ECAI 96

BUDAPEST

12th European Conference on Artificial Intelligence

T10

Budapest University of Economic Sciences
HUNGARY
11-16 August 1996

Organized by the
European Coordinating Committee for Artificial Intelligence (ECCAI)
Hosted by the
John von Neumann Computer Society (NJSZT)

ITA/322

T10

Presenting Knowledge with Logic Programming Extensions

ECAI'96

Budapest, Hungary

Knowledge Representation with Logic Programs

Gerhard Brewka
TU Wien,
Abteilung für wissensbasierte Systeme,
Treitlstr. 3, 1040 Wien, Austria

Jürgen Dix
Universität Koblenz-Landau,
Institut für Informatik,
Rheinau 1, 56075 Koblenz, Germany

June 10, 1996

Abstract

In this tutorial-overview we show how Knowledge Representation (KR) can be done with the help of *generalized* logic programs. We start by introducing the core of PROLOG, which is based on *definite* logic programs. Although this class is very restricted (and will be enriched by various additional features in the rest of the paper), it has a very nice property for KR-tasks: there exist efficient *Query-answering procedures* — a *Top-Down* approach and a *Bottom-Up* evaluation. In addition we can not only handle ground queries but also queries with variables and compute *answer-substitutions*.

It turns out that more advanced KR-tasks can not be properly handled with definite programs. Therefore we extend this basic class of programs by additional features like *Negation-as-Finite-Failure*, *Default-Negation*, *Explicit Negation*, *Preferences*, and *Disjunction*. The need for these extensions is motivated by suitable examples and the corresponding semantics are discussed in detail.

Clearly, the more expressive the respective class of programs under a certain semantics is, the less efficient are potential Query-answering methods. This point will be illustrated and discussed for every extension. By well-known recursion-theoretic results, it is obvious that there do not exist complete Query-answering procedures for the general case where variables and function symbols are allowed. Nevertheless we consider it an important topic of further research to extract *feasible* classes of programs where answer-substitutions can be computed.

Contents

List of Figures

List of Tables

1 Introduction

One of the major reasons for the success story (if one is really willing to call it a success story) of human beings on this planet is our ability to invent tools that help us improve our — otherwise often quite limited — capabilities. The invention of machines that are able to do interesting things, like transporting people from one place to the other (even through the air), sending moving pictures and sounds around the globe, bringing our email to the right person, and the like, is one of the cornerstones of our culture and determines to a great degree our everyday life.

Among the most challenging tools one can think of are machines that are able to handle knowledge adequately. Wouldn't it be great if, instead of the stupid device which brings coffee from the kitchen to your office every day at 9.00, and which needs complete reengineering whenever your coffee preferences change, you could (for the same price, admitted) get a smart robot whom you can simply tell that you want your coffee black this morning, and that you need an extra Aspirin since it was your colleague's birthday yesterday? To react in the right way to your needs such a robot would have to know a lot, for instance that Aspirin should come with a glass of water, or that people in certain situations need their coffee extra strong.

Building smart machines of this kind is at the heart of Artificial Intelligence (AI). Since such machines will need tremendous amounts of knowledge to work properly, even in very limited environments, the investigation of techniques for representing knowledge and reasoning is highly important.

In the early days of AI it was still believed that modeling general purpose problem solving capabilites, as in Newell and Simon's famous GPS (General Problem Solver) program, would be sufficient to generate intelligent behaviour. This hypothesis, however, turned out to be overly optimistic. At the end of the sixties people realized that an approach using available knowledge about narrow domains was much more fruitful. This led to the expert systems boom which produced many useful application systems, expert system building tools, and expert system companies. Many of the systems are still in use and save companies millions of dollars per year[1].

Nevertheless, the simple knowledge representation and reasoning methods underlying the early expert systems soon turned out to be insufficient. Most of the systems were built based on simple rule languages, often enhanced with ad hoc approaches to model uncertainty. It became apparent that more advanced methods to handle incompleteness, defeasible reasoning, uncertainty, causality and the like were needed.

This insight led to a tremendous increase of research on the foundations of knowledge representation and reasoning. Theoretical research in this area

[1]We refer the interested reader to the recent book [RN95] which gives a very detailed and nice exposition of what has been done in AI since its very beginning until today.

has blossomed in recent years. Many advances have been made and important results were obtained. The technical quality of this work is often impressive.

On the other hand, most of these advanced techniques have had surprisingly little influence on practical applications so far. To a certain degree this is understandable since theoretical foundations had to be laid first and pioneering work was needed. However, if we do not want research in knowledge representation to remain a theoreticians' game more emphasis on computability and applicability seems to be needed. We strongly believe that the kind of research presented in this tutorial, that is research aiming at interesting combinations of ideas from logic programming and nonmonotonic reasoning, provides an important step into this direction.

1.1 Some History

Historically, logic programs have been considered in the logic programming community for more than 20 years. It began with [CKPR73, Kow74, vEK76] and led to the definition and implementation of *PROLOG*, a by now theoretically well-understood programming language (at least the declarative part consisting of Horn-clauses: *pure PROLOG*). Extensions of PROLOG allowing negative literals have been also considered in this area: they rely on the idea of *negation-as-finite-failure*, we call them *Logic-Programming-semantics* (or shortly *LP-semantics*).

In parallel, starting at about 1980, *Nonmonotonic Reasoning* entered into computer science and began to constitute a new field of active research. It was originally initiated because *Knowledge Representation* and *Common-Sense Reasoning* using classical logic came to its limits. Formalisms like classical logic are inherently monotonic and they seem to be too weak and therefore inadequate for such reasoning problems.

In recent years, independently of the research in logic programming, people interested in knowledge representation and nonmonotonic reasoning also tried to define declarative semantics for programs containing *default* or *explicit* negation and even *disjunctions*. They defined various semantics by appealing to (different) intuitions they had about programs.

This second line of research started in 1986 with the *Workshop on the Foundations of Deductive Databases and logic programming* organized by Jack Minker: the revised papers of the proceedings were published in [Min88]. The *stratified* (or the similar *perfect*) semantics presented there can be seen as a splitting-point: it is still of interest for the logic programming community (see [CL89]) but its underlying intuitions were inspired by nonmonotonic reasoning and therefore much more suitable for knowledge representation tasks. Semantics of this kind leave the philosophy underlying classical logic programming in that their primary aim is not to model *negation-as-finite-failure*, but to construct new, more powerful semantics suitable for applications in knowledge representation. Let us call such semantics *NMR-semantics*.

Nowadays, due to the work of Apt, Blair and Walker, Fitting, Lifschitz, Przymusinski and others, very close relationships between these two independent research lines became evident. Methods from logic programming, e.g. least fixpoints of certain operators, can be used successfully to define NMR-semantics.

The NMR-semantics also shed new light on the understanding of the classical nonmonotonic logics such as *Default Logic*, *Autoepistemic Logic* and the various versions of *Circumscription*. In addition, the investigation of possible semantics for logic programs seems to be useful because

1. parts of nonmonotonic systems (which are usually defined for full predicate logic, or even contain additional (modal)-operators) may be "implemented" with the help of such programs,

2. nonmonotonicity in these logics may be described with an appropriate treatment of negation in logic programs.

1.2 Non-Monotonic Formalisms in KR

As already mentioned above, research in nonmonotonic reasoning has begun at the end of the seventies. One of the major motivations came from reasoning about actions and events. John McCarthy and Patrick Hayes had proposed their situation calculus as a means of representing changing environments in logic. The basic idea is to use an extra situation argument for each fact which describes the situation in which the fact holds. Situations, basically, are the results of performing sequences of actions. It soon turned out that the problem was not so much to represent what changes but *to represent what does not change* when an event occurs. This is the so-called *frame problem*. The idea was to handle the frame problem by using a default rule of the form

If a property P holds in situation S then P typically also holds in the situation obtained by performing action A in S.

Given such a rule it is only necessary to explicitly describe the changes induced by a particular action. All non-changes, for instance that the colour of the kitchen wall does not change when the light is turned on, are handled implicitly. Although it turned out that a straightforward formulation of this rule in some of the most popular nonmonotonic formalisms may lead to unintended results the frame problem was certainly the challenge motivating many people to join the field.

In the meantime a large number of different nonmonotonic logics have been proposed. We can distinguish four major types of such logics:

1. Logics using nonstandard inference rules with an additional consistency check to represent default rules. Reiter's default logic (see Appendix A.3) and its variants are of this type.

2. Nonmonotonic modal logics using a modal operator to represent consistency or (dis-) belief. These logics are nonmonotonic since conclusions may depend on disbelief. The most prominent example is Moore's autoepistemic logic.

3. Circumscription (see Appendix A.4) and its variants. These approaches are based on a preference relation on models. A formula is a consequence iff it is true in all most preferred models of the premises. Syntactically, a second order formula is used to eliminate all non-preferred models.

4. Conditional approaches which use a non truth-functional connective $\mid\!\sim$ to represent defaults. A particularly interesting way of using such conditionals was proposed by Kraus, Lehmann and Magidor. They consider p as a default consequence of q iff the conditional $q \mid\!\sim p$ is in the closure of a given conditional knowledge base under a collection of rules. Each of the rules directly corresponds to a desirable property of a nonmonotonic inference relation.

The various logics are intended to handle different intuitions about nonmonotonic reasoning in a most general way. On the other hand, the generality leads to problems, at least from the point of view of implementations and applications. In the first order case the approaches are not even semi-decidable since an implicit consistency check is needed. In the propositional case we still have tremendous complexity problems. For instance, the complexity of determining whether a formula is contained in all extensions of a propositional default theory is on the second level of the polynomial hierarchy. As mentioned earlier we believe that logic programming techniques can help to overcome these difficulties.

Originally, nonmonotonic reasoning was intended to provide us with a *fast* but *unsound* approximation of classical reasoning in the presence of incomplete knowledge. Therefore one might ask whether the higher complexity of NMR-formalisms (compared to classical logic) is not a real drawback of this aim? The answer is that NMR-systems allow us to formulate a problem in a very *compact* way as a theory T. It turns out that any equivalent formulation in classical logic (if possible at all) as a theory T' is much larger: the size of T' is exponential in the size of T! We refer to [GPSK95] and [CDS95a, CDS95b, CDLS95] where such problems are investigated.

1.3 How this Paper is organized

In this tutorial paper we show how Knowledge Representation can be done with the help of *generalized* logic programs. We start by introducing the core of PROLOG, which is based on *definite* logic programs. Although this class is very restricted (and will be enriched by various additional features in the rest of

the paper), it has a very nice property for KR-tasks: there exist efficient *Query-answering procedures*— a *Top-Down* approach and a *Bottom-Up* evaluation. In addition we can not only handle ground queries but also queries with variables and compute *answer-substitutions*.

It turns out that more advanced KR-tasks can not be properly handled with definite programs. Therefore we extend this basic class of programs by additional features like *Negation-as-Finite-Failure*, *Default-Negation*, *Explicit Negation*, *Preferences*, and *Disjunction*. The need for these extensions is motivated by suitable examples and the corresponding semantics are also discussed.

Clearly, the more expressive the respective class of programs under a certain semantics is, the less efficient are potential Query-answering methods. This point will be illustrated and discussed for every extension. By well-known recursion-theoretic results, it is obvious that there do not exist complete Query-answering procedures for the general case where variables and function symbols are allowed. Nevertheless we consider it an important topic of further research to extract *feasible* classes of programs where answer-substitutions can be computed.

2 Definite Logic Programs

In this section we consider the most restricted class of programs: *definite* logic programs, programs without any negation at all. All the extensions of this basic class we will introduce later contain at least some kind of negation (and perhaps additional features). But here we also allow the ocurrence of free variables as well as function symbols.

In Section 2.1 we introduce as a representative for the *Top-Down* approach the SLD-Resolution. Section 2.2 presents the main competing approach of SLD: *Bottom-Up Evaluation*. This approach is used in the Database community and it is efficient when additional assumptions are made (*finiteness-assumptions*, *no function symbols*). In Section 2.3 we consider the influence and appropriateness of Herbrand models and their underlying intuition. Finally in Section 2.4 we present and discuss two important examples in KR: *Reasoning in Inheritance Hierarchies* and *Reasoning about Actions*. Both examples clearly motivate the need of extending definite programs by a kind of *default-negation* "*not* ".

First some notation used throughout this paper. A language $\mathcal{L}$ consists of a set of relation symbols and a set of function symbols (each symbol has an associated arity). Nullary functions are called constants. Terms and atoms are built from $\mathcal{L}$ in the usual way starting with variables, applying function symbols and relation-symbols.

Instead of considering arbitrary $\mathcal{L}$-formulae, our main object of interest is a program:

Definition 2.1 (Definite Logic Program)
A definite *logic program consists of a finite number of* rules *of the form*

$$A \leftarrow B_1, \ldots, B_m,$$

where $A, B_1, \ldots, B_m$ are positive atoms (containing possibly free variables). We call A the head *of the rule and $B_1, \ldots, B_m$ its* body.

We can think of a program as formalizing our knowledge about the world and how the world behaves. Of course, we also want to derive new information, i.e. we want to ask queries:

Definition 2.2 (Query)
Given a definite program we usually have a definite query in mind that we want to be solved. A definite query Q is a conjunction of positive atoms $C_1 \wedge \ldots \wedge C_l$ which we denote by

$$?\text{-}\ C_1, \ldots, C_l.$$

These C_i may also contain variables. Asking a query Q to a program P means asking for all possible substitutions Θ of the variables in Q such that $Q\Theta$ follows from P. Often, Θ is also called an answer to Q. Note that $Q\Theta$ may still contain free variables.

Note that if a program P is given, we usually assume that it also determines the underlying language $\mathcal{L}$, denoted by $\mathcal{L}_P$, which is generated by exactly the symbols ocurring in P. The set of all these atoms is called the *Herbrand base* and denoted by $B_{\mathcal{L}_P}$ or simply B_P. The corresponding set of all ground terms is the *Herbrand universe*. Another important notion that we are not explaining in detail here is that of *Unification*. Given two atoms A and B with free variables we can ask if we can compute two substitutions Θ_1, Θ_2 for the variables such that

$$A\Theta_1 \text{ is identical to } B\Theta_2,$$

or if we can decide that this is not possible at all. In fact, if the two atoms are *unifiable* we can indeed compute a *most general unifier*, called mgU (see [Llo87]). This will be important in our framework because if an atom appears as a subgoal in a query, we may want to determine if there are rules in the program whose heads unify with this atom.

How are our programs related to classical predicate logic? Of course, we can map a program-rule into classical logic by interpreting "$\leftarrow$" as material implication "$\supset$" and universally quantifying. This means we view such a rule as the following universally quantified formula

$$B_1 \wedge \ldots \wedge B_m \supset A.$$

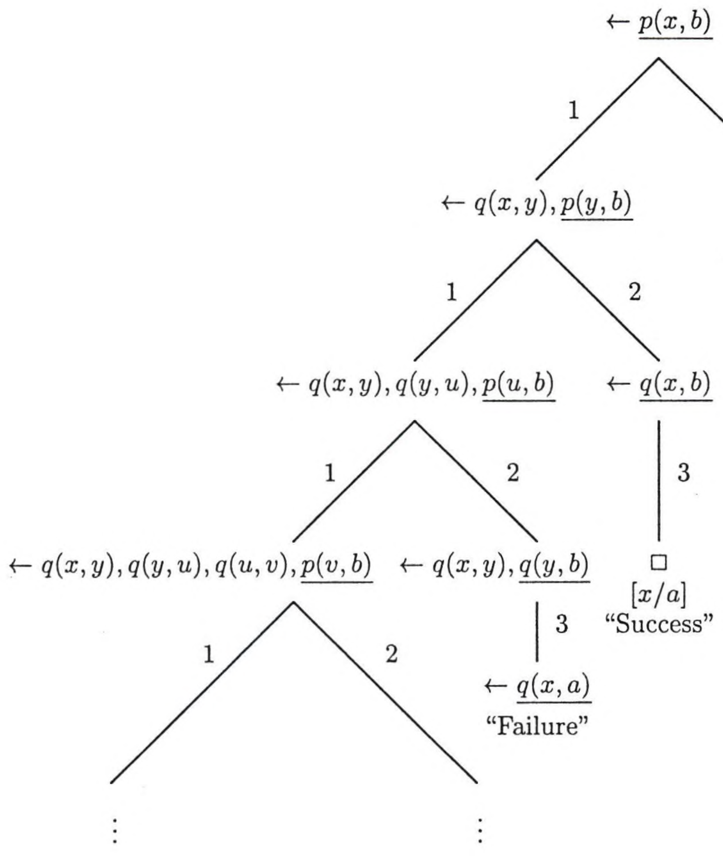
However, as we will see later, there is a great difference: a logic program-rule takes some orientation with it. This makes it possible to formulate the following principle as an underlying intuition of all semantics of logic programs:

Principle 2.3 (Orientation)
If a ground atom A does not unify with some head of a program rule of P, then this atom is considered to be false. In this case we say that "not A" is derivable from P to distinguish it from classical $\neg A$.

2.1 Top-Down

SLD-Resolution[2] is a special form of Robinson's general Resolution rule. While Robinson's rule is complete for full first order logic, SLD is complete for definite logic programs (see Theorem 2.5). We do not give a complete definition of SLD-Resolution (see [Llo87]) but rather prefer to illustrate its behaviour on the following example.

[2]SL-resolution for Definite clauses. SL-resolution stands for Linear resolution with Selection function.



$\leftarrow \underline{p(x,b)}$
1
2
$\leftarrow q(x,y), \underline{p(y,b)}$
$\Box$
$[x/b]$
"Success"
1
2
$\leftarrow q(x,y), q(y,u), \underline{p(u,b)}$
$\leftarrow \underline{q(x,b)}$
1
2
3
$\leftarrow q(x,y), q(y,u), q(u,v), \underline{p(v,b)}$
$\leftarrow q(x,y), \underline{q(y,b)}$
$\Box$
$[x/a]$
"Success"
3
1
2
$\leftarrow \underline{q(x,a)}$
"Failure"

Figure 1: An Infinite SLD-Tree

Example 2.4 (SLD-Resolution)
Let the program P_{SLD} consist of the following three clauses

(1)	$p(x,z)$	$\leftarrow$	$q(x,y), p(y,z)$
(2)	$p(x,x)$		
(3)	$q(a,b)$		

The query Q we are interested in is given by $p(x,b)$. I.e. we are looking for all substitutions Θ for x such that $p(x,b)\Theta$ follows from P.

Figure 1 illustrates the behaviour of SLD-resolution. We start with our query in the form $\leftarrow Q$. Sometimes the notation $\Box \leftarrow Q$ is also used, where $\Box$ denotes the falsum. In any round the selected atom is underlined: numbers 1, 2 or 3 indicate the number of the clause which the selected atom is resolved

against. Obviously, there are three different sorts of branches, namely

1. *infinite* branches,
2. branches that *end up with the empty clause*, and
3. branches that *end in a deadlock ("Failure")*: no applicable rule is left.

In this example we always resolve with the *last* atom in the goal under consideration. If we choose always the *first* atom in the goal, we will obtain, at least in this example, a *finite* tree.

Definite programs have the nice feature that the intersection of all Herbrand-models exists and is again a Herbrand model of P. It is denoted by M_P and called the least Herbrand-model of P. Note that our original aim was to find substitutions Θ such that $Q\Theta$ is derivable from the program P. This task as well as M_P is closely related to SLD:

Theorem 2.5 (Soundness and Completeness of SLD)
The following properties are equivalent:

- $P \models \forall\, Q\Theta$, *i.e.* $\forall\, Q\Theta$ *is true in all models of* P,
- $M_P \models \forall\, Q\Theta$,
- *SLD computes an answer* τ *that subsumes*[3] Θ *wrt* Q.

Note that not any correct answer is computed, only the most general one is (which of course subsumes all the correct ones).

The main feature of SLD-Resolution is its *Goal-Orientedness*. SLD automatically ensures (because it starts with the Query) that we consider only those rules that are relevant for the query to be answered. Rules that are not at all related are simply not considered in the course of the proof.

2.2 Bottom-Up

We mentioned in the last section the least Herbrand model M_P. The bottom-up approach can be described as computing this least Herbrand model from below. We start first with rules with empty bodies (in our example these are all instantiations of rules (2) and (3)). We get as facts all atoms that are in the heads of rules with empty bodies (namely $p(a,a), p(b,b), q(a,b)$ in Example 2.4). In the next round we use the facts that we computed before and try to let the rules "fire", i.e. when their bodies are true, we add their heads to the atoms we already have (this gives us $p(a,b)$).

To be more precise we introduce the immediate consequence operator T_P which associates to any Herbrand model another Herbrand model.

[3] i.e. $\exists\sigma : Q\tau\sigma = Q\Theta$.

