Logic Programming and Negation: A Survey

Krzysztof R. Apt
CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
and
Faculty of Mathematics and Computer Science
University of Amsterdam, Plantage Muidergracht 24
1018 TV Amsterdam, The Netherlands

Roland Bol
Department of Mathematics and Computer Science
Eindhoven University of Technology
P.O. Box 513, 5600 MB Eindhoven, The Netherlands

Abstract

We survey here various approaches which were proposed to incorporate negation in logic programs. We concentrate on the proof-theoretic and model-theoretic issues and the relationships between them.

AMS Classification (1991): 68Q40, 68T15.
CR Subject Classification (1992): F.3.2., F.4.1, H.3.3, I.2.3.
Keywords and Phrases: negation, general logic programs, non-monotonic reasoning.

Notes. The work of the first author was partly supported by ESPRIT Basic Research Action 6810 (Compulog 2). The work of the second author was partly supported by the Netherlands Organization for Scientific Research (NWO). This paper will appear in Journal of Logic Programming.

1 Introduction

1.1 Motivation

Non-monotonic reasoning grew out of attempts to capture the essential aspects of commonsense reasoning. It resulted in a number of important formalisms, the most known of them being the *circumscription method* of McCarthy [107], the *default theory* of Reiter [145] and *autoepistemic logic* of Moore [111]. (For a systematic comparison of these approaches see the recently published Marek and Truszczyński [104].)

One of the striking features of logic programming is that it can naturally support non-monotonic reasoning — by means of negative literals. Many concepts introduced in the area of non-monotonic reasoning have a natural counterpart within logic programming in spite of its limited syntax. The dual interpretation of logic programs — as a computational mechanism and as a formalism for knowledge representation — provided a fertile ground for a study of proof theory and semantics of programs which support non-monotonic reasoning.

This paper attempts to survey the outcome of this research. This subject, or some fragments of it, were already discussed in no less than five previous survey articles: Shepherdson [158], Shepherdson [159], Przymusinska and Przymusinski [120], Bidoit [20] and Clark [38]. Moreover, while writing this paper we learned of another survey – that of Dix [47] who focuses on the non-monotonic reasoning aspects of logic programming. However, this field is so fast growing – about half of the references cited here were published during the last 3 years – that another survey of it might be justified. We provide here an alternative overview of this area by concentrating on the main developments in the proof theory and model theory and on the relationships between them.

No unified picture emerges from this endevour. A number of interesting proposals were made dealing with proof theory and semantics. The multifarious relationship between them, revealed by often intricate mathematical arguments, brings to light the complex nature of logic programming and of non-monotonic reasoning in the logic programming setting.

1.2 Setting the Stage

The SLD-resolution of Kowalski [85] allows us to derive only positive consequences (namely conjunctions of atoms) from a (positive) program. However, in many circumstances it is also useful to derive negative consequences. As we shall see in the next subsection, this naturally leads to non-monotonic reasoning.

A classic example for the usefulness of negative consequences is the timetable, which states connections explicitly, but the absence of connections only implicitly. In the case of positive programs three approaches to derive negative information became most known. Each of them is treated more extensively for the case of general programs (called normal programs in Lloyd [93]).

1. Use *the negation as finite failure rule* of Clark [39], which states that $\neg Q$ is a consequence of a program P if a finitely failed SLD-tree for the query Q w.r.t. P exists (in short, if Q *finitely fails*),

2. use *program completion* of Clark [39], which strengthens the program by — informally — interpreting implications as equivalences,

3. use the *closed world assumption* (in short CWA) of Reiter [144], which states that for a ground atom A, $\neg A$ is a consequence of a program P if A cannot be proved from P.

The relationships between these concepts for positive programs are by now well-understood (see e.g. Lloyd [93] or Apt [2] for an overview of these results).

Once negative consequences can be derived from a positive program, it is natural to extend the syntax of programs and allow negative assumptions. This leads to the class of general programs in which negative literals are allowed in the bodies of the clauses. However, when trying to extend the above approaches to the case of general programs several complications arise. The approaches become self-referential, and thereby potentially paradoxical. Moreover, as we shall see, a naive amalgamation of the SLD-resolution and the negation as finite failure rule yields an unsound reasoning method, completion of a general program can be inconsistent, and the closed world assumption can yield an inconsistent theory. So to treat this subject we have to carefully review the concepts it relies upon. Let us start by discussing some relevant aspects of non-monotonic reasoning.

1.3 Non-monotonic Inference Relations

Properties of non-monotonic inference relations have been extensively studied, for example by Kraus, Lehmann and Magidor [86]. (For an overview of this topic see Makinson [100].) Dix [44] defined how a (proof- or model-theoretic) semantics for logic programs can be viewed as an *inference relation* (*consequence relation*). Given a particular semantics SEM, he defined the inference relation $\mid\!\sim_P^{SEM}$ of a program P as a relation between sets of ground atoms[1] and sets of ground literals:

$$\{A_1, \ldots, A_n\} \mid\!\sim_P^{SEM} \{L_1, \ldots, L_m\} \text{ if } SEM(P \cup \{A_1, \ldots, A_n\}) \models L_1, \ldots, L_m.^2$$

One way of classifying semantics for logic programming is by studying which properties they satisfy (a property is satisfied by SEM if it is satisfied by $\mid\!\sim_P^{SEM}$ for all programs P for which SEM is defined). Eventually, if one agrees on which properties are desirable and which ones are not, this study can be one of the reasons for considering one semantics better than another (another one could be for example computability).

A very strong property of inference relations is *monotonicity*. Below we use the symbol $\mid\!\sim$ to denote an arbitrary inference relation, Γ for a set of atoms, A for an atom and L for a literal.

Definition 1.1 (Monotonicity) An inference relation is *monotonic* if it satisfies:

$$\Gamma \mid\!\sim L \text{ implies } \Gamma \cup \Gamma' \mid\!\sim L.$$

□

Classical logic is monotonic, thus so is the inference relation determined by the SLD-resolution, as it is a subset of classical logic. The negation as finite failure rule, the program completion, and the closed world assumption all introduce *non-monotonicity* when deriving negative literals from positive programs, since $\emptyset \mid\!\sim_{\{p \leftarrow q\}} \neg p$, whereas $\{q\} \not\mid\!\sim_{\{p \leftarrow q\}} \neg p$. Consequently, all semantics for logic programs with negation considered in this paper are non-monotonic.

The study of non-monotonic logics is such a large area, that it is impossible to give a complete overview of it in this paper. Therefore, we limit ourselves here to observations that are relevant for logic programming. One may wonder what makes an inference relation *logical* when it is not monotonic. Kraus, Lehmann and Magidor [86] considered the following properties desirable (we omit a number of simple properties that are satisfied by all logic programming semantics in this paper).

Cut:	$\Gamma \mid\!\sim A$ and	$\Gamma \cup \{A\} \mid\!\sim L$ imply	$\Gamma \mid\!\sim L$
Cautious Monotonicity:	$\Gamma \mid\!\sim A$ and	$\Gamma \mid\!\sim L$ imply	$\Gamma \cup \{A\} \mid\!\sim L$
Rationality:	$\Gamma \not\mid\!\sim \neg A$ and	$\Gamma \mid\!\sim L$ imply	$\Gamma \cup \{A\} \mid\!\sim L$

Cautious monotonicity is weaker than rationality (in the presence of simple properties). A logic that satisfies cautious monotonicity and cut is called *cumulative*. We shall use these properties in Sections 4.3, 7.4 and 10.

Dix calls these properties *strong principles* [47, 48], as opposed to certain *weak principles* he identifies [47, 49]; these weak principles are more specific to logic programs, and should be satisfied by every reasonable semantics. Examples of weak principles are the *Principle of Partial Evaluation* (PPE), which roughly means that a positive body literal can be replaced by

[1] Disjunctions of ground atoms in the case of disjunctive logic programs, see Section 10.

[2] SEM can return a single model, a set of models or a theory. In the case of a set of models, the sceptical approach is chosen: $SEM(P) \models L$ if L is true in *all* models in $SEM(P)$.

its definition, and *Relevance*, which means that the truth value of an atom is determined solely by the part of the program that atom depends on (the notion made precise in Definition 2.2).

There are a number of good reasons for adopting a non-monotonic semantics for negation.

- Historically, a classical interpretation of negation was ruled out, because it would result in full first-order theorem proving, with too high a complexity. This argument is hardly valid any more, as the semantics for logic programs with negation studied in this paper are highly undecidable to various degrees in the first-order case (see e.g. Apt and Blair [5], and Cadoli and Schaerf [28] for an overview), but some of them can be computed in polynomial time in the propositional case (see e.g. Van Gelder, Ross and Schlipf [66]).
- In many situations, for example in databases, it is natural to record only positive information, leaving all negative information implicit.
- Recently, researchers in artificial intelligence recognised that commonsense reasoning is non-monotonic. Therefore non-monotonic logics, that is, logics with a non-monotonic inference relation, became popular. Logic programs with non-monotonic negation constitute a small, yet quite expressive class of non-monotonic logics, which is of particular interest because they are implementable. We observed that most motivating examples in papers on the semantics of negation in logic programming are taken from commonsense reasoning.

We distinguish these last two reasons as 'static', respectively 'dynamic' non-monotonicity. Non-monotonicity is used statically, when the available information is complete and can be theoretically, though not practically, captured as classical logic consequences of a theory. The standard example for this case is the already mentioned timetable problem for which it is possible though not practical to list all existing connections and all absent connections. This form of non-monotonicity justifies directly the Closed World Assumption, introduced in Section 1.2.

Non-monotonicity is used dynamically for 'jumping to conclusions' when the available information is incomplete. If, later, more information becomes available, it may turn out that the conclusion is no longer justified, and must be withdrawn. The standard example for this case is that, if we learn that Tweety is a bird, we jump to the conclusion that it can fly, but if we subsequently find out that Tweety is a penguin, we withdraw that conclusion. This use of logic, called *belief revision*, is clearly non-monotonic.

In this example, there is apparently a *default* assumption, namely that birds can fly, unless there is evidence to the contrary. Reiter [145] proposed *Default Logic* as a framework for formalizing such defaults. Also, the example reasons about the *beliefs* of an agent, for which Moore [111] proposed *auto-epistemic logic*. In fact, some semantics for negation in logic programming are closely related to these proposals.

One way of using negation in logic programming for belief revision is by means of *abnormality*-relations. The example of the penguin Tweety can be described by the addition of the fact penguin(Tweety) to the program TWEETY:

bird(Tweety) $\leftarrow$		% Tweety is a bird.
fly(x) $\leftarrow$	bird(x), $\neg$abnormal$_{\mathtt{fly,bird}}$(x)	% Normal birds can fly.
abnormal$_{\mathtt{fly,bird}}$(x) $\leftarrow$	penguin(x)	% Penguins are abnormal birds w.r.t. flying.

We return to this program in Section 11.1; all semantics mentioned in this paper coincide on this program: they derive fly(Tweety) from TWEETY, but not from TWEETY$\cup${penguin(Tweety) $\leftarrow$ }.

1.4 Plan of This Paper

This paper is organized as follows. In the next section we introduce the syntax and discuss the choice of the underlying first-order language. In Section 3 we introduce the basic resolution procedure used for general programs — the SLDNF-resolution. Next, in Section 4 we discuss another classical concept, that of program completion, and discuss soundness and completeness of SLDNF-resolution w.r.t. program completion. Then in Section 5 we return to the SLDNF-resolution by discussing some of its variants and extensions.

In Section 6 we study semantics of general programs by concentrating on 2-valued candidates for a natural model which were proposed in the literature. Then in Section 7 we consider 3-valued options. In Section 8 we relate these special models to various modifications of program completion. Next, in Section 9 we return to the study of proof-theoretic issues and analyze another form of resolution, called SLS-resolution. In particular we discuss there soundness and completeness of SLS-resolution w.r.t. the semantics considered in Section 7 and the issue of its implementation. In Section 10 we discuss disjunctive logic programs, i.e. programs built from clauses whose heads are disjunctions of atoms and relate various approaches to their semantics to the case of general programs.

Finally, in Section 11 we summarize the results of the paper by indicating for which classes of programs all the considered approaches coincide. We also indicate there which topics were not treated in this paper.

2 Preliminaries

2.1 Syntax

We recall the usual definitions. A *literal* is an atom or its negation. A *positive literal* is a synonym for an atom and a *negative literal* is a negated atom. Literals are denoted here by letters L, M. A *general query* is a finite conjunction of literals. (Instead of general queries, one usually considers *general goals* which are expressions $\leftarrow \mathbf{L}$ where $\mathbf{L}$ is a query.) The empty general query is denoted by $\Box$. To adhere to the syntax of logic programming, we write the general query $L_1 \wedge \ldots \wedge L_n$ as $L_1, \ldots, L_n$.

A *general clause* is a construct of the form $H \leftarrow \mathbf{L}$ where H is an atom and $\mathbf{L}$ query; H is called its *head* and $\mathbf{L}$ its *body*. When the body is empty, the general clause is called a *unit clause*. Finally, a *general program* is a finite set of general clauses. We say that a relation p *is defined* in P if it occurs in a head of a general clause of P and that P *uses* a relation q if q occurs in the body of a general clause of P.

As in the paper we shall deal exclusively with general queries, clauses and programs, we omit from now on the qualification "general", unless some confusion arises. When all literals used in the bodies of the program clauses are positive, we call the program *positive*.

As in the case of queries we often use bold letters to denote finite sequence of syntactic objects. Given two sequences of terms $\mathbf{s} = s_1, \ldots, s_n$ and $\mathbf{t} = t_1, \ldots, t_n$ of the same length we abbreviate $s_1 = t_1 \wedge \ldots \wedge s_n = t_n$ to $\mathbf{s} = \mathbf{t}$.

We recall now a number of auxiliary notions.

Definition 2.1 By an *expression* we mean here a term, atom, literal, query, negation of a query or a clause. $Var(E)$ is the set of variables occurring in the expression E, $\forall E$ denotes the universal closure of E, and $\exists E$ the existential closure of E.

A *substitution* θ is a function from variables to terms with a finite domain. Its domain is denoted by $Dom(\theta)$, the set of variables occurring in the terms forming its range by $Ran(\theta)$, and its restriction to the set of variables V by $\theta | V$. For an expression E we abbreviate $\theta | Var(E)$ to $\theta | E$. We write $E\theta$ for the result of applying the substitution θ to the expression E. θ is called a *renaming substitution* for E, if for some substitution η we have $A\theta\eta = A$. ϵ denotes the identity substitution.

The application of a substitution to a (set of) expression(s) and the relation *"more general than"* between the substitutions is defined in the usual way. Given two atoms A and B, a substitution θ is called a *unifier* of A and B if $A\theta \equiv B\theta$, and is called a *most general unifier* (in short: *mgu*) of A and B if it is a unifier which is more general than all unifiers of A and B. Finally, an mgu θ of two atoms A and B is called *relevant* if $Dom(\theta) \cup Ran(\theta) \subseteq Var(A) \cup Var(B)$. □

When studying programs the relationship between the relations used is of importance.

Definition 2.2 (Dependency) Consider a program P.

- The *dependency graph* D_P *for* P is a directed graph with signed edges. Its nodes are the relations occurring in P. For every clause in P which uses relation p in its head and relation q in a positive (resp. negative) literal in its body there is a *positive* (resp. *negative*) edge (p, q) in D_P. We say then that P uses q *positively* (resp. *negatively*).
- We say that p *depends positively* (resp. *negatively*) *on* q if there is a path in D_P from p to q with only positive edges (resp. at least one negative edge).
- We say that p *depends evenly* (resp. *oddly*) *on* q if there is a path in D_P from p to q with an even (resp. odd) number of negative edges. □

2.2 The Universal Query Problem

A simple completeness result for SLD-resolution reads as follows:

> Let P be a positive program, A an atom and θ a substitution. If $P \models \forall A\theta$, then $P \vdash_{SLD} A\sigma$, for some substitution σ such that $A\sigma$ is more general than $A\theta$.

We shall not use classical logic as the semantics for general logic programs, for reasons explained in Section 1.3. In some cases, the semantics of a program will be given by a logical theory, such as the program completion. In many other cases, the semantics of a program will be given by some canonical Herbrand model, such as the least Herbrand model M_P for a positive program P. The relative merits of both approaches are discussed in Wallace [172], among others. When using the canonical model approach, restricting ones attention to *Herbrand* models often leads to considerable technical simplifications. Yet the following statement is *false*:

> Let P be a positive program, A an atom and θ a substitution. If $M_P \models \forall A\theta$, then $P \vdash_{SLD} A\sigma$, for some substitution σ such that $A\sigma$ is more general than $A\theta$.

As a counterexample, take $P = \{p(a) \leftarrow \}$, $A = p(x)$ and $\theta = \epsilon$. Since a is the only term in the Herbrand Universe, $M_P = \{p(a)\} \models \forall x\ p(x)$. There are essentially two ways to avoid this problem.

1. Ensure that the language under consideration has sufficiently many terms. This can be done by
   - adding a clause $p(f(c)) \leftarrow$ to the program P, where p, f and c do not occur in P (as e.g. in Ross [147]).

- postulating, as in Kunen [88], an infinite 'universal' language in which all programs and queries are expressed.

2. Consider arbitrary models instead of only Herbrand models. This approach is taken by Kunen [87] and by Przymusinski [125], who also termed the above problem the *universal query problem*.

In this paper we adopt the 'universal language' approach, as it gives rise to simpler formulations of results than the other approaches. It also solves the problem of how to deal formally with language elements that occur in the query but not in the program. Each approach has its merits and drawbacks. For example, in the case of the approach here adopted, taking the program $P = \{p(a) \leftarrow \}$ again, $\neg\forall x\ p(x)$ holds in the least Herbrand model of P w.r.t. the universal language, whereas it does not hold in all models of P. So now we have the "opposite" of the universal query problem: given the program $\{p(a) \leftarrow ;\ q \leftarrow \neg p(x)\}$, should q be "true"? We leave this problem aside, and for a more detailed discussion of this issue, we refer to Shepherdson [158].

In the sequel, B_P denotes the Herbrand Base of P, M_P the least Herbrand model of a positive program P, and $ground(P)$ the set of all ground instances of clauses from P, all considered w.r.t. this universal language. Finally, by L_P we denote the language defined by the program P, that is the language whose constants, function and relation symbols are those occurring in P.

3 Proof Theory I: SLDNF-resolution

3.1 A Discussion

In order to compute with general programs, one needs to be able to resolve negative literals. A natural idea is to use the *closed world assumption*, that is to stipulate for an atom A

$$\neg A \text{ succeeds iff } A \text{ cannot be proved.}$$

The problem with this rule is that it is in general undecidable whether an atom can (cannot) be proved, even if we restrict our attention to positive programs. Later, in Section 9 we shall consider an ineffective form of resolution which formalizes the above idea.

To make the above rule effective Clark [39] proposed to replace the statement "A cannot be proved" by its finitary version, the *negation as finite failure* rule, which makes this rule decidable. So, according to Clark [39], the statement "A cannot be proved" should be interpreted as "A finitely fails".

However, for general programs the considered trees for a query A can contain negative literals, so the question now arises when these literals fail. A natural idea is to stipulate that for an atom A,

$$\neg A \text{ fails iff } A \text{ can be proved.}$$

Interpreting the statement "A can be proved" as "there exists a successful derivation for the query A", we end up with a resolution method, called *SLDNF-resolution*, which is appropriate for general programs and general queries. It should be mentioned here that another interpretation of the above statement is possible which leads to another form of resolution. We shall consider it in Section 5.1.

Thus according to the SLDNF-resolution, when the selected literal is positive, the usual SLD-like procedure is used to obtain a new resolvent, and when the selected literal, say $\neg A$, is negative, the following rule is used to obtain the new resolvent:

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

That is, if $\neg A$ succeeds, it is deleted from the query, and if it finitely fails the query fails.

As in the case of the SLD-resolution, this notion of resolution can be used not only to prove but also to compute. Let us introduce the following notation.

- $P \vdash_{SLDNF} \forall\, Q\theta$ if there exists a successful SLDNF-derivation of $P \cup \{Q\}$ with computed answer θ,
- $P \vdash_{SLDNF} \forall\, \neg Q$ if there exists a finitely failed SLDNF-tree for $P \cup \{Q\}$.

Without any restrictions the above notion of SLDNF-resolution becomes a problematic concept. Indeed, take the following program NUMBERS $= \{positive(x) \leftarrow \neg zero(x);\ zero(0) \leftarrow \}$. Then the query $zero(x)$ succeeds, so $\neg zero(x)$ finitely fails and consequently $positive(x)$ finitely fails, as well. Thus NUMBERS $\vdash_{SLDNF} \forall\, x\, \neg positive(x)$. However, for any ground term t different from 0, $zero(t)$ finitely fails, so $positive(t)$ succeeds. Thus NUMBERS $\vdash_{SLDNF} positive(t)$. This excludes any soundness results. In fact, these conclusions will be drawn by most Prolog systems. So Prolog is not "sound".

The problem is caused by the use of variables in nonground negative literals. To ensure soundness Clark [39] imposed the restriction that only ground negative literals can be selected.

However, the definition of the SLDNF-resolution sketched above is difficult to formalize. Consider for example the program $P = \{p \leftarrow p\}$. The query $\neg p$ neither succeeds nor finitely fails, since the query p neither succeeds nor finitely fails. So it is not clear whether there is a resolvent. (This also shows that SLDNF-resolution is incomplete, since neither $P \vdash_{SLDNF} p$ nor $P \vdash_{SLDNF} \neg p$ holds here.) The problem is that success and finite failure are not the only possible outcomes of an evaluation: also an unsuccessful tree which is not finitely failed can be generated.

This problem was not properly taken care of in the definition of SLDNF-resolution given in Clark [39] and reproduced in Lloyd [92]. In Lloyd [93] a revised definition of SLDNF-resolution was proposed according to which the SLDNF-trees are constructed "bottom-up" by induction on the number of alternations through negation. Unfortunately, according to this definition for the above mentioned example and some other problematic cases no SLDNF-trees or SLDNF-derivations exist. This is clearly undesirable, especially if one reasons about "run time" properties of the SLDNF-resolution, like termination.

