

Obviously, $CET_{\mathcal{L}}$ depends on the underlying language $\mathcal{L}$, which we assume to be fixed and to contain all the functions symbols occurring in all the modules we consider.

A known problem that semantics based on program completion face is that when $\mathcal{L}$ is finite (that is, when it contains only a finite number of functions symbols) $CET_{\mathcal{L}}$ is not a complete theory. Typically, this problem is solved by adopting one of the following solutions:

1. adding to $CET_{\mathcal{L}}$ some domain closure axioms which are intended to restrict the interpretation of the quantification to $\mathcal{L}$-terms (as in [She88a]),

2. assuming that the language contains always an infinite set of predicate symbols (as in [Kun87]) or

3. by considering only interpretations and models over a specific fixed domain D (as in [Fit85]).

This latter solution requires the adoption of axioms which are usually not first order (unless all the functions symbols are 0-ary, i.e. constants), and consequently leads to a semantics which is (usually) noncomputable. For these reasons we adopt either solutions (1) or (2). Luckily, these two solutions yield basically the same semantics. For an extended discussion of the subject, we refer to [Kun87, She88a].

Definition 6.5.3 Let $\mathcal{L}$ be a finite language (i.e. a language with a finite set of predicate symbols). The *Domain Closure Axiom* for the language $\mathcal{L}$, $DCA_{\mathcal{L}}$, is

$$\exists_{\mathbf{y}_1}(x = f_1(\mathbf{y}_1)) \ \vee \ \ldots \ \vee \ \exists_{\mathbf{y}_r}(x = f_r(\mathbf{y}_r))$$

where $f_1, \ldots, f_r$ are all the function symbols in $\mathcal{L}$ and $\mathbf{y}_1, \ldots, \mathbf{y}_r$ are tuples of variables of the appropriate arity. □

This axiom is also referred to as the *weak* domain closure axiom[1].

It is now easy to see that in this context, the semantics for open normal logic modules finds a natural embedding in the one proposed for first order modules in Section 6.4 (the underlying language $\mathcal{L}_B$ contains only the equality

[1]As opposed to it, the *strong* domain closure axiom for a language $\mathcal{L}$ is $x = t_1 \vee x = t_2 \vee \ldots$ where $t_1, t_2, \ldots$ is the (usually infinite) sequence of all the ground $\mathcal{L}$-terms. This axiom is equivalent to choice (c) above, and determines uniquely the universe of the possible interpretation. Again, if $\mathcal{L}$ contains a non-constant function symbol then the above axiom is not a first order formula, and leads to a noncomputable semantics.

predicate). Module composition is defined exactly as for the case of first-order modules: if M_1 and M_2 are normal modules, we define $M_1 \oplus M_2 = M_1 \cup M_2$ provided that $Def(M_1) \cap Def(M_2) = \emptyset$ holds. Otherwise $M_1 \oplus M_2$ is undefined.

Corollary 6.5.4 *Let $M_1, \ldots, M_k$ be normal modules such that $M_1 \oplus \ldots \oplus M_k$ is defined. Then, for each allowed formula ϕ there exists an integer n such that the following statements are equivalent:*

1. $comp(M_1 \oplus \ldots \oplus M_k) \cup CET_{\mathcal{L}} \models \phi$
2. $[comp(M_1)^n] \oplus \ldots \oplus [comp(M_k)^n] \cup CET_{\mathcal{L}} \models \phi$

where we assume that, if $\mathcal{L}$ is finite, $CET_{\mathcal{L}}$ incorporates $DCA_{\mathcal{L}}$.

As an example, consider again the problem of deciding whether a node in a graph is critical. The program given in the Section 6.3.3 can also be written as a modular normal program composed by the modules defining *arc*, *member*, together with the following two modules.

$$\begin{array}{lrcl} N_p : & path(x,z,a) & \leftarrow & arc(x,z) \\ & path(x,z,a) & \leftarrow & arc(x,y), \neg member(y,a), path(y,z,[y|a]) \\ N_c : & critical(x) & \leftarrow & x \neq y, x \neq z, path(y,z,[]), \neg path(y,z,[x]) \end{array}$$

In fact it is easy to check that M_p and M_c coincide with the completion of N_p and N_c.

6.6 Constraint Logic Programs

The *Constraint Logic Programming* paradigm (CLP for short) has been proposed by Jaffar and Lassez [JL87] in order to integrate a generic computational mechanism based on constraints with the logic programming framework. Such an integration results in a framework which – for programs without negation – preserves the existence of equivalent operational, model-theoretic and fixpoint semantics. Indeed, as discussed in [Mah93], most of the results which hold for *definite* (i.e. negation-free) constraint logic programs can be lifted to CLP in a quite straightforward way. We refer to the recent survey [JM94] by Jaffar and Maher for the notation and the necessary background material about CLP. A CLP clause is a formula of the form $A \leftarrow c \wedge L_1 \wedge \ldots \wedge L_k$ where A is an atom, $L_1, \ldots, L_k$ are literals and c is a *constraint*, i.e a first order formula in a specific language $\mathcal{L}_C$.