Example 2.6 **(T_P)**
Given a definite program P let $T_P : \; 2^{B_P} \longmapsto 2^{B_P}; \; \mathcal{I} \longmapsto T_P(\mathcal{I})$

$$T_P(\mathcal{I}) := \{A \in B_P : \text{ there is an instantiation of a rule in } P \text{ s.t. } A \text{ is the head of this rule and all body-atoms are contained in } \mathcal{I} \; \}$$

It turns out that T_P is monotone and continuous so that (by a general theorem of Knaster-Tarski) the least fixpoint is obtained after ω steps. Moreover we have

Theorem 2.7 **(T_P and M_P)**
$M_P = T_P{\uparrow}^{\omega} = lfp(T_P)$.

This approach is especially important in Database applications, where the underlying language does not contain function symbols (DATALOG) — this ensures the Herbrand universe to be finite. Under this condition the iteration stops after finitely many steps. In addition, rules of the form

$$p \leftarrow p$$

do not make any problems. They simply can not be applied or do not produce anything new. Note that in the Top-Down approach, such rules give rise to infinite branches! Later, elimination of such rules will turn out to be an interesting property. We therefore formulate it as a principle:

Principle 2.8 (Elimination of Tautologies)
Suppose a program P has a rule which contains the same atom in its body as well as in its head (i.e. the head consists of exactly this atom). Then we can eliminate this rule without changing the semantics.

Unfortunately, such a bottom-up approach has two serious shortcomings. First, the goal-orientedness from SLD-resolution is lost: we are always computing the whole M_P, even those facts that have nothing to do with the query. The reason is that in computing T_P we do not take into account the query we are really interested in. Second, in any step facts that are already computed before are recomputed again. It would be more efficient if only new facts were computed. Both problems can be (partially) solved by appropriate refinements of the naive approach:

- *Semi-naive* bottom-up evaluation ([Bry90, Ull89b]),
- *Magic Sets* techniques ([BR91, Ull89a]).

2.3 Herbrand-Models and the underlying language

Usually when we represent some knowledge in first order logic or even in logic programs, it is understood that the underlying language is given exactly by the

symbols that occur in the formal theory. Suppose we have represented some knowledge about the world as a theory T in a language $\mathcal{L}$. Classical predicate logic formalizes the notion of a formula ϕ *derivable* from the theory T. This means that ϕ is true in all models of T (we denote this set by MOD(T)). Why are we considering all models? Doesn't it make sense to look only at Herbrand models, i.e. to models generated by the underlying language? After all we are not interested in models that contain elements which are not representable as terms in our language. These requirements are usually called *Unique Names Assumption* and *Domain Closure Assumption*:

Definition 2.9 (UNA and DCA)
Let a language $\mathcal{L}$ be given. We understand by the Unique Names Assumption *the restriction to those models $\mathcal{I}$, where syntactically different ground $\mathcal{L}$-terms t_1, t_2 are interpreted as nonidentical elements: $t_1^{\mathcal{I}}$ is not identical to $t_2^{\mathcal{I}}$.*

By the Domain Closure Assumption *we mean the restriction to those models $\mathcal{I}$ where for any element a in $\mathcal{I}$ there is a $\mathcal{L}$-term t that represents this element: $a = t^{\mathcal{I}}$.*

As an example, in Theorem 2.5 of Section 2.1 we referred to M_P, the least Herbrand model of P. The reason that the first equivalence in this theorem holds is given by the fact that for universal theories T and existential formulae ϕ the following holds

$$\text{MOD}(T) \models \phi \quad \text{iff} \quad \text{Herb}_{\mathcal{L}}\text{-MOD}(T) \models \phi.$$

In our particular case, where T is a definite program P, we can even replace Herb$_{\mathcal{L}}$-MOD(T) in the above equation by the single model M_P.

This last result does not hold in general. But what happens if we nevertheless are interested in only the Herbrand-models of a theory T (and therefore automatically[4] assume UNA and DCA)? At first sight one can argue that such an approach is much simpler: in contrast to *all* models we only need to take care about the very specific Herbrand models. But it turns out that determining the truth of a formula in all Herbrand models is a much more complex task (namely Π_1^1-complete) than to determine if it is true in all models. This latter task is also undecidable in general, but it is recursively enumerable, i.e. Π_1^0-complete. The fact that this task is recursively enumerable is the content of the famous completeness theorem of Gödel, where "truth of a formula in all models" is shown to be equivalent to deriving this formula in a particular axiomatization of the predicate calculus of first order. We refer to the appendix (Section A.1 and Section A.2) where the necessary notions are introduced.

But we have still a problem with Theorem 2.5 in our restricted setting:

[4]The only difference between Herbrand models and models satisfying UNA and DCA is that the interpretation of terms is uniquely determined in Herbrand models. It is required that a term "$f(t_1, \ldots, f_n)$" is interpreted in a Herbrand model $\mathcal{I}$ as "$f^{\mathcal{I}}(t_1^{\mathcal{I}}, \ldots, t_n^{\mathcal{I}})$".

Example 2.10 (Universal Query Problem)
Consider the program $P := p(a)$, *the query* $Q := p(x)$ *and the empty substitution* $\Theta := \epsilon$. *We have*

- $M_P \models \forall x p(x)$
- *but SLD only computes the answer* x/a.

Przymusinski called this the universal query problem.

There are essentially two solutions to avoid this behaviour: to use a language which is *rich enough* (i.e. contains sufficiently many terms, not only those ocurring in the program P itself) or to consider arbitrary models, not only Herbrand models. Both approaches have been followed in the literature but they are beyond the scope of this paper.

2.4 Why going beyond Definite Programs?

So far we have a nice query-answering procedure, SLD-Resolution, which is goal-oriented as well as sound and complete with respect to general derivability. But note that up to now we are not able to derive any *negative* information. Not even our queries allow this. From a very pragmatic viewpoint, we can consider "*not* A" to be derivable if A is not. Of course, this is not sound with respect to classical logic but it is with respect to M_P.

In KR we do not only want to formulate negative queries, we also want to express *default-statements* of the form

Normally, unless something abnormal holds, then ψ implies ϕ.

Such statements were the main motivation for nonmonotonic logics, like Default Logic or Circumscription (see Section A.3 and Section A.4 of the appendix). How can we formulate such a statement as a logic program? The most natural way is to use negation "*not* "

$$\phi \leftarrow \psi, \ not\ ab$$

where *ab* stands for *abnormality*. Obviously, this forces us to extend definite programs by negative atoms.

A typical example for such statements occurs in Inheritance Reasoning. We take the following example from [BG94]:

Example 2.11 (Inheritance Hierachies)
Suppose we know that birds typically fly and penguins are non-flying birds. We also know that Tweety is a bird. Now an agent is hired to build a cage for Tweety. Should the agent put a roof on the cage? After all it could be still the case that Tweety is a penguin and therefore can not fly, in which case we would

not like to pay for the unneccessary roof. But under normal conditions, it should be obvious that one should conclude that Tweety is flying.

A natural axiomatization is given as follows:

$$\begin{array}{llll} P_{Inheritance}: & flies(x) & \leftarrow & bird(x), \quad \text{not } ab(r_1, x) \\ & bird(x) & \leftarrow & penguin(x) \\ & ab(r_1, x) & \leftarrow & penguin(x) \\ & make\_top(x) & \leftarrow & flies(x) \end{array}$$

together with some particular facts, like e.g. bird(Tweety) and penguin(Sam). The first rule formalizes our default-knowledge, while the third formalizes that the default-rule should not be applied in abnormal or exceptional cases. In our example, it expresses the famous Specificity-Principle *which says that more specific knowledge should override more general one ([THT86]).*

For the query "make_top(Tweety)" we expect the answer "yes" while for "make_top(Sam)" we expect the answer "no".

Another important KR task is to formalize knowledge for reasoning about action. We again consider a particular important instance of such a task, namely *temporal projection.* The overall framework consists in describing the initial state of the world as well as the effects of all actions that can be performed. What we want to derive is how the world looks like after a sequence of actions has been performed.

Example 2.12 (Temporal Projection: Yale-Shooting Problem)
We distinguish between three sorts[5] *of variables:*

- *situation variables:* $S, S', \ldots,$
- *fluent variables:* $F, F', \ldots,$
- *action variables:* $A, A', \ldots.$

The initial situation is denoted by the constant s_0*, and the two-ary function symbol* $res(A, S)$ *denotes the situation that is reached when in situation* S *the action* A *has been performed. The relation symbol* $holds(F, S)$ *formalizes that the fluent* F *is true in situation* S*.*

For the YSP there are three actions (wait, load and shoot) and two fluents (alive and loaded). Initially a turkey called Fred is alive. We then load a gun, wait and shoot. The effect should be that Fred is dead after this sequence of actions. The common-sense argument from which this should follow is the

Law of Inertia: *Things normally tend to stay the same.*

[5]To be formally correct we have to use many-sorted logic. But since all this could also be coded in predicate logic by using additional relation symbols, we do not emphasize this fact. We also understand that instantiations are done in such a way that the sorts are respected.

Using our intuition from the last example, a natural formalization is given as follows:

$$
\begin{array}{lll}
P_{YSP}: & holds(F, res(A, S)) & \leftarrow \; holds(F, S), \text{ not } ab(r_1, A, F, S) \\
& holds(loaded, res(load, S)) & \leftarrow \\
& ab(r_1, shoot, alive, S) & \leftarrow \; holds(loaded, S) \\
& holds(alive, s_0) & \leftarrow
\end{array}
$$

Such a straightforward formalization leads in most versions of classical non-monotonic logic to the unexpected result, that Fred is not neccesarily dead. But obviously we expect to derive $holds(alive, res(load, s_0))$ and

$$\text{not } holds(alive\,, res(shoot, res(wait, res(load, s_0))))$$

Up to now we only have stated some very "natural" axiomatizations of given knowledge. We have motivated that something like default-negation "*not* " should be added to definite programs in order to do so and we have explicitly stated the answers to particular queries. What is still missing are solutions to the following very important problems

- *How should an appropriate query answering mechanism handling default-negation "*not *" look like?*
- *What is the formal semantics that such a procedural mechanism should be checked against?*

Such a semantics is certainly not classical predicate logic because of the default character of "*not* " — *not* is not classical $\neg$. Both problems will be considered in detail in Section 3.

2.5 What is a Semantics?

In the last subsections we have introduced two principles (*Orientation* and *Elimination of Tautologies*) and used the term *semantics of a program* in a loose, imprecise way. We end this section with a precise notion of what we understand by a semantics.

As a first attempt, we can view a semantics as a mapping that associates to any program a set of positive atoms and a set of default atoms. In the case of SLD-Resolution the positive atoms are the ground instances of all derivable atoms. But sometimes we also want to derive negative atoms (like in our two examples above). Our *Orientation*-Principle formalizes a minimal requirement for deriving such default-atoms.

Of course, we also want that a semantics SEM should *respect* the rules of P, i.e. whenever SEM makes the body of a rule true, then SEM should also make the head of the rule true. But it can (and will) happen that a semantics SEM does not always decide *all* atoms. Some atoms A are not derivable nor are their

default-counterparts *not* A. This means that a semantics SEM can view the body of a rule as being *undefined*.

This already happens in classical logic. Take the theory

$$T := \{(A \wedge B) \supset C,\ \neg A \supset B\}.$$

What are the atoms and negated atoms derivable from T, i.e. true in all models of T? No positive atom nor any negated atom is derivable! The classical semantics therefore makes the truthvalue of $A \wedge B$ undefined in a sense.

Suppose a semantics SEM treats the body of a program rule as undefined. What should we conclude about the head of this rule? We will only require that this head is not treated as false by SEM — it could be true or undefined as well. This means that we require a semantics to be compatible with the program *viewed as a 3-valued theory* — the three values being "true", "false" and "undefined". For the understanding it is not neccessary to go deeper into 3-valued logic. We simply note that we interpret "$\leftarrow$" as the Kleene-connective which is true for "$undefined \leftarrow undefined$" and false for "$false \leftarrow undefined$".

Our discussion shows that we can view a semantics SEM as a 3-valued model of a program. In classical logic, there is a different viewpoint. For a given theory T we consider there the set of all classical models MOD(T) as the semantics. The intersection of all these models is of course a 3-valued model of T, but MOD(T) contains more information. In order to formalize the notion of semantics as general as possible we define

Definition 2.13 (SEM)
A semantics SEM is a mapping from the class of all programs into the powerset of the set of all 3-valued structures. SEM assigns to every program P a set of 3-valued models of P:

$$SEM(P) \subseteq MOD^{\mathcal{L}_P}_{3-val}(P).$$

This definition covers both the classical viewpoint (classical models are 2-valued and therefore special 3-valued models) as well as our first attempt in the beginning of this subsection. Later on, in most cases we will be really interested only in Herbrand models.

Formally, we can associate to any semantics SEM in the sense of Definition 2.13 two entailment relations

sceptical: $\text{SEM}^{scept}(P)$ is the set of all atoms or default atoms that are true in *all models* of SEM(P).

credulous: $\text{SEM}^{cred}(P)$ is the set of all atoms or default atoms that are true in *at least one model* of SEM(P).

In this tutorial we only consider the sceptical viewpoint. Also, to facilitate notation, we will not formally distinguish between SEM and SEM^{scept}. In cases

where by definition SEM can only contain a single model (like in the case of well-founded semantics) we will omit the outer brackets and write

$$\mathrm{SEM}(P) = M$$

instead of $\mathrm{SEM}(P) = M$. We will also slightly abuse notation and write $l \in \mathrm{SEM}(P)$ as an abbreviation for $l \in M$ for all $M \in \mathrm{SEM}(P)$.

3 Adding Default-Negation

In the last section we have illustrated that logic programs with negation are very suitable for KR — they allow a natural and straightforward formalization of default-statements. The problem still remained to define an appropriate semantics for this class and, if possible, to find efficient query-answering methods. Both points are adressed in this section.

We can distinguish between two quite different approaches:

LP-Approach: This is the approach taken mainly in the Logic Programming community. There one tried to stick as close as possible to SLD-Resolution and treat negation as "Finite-Failure". This resulted in an extension of SLD, called SLDNF-Resolution, a procedural mechanism for query answering. For a nice overview, we refer to [AB94].

NML-Approach: This is the approach suggested by non-monotonic reasoning people. Here the main question is *"What is the right semantics?"* I.e. we are looking first for a semantics that correctly fits to our intuitions and treats the various KR-Tasks in the right (or appropriate) way. It should allow us to jump to conclusions even when only little information is available. Here it is of secondary interest how such a semantics can be implemented with a procedural calculus. Interesting overviews are [Min93] and [Dix95c].

The LP-Approach is dealt with in Section 3.1. It is still very near to classical predicate logic — default negation is interpreted as *Finite-Failure*. To get a stronger semantics, we interpret "*not* " as *Failure* in Section 3.2. The main difference is that the principle *Elimination of Tautologies* holds. We then introduce a principle GPPE which is related to partial evaluation. In KR one can see this principle as allowing for definitional extensions — names or abbreviations can be introduced without changing the semantics.

All these principles do not yet determine a unique semantics — there is still room for different semantics and a lot of them have been defined in the last years. We do not want to present the whole zoo of semantics nor to discuss their merits or shortcomings. We refer the reader to the overview articles [AB94] and [Dix95c] and the references given therein. We focus on the two main competing approaches that still have survived. These are are the Wellfounded semantics WFS (Section 3.3) and the Stable semantics STABLE (Section 3.4). Finally, in Section 3.5 we discuss complexity and expressibility results for the semantics presented so far.

3.1 Negation-as-Finite-Failure

The idea of negation treated as *finite-failure* can be best illustrated by still considering definite programs, but queries containing default-atoms. How should we handle such default-atoms by modifying our SLD-resolution? Let us try this:

- If we reach a default-atom "*not* A" as a subgoal of our original query, we keep the current SLD-tree in mind and start a new SLD-tree by trying to solve "A".
- If this succeeds, then we falsified "*not* A", the current branch is failing and we have to backtrack and consider a different subquery.
- But it can also happen that the SLD-tree for "A" is *finite with only failing branches*. Then we say that A *finitely fails*, we turn back to our original SLD-tree, consider the subgoal "*not* A" as successfully solved and go on with the next subgoal in the current list.

It is important to note that an SLD-tree for a positive atom can fail without *being finite*. The SLD-tree for the program consisting of the single rule $p \leftarrow p$ with respect to the query p is infinite but failing (it consists of one single infinite branch). In Figure 1 the leftmost branch is also failing but infinite.

Although this idea of *Finite-Failure* is very procedural in nature, there is a nice modeltheoretical counterpart — Clark's completion *comp*(P) ([Cla78]). The idea of Clark was that a program P consists not only of the implications, but also of the information that *these are the only ones*. Roughly speaking, he argues that one should interpret the "$\leftarrow$"-arrows in rules as equivalences "$\equiv$" in classical logic. We do not give the exact definitions here, as they are very complex; in the non-propositional case, a symbol for equality, together with axioms describing it[6], has to be introduced. However, for the propositional case, *comp*(P) is obtained from P by just

1. collecting all given clauses with the same head into one new "clause" with this respective head and a disjunctive body (containing all bodies of the old clauses), and
2. replacing the implication-symbols "$\leftarrow$" by "$\equiv$".

Definition 3.1 (Clark's Completion *comp*(P)**)**
Clark's semantics for a program P is given by the set of all classical models of the theory comp(P).

We can now see the classical theory *comp*(P) as the information contained in the program P. *comp*(P) is like a sort of closed world assumption applied to P. We are now able to derive negative information from P by deriving it from *comp*(P). In fact, the following soundness and completeness result for definite programs P and definite queries $Q = \bigwedge_i A_i$ (consisting of only *positive* atoms) holds:

[6]CET: Clark's Equational Theory. CET($\mathcal{L}_P$) axiomatizes the equality theory of all Herbrand($\mathcal{L}_P$)-models. See [MMP88, She88a] for the problem of equality and the underlying language.

Theorem 3.2 (COMP and Fair FF-Trees)
The following conditions are equivalent:

- $comp(P) \models \forall\neg Q$
- *Every fair SLD-tree for P with respect to Q is finitely failed.*

Note that in the last theorem we did not use default negation but classical negation $\neg$ because we just mapped all formulae into classical logic. We need the fairness assumption to ensure that the selection of atoms is reasonably well-behaving: we want that every atom or default-atom occurring in the list of preliminary goals will eventually be selected.

But even this result is still very weak — after all we want to handle not only negative queries but programs containing default-atoms. From now on we consider programs with default-atoms in the body. We usually denote them by

$$A \leftarrow \mathcal{B}^+ \wedge not\ \mathcal{B}^-,$$

where $\mathcal{B}^+$ contains all the positive body atoms and $not\ \mathcal{B}^-$ all default atoms "*not C*".

Our two motivating examples in Section 2.4 contain such default atoms. This gives rise to an extension of SLD, called SLDNF, which treats negation as *Finite-Failure*

$$\text{SLDNF} = \text{SLD} + not\ L \text{ succeeds, if } L \text{ finitely fails.}$$

The precise definitions of SLDNF-*resolution*, *tree*, etc. are very complex: we refer to [Llo87, Apt90]. Recently, Apt and Bol gave interesting improved versions of these notions: see [AB94, Section 3.2]. In order to get an intuitive idea, it is sufficient to describe the following underlying principle:

Principle 3.3 (A "Naive" SLDNF-Resolution)
If in the construction of an SLDNF-tree a default-atom not L_{ij} *is selected in the list* $\mathcal{L}_i = \{L_{i1}, L_{i2}, \ldots\}$*, then we try to prove* L_{ij}*.*
If this fails finitely *(it fails because the generated subtree is finite and failing), then we take* not L_{ij} *as proved and we go on to prove* $L_{i(j+1)}$*.*
If L_{ij} *succeeds, then* not L_{ij} *fails and we have to backtrack to the list* $\mathcal{L}_{i-1}$ *of preliminary subgoals (the next rule is applied: "backtracking").*

Does SLDNF-Resolution properly handle Examples 2.11 and 2.12? It does indeed:

Inheritance: The query *make_top(Tweety)* generates an SLD-tree with one main branch, the nodes of which are:

$flies(Tweety)$,
$bird(Tweety),\ not\ ab(r_1, Tweety)$,
$not\ ab(r_1, Tweety)$,
$Success$.

The third node has a sibling-node *penguin(Tweety), not* $ab(r_1, Tweety)$ which immediately fails because *Tweety* does not unify with *Sam*. The *Success*-node is obtained from *not* $ab(r_1, Tweety)$ because the corresponding SLD-tree for $ab(r_1, Tweety)$ fails finitely (this tree consists only of $ab(r_1, Tweety)$ and $penguin(Tweety)$).