These problems were first tackled by Martelli and Tricomi [106] who proposed a revision of the original definition in which the subsidiary trees used to resolve negative literals are built "inside" the main tree. The solution presented here is due to Apt and Doets [11].

3.2 A New Definition

Definition 3.1 (Resolvent)
(i) We say that Q *resolves to* Q' *via* α *w.r.t.* Σ, or: Q' (more explicitly, the pair (α, Q')) is a *resolvent* of Q w.r.t. Σ, notation: $Q \overset{\alpha}{\Longrightarrow} Q'\ (\Sigma)$, if
either: $\Sigma = (L, R)$, L is (an occurrence of) a positive literal in Q, R is a program clause, and for some variant $H \leftarrow \mathbf{L}$ (the *input clause*) of R: α is mgu of L and H and $Q' = Q\alpha[L\alpha := \mathbf{L}\alpha]$ is obtained from $Q\alpha$ by replacing $L\alpha$ by $\mathbf{L}\alpha$,
or: Σ is (an occurrence of) a negative literal in Q, $\alpha = \epsilon$, and $Q' = Q - \{\Sigma\}$ is obtained from Q by removing Σ.

(ii) A clause R is called *applicable to* an atom if it has a variant the head of which unifies with the atom. □

Definition 3.2 (Pseudo derivation) A (finite or infinite) sequence $Q_0 \stackrel{\alpha_1}{\Longrightarrow} \cdots Q_n \stackrel{\alpha_{n+1}}{\Longrightarrow} Q_{n+1} \cdots$ of resolution steps is a *pseudo derivation* if for every step involving a program clause:

- ("standardisation apart") the input clause employed does not contain a variable from the initial query Q_0 or from an input clause used at some earlier step,
- ("relevance") the mgu employed is relevant. □

Intuitively, an SLDNF-derivation is a pseudo derivation in which the deletion of every (ground) negative literal is justified by means of a subsidiary (finitely failed SLDNF-) tree. This brings us to consider special types of trees.

Definition 3.3 A tree is called

- *successful* if it contains a leaf marked as *success*,
- *finitely failed* if it is finite and all its leaves are marked as *failed.* □

In the sequel we consider systems of trees called *forests.*

Definition 3.4 (Forest) 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,
- *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 a child of N_i in some tree in $\mathcal{F}$ or the root of the tree $subs(N_i)$. □

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 SLDNF-tree is a special type of a forest built as a limit of certain finite forests: *pre-SLDNF trees.* The nodes of these trees are labeled by queries. Below we shall identify a node with its label.

The construction begins with the main tree which consists of just one node – the original query. During the construction new, subsidiary, trees can be added. In each "round" the branches of all trees are extended in parallel. The final object is an SLDNF-tree. As in the original definition of Clark [39] the subsidiary trees are kept "aside" of the "main" tree. The difference is that their construction is no longer viewed as an atomic step in the resolution process. If a subsidiary tree T becomes successful or finitely failed, this information is used in the "next round" of the extension process to determine the status of the query which originated the construction of T.

For the rest of this section, we fix a program P. The next definition is crucial.

Definition 3.5 (Pre-SLDNF-tree) A *pre-SLDNF-tree* (relative to P) is a forest whose nodes are (possibly marked) queries of (possibly marked) literals. (For queries, there are markers *failed*, *success*, and *floundered*; for literals, we have the marker *selected.*) The function *subs* assigns to nodes containing a marked negative ground literal $\neg A$ a tree in $\mathcal{F}$ with root A. The class of pre-SLDNF-trees is defined inductively.

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

Here, an *extension* of a pre-SLDNF-tree $\mathcal{F}$ is defined by performing the following actions for every non-empty query C which is an unmarked leaf in some tree $T \in \mathcal{F}$:

First, if no literal in C is marked yet as *selected*, mark one as *selected.* Let L be the selected literal of C.

- L is positive.
  - C has no resolvents w.r.t. L and a clause from P.
    Then C is marked as *failed*.
  - C has such resolvents.
    For every clause R from P which is applicable to L, choose one resolvent (α, D) of C w.r.t. L and R and add this as child of C in T. These resolvents are chosen in such a way that all branches of T remain pseudo derivations.
- $L = \neg A$ is negative.
  - A is nonground. Then C is marked as *floundered*.
  - A is ground.
    * *subs*(C) is undefined.
      Then a new tree T' with the single node A is added to $\mathcal{F}$ and *subs*(C) is set to T'.
    * *subs*(C) is defined and successful.
      Then C is marked as *failed*.
    * *subs*(C) is defined and finitely failed.
      Then the resolvent $(\epsilon, C - \{L\})$ of C is added as the only child of C in T.

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

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

Every pre-SLDNF-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 clear meaning.

Definition 3.6 (SLDNF-tree)

- An *SLDNF-tree* is a limit of a sequence $\mathcal{F}_0, \ldots, \mathcal{F}_i, \ldots$ such that $\mathcal{F}_0$ is an initial pre-SLDNF-tree, and for all i, $\mathcal{F}_{i+1}$ is an extension of $\mathcal{F}_i$.
- *An SLDNF-tree for C* is an SLDNF-tree in which C is the root of the main tree.
- A (pre-)SLDNF-tree is called *successful* (resp. *finitely failed*) if the main tree is successful (resp. *finitely failed*).
- An SLDNF-tree is called *finite* if no infinite paths exist in it. □

Next, we define the concept of SLDNF-derivation.

Definition 3.7 (SLDNF-derivation) A *(pre-) SLDNF-derivation for C* is a branch in the main tree of a (pre-) SLDNF-tree $\mathcal{F}$ for C together with the set of all trees in $\mathcal{F}$ whose roots can be reached from the nodes of this branch. It is called *successful* if it ends with the empty query. An SLDNF-derivation is called *finite* if all paths of $\mathcal{F}$ fully contained within this branch and these trees is finite. □

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

Definition 3.8 (Computed answer substitution) Consider a branch in the main tree of a (pre-) SLDNF-tree $\mathcal{F}$ for C which ends with the empty query. Let $\alpha_1, \ldots, \alpha_n$ be the consecutive substitutions along this branch.

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

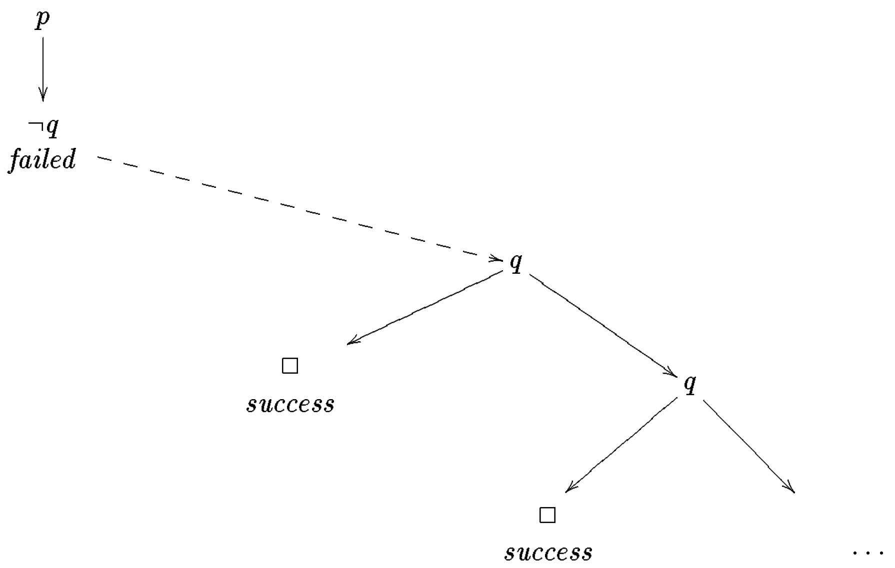
Let us illustrate the above definitions by depicting the SLDNF-trees for two "difficult" cases.

Example 3.9 (Infinite SLDNF-trees)
(i) Consider the "problematic" case of $P = \{p \leftarrow p\}$ and $C = \neg p$ mentioned in Section 3.1. The only SLDNF-tree has then the following form:

$$\begin{tikzcd}
\neg p \arrow[dr, dashed] & \\
& p \arrow[d] \\
& p \arrow[d] \\
& \cdots
\end{tikzcd}$$

(ii) It is important to realize that according to this definition the construction of a subsidiary tree can go on forever even if the information about its "status" has already been passed to the main tree. The following program illustrates this point.

Consider $P = \{p \leftarrow \neg q;\ q \leftarrow\ ;\ q \leftarrow q\}$. Then the only SLDNF-tree for p looks as follows:



Here the subsidiary tree with the root q grows forever. However, once an extension of the initial subsidiary tree with the single node q becomes successful, in the next extension the node $\neg q$ is marked as *failed*. Consequently, the SLDNF-tree for p is finitely failed even though it is not finite. □

Now note the following simple result.

Theorem 3.10 (Limit)

(i) Every SLDNF-tree is the limit of a unique sequence of pre-SLDNF-trees.

(ii) If the SLDNF-tree $\mathcal{F}$ is the limit of the sequence $\mathcal{F}_0, \ldots, \mathcal{F}_i, \ldots$, then for all τ

(a) $\mathcal{F}$ is successful and yields τ as c.a.s. iff some $\mathcal{F}_i$ is successful and yields τ as c.a.s.,

(b) $\mathcal{F}$ is finitely failed iff some $\mathcal{F}_i$ is finitely failed. □

This result allows us to associate with every successful or finitely failed SLDNF-tree $\mathcal{F}$ a natural number, $rank(\mathcal{F}, \tau)$, which is the least i for which the corresponding equivalence in (ii) holds, with $\tau = \epsilon$ when $\mathcal{F}$ is finitely failed. This measure is useful for carrying out inductive proofs about SLDNF-resolution.

Finally, let us mention that it is straightforward to show that if a successful SLDNF-derivation or finitely failed SLDNF-tree exists according to the definition given in Lloyd [93], then so it does according to the definition here presented.

3.3 Floundering

For further discussion it is useful to introduce the following notion. An SLDNF-tree $\mathcal{F}$ is *via* a selection rule R if in the sequence of pre-SLDNF-trees whose limit is $\mathcal{F}$ all the markings of literals as *selected* are performed according to R. A *selection rule* is a function which given a pre-SLDNF-tree F selects a literal in every non-empty unmarked leaf in some tree of F. A selection rule is called *safe* if it never selects a nonground negative literal.

One of the complications concerning SLDNF-resolution is so-called "floundering" — a generation of a node which consists exclusively of nonground negative literals, as then selection of any literal ends the derivation in an abnormal way. In the definition here provided floundering is treated differently — it arises as soon as a nonground negative literal is selected. Clearly, this small change has no effect on the theory of SLDNF-resolution, since the original notion of floundering can be easily defined.

Definition 3.11 (Floundering)

- We call a query *blocked* if it consists exclusively of nonground negative literals.
- We say that P and Q *flounder* if some SLDNF-tree for P and Q contains a blocked node.

□

Note the difference between a blocked node and a node marked as floundered. Thus an SLDNF-tree via a safe selection rule does not flounder. Börger [27] (see Apt [2] for a more direct proof) proved that it is undecidable whether P and Q flounder. In the literature a number of syntactic conditions was proposed which ensure that a program and a query do not flounder. The following notion due to Lloyd and Topor [95] (see also Lloyd [93]) has become best known.

Definition 3.12 (Allowedness)

- A query $\mathbf{L}$ is called *allowed* if every variable of it occurs in a positive literal.
- A clause $H \leftarrow \mathbf{L}$ is called *allowed* if $\neg H, \mathbf{L}$ is.
- A program is called *allowed* if all its clauses are. □

Thanks to the use of the new definition, the following result of Lloyd and Topor [95] now refers to a larger class of SLDNF-trees.

Theorem 3.13 *Suppose that P and Q are allowed. Then*

- *P and Q do not flounder.*
- *if θ is a c.a.s. of Q, then $Q\theta$ is ground.* □

When θ is a c.a.s. of Q such that $Q\theta$ is ground, we say that θ is a *ground computed answer substitution*. Actually, the definition of allowedness proposed in Lloyd and Topor [95] is slightly more general than the one we considered. Even this stronger version excludes many natural logic programs, because every allowed unit clause is ground and every computed answer is grounding. Decker and Cavedon [43] and Decker [42] proposed more general syntactic conditions which prevent floundering.

3.4 Kunen's Definition

Kunen [88] provided a much simpler definition of the computed answer substitutions and finitely failed queries of the SLDNF-resolution and used it to prove completeness in the sense discussed in the next section for allowed programs and allowed queries. We now present his definition and compare it with the one given before.

Definition 3.14 The set $\mathbf{F}$ of queries and the set $\mathbf{R}$ of pairs (C, σ), where C a query and σ a substitution for which $Dom(\sigma) \subseteq Var(C)$, are defined by a simultaneous inductive definition as follows.

0) $\Box \mathbf{R} \epsilon$,

R+) if C resolves to D via α w.r.t. some positive literal of C and a clause from P and $D\mathbf{R}\sigma$, then $C\mathbf{R}(\alpha\sigma)|C$,

R−) if A is a ground atom in $\mathbf{F}$ and $(C, C')\mathbf{R}\sigma$, then $(C, \neg A, C')\mathbf{R}\sigma$,

F+) if L is a positive literal in C and for every clause R from P which is applicable to L there exist α and $D \in \mathbf{F}$ such that $C \stackrel{\alpha}{\Longrightarrow} D$ (L, R), then $C \in \mathbf{F}$,

F−) if A is a ground atom such that $A\mathbf{R}\epsilon$, then $(C, \neg A, C') \in \mathbf{F}$. □

Recall that for a query C, $\theta | C$ stands for the restriction of the substitution θ to the variables of C. The intention here is that $\mathbf{R}$ is the set of pairs (C, σ) such that σ is a c.a.s. for C and $\mathbf{F}$ is the set of queries C such that there is a finitely failed tree for C.

Note that the formulation of R+) does *not* ensure that the resulting answer substitutions are most general. Indeed, consider the following program

$Q(x, y) \leftarrow Q(y, y)$,

$Q(x, x) \leftarrow$.

Then $\Box \mathbf{R} \epsilon$ by 0), $Q(y, y)\mathbf{R}\{y/x\}$ by R+) and the second clause and consequently $Q(x, y)\mathbf{R}\{y/x\}$ by R+), since $Q(x, y)$ resolves to $Q(y, y)$ via ϵ and the first clause. But $\{y/x\}$ is not a c.a.s. for $Q(x, y)$ whereas $\{y/x'\}$ is.

In order that R+) produces most general answer substitutions, we amend it as follows:

†R+) if C resolves to D via α w.r.t. some positive literal of C and a clause from P, $D\mathbf{R}\sigma$, *and*

$$Var(C\alpha) \cap Var(D\sigma) \subseteq Var(D),$$

then $C\mathbf{R}(\alpha\sigma)|C$.

The following theorem of Apt and Doets [11] shows the equivalence between the SLDNF-resolution and Kunen's definition as modified above.

Theorem 3.15 (Equivalence) *If C is a query, then*

- *$C\mathbf{R}\tau$ iff τ is a c.a.s. for C,*
- *$C \in \mathbf{F}$ iff C has a finitely failed SLDNF-tree.* □

3.5 Termination

It is natural to ask then what is the use of the definition of SLDNF-resolution given in Section 3.2. To show its usefulness we now consider the issue of termination which cannot be handled using Kunen's approach.

Definition 3.16 (Terminating program) A program is called *terminating* if all its SLDNF-trees for ground queries are finite. □

Of course, in general one is actually interested in proving termination of a given program not only for all ground queries but also for a class of nonground queries constituting the intended queries. The approach to prove termination discussed here allows us to identify for each program such a class of nonground queries. To characterize terminating programs, following Cavedon [29] and Apt and Bezem [4] we introduce the following notions.

Definition 3.17 (Acyclic program)

- A *level mapping* for a program P is a function $|\ |: B_P \rightarrow N$ of ground atoms to natural numbers. For $A \in B_P$, $|A|$ is the *level of* A.
- Given a level mapping $|\ |$, we extend it to ground negative literals by putting $|\neg A| = |A|$.
- A clause of P is called *acyclic with respect to a level mapping* $|\ |$, if for every ground instance $A \leftarrow \mathbf{K}, L, \mathbf{L}$ of it
$$|A| > |L|.$$
- A program P is called *acyclic with respect to a level mapping* $|\ |$, if all its clauses are. P is called *acyclic* if it is acyclic with respect to some level mapping. □

Definition 3.18 (Boundedness)

- A literal L is called *bounded with respect to a level mapping* $|\ |$, if $|\ |$ is bounded on the set $[L]$ of ground instances of L. For L bounded w.r.t. $|\ |$, we define $|L|$, the *level of* L w.r.t. $|\ |$, as the maximum $|\ |$ takes on $[L]$.
- A query is called *bounded with respect to a level mapping* $|\ |$, if all its literals are. For $Q = L_1, \ldots, L_n$ bounded w.r.t. $|\ |$, we define $|Q|$, the *level of* Q w.r.t. $|\ |$, as the multiset $bag(|L_1|, \ldots, |L_n|)$. □

The following result explains why bounded queries are relevant.

Lemma 3.19 (Finiteness) *Let P be an acyclic program and Q a bounded query. Then every SLDNF-tree for P and Q is finite.* □

This leads to the following conclusion.

Corollary 3.20 *Every acyclic program is terminating.* □

Further work on this subject was done by Ross [149] and on termination of programs w.r.t. SLDNF-resolution with the leftmost selection rule of Prolog by Apt and Pedreschi [6]. We return to acyclic programs in Section 11.1.

4 Program Completion

4.1 Definition

In first-order logic the soundness and completeness results equate the notions of semantic and proof-theoretic implication: for every set of formulas $T \cup \{\phi\}$ we have

$$T \models \phi \text{ iff } T \vdash \phi.$$

A similar result cannot be established for the SLDNF-resolution and the programs. Indeed, using SLDNF-resolution we can prove ground negative literals, but all of them are false in the largest Herbrand model of a program, B_P.

Clark [39] proposed to solve this problem by strengthening a program P to its *completion*, $comp(P)$ and compare the SLDNF-resolution with $comp(P)$. Intuitively, in the completion the implications are replaced by equivalences. The formal definition is a bit subtle, since this replacement has to be made at the right moment, and the equality relation has to be interpreted in an appropriate way. We recall here the definition.

First, assume that "=" is a new binary relation symbol not appearing in P. We write $s \neq t$ as an abbreviation for $\neg(s = t)$. "=" is interpreted as identity in all models.

We perform successively the following steps, where $x_1, \ldots, x_n, \ldots$ are new variables.

Step 1: Transform each clause $p(\mathbf{t}) \leftarrow \mathbf{L}$ of P into $p(\mathbf{x}) \leftarrow \mathbf{x} = \mathbf{t} \wedge \mathbf{L}$.

Step 2: Transform each formula $p(\mathbf{x}) \leftarrow F$ obtained in the previous step into $p(\mathbf{x}) \leftarrow \exists \mathbf{y} F$, where $\mathbf{y}$ are the variables of the original clause.

Step 3: Let $p(\mathbf{x}) \leftarrow \exists \mathbf{y} F_1, \ldots, p(\mathbf{x}) \leftarrow \exists \mathbf{y} F_k$ be all formulas obtained in the previous step with a relation p on the left-hand side. Replace them by one formula $p(\mathbf{x}) \leftarrow F_1 \vee \ldots \vee F_k$. If $F_1 \vee \ldots \vee F_k$ is empty, replace it by **true**.

Step 4: For each relation symbol q not appearing in a head of a clause in P add a formula $q(\mathbf{x}) \leftarrow \mathbf{false}$.

Step 5: Replace each formula $p(\mathbf{x}) \leftarrow F$ by $\forall \mathbf{x}(p(\mathbf{x}) \leftarrow F)$.

Step 6: In each formula replace " $\leftarrow$ " by " $\leftrightarrow$ ".

Additionally, we add the following *free equality axioms*, EQ, which enforce that the equality theory of $comp(P)$ is the same as that of Herbrand universe:

(1) $f(\mathbf{x}) = f(\mathbf{y}) \rightarrow \mathbf{x} = \mathbf{y}$ for each function symbol f,

(2) $f(\mathbf{x}) \neq g(\mathbf{y})$ for all function symbols f and g such that $f \neq g$,

(3) $x \neq t$ for each variable x and term t such that $x \not\equiv t$ and x occurs in t,

and call the resulting set of formulas $comp(P)$.

Additionally, we interpret "=" in all 2-valued and 3-valued interpretations as identity. This allows us to dispose of the usual equality axioms.

4.2 Two-valued Model Theory

While program completion is a natural concept in the case of positive programs, in the case of general programs things dramatically change, due to the following disturbing observation.

Note 4.1 For $P = \{p \leftarrow \neg p\}$, $comp(P)$ is inconsistent. □

As inconsistent program completion allows us to derive arbitrary first-order formulas from the program, the above note seems to rule out the use of program completion to model negative information.

Before we discuss some ways of resolving this difficulty, it is useful to recall the *immediate consequence operator* T_P of Van Emden and Kowalski [169] which acts on Herbrand interpretations of a given program. This operator plays an important role in the theory of positive programs.

Definition 4.2 (Immediate consequence operator) For a program P and a Herbrand interpretation I for P we define

$$T_P(I) = \{H \mid \exists\, \mathbf{L}\ (H \leftarrow \mathbf{L} \in ground(P),\ I \models \mathbf{L})\}. \qquad \square$$

The following simple observation (originally made for positive programs) by Van Emden and Kowalski [169] explains the interest in this operator by characterizing the Herbrand models of P in terms of the operator T_P.

Lemma 4.3 *For every Herbrand interpretation I, $I \models P$ iff $T_P(I) \subseteq I$.* $\square$

A bit more complicated argument (originally made for positive programs) by Apt and Van Emden [8] characterizes the Herbrand models of *comp*(P) in terms of the operator T_P.