Historically, the semantics of the constraints is determined in either one of the following two ways:

1. by providing a consistent *theory*, that their interpretation has to satisfy (like Peano's arithmetic), or

2. by giving a *structure* Σ over which they have to be interpreted, (for instance, the natural numbers).

It is clear that if we follow the first approach then the results of the previous section can be naturally used to provide a semantics to normal CLP. All we have to do is to incorporate in the base theory Δ the theory that provides a meaning to the constraints and to refer to the modules *completion* (which is defined exactly as in the case of normal logic programs). The rest is straightforward.

Regrettably, the second approach is certainly more popular in the CLP community (even though also the first one is considered standard [JM94]). The problem with this approach is that the given structure determines uniquely the universe of the models, and this – in presence of negation – leads to a semantics which is again usually noncomputable. As already done in [Kun87, Sat92], we can avoid this problem by referring to some *elementary extension* of the structure itself. This has been done by Sandro Etalle in an extended version of [ET96]. Here, we limit ourselves to stating the following corollary.

Corollary 6.6.1 *Let $C_1, \ldots, C_k$ be normal constraint logic modules such that $C_1 \oplus \ldots \oplus C_k$ is defined. If $\mathcal{L}_C$ is the language of the constraints and Σ is a structure for $\mathcal{L}_C$, then there exists an elementary extension Σ' of Σ such that, for each ϕ the following statements are equivalent*

1. $comp(C_1 \oplus \ldots \oplus C_k) \models_{\Sigma'} \phi$
2. $\exists_n [comp(C_1)^n] \oplus \ldots \oplus [comp(C_k)^n] \models_{\Sigma'} \phi$

The need to refer to an enriched structure Σ' is shown by the following example. Consider the following CLP modules over the language of integer arithmetics

$$N_1 : \{p \leftarrow \neg n(x)\}$$
$$N_2 : \{n(0) \leftarrow \mathbf{t}, n(x) \leftarrow x = y + 1 \ \wedge \ n(y)\}$$

If the interpretation of the constraint is determined by the standard structure Nat, with the set $\mathbb{N}$ of natural numbers as universe, then we have that $comp(N_1) \oplus comp(N_2) \models_{\mathsf{Nat}} \neg p$. On the other hand, there does not exist a natural number n such that $[comp(N_1)^n] \oplus [comp(N_2)^n] \models_{\mathsf{Nat}} \neg p$. This shows the need of extending the extend the structure Nat. Further, in our opinion, p should not be considered false in the semantics of $N_1 \oplus N_2$: firstly because if we take *any* non-trivial extension Nat' of Nat, $comp(N_1) \oplus comp(N_2) \not\models_{\mathsf{Nat}'} \neg p$, so the falsehood of p depends in a way on the limits of the universe of Nat, and, secondly, because the falsehood of p is in any case not computable (one would need $\omega + 1$ computation steps in order to calculate it).

6.7 Conclusions

In this chapter we propose a semantics for first order programs which is compositional with respect to the $\oplus$ (module composition) operator. This semantics is built via a first-order unfolding operator and allows to characterize (compositionally) the set of logical consequences of the module in three valued logics. Further, we have shown how our results may be applied to modular normal programs and normal CLP. The semantics we have proposed may be regarded as a compositional counterpart of Kunen's semantics for normal programs [Kun87] and its first-order version due to Sato [Sat92].

Another recent proposal for a compositional semantics for logic programs is the one of G. Ferrand and A. Lallouet [FL95]. In this paper, Ferrand and Lallouet propose two compositional semantics, one based on Fitting semantics and one based on well-founded semantics. The notion of program unit they use is similar to the notion of (open) module. The differences between their approach and ours stem mostly from the kind of models that are considered. In both Fitting semantics and well-founded semantics for normal logic programs, interpretations are only considered over a fixed universe (typically, the Herbrand universe of the program). As a result, these semantics cannot be axiomatized within first-order logics. Consequently, – and we think this is even more important – these semantics are in general noncomputable (they may require more than ω iterations in order to be built). In contrast, our semantics for modular normal and first-order logic programs is based upon arbitrary three-valued models and characterized by a countably infinite sequence of approximations, and is thus recursively enumerable.

In [Mah93] Maher presents a transformation system for normal programs with respect to a compositional version of the perfect model semantics, which is defined in the same paper. From the point of view of modularity the main difference between this chapter and [Mah93] is that in [Mah93] modules are also required to have a hierarchical calling pattern. For instance mutual recursion among modules is prohibited. From the purely semantics point of view the differences between this chapter and [Mah93] may be assimilated to the differences between the perfect model semantics and Kunen's semantics (the first is based on two-valued logics, imposes some syntactic restriction on the syntax of modules (stratification, or local stratification), and, in particular, it is usually not computable).