YSP: The crucial query is

$$?\!-\ \text{not}\ holds(alive\,,\ res(shoot, res(wait, res(load, s_0)))).$$

So we consider $?\!-\ holds(alive\,,\ res(shoot, res(wait, res(load, s_0))))$. Again the SLD-tree for this query consists mainly of one branch: the nodes are obtained from the query by applying successively the first program rule (*law of inertia*). By evaluation of the holds-predicate, we eventually arrive at the fact $holds(alive, s_0)$ and the "*not ab*" predicates remain to be solved. For any of these predicates we again have to consider separate SLD-trees. But for $ab(r_1, shoot, alive, res(wait, res(load, s_0)))$ it is easy to see that the associated tree already finitely fails (because it generates the subgoal "*not* $ab(r_1, wait, loaded, res(load, s_0))$" the corresponding SLD-tree of which immediately finitely fails) and therefore, since no backtracking is possible, the tree for

$$?\!-\ holds(alive\,,\ res(shoot, res(wait, res(load, s_0))))$$

finitely fails and our original query succeeds: *Fred is dead.*

Up to now it seems that SLDNF-resolution solves all our problems. It handels our examples correctly, and is defined by a procedural calculus strongly related to SLD. There are two main problems with SLDNF:

- SLDNF can not handle free variables in negative subgoals,
- SLDNF is still too weak for Knowledge Representation.

The latter problem is the most important one. By looking at a particular example, we will motivate in Section 3.2 the need for a stronger semantics. This will lead us in the remaining sections to the wellfounded and the stable semantics.

For the rest of this section we consider the first problem, known as the *Floundering Problem.* This problem will also occur later in implementations of the wellfounded or the stable semantics. We consider the program $P_{flounder}$ consisting of the three facts

$$p(c,c),\ q(b),\ r(f(c)).$$

Our query is $?-p(x,c), \text{not}\ q(x), r(f(x))$, that is, we are interested in instantiations of x such that the query follows from the program. The situation is

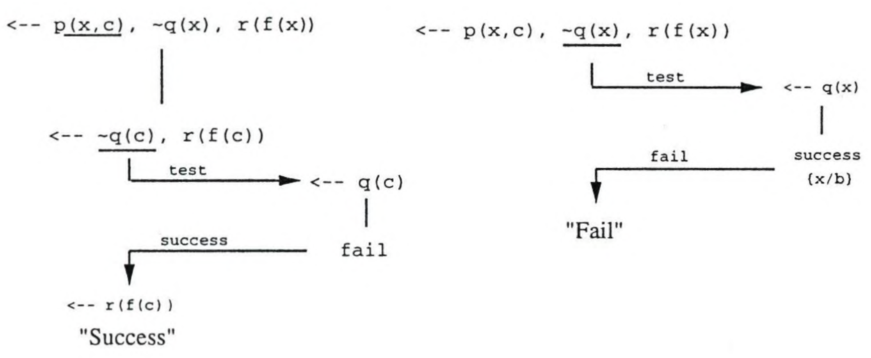


Figure 2: The Floundering-Problem

illustrated in Figure 2. Let us suppose that we always select the first atom or default-atom: it is underlined in the sequel. The SLDNF-tree of this trivial example is linear and has three nodes: the first node is the query itself

$$? - \underline{p(x,c)}, \text{not } q(x), r(f(x)),$$

the second node is $? - \underline{\text{not } q(c)}, r(f(c))$. Now, we enter the negation-as-failure mode and ask $? - q(c)$. This query immediately fails (the generated tree exists, is finite and fails) so that we give back the answer "yes, the default atom *not* $q(c)$ succeeds and can be skipped from the list". The last node is $? - \underline{r(f(c))}$ which immediately succeeds.

Note that in the last step, the test for $? - q(c)$ has to be finished before the tree can be extended. If we get no answer, the SLDNF-tree simply does not exist: this can not happen with SLD-trees.

So far everything was fine. But what happens if we select the second atom in the first step

$$? - p(x,c), \underline{\text{not } q(x)}, r(f(x))?$$

Example 3.4 (Floundering)
We again consider the program $P_{flounder}$ consisting of the three facts

$$p(c,c),\ q(b),\ r(f(c)).$$

Our query is $? - p(x,c), not\ q(x), r(f(x))$, and in the first step we will select the second default-atom, i.e. one with a free variable. Thus we enter the negation-as-failure mode with the query $? - not\ q(x)$. In this case, x may be instantiated to b so that we have to give back the answer "no, the default-atom not $q(x)$ *fails" and the whole query will fail. This is because SLDNF treats the subgoal as "$\forall x \text{not } q(x)$" instead of "$\exists x \text{not } q(x)$" which is intended. There exist*

approaches to overcome this shortcoming by treating negation as constructive negation*: see [Cha88, CW89, Dra94].*

In the classical SLDNF-resolution *negation-as-finite-failure is only a test, no bindings are produced*. On the one hand this may be considered a shortcoming, on the other hand, it makes the SLDNF procedure more tractable. Note that the problem to decide if a given program flounders is undecidable [Bör87]). See also [She91] for more unsolvable problems related to SLDNF.

SLDNF is a procedural mechanism. It would be nice to have a modeltheoretical counterpart. In Theorem 3.2 we already related a restricted form of finite failure to Clark's completion. We will see later that $comp(P)$ is inconsistent even in cases where we would not expect it. Therefore Fitting [Fit85] introduced a three-valued formulation $comp_3(P)$ of the original completion. Kunen ([Kun87]) then proved in the propositional case *SLDNF is sound and complete with respect to* $comp_3(P)$.

In the predicate logic case, SLDNF is not complete but it is always correct [She88b, Theorem 39]) with respect to $comp_3(P)$: given a query Q,

- if SLDNF succeeds with answer Θ, then $comp_3(P) \models_3 \forall Q\Theta$, and
- if SLDNF fails, then $comp_3(P) \models_3 \neg\exists Q$.

This correctness result is also the reason for the incompleteness of SLDNF with respect to two-valued $comp(P)$. It states that any formula derivable by SLDNF is a three-valued consequence of $comp_3(P)$. But, since there are two-valued consequences of a theory that are not three-valued ones (three-valued logic is weaker than two-valued logic), SLDNF can not be complete. Extensions of the above completeness result to certain subclasses of predicate logic programs require severe restrictions on the syntactic form of P. To define these syntactic restrictions, we need the notion of the dependency-graph:

Definition 3.5 (Dependency-Graph $\mathcal{G}_P$)
For a logic program P with negation, the dependency graph *$\mathcal{G}_P$ is a finite directed graph whose vertices are the predicate symbols from P. There is a* positive *(respectively* negative*) edge from R to R' iff there is a clause in P with R in its head and R' occurring positively (respectively negative) in its body.*

We also say

- *R* depends on *R' if there is a path in $\mathcal{G}_P$ from R to R' (by definition, R depends on itself),*
- *R* depends positively *(resp.* negatively*) on R' if there is a path in $\mathcal{G}_P$ from R to R' containing only* positive *edges (resp.* at least one negative *edge). (by definition R depends positively on itself),*
- *R* depends evenly *(resp.* oddly*) on R' if there is a path in $\mathcal{G}_P$ from R to R' containing an* even *(resp.* odd*) number of negative edges (by definition R depends evenly on itself).*

Prog. P	Semantics	Completeness
allowed + hierarchical	$comp(P)$	yes, no recursion at all
allowed + stratified	$comp(P)$	yes, if $P \cup \{\leftarrow A\}$ strict
allowed	$comp_3(P)$	yes, w.r.t. $\vdash_3$
allowed + call-consistent	$comp(P)$	yes, if $P \cup \{\leftarrow A\}$ strict: $comp(P) \vdash \forall A$ iff $comp_3(P) \vdash_3 \forall A$

Table 1: Completeness for SLDNF

The following properties of a program P turn out to be very important:

> *stratified:* no predicate depends negatively on itself[7],
> *strict:* there are no dependencies that are both even and odd,
> *call-consistent:* no predicate depends oddly on itself[8],
> *allowedness:* every variable occurring in a clause must occur in at least one positive atom of the body of that clause.

Strictness and allowedness turn out to be the most important restrictions that imply completeness results for SLDNF:

While *strictness* excludes situations of the form $p(x) \leftarrow q(x), p(c) \leftarrow \neg q(f(c'))$, *allowedness* excludes constructs of the form $equal(x, x) \leftarrow$ and also solves the floundering-problem.

Strictness implies that $comp_3(P)$ and $comp(P)$ are equivalent [Kun89])

$$comp_3(P) \models_3 \forall Q\Theta \text{ iff } comp(P) \models \forall Q\Theta.$$

Table 1 gives an overview of the different completeness results. Note that the query A is always considered to be allowed.

Much work was done in LP (see [DC90, BM86, Stä94]) to find other syntactically characterizable classes, for which SLDNF is also *complete.*

3.2 Negation-as-Failure

Let us first illustrate that SLDNF answers quite easily our requirements of a semantics SEM (stated explicitly in Definition 2.13). We can formulate these

[7]or: there are no cycles containing at least one edge.
[8]or: there are no odd cycles.

requirements as two program-transformations (they will be used later for computing a semantics). We call them *Reductions* for obvious reasons.

Principle 3.6 (Reduction)
*Suppose we are given a program P with possibly default-atoms in its body. If a ground atom A does not unify with any head of the rules of P, then we can delete in every rule any occurrence of "*not *A" without changing the semantics.*

*Dually, if there is an instance of a rule of the form "$B \leftarrow$ " then we can delete all rules that contain "*not *B" in their bodies.*

It is obvious that SLDNF "implements" these two reductions automatically. The weakness of SLDNF for Knowledge Representation is in a sense inherited from SLD. When we consider rules of the form "$p \leftarrow p$", then SLD resolution gets into an infinite loop and no answer to the query *?- p* can be obtained. This has often the effect that when we enter into negation-as-failure mode, the SLD-tree to be constructed is not finite, although he is not successful and therefore should be considered as failed.

Let us discuss this point with a more serious example.

Example 3.7 (The Transitive Closure)
Assume we are given a graph consisting of nodes and edges between some of them. We want to know which nodes are reachable from a given one. A natural formalization of the property "reachable" would be

$$reachable(x) \leftarrow edge(x,y), reachable(y).$$

What happens if we are given the following facts

$$edge(a,b),\ edge(b,a),\ edge(c,d)$$

and $reachable(c)$? Of course, we expect that neither a nor b are reachable because there is no path from c to either a or b.

*But SLDNF-Resolution does not derive "*not *$reachable(a)$"!*

How does this result relate to Theorem 3.2? Note that our query has exactly the form as required there. Clark's completion of our program rule is

$$reachable(x)\ \equiv\ (x \doteq c \lor \exists y\,(reachable(y) \land edge(y,x)))$$

from which, together with our facts about the edge-relation, $\neg reachable(a)$ is indeed not derivable. This is due to the wellknown fact that transitive closure is not expressible in first order predicate logic.

Note also that our Principle 2.8 does not help, because it simply does not apply. It turns out that we can augment our two principles by a third one, that constitutes together with them a very nice calculus handling the above example in the right way. This principle is related to *Partial Evaluation*, hence its

name GPPE[9]. Let us motivate this principle with the last example. The query "*not* $reachable(a)$" leads to the rule "$reachable(a) \leftarrow edge(a,b), reachable(b)$" and "$reachable(b)$" leads to "$reachable(b) \leftarrow edge(b,a), reachable(a)$". Both rules can be seen as definitions for $reachable(a)$ and $reachable(b)$ respectively. So it should be possible to replace in these rules the body atoms of *reachable* by their definitions. Thus we obtain the two rules

$$\begin{array}{l} reachable(a) \leftarrow edge(a,b), edge(b,a), reachable(a) \\ reachable(b) \leftarrow edge(b,a), edge(a,b), reachable(b) \end{array}$$

that can both be eliminated by applying Principle 2.8. So we end up with a program that does neither contain $reachable(a)$ nor $reachable(b)$ in one of the heads. Therefore, according to Principle 2.3 both atoms should be considered false. The precise formulation of this principle is as follows:

Principle 3.8 (GPPE)
We say that a semantics SEM satisfies GPPE, if the following transformation does not change the semantics. Replace a rule $\mathcal{A} \leftarrow \mathcal{B}^+ \wedge not\ \mathcal{B}^-$ where $\mathcal{B}^+$ contains a distinguished atom B by the rules

$$\mathcal{A} \cup (\mathcal{A}_i \setminus \{B\}) \ \leftarrow\ (\mathcal{B}^+ \setminus \{B\}) \cup \mathcal{B}_i^+ \ \wedge\ not\ (\mathcal{B}^- \cup \mathcal{B}_i^-)\ (i = 1, \ldots, n)$$

where $\mathcal{A}_i \leftarrow \mathcal{B}_i^+ \wedge \text{not } \mathcal{B}_i^-$ $(i = 1, \ldots, n)$ *are all the rules with* $B \in \mathcal{A}_i$.

Note that any semantics SEM satsfying GPPE and Elimination of Tautologies can be seen as extending SLD by doing some *Loop-checking*. We will call such semantics *NMR-semantics* in order to distinguish them from the classical *LP-semantics* which are based on SLDNF or variants of Clark's completion $comp(P)$:

- *NMR-Semantics* = *SLDNF + Loop-check.*

The following, somewhat artificial example illustrates this point.

Example 3.9 (COMP vs. NMR)

$$\begin{array}{rlll} P_{NMR}: & p & \leftarrow & p \\ & q & \leftarrow & \text{not } p \end{array} \qquad \begin{array}{rlll} P'_{NMR}: & p & \leftarrow & p \\ & q & \leftarrow & \text{not } p \\ & r & \leftarrow & \text{not } r \end{array}$$

$$\begin{array}{rlll} comp(P_{NMR}): & p & \equiv & p \\ & q & \equiv & \neg p \end{array} \qquad \begin{array}{rlll} comp(P'_{NMR}): & p & \equiv & p \\ & q & \equiv & \neg p \\ & r & \equiv & \neg r \end{array}$$

?-q: *No (COMP).* *Yes (NMR).*   *?-p*: *Yes (COMP).* *No (NMR).*

[9]Generalized Principle of Partial Evaluation

For both programs, the answers of the completion-semantics do not match our NMR-intuition! In the case of P_{NMR} we expect q to be derivable, since we expect not p *to be derivable: the only possibility to derive p is the rule $p \leftarrow p$ which, obviously, will never succeed. But $q \notin Th(\{q \equiv \neg p\}) = comp(P_{NMR})$! In the case of P'_{NMR} we expect p not to be derivable, for the same reason: the only possibility to derive p is the rule $p \leftarrow p$. But $p \in Fml = Th(\{r \equiv \neg r\}) = comp(P'_{NMR})$!*

Note that the answers of the completion-semantics agree with the mechanism of SLDNF: $p \leftarrow p$ represents a loop. The completion of P' is inconsistent: this led Fitting to consider the three-valued version of $comp(P)$ mentioned at the end of Section 3.1. This approach avoids the inconsistency (the query $? - p$ is not answered "yes") but it still does not answer "no" as we would like to have.

The last principle in this section is related to *Subsumption*: we can get rid of non-minimal rules by simply deleting them.

Principle 3.10 (Subsumption)
In a program P we can delete a rule $A \leftarrow \mathcal{B}^+ \wedge \text{not}\ \mathcal{B}^-$ whenever there is another rule $A \leftarrow \mathcal{B}'^+ \wedge \text{not}\ \mathcal{B}'^-$ with

$$\mathcal{B}'^+ \subseteq \mathcal{B}^+ \text{ and } \mathcal{B}'^- \subseteq \mathcal{B}^-.$$

As a simple example, the rule $A \leftarrow B, C, not\ D, not\ E$ is subsumed by the 3 rules $A \leftarrow C, not\ D, not\ E$ or $A \leftarrow B, C, not\ E$ and by $A \leftarrow C, not\ E$.

3.3 The Wellfounded Semantics: WFS

The wellfounded semantics, originally introduced in [vGRS88], is the *weakest* semantics satisfying our 4 principles (see [BD95a, Dix95b]). We call a semantics SEM_1 weaker than SEM_2, if for all programs P and all atoms or default-atoms l the following holds: $\text{SEM}_1(P) \models l$ implies $\text{SEM}_2(P) \models l$. I.e. all atoms derivable from SEM_1 with respect to P are also derivable from SEM_2. This is a nice theorem and gives rise to the following definition:

Theorem 3.11 (WFS)
There exists the weakest semantics satisfying our four principles Elimination of Tautologies, Reduction, Subsumption and GPPE. This semantics is called wellfounded semantics WFS.

It can also be shown, that for propositional programs, our transformations can be applied to compute this semantics.

Theorem 3.12 (Confluent Calculus for WFS)
The calculus consisting of these four transformations is confluent, i.e. whenever we arrive at an irreducible program, it is uniquely determined. The order of the transformations does not matter.

For finite propositional programs, it is also terminating: any program P is therefore associated a unique normalform $res(P)$. The wellfounded semantics of P can be read off from $res(P)$ as follows

$$WFS(P) = \{A: \ A \leftarrow \ \in res(P)\} \cup \{\text{not } A: \ A \textit{ is in no head of } res(P)\}$$

Therefore the wellfounded semantics associates to every program P with negation a set consisting of atoms and default-atoms. This set is a 3-valued model of P. It can happen, of course, that this set is empty. But it is always consistent, i.e. it does not contain an atom A and its negation *not* A. Moreover, it extends SLDNF: whenever SLDNF derives an atom or default-atom and does not flounder, then WFS derives it as well. Therefore the two examples of Section 2.4 are handled in the right way. But also for Example 3.7 we get the desired answers.

As we said above, loop-checking is in general undecidable. Therefore WFS is in the most general case where variables and function-symbols are allowed, undecidable. Only for finite propositional programs it is decidable. In fact, it is of quadratic complexity (see Section 3.5).

Let us end this section with another example, which contains negation.

Example 3.13 (Van Gelder's Example)
Assume we are describing a two-players game like checkers. The two players alternately move a stone on a board. The moving player wins when his opponent has no more move to make. We can formalize that by

- $wins(x) \leftarrow move\_from\_to(x,y)$, not $wins(y)$

meaning that

- *the situation x is won (for the moving player A), if he can lead over*[10] *to a situation y that can never be won for B.*

Assume we also have the facts $move\_from\_to(a,b)$, $move\_from\_to(b,a)$ and $move\_from\_to(b,c)$. Our query to this program P_{game} is ?- $wins(b)$. Here we have no problems with floundering, but using SLDNF we get an infinite sequence of oscillating SLD-trees (none of which finitely fails).

WFS, however, derives the right results

$$WFS(P_{game}) = \{not\ wins(c), wins(b), not\ wins(a)\}$$

which matches completely with our intuitions.

[10]With the help of a regular move, given by the relation $move\_from\_to(,)$.

3.4 The Stable Semantics: STABLE

We defined WFS as the weakest semantics satisfying our four principles. This already indicates that there are even stronger semantics. One of the main competing approaches is the stable semantics STABLE. The stable semantics associates to any program P a set of 2-valued models, like classical predicate logic. STABLE satisfies the following property, in addition to those that have been already introduced:

Principle 3.14 (Elimination of Contradictions)
Suppose a program P has a rule which contains the same atom A and not A *in its body. Then we can eliminate this rule without changing the semantics.*

This principle can be used, in conjunction with the others to define the stable semantics

Theorem 3.15 (STABLE)
There exists the weakest semantics satisfying our five principles Elimination of Tautologies, Reduction, Subsumption, GPPE and Elimination of Contradictions.

If a semantics SEM satisfies *Elimination of Contradictions* it is based on 2-valued models ([BD95b]). The underlying idea of STABLE is that any atom in an intended model should have a definite reason to be true or false. This idea was made explicit in [BF91a, BF91b] and, independently, in [GL88]. We use the latter terminology and introduce the Gelfond-Lifschitz transformation: for a program P and a model $N \subseteq B_P$ we define

$$P^N := \{rule^N : rule \in P\}$$

where $rule := A \leftarrow B_1, \dots, B_n, not\ C_1, \dots, not\ C_m$ is transformed as follows

$$(rule)^N := \begin{cases} A \leftarrow B_1, \dots, B_n, & \text{if } \forall j : C_j \notin N, \\ \mathbf{t}, & \text{otherwise.} \end{cases}$$

Note that P^N is always a *definite* program. We can therefore compute its least Herbrand model M_{P^N} and check whether it coincides with the model N with which we started:

Definition 3.16 (STABLE)
N is called a stable *model*[11] *of P iff $M_{P^N} = N$.*

What is the relationship between STABLE and WFS? We have seen that they are based on rather identical principles.

- Stable models N extend WFS: $l \in \text{WFS}(P)$ implies $N \models l$.

[11]Note that we only consider Herbrand models.

- If WFS(P) is two-valued, then WFS(P) is the unique stable model.