Lemma 4.4 (Fixpoint) *For every Herbrand interpretation I, $I \models comp(P)$ iff $T_P(I) = I$.* $\square$

For positive programs T_P exhibits a very regular behaviour –w.r.t. the set inclusion it is monotonic ($I \subseteq J$ implies $T_P(I) \subseteq T_P(J)$) and continuous (for every infinite sequence $I_0 \subseteq I_1 \subseteq \ldots$, $T_P(\bigcup_{n=0}^{\infty} I_n) = \bigcup_{n=0}^{\infty} T_P(I_n)$). Thanks to the first property the least Herbrand model M_P is the $\subseteq$-least fixpoint of T_P and thanks to the second property this model can be reached in ω iterations of T_P starting with the empty Herbrand interpretation.

For general programs both properties of T_P are lost. Indeed, consider again $P = \{p \leftarrow \neg p\}$. Then $T_P(\emptyset) = \{p\}$, whereas $T_P(\{p\}) = \{\emptyset\}$, so T_P is not monotonic and a fortiori not continous. Consequently, for general programs the well-known Knaster-Tarski theorem cannot be used to find a fixpoint of T_P. In fact, the fixpoints need not to exist: just take T_P for $P = \{p \leftarrow \neg p\}$.

A natural question is under what conditions completion is consistent. The following result was established by Sato [153].

Definition 4.5 (Call-consistent) A program is called *call-consistent* if no relation depends oddly on itself.

Theorem 4.6 *If P is call-consistent, then comp(P) has a Herbrand model.* $\square$

Further work on the subject of consistency of *comp*(P) can be found in Kunen [88], Cavedon [30] Cortesi and Filé [40], Cortesi and Filé [41], Baratella [17] and Fages [60].

An alternative solution is to use three valued logic.

4.3 Three-valued Model Theory

Fitting [61] proposed to use a 3-valued logic to provide semantics to programs and their completions. The idea is that a query can yield *three* outcomes: it may succeed, it may fail, and it may also diverge. The third value is meant to capture the last possibility.

Fitting [61] based his approach on a logic due to Kleene [83], in which 3 values are assumed: $\{0,\ \frac{1}{2},\ 1\}$, 0 representing **false**, $\frac{1}{2}$ representing **unknown** and 1 representing **true**. Assume now a mapping $|\ |$ from B_P to $\{0,\ \frac{1}{2},\ 1\}$. To define the meaning of the programs we put for ground quantifier-free formulas

$$\begin{aligned}
|\neg A| &= 1 - |A|, \\
|A \wedge B| &= \min(|A|,\ |B|), \\
|A \leftarrow B| &= \begin{cases} 1 & \text{if } |A| \geq |B|, \\ 0 & \text{if } |A| < |B| \end{cases}
\end{aligned}$$

and identify the program with the set $ground(P)$. Note that " $\leftarrow$ " received here a 2-valued interpretation. (Actually in Fitting [61] the valuation of " $\leftarrow$ " is not used. The above interpretation differs from that of Kleene [83] and was later added in Przymusinski [130].)

For the moment the meaning of other connectives is not needed. When a ground formula evaluates to 1, we say it is *true relative to* $|\ |$, and when it evaluates to 0, we say it is *false relative to* $|\ |$.

The mapping $|\ |$ can be conveniently presented in the form of a 3-valued Herbrand interpretation.

Definition 4.7 A pair $I = (I^+, I^-)$, with $I^+, I^- \subseteq B_P$, is called a *3-valued Herbrand interpretation.* I^+ are atoms assumed true, and I^- are atoms assumed false. $\square$

For example when $I = (\{A\}, \{B\})$, then A and $\neg B$ are true in I, B and $\neg A$ are false in I, and C and $\neg C$ are undefined in I.

Definition 4.8

- I is *total* if $I^+ \cup I^- = B_P$,
- I is *consistent* if $I^+ \cap I^- = \emptyset$. $\square$

Note that every (2-valued) Herbrand interpretation I can be identified with the 3-valued, total, consistent Herbrand interpretation $(I, B_P - I)$ in the sense that truth and falsity coincides in both interpretations for all formulas.

The following natural ordering on 3-valued on Herbrand interpretations

$$I \subseteq J \text{ iff } I^+ \subseteq J^+ \text{ and } I^- \subseteq J^-$$

formalizes the intuition: J contains *more* information than I (determines status of more literals). This ordering is usually called "information ordering". Other natural orderings can be considered – see e.g. Section 7.

Note that *both* truth and falsity behave monotonically w.r.t. the information ordering in the following sense.

Lemma 4.9 *Let $I \subseteq J$. Then for a ground query Q, Q is true (false) in I implies that Q is true (false) in J.* $\square$

These two implications do not hold for 2-valued Herbrand interpretations and $\subseteq$ interpreted as set-theoretical inclusion. Also, in contrast to 2-valued Herbrand interpretations, consistent 3-valued Herbrand interpretations with the $\subseteq$ ordering do *not* form a lattice. Indeed, if I and J are total and $I \neq J$, then $I \cup J$ does not exist. However, consistent 3-valued Herbrand interpretations do form a *cpo*, that is a partial ordering in which the limits of growing chains exist. This is sufficient for building 3-valued models inductively.

Following Fitting [61] we now introduce a 3-valued analogue of the T_P operator (originally denoted by Φ_P), which acts on 3-valued Herbrand interpretations of a given program.

Definition 4.10 (Immediate consequence operator) For a program P and a 3-valued Herbrand interpretation I for P we define

$$T3_P(I) = (T, F),$$

where

$$T = \{H \mid \exists\, \mathbf{L}\ (H \leftarrow \mathbf{L} \in ground(P),\ \mathbf{L} \text{ is true in I})\},$$
$$F = \{H \mid \forall\, \mathbf{L}\ (H \leftarrow \mathbf{L} \in ground(P) \text{ implies } \mathbf{L} \text{ is false in I})\}.$$
$\square$

The following lemma summarizes the relevant properties of the $T3_P$ operator.

Lemma 4.11

- *If I consistent, then $T3_P(I)$ consistent,*
- *$T3_P$ is monotonic,*
- *In general $T3_P$ is not continuous.* □

Let us return now to the program completion. To define its meaning in 3-valued logic we need also to assign meaning to disjunction, equivalence and quantifiers. We do it as follows:

$$
\begin{array}{rcl}
|A \vee B| & = & \max(|A|,\ |B|), \\
|A \leftrightarrow B| & = & \begin{cases} 1 & \text{if } |A| = |B|, \\ 0 & \text{if } |A| \neq |B|, \end{cases}
\end{array}
$$

so " $\leftrightarrow$ " as " $\leftarrow$ " receives a 2-valued interpretation. The quantifiers are interpreted in the standard way. This definition allows us to determine when a first-order formula ϕ is true in an arbitrary 3-valued interpretation I, written as $I \models_3 \phi$. In analogy with the 2-valued semantics, we also use the $\models_3$ relation to state that a formula is true in all 3-valued models of a theory (e.g. $comp(P) \models_3 Q$).

The Fixpoint Lemma 4.4 has a counterpart for the 3-valued case.

Lemma 4.12 (Fixpoint) *For every Herbrand interpretation I, $I \models_3 comp(P)$ iff $T3_P(I) = I$.* □

Consequently, by Lemma 4.11 and the generalization of the Knaster-Tarski Theorem to cpo's we get

Corollary 4.13 *The $\subseteq$-least fixpoint of $T3_P$ is a consistent 3-valued model of $comp(P)$.* □

For example, for the program $P = \{p \leftarrow \neg p\}$ we now get a 3-valued model, namely $(\emptyset, \emptyset)$, in which every ground atom is undefined, and consequently in which $p \leftrightarrow \neg p$ is true. Thus the 3-valued logic approach offers a solution to the problem of possible inconsistency of completion w.r.t. 2-valued logic.

A natural question is for which programs the 3-valued and 2-valued semantics of $comp(P)$ coincide. An answer was provided by Kunen [88].

Definition 4.14 (Strictness) Consider a program P and a query Q. We say that P *is strict w.r.t.* Q if no relation occurring in Q depends both evenly and oddly on a relation defined in the program.

Theorem 4.15 (Equivalence) *Suppose that P is call-consistent and P is strict w.r.t. Q. Then $comp(P) \models_3 Q$ iff $comp(P) \models Q$.* □

It was shown by Dix [44] that the two-valued completion semantics does not satisfy cautious monotonicity, but that the three-valued completion semantics is rational. For the first statement consider the following program $P = \{q \leftarrow \neg p;\ q \leftarrow r;\ p \leftarrow q;\ p \leftarrow r;\ r \leftarrow r\}$. Then $comp(P) \models \{p, q, r\}$ but $comp(P \cup \{p\}) = Th(\{p, q \leftrightarrow r\}) \not\models \{q, r\}$.

For a further discussion of the program completion we refer the reader to Section 8.

4.4 Soundness and Completeness Results

Let us relate now SLDNF-resolution and program completion. Clark [39] proved soundness of the SLDNF-resolution w.r.t. 2-valued semantics of program completion. In fact (see Shepherdson [158] for a sketch and Doets [50] for a complete proof), soundness holds also w.r.t. 3-valued semantics. More precisely, we have the following result.

Theorem 4.16 (Soundness) *Given a program P and a query Q we have*

- *if τ is a c.a.s. for Q, then $comp(P) \models_3 \forall\, Q\tau$,*
- *if there is a finitely failed SLDNF-tree for Q, then $comp(P) \models_3 \forall\, \neg Q$.* □

A lot of effort has been devoted to establish some sort of completeness of SLDNF-resolution. Already Clark [39] noticed that when comparing SLDNF-resolution with $comp(P)$ some restrictions are necessary. For example, for $P = \{p \leftarrow q,\ p \leftarrow \neg q,\ q \leftarrow q\}$ we have $comp(P) \models p$ but no successful SLDNF-derivation exists. In this example p depends both positively and negatively on q. The definition of strictness was designed to avoid this type of situations. Cavedon and Lloyd [31] established a conjecture of Apt, Blair, Walker [9] and proved completeness of SLDNF-resolution w.r.t. 2-valued semantics of $comp(P)$ for allowed P and Q such that P is strict w.r.t. Q and P is stratified (the concept to be introduced in Section 6). Independently Kunen [88] established the following stronger result which refers to 3-valued semantics.

Theorem 4.17 (Completeness I) *Suppose that P and Q are allowed. Then*

- *if $comp(P) \models_3 \forall Q\theta$, then $Q\mathbf{R}\theta$,*
- *if $comp(P) \models_3 \forall \neg Q$, then $Q \in \mathbf{F}$.* □

A crucial lemma for establishing this completeness theorem, and numerous generalizations of it discussed further in the text, is the following result of Kunen [87] which allows us to set up induction in a proper way.

Lemma 4.18 *For every first-order formula ϕ not containing $\leftarrow$ and $\leftrightarrow$ we have*

$$comp(P) \models_3 \phi \text{ iff } T3_P \uparrow n \models_3 \phi \text{ for some finite } n.$$ □

$T3_P \uparrow n$ denotes here the n-fold iteration of the operator $T3_P$ starting at the empty 3-valued interpretation $(\emptyset, \emptyset)$. In this lemma non-Herbrand models of $comp(P)$ are used in an essential way. Also, as noted by Shepherdson [158] this lemma critically depends on the existence of infinitely many function symbols (counting constants as 0-ary function symbols), a property satisfied by the universal language adopted in this paper. When the used language has only finitely function symbols the free equality axioms have to be appropriately strenghtened.

Recently, Doets [50] provided a simpler presentation of its proof. See also Stärk [161] for another proof.

When $comp(P) \models \forall Q\theta$ (resp. $comp(P) \models_3 \forall Q\theta$) we say that θ is a *2-valued* (resp. *3-valued*) *correct answer substitution* for Q. Additionally, when $Q\theta$ is ground, we say that θ is a *ground correct answer substitution.* The Completeness I Theorem 4.17 in conjunction with Theorems 3.13 and the Equivalence Theorem 3.15 implies that 3-valued correct answer substitutions for allowed programs and queries are ground. Shepherdson [157] showed that this claim also holds for the 2-valued case for allowed programs whose completion is consistent.

A problem with the above completeness result is that, as already mentioned at the end of Section 3.3, the class of allowed programs is quite restricted and excluded many natural Prolog programs. So a natural question arises how to generalize the above completeness result to a larger class of programs. This problem was studied by several researchers.

By providing more general conditions preventing floundering Decker and Cavedon [43] and Decker [42] generalized the Completeness I Theorem 4.17 to a larger class of programs. Cavedon [30] proved completeness of SLDNF-resolution for acyclic programs which subsumes an early result of Clark [39] who (essentially) proved completeness w.r.t. 2-valued completion for recursion-free programs which satisfy a syntactic condition which prevents floundering. Numerous other extensions of the Completeness I Theorem 4.17 were obtained by modifying the underlying computation mechanism, so the SLDNF-resolution.

5 Proof Theory II: SLDNF-resolution Revisited

We explained in Section 3.1 why in the definition of SLDNF-resolution only ground negative literals are allowed to be selected. In this section we discuss how this restriction can be imposed or modified.

5.1 Modifications of SLDNF-resolution

An interesting theoretical alternative is to modify the SLDNF-resolution by allowing the selection of nonground negative literals under certain circumstances. Consider the following modification of the definition of SLDNF-resolution, already mentioned in Clark [39]. Let $L = \neg A$ be the selected literal of a query C. If there exists an empty c.a.s. for the query A, then C is marked as *failed*. If the subsidiary tree $subs(C)$ is defined and finitely failed, then $C - \{L\}$ is the only child of C. In terms of Kunen's definition (Definition 3.14) this modification simply amounts to dropping in clauses R−) and F−) the qualification "ground".

Call the resulting notion SLDNFE-resolution (for SLDNF *extended*). Then for the SLDNFE-resolution the Soundness Theorem 4.16 still holds.

Shepherdson [156] further generalized this form of resolution by allowing a preliminary substitution θ to be applied to nonground negative literals when trying to build a finitely failed subsidiary tree. In terms of Definition 3.14 this modification amounts to changing the clause R−) to

R'−) if A is an atom such that for some θ, $A\theta \in \mathbf{F}$ and $(C, C')\mathbf{R}\sigma$, then $(C, \neg A, C')\mathbf{R}\sigma\theta$,

and dropping the qualification "ground" in clause F−). He called this form of resolution SLDNFS (for SLDNF with substitution) and established its soundness in the sense of the Soundness Theorem 4.16. Also, he proved its completeness w.r.t. a rather involved semantics.

Stärk [162] observed that the same soundness and completeness results hold for simple generalizations of SLDNF- and SLDNFS-resolution, called, respectively, ESLDNF- and ESLDNFS-resolution. In these resolution methods the qualification "ground" is dropped in clause R−) and clause F−) is replaced by

F'−) if A is an atom such that for a renaming θ for A, $A\mathbf{R}\theta$, then $(C, \neg A, C') \in \mathbf{F}$.

He also studied transformation of proofs in the sequent calculus into proofs using these resolution methods.

Moreover Stärk [162] proved completeness of ESLDNF-resolution for a syntactically defined class of *decomposable* programs which includes the positive programs and allowed programs. As allowed programs and allowed queries by Theorem 3.13 do not flounder, the ESLDNF-resolution (with the selection of negative literals delayed until no more positive literals are available) coincides with the SLDNF-resolution. Consequently, this result generalizes the Completeness I Theorem 4.17.

Recently, Stärk [165] proved a much stronger and more natural generalization of the Completeness I Theorem 4.17. The point of departure for Stärk is the observation that completeness depends on certain closure properties.

Definition 5.1 Let $\mathcal{C}^+$ and $\mathcal{C}^-$ be two sets of queries. A program is called a $(\mathcal{C}^+,\mathcal{C}^-)$ - *program* if the following conditions are satisfied, where $inst(P)$ denotes the set of all instances of clauses from P:

(A1) If $Q \in \mathcal{C}^+$ then $Q\theta \in \mathcal{C}^+$.

(A2) If $\mathbf{K}, A, \mathbf{M} \in \mathcal{C}^+$ and $A \leftarrow \mathbf{L} \in inst(P)$ then $\mathbf{K}, \mathbf{L}, \mathbf{M} \in \mathcal{C}^+$.

(A3) If $(\neg A_1, \ldots, \neg A_K) \in \mathcal{C}^+$ then for $i \in [1, k]$ A_i is ground and $A_i \in \mathcal{C}^-$.

(B1) If $Q \in \mathcal{C}^-$ then $Q\theta \in \mathcal{C}^-$.

(B2) If $\mathbf{K}, A, \mathbf{M} \in \mathcal{C}^-$ and $A \leftarrow \mathbf{L} \in inst(P)$ then $\mathbf{K}, \mathbf{L}, \mathbf{M} \in \mathcal{C}^-$.

(B3) If $\mathbf{K}, \neg A, \mathbf{M} \in \mathcal{C}^-$ then $A \in \mathcal{C}^+$. □

The following result of Stärk [165] explains the importance of this notion. Here yet another modification of the SLDNF-resolution is used according to which in Definition 3.14 clause F–) is replaced by F'–).

Theorem 5.2 (Completeness II) *Suppose that P is a $(\mathcal{C}^+,\mathcal{C}^-)$ - program. Then for a query Q*

- *if $comp(P) \models_3 \forall Q\theta$ and $Q \in \mathcal{C}^+$, then for some substitution σ, $Q\mathbf{R}\sigma$ and $Q\sigma$ is more general than $Q\theta$,*
- *if $comp(P) \models_3 \forall \neg Q$ and $Q \in \mathcal{C}^-$, then $Q \in \mathbf{F}$.* □

This result generalizes the Completeness I Theorem 4.17 because for $\mathcal{C}^+ = \{Q \mid Q \text{ is allowed}\}$ and $\mathcal{C}^-$ the set of all queries we get that an allowed program is a $(\mathcal{C}^+,\mathcal{C}^-)$ - program and by Theorems 3.13 and 4.17 both computed and 3-valued correct answer substitutions for allowed programs and queries are ground. Stärk found a systematic way of reducing previous completeness results to the Completeness II Theorem 5.2 by means of modes, that is input/output specifications.

Another modification was proposed by Di Pierro, Martelli and Palamidessi [119]. Their approach is based on a rule termed "negation as instantiation" according to which in the case of SLD-resolution a query consisting of one (possibly nonground) atom fails if all the branches in the SLD-tree either fail or instantiate the atom. This rule is then incorporated into a resolution method for general programs. The resulting method, called SLDNI-resolution, was proved sound w.r.t. 2-valued semantics of program completion.

Finally, let us mention here Shepherdson [160], where an extension of the SLDNF-resolution with unification w.r.t. an equality is studied.

5.2 Prolog and its Variants

Let us consider now Prolog. From the pure theoretical point of view it is an implementation of SLDNF-resolution with the leftmost selection rule with the exception that the selection of nonground negative literals is allowed, that is floundering is ignored. This leads to various difficulties.

As already noted in Section 3.1, we obtain undesired conclusions for the program NUMBERS = $\{positive(x) \leftarrow \neg zero(x);\ zero(0) \leftarrow \}$, as both $\forall x\, \neg positive(x)$ and $positive(t)$, for any ground term t different from 0, can be established. However, for its completion

$$comp(\texttt{NUMBERS}) = \{\forall x\ (positive(x) \leftrightarrow \neg zero(x)),\ \forall x\ (zero(x) \leftrightarrow x = 0\} \cup EQ$$

we do get the intended conclusions, since $comp(\texttt{NUMBERS}) \models \forall x\ (positive(x) \leftrightarrow x \neq 0)$.

In turn, consider the following program SINK where G is a finite graph:

$$\begin{array}{lll} p(a,b) & \leftarrow & \text{for } (a,b) \in G, \\ sink(x) & \leftarrow \neg p(x,y) & . \end{array}$$

Then for a constant a, the query $sink(a)$ succeeds iff for no b, $(a,b) \in G$, that is iff $\neg\exists y\, p(a,y)$. On the other hand, the completion interpretation of the $sink$ relation is: $\forall x(sink(x) \leftrightarrow \exists y \neg p(x,y))$. Thus for some programs the right interpretation is provided by its completion and for others by its computation mechanism. In general, it is not clear whether to interpret the negative literal $\neg A$ in a clause as $\exists \mathbf{y}\ \neg A$ or $\neg\exists \mathbf{y}\ A$, where $\mathbf{y}$ stands for the sequence of local variables of $\neg A$.

A natural solution is to find conditions which prevent selection of nonground negative literals in Prolog computations. This problem was studied by Apt and Pellegrini [7] and, independently, Stroetman [166]. Using the notion of modes they introduced a syntactically defined class of programs and queries for which they proved absence of floundering w.r.t. the SLDNF-resolution with the leftmost selection rule.

However, it is useful to note that in some restricted situations the choice of nonground negative literals does not lead to any complications. Namely, the following result is a direct consequence of the soundness of SLDNFE-resolution, where by SLDNF$^+$ resolution we mean SLDNF-resolution with floundering ignored.

Theorem 5.3 *Given a positive program P and a general query Q we have*

- *if $P \vdash_{\text{SLDNF}^+} \forall\, Q\tau$, then $comp(P) \models_3 \forall\, Q\tau$.* □

The Soundness Theorem 4.16 states that SLDNF-resolution is sound for all safe selection rules, i.e. selection rules which never select a nonground negative literal. In MU-Prolog of Naish [113], a safe selection rule is used by delaying the nonground negative literals until they become ground. In other words, MU-Prolog implements SLDNF-resolution with the "leftmost admissible literal" selection rule, where a literal is admissible if it is negative and ground, or positive. Even more complicated selection rules are allowed in NU-Prolog, the successor of MU-Prolog, of Naish [114] and in Gödel, the language proposed by Hill and Lloyd [76]. In these languages so-called *delay* control declarations cause certain literals to be delayed until they become sufficiently instantiated. Lüttringhaus-Kappel [99] provides a thorough theoretical account of such delay declarations.

The restriction of the SLDNF-resolution to the leftmost selection rule results in loss of completeness, even for very simple programs. Indeed, take $P = \{p \leftarrow p\}$ and $Q = p, q$. Then $comp(P) \models \neg Q$, but the only SLDNF-derivation of Q w.r.t. the leftmost selection rule diverges.