Bibliography

[AB87] B. Arbab and D.M. Berry. Operational and denotational semantics of Prolog. *Journal of Logic Programming*, 4(4):309–329, 1987.

[AB91] K.R. Apt and M. Bezem. Acyclic programs. *New Generation Computing*, 9:335–363, 1991.

[AB94] K.R. Apt and R. Bol. Logic programming and negation: a survey. *Journal of Logic Programming*, 19-20:9–71, 1994.

[AD94] K.R. Apt and K. Doets. A new definition of SLDNF-resolution. *Journal of Logic Programming*, 18(2):177–190, 1994.

[Apt90] K.R. Apt. Logic programming. In L. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 494–574. Elsevier Science Publishers B.V., 1990.

[Apt94] K. R. Apt. Program verification and Prolog. In E. Börger, editor, *Specification and Validation methods for Programming languages and systems*. Oxford University Press, 1994.

[AT95] K.R. Apt and F.J.M. Teusink. *Comparing Negation in Logic Programming and in Prolog*, pages 111–133. MIT Press, 1995.

[BBG$^+$93] A. Bossi, M. Bugliesi, M. Gabbrielli, G. Levi, and M. C. Meo. Differential logic programming. In *Proceedings of the Annual ACM Symposium on Principles of Programming Languages*, pages 359–370. ACM Press, 1993.

[BGLM94] A. Bossi, M. Gabbrielli, G. Levi, and M. C. Meo. A Compositional Semantics for Logic Programs. *Theoretical Computer Science*, 122(1-2):3–47, 1994.

[BLM92] A. Brogi, E. Lamma, and P. Mello. Compositional Model-theoretic Semantics for Logic Programs. *New Generation Computing*, 11(1):1–21, 1992.

[BLM94] M. Bugliesi, E. Lamma, and P. Mello. Modularity in logic programming. *Journal of Logic Programming*, 19-20:443–502, 1994.

[BMPT90] A. Brogi, P. Mancarella, D. Pedreschi, and F. Turini. Composition Operators for Logic Theories. In J. W. Lloyd, editor, *Proceedings of the Symposium on Computational Logic*, Basic Research Series, pages 117–134. Springer Verlag, 1990.

[Car88] M. Carlsson. *SICStus Prolog User's Manual.* Po Box 1263, S-16313 Spanga, Sweden, February 1988.

[CDT91] L. Console, D.T. Dupre, and P. Torasso. On the relationship between abduction and deduction. *Journal of Logic and Computation*, 1(5):661–690, 1991.

[Cha88] D. Chan. Constructive negation based on the completed database. In *Proceedings of the International Conference on Logic Programming*, pages 111–125. MIT Press, 1988.

[CK73] C.C. Chang and H.J. Keisler. *Model Theory.* Number 73 in Studies in Logic and the Foundations of Mathematics. North-Holland Publishing Company, 1973.

[CKW89] W. Chen, M. Kifer, and D.S. Warren. Hilog: A first-order semantics for higher-order logic programming constructs. In *Proceedings of the North American Conference on Logic Programming*, Cleveland, Ohio, October 1989.

[Cla78] K.L. Clark. Negation as failure. In H. Gallaire and G. Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, 1978.

[Den93] M. Denecker. *Knowledge Representation and Reasoning in Incomplete Logic Programming.* PhD thesis, Katholieke Universiteit Leuven, Leuven, Belgium, September 1993.

[Dix93] J. Dix. Semantics of Logic Programs: Their Intuitions and Formal Properties. An Overview. In Andre Fuhrmann and Hans Rott, editors, *Logic, Action and Information. Proceedings of the Konstanz Colloquium in Logic and Information (LogIn '92).* DeGruyter, 1993.

[DM88] S.K. Debray and P. Mishra. Denotational and operational semantics for Prolog. *Journal of Logic Programming*, 5(1):61–91, 1988.

[Doe93] K. Doets. *From Logic to Logic Programming.* The MIT Press series in Foundations of Computing. MIT Press, 1993.

[Dra93a] W. Drabent. On completeness of SLDNF-resolution for nonfloundering queries. In *Proceedings of the International Logic Programming Symposium*, page 643, 1993.

[Dra93b] W. Drabent. SLS-resolution without floundering. In *Proceedings of the workshop on Logic Programming and Non-Monotonic Reasoning*, 1993.

[Dra95] W. Drabent. What is failure? an aproach to constructive negation. *Acta Informatica*, 32(1), 1995.

[DS92] M. Denecker and D. De Schreye. SLDNFA: an abductive procedure for normal abductive programs. In *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 686–700, 1992.

[DS93] M. Denecker and D. De Schreye. Representing incomplete knowledge in abductive logic programming. In *Proceedings of the International Logic Programming Symposium*, 1993.

[Dun91] Phan Minh Dung. Negation as hypotheses: an abductive foundation for logic programming. In *Proceedings of the International Conference on Logic Programming*, pages 3–17, 1991.