But there are also differences. We refer to Example 3.13 and consider the program P consisting of the clause

$$wins(x) \leftarrow move\_from\_to(x,y),\ not\ wins(y)$$

together with the following facts: $move\_from\_to(a,b)$, $move\_from\_to(b,a)$, as well as $move\_from\_to(b,c)$, and $move\_from\_to(c,d)$. In this particular case we have two stable models: $\{wins(a), wins(c)\}$ and $\{wins(b), wins(c)\}$ and therefore

$$\text{WFS}(P) = \{wins(c),\ not\ wins(d)\} = \bigcap_{\mathcal{N}\text{ a stable model of }P} \mathcal{N}.$$

This means that the 3-valued wellfounded model is exactly the set of all atoms or default-atoms true in all stable models. But this is not always the case, as the program of $P_{splitting}$ shows:

Example 3.17 (Reasoning by cases)

$$\begin{array}{llll} P_{splitting}: & a & \leftarrow & \text{not } b \\ & b & \leftarrow & \text{not } a \\ & p & \leftarrow & a \\ & p & \leftarrow & b \end{array}$$

Although neither a, nor b can be derived in any semantics based on two-valued models (as STABLE for example), the disjunction $a \vee b$, thus also p, is true. In this way the example is handled by the completion semantics, too. WFS(P), however, is empty; if the WFS cannot decide between a or not a, *then a is undefined.*

The main differences between STABLE and WFS are

- STABLE is not always consistent,
- STABLE does not allow for a goal-oriented implementation.

The inconsistency comes from odd, negative cycles

$$STABLE(p \leftarrow not\ p) = \emptyset.$$

The idea to consider 2-valued models for a semantics neccessarily implies its inconsistency ([BD95b]). Note that $WFS(p \leftarrow not\ p) = \{\emptyset\}$ which is quite different! Sufficient criteria for the existence of stable models are contained in [Dun92, Fag93].

That STABLE does not allow for a Top-Down evaluation is a more serious drawback and has nothing to do with inconsistency. This behaviour led

Dix to define the notion of *Relevance* and *Modularity* (see Section 7.1 and [Dix92a, Dix92b, Dix95b]. Recently, Bry reinvented *Modularity* (he termed it *compositionality*) and argued that a semantics should satisfy it.

Example 3.18 (STABLE is not Goal-Oriented)

$$P_{rel(a)}: \begin{array}{lcl} a & \leftarrow & \text{not } b \\ b & \leftarrow & \text{not } a \end{array} \qquad P: \begin{array}{lcl} a & \leftarrow & \text{not } b \\ b & \leftarrow & \text{not } a \\ p & \leftarrow & \text{not } p \\ p & \leftarrow & a \end{array}$$

$P_{rel(a)}$ is the subprogram of P that consists of all rules that are relevant to answer the query ?- a. It has two stable models $\{a\}$ and $\{b\}$ — a is not true in all of them. But the program P has the unique stable model $\{p, a\}$, so a is true in all stable models of P.

The last example shows that the truthvalue of an atom a also depends on atoms that are totally unrelated with a! This is considered a drawback of STABLE by many people. Note that a straightforward modification of STABLE is not possible ([DM94b, DM94c]).

We end this section with another description of WFS and STABLE that will be useful in later sections. It was introduced in [BS91, BS92]:

Definition 3.19 (Antimonotone Operator γ_P)
For a program P and a set $N \subset B_P$ we define an operator γ_P mapping Herbrand-structures to Herbrand structures:

$$\gamma_P(N) := M_{P^N}.$$

It is easy to see that γ_P is antimonotone. Therefore its twofold application γ^2 is monotone ([Tar55]).

Obviously, the stable models of a program P are exactly the fixpoints of γ_P. This is just a reformulation of Definition 3.16. WFS is related to γ as follows

Theorem 3.20 (WFS and γ^2)
A positive atom A is in WFS(P) iff $A \in lfp(\gamma_P^2)$. A default-atom not A *is in WFS(P) iff $A \notin gfp(\gamma_P^2)$:*

$$WFS(P) = lfp(\gamma_P^2) \cup \{\text{not } A : A \notin gfp(\gamma_P^2)\}.$$

Atom or default-atoms that do occur in neither of the two sets are undefined.

3.5 Complexity and Expressibility

In this section we collect some complexity results for the semantics considered so far. The reason why NMR-semantics are in the general case (free variables

	Complexity	
	1. ord. prog. (with functions)	*prop. prog. (no variables)*
$\mathrm{M_P}$ *(P is Horn)*	*A*: Σ^0_1-compl. *not A*: Π^0_1-compl.	**linear** in $\lvert P\rvert$
$\mathrm{M_P^{supp}}$ *(P is stratified)*	**arithm.**-compl. (M_P^{supp} is Σ^0_n)	**linear** in $\lvert P\rvert$
COMP	Π^1_1-compl. over $\mathbb{N}$	**co-NP**-compl.
$\mathrm{COMP_3}$	Π^1_1-compl. over $\mathbb{N}$	**linear** in $\lvert P\rvert$
STABLE	Π^1_1-compl. over $\mathbb{N}$	**co-NP**-compl.
REG-SEM	Π^1_1-compl. over $\mathbb{N}$	**co-NP**-compl.
WFS	Π^1_1-compl. over $\mathbb{N}$	**linear** in $\#At \times \lvert P\rvert$
WFS′	Π^1_1-compl. over $\mathbb{N}$	**co-NP**-compl.
$\mathrm{WFS^+}$	Π^1_1-compl. over $\mathbb{N}$	**co-NP**-compl.

Table 2: Complexity of Non-Disjunctive Semantics

and function symbols) undecidable is strongly related to loop-checking. Let us consider the program

$$P(x) \leftarrow P(f(x))$$

or, equivalently, the infinite propositional program

$$p_0 \leftarrow p_1,\ p_1 \leftarrow p_2,\ \ldots, p_i \leftarrow p_{i+1}, \ldots$$

Any NMR-semantics should derive "*not* $P(t)$" (resp. "*not* p_i") for all terms t, but a procedure to detect such infinite loops is impossible in general. Our principles *GPPE* and *Elimination of Tautologies* can detect *finite* loops.

From a modeltheoretic point of view it is easy to *define* a semantics that derives "*not* $P(t)$": we could just take all minimal Herbrand models as the intended semantics. Of course, this does not change the general undecidability.

For the exact terminology, definitions and results presented in this section we refer the interested reader to the following interesting overviews [Sch90, Sch92, CS93]. Further results are contained in [EGM93, Sac93, CS90, EG93].
While Table 2 treats the complexity Table 3 treats the expressibility problem. Some general explanations are appropriate:

Table 2: We consider the complexity of deciding if a *given ground atom*

	Expressibility *1. ord. prog.* *(no functions)*
M_P *(P is Horn)*	$\subset$ **IND** (thus $\subset$ **P**)
COMP	= **co-NP**
COMP$_3$	= **IND** (thus $\subset$ **P**)
STABLE	= **co-NP**
REG-SEM	$= \Pi_2^P$
WFS	= **IND** (thus $\subset$ **P**)
WFS′	= **IND** (thus $\subset$ **P**)
WFS$^+$	= **IND** (thus $\subset$ **P**)

Table 3: Expressibility of Non-Disjunctive Semantics

or default-atom is contained in the respective semantics (i.e. if it is true in all *intended* models).

For the 1. column, we consider arbitrary first-order programs with function symbols. We therefore get undecidability results of varying strength. Since we restrict to Herbrand models, we can assume (by standard recursive encoding techniques, like *Gödel-numberings*) that all models have universes which are subsets of the natural numbers $\mathbb{N}$. The completeness results mean that for every set of the respective complexity class there is a program that defines this set under the respective sceptical semantics. Unless indicated otherwise, there is no difference between deciding ground *atoms* or ground *negated atoms*.

For the 2. column, we consider *propositional* programs. Hence we get decidable problems of various degrees. We denote by $|P|$ the total length of the program and by $\#At$ the number of distinct proposition letters in P. See also [BED92, Sch92, Imi91, MRT92, Wit91b, JdL92] for more results on the complexity of propositional programs.

Table 3: Here we consider the *expressibility* (or expressive power) of first order programs *without* function symbols. The idea is to distinguish between EDB-relations (relations that do not appear in the head of a program) and IDB-relations (which are contained in some heads). For a given program P we can view any instance $\mathcal{D}$ of the (finite) EDB-relations

as an *input* argument and then compute the (finite) IDB-relations (the *output*) under the respective sceptical semantics. So we are asking

What are the relations expressible with logic programs under certain semantics?

Roughly speaking, a relation R over finite EDB's $\mathcal{D}$ (i.e. for every finite $\mathcal{D}$ is associated a relation $R^{\mathcal{D}}$ on $\mathcal{D}$) is *expressible* if there is a program P containing an IDB-symbol r s. t. for every relational database $\mathcal{D}$ and tuple $\underline{t}$ corresponding to r:

$$r(\underline{t}) \in SEM(P + \mathcal{D}) \text{ if and only if } R(\underline{t}) \text{ holds in } \mathcal{D}.$$

This is the classical notion of *expressibility* ([Sch90, EGM93]).

We are in particular interested to express all relations of some complexity class (note that the complexity is always with respect to the finite relational database as input, the program is fixed). It is well-known that the relations *inductively definable* over $\mathcal{D}$, we denote them by **IND**$(\mathcal{D})$ (or simply **IND** to avoid the explicit occurrence of the EDB), is a strict subclass of the relations that are *polynomial* over $\mathcal{D}$ (see [Bar75, Mos74, Gur88]).

It is worth noting that in the general predicate logic case, all semantics are highly undecidable. The entries for comp and comp$_3$ are to be understood as restricted to Herbrand models.

In the propositional case, WFS is of quadratic complexity (a folklore result — for a proof see [Wit91a]), while STABLE is co-NP-complete. The low complexity of WFS can be traced back to Dowling and Gallier's result whereby satisfiability of Horn clauses can be tested in linear time([DG84]). In Dowling and Gallier's approach it is actually a minimal model of a Horn theory that is computed in linear time. Since minimal models of Horn theories are equivalent to closures of rules without negation the result is directly applicable to well-founded semantics for logic programs with default-atoms.

As far as expressibility is concerned, STABLE is more expressive: all co-NP-relations can be expressed, while WFS can only describe all inductively definable relations. As an example, STABLE can express the satisfiability problem. WFS is not able to do this (unless the polynomial hierachy collapses).

4 Adding Explicit Negation

So far we have considered programs with one special type of negation, namely default negation. Default negation is particularly useful in domains where complete positive information can be obtained. For instance, if one wants to represent flight connections from Budapest to the US it is very convenient to represent all existing flights and to let default negation handle the derivation of negative information. There are domains, however, where the lack of positive information cannot be assumed to support (or support with enough strength) that this information is false. In such domains it becomes important to distinguish between cases where a query does not succeed and cases where the negated query succeeds. The following example was used by McCarthy to illustrate the issue. Assume one wants to represent the rule: cross the railroad tracks if no train is approaching. The straightforward representation of this rule with default negation would be

$$crosstracks \leftarrow not\ train$$

It seems obvious that in many practical settings the use of such a rule would not lead to intended behaviour, in fact it might even have disasterous consequences. What seems to be needed here is the possibility of using a different negation symbol representing a stronger form of negation. This new negation — we will call it explicit negation — should be true only if the corresponding negated literal can actually be derived. We will use the classical negation symbol $\neg$ to represent explicit negation. The track crossing rule will be represented as

$$crosstracks \leftarrow \neg train$$

The idea is that this latter rule will only be applicable if $\neg train$ has been proved, contrary to the first rule which is applicable whenever *train* is not provable.

In the next subsection we will shortly discuss that explicit negation is (or should not be) *classical* negation and how it should interfere with default negation. In the two following subsections we will generalize the semantics STABLE and WFS, respectively, to programs with explicit negation.

4.1 Explicit vs. Classical and Strong Negation

First we define the language we are using more precisely.

Definition 4.1 (Extended Logic Program)
An extended logic program consists of rules of the form

$$c \leftarrow a_1, \ldots, a_n, \text{not } b_1, \ldots, \text{not } b_m$$

where the a_i, b_j and c are literals, i.e., either propositional atoms or such atoms preceded by the classical negation sign. The symbol "not " denotes negation by failure (default negation), "$\neg$" denotes explicit negation.

We have already motivated the need of a second kind of negation "$\neg$" different from "*not* ". What should the semantics of "$\neg$" be? Should it be just like in classical logic? Note that classical negation satisfies the law of excluded middle

$$A \vee \neg A.$$

The following example taken from [APP96] shows that classical negation is sometimes inappropriate for KR-tasks.

Example 4.2 (Behaviour of Classical Negation)
Suppose an employer has several candidates that apply for a job. Some of them are clearly qualified while others are not. But there may also be some candidates whose qualifications are not clear and who should therefore be interviewed in order to find out about their qualifications. If we express the situation by

$$hire(X) \leftarrow qualified(X) \text{ and } reject(X) \leftarrow \neg qualified(X)$$

then, interpreting "$\neg$" as classical negation, we are forced to derive that every candidate must either be hired or rejected! There is no room for those that should be interviewed. Also, applying the law of excluded middle has a highly non-constructive flavor.

Let us now again consider again the example $crosstracks \leftarrow \neg train$ from the beginning of this section. Suppose that we replace $\neg train$ by $free\_track$. We obtain

$$crosstracks \leftarrow free\_track.$$

From this program, "*not crosstracks*" will be derivable for any semantics. Therefore we should make sure that "*not crosstracks*" is also derivable from $crosstracks \leftarrow \neg train$ — after all, the second program is obtained from the first one by a simple syntactic operation. This means we have to make sure that default negation "*not* " treats positive and negative atoms symmetrically.

Such a negation, we will call it *explicit* will be introduced in the next two subsections. Note that Gelfond/Lifschitz called the negation they introduced in their stable semantics *classical*, although it is not classical in the sense that we just discussed. Sometimes explicit negation is also called *strong* negation and denotes still a variant of our explicit negation. In [APP96] the authors introduce both a strong and explicit negation and discuss their relation with classical and default negation at length.

4.2 STABLE for Extended Logic Programs

The extension of STABLE to extended logic programs is based on the notion of answer sets which generalize the original notion of stable models in a rather straightforward manner. Let us first introduce some useful notation. We say a rule $r = c \leftarrow a_1, \ldots, a_n, not\ b_1, \ldots, not\ b_m \in P$ is defeated by a literal l

iff $l = b_i$ for some $i \in \{1, \ldots, m\}$. We say r is defeated by a set of literals X if X contains at least one literal that defeats r. Furthermore, we call the rule obtained by deleting weakly negated preconditions from r the monotonic counterpart of r and denote it with $Mon(r)$. We also apply Mon to sets of rules with the obvious meaning.

Definition 4.3 (**X-reduct**)
Let P be an extended logic program, X a set of literals. The X-reduct of P, denoted P^X, is the program obtained from P by

- *deleting each rule defeated by X, and*
- *replacing each remaining rule r with its monotonic counterpart $Mon(r)$.*

Definition 4.4 (**Consequences of Rules**)
Let R be a set of rules without negation as failure. $Cn(R)$ denotes the smallest set of literals that is

1. *closed under R, and*
2. *logically closed, i.e., either consistent or equal to the set of all literals.*

Definition 4.5 (**γ_P**)
Let P be a logic program, X a set of literals. Define an operator γ_P as follows:

$$\gamma_P(X) = Cn(P^X)$$

X is an answer set of P iff $X = \gamma_P(X)$.

A literal l is a consequence of a program P under the new semantics, denoted $l \in STABLE(P)$, iff l is contained in all answer sets of P.

It is not difficult to see that for programs without explicit negation stable models and answer sets coincide. Here is an example involving both types of negation. The example describes the strategy of a certain college for awarding scholarships to its students. It is taken from [BG94]:

$$\begin{array}{lllll} P_{el}: & (1) & eligible(x) & \leftarrow & highGPA(x) \\ & (2) & eligible(x) & \leftarrow & minority(x), fairGPA(x) \\ & (3) & \neg eligible(x) & \leftarrow & \neg fairGPA(x), \neg highGPA(x) \\ & (4) & interview(x) & \leftarrow & not\ eligible(x), not\ \neg eligible(x) \end{array}$$

Assume in addition to the rules above the following facts about Anne are given:

$$fairGPA(Anne), \neg highGPA(Anne)$$

We obtain exactly one answer set, namely

$$\{fairGPA(Anne), \neg highGPA(Anne), interview(Anne)\}$$

Anne will thus be interviewed before a decision about her eligibility is made. If we use the above rules together with the facts

$$minority(Mike), fairGPA(Mark)$$

then the program entails $eligible(Mike)$.

The following results are taken from [Lif96]:

Lemma 4.6 (Program Types)
Let P be an extended logic program. P satisfies exactly one of the following conditions:

- *P has no answer sets,*
- *the only answer set for P is Lit,*
- *P has an answer set, and all its answer sets are consistent.*

A program is consistent if the set of its consequences is consistent, and inconsistent otherwise. The former corresponds to the first two cases listed in the proposition, the latter to the third case.

We say that a set X of literals is supported by P if, for each literal $l \in X$, there exists a rule $l \leftarrow a_1, \ldots, a_n, not\ \ b_1, \ldots, not\ \ b_m$ in P such that

1. $\{a_1, \ldots, a_n\} \subseteq X$, and
2. $\{b_1, \ldots, b_m\} \cap X = \emptyset$.

Lemma 4.7 (Properties of answer sets)
Let P be an extended logic program. The following properties hold:

- *Any consistent answer set for P is supported by P.*
- *If X and Y are answer sets of P and $X \subseteq Y$ then $X = Y$.*
- *Each element of a consistent answer set of P is a head literal[12] of P.*

From the last property it follows immediately that every consequence of P is a head literal of P whenever P is consistent. We would finally like to mention the following theorem:

Theorem 4.8 (Head Consistency)
If the set of head literals of an extended program P is consistent then every answer set of P is consistent.

[12]A head literal of a program P is the head of a rule of P (see also Principle 2.3 and Definition 6.5).

Note that a program satisfying the conditions of the last theorem can still be inconsistent since it may have no answer set at all.

We would finally like to mention that extended logic programs under answer set semantics can be reduced to general logic programs as follows: for any predicate p occurring in a program P we introduce a new predicate symbol p' of the same arity representing the explicit negation of p. We then replace each occurrence of $\neg p$ in the program with p', thus obtaining the general logic program P'. It can be proved that a consistent set of literals S is an answer set of P iff the set S' is a stable model of P', where S' is obtained from S by replacing $\neg p$ with p'.

4.3 WFS for Extended Logic Programs

We now show how the second major semantics for general logic programs, WFS, can be extended to logic programs with explicit negation. For our purposes the characterization of WFS given in Theorem 3.20 will be useful. WFS is based on a particular three-valued model. To simplify our presentation in this section we will restrict ourselves to the literals which are true in this three-valued model. The literals which are false will be left implicit. They can be added in a canonical way as follows: let T, the set of true literals, be defined as the least fixed point of a monotone operator composed of two antimonotone operators $op_1 op_2$. Then the literals which are false in the three-valued model are exactly those which are not contained in $op_2(T)$. Given this canonical extension to the full three-valued model we can safely leave the false literals implicit from now on.

We will first present a formulation which can be found in various papers, e.g. [BG94, Lif96]. We then slightly modify this formulation to obtain stronger results. We finally discuss a further modification by Pereira and Alferes.

Like answer set semantics well-founded semantics for extended logic programs can be based on the operator γ_P. However, the operator is used in a totally different way. Since γ_P is anti-monotone the operator $\Gamma_P = (\gamma_P)^2$ is monotone. According to the famous Knaster-Tarski theorem [Tar55] every monotone operator has a least fixpoint. We can thus define

Definition 4.9 (WFS for extended programs)
Let P be an extended logic program. The set of well-founded conclusions of P, denoted $WFS(P)$, is the least fixpoint of Γ_P.

The fixpoint can be approached from below by iterating Γ_P on the empty set. In case P is finite this iteration is guaranteed to actually reach the fixpoint.

The intuition behind this use of the operator is as follows: whenever γ_P is applied to a set of literals X known to be true it produces the set of all literals that are still potentially derivable. Applying it to such a set of potentially derivable literals it produces a set of literals known to be true, often larger than the original set X. Starting with the empty set and iterating until the fixpoint is reached thus produces a set of true literals.