Still, some limited forms of completeness can be obtained here by restricting one's attention to terminating programs – see Apt and Pedreschi [6] and Stroetman [166]. Lately, another aproach to this issue was proposed in Stärk [163] – see Section 8.3.

In Prolog negation can be applied to an arbitrary query, and not only to an atom, as in the SLDNF-resolution and its variants. Also disjunction can be used in queries and bodies of the clauses. Lloyd and Topor [94] (see also Lloyd [93]) modelled these syntactic extensions by means of a more general syntax in which the queries and bodies of clauses can be arbitrary first-order formulas. These generalized queries and programs can be interpreted by means of a syntactic transformation which transforms them to a general query and a general program combined with the SLDNF-resolution. Lloyd and Topor [94] showed that this transformation preserves program completion (which is defined for the generalized programs in the expected way).

This syntactic extension of general programs allows us to deal properly with the program `SINK` discussed above – to enforce its right interpretation w.r.t. program completion it suffices to replace its second clause by

$$\texttt{sink(x)} \leftarrow \neg\exists\texttt{p(x,y)}.$$

This extended syntax is used in the language Gödel of Hill and Lloyd [76] mentioned above.

5.3 Constructive Negation

In SLDNF-resolution only positive literals can generate a computed answer substitution. In SLDNFS-resolution negative literals can generate answers, as well. Unfortunately, these answer substitutions need to be guessed and subsequently verified. Chan [34] suggested a modification of SLDNF-resolution in which nonground negative literals can be selected and can generate answers but, in contrast to the SLDNFS-resolution, these answers can be effectively computed. This way of using negative literals is called *constructive negation* and the resulting form of resolution SLD-CNF-resolution.

First, let us introduce the following helpful notation. For a substitution $\theta = \{x_1/t_1, \ldots, x_n/t_n\}$, let $\hat{\theta}$ denote the formula $\exists\mathbf{y}(x_1 = t_1 \wedge \ldots \wedge x_n = t_n)$, where $\mathbf{y}$ is the sequence of variables from $Ran(\theta) - Dom(\theta)$.

The departure point for Chan's approach is the following property of SLD-resolution (essentially proved by Clark [39]).

> Consider a finite SLD-tree for a query Q. Let $\theta_1, \ldots, \theta_k$ be all c.a.s's for Q present in this SLD-tree. Denote by F_Q the formula $\hat{\theta_1} \vee \ldots \vee \hat{\theta_k}$. Then $comp(P) \models \forall(Q \leftrightarrow F_Q)$.

Consequently $comp(P) \models \forall(\neg Q \leftrightarrow \neg F_Q)$, which suggests to interpret $\neg F_Q$ as the computed answers generated by $\neg Q$.

There are two problems which have to be solved for this interpretation. First, the formula $\neg F_Q$ cannot be interpreted as a set of substitutions any more. Thus it has to be defined what it means to apply this formula to a query. Secondly, F_Q is not always defined. To solve the first problem Chan [34] extended the language of logic programs by allowing equalities $s = t$ and inequalities $\forall(s \neq t)$ in the queries and bodies of the clauses, and provided a normalization algorithm which transforms every formula of the form $\neg F_Q$ to a disjunction of simple equality formulas, that is existentially quantified conjunctions of equalities or their negations.

The second problem is that Chan's definition is based on the original definition of SLDNF-resolution due to Clark [39], according to which, as noted as noted in Section 3.2, for some problematic cases no SLDNF-trees exists. It was adequately solved in Marchiori [103] who provided a formal definition of SLD-CNF resolution in the style of Apt and Doets [11].

Chan [34] noticed that SLD-CNF-resolution is sound w.r.t. program completion (for the 2-valued semantics). In particular, SLD-CNF-resolution allows us to treat correctly the previously mentioned program `NUMBERS` – the query *positive(x)* succeeds with the desired answer $x \neq 0$.

Marchiori [103] studied termination of programs w.r.t. constructive negation and among others proved completeness of the SLD-CNF-resolution for acyclic programs w.r.t. program completion for bounded queries. Further generalizations of constructive negation were proposed by Stuckey [167], and more recently by Drabent [51]. Both of them proved completeness results which subsume the Completeness I Theorem 4.17.

6 2-valued Alternatives for the Least Herbrand Model

In the case of positive logic programs the least Herbrand model of the program exists. This model enjoys a number of natural properties. For example, it is the least pre-fixpoint of the operator T_P and also its least fixpoint. Consequently it is customary to view it as the *standard model* of the program. In the case of general programs the situation dramatically changes because there is no least Herbrand model. Just take $P = \{p \leftarrow \neg q\}$. Then $\{p\}$, $\{q\}$ are the only minimal Hebrand models but none is the least. Thus by Lemma 4.3 T_P may have no least pre-fixpoint and at the end of Subsection 4.2 we already noted that T_P may have no fixpoint at all.

So what is then the standard model of a general program? There is no generally agreed upon answer to this question. With this section we begin a review of some of the plausible answers suggested in the literature.

6.1 Stratified Programs and the Standard Model

Let us first agree on the desired properties of the natural model. Clearly, for every fact in the model we would like to have some explanation why it is there. The following definition suggested by Apt, Blair and Walker [9] and Bidoit and Froidevaux [21] attempts to formalize this requirement.

Definition 6.1 (Supported interpretation) A Herbrand interpretation I is called *supported* if

$$A \in I \ \Rightarrow \exists \ \mathbf{L} \ (A \leftarrow \mathbf{L} \in ground(P), \ I \models \mathbf{L}). \qquad \square$$

Intuitively, $\mathbf{L}$ is an explanation for A. We clearly have

Lemma 6.2 *I is a supported model of P iff $T_P(I) = I$.* □

Thus in view of the observation on the behaviour of the T_P operator we see that for some programs no supported models exist. One possible approach is to accept that some programs have no natural, supported model and to identify classes of programs for which a "natural" supported model exists.

The following notion was first considered in the context of database queries by Chandra and Harel [35] and was introduced in the area of logic programming by Apt, Blair and Walker [9] and Van Gelder [170].

Definition 6.3 (Stratified program) A program is called *stratified* if no cycle with a negative edge exists in its dependency graph. □

In other words, a program is stratified if no negative recursion, that is recursion "through" negation is used in it. For example, the program $P = \{p \leftarrow \neg q,\ q \leftarrow r\}$ is stratified, whereas $P = \{p \leftarrow \neg p\}$ is not. Note that every stratified program is call-consistent, but not conversely. The following equivalent formulation shows that in a stratified program the use of negation is restricted to already known (i.e. defined) relations.

Definition 6.4 (Stratification) Consider a program P. $P = P_1 \cup \ldots \cup P_n$ is called a *stratification of* P if for $i \in [1, n]$ P_i uses
- positively only relations defined in $\bigcup_{j=1}^{i} P_j$,
- negatively only relations defined in $\bigcup_{j=1}^{i-1} P_j$.

P_1 can be empty. For convenience, when some relations used in P are not defined, we assume that they are defined (by the empty set of clauses) in P_1. □

Lemma 6.5 *A program is stratified iff it admits a stratification.* □

Note that a program can admit several stratifications. Following the intuition on the use of negation the following model was defined for stratified programs.

Definition 6.6 (Standard model) Consider a stratified program P. Assume a stratification $P = P_1 \cup \ldots \cup P_n$. Denote by $I \mid R$ - the restriction of the interpretation I to relations in R. Each P_i defines a set of relations rel_i. Define a sequence of Herbrand interpretations as follows:

M_1 = the least model of P_1,
M_2 = the least model of P_2 such that $M_2 \mid rel_1 = M_1$,
...
M_n = the least model of P_n such that $M_n \mid rel_{1,\ldots,n-1} = M_{n-1}$.

We call $M_P = M_n$ the *standard model* of P. □

For example, consider $P = \{p \leftarrow \neg q; q \leftarrow r\}$ and its stratification $P = \{q \leftarrow r\} \cup \{p \leftarrow \neg q\}$. Then $M_1 = \emptyset$ and $M_2 = M_P = \{p\}$.

The following result of Apt, Blair, Walker [9] explains why the model M_P is of interest.

Theorem 6.7 (Standard model) *Consider a stratified program P. Then*

- *M_P does not depend on the stratification of P,*
- *M_P is a minimal model of P,*
- *M_P is a supported model of P.* □

Thus, by the Fixpoint Lemma 4.4 completion of a stratified program has a Herbrand model.

6.2 Locally Stratified Programs and Perfect Models

Still, the above theorem does not uniquely characterize the standard model M_P since for some stratified programs more than one supported model exists. Just take $P = \{p \leftarrow \neg q;\ q \leftarrow q\}$. Then both $\{p\}$ and $\{q\}$ are supported.

To provide a unique characterization of the model M_P, Przymusinski [128] introduced the notion of preferable models. Fix a program P and a well-founded ordering $<$ on B_P. If $A < B$, then we say that A has a *higher priority than* B.

Definition 6.8 (Perfect model) Let M, N be Herbrand interpretations of P. We call N *preferable to* M, and write $N \prec M$, if for every $B \in N - M$ there exists $A \in M - N$ such that $A < B$. We write $N \preceq M$ if $N = M$ or $N \prec M$. We call a Herbrand model of P *perfect* if there are no Herbrand models of P preferable to it. □

Thus a perfect model of P is a $\prec$-minimal Herbrand model of P. The intuition behind these definitions is the following. N is preferable to M if it is obtained from M by possible adding/removing some atoms and an addition of an atom (B) to N is always compensated by the simultaneous removal from M of an atom (A) of higher priority. This reflects the fact that we are determined to minimize higher priority atoms even at the cost of adding atoms of lower priority. A model is then perfect if this form of minimization of higher priority atoms is achieved in it.

The following lemma clarifies the status of perfect models.

Lemma 6.9 *Let P be a program and $<$ a well-founded ordering on B_P.*

- *Every perfect model of P is minimal.*
- *The relation "N is preferable to M" is a partial order.* □

The standard model M_P of a stratified program P is related to perfect models by the following theorem of Przymusinski [128].

Theorem 6.10 *Let P be a stratified program and let for $A, B \in B_P$: $A < B$ iff the relation symbol of B depends negatively on the relation symbol of A. Then M_P is a unique perfect model of P.* □

In other words, M_P is the $\prec$-smallest Herbrand model of P. This theorem provides an alternative proof of the first claim of the Standard Model Theorem 6.7. Thus the notion of a perfect model turns out to be the key concept in assessing the character of M_P.

The previous result immediately suggests a generalization of the concept of stratification which was, again, proposed by Przymusinski [128]. He observed that some programs that are not stratified still have an intuitively clear meaning. The standard example is the program `EVEN`:

$$\texttt{even(0)} \leftarrow$$
$$\texttt{even(s(X))} \leftarrow \neg\texttt{even(X)}$$

The program `EVEN` is clearly not stratified, as the relation `even` depends negatively on itself. Yet, if we consider all ground instances of the clauses of `EVEN`, then we see that no ground atom depends negatively on itself. In other words, if we consider the ground atoms as proposition symbols, then the instantiated program is stratified (albeit infinite). A program that has this property is *locally stratified*.

Definition 6.11 (Local stratification)

- A *local stratification* for a program P is a function *stratum* from B_P to the countable ordinals.
- Given a local stratification *stratum*, we extend it to ground negative literals by putting $\mathit{stratum}(\neg A) = \mathit{stratum}(A) + 1$.

- A clause of P is called *locally stratified with respect to a local stratification stratum*, if for every ground instance $A \leftarrow \mathbf{K}, L, \mathbf{L}$ of it

$$\mathit{stratum}(A) \geq \mathit{stratum}(L).$$

- A program P is called *locally stratified with respect to a local stratification*, if all its clauses are. P is called *locally stratified* if it is locally stratified with respect to some local stratification. □

Lemma 6.12

- *An acyclic program is locally stratified.*
- *A stratified program is locally stratified.* □

Instead of comparing ground atoms by their relation symbols, a local stratification of a program P immediately induces a well-founded ordering on B_P. The following theorem, due to Przymusinski [128], shows that perfect models unambiguously define a semantics for locally stratified programs.

Theorem 6.13 (Unique perfect model) *Let P be a locally stratified program and let for $A, B \in B_P$: $A < B$ iff $\mathit{stratum}(A) < \mathit{stratum}(B)$. Then P has a unique perfect model.* □

It was soon realised, that some programs are not locally stratified but still have a clear meaning. For example, we could rewrite the program `EVEN` to `EVEN'`:

$$\begin{aligned} \texttt{even(X)} &\leftarrow \texttt{zero(X)} \\ \texttt{even(Y)} &\leftarrow \texttt{successor(X,Y)},\neg\texttt{even(X)} \\ \texttt{zero(0)} &\leftarrow \\ \texttt{successor(X,s(X))} &\leftarrow \end{aligned}$$

(In this program, we can change the representation of numbers without changing the clauses defining the relation `even`.) This program is no longer locally stratified, as

$$\texttt{even(0)} \leftarrow \texttt{successor(0,0)},\neg\texttt{even(0)}$$

is an instance of the second clause.

Of course, the premise `successor(0,0)` of this instance is false, but that is part of the semantics of the program, while (local) stratification is a syntactic property. There are two proposals for adapting local stratification and perfect model semantics to capture this phenomenon: *weak stratification* by Przymusinska and Przymusinski [123, 124] and *effective stratification* by Bidoit and Froidevaux [22].

For weak stratification, it is observed that for each iteration in the construction of the model, only the next *lowest* stratum must be identified. The truth values obtained for the atoms in this stratum can then be used to discard clauses with false premises. This in turn may remove some dependencies, thereby allowing identification of the next lowest stratum. We omit the formal definition. For the program `EVEN'`, the lowest stratum consists of the `zero`- and `successor`-atoms. Discarding clauses with false `zero`- and `successor`-premises yields the (already locally stratified) program

$$\begin{aligned} \texttt{even(0)} &\leftarrow \texttt{zero(0)} \\ \texttt{even(s(X))} &\leftarrow \texttt{successor(X,s(X))},\neg\texttt{even(X)} \\ \texttt{zero(0)} &\leftarrow \\ \texttt{successor(X,s(X))} &\leftarrow \end{aligned}$$

Theorem 6.14 (Przymusinska and Przymusinski [123, 124]) *A locally stratified program is weakly stratified.* □

Bidoit and Froidevaux [22] define the notion of *effective stratification*, which takes this approach even further. As it is closely related to the (still to be introduced) well-founded models, we discuss it in Section 7.

6.3 Well-supported or Stable Models

In Section 6.2, we noted that, for a program $P = \{p \leftarrow \neg q;\ q \leftarrow q\}$, both $\{p\}$ and $\{q\}$ are supported models. However, the support for q is unfounded, in the sense that q is the explanation why q is true. So we would like to rule out the second supported model. The following approach of Fages [59] makes this idea precise.

Definition 6.15 (Well-supported interpretation) For a query $\mathbf{L}$, denote by $pos(\mathbf{L})$ the sequence of positive literals of $\mathbf{L}$. A Herbrand interpretation I is called *well-supported* if for some well-founded ordering $<$ on B_P

$$A \in I \text{ implies } \exists \mathbf{L}(A \leftarrow \mathbf{L} \in ground(P),\ I \models \mathbf{L},\ \text{and } B < A \text{ for } B \in pos(\mathbf{L})).$$ □

Intuitively, I is well-supported if every $A \in I$ has an explanation which does not use A. For example, for $P = \{p \leftarrow \neg q;\ q \leftarrow q\}$, the model $\{p\}$ is well-supported, whereas $\{q\}$ is not. It should be noted that some programs have no well-supported models. Take for example $P = \{p \leftarrow q;\ p \leftarrow \neg q;\ q \leftarrow p;\ q \leftarrow \neg p\}$. Its only Herbrand model, $\{p, q\}$, is not well-supported.

By using the intuition of "rational beliefs" from autoepistemic logic, Gelfond and Lifschitz [68] introduced an important notion of a stable model. We begin with the following auxiliary notions.

Definition 6.16 (Gelfond-Lifschitz transformation) For a query $\mathbf{L}$, denote by $neg(\mathbf{L})$ the sequence of negative literals of $\mathbf{L}$. Let P be a program and I an interpretation. Let

$$H(P, I) = \{H \leftarrow pos(\mathbf{L}) \mid\ H \leftarrow \mathbf{L} \in ground(P),\ I \models neg(\mathbf{L}).\}$$

Now define

$$\Gamma_P(I) = M_{H(P,I)}.$$ □

Thus $H(P, I)$ is the positive program obtained from P by removing all clauses that contain one or more negative literals that are **false** in I, and by deleting all negative literals that are **true** in I. In turn, $\Gamma_P(I)$ is a Herbrand model equal to the least Herbrand model of the positive program $H(P, I)$.

Definition 6.17 (Stable model) A Herbrand interpretation I of a program P is called *stable* if $\Gamma_P(I) = I$. □

Gelfond and Lifschitz [68] explain the intuition behind the definition of a stable model as follows. Consider a "rational agent" with a set of beliefs I and a set of premises P. Then any clause that has a literal $\neg A$ with $A \in I$ in its body is useless, so it can be removed. Moreover, any literal $\neg A$ with $A \notin I$ is trivial, so it can be deleted. This yields the simplified program $H(P, I)$. If now I happens to be precisely the set of atoms that follow logically from this simplified set of premises, then the set of beliefs I is stable. Thus stable models are "possible sets of belief a rational agent might hold".

The following theorem of Fages [59] shows that the concepts of well-supported and stable models coincide. It was independently established by Elkan [57] for the case of propositional programs.

Theorem 6.18 *Suppose that I is a model of P. Then I is stable iff it is well-supported.* □

Thus, a fortiori stable models of a program P are supported models and consequently, by the Fixpoint Lemma 4.4 they are also models of $comp(P)$. The converse is in general not true (see the beginning of this section), but for certain programs the Herbrand models of $comp(P)$ and stable models coincide. Namely, we have the following corollary to the above theorem, due to Fages [60] and, independently, Ben-Eliyahu [19].

Corollary 6.19 *Suppose that no cycle with only positive edges exists in the dependency graph of P. Then the Herbrand models of $comp(P)$ coincide with the stable models of P.* □

The following results of Gelfond and Lifschitz [68] clarify the relation between stable models and the notions introduced in Sections 6.1 and 6.2.

Theorem 6.20 (Unique stable model) *Consider a program P. Then*

- *any stable model of P is a minimal model of P,*
- *if P is locally stratified, then it has a unique stable model which coincides with its perfect model considered in the Unique perfect model Theorem 6.13.* □

In particular, if P is stratified, then by Theorem 6.10 it has a unique stable model which coincides with its standard model M_P. Thus, similarly to the notion of a perfect model, the concept of a stable model allows us to characterize the notion of a standard model for stratified programs in a unique way.

The second result also shows that a sufficient condition for the existence of a stable model of a program is that it is locally stratified. Dung [53] proves that call-consistency is also sufficient. More results can be found in Dung [53] and Fages [60].

7 Three-valued Alternatives for the Least Herbrand Model

Stable model semantics allows more than one stable model, or none at all. This reflects some uncertainty about the conclusions that should be drawn from a program. In some cases, a 'local' uncertainty can destroy too much information. For example, if P is a stratified program in which the relation symbol p does not occur, then $P \cup \{p \leftarrow \neg p\}$ has no stable models. Thus the information contained in P is not reflected in the stable model semantics, although it is not related to the uncertainty about the truth value of p.

Well-founded semantics (WFS) avoids this problem, by producing one *3-valued* model, instead of multiple 2-valued ones. In contrast to Section 4.3, 3-valued *logic* (that is, a 3-valued interpretation of the connectives) is not needed to obtain these 3-valued models. There are numerous characterizations of the well-founded semantics; we present here a few of them. Apart from the information ordering $\subseteq$ on 3-valued interpretations, as defined in Section 4.3, we sometimes use the *truth* ordering: I is truth-less than J iff $I^+ \subseteq J^+$ and $I^- \supseteq J^-$.

7.1 Iterated Least Fixpoint Characterization of WFS

Suppose that one prefers the least Herbrand model/closed world assumption (rather than the completion or classical negation) to decide whether a negative literal holds w.r.t. a positive program. Then, given a general program, one can observe that *regardless of the semantics of negative literals in clause bodies*, some atoms must be true in its semantics (e.g., facts in the program), and some must be false (e.g., atoms that do not unify with the head of any clause). One of the weaknesses of the proposals in Section 6 is, that such information is lost if no model is produced. When 'guessing' an interpretation to see if it is a stable model, we know what guess to make for those atoms. We can also use those atoms to simplify the program, as in the example on weak stratification. As a result of this simplification, more atoms may become certainly true or certainly false.

If the truth value of all atoms can be decided in this way, then the program is called *effectively stratifiable* by Bidoit and Froidevaux [22]. If some atoms remain undecided, then we might start guessing, in order to find stable models. But another interesting option is to stop just there, and to return a 3-valued model. This model shows which atoms are true, respectively false, regardless of the semantics of negation, and which atoms cannot be decided in this way. It is called the *well-founded model.*

The original definition of the well-founded semantics is usually attributed to Van Gelder, Ross and Schlipf [66]. Here, we loosely follow the somewhat more constructive definition of Bidoit and Froidevaux [22]. Two significantly different characterizations of the well-founded model are presented in the next sections.

The first step of our definition is to derive from a program P which atoms are certainly true, respectively false, in its semantics. An atom is certainly true, if it can be derived without using clauses that contain negative literals. An atom is possibly true, if it cannot be derived, even when ignoring all negative premises. An atom is certainly false if it is not possibly true. We collect the 'certain' atoms in a 3-valued interpretation $I_3(P)$, leaving the 'uncertain' atoms **unknown**.

Definition 7.1 Let P be a program. By P^+ we denote the program obtained from P by deleting all clauses that contain a negative literal. By P^- we denote the program obtained from P by deleting all negative literals. Let $I_3(P) = (M_{P^+}, \overline{M_{P^-}})$. □

Here $\overline{M}$ denotes the complement of M w.r.t. the set of ground atoms in the considered universal language, which is larger than the language L_P defined by P.