[dV88] E.P. de Vink. Equivalence of an operational and a denotational semantics for a Prolog-like language with cut. Technical Report IR-151, Vrije Universiteit Amsterdam, Faculteit Wiskunde en Informatica, De Boelelaan 1081, 1081 HV Amsterdam, The Netherlands, March 1988.

[EK89] K. Eshghi and R.A. Kowalski. Abduction compares with negation by failure. In G. Levi and M. Martelli, editors, *Proceedings of the International Conference on Logic Programming*, pages 234–254, 1989.

[Esh88] K. Eshghi. Abductive planning with the event calculus. In K.A. Bowen and R.A. Kowalski, editors, *Proceedings of the International Conference on Logic Programming*, pages 562–579, 1988.

[ET96] S. Etalle and F. Teusink. A compositional semantics for modular logic programs. In *Proceedings of the Joint International Conference and Symposium on Logic Programming*, 1996.

[Fag] F. Fages. Constructive negation by pruning. *Journal of Logic Programming.* To appear.

[Fag91] F. Fages. A new fixpoint semantics for general logic programs compared with the well-founded and the stable model semantics. *New Generation Computing*, 9:425–443, 1991.

[Fit85] M. Fitting. A Kripke-Kleene semantics for logic programs. *Journal of Logic Programming*, 2(4):295–312, 1985.

[Fit87] M. Fitting. Enumeration operators and modular logic programming. *Journal of Logic Programming*, 4:11–21, 1987.

[Fit91] M. Fitting. Well-founded semantics, generalized. In Vijay Saraswat and Kazunori Ueda, editors, *International Symposium on Logic Programming*, pages 71–84, 1991.

[Fit93] M. Fitting. The family of stable models. *Journal of Logic Programming*, 17(2/3&4):197–225, 1993.

[FL95] Gérard Ferrand and Arnaud Lallouet. A compositional proof method of partial correctness for normal logic programs. In John Lloyd, editor, *Proceedings of the International Logic Programming Symposium*, pages 209–223, 1995.

[Fle] A.C. Fleck. A semantic alternative to the 'cut' operator of Prolog.

[GDL95] M. Gabbrielli, G.M. Dore, and G. Levi. Observable Semantics for Constraint Logic Programs. *Journal of Logic and Computation*, 5(2):133–171, 1995.

[GHK$^+$80] G. Gierz, K.H. Hofmann, K. Keimel, J.D. Lawson, M.W. Mislove, and D.S. Scott. *A Compendium of Continuous Lattices*. Springer Verlag, 1980.

[GL88] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proceedings of the International Conference on Logic Programming*, pages 1070–1080, 1988.

[GL90] M. Gelfond and V. Lifschitz. Logic programs with classical negation. In *Proceedings of the International Conference on Logic Programming*, pages 579–597, 1990.

[GS89] H. Gaifman and E. Shapiro. Fully abstract compositional semantics for logic programs. In *Proceedings of the Annual ACM Symposium on Principles of Programming Languages*, pages 134–142. ACM Press, 1989.

[HL94] P. M. Hill and J. W. Lloyd. *The Gödel programming language*. MIT Press, 1994.

[HLS90] P.M. Hill, J.W. Lloyd, and J.C. Shepherdson. Properties of a pruning operator. *Journal of Logic and Computation*, 1(1):99–143, 1990.

[Jia94] Y. Jiang. Ambivalent logic as the semantic basis fo metalogic programming: I. In P. Van Hentenryck, editor, *Proceedings of the International Conference on Logic Programming*, pages 387–401. MIT Press, June 1994.

[JL87] J. Jaffar and J.-L. Lassez. Constraint Logic Programming. In *Proceedings of the Annual ACM Symposium on Principles of Programming Languages*, pages 111–119. ACM Press, 1987.

[JM84] N.D. Jones and A. Mycroft. Stepwise development of operational and denotational semantics for Prolog. In *International Symposium on Logic Programming*, pages 281–288, 1984.

[JM94] Joxan Jaffar and Michael J. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19/20:503–581, 1994.

[Kal93] M. Kalsbeek. The vanilla meta-interpreter for definite logic programs and ambivalent syntax. Technical Report CT-93-01, Department of Mathematics and Computer Science, University of Amsterdam, The Netherlands, 1993.

[KKT93] A.C. Kakas, R.A. Kowalski, and F. Toni. Abductive logic programming. *Journal of Logic and Computation*, 2(6):719–770, 1993.

[Kle52] S.C. Kleene. *Introduction to Metamathematics.* Van Nostrand, Princeton, 1952.

[KM90] A.C. Kakas and P. Mancarella. Generalized stable models: a semantics for abduction. In *Proc. 9th European Conference on Artificial Intelligence*, pages 385–391, Stockholm, 1990.

[KM91] A.C. Kakas and P. Mancarella. Negation as stable hyptheses. In Nerode, Marek, and Subrahmanian, editors, *Proceedings of the workshop on Logic Programming and Non-Monotonic Reasoning*, Washington, DC, 1991.