We first want to illustrate this using an example without explicit negation:

$$
\begin{array}{llcll}
P: & (1) & b & \leftarrow & \textit{not } a \\
 & (2) & c & \leftarrow & \textit{not } b \\
 & (3) & e & \leftarrow & \textit{not } d \\
 & (4) & d & \leftarrow & \textit{not } e
\end{array}
$$

In the beginning we know nothing about derivable literals, i.e., we start with empty set X. The X-reduct of the program is

$$
\begin{array}{ll}
(1) & b \\
(2) & c \\
(3) & e \\
(4) & d
\end{array}
$$

The set of consequences of this program, or in other words, the literals still considered to be potentially derivable, is thus $\{b, c, d, e\}$. If we now reduce the program with this set we obtain

$$
\begin{array}{ll}
(1) & b
\end{array}
$$

that is, the first iteration of the two-fold application of γ_P tells us that b is provable.

If we now use $X = \{b\}$ to continue the iteration we obtain the reduced program

$$
\begin{array}{ll}
(1) & b \\
(3) & e \\
(4) & d
\end{array}
$$

that is $\{b, d, e\}$ is the current set of potential conclusions. Using this set to reduce the program gives us again

$$
\begin{array}{ll}
(1) & b
\end{array}
$$

We thus have reached the least fixed point of γ_P^2 and b is the single literal provable under WFS. it turns out that no new literal

It can be shown that every well-founded conclusion is a conclusion under the answer set semantics. Well-founded semantics can thus be viewed as an approximation of answer set semantics.

Unfortunately it turns out that for many programs the set of well-founded conclusions is extremely small and provides a very poor approximation of answer set semantics. Consider the following program P_0 which has also been discussed by Baral and Gelfond [BG94]:

$$
\begin{array}{llcll}
P_0: & (1) & b & \leftarrow & \textit{not } \neg b \\
 & (2) & a & \leftarrow & \textit{not } \neg a \\
 & (3) & \neg a & \leftarrow & \textit{not } a
\end{array}
$$

The set of well-founded conclusions is empty since $\gamma_{P_0}(\emptyset)$ equals *Lit*, the set of all literals, and the *Lit*-reduct of P_0 contains no rule at all. This is surprising since, intuitively, the conflict between (2) and (3) has nothing to do with $\neg b$ and b.

This problem arises whenever the following conditions hold:

1. a complementary pair of literals is provable from the monotonic counterparts of the rules of a program P, and
2. there is at least one proof for each of the complementary literals whose rules are not defeated by $Cn(P')$, where P' consists of the "strict" rules in P, i.e., those without negation as failure.

In this case well-founded semantics concludes l iff $l \in Cn(P')$. It should be obvious that such a situation is not just a rare limiting case. To the contrary, it can be expected that many commonsense knowledge bases will give rise to such undesired behaviour. Let us consider again our Example 2.11 from Section 2.

(1) $fly(x) \leftarrow not\ \neg fly(x), bird(x)$
(2) $\neg fly(x) \leftarrow not\ \ fly(x), penguin(x)$

Assume further that the knowledge base contains the information that Tweety is a penguin bird. Now if neither $fly(Tweety)$ nor $\neg fly(Tweety)$ follows from strict rules in the knowledge base we are in the same situation as with P_0: well-founded semantics does not draw any "defeasible" conclusion, i.e. a conclusion derived from a rule with default negation in the body, at all.

We want to show that a minor reformulation of the fixpoint operator can overcome this weakness and leads to better results. Consider the following operator

$$\gamma_P^\star(X) = Cl(P^X)$$

where $Cl(R)$ denotes the minimal set of literals closed under the (classical) rules R. $Cl(R)$ is thus like $Cn(R)$ without the requirement of logical closedness. Now define

$$\Gamma_P^\star(X) = \gamma_P(\gamma_P^\star(X))$$

Again we iterate on the empty set to obtain the well-founded conclusions of a program P which we will denote $WFS^\star(P)$.

Consider the effects of this modification on our example P_0. $\gamma_{P_0}^\star(\emptyset) = \{a, \neg a, b\}$. Rule (1) is contained in the $\{a, \neg a, b\}$-reduct of P_0 and thus $\Gamma_{P_0}^\star(\emptyset) = \{b\}$. Since b is also the only literal contained in all answer sets of P_0 our approximation actually coincides with answer set semantics in this case.

In the Tweety example both $fly(Tweety)$ and $\neg fly(Tweety)$ are provable from the $\emptyset$-reduct of the knowledge base. However, this has no influence on whether a rule not containing the default negation of one of these two literals in the body is used to produce $\gamma_P^\star(\emptyset)$ or not. The effect of the conflicting information about Tweety's flying ability is thus kept local and does not have

the disastrous consequences it has in the original formulation of well-founded semantics.

It is not difficult to see that the new monotone operator is equivalent to the original one whenever P does not contain negation as failure. In this case the X-reduct of P, for arbitrary X, is equivalent to P and for this reason it does not make any difference whether to use γ_P or $\gamma_P^\star$ as the operator to be applied first in the definition of Γ_P. The same is obviously true for programs without classical negation: for such programs Cn can never produce complementary pairs of literals and for this reason the logical closedness condition is obsolete.

In the general case the new operator produces more conclusions than the original one:

Lemma 4.10 *Let P be an extended logic program. For an arbitrary set of literals X we have*

$$\Gamma_P(X) \subseteq \Gamma_P^\star(X).$$

It can also be shown that the new operator produces no unwanted results, i.e., that our new semantics can still be viewed as an approximation of answer set semantics.

Lemma 4.11 *Let P be an extended logic program. $WFS^\star$ is correct wrt. $STABLE$, i.e., $l \in WFS^\star(P)$ implies $l \in STABLE(P)$.*

An alternative, somewhat stronger approach, was developed by Pereira and Alferes [PA92, AP95, AP96], the semantics WFSX. This semantics implements the intuition that a literal with default negation should be derivable from the corresponding explicitly negated literal. The authors call this the coherence principle. To satisfy the principle they use the seminormal version of a program P, denoted $S(P)$, which is obtained from P by replacing each rule

$$c \leftarrow a_1, \ldots, a_n, not\ b_1, \ldots, not\ b_m$$

by the rule

$$c \leftarrow a_1, \ldots, a_n, not\ b_1, \ldots, not\ b_m, not\ -c$$

where $-c$ is the complement of c, i.e. $\neg c$ if c is an atom and a if $c = \neg a$. Based on this notion Pereira and Alferes consider the following monotone operator:

$$\Omega_P(X) = \gamma_P^\star \gamma_{S(P)}^\star(X)$$

The use of the seminormal version of the program in the first application of $\gamma^\star$ guarantees that a literal l is not considered a potential conclusion whenever the complementary literal is already known to be true. In the general case $S(P)^X$ contains fewer rules than P^X. Therefore, fewer literals are considered as potential conclusions and thus more conclusions are obtained in each iteration

of the monotone operator. Here is an example [BG94]:

$$
\begin{array}{llll}
P_{WFSX}: & (1) & a & \leftarrow \; not\ b \\
 & (2) & b & \leftarrow \; not\ a \\
 & (3) & \neg a & \leftarrow
\end{array}
$$

The original version of WFS does not conclude b. In WFSX the set $X = \{\neg a\}$ is obtained after the first iteration of the monotone operator. Since rule (1) is not contained in the X-reduct of the seminormal version of the program the monotonic counterpart of (2) produces b after the second iteration.

Although a number of researchers consider WFSX to be the more adequate extension of well-founded semantics to extended logic programs the original formulation is still very often found in the literature. For this reason we will base our treatment of preferences in the next section on the earlier formulation based on $\gamma^\star$. However, we will briefly show how the coherence principle can be added in a simple way.

For the next section a minor reformulation turns out to be convenient. Instead of using the monotonic counterparts of undefeated rules we will work with the original rules and extend the definitions of the two operators Cn and Cl accordingly, requiring that default negated preconditions be neglected, i.e., for an arbitrary set of rules P with default negation we define $Cn(P) = Cn(Mon(P))$ and $Cl(P) = Cl(Mon(P))$. We can now equivalently characterize γ_P and $\gamma_P^\star$ by the equations

$$\gamma_P(X) = Cn(P_X)$$

$$\gamma_P^\star(X) = Cl(P_X)$$

where P_X denotes the set of rules not defeated by X.

An alternative characterization of $\Gamma_P^\star$ will also turn out to be useful in the next section. It is based on the following notion:

Definition 4.12 (X-SAFE)
Let P be a logic program, X a set of literals. A rule r is X-safe wrt. P ($r \in SAFE_X(P)$) if r is not defeated by $\gamma_P^\star(X)$ or, equivalently, if $r \in P_{\gamma_P^\star(X)}$.

With this new notion we can obviously characterize $\Gamma_P^\star$ as follows:

$$\Gamma_P^\star(X) = Cn(P^{\gamma_P^\star(X)}) = Cn(P_{\gamma_P^\star(X)}) = Cn(SAFE_X(P))$$

It is this last formulation that we will modify. More precisely, the notion of X-safeness will be weakened to handle preferences adequately.

5 Adding Preferences

In this section we describe an extension of well-founded semantics for logic programs with two types of negation where information about preferences between rules can be expressed in the logical language. Conflicts among rules are resolved whenever possible on the basis of derived preference information. As it turns out the well-founded conclusions of propositional prioritized logic programs can be computed in polynomial time.

After giving some motivation in Section 5.1 we introduce our dynamic treatment of preferences together with several small motivating examples in Section 5.2. We show that our conclusions are, in general, a superset of the well-founded conclusions. Subsection 5.3 illustrates the expressive power of our approach using a more realistic example from legal reasoning.

5.1 Motivation

Preferences among defaults play a crucial role in nonmonotonic reasoning. One source of preferences that has been studied intensively is specificity [Poo85, Tou86, TTH91] — we already discussed it in Example 2.11. In case of a conflict between defaults we tend to prefer the more specific one since this default provides more reliable information. E.g., if we know that students are adults, adults are normally employed, students are normally not employed, we want to conclude "Peter is not employed" from the information that Peter is a student, thus preferring the student default over the conflicting adult default.

Specificity is an important source of preferences, but not the only one, and at least in some applications not necessarily the most important one. In the legal domain it may, for instance, be the case that a more general rule is preferred since it represents federal law as opposed to state law [Pra93]. In these cases preferences may be based on some basic principles regulating how conflicts among rules are to be resolved.

Also in other application domains, like model based diagnosis or configuration, preferences play a fundamental role. Model based diagnosis uses logical descriptions of the normal behaviour of components of a device together with a logical description of the actually observed behaviour. One tries to assume normal behaviour for as many components as possible. A diagnosis corresponds to a set of components for which these normalcy assumptions lead to inconsistency. Very often a large number of possible diagnoses is obtained. In real life some components are less reliable than others. To eliminate less plausible diagnoses one can give the normalcy assumptions for reliable components of higher priority.

In configuration tasks it is often impossible to achieve all of the design goals. Often one can distinguish more important goals from less important ones. To construct the best possible configurations goals then have to be represented as defaults with different preferences according to their desirability.

The relevance of preferences is well-recognized in nonmonotonic reasoning, and prioritized versions for most of the nonmonotonic logics have been proposed, e.g., prioritized circumscription [Lif85], hierarchic autoepistemic logic [Kon88], prioritized default logic [Bre94]. In these approaches preferences are handled in an "external" manner in the following sense: some ordering among defaults is used to control the generation of the nonmonotonic conclusions. For instance, in the case of prioritized default logic this information is used to control the generation of extensions. However, the preference information itself is not expressed in the logical language. This means that this kind of information has to be fully pre-specified, there is no way of reasoning *about* (as opposed to reasoning *with*) preferences. This is in strong contrast to the way people reason and argue with each other. In legal argumentation, for instance, preferences are context-dependent, and the assessment of the preferences among involved conflicting laws is a crucial (if not the most crucial) part of the reasoning. What we would like to have, therefore, is an approach that allows us to represent preference information *in* the language and derive such information dynamically.

5.2 Handling Preferences

In order to handle preferences we need to be able to express preference information explicitly. Since we want to do this *in* the logical language we have to extend the language. We do this in two respects:

1. we use a set of rule names N together with a naming function *name* to be able to refer to particular rules,
2. we use a special (infix) symbol $\prec$ that can take rule names as arguments to represent preferences among rules.

Intuitively, $n_1 \prec n_2$ where n_1 and n_2 are rule names means the rule with name n_1 is preferred over the rule with name n_2.[13]

Definition 5.1 (Prioritized Program)
A prioritized logic program is a pair $(R, name)$ where R is a set of rules and name a naming function. To make sure that the symbol $\prec$ has its intended meaning, i.e., represents a transitive and anti-symmetric relation, we assume that R contains all ground instances of the schemata

$$N_1 \prec N_3 \leftarrow N_1 \prec N_2, N_2 \prec N_3$$

and

$$\neg(N_2 \prec N_1) \leftarrow N_1 \prec N_2$$

where N_i are parameters for names. Note that in our examples we won't mention these rules explicitly.

[13]Note that for historical reasons we follow the convention that the minimal rules are the preferred ones.

The function *name* is a partial injective naming function that assigns a name $n \in N$ to some of the rules in R. Note that not all rules do necessarily have a name. The reason is that names will only play a role in conflict resolution among defeasible rules, i.e., rules with weakly negated preconditions. For this reason names for strict rules, i.e., rules in which the symbol *not* does not appear, won't be needed. A technical advantage of leaving some rules unnamed is that the use of rule schemata with parameters for rule names does not necessarily make programs infinite. If we would require names for all rules we would have to use a parameterized name for each schema and thus end up with an infinite set N of names.

In our examples we assume that N is given implicitly. We also define the function *name* implicitly. We write:

$$n_i : c \leftarrow a_1, \dots, a_n, not\ \ b_1, \dots, not\ \ b_m$$

to express that $name(c \leftarrow a_1, \dots, a_n, not\ \ b_1, \dots, not\ \ b_m) = n_i$.

For convenience we will simply speak of programs instead of prioritized logic programs whenever this does not lead to misunderstandings.

Before introducing our new definitions we would like to point out how we want the new explicit preference information to be used. Our approach follows two principles:

1. *We want to extend well-founded semantics, i.e. we want that every $WFS^\star$-conclusion remains a conclusion in the prioritized approach.*

2. *We want to use preferences to solve conflicts whenever this is possible without violating principle 1.*

Let us first explain what we mean by conflict here. Rules may be conflicting in several ways. In the simplest case two rules may have complementary literals in their heads. We call this a type-I conflict.

Definition 5.2 (Type-I Conflict)
Let r_1 and r_2 be two rules. We say r_1 and r_2 are type-I conflicting iff the head of r_1 is the complement of the head of r_2.

Conflicts of this type may render the set of well-founded conclusions inconsistent, but do not necessarily do so. If, for instance, a precondition of one of the rules is not derivable or a rule is defeated the conflict is implicitly resolved. In that case the preference information will simply be neglected. Consider the following program P_1:

$n_1 : b \leftarrow not\ \ c$
$n_2 : \neg b \leftarrow not\ \ b$
$n_3 : n_2 \prec n_1$

There is a type-I conflict between n_1 and n_2. Although the explicit preference information gives precedence to n_2 we want to apply n_1 here to comply with the first of our two principles. Technically, this means that we can apply a preferred rule r only if we are sure that r's application actually leads to a situation where literals defeating r can no longer be derived.

The following two rules exhibit a different type of conflict:

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

The heads of these rules are not complementary. However, the application of one rule defeats the other and vice versa. We call this a direct type-II conflict. Of course, in the general case the defeat of the conflicting rule may be indirect, i.e. based on the existence of additional rules.

Definition 5.3 (Type-II Conflict)
Let r_1 and r_2 be rules, R a set of rules. We say r_1 and r_2 are type-II conflicting wrt. R iff

1. *$Cl(R)$ neither defeats r_1 nor r_2,*
2. *$Cl(R + r_1)$ defeats r_2, and*
3. *$Cl(R + r_2)$ defeats r_1*

Here $R+r$ abbreviates $R \cup \{r\}$. A direct type-II conflict is thus a type-II conflict wrt. the empty set of rules. The rule sets R that have to be taken into account in our well-founded semantics based approach are subsets of the rules which are undefeated by the set of literals known to be true. Note that the two types of conflict are not disjoint, i.e. two rules may be in conflict of both type-I and type-II. Consider the following program P_2, a slight modification of P_1:

$n_1 : b \leftarrow not\ \ c, not\ \neg b$
$n_2 : \neg b \leftarrow not\ \ b$
$n_3 : n_2 \prec n_1$

Now we have a type-II conflict between n_1 and n_2 (more precisely, a direct type-II and a type-I conflict) that is not solvable by the implicit mechanisms of well-founded semantics alone. It is this kind of conflict that we try to solve by the explicit preference information. In our example n_2 will be used to derive $\neg b$. Note that now the application of n_2 defeats n_1 and there is no danger that a literal defeating n_2 might become derivable later. Generally, a type-II conflict between r_1 and r_2 (wrt. some undefeated rules of the program) will be solved in favour of the preferred rule, say r_1, only if applying r_1 excludes any further possibility of deriving an r_1-defeating literal.

Note that every type-I conflict can be turned into a direct type-II conflict by a (non-equivalent!) rerepresentation of the rules: if each conflicting rule r

is replaced by its *seminormal form*[14] then all conflicts become type-II conflicts and are thus amenable to conflict resolution through preference information.

After this motivating discussion let us present the new definitions. Our treatment of priorities is based on a weakening of the notion of X-safeness (Definition 4.12). In Sect. 2 we considered a rule r as X-safe whenever there is no proof for a literal defeating r from the monotonic counterparts of X-undefeated rules. Now in the context of a prioritized logic program we will consider a rule r as X-safe if there is no such proof from monotonic counterparts *of a certain subset* of the X-undefeated rules. The subset to be used depends on the rule r and consists of those rules that are not "dominated" by r. Intuitively, r' is dominated by r iff r' is

1. known to be less preferred than r and
2. defeated when r is applied together with rules that already have been established to be X-safe.

(2) is necessary to make sure that explicit preference information is used the right way, according to our discussion of P_1.

It is obvious that whenever there is no proof for a defeating literal from all X-undefeated rules there can be no such proof from a subset of these rules. Rules that were X-safe according to our earlier definition thus remain to be X-safe. Here are the precise definitions:

Definition 5.4 (Dominated Rules)
Let $P = (R, name)$ be a prioritized logic program, X a set of literals, Y a set of rules, and $r \in R$. The set of rules dominated by r wrt. X and Y, denoted $Dom_{X,Y}(r)$, is the set

$$\{r' \in R \mid name(r) \prec name(r') \in X \text{ and } Cl(Y + r) \text{ defeats } r'\}$$

Note that $Dom_{X,Y}(r)$ is monotonic in both X and Y. We can now define the X-safe rules inductively:

Definition 5.5 ($SAFE^{pr}_X(P)$)
Let $P = (R, name)$ be a prioritized logic program, X a set of literals. The set of X-safe rules of P, denoted $SAFE^{pr}_X(P)$, is defined as follows: $SAFE^{pr}_X(P) = \bigcup_{i=0}^{\infty} R_i$, where

$$R_0 = \emptyset, \text{ and for } i > 0,$$
$$R_i = \{r \in R \mid r \text{ not defeated by } Cl(R_X \setminus Dom_{X,R_{i-1}}(r))\}$$

[14] As discussed in Section 4 the seminormal form of $c \leftarrow a_1, \ldots, a_n, not\ b_1, \ldots, not\ b_m$ is

$$c \leftarrow a_1, \ldots, a_n, not\ b_1, \ldots, not\ b_m, not\ c'$$

where c' is the complement of c. The term seminormal is taken from Reiter [Rei80].

Note that X-safeness is obviously monotonic in X. Based on this notion we introduce a new monotonic operator Γ_P^{pr}:

Definition 5.6 (WFS^{pr})
Let $P = (R, name)$ be a prioritized logic program, X a set of literals. The operator Γ_P^{pr} is defined as follows:

$$\Gamma_P^{pr}(X) = Cn(SAFE_X^{pr}(P))$$

As before we define the (prioritized) well-founded conclusions of P, denoted $WFS^{pr}(P)$, as the least fixpoint of Γ_P^{pr}. If a program does not contain preference information at all, i.e., if the symbol $\prec$ does not appear in R, the new semantics coincides with $WFS^\star$ since in that case no rule can dominate another rule. In the general case, since the new definition of X-safeness is weaker than the one used earlier we may have more X-safe rules and for this reason obtain more conclusions than via $\Gamma_P^\star$. The following result is thus obvious:

Lemma 5.7 *Let $P = (R, name)$ be a prioritized logic program. For every set of literals X we have $\Gamma_R^\star(X) \subseteq \Gamma_P^{pr}(X)$.*

¿From this and the monotonicity of both operators it follows immediately that $l \in WFS^\star(R)$ implies $l \in WFS^{pr}(P)$.[15]

As a first simple example let us consider the following program P_3:

$n_1 : b \leftarrow not\ \ c$
$n_2 : c \leftarrow not\ \ b$
$n_3 : n_2 \prec n_1$

We first apply $\Gamma_{P_3}^{pr}$ to the empty set. Besides the instances of the transitivity and anti-symmetry schema that we implicitly assume only n_3 is in $SAFE_\emptyset^{pr}(P_3)$. We thus obtain

$$S_1 = \{n_2 \prec n_1, \neg(n_1 \prec n_2)\}$$

We next apply $\Gamma_{P_3}^{pr}$ to S_1. Since $n_2 \prec n_1 \in S_1$ we have $n_1 \in Dom_{S_1,\emptyset}(n_2)$. $n_2 \in SAFE_{S_1}^{pr}(P_3)$ since $Cl(P_{3_{S_1}} \setminus \{n_1\})$ does not defeat n_2 and we obtain

$$S_2 = \{n_2 \prec n_1, \neg(n_1 \prec n_2), c\}$$

Further iteration of $\Gamma_{P_3}^{pr}$ yields no new literals, i.e. S_2 is the least fixpoint. Note that c is not a conclusion under the original well-founded semantics.