Bidoit and Froidevaux [22] call M_{P^+} the set of *defined* atoms (Def(P)) and M_{P^-} the set of *potentially defined* atoms (PotDef(P)). Van Gelder, Ross and Schlipf [66] call the atoms in M_{P^+} *well-founded* and $\overline{M_{P^-}}$ an *unfounded set* (see below).

In this section, we use a simplification of a program w.r.t. a set of certain literals that differs from the Gelfond-Lifschitz transformation (Definition 6.16) in a significant way: not only negative literals, but also positive literals are considered for simplification. A generalization of Definition 6.16 to 3-valued interpretations is considered in the next section.

Definition 7.2 Let P be a program and I a 3-valued interpretation. By $P \backslash I$, we denote the program that is obtained from *ground*(P) by deleting all clauses that contain one or more literals that are **false** in I, and by deleting all literals that are **true** in I. Furthermore,

$$\Phi_P(I) = I_3(P \backslash I).$$

□

Lemma 7.3 (Przymusinski [130]) *The Φ_P-operator is monotonic w.r.t. the information ordering.* □

This lemma implies that the least fixpoint of Φ_P exists, and that it can be reached by iterating the Φ_P-operator from $(\emptyset, \emptyset)$, taking the pairwise union at limit ordinals.

Definition 7.4 (Well-Founded model) The information-least fixpoint of Φ_P is called the *well-founded model* of P, $WFM(P)$. □

As Φ_P is in general not continuous, more than ω iterations are usually needed to reach the least fixpoint. However, if the number of atoms in the language is finite, say n, then the computation of $WFM(P)$ in this way takes $O(n^2)$ iterations, as shown by Van Gelder, Ross and Schlipf [66].

The original definition of the well-founded model by Van Gelder, Ross and Schlipf [66] slightly differs from this one. Instead of Φ_P, they define and iterate the operator

$$V_P(I) = (\text{the set of facts in } (P \backslash I)^+, \overline{M_{(P\backslash I)^-}}).$$

Thus, the derivation of positive facts using V_P goes much 'slower' than using Φ_P. Moreover, they define $\overline{M_{(P\backslash I)^-}}$ in another, non-constructive way, namely, as the largest *unfounded set*, where an unfounded set is a set of ground atoms U such that for all atoms $A \in U$, all instances of clauses that conclude A have a premise in U (thus, if we assume the atoms in U to be false, no clause that could derive them remains applicable, which justifies the assumption).

The well-founded model is related to stable models, and hence to the other models in Section 6, in the following way.

Theorem 7.5 (Extension) (Van Gelder, Ross and Schlipf [66]) *All stable models of a program extend*[3] *its well-founded model.* □

Corollary 7.6 *If the well-founded model of a program is total, then it is its unique stable model.* □

The converse of this implication is not true: the program $\{\mathtt{p} \leftarrow \neg\mathtt{p};\ \mathtt{p} \leftarrow \neg\mathtt{q};\ \mathtt{q} \leftarrow \neg\mathtt{p}\}$ has $\{\mathtt{p}\}$ as its unique stable model, but its well-founded model is $(\emptyset, \emptyset)$. But Theorem 6.14 implies that the well-founded model of a locally (weakly) stratified program is total.

Corollary 7.7 *The well-founded model of a locally stratified program coincides with its unique perfect model.* □

We can now define the generalization of the notion of weak stratification, already mentioned in Section 6.2, due to Bidoit and Froidevaux [22].

Definition 7.8 (Effective stratification) A program P is *effectively stratifiable* if $WFM(P)$ is total. □

Theorem 7.9 (Bidoit and Froidevaux [22]) *A weakly stratified program is effectively stratifiable.* □

[3] I.e., the well-founded model is lower in the information ordering than any stable model. This notion of extension should not be confused with another one: a semantics $\mathcal{S}$ defined for a class of programs $\mathcal{P}$ is sometimes said to extend a semantics $\mathcal{S}'$ defined for a smaller class $\mathcal{P}' \subseteq \mathcal{P}$ if $\mathcal{S}$ and $\mathcal{S}'$ *coincide* on $\mathcal{P}'$. To avoid confusion, we shall not use the word 'extend' in this sense.

7.2 Stationary Models and Stationary Expansions

In this section we present an alternative characterization of the well-founded model, due to Przymusinski [136], which relies somewhat more on 3-valued logic, but stays closer to the definition of stable models. Moreover, this characterization also suggests other interesting 3-valued models of the program, which extend the well-founded model.

Definition 6.16 presents a function $H(P, I)$ that simplifies a program P with respect to a 2-valued interpretation I. In fact, this function replaces each negative literal in the program by the truth-value it has in the interpretation. The result is a positive program, except that the logical constants **true** and **false** occur in it. When considering the semantics of such a program, i.e., its least Herbrand model, the constants **true** can be ignored. A constant **false** in a clause body means, that this clause is never applicable, so the whole clause can be ignored. By syntactically removing the parts of the program that can be ignored, we bypassed the introduction of the logical constants, and defined the result of the function to be a positive program.

It is straightforward to generalize this function so that it simplifies a program with respect to a *3*-valued interpretation. The result is a positive program, in which the constants **true**, **false** and **unknown** occur. We can get rid of the constants **true** and **false** again, but the constants **unknown** remain. This is not a problem: the *truth*-least partial Herbrand model of such programs is well-defined.

Definition 7.10 (u-program) A *u-program* is a positive program in which the constants **true**, **false** and **unknown** may occur. $M_3(P)$ denotes the truth-least 3-valued Herbrand model of a u-program P.[4]

Let P be a program and I a 3-valued interpretation. The u-program $H_3(P, I)$ is obtained from *ground*(P) by replacing every negative literal in P by the truth value it has in I.

$$\Gamma_{3P}(I) = M_3(H_3(P, I)).$$

□

Analogously to the stable models of Section 6.3, the fixpoints of Γ_{3P} are considered as possible 'meanings' of the program P. Przymusinski called these models *partial, extended* or *3-valued* stable models, or *stationary* models [133, 136, 135, 127, 138]. From now on, we shall refer to them as stationary models; by 'stable models' we shall always mean 2-valued ones.

Definition 7.11 (Stationary model) Let P be a program. A *stationary* model of P is a 3-valued Herbrand interpretation I such that $\Gamma_{3P}(I) = I$. □

In contrast to stable models, each program has at least one stationary model. Moreover, the set of stationary models of a program has an *information*-least element, which happens to coincide with the *well-founded model.*

Theorem 7.12 (Least stationary model) (Przymusinski [136]) *Let P be a program. The information-least stationary model of P exists and coincides with $WFM(P)$.* □

If I is a 2-valued interpretation, then $\Gamma_{3P}(I)$ obviously coincides with $\Gamma_P(I)$. Thus all stable models of a program are also (information-maximal) stationary models of it. This clarifies the Extension Theorem 7.5.

[4]Note that $M_3(P)$ coincides with $I_3(P)$, if we get rid of occurrences of **true** and **false** as before, and treat occurrences of **unknown** as negative literals.

Instead of considering the information-*minimal* stationary model, we can also consider information-*maximal* ones as plausible 'belief states' associated with the program. Among these are the stable models of a program, if it has any. But, while the stable model semantics of a program is easily destroyed by local 'impossibilities', maximal stationary model semantics is much more robust. A local impossibility simply means that some atoms remain **unknown** in all models; it does not affect the (global) existence of the models.

Neither the definition of a stable model or of a stationary model is constructive—it involves a "guess" of an interpretation which is then checked whether it is a stable, respectively, stationary model. Saccà and Zaniolo [150] characterized all stable models by means of fixpoints of a backtracking operator which generates all stable models of a program. This work was further extended and generalized by Teusink [168], who characterized all stationary models by means of fixpoints of another nondeterministic, non-monotonic operator.

The following characterization of stationary models, proposed by Przymusinski [126], stays within 2-valued logic. First we identify a program with the program obtained by replacing every occurrence of a negative literal $\neg A$ by the new atom not_A. This gives a positive program, in which the atoms of the form not_A occur only in the bodies of clauses. A stationary expansion is obtained by adding to such a program a suitable set of not_A atoms: these fully determine a stationary model.

Definition 7.13 (Stationary expansion) Let P be a positive program with not_A atoms in bodies of clauses. Let C be a set of not_A atoms.

- A Herbrand interpretation for P is a set of atoms (containing both ordinary atoms and not_A atoms, in general).
- By the *minimal models* of $P \cup C$, we mean the Herbrand interpretations that are minimal w.r.t. set inclusion for the *ordinary atoms* (but not necessarily w.r.t. not_A-atoms) among those interpretations I that satisfy:
  - $I \models P \cup C$ (in the classical sense), and
  - if $P \cup C \models A$, then $not\_A \notin I$.
- For a negative literal $\neg A$, $P \cup C \models_{min} \neg A$ if $\neg A$ is true in all minimal models of $P \cup C$.
- A *stationary expansion* of P is a consistent theory $E(P)$ which satisfies

$$E(P) = P \cup \{not\_A \mid E(P) \models_{min} \neg A\}.$$

- The least stationary expansion of P is called its *stationary completion*. □

Theorem 7.14 (Correspondence) (Przymusinski [126]) *Let P be a program. There is the following one-to-one correspondence between stationary models and stationary expansions of P.*

- *If M is a stationary model of P, then $P \cup \{not\_A \mid M \models_3 \neg A\}$ is a stationary expansion of P.*
- *If $E(P)$ is a stationary expansion of P, then $\{A \mid E(P) \models A\} \cup \{\neg A \mid E(P) \models_{min} \neg A\}$ is a stationary model of P.*

In this way, the well-founded model of P corresponds with the stationary completion of P. □

The information-least stationary model (i.e., the well-founded model) of a program can be computed by iterating Γ_{3P} from $(\emptyset, \emptyset)$. This corresponds to the following theorem.

Theorem 7.15 (Przymusinski [126]) *Let P be a program.*

> *Let $P_0 = P$.*
> *For a successor ordinal $\alpha + 1$, let $P_{\alpha+1} = P_\alpha \cup \{not\_A \mid P_\alpha \models_{min} \neg A\}$.*
> *For a limit ordinal β, let $P_\beta = \bigcup_{\alpha<\beta} P_\alpha$.*

The sequence $P_0, P_1, \ldots, P_\alpha, \ldots$ has a fixpoint which coincides with the stationary completion of P. □

We shall discuss a generalization of stationary expansions to the class of general *disjunctive* programs in Section 10.3.

7.3 The Alternating Fixpoint Characterization of WFS

Yet another characterization of the well-founded model is the one given by Van Gelder [65]. It is based solely on 2-valued interpretations, which only in the end are combined into a 3-valued model.

As observed by Van Gelder [65], Γ_P is an *anti*monotonic operator (on 2-valued interpretations, thus w.r.t. the truth ordering). Thus Γ_P^2, i.e., Γ_P iterated twice, is monotonic and, on the lattice we work on, has a least fixpoint, say I_P. Then $\Gamma_P(I_P)$ is the greatest fixpoint of Γ_P^2.

Theorem 7.16 (Alternating fixpoint I) (Van Gelder [65]) *Let P be a program. Then the least fixpoint I_P of Γ_P^2 exists and*

$$WFM(P) = (I_P, \overline{\Gamma_P(I_P)}).$$

□

The Extension Theorem 7.5 is also a corollary of this theorem. The following theorem is a more general version of the Alternating fixpoint Theorem 7.16.

Theorem 7.17 (Alternating fixpoint II) (Przymusinska and Przymusinski [121]) *Let P be a program and I a 2-valued interpretation. $(I, \overline{\Gamma_P(I)})$ is a stationary model of P iff $\Gamma_P^2(I) = I \subseteq \Gamma_P(I)$.* □

Note that, for $P = \{\mathrm{p} \leftarrow \neg\mathrm{p};\ \ \mathrm{q} \leftarrow \neg\mathrm{q}\}$, Γ_P oscillates between $\{\mathrm{p}\}$ and $\{\mathrm{q}\}$, but that there is no corresponding stationary model, because the interpretations $(\{\mathrm{p}\}, \overline{\{\mathrm{q}\}})$ and $(\{\mathrm{q}\}, \overline{\{\mathrm{p}\}})$ are inconsistent.

Such pairs of interpretations are generalized to finite sets by Baral and Subrahmanian [16].

Definition 7.18 (Stable class) Let P be a program. A *stable class* of P is a finite set of (2-valued) interpretations A such that $A = \{\Gamma_P(I) \mid I \in A\}$. □

If a program P has a stable model M, then $\{M\}$ is a stable class of P. An interpretation I is a fixpoint of Γ_P^2 iff $\{I, \Gamma_P(I)\}$ is a stable class of P.

This approach of Van Gelder has been generalized in another direction by Fitting [62], namely to the case of programs interpreted over 4-valued models, or more generally, bilattices.

7.4 Properties of the Well-Founded Semantics and its Extensions

Well-founded semantics has as a drawback, that it does not infer all atoms that one would expect to be true. Consider for example the program $P = \{p \leftarrow \neg q;\ q \leftarrow \neg p;\ r \leftarrow p;\ r \leftarrow q\}$. It has two stable models: p is true in one and q in the other. In both $p \vee q$, and therefore r, is true. Yet r is unknown in the well-founded model.

Numerous semantics have been proposed that extend the well-founded semantics: WFS$'$, WFS$^+$ and EWFS by Dix [46], GWFS by Baral, Lobo and Minker [13], WFS$_C$ by Schlipf [154] (equivalent to WFS$^+$), WFS$_E$ by Hu and Yuan [77], WFS$_S$ by Chen and Kundu [36] and finally the O-semantics by Pereira, Aparício, and Alferes[118]. The properties of these semantics were investigated in Dix [48] and Dix [49].

Theorem 7.19 (Properties) Dix [48, 49]

- *The well-founded semantics, WFS' and WFS^+ are rational.*
- *EWFS and O-semantics are cautious, but not rational.*
- *EWFS, WFS$_E$ and WFS$_S$ do not satisfy the cut-rule.*
- *GWFS is not cautious and moreover does not satisfy the Principle of Partial Evaluation.*

□

8 Program Completion Revisited

In the previous two sections, we have defined semantics for negation by means of canonical models: stable models and well-founded models. The question arises whether these semantics can be characterized by some form of completion as well — the stationary completion (Definition 7.13) is technically a logical theory, but, as all negative conclusions are stated as facts, it is still very close to a model. Wallace [172] answered this question affirmatively. In this section we summarize his results, which are obtained by defining two simple program transformations, and considering the completion of the transformed programs. Then we discuss briefly recent results of Stärk [164, 163].

8.1 Tightened Completion

The standard program completion, as discussed in Section 4 results in a "loose" interpretation of negation, corresponding to the negation as finite failure rule (the Soundness Theorem 4.16 and the Completeness Theorem 4.17). In order to obtain a "tight" interpretation of negation, Wallace encoded the iterations of the T_P-operator into the program.

Definition 8.1 (Tightened program) Let P be a program. The *tightened program* P_T is derived from P as follows, where N is a variable:

- The language of P_T consists of L_P augmented with a new relation symbol p of arity $n+1$ for every relation symbol p of arity n in L_P. A new unary function symbol s is also added.
- In each clause of P, the head $p(\mathbf{t})$ is replaced by $p(\mathbf{t}, s(N))$ and each positive literal $p(\mathbf{t})$ in the body is replaced by $p(\mathbf{t}, N)$.
- For each relation symbol p in L_P, the clause $p(\mathbf{x}) \leftarrow p(\mathbf{x}, N)$ is added. □

The *tightened completion* of a program P is defined as the completion of P_T. The following result clarifies the relation between the stable models of a program and its tightened completion.

Theorem 8.2 (Tightened completion) (Wallace [172]) *The stable models of a program P are precisely the restrictions of the Herbrand models of $comp(P_T)$ to L_P.* $\square$

8.2 Rounded Completion

As a special case of the previous theorem, one can observe that the tightened completion of a program is inconsistent if and only if the program has no stable models.

One of the motivations for considering 3-valued models of the completion in Section 4.3 and well-founded semantics in Section 7 was avoiding inconsistency. The following program transformation, suggested independently by Drabent and Martelli [52] and Wallace [172], results always in a call-consistent program, thus by Theorem 4.6 its completion is consistent.

Definition 8.3 (Doubled program) Let P be a program. The *doubled program* (called *split program* in [52]) P_D is derived from P as follows.

- The language of P_D consists of L_P augmented with a new relation symbol p' of arity n for every relation symbol p of arity n in L_P.
- Each clause of P is replaced by two new clauses:
  - in the first clause, each occurrence of a relation symbol p in a negative literal is replaced by p'.
  - in the second clause, each occurrence of a relation symbol p in a positive literal or the head of a clause is replaced by p'. $\square$

The *doubled completion* (called *strict completion* in [52]) of a program P is defined as the completion of P_D. There is a close connection between the doubled completion of a program and the 3-valued interpretation of its standard completion.

Theorem 8.4 (Doubled completion) (Drabent and Martelli [52]) *Let P be a program and L and atom or a ground negative literal. Then*

$$comp(P_D) \models L \textit{ iff } comp(P) \models_3 L.$$

$\square$

The tightening and doubling program transformations are orthogonal: $(P_T)_D = (P_D)_T$ is called the *rounded program* derived from P; its completion is called the *rounded completion*. The following result clarifies the relation between the well-founded model of a program and its rounded completion.

Theorem 8.5 (Rounded completion) (Wallace [172]) *The well-founded model of a program P consists exactly of those ground literals from L_P that are true in all Herbrand models of the rounded completion of P.* $\square$

Intuitively, one can explain this relation between the rounded completion of a program and its well-founded model through the alternating fixpoint characterization of the latter. We can split a Herbrand model of the rounded completion into two sets, one containing the dashed atoms, the other the undashed atoms. By removing the dashes in the first one, two interpretations are obtained. It can be easily seen that Γ_P oscillates between them.

Finally, Wallace describes yet another completion, the *full completion* of a program, which is obtained from the rounded completion by dropping the free equality axioms, and adding, for each relation p, the induction axiom

$$\neg p(\mathbf{x}, 0) \wedge \forall N(\neg p(\mathbf{x}, N) \rightarrow \neg p(\mathbf{x}, s(N))) \rightarrow \forall N \neg p(\mathbf{x}, N).$$

The result is that the effect of the counter in the tightened program is weakened: a loop still leads to failure, but an infinite descending chain does not. For example, the full completion of the program $\{p(f(x)) \leftarrow p(x)\}$ entails $\neg p(t)$ for each ground term t, but it does not entail $\forall x\ p(x)$, as there exists a model with infinitely many individuals $a_1, a_2, \ldots$ such that, for all i, $a_i = f(a_{i+1})$. This semantics coincides with the one presented by Van Gelder [170].

8.3 Other Approaches

We conclude this section by mentioning two other modifications of completion, proposed by Stärk. The first one is called *partial completion*. As in the definition of a doubled program (Definition 8.3) for each relation symbol p a new relation symbol p' of the same arity is introduced. The relations p' are used in a modified Step 6 of building the completion. Now instead of replacing " $\leftarrow$ " by " $\leftrightarrow$ " the formulas $\forall \mathbf{x}(\sim p(\mathbf{x}) \leftarrow\ \sim F)$ are added. Here $\sim$ behaves like the classical negation with the exception that $\sim p(\mathbf{t})$ is $p'(\mathbf{t})$ and $\sim p(\mathbf{t})$ is $p'(\mathbf{t})$.

The resulting theory is called *partcomp*(P) for partial completion. The usual completion is obtained by adding the axiom $\forall \mathbf{x}(p'(\mathbf{x}) \leftrightarrow \neg p(\mathbf{x}))$ for each relation p. Stärk [164] showed that the Completeness II Theorem 5.2 holds when $comp(P) \models_3$ is replaced by $partcomp(P) \models$. This result generalizes Theorem 5.2 because Stärk also showed that for all queries Q we have $partcomp(P) \models \forall Q$ iff $comp(P) \models_3 \forall Q$ and $partcomp(P) \models \forall \sim Q$ iff $comp(P) \models_3 \forall \neg Q$.

Then, in Stärk [163] a modification of this approach dealing with Prolog is proposed. To this end the SLDNF-resolution with the leftmost selection rule is related to a theory called $\ell comp(P)$. This theory is a modification of $comp(P)$ obtained by introducing for each relation symbol p three new relation symbols: p^s, p^f and p^t, with the intuitive meaning "p succeeds", "p (finitely) fails" and "p terminates". $\ell comp(P)$ is built in a similar way as $partcomp(P)$, but now the construction involves three operators, **S**, **F** and **T**, which transform the queries of the original language into formulas of the enriched language which includes the relation symbols p^s, p^f,p^t. A typical and crucial law is $\mathbf{F}(L_1 \wedge L_2) = \mathbf{F}L_1 \vee (\mathbf{T}L_1 \wedge \mathbf{F}L_2)$, which intuitively expresses when the query L_1, L_2 finitely fails w.r.t. the SLDNF-resolution with the leftmost selection rule. The corresponding result connects this resolution method with $\ell comp(P)$ in a way analogous to the Completeness II Theorem 5.2. This generalizes in an essential way a completeness result of Stroetman [166], where only terminating programs are considered.

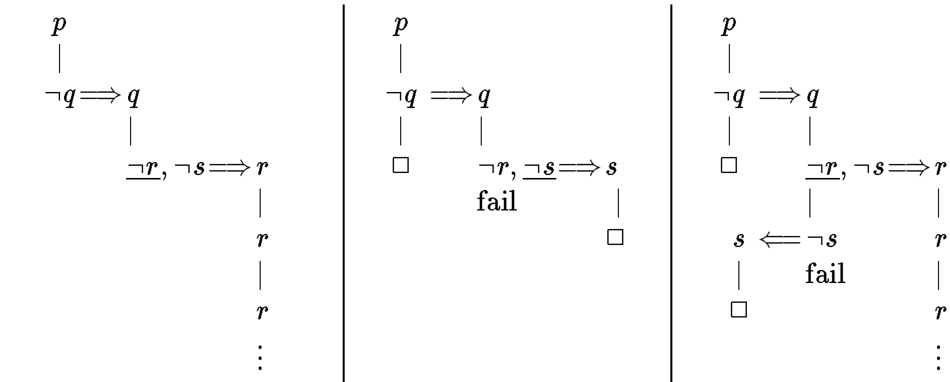
9 Proof Theory III: SLS-resolution

SLS-resolution is a modified version of SLD-resolution that can deal with stratified programs rather than just definite (i.e., positive) programs (hence the second 'S' replacing the 'D'). In fact, similar resolution methods for all general programs are also called SLS-resolution. First we present the definition for stratified programs due to Przymusinski [125] (or more precisely a mild generalization to locally stratified programs adapted from Bol [24]).