[Kow74] R.A. Kowalski. Predicate logic as a programming language. In *Proceedings IFIP'74*, pages 569–574, Amsterdam, 1974. North-Holland.

[Kun87] K. Kunen. Negation in logic programming. *Journal of Logic Programming*, 4:289–308, 1987.

[LB92] A. Lilly and B.R. Bryant. A prescribed cut for Prolog that ensures soundness. *Journal of Logic Programming*, 14(4):287–339, 1992.

[Llo87] J.W. Lloyd. *Foundations of Logic Programming.* Symbolic Computation – Artificial Intelligence. Springer Verlag, 1987. Second, extended edition.

[LM84] J.-L. Lassez and M. J. Maher. Closures and Fairness in the Semantics of Programming Logic. *Theoretical Computer Science*, 29:167–184, 1984.

[Mah88] M. J. Maher. Equivalences of Logic Programs. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 627–658. Morgan Kaufmann, Los Altos, Ca., 1988.

[Mah93] M. J. Maher. A transformation system for deductive databases with perfect model semantics. *Theoretical Computer Science*, 110(2):377–403, 1993.

[McC80] J. McCarthy. Circumscription – a form of non-monotonic reasoning. *Artificial Intelligence*, 13:27–39, 1980.

[McC86] J. McCarthy. Applications of circumscription to formalized common sense knowledge. *Artificial Intelligence*, 28:89–116, 1986.

[MF92] J. Minker and J.A. Fernández. Semantics of disjunction in deductive databases. In *1992 International Conference in Database Theory*, 1992.

[Mil86] D. Miller. A Theory of Modules for Logic Programming. In *Proceedings of the IEEE International Symposium on Logic Programming*, pages 106–114, 1986.

[Mil89] D. Miller. A Logical Analysis of Modules in Logic Programming. *Journal of Logic Programming*, 6:79–108, 1989.

[MNR92] V.W. Marek, A. Nerode, and J.B. Remmel. The stable models of a predicate logic program. In *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 446–460, 1992.

[Moo85] R.C. Moore. Semantic considerations on non-monotonic logic. *Artificial Intelligence*, 25:75–94, 1985.

[Mos86] C. Moss. Cut & Paste – defining the impure primitives of Prolog. In E. Shapiro, editor, *Proceedings of the International Conference on Logic Programming*, number 225 in Lecture Notes in Computer Science, pages 686–694. Springer Verlag, 1986.

[MP88] P. Mancarella and D. Pedreschi. An Algebra of Logic Programs. In R. A. Kowalski and K. A. Bowen, editors, *Proceedings of the International Conference on Logic Programming*, pages 1006–1023. MIT Press, 1988.

[MT92] M. Martelli and C. Tricomi. A new SLDNF-tree. *Information Processing Letters*, 43(2):57–62, 1992.

[MT95] Elena Marchiori and Frank Teusink. Proving termination of logic programs with delay declarations. In John Lloyd, editor, *Proceedings of the International Logic Programming Symposium*. MIT, 1995.

[O'K85] R. A. O'Keefe. Towards an Algebra for Constructing Logic Programs. In *Proceedings of the IEEE International Symposium on Logic Programming*, pages 152–160, 1985.

[Prz89a] T.C. Przymusinski. Every logic program has a natural stratification and an iterated fixed point model. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 11–21, 1989.

[Prz89b] T.C. Przymusinski. On constructive negation in logic programming. In *Proceedings of the North American Conference on Logic Programming*, page 16, 1989. draft 30/4/88.

[Prz90a] T.C. Przymusinski. Extended stable semantics for normal and disjunctive programs. In *Proceedings of the International Conference on Logic Programming*, pages 459–477, 1990.

[Prz90b] T.C. Przymusinski. Well-founded semantics coincides with three-valued stable semantics. *Fundamenta Informaticae*, 13(4):445–463, 1990.

[Qui86] *Quintus Prolog User's Guide and Reference Manual—Version 6*, April 1986.

[Rei78] R. Reiter. On closed-world databases. In H. Gallaire and J.Minker, editors, *Proceedings of the Symposium on Logic and Databases*, pages 55–76. Plenum Press, 1978.

[Rei80] R. Reiter. A logic for default theory. *Artificial Intelligence*, 13:81–132, 1980.

[Ric74] B. Richards. A point of reference. *Synthese*, 28:431–445, 1974.

[Sat92] T. Sato. Equivalence-preserving first-order unfold/fold transformation system. *Theoretical Computer Science*, 105(1):57–84, 1992.

[She88a] J. C. Shepherdson. Negation in logic programming. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 19–88. Morgan Kaufmann, 1988.

[She88b] J.C. Shepherdson. Language and equality theory in logic programming. Technical Report PM-88-08, School of Mathematics, University Walk, Bristol, BS8 1 TW, England, 1988.

[SI92] K. Satoh and N. Iwayama. A query evaluation method for abductive logic programming. In K.R. Apt, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 671–685, 1992.