We next show that the programs P_1 and P_2 discussed earlier are handled as intended. Here is P_1:

[15]To model the coherence principle of Pereira and Alferes [PA92] in our approach one would have to weaken the notion of X-safeness even further. In the inductive definition, a rule r would have to be considered a member of R_i whenever for each weak precondition *not* b of r

- $b \notin Cl(R_X \setminus Dom_{X,R_{i-1}}(r))$, or
- $b' \in X$, where $b' = \neg b$ if b is an atom and a if $b = \neg a$.

$n_1 : b \leftarrow not\ \ c$
$n_2 : \neg b \leftarrow not\ \ b$
$n_3 : n_2 \prec n_1$

Since $\gamma^{\star}_{P_1}(\emptyset)$ does not defeat n_1 this rule is safe from the beginning, i.e., $n_1 \in SAFE^{pr}_{\emptyset}(P_1)$. $\Gamma^{pr}_{P}(\emptyset)$ yields

$$\{n_2 \prec n_1, \neg(n_1 \prec n_2), b\}$$

which is also the least fixpoint. The explicit preference does not interfere with the implicit one, as intended.

The situation changes in P_2 where the first rule in P_1 is replaced by

$n_1 : b \leftarrow not\ \ c, not\ \neg b$

The new rule n_1 is not in $SAFE^{pr}_{\emptyset}(P_2)$ since it is defeated by the consequence of n_2 and n_2 is not dominated by n_1. $\Gamma^{pr}_{P_2}(\emptyset)$ yields

$$S_1 = \{n_2 \prec n_1, \neg(n_1 \prec n_2)\}$$

Now $n_2 \in SAFE^{pr}_{S_1}(P_2)$ since n_2 dominates n_1 wrt. S_1 and the empty set of rules. We thus conclude $\neg b$ as intended. The least fixpoint is

$$S_2 = \{n_2 \prec n_1, \neg(n_1 \prec n_2), b\}$$

It also applies to well-founded semantics for extended logic programs since for the computation of the least fixed point of Γ_P respectively $\Gamma^{\star}_P$ the complementary literals l and $\neg l$ can be viewed as two distinct atoms.

For the complexity analysis of our prioritized approach let n be the number of rules in a prioritized program $P = (R, name)$. A straightforward implementation would model the application of Γ^{pr}_P in an outer loop and the computation of $SAFE^{pr}_X$ in an inner loop. Fortunately, we can combine the two loops into a single loop whose body is executed at most n times. The reason is that $SAFE^{pr}_X$ grows monotonically with X and Γ^{pr}_P grows monotonically with $SAFE^{pr}_X$. Here is a nondeterministic algorithm for computing the least fixed point of Γ^{pr}_P:

```
**Procedure** WFS$^{pr}$
**Input:** A prioritized logic program $P = (R, name)$ with $|R| = n$
**Output:** the least fixed point of $\Gamma^{pr}_P$
$S_0 := \emptyset$;
$R_0 := \emptyset$;
for $i = 1$ to $n$ do

      if there is a rule $r \in R_{S_{i-1}} \setminus R_{i-1}$ such that
      $Cl(R_{S_{i-1}} \setminus Dom_{S_{i-1},R_{i-1}}(r))$ does not defeat $r$
      then $R_i := R_{i-1} + r; S_i := Cn(R_i)$
      else return $S_{i-1}$
```

```
    endfor
    end WFS^pr
```

In each step S_i and R_i denote the well-founded conclusions, respectively safe rules established so far. The body of the for-loop is executed at most n times and there are at most n rules that have to be checked for satisfaction of the if-condition. The if-condition itself can, according to the results of Dowling and Gallier, be checked in linear time: we need to establish $Dom_{S_{i-1},R_{i-1}}(r)$ which involves the computation of a minimal model of the monotonic counterparts of $R_{i-1} + r$. We then have to eliminate the rules dominated by r form $R_{S_{i-1}}$ and compute another minimal model to see whether r is defeated. This leads to an overall time complexity of $O(n^3)$.

5.3 A Legal Reasoning Example

In this section we want to show that the additional expressiveness provided by our approach actually helps representing real world problems. We will use an example first discussed by Gordon [Gor93]. We somewhat simplified it for our purposes.

Example 5.8 (Legal Reasoning)
Assume a person wants to find out if her security interest in a certain ship is perfected. She currently has possession of the ship. According to the Uniform Commercial Code (UCC, §9-305) a security interest in goods may be perfected by taking possession of the collateral. However, there is a federal law called the Ship Mortgage Act (SMA) according to which a security interest in a ship may only be perfected by filing a financing statement. Such a statement has not been filed. Now the question is whether the UCC or the SMA takes precedence in this case. There are two known legal principles for resolving conflicts of this kind. The principle of Lex Posterior *gives precedence to newer laws. In our case the UCC is newer than the SMA. On the other hand, the principle of* Lex Superior *gives precedence to laws supported by the higher authority. In our case the SMA has higher authority since it is federal law.*

The available information can nicely be represented in our approach. To make the example somewhat shorter we use the notation

$$c \Leftarrow a_1, \ldots, a_n, not \ \ b_1, \ldots, not \ \ b_m$$

as an abbreviation for the rule

$$c \leftarrow a_1, \ldots, a_n, not \ \ b_1, \ldots, not \ \ b_m, not \ \ c'$$

where c' is the complement of c, i.e. $\neg c$ if c is an atom and a if $c = \neg a$. Such rules thus correspond to semi-normal or, if $m = 0$, normal defaults in Reiter's default logic [Rei80].

We use the ground instances of the following named rules to represent the relevant article of the UCC, the SMA, Lex Posterior (LP), and Lex Superior (LS). The symbols d_1 and d_2 are parameters for rule names:

$UCC : perfected \Leftarrow possession$
$SMA : \neg perfected \Leftarrow ship, \neg fin\text{-}statement$
$LP(d_1, d_2) : d_1 \prec d_2 \Leftarrow more\text{-}recent(d_1, d_2)$
$LS(d_1, d_2) : d_1 \prec d_2 \Leftarrow fed\text{-}law(d_1), state\text{-}law(d_2)$

The following facts are known about the case and are represented as rules without body (and without name):

$possession$
$ship$
$\neg fin\text{-}statement$
$more\text{-}recent(UCC, SMA)$
$fed\text{-}law(SMA)$
$state\text{-}law(UCC)$

Let's call the above set of literals H. Iterated application of Γ_P^{pr} yields the following sequence of literal sets (in each case $S_i = (\Gamma_P^{pr})^i(\emptyset)$):

$$\begin{array}{rcl} S_1 & = & H \\ S_2 & = & S_1 \end{array}$$

The iteration produces no new results besides the facts already contained in the program. The reason is that UCC and SMA block each other, and that no preference information is produced since also the relevant instances of Lex Posterior and Lex Superior block each other. The situation changes if we add information telling us how conflicts between the latter two are to be resolved. Assume we add the following information:[16]

$$LS(SMA, UCC) \prec LP(UCC, SMA)$$

Now we obtain the following sequence:

$$\begin{array}{rcccl} S_1 & = & H & \cup & \{LS(SMA, UCC) \prec LP(UCC, SMA), \\ & & & & \neg LP(UCC, SMA) \prec LS(SMA, UCC)\} \\ S_2 & = & S_1 & \cup & \{SMA \prec UCC, \neg UCC \prec SMA\} \\ S_3 & = & S_2 & \cup & \{\neg perfected\} \\ S_4 & = & S_3 & & \end{array}$$

This example nicely illustrates how in our approach conflict resolution strategies can be specified declaratively, by simply asserting relevant preferences among the involved conflicting rules.

[16]In realistic settings one would again use a schema here. In order to keep the example simple we use the relevant instance of the schema directly.

6 Adding Disjunction

In this section we will extend our programs to *disjunctive* statements. In Knowledge Representation it often occurs that we know $A \vee B \vee C$ without being sure which of these propositions hold. In fact, such a disjunction leaves it open: there might be states in the world where A holds or B or C or any combination thereof. Nevertheless, we can have information that A implies D and B implies D and C implies D from which we would like to derive that D holds for sure. We will see in Section 6.5 that even with disjunctive programs without negation we can already express relations which belong to the second level of the polynomial hierarchy.

Concerning the right semantics for such programs, we are in the same situation as in Section 3 — for positive programs there is general agreement while for disjunctive programs with default-negation there exist several competing approaches.

We present in Section 6.1 the generalized closed world assumption introduced by Minker. In Section 6.2 we show that our definition of WFS from Section 3.3 immediately carries over to the disjunctive case. The original definition of STABLE (Definition 3.16) also carries over — we present it in Section 6.3. We mention some other attempts to define disjunctive semantics in Section 6.4. Finally we discuss complexity and expressibility in Section 6.5.

6.1 GCWA

GCWA was defined by Minker ([Min82]) and can bee seen as a refined version of the CWA introduced by Reiter ([Rei78]):

Definition 6.1 (CWA)

$$\mathrm{CWA(DB)} = \mathrm{DB} \cup \{\neg P(t) : \ \mathrm{DB} \not\models P(t)\},$$

where $P(t)$ is a ground predicate instance.

That is, if a ground term cannot be inferred from the database, its negation is added to the closure. A weakness of CWA is that already for very simple theories, like $A \vee B$ it is inconsistent. Since neither $\neg A$ nor $\neg B$ is derivable, we have to add them both which makes the whole set inconsistent.

GCWA is defined for positive disjunctive programs consisting of rules of the form

$$A_1 \vee \ldots \vee A_n \ \leftarrow \ B_1, \ldots, B_m$$

by declaring all the minimal models to be the intended ones:

Definition 6.2 (GCWA)
The generalized closed world assumption GCWA of P is the semantics given by the set of all minimal Herbrand models of P:

$$GCWA(P) := \mathit{Min\text{-}MOD}(P)$$

Originally, Minker denoted by GCWA(P) a set of negative atoms with the property that $P \cup GCWA(P) \models$ *not* A if and only if MinMOD(P)$\models$ *not* A but we prefer here to denote by GCWA a semantics in the sense of Definition 2.13.

GCWA is very important because it plays the same role for positive disjunctive programs as the least Herbrand model M_P does for definite programs. In addition it turns out that some semantics SEM defined for arbitrary disjunctive programs (i. e. with default-negation) can be characterized, sometimes even implemented, by reducing them to positive programs and then applying recursively GCWA. Thus an appropriate procedure iterating GCWA can "implement" such semantics SEM.

Note also that as far as we consider deriving ***positive*** disjunctions, we stay entirely within classical logic — a positive disjunction is true in GCWA if and only if it follows from the program considered as a classical theory. Therefore this task can be accomplished be methods and techniques developed in theorem proving in the last 30 years. In fact this was one of the main starting points of the DisLoP-project in Koblenz (see Section 7.2). As an example, in [BFS95] the authors show how to *compute* definite answers from a positive disjunctive program using an underlying theorem prover. While indefinite answers are easy to obtain (because most of them are trivial) definite ones are much harder to obtain.

Of course, GCWA is nothing else than Circumscription (see Section A.4) for a special class of theories. Methods developed for CIRC can be used to compute GCWA. For recent approaches that work in *polynomial space* see [Nie96a, Nie96b].

In Sections 2 and 3 we have introduced the general notion of a semantics and various principles. Do they carry over to the disjunctive case? Fortunately, the answer is yes. In addition, GCWA not only satisfies all these properties, it is also uniquely characterized by them as the next theorem shows (we will introduce these properties in the next section).

Theorem 6.3 (Characterization of GCWA)
Let SEM be a semantics satisfying GPPE *and* Elimination of Tautologies.

a) *Then: SEM(P)* $\subseteq$ *Min-MOD*$_{2-val}$*(P) for positive disj. programs P.*

   I.e. any such semantics is already based on 2-valued minimal models. In particular, GCWA is the weakest semantics with these properties.

b) *If SEM is non-trivial and satisfies in addition*[17] Isomorphy *and* Relevance, *then it coincides with GCWA on positive disjunctive programs.*

We end this section with the discussion of a well-known example that can not be handled adequately by Circumscription:

[17]See Section 7.1 for the precise definitions of Relevance and Isomorphy.

Example 6.4 (Poole's Broken Arm)
Usually, a person's left arm is useable. But if the left arm is broken, it is an exception. The same statement holds for the right arm. Suppose that we saw Fred yesterday with a broken arm but we do not remember if it was the left or the right one. We also know that Fred can make out a cheque if he has at least one useable arm (he is ambidextrous) but that he is completely disabled if both arms are broken. Here is the natural formalization:

$$\begin{array}{lcl}
left\_use(x) & \leftarrow & \text{not } ab(left, x) \\
ab(left, x) & \leftarrow & left\_brok(x) \\
right\_use(x) & \leftarrow & \text{not } ab(right, x) \\
ab(right, x) & \leftarrow & right\_brok(x) \\
left\_brok(Fred) \vee right\_brok(Fred) & \leftarrow & \\
make\_cheque(x) & \leftarrow & left\_use(x) \\
make\_cheque(x) & \leftarrow & right\_use(x) \\
disabled(x) & \leftarrow & left\_brok(x), right\_brok(x)
\end{array}$$

Of course, we expect that Fred is able to make out a cheque even without knowing which arm he is actually using. Also we derive that he is not (completely) disabled.

For general Circumscription, the problem is to rule out the unintended model where both arms are broken and Fred is disabled. As we will see later, both D-WFS and DSTABLE derive that Fred is not disabled but only DSTABLE is strong enough to also conclude that Fred can make out a cheque.

6.2 D-WFS

Before we can state the definition of D-WFS we have to extend our principles to disjunctive programs with default-negation. We abbreviate general rules

$$A_1 \vee \ldots \vee A_k \leftarrow B_1, \ldots, B_m, not\ C_1, \ldots, not\ C_n,$$

by

$$\mathcal{A} \leftarrow \mathcal{B}^+, not\ \mathcal{B}^-$$

where $\mathcal{A} := \{A_1, \ldots, A_k\}$, $\mathcal{B}^+ := \{B_1, \ldots, B_m\}$, $\mathcal{B}^- := \{C_1, \ldots, C_n\}$. We also generalize our notion of a semantics slightly:

Definition 6.5 (Operator $\mid\!\sim$, Semantics $\mathcal{S}_{\mid\sim}$)
By a semantic operator $\mid\!\sim$ we mean a binary relation between logic programs and pure disjunctions which satisfies the following three arguably obvious conditions:

1. Right Weakening: *If $P \mid\!\sim \psi$ and $\psi \subseteq \psi'$*[18]*, then $P \mid\!\sim \psi'$.*
2. Necessarily True: *If $\mathcal{A} \leftarrow true \in P$ for a disjunction $\mathcal{A}$, then $P \mid\!\sim \mathcal{A}$.*

[18] I. e. ψ is a subdisjunction of ψ'.

3. Necessarily False: *If $A \notin Head\_atoms(P)$*[19] *for some $\mathcal{L}$-ground atom A, then $P \mathrel{|\!\!\sim} \text{not } A$.*

Given such an operator $\mathrel{|\!\!\sim}$ and a logic program P, by the semantics $\mathcal{S}_{\mathrel{|\!\sim}}(P)$ of P determined by $\mathrel{|\!\!\sim}$ we mean the set of all pure disjunctions derivable by $\mathrel{|\!\!\sim}$ from P, i.e., $\mathcal{S}_{\mathrel{|\!\sim}}(P) := \{\psi \mid P \mathrel{|\!\!\sim} \psi\}$.

In order to give a unified treatment in the sequel, we introduce the following notion:

Definition 6.6 (Invariance of $\mathrel{|\!\!\sim}$ under a Transformation)
Suppose that a program transformation Trans : $P \mapsto \text{Trans}(P)$ *mapping logic programs into logic programs is given. We say that the operator $\mathrel{|\!\!\sim}$ is invariant under* Trans *(or that* Trans *is a $\mathrel{|\!\!\sim}$-equivalence transformation) iff*

$$P \mathrel{|\!\!\sim} \psi \iff \text{Trans}(P) \mathrel{|\!\!\sim} \psi$$

for any pure disjunction ψ and any program P.

All our principles introduced below can now be naturally extended.

Definition 6.7 (Elimination of Tautologies, Non-Minimal Rules)
Semantics $\mathcal{S}_{\mathrel{|\!\sim}}$ satisfies **a)** *the* Elimination of Tautologies, *resp.* **b)** *the* Elimination of Non-Minimal Rules *iff $\mathrel{|\!\!\sim}$ is invariant under the following transformations:*

a) *Delete a rule $\mathcal{A} \leftarrow \mathcal{B}^+ \wedge \text{not } \mathcal{B}^-$ with $\mathcal{A} \cap \mathcal{B}^+ \neq \emptyset$.*

b) *Delete a rule $\mathcal{A} \leftarrow \mathcal{B}^+ \wedge \text{not } \mathcal{B}^-$ if there is another rule $\mathcal{A}' \leftarrow \mathcal{B}^{+\prime} \wedge \text{not } \mathcal{B}^{-\prime}$ with $\mathcal{A}' \subseteq \mathcal{A}$, $\mathcal{B}^{+\prime} \subseteq \mathcal{B}^+$, and $\mathcal{B}^{-\prime} \subseteq \mathcal{B}^-$.*

Our partial evaluation principle has now to take into account disjunctive heads. The following definition was introduced independently by Sakama/Seki and Brass/Dix ([BD95d, SS95]):

Definition 6.8 (GPPE)
Semantics $\mathcal{S}_{\mathrel{|\!\sim}}$ satisfies GPPE iff it is invariant under the following transformation: Replace a rule $\mathcal{A} \leftarrow \mathcal{B}^+ \wedge \textit{not } \mathcal{B}^-$ where $\mathcal{B}^+$ contains a distinguished atom B by the rules

$$\mathcal{A} \cup (\mathcal{A}_i \setminus \{B\}) \;\leftarrow\; (\mathcal{B}^+ \setminus \{B\}) \cup \mathcal{B}_i^+ \;\wedge\; \textit{not } (\mathcal{B}^- \cup \mathcal{B}_i^-) \;\; (i = 1, \ldots, n)$$

where $\mathcal{A}_i \leftarrow \mathcal{B}_i^+ \wedge \text{not } \mathcal{B}_i^-$ $(i = 1, \ldots, n)$ are all the rules with $B \in \mathcal{A}_i$.

[19]We denote by *$Head\_atoms(P)$* the set of all (instantiations of) atoms ocurring in some rule-head of P.

62

2.

Give
actio
f, r

In t
nitio

Defi
Supp
prope
whe

Lem

Alte

Defi
Beca
suffi
there

(a)

(b)

Our p
The
Prod

Defi
Tem
num
stro

whe

Two
bala

Note that we are free to select a specific positive occurrence of an atom B and then perform the transformation. The new rules are obtained by replacing B by the bodies of all rules r with head literal B and adding the remaining head atoms of r to the head of the new rule.

Here is the analogue of Principle 3.6:

Definition 6.9 (Positive and Negative Reduction)
Semantics $\mathcal{S}_{\mid\!\sim}$ satisfies a) Positive, *resp.* b) Negative *Reduction iff $\mid\!\sim$ is invariant under the following transformations:*

a) *Replace a rule $\mathcal{A} \leftarrow \mathcal{B}^+ \wedge \text{not } \mathcal{B}^-$ by $\mathcal{A} \leftarrow \mathcal{B}^+ \wedge \text{not } (\mathcal{B}^- \cap Head\_atoms(P))$.*

b) *Delete a rule $\mathcal{A} \leftarrow \mathcal{B}^+ \wedge \text{not } \mathcal{B}^-$ if there is a rule $\mathcal{A}' \leftarrow true$ with $\mathcal{A}' \subseteq \mathcal{B}^-$.*

Now the definition of a disjunctive counterpart of WFS is straightforward:

Definition 6.10 (D-WFS)
There exists the weakest semantics satisfying positive and negative Reduction, GPPE, Elimination of Tautologies and non-minimal Rules. We call this semantics D-WFS.