The main difference with SLDNF-resolution is, that the computation-oriented negation as finite failure rule is replaced by the more idealistic negation as (not necessarily finite) failure rule.

As a consequence, SLS-resolution is not effective: an implementation can only approximate it. In contrast, SLDNF-resolution can be implemented, but not in a straightforward way: the sets of SLDNF-successful and finitely failed queries are recursively enumerable, but not by building SLDNF-trees via all (possibly infinitely many) selection rules.

Example 9.1 (SLDNF versus SLS-resolution) Let $P = \{p \leftarrow \neg q;\ q \leftarrow \neg r, \neg s;\ r \leftarrow r;\ s \leftarrow\}$. For the query p we have the following trees (where needed, selected literals are underlined; $\Longrightarrow$ denotes the *subs* relation between nodes and trees).



unsuccessful SLDNF-tree	SLS-tree successful SLDNF-tree	SLS-tree

We see (in the middle picture) that using the rightmost selection rule yields a finite and successful SLDNF-tree, which is also an SLS-tree. But using the leftmost selection rule yields an infinite and unsuccessful SLDNF-tree (left picture). Thus the value of the completeness results for SLDNF-resolution, stating the *existence* of a successful SLDNF-tree, is limited. The rightmost picture shows that the SLS-tree via the leftmost selection rule is successful, although infinite. □

9.1 SLS-resolution for Locally Stratified Programs

We now provide a formal definition of SLS-resolution, for locally stratified programs using the concepts introduced in Section 3.2 when defining SLDNF-resolution.

Definition 9.2 (Stratum) Let P be a program that is locally stratified w.r.t. *stratum*.

- For an atom A, not necessarily ground, we define

$$\mathit{stratum}(A) = \mathit{sup}\{\mathit{stratum}(Ag) \mid Ag \text{ is a ground instance of } A\},$$

- For a negative literal $\neg A$, not necessarily ground, we define

$$\mathit{stratum}(\neg A) = \mathit{stratum}(A) + 1,$$

- We define $\mathit{stratum}(\Box) = 0$ and for a query $Q = L_1, \ldots, L_n$ $(n > 0)$,

$$\mathit{stratum}(Q) = \mathit{max}\{\mathit{stratum}(L_i) \mid i \in [1, n]\}.$$

□

Definition 9.3 (SLS-tree) An *SLS-tree* is a forest $\mathcal{F}$, whose nodes are (possibly marked) queries of (possibly marked) literals. (The markers are the same as in SLDNF-trees.) The function *subs* assigns to nodes containing a marked negative ground literal $\neg A$ a tree in $\mathcal{F}$ with root A.

A tree is *successful* if it has a leaf marked as *success*. A tree is *floundered* if it has a leaf marked as *floundered*. Hence a tree may be both successful and floundered. A tree is *failed* if it is neither successful nor floundered.

Let P be a locally stratified program and R a selection rule. For every query Q, we define the *SLS-tree* $\mathcal{F}$ *for* P *and* Q *via* R by induction on $stratum(Q)$. The root of the main tree T of $\mathcal{F}$ is Q. For *any* node N in T we have:

- If N is the empty query, then N is marked as *success* and has no children.
- If R selects a positive literal L in N, then N has as children the nodes that are obtained by extending T at N in the sense of Definition 3.5. If no children can be obtained in this way, then N is marked as *failed*.
- If R selects a negative literal $\neg A$ in N, then
  - If A is nonground, then N has no children and is marked as *floundered*.
  - If A is ground, then $stratum(A) < stratum(Q)$, thus the SLS-tree $(\mathcal{F}', T', subs')$ for P and A via R is already defined. Then set $subs(N)$ to T', extend $subs$ by $subs'$ and extend $\mathcal{F}$ by $\mathcal{F}'$.
    - If T' is successful, then N has no children and is marked as *failed*.
    - Otherwise, if T' is floundered, then N has no children and is marked as *floundered*.
    - Otherwise, T' is failed, and the resolvent $(\epsilon, N - \{\neg A\})$ is the only child of N. (Thus, in contrast to SLDNF-trees, finiteness of $\mathcal{F}'$ is not required here.) □

Definition 9.4 (Computed answer substitution) Let P be a locally stratified program and Q a query. Consider a branch in the main tree T of an SLS-tree for P and Q which ends with the empty query. 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 an *SLS-computed answer substitution* (*c.a.s.* for short) for Q in T. □

We saw in Section 4.4 that SLDNF-resolution is sound w.r.t. the program completion, *comp*(P). A natural question arises: w.r.t. which semantics is SLS-resolution sound? The answer was provided by Przymusinski [125] — it turns out that SLS-resolution is a proof-theoretic counterpart of the perfect model semantics. More precisely, he established the following results.

Theorem 9.5 (Soundness) *Let P be a locally stratified program, Q a query and R a selection rule. Let M_P be the unique perfect Herbrand model of P. Consider the main tree T of the SLS-tree for P and Q via R.*

- *If τ is a c.a.s. for Q in T, then $M_P \models \forall Q\tau$,*
- *if T is failed, then $M_P \models \forall\neg Q$.* □

Corollary 9.6 *SLS-resolution for locally stratified programs is also sound w.r.t. the unique stable model semantics and well-founded semantics.* □

Theorem 9.7 (Completeness) *Let P be a locally stratified program, Q a query and R a selection rule. Let M_P be the unique perfect model of P. Consider the main tree T of the SLS-tree for P and Q via R. Suppose T does not flounder.*

- *If $M_P \models \forall Q\tau$, then there is a c.a.s. σ for Q in T such that $Q\sigma$ is more general than $Q\tau$,*
- *if $M_P \models \forall\neg Q$, then T is failed.* □

Corollary 9.8 *SLS-resolution for locally stratified programs is also complete in the absence of floundering w.r.t. the unique stable model semantics and well-founded semantics.* □

9.2 SLS-resolution for General Programs

Although its name suggests that SLS-resolution can only be used for stratified programs, several proposals for top-down computation of the well-founded semantics are also called SLS-resolution. The ones we found in the literature, which we discuss in this section, all have the disadvantage of requiring a *positivistic* selection rule. This means that a negative literal is selected only if no more positive literals are available.

Przymusinski [130] observed that (a suitable variant of) the iterated least fixpoint definition of the well-founded semantics suggests a *dynamic stratification* of the program: if a ground atom A is decided (becomes true or false) in iteration α, then α is the dynamic stratum of A. An SLS-derivation for an atom A in stratum α is defined by induction on α and consists now of two phases.

In the first phase, positive literals are selected, and the derivation proceeds like an SLD-derivation. This derivation *fails* if it is finitely failed or diverges. If the derivation does not fail in this phase, then it ends in a query with only negative literals (possibly none).

In the second phase, ground negative literals $\neg B$ for which the stratum of B is less than α are selected one by one. By induction on stratum, the SLS-tree T for B is already defined. This case is handled as in Definition 9.3:

- if T is contains the empty query, then the derivation fails,
- otherwise, if T contains a floundering derivation, then the derivation flounders,
- otherwise, $\neg B$ is removed; the derivation continues with the remaining negative literals.

If the derivation completes both phases, then there are three possible outcomes:

- if the derivation ends in the empty query, then it is successful,
- if the derivation ends in a query containing a nonground negative literal, then it flounders,
- otherwise, the derivation ends in an *undefined leaf*.

In addition to the ineffective negation as failure rule, here also the criteria for the selection rule seem to be very ineffective: how can we compare the strata of atoms without computing their truth value in the well-founded model? Przymusinski remarks ([130], Remark 9.1) that the requirement translates into 'no negative recursion is allowed in the derivation'. Thus an interpreter implementing this form of resolution may select a 'wrong' negative literal, find that it leads to negative recursion and 'backtrack' over the selection. A problem with this approach is that in this way, part of the search of the interpreter is not represented in the resulting SLS-tree.

In later versions [122, 140], a sequence of 'SLS-trees of rank α' is created, in which negative literals are decided on the basis of an SLS-tree of a one lower rank, if possible, and skipped otherwise. Skipping here means that another literal is selected. The selection rule is not explicitly required to be positivistic, but in the SLS-tree of rank 1, all negative selected literals will be skipped, thus the effect is that of a positivistic selection rule. (An SLS-tree of rank $\alpha+1$ extends the tree of rank α only at its nodes that contain exclusively skipped negative literals.)

Another hidden property of this selection mechanism is that negative literals are effectively selected in parallel: for each of them it is tried at each rank if it can be resolved (until one fails, or until they have all succeeded). A positivistic selection rule that is *negatively parallel* (selects all negative literals at once) is explicitly used by Ross [147]; it is called *preferential.*

Ross defines *SLP-trees* (the 'P' stands for 'Positivistic') as the result of the first phase described above. Then he defines *Global SLS-resolution* by means of global SLS-trees as follows.

Definition 9.9 (Global SLS-Tree) A global tree Γ for a query Q has three types of nodes:

- tree nodes, which are labeled by SLP-trees for intermediate goals,
- negation nodes, which are labeled by a query with only negative literals (possibly none), and
- nonground nodes, which have no label. □

The root of Γ is the SLP-tree for Q.

Each tree node T in Γ_G has as its children negation nodes: if Q is a leaf of T that contains only negative literals, then Q is a child of T in Γ_G.

Each negation node $Q = \neg A_1, \ldots, \neg A_n$ $(n \geq 0)$ has n children: for $i \in [1, n]$ if A_i is ground, then the child is a tree node, namely the SLP-tree for A_i, otherwise the child is a nonground node. □

Every node in a global tree has a *status*: successful, failed, indeterminate or floundered.

Definition 9.10 (Status of nodes) Consider a global tree.

- A nonground node is always *floundered*.
- A negation node is *failed* if one of its children is successful,
successful if all its children are failed,
floundered if none of its children is successful, and at least one is floundered.
- A tree node is *failed* if all its children are failed,
successful if one of its children is successful,
floundered if one of its children is floundered.
- Nodes that are not assigned a status according to these rules are *indeterminate*. □

A tree node can be both successful and floundered, but no other pair of status is possible for a single node.

Definition 9.11 Let Q be a query. Let T be the root node of a global SLS-tree Γ for Q (thus T is an SLP-tree for Q). A *successful branch* of T is a branch that ends in a leaf labeled N, such that the corresponding negation node labeled N is successful. The *computed answer substitution* of a successful branch is, again, the composition of the consecutive substitutions along the branch, restricted to the variables of Q. □

Ross [147] proved the following results.

Theorem 9.12 (Soundness) *Let P be a program and Q a query. Let Γ be a global SLS-tree for Q.*

- *If θ is a computed answer substitution in Γ, then $WFM(P) \models \forall(Q\theta)$,*
- *if the root of Γ is failed, then $WFM(P) \models \forall(\neg Q)$.* □

Theorem 9.13 (Completeness) *Let P be a program and Q a query. Let Γ be a non-floundering global SLS-tree for Q.*

- *If $WFM(P) \models \exists Q$, then the root of Γ is successful,*
- *if $WFM(P) \models \forall(\neg Q)$, then the root of Γ is failed,*
- *if $WFM(P) \models \forall(Q\theta)$, then there is a computed answer substitution σ in Γ such that $G\sigma$ is more general than $G\theta$.* □

9.3 SLS-resolution for General Programs via all Selection Rules

In this section we present a definition of SLS-resolution that deals with all general programs and all selection rules; it is new to the best of our knowledge. As the first step, we define *oracle* SLS-trees; in these trees we resolve selected positive literals against program clauses, as usual, but ground negative literals are resolved by using the well-founded model as an oracle. Thus we eliminate all negative recursion. The oracle produces one of the answers **true**, **false** and **unknown**. In order to record the last case properly, substitutions may be annotated by **u** in the following definitions.

Definition 9.14 (Oracle SLS-tree) Let P be a program and R a selection rule. For a query Q, we define the *oracle* SLS-tree T for P and Q via R as follows. The root of T is Q. For *any* node N in T we have:

- If N is the empty query, then N is marked as *success* and has no children.
- If R selects a positive literal L in N, then N has as children those nodes that can be obtained by extending T at N in the sense of Definition 3.5. If no children can be obtained in this way, then N is marked as *failed*.
- If R selects a negative literal $\neg A$ in N, then
  - If A is nonground, then N has no children and is marked as *floundered*.
  - If A is ground, then
    * if A is **true** in $WFM(P)$, then N has no children and is marked as *failed*,
    * if A is **false** in $WFM(P)$, then the resolvent $(\epsilon, N - \{\neg A\})$ is the only child of N,
    * if A is **unknown** in $WFM(P)$, then the resolvent $((\mathbf{u}, \epsilon), N - \{\neg A\})$ is the only child of N. □

Definition 9.15 (Oracle answer substitution) Let P be a program and Q a query. Consider a branch in an oracle SLS-tree T for P and Q which ends with the empty query. 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 an *oracle SLS-computed answer substitution* (*o.c.a.s.* for short) for Q in T, if none of the substitutions α_i is annotated by $\mathbf{u}$; otherwise it is called an *oracle SLS-unknown answer substitution* (*o.u.a.s.* for short) for Q in T.

An oracle SLS-tree T for a query Q is

- successful, if it gives an o.c.a.s. for Q,
- floundered, if it contains a leaf marked as floundered,
- indeterminate, if it is not successful and not floundered, and gives an o.u.a.s. for Q,
- failed, otherwise. □

The following results relate oracle SLS-trees to the well-founded semantics.

Lemma 9.16 (Soundness) *Consider an oracle SLS-tree T for a program P and a query Q.*

- *If τ is a o.c.a.s. for Q in T, then $WFM(P) \models_3 \forall Q\tau$,*
- *if τ is a o.u.a.s. for Q in T, then $WFM(P) \not\models_3 \neg\forall Q\tau$,*
- *if T is failed, then $WFM(P) \models_3 \forall\neg Q$.* □

Lemma 9.17 (Completeness) *Consider an oracle SLS-tree T for a program P and a query Q. Suppose T does not flounder.*

- *If $WFM(P) \models_3 \forall Q\tau$, then there is an o.c.a.s. σ for Q in T such that $Q\sigma$ is more general than $Q\tau$,*
- *if $WFM(P) \not\models_3 \neg\forall Q\tau$, then there is an o.c.a.s. or an o.u.a.s. σ for Q in T such that $Q\sigma$ is more general than $Q\tau$,*
- *if $WFM(P) \models_3 \forall\neg Q$, then T is failed.* □

Proving these lemmas is straightforward: negative literals are given their correct truth value by definition; positive literals are treated as in SLD-resolution.

These results allow the second step of the construction of the SLS-tree. For all nodes N where a ground negative literal $\neg A$ is selected and the oracle is used, we can 'justify' the outcome of the oracle by a subsidiary oracle SLS-tree for A. Either this tree produces the same answer as the oracle, or it flounders. In the latter case, the descendants of N are removed and N becomes a flounder leaf. By recursively adding subsidiary trees for all nodes where the oracle was used, no step involving a selected ground negative literal will remain unjustified.

Definition 9.18 (SLS-tree) Let P be a program, Q a query and R a selection rule. An *SLS-tree* for P and Q via R is defined as the limit of a sequence of *oracle SLS-trees of depth* n $(n \geq 1)$. These are defined by induction. An oracle SLS-tree of depth 1 for P and Q via R consists of only one tree, which is the oracle SLS-tree for P and Q via R. For $n > 1$, an oracle SLS-tree *of depth* n for P and Q via R is a forest $(\mathcal{F}, T, subs)$ obtained as follows.

The main tree T is the oracle SLS-tree for P and Q via R, of which some nodes can be removed. From the root, follow each branch and for every ground negative literal $\neg A$ selected in a node N in T: let $(\mathcal{F}', T', subs')$ be the oracle SLS-tree of depth $n-1$ for P and A via R, set $subs(N)$ to T', extend $subs$ by $subs'$ and $\mathcal{F}$ by $\mathcal{F}'$; if T' is floundered and not successful, then mark N as floundered and remove the children of N, if any. □

The following results relate SLS-trees to the well-founded semantics.

Theorem 9.19 (Soundness) *Consider the main tree T of an SLS-tree for a program P and a query Q.*

- *If τ is a o.c.a.s. for Q in T, then $WFM(P) \models_3 \forall Q\tau$,*
- *if τ is a o.u.a.s. for Q in T, then $WFM(P) \not\models_3 \neg\forall Q\tau$,*
- *if T is failed, then $WFM(P) \models_3 \forall\neg Q$.* □

Theorem 9.20 (Completeness) *Consider the main tree T of an oracle SLS-tree for a program P and a query Q. Suppose T does not flounder.*

- *If $WFM(P) \models_3 \forall Q\tau$, then there is an o.c.a.s. σ for Q in T such that $Q\sigma$ is more general than $Q\tau$,*
- *if $WFM(P) \not\models_3 \neg\forall Q\tau$, then there is an o.c.a.s. or an o.u.a.s. σ for Q in T such that $Q\sigma$ is more general than $Q\tau$,*
- *if $WFM(P) \models_3 \forall\neg Q$, then T is failed.* □

Remark 9.21 Instead of the well-founded model of the program, any stationary model can be used as the oracle in the above definitions. The (oracle) SLS-trees, obtained using the stationary model M as the oracle, will be sound and complete (in the above sense) w.r.t. M. □

Although we have defined SLS-trees in such a way that they are sound and complete w.r.t. the well-founded semantics, it is not at all clear how an interpreter could construct these trees in a top-down way. This brings us to the issue of implementation.

9.4 Implementation

As mentioned before, SLS-resolution is not effective, thus it is not fully implementable. But it is possible to make a sound implementation that is complete for a limited class of programs, e.g. programs without function symbols. It is then essential to capture those infinite derivations that have the form of a loop. This can be done by simple loop checking techniques, or by tabulation (also known as memo-ization or lemma resolution).

For locally stratified programs, loop checking was studied by Bol [24]. Tabulation for stratified programs was studied by Kemp and Topor [81] for SLS-resolution and Seki and Itoh [155] for SLDNF-resolution. By definition, in this setting only positive loops have to be dealt with. So their approach can remain close to tabulation for positive programs, by maintaining a table for each stratum.

Chen and Warren [37] added a tabulation mechanism to the form of SLS-resolution proposed by Przymusinski and Warren [130, 140] in order to detect positive loops. Negative loops are detected by maintaining a *negative context*: the set of negative literals that may be assumed undefined, because they are encountered in a loop. In this way the tables must be constructed for all relevant negative contexts. This gives many redundant computations, and a rather complex result (the final construction is a forest of forests ...). Bidoit and Legay [23] proposed a similar system, computing the defined atoms and the potentially defined atoms separately.

Recently, Bol and Degerstedt [25] proposed a simpler method that uses tabulation to detect both positive and negative loops. Only one table needs to be constructed, but their definition of failure is somewhat complicated.

Finally, it should be mentioned that a top-down computation of the well-founded semantics for ground programs is described by Pereira, Aparício and Alferes [117]; instead of tabulation, it uses both positive and negative contexts. Such use of positive contexts does not generalize to the nonground case (as was shown by Apt, Bol and Klop [10]).

10 Disjunctive Programs

In a disjunctive logic program, the heads of clauses can be disjunctions of one or more atoms. Numerous semantics were proposed for such programs. They are classified in Dix [45].

Positive disjunctive programs allow the expression of indefinite (incomplete) knowledge, which is impossible in definite programs. As examples consider the following natural statements:

$$\texttt{mother(X)} \vee \texttt{father(X)} \leftarrow \texttt{parent(X)},$$
$$\texttt{red(X)} \vee \texttt{blue(X)} \vee \texttt{green(X)} \leftarrow \texttt{primary\_colour(X)}.$$

The addition of negation allows us to express indefinite knowledge, as well, so one may wonder whether there is any use in allowing disjunctions in general logic programs. Indeed there is: as negation in logic programming is not classical negation, the effect of a clause $p \leftarrow \neg q$ is quite different from $p \vee q$. The pair $\{p \leftarrow \neg q;\ q \leftarrow \neg p\}$ is a better approximation — at least it retains the symmetry between p and q — but it is still not adequate. It introduces a loop through negation, which renders some semantics inapplicable and causes obvious problems in the proof theory. Furthermore, the well-founded model of the program $\{p \leftarrow \neg q;\ q \leftarrow \neg p;\ r \leftarrow p;\ r \leftarrow q\}$ does not contain r, as one might expect.

10.1 Positive Disjunctive Programs

Lobo, Minker and Rajasekar [98] recently published a book about the foundations of disjunctive logic programming, of which the larger part deals with positive programs. We shall briefly recall some semantics for positive disjunctive programs; for a more elaborate discussion, motivation and proof theory, we refer to this book.

An important distinction, which can be made already for positive disjunctive programs, is that between an *inclusive* and *exclusive* interpretation of disjunctions. For example, if we have the program $\{p \leftarrow;\ p \vee q \leftarrow\}$, then the exclusive reading concludes that q is false, whereas the inclusive reading does not conclude anything about q.

Recall that, for definite programs, the negation as (finite or infinite) failure rule can be viewed as the application of the Closed World Assumption (see Section 1.2)

$$P \models_{CWA} \neg A \text{ iff } P \not\models A.$$

This rule must be rephrased for disjunctive programs, as in this form it gives rise to inconsistencies. Indeed we have $p \vee q \models_{CWA} \neg p$ and $p \vee q \models_{CWA} \neg q$, so $P \cup \{\neg A \mid P \models_{CWA} \neg A\}$ is inconsistent.