[Stu91] P.J. Stuckey. Constructive negation for constraint logic programming. In *Proceedings of the IEEE Symposium on Logic in Computer Science*, pages 328–339. IEEE Computer Society Press, July 1991.

[SZ90] D. Saccà and C. Zaniolo. Stable models and non-determinism in logic programs with negation. In *Proceedings of the ACM Symposium on Principles of Database Systems*, page 16, 1990.

[Teu91] F.J.M. Teusink. A lazy justification-based reason maintenance system. Master's thesis, University of Utrecht, Department of Mathematics and Computer Science, Utrecht, The Netherlands, December 1991.

[Teu93a] F.J.M. Teusink. A characterization of stable models using a non-monotonic operator. In L.M. Pereira and A. Nerode, editors, *Proceedings of the workshop on Logic Programming and Non-Monotonic Reasoning*, pages 206–222, 1993.

[Teu93b] F.J.M. Teusink. A proof procedure for extended logic programs. In D. Milner, editor, *Proceedings of the International Logic Programming Symposium*, 1993.

[Teu94a] F.J.M. Teusink. A characterization of stable models using a non-monotonic operator. *Methods of Logic in Computer Science*, 1(4), 1994.

[Teu94b] F.J.M. Teusink. Three-valued completion for abductive logic programs. In Giorgio Levi and Mario Rodríguez-Artalejo, editors, *Proceedings of the International Conference on Algebraic and Logic Programming*, number 850 in Lecture Notes in Computer Science, pages 150–167. Springer Verlag, 1994.

[Teu96] Frank Teusink. Three-valued completion for abductive logic programs. *Theoretical Computer Science*, 1996. to appear.

[vGRS91] A. van Gelder, K.A. Ross, and J.S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650, July 1991.

[Wag91] G. Wagner. Ex contradictione nihil sequitur. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 538–543, 1991.

[Wag93] G. Wagner. Neutralization and preemption in extended logic programs. Technical Report Bericht Nr. 20/93, Freien Universität Berlin, 1993.

Index

Samenvatting

Logisch programmeren is gebasseerd op het idee dat men de computer problemen kan laten oplossen met behulp van de logica (een wiskundige manier om om te gaan met termen als 'waar' en 'onwaar'). Dit gebeurt als volgt. In een logisch programma wordt kennis over een bepaald probleemgebied opgeslagen met behulp van regels van de vorm "als A waar is dan is B ook waar". Om iets preciezer te zijn, een *regel* heeft de vorm

$$B \leftarrow A_1, \ldots, A_n$$

waarbij B en A_1 t/m A_n 'woorden' zijn, de $\leftarrow$ staat voor 'als', en de komma's staan voor 'en'. Dus: B is waar als A_1 t/m A_n allemaal waar zijn. We noemen B de *conclusie* en noemen A_1 t/m A_n de *voorwaarden*. Hier zijn drie (eenvoudige) regels:

$$\begin{aligned}&\mathit{voeten\_nat} \leftarrow \mathit{op\_gras}, \mathit{gras\_nat}\\ &\mathit{gras\_nat} \leftarrow\\ &\mathit{op\_gras} \leftarrow\end{aligned}$$

De eerste regel zegt dat je voeten nat worden als je op gras loopt en het gras nat is. De tweede en derde regel hebben beide geen voorwaarden (het gedeelte rechts van de $\leftarrow$ is leeg). Als een regel geen voorwaarden heeft, is de conclusie altijd waar. Zo'n regel noemen we dan ook een *feit*. Een aantal regels samen noemen we een *(logisch) programma*. De bovenstaande drie regels vormen dus een programma.

Om een computer met behulp van een logisch programma problemen te laten oplossen, stellen we vragen aan de computer. In deze samenvatting zullen we alleen werken met vragen waarop de computer alleen 'ja' of 'nee' hoeft te antwoorden. Bij het bovenstaande programma kunnen we bijvoorbeeld de vraag

$$?-\mathit{voeten\_nat}$$

stellen. Ofwel: "is het waar dat mijn voeten nat zijn?"

De computer probeert de vraag te beantwoorden door (enkel en alleen) te redeneren op basis van de vraag, en de regels in het programma. De computer gaat op zoek naar een *bewijs* voor de vraag. In het bovenstaande voorbeeld beredeneert de computer dat *voeten_nat* waar is als *op_gras* en *gras_nat* waar zijn. Vervolgens komt de computer erachter dat dit allebei feiten in het programma zijn. Uiteindelijk zal de computer de vraag dus met 'ja' beantwoorden. We zeggen dat de vraag *slaagt*. Wat als we nu de vraag ?−*broek_nat* stellen? Het programma heeft geen enkele kennis met betrekking tot het woord *broek_nat*. Omdat de computer niet kan beredeneren dat *broek_nat* waar is, zal het de vraag met 'nee' beantwoorden. We zeggen dat de vraag *faalt*.