As it was the case for WFS, our calculus of transformations is also confluent ([BD95c, BD96]).

Theorem 6.11 (Confluent Calculus for D-WFS)
The calculus consisting of our four transformations is confluent and terminating for propositional programs. I.e. we always arrive at an irreducible program, which is uniquely determined. The order of the transformations does not matter.

Therefore any program P is associated a unique normalform $res(P)$. The disjunctive wellfounded semantics of P can be read off from $res(P)$ as follows

$$\psi \in \textit{D-WFS}(P) \iff \begin{array}{l} \textit{there is } \mathcal{A} \subseteq \psi \textit{ with } \mathcal{A} \leftarrow true \in res(P) \textit{ or} \\ \textit{there is } \text{not } A \in \psi \textit{ and } A \notin Head\_atoms(res(P)). \end{array}$$

Note that the original definition of WFS, or any of its equivalent characterizations, does not carry over to disjunctive programs in a natural way.

Let us see how Example 6.4 is handled by D-WFS. Applying GPPE and Reduction gives us the following residual program (we consider just the *Fred*-instantiations):

$$\begin{array}{lll}
left\_use(F) & \leftarrow & not\ ab(left, F) \\
ab(left, F) \vee right\_brok(F) & \leftarrow & \\
right\_use(F) & \leftarrow & not\ ab(right, F) \\
ab(right, F) \vee left\_brok(F) & \leftarrow & \\
left\_brok(F) \vee right\_brok(F) & \leftarrow & \\
make\_cheque(F) & \leftarrow & not\ ab(left, F) \\
make\_cheque(F) & \leftarrow & not\ ab(right, F)
\end{array}$$

Therefore we derive *not disabled*(F), because it does not appear in any head of the residual program. All the remaining atoms are undefined.

Two properties of D-WFS are worth noticing

- For positive disjunctive programs, D-WFS coincides with GCWA.
- For non-disjunctive programs with negation, D-WFS coincides with WFS.

6.3 DSTABLE

Unlike the wellfounded semantics, the original definition of stable models carries over to disjunctive programs quite easily:

Definition 6.12 (DSTABLE)
N is called a stable *model*[20] *of P iff $N \in$ Min-Mod(P^N).*

In the last definition P^N is the positive disjunctive program obtained from P by applying the Gelfond/Lifschitz transformation (as introduced before Definition 3.16 — its generalization to disjunctive programs is obvious).

Analogously to D-WFS the following two properties of DSTABLE hold:

- For positive disjunctive programs, DSTABLE coincides with GCWA.
- For non-disjunctive programs with negation, DSTABLE coincides with STABLE.

What about our transformations introduced to define D-WFS? Do they hold for DSTABLE? Yes, they are indeed true. The most difficult proof is the one for GPPE. It was proved in [BD95d, SS95] independently that stable models are preserved under GPPE. Moreover, Brass/Dix proved in [BD95b] that STABLE can be almost uniquely determined by GPPE:

Theorem 6.13 (Characterization of DSTABLE)
Let SEM be a semantics satisfying GPPE, Elimination of Tautologies, *and* Elimination of Contradictions. *Then: SEM$(P) \subseteq$ STABLE(P).*

Moreover, DSTABLE is the weakest semantics satisfying these properties.

DSTABLE is stronger than D-WFS as can be seen from Example 6.4. There we have exactly two stable models

1. $left\_use(F)$, $not\ ab(left, F)$, $ab(right, F)$, $not\ right\_use(F)$, $left\_brok(F)$, $not\ right\_brok(F)$, $make\_cheque(F)$, and $not\ disabled(F)$,
2. $right\_use(F)$, $not\ ab(right, F)$, $ab(left, F)$, $not\ left\_use(F)$, $right\_brok(F)$, $not\ left\_brok(F)$, $make\_cheque(F)$, and $not\ disabled(F)$.

In all of them, Fred is not disabled and can make out a cheque.

Of course, DSTABLE inherits the shortcomings of STABLE such as inconsistency and no goal-orientedness.

[20]Note that we only consider Herbrand models.

6.4 Other Semantics

In this section we just want to mention some other disjunctive semantics proposed in the last years. First, there are semantics differing from GCWA in that they interpret "∨" inclusively, rather than exclusively (like GCWA does).

The corresponding semantics is called WGCWA (see [RLM89]) and is equivalent to the disjunctive database rule DDR considered in [RT88]. WGCWA has been considered as a more tractable (weaker) variant of GCWA (from the procedural point of view developed in [RLM89]).

Example 6.14 (Inclusive versus Exclusive)

$$\begin{array}{lll} P_{incl/excl}: & a \vee b & \\ & c & \leftarrow \quad a, b \end{array}$$

Under an exclusive *interpretation,* not c *should be derivable. Indeed, we have* $GCWA(P_{incl/excl}) = \{\text{not } c\}$.
Under an inclusive *interpretation however,* not c *should not be derivable. This is the case for WGCWA:* $M_{P^{*}_{incl/excl}} = \{a, b, c\}$. *The set of* positive *derivable literals is in both cases the same! If we replace the first clause with a or b, then* not c *is derivable.*

There are extensions of WGCWA to disjunctive programs with negation: [SI93, Ros89, Dix92b, DM94a, Sak89].

There is also the book [LMR92] — the first in-depth-study of disjunctive semantics with negation. However, we feel that these semantics have a drawback in that they are based on rather technical, complicated and not-easy-to-understand fixpoint definitions. These definitions leave a lot of room for modifications. But small modifications usually have a tremendous impact on the outcoming semantics. In addition these semantics do not allow for a proper treatment of definitional extensions (see Example 7.4).

Other approaches are due to Przymusinski: *stationary*-semantics $\mathcal{STN}$ defined in [Prz91a] and *static*-semantics STATIC defined in [Prz95, BDP96]. STATIC is an improvement of his former stationary semantics that is very close to D-WFS: in fact it coincides with D-WFS if it is restricted to a common sublanguage ([BDNP96]). This approach also allows us to consider a larger class of programs, namely those that contain *not* $(A_1 \wedge \ldots \wedge A_n)$ in their bodies. Such programs are more expressible and therefore turn out to be even better suited for representation tasks.

Another approach differing from GCWA and WGCWA is considered in [DGM94, DGM96, Bon93].

	Complexity	
	1. ord. prog. *(with functions)*	*prop. prog.* *(no variables)*
GCWA *(P is positive)*	*A*: Σ_1^0-compl. *not A*: Π_2^0-compl.	*A*: co-NP-compl. *not A*: Π_2^P-compl.
WGCWA *(P is positive)*	*A*: Σ_1^0-compl. *not A*: Π_1^0-compl.	*A*: co-NP-compl. *not A*: linear in $\|P\|$
PERFECT *(P is stratified)*	arithm.-compl.	Π_2^P-compl.
WPERFECT *(P is stratified)*	arithm.-compl.	Π_2^P-compl.
DSTABLE	Π_1^1-compl. over $\mathbb{N}$	Π_2^P-compl.

Table 4: Complexity of Disjunctive Semantics

6.5 Complexity and Expressibility

From the complexity point of view GCWA lies between CWA (which is Π_1^0-complete, see [AB90] and general Circumscription (Σ_1^1-complete, see [CEG92]): GCWA is Π_2^0-complete. For propositional programs we have to distinguish between deriving an *atom* or a *literal*. The first problem is co-NP-complete while the second is even Π_2^P-complete (see [Imi91]).

For deriving negated literals *not A*, WGCWA is Π_1^0-complete (like CWA) and therefore "better" than GCWA (Π_2^0-complete). In the propositional case, WGCWA is polynomial while GCWA is Π_2^p-complete (both for the derivation of literals of the form *not A*).

	Expressibility *1. ord. prog.* *(no functions)*
GCWA *(P is positive)*	$\subset \Pi_2^P$
WGCWA *(P is positive)*	$\subset \Pi_2^P$
PERFECT *(P is stratified)*	$= \Pi_2^P$
WPERFECT *(P is stratified)*	$= \Pi_2^P$
DSTABLE	$= \Pi_2^P$

Table 5: Expressibility of Disjunctive Semantics

7 What Do We Want and What Is Implemented?

In this part we first consider the question *Is there an optimal semantics?* (Section 7.1) and give in Section 7.2 an overview of all the existing implementations we are aware of. We also describe theoretical approaches that have not yet been implemented.

7.1 What is the Best Semantics?

Most probably there is no definite answer to the question in the title. Different knowledge representation tasks may ask for different semantics. Some might be better suited in special domains than others. What are reasonable properties that semantics should be checked against?

While many people defined in the last years new semantics by considering only few examples and appealing to their own personal intuitions they had about how these few examples should be handled, Dix tried to adjust and investigate abstract properties known in general nonmonotonic reasoning to semantics of logic programs ([Dix91, Dix92b, Dix95a, Dix95b]). He showed for example that WFS is cumulative and rational and that a semantics defined independently by Schlipf and Dix is the weakest extension of WFS satisfying *Cut* and *Supraclassicality*. Figure 3 illustrates the properties and the relationship between many semantics. In Figure 4 normal programs are considered.

Besides such properties (which he calls *strong*) he defined also *weak* properties — these are conditions that any reasonable semantics should satisfy ([Dix92a, Dix95b]). The principles we have introduced in Sections 2, 3 belong to this sort. Let us take a closer look into some weak properties already mentioned (but not yet defined). We start with a property that is satisfied for any semantics we know:

Definition 7.1 (Isomorphy)
A semantics SEM satisfies Isomorphy, iff

$$SEM(\mathcal{I}(P)) = \mathcal{I}(SEM(P))$$

for all programs P and isomorphisms $\mathcal{I}$ on the Herbrand base B_P.

Isomorphy formalizes the intuition that a renaming of the program should have no influence on the semantics, as long as we also apply this same renaming to the semantics.

The next property gives a formal definition of the notion *Goal-Orientedness*. To state this conditions, we need the notion of the Dependency-Graph (Definition 3.5) and the two definitions

- $dependencies\_of(X) := \{A : X \text{ depends on } A\}$, and
- $rel\_rul(P, X)$ is the set of *relevant rules* of P with respect to X, i.e. the set of rules that contain an $A \in dependencies\_of(X)$ in their head.

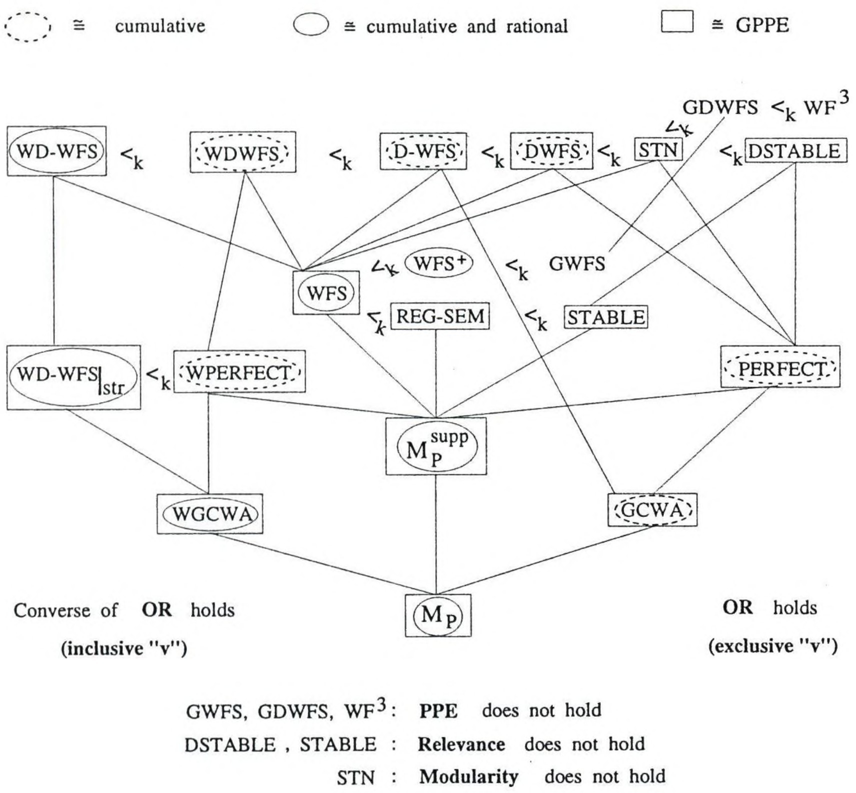

≅ cumulative
≅ cumulative and rational
≅ GPPE
GDWFS <k WF3
WD-WFS <k WDWFS <k D-WFS <k DWFS <k STN <k DSTABLE
WFS+
GWFS
WFS
REG-SEM
STABLE
WD-WFS|str <k WPERFECT
PERFECT
M_P^supp
WGCWA
GCWA
M_P
Converse of OR holds (inclusive "v")
OR holds (exclusive "v")
GWFS, GDWFS, WF3: PPE does not hold
DSTABLE, STABLE: Relevance does not hold
STN: Modularity does not hold

Figure 3: Semantics for Disjunctive Programs

Given any semantics SEM and a program P, it is perfectly reasonable that the truthvalue of a literal L, with respect to SEM(P), only depends on the subprogram formed from the *relevant rules* of P with respect to L.[21] This idea is formalized by:

Definition 7.2 (Relevance)
The principle of Relevance states: $L \in SEM(P)$ *iff* $L \in SEM(rel\_rul(P, L))$.

Note that the set of relevant rules of a program P with respect to a literal L contains all rules, that could ever contribute to L's derivation (or to its nonderivability). In general, L depends on a large set of atoms: $dependencies\_of(L) := \{A : L \text{ depends on } A\}$. But rules that do not contain these atoms in their heads,

[21]Let $dependencies\_of(not\ X) := dependencies\_of(X)$, and $rel\_rul(P, not\ X) := rel\_rul(P, X)$.

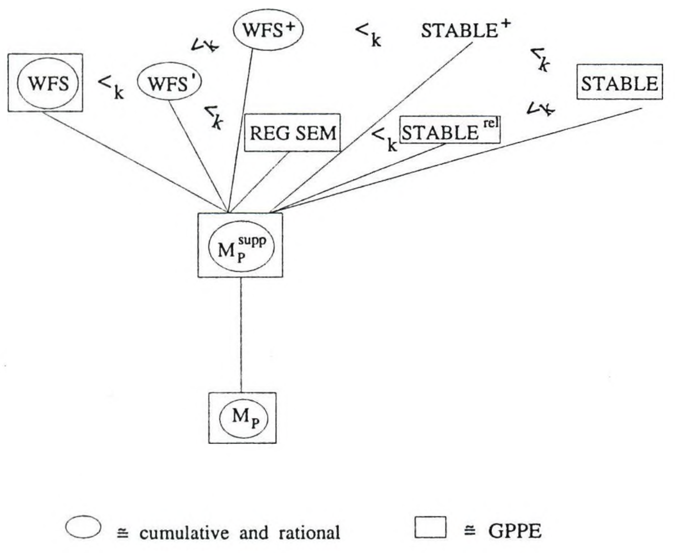


Figure 4: Semantics for Normal Programs

will never contribute to their derivation or non-derivation. Therefore, these rules should not affect the meaning of L in P. STABLE does not satisfy this principle. This is due to the nonexistence of stable models by adding a clause "$c \leftarrow$ *not* c" to a program.

We have already introduced GPPE above. It is an extension of the following property for non-disjunctive programs:

Definition 7.3 (PPE)
Let P be an instantiated program and let the atom c occur positively in P. Let $c \leftarrow rhs_1, \ \ldots\ , c \leftarrow rhs_n$ be all the rules of P with c in their heads.

Any program clause of the form "$head \leftarrow c, body$" can be replaced by the rules

$$\begin{array}{rcl} head & \leftarrow & rhs_1, body \\ & \vdots & \\ head & \leftarrow & rhs_n, body \end{array}$$

Note that the rules $c \leftarrow rhs_1, \ \ldots\ c \leftarrow rhs_n$ are not removed (in contrast to the weak version of PPE). We call the program obtained in this way P'.

The principle of partial evaluation is: $SEM(P') = SEM(P)$.

GPPE is obtained from PPE by weakening the assumption that c *only occurs positively*. We note that most semantics defined by Minker and his group do not satisfy this condition:

Example 7.4 (Extension-by-Definition)
We consider the following two programs:

$$P_{GWFS}: \begin{array}{lcl} p & \leftarrow & \text{not } b \\ a & \leftarrow & \text{not } b \\ b & \leftarrow & c \\ c & \leftarrow & p, \text{not } a \end{array} \qquad P_{GWFSc}: \begin{array}{lcl} p & \leftarrow & \text{not } b \\ a & \leftarrow & \text{not } b \\ b & \leftarrow & p, \text{not } a \end{array}$$

GWFS(P_{GWFS}) entails not c, *because Min-MOD(P_{GWFS})* $= \{ \{p,a\}, \{b\} \}$ *and thus also (by simple negation-as-failure reasoning)* not b, p *and* a. *But Min-MOD($P_{GWFS} \cup \{p\}$)=$\{ \{p,a\}, \{p,c,b\} \}$, thus GWFS($P_{GWFS} \cup \{p\}$) does not entail* not c.

P_{GWFSc} *partial evaluates* P_{GWFS}: the last two clauses were amalgamated in one single clause. Obviously, a semantics should assign the same meaning to these programs: unfortunately GWFS does not!

The next principle, *Modularity*, has some similarities with PPE. It enables us to compute a semantics by modularizing it into certain "subprograms" (formed of the relevant rules). The semantics of these modules can be computed first and the semantics of the whole program can be determined by reducing this program with literals that were already determined.

Definition 7.5 (Modularity)
Let $P = P_1 \cup P_2$ and for every $A \in B_{P_2}$: $rel\_rul(P,A) \subseteq P_2$.
The principle of Modularity is: $SEM(P) = SEM({P_1}^{SEM(P_2)} \cup P_2)$.

To illustrate this property, we compare the program

$$P_1: \begin{array}{lcl} b & \leftarrow & y \\ y \vee x & & \\ z \vee y & & \\ m & \leftarrow & x, z, b \\ y & \leftarrow & not\ m \end{array}$$

with the union of the following two programs

$$P_1': \begin{array}{lcl} b & \leftarrow & y \\ y \vee x & & a \\ z \vee y & & a \\ m & \leftarrow & x, z, b \\ y & \leftarrow & not\ m \end{array} \qquad P_2: \begin{array}{lcl} a & \leftarrow & e, not\ g \\ a & \leftarrow & f, not\ g \\ a & \leftarrow & f, not\ e \\ a & \leftarrow & g, not\ e \\ e \vee f \vee g & & \end{array}$$

Properties of Logic-Programming Semantics							
Semantics	Reference	Domain	Taut.	GPPE	Red.	Non-Min.	Rel.
comp	[Cla78]	Nondis.	—	•	•	•	—
GCWA	[Min82]	Pos.	•	•	•	•	•
WGCWA	[RT88]	Pos.	—	•	•	—	•
DSTABLE	[GL91]	Dis.	•	•	•	•	—
WFS	[vGRS91]	Nondis.	•	•	•	•	•
$\mathcal{STN}$	[Prz91b]	Dis.	•	•	•	•	•
STATIC	[Prz95]	Dis.	•	•	•	•	•
D-WFS	[BD95d]	Dis.	•	•	•	•	•
DWFS	[Dix92b]	Dis.	•	•	•	•	•
Strong WFS	[Ros92]	Dis.	—	—	•	—	•
WD-WFS	[BD95d]	Dis.	—	•	•	—	•
WDWFS	[Dix92b]	Dis.	—	•	•	—	•

Table 6: Semantics and Their Equivalence-Transformations

P_2 is a stratified program and $\mathcal{STN}$ derives a. Concerning P_1, different intuitions seem possible. One can argue, that *not* m should be derivable, since the only way to derive m is by using the fourth clause, which means deriving b, which means deriving y which excludes deriving x or z. This is the way, P_1 is handled by the first version of $\mathcal{STN}$. The second (final) version $\mathcal{STN}$ does not derive *not* m. But if we apply $\mathcal{STN}$ to $P_1' \cup P_2$, then *not* m is derivable. This shows that *weak Modularity* is not satisfied: we consider this to be a serious shortcoming.

Typical results of Dix are

- WFS is the weakest semantics satisfying some of these weak properties,
- WFS can be uniquely characterized if some strong properties are added.

We conclude with Table 6: an overview of the properties of some semantics mentioned above.

7.2 Query-Answering Systems and Implementations

In this section we give a rough overview of what semantics have been implemented so far and where they are available. As already explained in Sections 3.5, 6.5, our NMR-semantics are undecidable in general. Nevertheless we think it is very important to have running systems that

1. can handle programs with free variables, and
2. are Goal-Oriented.