The *Generalized Closed World Assumption* (GCWA) of Minker [108] is such a rephrasing; it says:

$$P \models_{GCWA} \neg A \text{ iff } \neg A \text{ is true in all minimal models of } P.$$

GCWA gives rise to an *exclusive* interpretation of disjunctions.

The *Weak Generalized Closed World Assumption* (WGCWA) was developed independently by Lobo, Minker and Rajasekar [141] and by Ross and Topor [148]. It was originally defined as a computational simplification of GCWA, that infers less negative literals. Let P^* be the program obtained from P by replacing $\vee$ by $\wedge$, i.e., a clause $A_1 \vee \ldots A_n \leftarrow \mathbf{B}$ in P yields the clauses $A_1 \leftarrow \mathbf{B} \ \ldots \ A_n \leftarrow \mathbf{B}$ in P^*. Then

$$P \models_{WGCWA} \neg A \text{ iff } P^* \models_{CWA} \neg A$$

WGCWA gives rise to an *inclusive* interpretation of disjunctions. Notice that CWA, GCWA and WGCWA coincide on definite programs.

Even less negative literals than from WGCWA can be inferred from the *completion* of a disjunctive program, which was defined by Lobo, Rajasekar and Minker [97, 98]. It consists of P, augmented with EQ and the only-if (i.e. $\rightarrow$) part of the completion of P^*.

Theorem 10.1 (Dix [45]) *WGCWA is rational, GCWA is cumulative, but not rational.* $\square$

The program $P = \{p \vee q \leftarrow ; \ r \leftarrow p; \ s \leftarrow q, r\}$ is a counterexample against the rationality of GCWA. The minimal models of P are $\{p, r\}$ and $\{q\}$, thus $P \not\models_{GCWA} \neg r$ and $P \models_{GCWA} \neg s$. But $P \cup \{r\}$ has the minimal models $\{p, r\}$ and $\{q, r, s\}$, thus $P \cup \{r\} \not\models_{GCWA} \neg s$. Notice that $P \not\models_{WGCWA} \neg s$.

10.2 Locally Stratified Disjunctive Programs

The definition of locally stratified programs can be generalized to disjunctive programs: if two atoms are disjuncts in the head of a ground instance of a program clause, then these atoms must be in the same stratum. The definition of perfect models (Definition 6.8) generalizes immediately to locally stratified disjunctive programs. Of course, a disjunctive program may have more than one perfect model.

Definition 10.2 (Perfect model semantics) Przymusinski [128] The *perfect model semantics* of a disjunctive program P is defined by putting for a ground atom A:

A is true (false), if A is true (false) in all perfect models of P. $\square$

Definition 10.3 (Weak perfect model semantics) (Dix [45]) The *weak perfect model semantics* of a disjunctive program P is defined by putting for a ground atom A:
A is true (false), if A is true (false) in all perfect models of P and in the perfect model of P^*.[5]
$\square$

Again, perfect model semantics interprets disjunctions exclusively, whereas weak perfect model semantics inclusively. Perfect model semantics extends GCWAS, the Generalized Closed World Assumption for Stratified programs, which was defined by Rajasekar and Minker [142]. A weak version of GCWAS, called WGCWAS, was defined by Dix [45]; weak perfect model semantics extends it.

[5]Notice that P^* is a locally stratified program, because P is a locally stratified disjunctive program.

Theorem 10.4 (Dix [45])

- *Perfect model semantics and GCWAS coincide with GCWA on positive disjunctive programs.*
- *Weak perfect model semantics and WGCWAS coincide with WGCWA on positive disjunctive programs.*
- *Perfect model semantics and GCWAS, Weak perfect model semantics and WGCWAS are cumulative.*
- *Of these semantics, only WGCWAS is rational.* □

10.3 General Disjunctive Programs

Semantics for all general disjunctive programs that coincide with the well-founded semantics on general programs, and that also coincide with the perfect (or weak perfect) model semantics on locally stratified disjunctive programs, have been proposed by Przymusinski [126] and by Dix [45].

Przymusinski defines *stationary expansions* of disjunctive programs by generalizing Definition 7.13 in the following ways.

- Instead of a set of not_A atoms, a set C of *disjunctions* of not_A atoms is added to the program P.
- The second condition on interpretations that are considered when determining minimal models is generalized to the *disjunctive inference rule*:

  $$\text{if } P \cup C \models A_1 \vee \ldots \vee A_n, \text{ then } I \models not\_A_1 \wedge \ldots \wedge not\_A_k \rightarrow A_{k+1} \vee \ldots \vee A_n,$$

  where $1 \leq k \leq n$ and the empty disjunction is interpreted as **false**.
- For a negative *disjunction* $F = \neg A_1 \vee \ldots \vee \neg A_n$, $P \cup C \models_{min} F$ if F is true in all minimal models of $P \cup C$ (according to this particular notion of minimality).
- The fixpoint equation that defines stationary expansions becomes

  $$E(P) = P \cup \{not\_A_1 \vee \ldots \vee not\_A_n \mid E(P) \models_{min} \neg A_1 \vee \ldots \vee \neg A_n\}.$$

In another version of the semantics, Przymusinski used

$$\text{if } P \cup C \models_{min} \neg A_1 \vee \ldots \vee \neg A_n, \text{ then } I \models A_1 \wedge \ldots \wedge A_k \rightarrow not\_A_{k+1} \vee \ldots \vee not\_A_n$$

as the disjunctive inference rule (which implicitly makes the definition of $\models_{min}$ recursive). Dix [45] reformulates and compares these two versions, together with a third version (using essentially the first disjunctive inference rule, restricted to $k = n$). This third version is weaker than the perfect model semantics on locally stratified disjunctive programs.

Dix also defines *weak stationary semantics*: a *weak stationary extension* satisfies the fixpoint equation

$$E(P) = P \cup \{not\_A_1 \vee \ldots \vee not\_A_n \mid E(P)^* \models_{min} \neg A_1 \vee \ldots \vee \neg A_n\}.$$

(The disjunctive inference rule is the third one of those mentioned above.) Weak stationary semantics interprets disjunctions inclusively.

Theorem 10.5 (Dix [45])

- *Stationary semantics for disjunctive programs is not cumulative.*
- *Weak stationary semantics is cumulative, but not rational.*
- *For locally stratified disjunctive programs, weak stationary semantics decides more atoms than WGCWAS, but less than weak perfect semantics.* □

Finally, Dix [45] defines a semantics, DWFS, which coincides with the well-founded semantics on general programs, and with the perfect model semantics on locally stratified disjunctive programs. It is weaker than the stationary semantics, and cumulative. A weak version of it, WDWFS, also coincides with the well-founded semantics on general programs, and with the weak perfect model semantics on locally stratified disjunctive programs. It is stronger than weak stationary semantics, and cumulative.

A rather different approach is taken by Ross [146]: he defines a semantics for general disjunctive programs through a top-down procedure generalizing Definition 9.9. He defines three versions: Strong well-founded semantics, with an exclusive interpretation of disjunctions; Weak well-founded semantics, with an inclusive interpretation of disjunctions; and finally Optimal well-founded semantics, where the program(mer) defines the inclusive or exclusive nature separately for each clause. On general programs, these semantics coincide with the well-founded semantics. However, when restricted to locally stratified disjunctive programs, the strong version is weaker than perfect model semantics and the weak version is weaker than weak perfect model semantics.

Two fixpoint semantics that extend the stationary semantics are GDWFS and WF^3 by Baral, Lobo and Minker [14, 98], respectively [15]. WF^3 extends GDWFS; both coincide with GWFS on general programs (thus they are not cautious and do not satisfy PPE, the properties defined in Section 1.3). They are incomparable with perfect model semantics on locally stratified disjunctive programs.

Sakama and Inoue [152] defined GCWA$\neg$ and WGCWA$\neg$, based on an extension of stable models to disjunctive programs. These semantics coincide with perfect, respectively weak perfect model semantics on locally stratified disjunctive programs.

Clearly, the issue of what is the *right* semantics for general disjunctive programs is far from being decided. Its seems that the weaker semantics have some advantages:

- they are cumulative and satisfy Dix's weak principles,
- the complexity of computing them is sometimes lower (for example WGCWA has lower complexity than GCWA, but the complexity of perfect and weak perfect model semantics is the same, see also Müller and Dix [112]),
- uncertainty is safe, that is, if the semantics draws more conclusions from the program than the programmer intended, then the results are probably worse than when some intended conclusions are missed.

11 Final Remarks

We introduced in this paper two lines of research dealing with semantics of general programs. The first one was considered in Section 4 and focused on the completion of a program. The second line was considered in Sections 6 and 7 and focused on various attempts of extending the concept of a "special" Herbrand model to general programs. In each category we studied a number of proposals which resulted in quite an array of possibilities.

11.1 Reconciliation

It is useful to characterize a class of programs for which these approaches coincide. This problem was considered by Apt and Bezem [4] who showed that for acyclic programs practically all approaches considered in this paper coincide. More specifically, they proved the following result.

Theorem 11.1 *Let P be an acyclic program. Then*

- *the T_P operator has a unique fixpoint, N_P,*
- *N_P is a unique fixpoint of the $T3_P$ operator,*
- *N_P is a unique perfect model of P,*
- *N_P is a unique Herbrand model of comp(P),*
- *SLDNF- and SLS-trees coincide for bounded queries.* □

Consequently, by the Fixpoint Lemma 4.12 N_P is also a unique 3-valued Herbrand model of *comp*(P). Additionally, as every acyclic program is locally stratified, by the Unique stable model Theorem 6.20 N_P is also a unique stable model of P, and consequently by Corollary 7.7 it is the well-founded model of P, as well.

These results were generalized by Apt and Pedreschi [6] to a larger class of programs corresponding with termination w.r.t. the leftmost selection rule, as opposed to termination w.r.t. all selection rules (in the sense of the Terminating Program Definition 3.16). Recently, Fitting [63] provided an alternative proof of these results by means of metrics and Banach Contraction Theorem.

A number of interesting programs turn out to be acyclic. By the above theorem all approaches to their semantics coincide. For instance, the program `TWEETY` of Section 1.3, and the programs `SINK`, `NUMBERS` and `EVEN` of Sections 3.1, 5 and 6.2 are acyclic. Another example is a natural formalization of the so-called Yale shooting problem of Hanks and McDermott [75], which is an example of temporal reasoning, an instance of non-monotonic reasoning. This problem was extensively discussed in the literature and its formalizations in various formalisms for non-monotonic reasoning were studied. In relation to logic programming we note three independent references – that of Apt and Bezem [4], who proved that the translation of the Yale Shooting Problem to a logic program results in an acyclic program, Elkan [56], who showed that this translation results in a locally stratified program, and Evans [58], who observed that SLDNF-resolution can be used to compute desired consequences of the original formulation of the problem in first-order logic.

In contrast, the program `EVEN'` of Section 6.2 is not locally stratified, so a fortiori not acyclic. However, it is possible to apply to it a result of Apt and Pedreschi [6] and draw the same conclusions as for the above programs.

11.2 Topics not Treated

The range of topics that fall within 'Logic Programming and Negation' is so enormous, that inevitably we have to refrain from treating them all. Here follows a short list of topics we left out.

Deductive databases form an extension of relational databases in which some of the relations are implicitly defined. Ignoring the built-in relations, their syntax coincides with that of logic programs. In the area of deductive databases, negation also formed an important research subject. Parts of this research (like stratification and the use of perfect model semantics) overlap with that of logic programming. Some other topics are more intrinsic for the field, in particular query processing (see e.g. Kemp and Topor [82] and Balbin et al. [12]), integrity constraint checking (see e.g. Lloyd, Sonenberg and Topor [96] and Sadri and Kowalski [151]), handling of updates (see e.g. Naqvi and R. Krishnamurthy [115]) and comparison of expressive power between various query languages (see e.g. Chandra and Harel [35]). More recent research in this area is surveyed in Kanellakis [80] and Bidoit [20].

Classical Negation, also called *explicit* or *strong negation*, was introduced by Gelfond and Lifschitz [70, 71]. It involves a second kind of negation that may occur both in the head and in the body of clauses. Their motivation was to capture in logic programming more complicated forms of temporal reasoning, than the one exemplified in the usual formalization of the Yale shooting problem.

When both kinds of negation are present, $\neg$ usually denotes classical negation; negation by failure is then denoted by $\sim$. Semantically, classically negated atoms are usually treated as new atoms. However, in the process of selecting 'intended' models, the 'inconsistent' ones (that is, the ones containing an atom A and its classical negation $\neg A$) are discarded. Overviews of this area can be found in Alferes and Pereira [1], Wagner [171] and and Minker and Ruiz [110].

Abductive Logic Programming views, roughly speaking, the query as an observation, which must be explained by means of additional hypotheses. Explanations can be found by following the rules of the program 'backwards', as in SLD-resolution and its generalizations. A survey on abductive logic programming, by Kakas, Kowalski and Toni [79] appeared recently.

Truth Maintenance Systems can be viewed as an extension of (propositional) general logic programs, where some clauses (called *constraints*) have the constant **false** as the head. Semantics have been proposed for truth maintenance systems by generalizing stable and well-founded semantics to deal with constraints. We mention here work by Elkan [55], Reinfrank [143], Giordano and Martelli [73], Witteveen [174] and Jonker [78]. The area is related to classical negation and to abduction.

Relations with Other Non-monotonic Formalisms are abundant (see e.g. Nerode et al. [116] and Przymusinski [131, 134]). As negation as failure is a non-monotonic inference rule, there has been a cross-fertilization between semantics for non-monotonic logics and logic programming.

In one direction, stable expansions of *auto-epistemic* logic (Moore [111]) inspired Gelfond [67, 68] to define the stable semantics. A parallel work on connections between the *default logic* of Reiter [145] and stable model semantics was carried out by Marek and Truszczyński [105] and by Bidoit and Froidevaux [21]. Recently, Przymusinski [135, 127] explained the stationary semantics by means of auto-epistemic logic (see also Bonatti [26]).

In the other direction, Przymusinski [139] introduced 3-valued versions of default logic and auto-epistemic logic, based on the well-founded semantics for logic programs. For default logic, this semantics was generalized further by Baral and Subrahmanian [16], Li and You [90] and Przymusinska and Przymusinski [121]. A unifying framework for the semantics of auto-epistemic logic, based on stationary semantics for logic programs, was presented by Przymusinski [137].

The relation between logic programming and *circumscription* (McCarthy [107]) was studied by Lifschitz [91, 69] and Gelfond, Przymusinski and Przymusinska [72].

Recursion Theoretic Analysis of the concepts disscussed here attracted a lot of interest. The complexity of the syntactic notions (like (local) stratifiability), of the proof theory (like SLS-resolution), and of semantics (like well-founded model) were studied both in the propositional and first-order case. These results are surveyed in Cadoli en Schaerf [28].

Intensional Negation is an approach to negation that transforms a program P (without local variables) into a program $\overline{P}$, defining a relation $\overline{p}$ for every relation p in P, such that $\overline{p}(\mathbf{t})$ succeeds from $\overline{P}$ iff $p(\mathbf{t})$ finitely fails from P, and vice versa. Intensional negation was mainly studied by Mancarella et al. [101, 102, 18].

Linear Logic is a modification of the classical Gentzen sequent calculus which was developed by Girard [74] to capture reasoning about resources. In particular linear logic is sensitive to how many times a formula is used as hypothesis in a proof. Cerrito [32, 33] showed that linear logic can be used to reason about logic programs by providing a translation of logic programs and program completions to linear logic theories for which soundness and completeness of the SLDNF-resolution for allowed programs can be established and for which SLDNF-resolution with the leftmost selection rule can be adequately interpreted.

Acknowledgements We would like to thank all five referees for useful comments, Rachel Ben-Eliyahu and Jürgen Dix for extensive suggestions, and Kees Doets, Marco Schaerf and Robert Stärk for helpful discussions on the subject of this paper.

References

[1] J.J. Alferes and L.M. Pereira. On logic program semantics with two kinds of negation. In Apt [3], pages 574–589.

[2] K. R. Apt. Logic programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 493–574. Elsevier, 1990. Vol. B.

[3] K. R. Apt, editor. *Proc. of the Joint International Conference and Symposium on Logic Programming*, Washington, 1992. ALP, MIT Press.

[4] K. R. Apt and M. Bezem. Acyclic programs. *New Generation Computing*, 29(3):335–363, 1991.

[5] K. R. Apt and H. A. Blair. Arithmetic classification of perfect models of stratified programs. *Fundamenta Informaticae*, 13:1–18, 1990. Addendum in vol. 14, pages 339-344, 1991.

[6] K. R. Apt and D. Pedreschi. Reasoning about termination of pure Prolog programs. *Information and Computation*, 106(1):109–157, 1993.

[7] K. R. Apt and A. Pellegrini. On the occur-check free Prolog programs. Technical Report CS-R9238, CWI, Amsterdam, 1992. To appear in *ACM Toplas*.

[8] K. R. Apt and M.H. van Emden. Contributions to the theory of logic programming. *Journal of the ACM*, 29(3):841–862, 1982.

[9] K.R. Apt, H. Blair, and A. Walker. Towards a Theory of Declarative Knowledge. In Minker [109], pages 193–216.

[10] K.R. Apt, R.N. Bol, and J.W. Klop. On the safe termination of Prolog programs. In Levi and Martelli [89], pages 353–368.

[11] K.R. Apt and H.C Doets. A new definition of SLDNF-resolution. ILLC Prepublication Series CT-92-03, Department of Mathematics and Computer Science, University of Amsterdam, The Netherlands, 1992. To appear in *Journal of Logic Programming*.

[12] I. Balbin, G.S. Port, Ramamohanarao K., and K. Meenakshi. Efficient bottom-up computation of queries on stratified databases. *J. of Logic Programming*, 11:295–344, 1991.

[13] C. Baral, J. Lobo, and J. Minker. Generalized well-founded semantics for logic programs. In M. Stickel, editor, *Proc. 10th International Conference on Automated Deduction,* LNAI 449, pages 102–116, July 1989.

[14] C. Baral, J. Lobo, and J. Minker. Generalized disjunctive well-founded semantics for logic programs. In Z.W. Ras and M. Zemankova, editors, *Proceedings of the Fifth International Symposium on Methodologies for Intelligent Systems*, pages 456–473, October 1990.

[15] C. Baral, J. Lobo, and J. Minker. WF^3: A semantics for negation in normal disjunctive logic programs. In *Proceedings of the Sixth International Symposium on Methodologies for Intelligent Systems*, Charlotte, NC, 1991.

[16] C. Baral and V.S. Subrahmanian. Stable and extension class theory for logic programs and default logics. *J. of Automated Reasoning*, 8(3):345–366, 1992.

[17] S. Baratella. Models of completion for some classes of logic programs. *Fundamenta Informaticae*, 14,:323–339, 1991.

[18] R. Barbuti, P. Mancarella, D. Pedreschi, and F. Turini. A transformational approach to negation in logic programming. *J. of Logic Programming*, 8:201–228, 1990.

[19] R. Ben-Eliyahu. From program completion to default logic. In *IAICVNN-93: proceedings of the 10th Israeli symposium on artificial intelligence, computer vision, and neural networks*, Ramat Gan, Israel, December 1993. Also a poster in the 2nd International workshop on Logic programming and nonmonotonic reasoning, Lisbon, Portugal, June 1993.

[20] N. Bidoit. Negation in rule-based database languages: a survey. *Theoretical Computer Science*, 78:3–83, 1991.

[21] N. Bidoit and C. Froidevaux. General logical databases and programs: Default logic semantics and stratification. *Information and Computation*, 91:15–54, 1991.

[22] N. Bidoit and C. Froidevaux. Negation by default and unstratifiable logic programs. *Theoretical Computer Science*, 78:85–112, 1991.

[23] N. Bidoit and P. Legay. WELL!: An evaluation procedure for all logic programs. In *Proceedings of the International Conference on Database Technology*, pages 335–348, 1990.

[24] R.N. Bol. Loop checking and negation. *J. of Logic Programming*, 15(2):147–175, 1993. Extended abstract in Eijck [54], pages 121–138.

[25] R.N. Bol and L. Degerstedt. Tabulated resolution for well-founded semantics. In D. Miller, editor, *Proc. of the 1993 International Logic Programming Symposium*, pages 199–219, 1993.

[26] P.A. Bonatti. Auto-epistemic logics as a unifying framework for the semantics of logic programs. In Apt [3], pages 417–430.

[27] E. Börger. Unsolvable decision problems for Prolog programs. In *Computation Theory and Logic*, Lecture Notes in Computer Science 270, pages 3–48. Springer-Verlag, 1987.

[28] M. Cadoli and M. Schaerf. A survey on complexity results for non-monotonic logics. *Journal of Logic Programming*, 17(2,3 & 4):127–160, 1993.

[29] L. Cavedon. Continuity, consistency, and completeness properties for logic programs. In G. Levi and M. Martelli, editors, *Proceedings of the Sixth International Conference on Logic Programming*, pages 571–584. The MIT Press, 1989.

[30] L. Cavedon. Acyclic programs and the completenss of SLDNF-resolution. *Theoretical Computer Science*, 86(1):81–92, 1991.

[31] L. Cavedon and J.W. Lloyd. A completeness theorem for SLDNF resolution. *The Journal of Logic Programming*, 7:177–191, 1989.

[32] S. Cerrito. A linear semantics for allowed logic programs. In *Proceedings of the 5th Symposium on Logic in Computer Science (LICS '90)*, pages 219–227, Philadelphia, PA., USA, 1991.

[33] S. Cerrito. A linear axiomatization of negation as failure. *Journal of Logic Programming*, 12(1 & 2):1–24, 1992.

[34] D. Chan. Constructive negation based on the completed database. In Kowalski and Bowen [84], pages 111–125.

[35] A.K. Chandra and D. Harel. Horn clause queries and generalizations. *Journal of Logic Programming*, 2(1):1–15, 1985.