In sommige gevallen willen we graag waarheid en onwaarheid van een woord kunnen omkeren. Hiervoor gebruiken we negatie, dat aangegeven wordt met het '¬'-teken. Het idee is dat als een woord waar is, zijn negatie onwaar is, en omgekeerd. Dus:

> *voeten_nat* is waar dan en slechts dan als ¬*voeten_nat* onwaar is
> ¬*voeten_nat* is waar dan en slechts dan als *voeten_nat* onwaar is

Het voorbeeldprogramma bevat geen enkel '¬'-teken. Hoe moet de computer nu redeneren om een vraag als ?−¬*voeten_nat* of ?−¬*broek_nat* te beantwoorden? Als we de noties van waarheid en onwaarheid uit de klassieke logica gebruiken, kan de computer dat niet. We mogen daarin namelijk wel concluderen dat een vraag 'waar' is, wanneer die vraag slaagt, maar we mogen *niet* concluderen dat een vraag 'onwaar' is, wanneer die vraag faalt. De reden hiervoor is, dat er in de logica vragen bestaan waarvoor geen bewijs gevonden kan worden, maar die ook niet 'onwaar' zijn. Als we de computer volgens de klassieke logica laten werken, zullen vragen met een ¬-teken dus altijd falen, zonder dat dat geïnterpreteerd mag worden als "de vraag is onwaar".

Toch willen we dat de computer iets anders doet, dan simpelweg met 'nee' antwoorden. Om te weten te komen wat we dan wel willen, moeten we beter kijken naar hoe mensen redeneren. Dat doen we hier aan de hand van het lezen van het spoorboekje. Neem het volgende programma dat de direkte verbindingen tussen een aantal stations als feiten van de vorm *trein*(van, naar) weergeeft:

$$\begin{aligned}
&trein(Amsterdam, Utrecht) \leftarrow\\
&trein(Utrecht, Amersfoort \leftarrow\\
&trein(Amsterdam, Amersfoort) \leftarrow\\
&trein(Amersfoort, Zutphen) \leftarrow
\end{aligned}$$

Als mens zal men uit dit programma concluderen dat er geen directe verbinding bestaat tussen Amsterdam en Zutphen. Wat men in feite doet, is het volgende. Men zoekt eerst in het spoorboekje of er een direkte verbinding is tussen Amsterdam en Zutphen. Die vindt men niet. Vervolgens gaat men ervan uit dat in het spoorboekje alle bestaande verbindingen vermeld worden. Op grond daarvan komt men tot de conclusie dat er dus geen direkte verbinding tussen Amsterdam en Zutphen bestaat.

Om de computer op deze manier te laten redeneren, voegen we de volgende twee 'wetten' toe:

$$\neg \mathit{trein}(\mathit{Amsterdam}, \mathit{Zutphen}) \text{ slaagt als } \mathit{trein}(\mathit{Amsterdam}, \mathit{Zutphen}) \text{ faalt}$$
$$\neg \mathit{trein}(\mathit{Amsterdam}, \mathit{Zutphen}) \text{ faalt als } \mathit{trein}(\mathit{Amsterdam}, \mathit{Zutphen}) \text{ slaagt}$$

(ook voor alle andere woorden in het programma). Als we de computer nu de vraag $?{-}\neg \mathit{trein}(\mathit{Amsterdam}, \mathit{Zutphen})$ stellen, zal hij die proberen te beantwoorden door zichzelf eerst de vraag $?{-}\mathit{trein}(\mathit{Amsterdam}, \mathit{Zutphen})$ te stellen. Die vraag faalt. Op grond daarvan weet de computer nu dat de oorspronkelijke vraag, $?{-}\neg \mathit{trein}(\mathit{Amsterdam}, \mathit{Zutphen})$, slaagt, en hem dus met 'ja' beantwoorden.

Deze vorm van redeneren wordt 'negatie als falen' genoemd. We illustreren het nut van negatie als falen aan de hand van een klassiek voorbeeld. Het idee van logisch programmeren is, om kennis over een bepaald (probleem) gebied in de computer op te slaan. Eén zo'n gebied is bijvoorbeeld kennis over dieren. We willen bijvoorbeeld het 'feit' vastleggen, dat vogels vliegen. Dat zou kunnen met de regel

$$\mathit{vliegt}(x) \leftarrow \mathit{vogel}(x)$$

die stelt dat als (een dier) x een vogel is, dan kan die x ook vliegen. Dat is een prachtige regel; als je een willekeurige voorbijganger vraagt of vogels kunnen vliegen, zal hij 'ja' antwoorden. Alleen, het is niet de hele waarheid, omdat er ook vogelsoorten zijn die niet kunnen vliegen, zoals penguins. Wat we eigenlijk willen vastleggen is, dat 'normaal gesproken' alle vogels vliegen, maar dat er ook uitzonderingen zijn op die regel. Dit kunnen we op de volgende manier doen met negatie als falen:

$$\begin{aligned}
&\mathit{vliegt}(x) \leftarrow \mathit{vogel}(x), \neg \mathit{vreemde\_vogel}(x)\\
&\mathit{vogel}(x) \leftarrow \mathit{adelaar}(x)\\
&\mathit{vogel}(x) \leftarrow \mathit{penguin}(x)\\
&\mathit{vreemde\_vogel}(x) \leftarrow \mathit{penguin}(x)\\
&\\
&\mathit{adelaar}(\mathit{Sam}) \leftarrow\\
&\mathit{penguin}(\mathit{Tweety}) \leftarrow
\end{aligned}$$

Laten we eerst kijken naar Sam, en de vraag $?{-}\mathit{vliegt}(\mathit{Sam})$ stellen. We kunnen afleiden dat $\mathit{vogel}(\mathit{Sam})$ waar is, omdat Sam een adelaar is, en omdat (volgens de tweede regel) alle adelaars vogels zijn. En $?{-}\neg \mathit{vreemde\_vogel}(\mathit{Sam})$? Die zal ook slagen, omdat we op geen enkele manier kunnen afleiden dat Sam een vreemde vogel is (alleen penguins zijn vreemde vogels). Sam vliegt dus, volgens het programma. En hoe zit het nou met Tweety? We weten dat Tweety een penguin is. Ook weten we dat alle penguins vreemde vogels zijn. Dus Tweety is een vreemde vogel. Daarom faalt de vraag $?{-}\neg \mathit{vreemde\_vogel}(\mathit{Tweety})$, en zal ook de vraag $?{-}\mathit{vliegt}(\mathit{Tweety})$ falen. Tweety vliegt dus niet. Merk op dat dit programma er dus normaal van uitgaat dat vogels vliegen. Alleen als we er bij een vogel of vogelsoort bijzeggen dat het een vreemde vogel is, wijken we af van die standaard regel.

Zoals al eerder gezegd, komt negatie als falen niet overeen met de negatie die in de klassieke logica gebruikt wordt. Toch komt negatie als falen erg van pas wanneer we de computer willen laten redeneren op een manier die enigzins overeenkomt met die van een mens. Daarom is men negatie als falen, en de logica die daaraan ten grondslag ligt, nader gaan bestuderen. Deze studie is ook het onderwerp van dit proefschrift. In mijn onderzoek heb ik mij met name bezig gehouden met de vraag hoe negatie als falen de noties van waarheid en onwaarheid beïnvloedt, en op welke manier negatie als falen (en een aantal verwante uitbreidingen) gebruikt kunnen worden om kennis over een bepaald probleemgebied beter weer te geven in een logisch programma.

Curriculum Vitae

Ik ben geboren op 4 oktober 1967 in Hengelo, Overijssel. De naam Frank (Franciscus) heb ik te danken aan het feit dat ik ben geboren op wereld-dierendag, en de tweede van een tweeling ben. Nadat mijn ouders genoeg van de schok (een verdubbeling van het aantal te wassen katoenen luiers) waren bekomen om op zoek te gaan naar een tweede jongensnaam, kwamen ze op het lumineuze idee me te noemen naar de Heilige Franciscus van Assisi, die naar verluid met dieren kon praten. De tocht naar een doctorstitel ben ik begonnen op kleuterschool 'De Nissenhut'. Deze school bestaat niet meer; op de plaats waar men destijds kon genieten van schreeuwende kinderen kan men tegenwoordig genieten van alternatieve popmuziek, in cultureel centrum 'Babylon'. Na de basisschool (op de HSV, Hengelose Schoolvereniging), heb ik op het Twickel College de HAVO gevolgd. Daarna ben ik informatica gaan studeren op de HIO (Hogere Informatica Opleiding), destijds onderdeel van het IHBO 'de Maere' in Enschede (inmiddels onderdeel van de Hogeschool Enschede), waar ik tijdens mijn afstudeer-stage heb gewerkt aan een expertsysteem voor het ontwerpen van compressoren. Omdat ik na de HIO niet het idee had dat ik er klaar voor was om met stropdas en koffer het bedrijfsleven in te trekken, ben ik het jaar daarop met koffer, maar zonder stropdas, informatica gaan studeren aan de Rijksuniversiteit Utrecht (die inmiddels het predikaat Rijks- is kwijtgeraakt). Mijn afstudeerproject had betrekking op zogenaamde 'Truth Maintenance Systems', die gebruikt worden in sommige expertsystemen. Na afgestudeerd te zijn aan de universiteit, had ik wel degelijk genoeg van studeren. Tijdens mijn afstuderen had ik echter de smaak van onderzoek te pakken gekregen, en toen ik dan ook de kans kreeg op een OIO-positie bij Krzysztof Apt op het CWI, heb ik deze met beide handen aangegrepen. Bij het drukken van dit proefschrift is dat een kleine vijf jaar geleden. Met een beetje geluk ben ik op 30 september 1996 gepromoveerd, om daarna de overstap naar het bedrijfsleven te maken.