[36] J. Chen and S. Kundu. The strong semantics for logic programs. In Z.W. Ras and M. Zemankova, editors, *Proc. of the Sixth International Symposium on Methodologies for Intelligent Systems* LNAI 542, pages 490–499, Charlotte, NC, 1991. Springer-Verlag.

[37] W. Chen and D. S. Warren. A goal-oriented approach to computing well-founded semantics. In Apt [3], pages 589–603.

[38] K. Clark. Logic-programming schemes and their implementation. In J.L. Lassez and G. Plotkin, editors, *Computational Logic: Essays in Honor of Alan Robinson*, pages 487–541. The MIT Press, Cambridge, Massachussets, 1991.

[39] K. L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Databases*, pages 293–322. Plenum Press, New York, 1978.

[40] A. Cortesi and G. Filé. Classes of programs with consistent completion. Technical report, Universitá di Padova, Dip. di Matematica Pura e Applicata, 1992.

[41] A. Cortesi and G. Filé. Graph properties for normal logic programs. *Theoretical Computer Science*, 107(2):277–303, 1993.

[42] H. Decker. On generalized cover axioms. In Furukawa [64], pages 693–707.

[43] H. Decker and L. Cavedon. Generalizing Allowedness While Retaining Completeness of SLDNF Resolution. In E. Börger, G. Jäger, H. Kleine-Büning, and M.M. Richter, editors, *CSL '89, 3rd Workshop on Computer Science Logic, Kaiserslautern, FRG, LNCS 440*, pages 98–125, 1989.

[44] J. Dix. Classifying Semantics of Logic Programs. In Anil Nerode, Wiktor Marek, and V. S. Subrahmanian, editors, *Logic Programming and Non-Monotonic Reasoning, Proceedings of the first International Workshop*, pages 166–180. Washington D.C, MIT Press, July 1991.

[45] J. Dix. Classifying semantics of disjunctive logic programs. In Apt [3], pages 589–603.

[46] J. Dix. A framework for representing and characterizing semantics of logic programs. In B. Nebel, C. Rich, and W. Swartout, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the Third International Conference (KR92)*, San Mateo, CA, 1992. Morgan Kaufmann.

[47] 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.

[48] J. Dix. A Classification-Theory of Semantics of Normal Logic Programs: I. Strong Principles. *Fundamenta Informaticae*, forthcoming, 1994.

[49] J. Dix. A Classification-Theory of Semantics of Normal Logic Programs: II. Weak Principles. *Fundamenta Informaticae*, forthcoming, 1994.

[50] H. C. Doets. Levationis laus. *Journal of Logic and Computation*, 3(5):487–516, 1993.

[51] W. Drabent. What is failure? An approach to constructive negation. Draft, 1992. Provisionally accepted by Acta Informatica.

[52] W. Drabent and M. Martelli. Strict completion of logic programs. *New Generation Computing*, 9(1):69–79, 1991.

[53] Phan Minh Dung. On the relation between stable and well-founded semantics of logic programs. *Theoretical Computer Science*, 105(1):7–25, 1992.

[54] J. van Eijck, editor. *Logics in AI – JELIA'90,* LNAI 478. Springer Verlag, 1990.

[55] C. Elkan. Logical characterizations of non-monotonic TMSs. In A. Kreczmar and G. Mirkowska, editors, *Mathematical Foundations of Computer Science,* LNCS 379, pages 218–224. Springer-Verlag, 1989.

[56] C. Elkan. A perfect logic for reasoning about action, 1989. Manuscript. University of Toronto.

[57] C. Elkan. A rational reconstruction of nonmonotonic truth maintenance systems. *Artificial Intelligence*, 43:219–234, 1990.

[58] C. Evans. Negation-as-failure as an approach to the Hanks and McDermott problem. In *Proceedings of the Second International Symposium on Artificial Intelligence, Monterrey, Mexico*, 1989.

[59] F. Fages. A new fixpoint semantics for general logic programs compared with the we ll-founded and the stable model semantics. *New Generation Computing*, 9(4), 1991.

[60] F. Fages. Consistency of Clark's completion and existence of stable models. *Methods of Logic in Computer Science*, 2, 1993. to appear.

[61] M. Fitting. A Kripke-Kleene semantics for general logic programs. *Journal of Logic Programming*, 2:295–312, 1985.

[62] M. Fitting. The family of stable models. *J. of Logic Programming*, 17(2,3 & 4):197–226, 1993.

[63] M. Fitting. Metric methods; three examples and a theorem. *J. of Logic Programming*, 1993. To appear.

[64] K. Furukawa, editor. *Proceedings of the Eighth International Conference on Logic Programming*, Paris, France, 1991. MIT Press.

[65] A. van Gelder. The Alternating Fixpoint of Logic Programs with Negation. In *Proc. of the Symposium on Principles of Database Systems*, pages 1–10. ACM SIGACT-SIGMOD, 1989.

[66] A. van Gelder, K. Ross, and J. Schlipf. The well-founded semantics for general logic programs. *J. of the ACM*, 38(3):620–650, 1991.

[67] M. Gelfond. On stratified auto-epistemic theories. In *Proceedings AAAI-87*, pages 207–211. American Association for Artificial Intelligence, Morgan Kaufmann, 1987.

[68] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In Kowalski and Bowen [84], pages 1070–1080.

[69] M. Gelfond and V. Lifschitz. Compiling circumscriptive theories into logic programs. In Reinfrank, De Kleer, Ginsberg, and Sandewall, editors, *Non-Monotonic Reasoning* LNAI 346, pages 74–99. Springer-Verlag, January 1989.

[70] M. Gelfond and V. Lifschitz. Logic programs with classical negation. In Warren and Szeredi [173], pages 579–597.

[71] M. Gelfond and V. Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9, 1991.

[72] M. Gelfond, H. Przymusinska, and T.C. Przymusinski. On the relationship between circumscription and negation as failure. *Artificial Intelligence*, 38:75–94, 1989.

[73] L. Giordano and A. Martelli. Generalized stable models, truth maintenance and conflict resolution. In Warren and Szeredi [173], pages 427–441.

[74] Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[75] S. Hanks and D. McDermott. Nonmonotonic logic and temporal projection. *Artificial Intelligence*, 33:379–412, 1987.

[76] P.M. Hill and J.W. Lloyd. The Gödel programming language. Technical Report CSTR-92-27, Department of Computer Science, University of Bristol, 1992. Revised May 1993. To appear in The MIT Press.

[77] Y. Hu and L.Y. Yuan. Extended well-founded model semantics for general logic programs. In Furukawa [64], pages 412–425.

[78] C.M. Jonker. Cautious backtracking and well-founded semantics in truth maintenance systems. Technical Report RUU-CS-91-26, Utrecht University, 1991.

[79] A.C. Kakas, R.A. Kowalski, and F. Toni. Abductive logic programming. *Journal of Logic and Computation*, 2(6):719–770, 1993.

[80] P. Kanellakis. Elements of relational database theory. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 1073–1156. Elsevier, 1990. Vol. B.

[81] D. B. Kemp and R. W. Topor. Completeness of a top-down query evaluation procedure for stratified databases. In Kowalski and Bowen [84], pages 178–194.

[82] D.B. Kemp and R.W. Topor. Completeness of a top-down query evaluation procedure for stratified databases. In R.A. Kowalski and K.A. Bowen, editors, *Proceedings of the Fifth International Conference on Logic Programming*, pages 178–194. The MIT Press, 1988.

[83] S. C. Kleene. *Introduction to Metamathematics.* van Nostrand, New York, 1952.

[84] R. A. Kowalski and K. A. Bowen, editors. *Proc. of the Fifth International Conference and Symposium on Logic Programming*, Seattle, 1988. ALP, MIT Press.

[85] R.A. Kowalski. Predicate logic as a programming language. In *Proceedings IFIP'74*, pages 569–574. North-Holland, 1974.

[86] S. Kraus, D. Lehmann, and M. Magidor. Nonmonotonic reasoning, preferential models and cumulative logics. *Artificial Intelligence*, 44(1):167–207, 1990.

[87] K. Kunen. Negation in logic programming. *J. of Logic Programming*, 4:289–308, 1987.

[88] K. Kunen. Signed data depedencies in logic programs. *Journal of Logic Programming*, 7:231–246, 1989.

[89] G. Levi and M. Martelli, editors. *Proc. of the Sixth International Conference on Logic Programming,* Lisbon, Lisbon, Portugal, 1989. ALP, MIT Press.

[90] L. Li and J.H. You. Making default inferences from logic programs. *J. of Computational Intelligence*, 7:142–153, 1991.

[91] V. Lifschitz. On the declarative semantics of logic programs with negation. In Minker [109], pages 177–192.

[92] J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, Berlin, 1984.

[93] J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, Berlin, second edition, 1987.

[94] J. W. Lloyd and R. W. Topor. Making PROLOG more expressive. *Journal of Logic Programming*, 1:225–240, 1984.

[95] J. W. Lloyd and R. W. Topor. A Basis for Deductive Database Systems II. *Journal of Logic Programming*, 3(1):55–67, 1986.

[96] J.W. Lloyd, E.A. Sonenberg, and R.W. Topor. Integrity constraint checking in stratified databases. *Journal of Logic Programming*, 4(4):331–345, 1987.

[97] J. Lobo, J. Minker, and A. Rajasekar. Weak completion theory for Non-Horn programs. In Kowalski and Bowen [84], pages 828–842.

[98] J. Lobo, J. Minker, and A. Rajasekar. *Foundations of Disjunctive Logic Programming.* MIT Press, 1992.

[99] S. Lüttringhaus-Kappel. *Laziness in Logic Programming.* PhD thesis, Universität Bonn, 1992.

[100] D. Makinson. General Patterns in Nonmonotonic Reasoning. In D.M. Gabbay, C.J. Hogger, and J.A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming Vol. 2, Nonmonotonic and Uncertain Reasoning*, chapter 2.2. Oxford University Press, 1993.

[101] P. Mancarella. *Intensional Negation of Logic Programs* (in Italian). PhD thesis, Università di Pisa, 1988.

[102] P. Mancarella and D. Pedreschi. An algebra of logic programs. In Kowalski and Bowen [84], pages 1006–1023.

[103] E. Marchiori. Proving run-time properties of general logic programs w.r.t. constructive negation. Res. Report CS-R9245, CWI, Amsterdam, 1992.

[104] V.W. Marek and M. Truszczyński. *Nonmonotonic logics; context-dependent reasoning.* Berlin: Springer-Verlag, 1993.

[105] W. Marek and M. Truszczyński. Stable semantics for logic programs and default theories. In E.Lusk and R. Overbeek, editors, *Proceedings of the North American conference on logic programming*, pages 243–256, Cambridge, MA., 1989. MIT Press.

[106] M. Martelli and C. Tricomi. A new SLDNF-tree. *Information Processing Letters*, 43(2):57–62, 1992.

[107] J. McCarthy. Circumscription – a form of non-monotonic reasoning. *Artificial Intelligence*, 13:27–39, 1980.

[108] J. Minker. On indefinite data bases and the closed world assumption. In *Proc. of the Sixth Conference on Automated Deduction,* LNCS 138, pages 292–308. Springer Verlag, New York, 1982.

[109] J. Minker, editor. *Foundations of Deductive Databases and Logic Programming.* Morgan Kaufmann, Los Altos, CA, 1988.

[110] J. Minker and C. Ruiz. Semantics for Disjunctive Logic Programs with Explicit and Default Negation. *Fundamenta Informaticae*, forthcoming, 1994.

[111] R. Moore. Semantical considerations on non-monotonic logic. *Artificial Intelligence*, 25(1):75–94, 1985.

[112] M. Müller and J. Dix. Implementing Semantics for Disjunctive Logic Programs Using Fringes and Abstract Properties. In Luis Moniz Pereira and Anil Nerode, editors, *Logic Programming and Non-Monotonic Reasoning, Proceedings of the Second International Workshop*, pages 43–59. Lisbon, MIT Press, July 1993.

[113] L. Naish. An Introduction to MU-PROLOG. Technical Report TR 82/2, Dept. of Computer Science, Univ. of Melbourne, 1982.

[114] L. Naish. Negation and Quantifiers in NU-Prolog. In *Third International Conference on Logic Programming*, pages 624–634, London, July 1986.

[115] S. Naqvi and R. Krishnamurthy. Database updates in logic programming. In *Proc. of the Seventh ACM Symposium on Principles of Database Systems*, 1988.

[116] A. Nerode, W. Marek, and V.S. Subrahmanian, editors. *Proceedings of the First International Workshop on Logic Programming and Non-monotonic Reasoning*, Washington, D.C., July 1991. MIT Press.

[117] L.M. Pereira, J.N. Aparício, and J.J. Alferes. Derivation procedures for extended stable models. In *Proc. of 12th International Conference on Artificial Intelligence*, pages 863–868. Morgan Kaufmann, 1991.

[118] L.M. Pereira, J.N. Aparício, and J.J. Alferes. Adding Closed World Assumptions to Well Founded Semantics. In ICOT, editor, *Proc. of the International Conference on Fifth Generation Computer Systems 92*, June 1992.

[119] A. Di Pierro, M. Martelli, and C. Palamidessi. Negation as instantiation. Technical report, Dipartimento di Informatica e Scienze dell'Informazione, Università di Genova, 1993.

[120] H. Przymusinska and T. C. Przymusinski. Semantic issues in deductive databases and logic programs. In R. Banerji, editor, *Formal Techniques in Artificial Intelligence*, pages 321–367. North-Holland, Amsterdam, 1990.

[121] H. Przymusinska and T. C. Przymusinski. Stationary extensions of default theories. In *Proc. of the Fourth Workshop on Non-Monotonic Reasoning*, Plymouth, Vermont, 1992. To appear in *Fundamenta Informaticae*.

[122] H. Przymusinska, T. C. Przymusinski, and H. Seki. Soundness and completeness of partial deductions for well-founded semantics. In A. Voronkov, editor, *Proc. of the International Conference on Automated Reasoning,* LNAI 624, St. Petersburg, Russia, July 1992.

[123] H. Przymusinska and T.C. Przymusinski. Weakly perfect model semantics for logic programs. In Kowalski and Bowen [84], pages 1106–1120.

[124] H. Przymusinska and T.C. Przymusinski. Weakly stratified logic programs. *Fundamenta Informaticae*, 13:51–65, 1990.

[125] T. C. Przymusinski. On the declarative and procedural semantics of logic programs. *J. of Automated Reasoning*, 5:167–205, 1989.

[126] T. C. Przymusinski. Stationary semantics for normal and disjunctive logic programs. In C. Delobel, M. Kifer, and Y. Masunaga, editors, *DOOD'91, Proc. of the Second International Conference,* LNCS 566, München, December 1991. Springer.

[127] T. C. Przymusinski. Well-founded completions of logic programs. In Furukawa [64], pages 726–741.

[128] T.C. Przymusinski. On the declarative semantics of logic programs with negation. In Minker [109], pages 193–216.

[129] T.C. Przymusinski. On the relationship between logic programming and non-monotonic reasoning. In *Proc of AAAI-88,* St. Paul, pages 444–448, 1988.

[130] T.C. Przymusinski. Every logic program has a natural stratification and an iterated fixed point model. In *Proc. of the 8th Symposium on Principles of Database Systems*, pages 11–21. ACM SIGACT-SIGMOD, 1989.

[131] T.C. Przymusinski. Non-monotonic formalisms and logic programming. In Levi and Martelli [89], pages 655–674.

[132] T.C. Przymusinski. Three-valued Formalizations of Non-Monotonic Reasoning and Logic Programming. In R. Brachman, H. Leveque, and R. Reiter, editors, *Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning*, pages 341–348, Toronto, Canada, 1989. Morgan Kaufmann.

[133] T.C. Przymusinski. Extended stable semantics for normal and disjunctive programs. In Warren and Szeredi [173], pages 459–477.

[134] T.C. Przymusinski. Non-monotonic reasoning vs. logic programming: A new perspective. In D. Partridge and Y. Wilks, editors, *The Foundations of Artificial Intelligence. A Sourcebook*, pages 49–71. Cambridge University Press, London, 1990. Extended abstract appeared as [129].

[135] T.C. Przymusinski. Stationary semantics for disjunctive logic programs and deductive databases. In S. Debray and M. Hermenegildo, editors, *Proceedings of the 1990 North American Conference on Logic Programming*, pages 40–59, Austin, TX, 1990. ALP, MIT Press.

[136] T.C. Przymusinski. The Well-founded Semantics Coincides with the Three-valued Stable Semantics. *Fundamenta Informaticae*, 13(4):445–463, 1990.

[137] T.C. Przymusinski. Auto-epistemic logics of closed beliefs and logic programming. In Nerode et al. [116], pages 3–20.

[138] T.C. Przymusinski. Stable Semantics for Disjunctive Programs. *New Generation Computing*, 9:401–424, 1991. Extended abstract appeared as [133].

[139] T.C. Przymusinski. Three-Valued Non-Monotonic Formalisms and Semantics of Logic Programs. *Artificial Intelligence*, 49:401–424, 1991. Extended abstract appeared as [132].

[140] T.C. Przymusinski and D.S. Warren. *Well-Founded Semantics: Theory and Implementation.* Draft, 1992.

[141] A. Rajasekar, J. Lobo, and J. Minker. Weak generalized closed world assumption. *J. of Automated Reasoning*, 5:293–307, 1989.

[142] A. Rajasekar and J. Minker. A Stratification Semantics for General Disjunctive Programs. In E.L. Lusk and R.A. Overbeek, editors, *Proceedings of the North American Conference on Logic Programming*, pages 573–586, Cleveland, Ohio, USA, 1989.

[143] M. Reinfrank. *Fundamentals and Logical Foundations of Truth Maintenance.* PhD thesis, Linköping University, 1989. ISBN 91-7870-546-0.

[144] R. Reiter. On Closed World Data Bases. In H. Gallaire and J. Minker, editors, *Logic and Databases*, pages 55–76. Plenum, 1978.

[145] R. Reiter. A logic for default theory. *Artificial Intelligence*, 13:81–132, 1980.

[146] K. Ross. The well-founded semantics for disjunctive logic programs. In *Proc. of the First International Conference on Deductive and Object Oriented Databases*, pages 352–369, Kyoto, Japan, December 1989.

[147] K. Ross. A procedural semantics for well-founded negation in logic programs. *J. of Logic Programming*, 13(1):1–22, 1992.

[148] K. Ross and R.A. Topor. Inferring negative information from disjunctive databases. *J. of Automated Reasoning*, 4:397–424, 1988.

[149] K.A. Ross. Modular acyclicity and tail recursion in logic programs. In *Proceedings of the Tenth ACM Symposium on Principles of Database Systems*, 1991.

[150] 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.

[151] F. Sadri and R. Kowalski. A Theorem-Proving Approach to Database Integrity. In Minker [109], pages 313–362.

[152] C. Sakama and K. Inoue. Negation in Disjunctive Logic Programs. In D. Warren and Peter Szeredi, editors, *Proceedings of the 10th Int. Conf. on Logic Programming, Budapest*, pages 703–719. MIT, July 1993.

[153] T. Sato. Completed logic programs and their consistency. *Journal of Logic Programming*, 9(1):33–44, 1990.

[154] John S. Schlipf. The Expressive Powers of the Logic Programming Semantics. In *Proceedings of the Ninth ACM Symposium on Principles of Databases*, pages 196–204, 1990.

[155] H. Seki and H. Itoh. A query evaluation method for stratified programs under the extended CWA. In Kowalski and Bowen [84], pages 195–211.

[156] J. C. Shepherdson. A sound and complete semantics for a version of negation as failure. *Theoretical Computer Science*, 65(3):343–371, 1989.

[157] J. C. Shepherdson. Correct answers to allowed queries are ground. *Journal of Logic Programming*, 11(3 & 4):359–362, 1991.

[158] J.C. Shepherdson. Negation in logic programming for general logic programs. In Minker [109], pages 19–88.

[159] J.C. Shepherdson. Negation as failure, completion and stratification. draft; to appear in Handbook of Artificial Intelligence and Logic Programming, 1990.

[160] J.C. Shepherdson. SLDNF resolution with equality. *J. of Automated Reasoning*, 8(2):297–306, 1992.

[161] R. Stärk. A complete axiomatization of the three-valued completion of logic programs. *Journal of Logic and Computation*, 1(6):811–834, 1991.

[162] R. Stärk. *The Proof Theory of Logic Programs with Negation.* PhD thesis, University of Berne, 1992.

[163] R. Stärk. The declarative semantics of the Prolog selection rule. Technical report, CIS, Universität München, 1993.

[164] R. Stärk. From logic programs to inductive definitions. Technical report, CIS, Universität München, 1993.

[165] R. Stärk. Input/output dependencies of normal logic programs. *Journal of Logic and Computation*, 1993. To appear.

[166] K. Stroetman. A completeness result for SLDNF resolution. *The Journal of Logic Programming*, 15:337–357, 1993.

[167] P.J. Stuckey. Constructive negation for constraint logic programming. In *Proceedings of the 6th Annual Symposium on Logic in Computer Science (LICS)*, pages 328–339, Amsterdam, The Netherlands, 1991.

[168] F. Teusink. A characterization of stable models using a non-monotonic operator. In L. M. Pereira and A. Nerode, editors, *Proceedings of the 1993 workshop on Logic Programming and Non -Monotonic Reasoning*, pages 206–222, 1993.

[169] M. H. van Emden and R. A. Kowalski. The semantics of predicate logic as a programming language. *Journal of the ACM*, 23(4):733–742, 1976.

[170] A. van Gelder. Negation as Failure Using Tight Derivations for General Logic Programs. In Minker [109], pages 149–176.

[171] G. Wagner. *Vivid Logic – Knowledge-Based Reasoning with Two Kinds of Negation.* PhD thesis, Freie Universität Berlin, 1993.

[172] M. Wallace. Tight, consistent, and computable completions for unrestricted logic programs. *Journal of Logic Programming*, 15:243–273, 1993.

[173] D.H.D. Warren and P. Szeredi, editors. *Proceedings of the Seventh International Conference on Logic Programming*, Jerusalem, 1990. MIT Press.

[174] C. Witteveen. Partial semantics for truth maintenance. In Eijck [54], pages 544–561.