To ensure *completeness* (or *termination*) we need then additional requirements like *allowedness* (to prevent floundering, see Section 3.1) and no function symbols.

Although these restrictions ensure the Herbrand-universe to be finite (and thus we are really considering a propositional theory) we think that such a system has great advantages over a system that can just handle ground programs. For a language $\mathcal{L}$, the fully instantiated program can be quite large and difficult to handle effectively.

The goal-orientedness (or *Relevance* as introduced in Section 7.1) is also important — after all this was one reason of the success of SLD-Resolution. As noted above, such a goal-oriented approach is not possible for STABLE.

LP-Semantics

Various commercial PROLOG-systems perform variants of SLDNF-Resolution. Chan's constructive negation has also been implemented as part of the master-theses [Lud91, Vor91].

Non-Disjunctive NMR-Semantics

There are many theoretical papers that deal with the problem of implementation ([BD93, KSS91, DN95, FLMS93]) but only few running systems. The problem of handling and representing ground programs given a non-ground one has also been adressed [KNS94, KNS95, EGLS96].

In [BNNS93, BNNS94] the authors showed how the problem of computing stable models can be transformed to an Integer-Linear Programming Problem. This has been extended in [DM93] to disjunctive programs.

Inoue et. al. show in [IKH92] how to compute stable models by transforming programs into propositional theories and then using a model-generation theorem prover.

Extended logic programs under the well-founded semantics are considered by Pereira and his colleagues: [PAA93, AP96].

[NS96] describes an implementation of WFS and STABLE with a special eye on complexity.

The most advanced system has been implemented by David Warren and his group in Stony Brook based on OLDT-algorithm of [TS86]. They first developed a meta-interpreter (SLG, see [CW96]) in PROLOG and then directly modified the WAM for a direct implementation of WFS (XSB). They use tabling-methods and a mixture of Top-Down and bottom-up evaluation to detect loops. Their system is complete and terminating for non-floundering DATALOG. It also works for general programs but termination is not guaranteed. This system is described in [CW93, CSW95, CW95], and is available by anonymous ftp from `ftp.cs.sunysb.edu/pub/XSB/`.

Disjunctive NMR-Semantics

There are theoretical descriptions of implementations that have not yet been implemented: [FM95, MR95, CL95].

Here are some implemented systems. Inoue et. al. show in [IKH92] how to compute stable models for extended disjunctive programs in a bottom-up-fashion using a theorem prover.

The approach of Bell et. al. ([NNS91]) was used by Dix/Müller to implement versions of the stationary semantics of Przymusinski: [MD93, DM92, Mül92].

Brass/Dix have implemented both D-WFS and DSTABLE for allowed DATALOG programs ([BD95a][22]). An implementation of static semantics is described in [BDP96][23].

Seipel has implemented in his DisLog-system various (modified versions of) semantics of Minker and his group. His system is publicly available at the URL `http://sunwww.informatik.uni-tuebingen.de:8080/dislog/dislog.tar.Z`. However we again point to the very irregular behaviour of these semantics illustrated by Example 7.4.

Finally, there is the DisLoP project undertaken by the Artificial Intelligence Research Group at the University of Koblenz and headed by J. Dix and U. Furbach ([DF96]). This project aims at extending certain theorem proving concepts, such as restart model elimination [BF94] and hyper tableaux [BFN96] calculi, for disjunctive logic programming. The hyper tableaux calculus can handle positive queries with respect to positive disjunctive logic programs and seems to facilitate minimal model generation. Restart model elimination calculus does not use any contrapositives of the given clauses and thus allows for their procedural reading. Moreover, it is answer complete for positive queries [BFS95]. Thus, they are suitable for implementing an interpreter for positive prorgams and the DisLoP system extends this further for non-monotonic negations too.

Currently, DisLoP system can perform minimal model reasoning based on GCWA and WGCWA. Minimal model reasoning is an important problem to tackle, since any well-known semantics for negation is a conservative extension of that. DisLoP can perform minimal model reasoning in both top-down and bottom-up manners. The bottom-up approach employs the hyper tableaux calculus to generate potential minimal models and then uses a novel technique to check the minimality of the generated model without any reference to other models. This approach is described in [Nie96a, Nie96b]. The top-down approach is based on an abductive framework studied in [Ara96]. This introduces an inference rule, negation as failure to explain, which allows us to assume the negation of a sentence if there are no abductive explanations for that. The DisLoP system uses a modified restart model elimination calculus to generate abductive explanations of the given sentence and employs *negation-as-failure-to-explain* inference rule for minimal model reasoning. This system can be extended to

[22]`ftp://ftp.informatik.uni-hannover.de/software/index.html`

[23]`ftp://ftp.informatik.uni-hannover.de/software/static/static.html`

handle non-monotonic semantics such as D-WFS, STATIC etc. Information on the DisLoP project and related publications can be obtained from the WWW page `<http://www.uni-koblenz.de/ag-ki/DLP/>`.

Acknowledgements

We thank Chandrabose Aravindan, Ulrich Furbach and Ilkka Niemelä who provided us with many useful remarks. Katrin Erk and Dorothea Schäfer proofread parts of the paper — special thanks to them.

A Appendix

A.1 Predicate Logic

We assume the reader is familiar with the basic notions of predicate logic such as *models*, *formulae*, *satisfiability* $\models$ and *derivability* $\vdash$. There exist several calculi for first-order predicate logic like Hibert-style, Resolution-style, Gentzen-style or natural deduction-style calculi. One of the main theorems states the completeness of such calculi with respect to the semantics given by models:

Theorem A.1 (Completeness)
A formula φ follows semantically from a theory T (is true in all models of T) iff φ is derivable from T by means of a particular calculus.

$$T \vdash \varphi \quad \textit{iff} \quad T \models \varphi$$

This theorem tells us that we can enumerate all the theorems of a theory, but it does not provide us with a decision-method to do so. In fact, as we will explain now, such a method does not exist.

Before turning to *undecidability*, let us emphasize that in the whole paper we are dealing with predicate logic *without equality "$\doteq$"*. But we can try to simulate "$\doteq$" as follows. We introduce a binary relation-symbol *eq* and require that it satisfies the following axioms with respect to an underlying language $\mathcal{L}$:

$\forall x \; eq(x,x)$

for all function-symbols f of suitable arity:
$\forall x_1 \ldots x_n, y_1 \ldots y_n \; (eq(x_1,y_1) \ldots eq(x_n,y_n)) \rightarrow eq(f(x_1,\ldots,x_n), f(y_1,\ldots,y_n))$

for all predicate-symbols P of suitable arity:
$\forall x_1 \ldots x_n y_1 \ldots y_n \; (eq(x_1,y_1) \ldots eq(x_n,y_n)) \rightarrow (P(x_1,\ldots,x_n) \rightarrow P(y_1,\ldots,y_n))$.

This set, is denoted by $\mathrm{EQ}_{\mathcal{L}}$. It can be shown that transitivity and symmetry of *eq* follow from these axioms. Let us consider the language of *Arithmetic* $\mathcal{L}_{Ar}$ which contains: 0 (a constant), s (a unary function-symbol), *eq* (a two-ary relation-symbol) and $\oplus$, $\otimes$ (ternary relation-symbols).

We have in mind to axiomatize the theory of natural numbers. Before we do so we introduce the following abbreviation. The formula $\exists! z \, \phi(z)$ stands for

$$\exists z \, (\, \phi(z) \wedge \forall y \, \phi(y) \rightarrow eq(y,z)).$$

Definition A.2 (Arithmetic $\mathbf{Ar}_{fin}$)
Ar_{fin} is the finite set consisting of $EQ_{\mathcal{L}_{Ar}}$ and the following axioms:

$$\begin{array}{ll}
\forall x \forall y \exists! z & \oplus(x,y,z) \\
\forall x & \oplus(x,0,x) \\
\forall x \forall y \forall z & \oplus(x,y,z) \rightarrow \oplus(x,s(x),s(z))
\end{array}$$

$$\begin{array}{ll}
\forall x \forall y \exists ! z & \otimes(x,y,z) \\
\forall x & \otimes(x,0,0) \\
\forall x \forall y \forall z \forall z' & \otimes(x,y,z) \rightarrow (\otimes(x,s(y),z') \land \oplus(z,x,z'))
\end{array}$$

The set of natural numbers $\mathcal{N} := (\mathbb{N}, 0^{\mathcal{N}}, s^{\mathcal{N}}, \oplus^{\mathcal{N}}, \otimes^{\mathcal{N}}, eq^{\mathcal{N}})$ is a model of Ar_{fin}. Here $0^{\mathcal{N}}$ is the "true" 0, $s^{\mathcal{N}}$ is the successor-function, $\oplus^{\mathcal{N}}$ is addition and $\otimes^{\mathcal{N}}$ is multiplication (viewed as relations), $eq^{\mathcal{N}}$ is identity. We note the following facts:

1. The set $\{\phi : \ Ar_{fin} \models \phi\}$ is recursively enumerable but not recursive.

2. The set $\{\phi : \ \mathcal{N} \models \phi\}$ is not even recursively enumerable.

We even have

Theorem A.3 (Gödel)
No set of formulae containing Ar_{fin} and having $\mathcal{N}$ as model, is recursive.

Every recursively enumerable set of formulae Φ that contains Ar_{fin} and has $\mathcal{N}$ as a model, is incomplete, i.e. there is ψ with: $\mathcal{N} \models \psi$ but $\Phi \not\models \psi$. Therefore no complete axiomatization of $\mathcal{N}$ is possible.

Note that, although $\mathcal{N}$ formally is not a Herbrand model, it is isomorphic to such a model. In fact, the axioms immediately imply that there is, up to isomorphy, only one single Herbrand-model of Ar_{fin} with respect to $\mathcal{L}_{Ar}$. Therefore to determine if a formula is true in all Herbrand-models of Ar_{fin} is just as complicated as the theory of $\mathcal{N}$ itself. $\mathcal{N}$ contains, for example, famous statements (or there negation) from number theory like Goldbach-conjecture or Fermat's last theorem.

A.2 Complexity Theory

We assume some familiarity with the classes P (problems solvable in deterministic polynomial time) and NP (problems solvable in nondeterministic polynomial time). The class co-NP is the complement of NP, i.e. a problem is in co-NP if its complement is in NP. From these sets we can build larger classes by considering problems solvable in deterministic (resp. nondeterministic) time where we allow to ask queries to an NP-oracle: i.e. whenever we come up with a subproblem that lies in NP, we just ask an oracle which immediately gives us the answer (we count this as just one step). This gives rise to the polynomial hierarchy:

Definition A.4 (Polynomial Hierarchy)
For a complexity class C we denote by P^{C} (resp. NP^{C}) the class of problems solvable in deterministic polynomial (resp. nondeterministic polynomial) time

using C-oracles. Let $\Sigma_0 := \Pi_0 :=$ P and

$$\begin{aligned}\Sigma_{k+1} &:= \mathrm{NP}^{\Sigma_k}\\ \Pi_{k+1} &:= \text{co-NP}^{\Sigma_k}\\ \Delta_{k+1} &:= \mathrm{P}^{\Sigma_k}\end{aligned}$$

Thus Σ_1 is NP with queries to a P-oracle, i.e. $\Sigma_1 =$ NP. Similarly we have $\Pi_1 =$ co-NP and $\Delta_1 =$ P. A problem is in $\Delta_2 = \mathrm{P}^{\mathrm{NP}}$ if it can be solved in deterministic polynomial time with subcalls to an NP-oracle. Although the index is 2, Δ_2 is considered to belong to the first level of the polynomial hierarchy.

The second level of this hierarchy consists of Σ_2, Π_2 and Δ_3. Here $\Sigma_2 := \mathrm{NP}^{\mathrm{NP}}$: nondeterministic polynomial time with queries to an NP-oracle. $\Pi_2 :=$ co-$\mathrm{NP}^{\mathrm{NP}}$ and $\Delta_3 := \mathrm{P}^{[\mathrm{NP}^{\mathrm{NP}}]}$.

It is immediate that

$$\Sigma_k \cup \Pi_k \subseteq \Delta_{k+1} \subseteq \Sigma_{k+1} \cap \Pi_{k+1}$$

but it has not yet been proved that the inclusions are proper. That is, it is not known if the hierarchy collapses at some point or not.

The polynomial hierarchy classifies a subclass of all decidable problems, namely those that are NP-hard. A problem is called NP-hard if any other problem in NP can be polynomially reduced to it. Of particular interest are those problems in a class Π_k or Σ_k that are *the hardest ones*: they are called *complete*. This means that all problems in the respective class can be polynomially reduced to such a complete problem and the problem itself belongs to this class. As an example, to determine if a formula is valid is co-NP-complete. Thus, satisfiability of a propositional formula is NP-complete.

An analogue hierarchy exists (in fact it was the prototype of the polynomial hierarchy) for undecidable problems. The notation is analog to the one just introduced. Therefore one often adds a superscript P to the Π_k and Σ_k which stands for *polynomial* (but not for an oracle) to denote the polynomial hierarchy.

To introduce the arithmetical hierarchy we consider the model $\mathcal{N}$ of the natural numbers and $\mathcal{L}_{Ar}$-formulae. We call such formulae for short *arithmetical*. We classify arithmetical formulae according to their quantifier-alternations:

Definition A.5 (Arithmetical Hierarchy)
We call an arithmetical formula Σ_k^0 (resp. Π_k^0) if it is of the form $\exists\forall\ldots\phi$ (resp. $\forall\exists\ldots\phi$) where ϕ is quantifier-free and there are at most $k-1$ alternations of quantifier-blocks.

We call a set M of natural numbers Σ_k^0-definable, if M is definable by a Σ_k^0-formula. This means that there is a Σ_k^0-formula $\phi(x)$ with one free variable x such that

$$\mathcal{N} \models \phi(i) \text{ iff } i \in M.$$

Note that the Σ_0^0-definable sets coincide with the Π_0^0-definable ones: they are exactly the recursive sets. The recursive enumerable sets are the Σ_1^0-definable ones, the Π_1^0-definable sets are their complements. The set corresponding to the famous *Halting Problem*, i.e. the set of all Gödel numbers of those Turing-machines that stop on their own Gödel number, is Σ_1^0, so this problem is located very low in the hierarchy.

The higher a problem lies in the hierarchy, the more undecidable it is. For example a problem located at the second level, say Σ_2^0, can be thought of as *being recursively enumerable using an oracle which solves Σ_1^0-problems* (like the halting problem).

Analogously to the polynomial hierarchy we have the notions of Σ_k^0-complete and Π_k^0-complete. As an example, the halting problem is Σ_1^0-complete.

In contrast to the polynomial hierarchy, the arithmetical hierarchy is strict. We denote by Δ_k^0 the intersection of Σ_{k+1}^0 and Π_{k+1}^0. We have

$$\Sigma_k^0 \cup \Pi_k^0 \subset \Delta_k^0 = \Sigma_{k+1}^0 \cap \Pi_{k+1}^0.$$

Are there more undecidable problems, not yet captured by our hierarchy? Yes, take for example the theory of $\mathcal{N}$ considered in Section A.1. Obviously, the general problem to determine if an arbitrary formula is true or not in $\mathcal{N}$ can not be captured at a certain level, because the class of formulae in question can have unlimited alternations of quantifiers. The careful reader may have asked himself what the superscript 0 means in Σ_k^0? It just means that we consider just first-order formulae and we do not allow our arithmetical formulae to contain second-order quantifiers.

This remark gives rise to the analytical hierarchy, denoted by Σ_k^1, Π_k^1, where we consider *second-order* arithmetical formulae. We only count the alternations of the quantifiers over sets. So any Σ_k^0-formula is in Σ_0^1.

Note that for the arithmetical hierarchy the identity $\Sigma_0^0 = \Sigma_1^0 \cap \Pi_1^0$ holds. The analogue for the analytical hierarchy does not hold. A counterexample is given by the theory of the natural numbers $\mathcal{N}$: the set of true sentences in arithmetic is in $\Sigma_1^1 \cap \Pi_1^1$ but not in Σ_0^1. This set is also called *hyperarithmetical* for obvious reasons.

For a more detailed treatment of the topics in this section we refer the reader to the standard literature: [BDG88, GJ79, Joh90] and [Pap94, Odi89] for undecidability.

A.3 Default Logic

Reiter's default logic [Rei80] is one of the most prominent nonmonotonic logics. Default logic assumes knowledge to be represented in terms of a default theory. A default theory is a pair (D, W). W is a set of first order formulas representing the facts which are known to be true with certainty. D is a set of defaults the

form

$$\frac{A : B_1, ..., B_n}{C}$$

where A, B_i and C are classical formulas. We will also frequently use the alternative, less space consuming notation $A{:}B_1, \ldots, B_n/C$ for this default. The default has the intuitive reading: if A is provable and, for all i ($1 \leq i \leq n$), $\neg B_i$ is not provable, then derive C. A is called the *prerequisite*, B_i a *consistency condition* or *justification*, and C the *consequent* of the default. For a default d we use $pre(d)$, $just(d)$, and $cons(d)$ to denote the prerequisite, the set of justifications, and the consequent of d, respectively. Open defaults, i.e., defaults with free variables, are usually interpreted as schemata representing all of their closed instances.[24]

Default theories induce so-called extensions which represent acceptable belief sets a reasoner may adopt based on the available information. A formula p is called a skeptical consequence of (D, W) iff p is contained in all extensions of (D, W). p is called a credulous consequence of (D, W) iff p is contained in at least one extension of (D, W).

We will first present a definition of extensions which is slightly different from (but equivalent to) Reiter's original definition. We have found that this definition is somewhat easier to digest. The original definition will be presented later.

Intuitively, E is an extension of (D, W) iff E is a deductively closed (in the sense of classical logic) superset of W satisfying the following two properties

1. all defaults that are "applicable" with respect to E have been applied,
2. every formula in E has a "derivation" from W and applicable defaults.

To make the two requirements more precise we introduce the following notion:

Definition A.6 (Default Proof)
Let (D, W) be a default theory, S a set of formulas, and p a formula. A (D, W)-default proof for p is a finite sequence $P = (d_1, \ldots, d_n)$ of defaults in D such that:

1. $W \cup \{cons(d_1), \ldots, cons(d_{i-1})\} \vdash pre(d_i)$, *for* $i \in \{1, \ldots, n\}$,
2. $W \cup \{cons(d_1), \ldots, cons(d_n)\} \vdash p$.

P is valid in S iff S does not contain the negation of a justification of a default in P.

As usual $\vdash$ denotes classical provability. We now can state the definition of extensions formally:

[24]Reiter treats open defaults somewhat differently and uses a more complicated method to define extensions for them.

Definition A.7 (Extension 1)
Let (D, W) be a default theory. E is an extension of (D, W) iff E is a deductively closed superset of W satisfying the conditions

1. *if $A{:}B_1, \ldots, B_n/C \in D$, $A \in E$ and for all i $(1 \leq i \leq n)$ $\neg B_i \notin E$, then C in E, and*

2. *$p \in E$ implies there is a (D, W)-default proof for p valid in E.*

Reiter's equivalent original definition is more compact. It defines extensions as fixed points of a certain operator.

Definition A.8 (Extension 2)
Let (D, W) be a default theory, S a set of formulas. Let $\Gamma(S)$ be the smallest set such that:

1. *$W \subseteq \Gamma(S)$,*

2. *$Th(\Gamma(S)) = \Gamma(S)$,*

3. *if $A{:}B_1, \ldots, B_n/C \in D$, $A \in \Gamma(S)$, $\neg B_i \notin S$ $(1 \leq i \leq n)$, then $C \in \Gamma(S)$.*

E is an extension of (D, W) iff $E = \Gamma(E)$, that is, if E is a fixed point of Γ.

We finally give a third, quasi-inductive characterization of extensions, also due to Reiter. This version is often used in proofs about default logic and makes the way in which formulas have to be grounded in the premises more explicit. Let E be a set of formulas and define, for a given default theory (D, W), a sequence of sets of formulas as follows:

$$E_0 = W, \text{ and for } i \geq 0$$
$$E_{i+1} = Th(E_i) \cup \{C \mid A{:}B_1, \ldots, B_n/C \in D, A \in E_i, \neg B_i \notin E\}.$$

It can be shown that E is an extension of (D, W) iff $E = \bigcup_{i=0}^{\infty} E_i$. The appearance of E in the definition of E_{i+1} is what renders this alternative definition of extensions non-constructive.

Default theories may have an arbitrary number of extensions (including zero). Extensions are always consistent if W is and if there are no degenerate defaults without consistency conditions. If W is inconsistent then the single extension of (D, W) is the set of all formulas. Extensions are maximal in the following sense: if E is an extension then there is no extension E' such that $E' \subset E$.

A.4 Circumscription

Circumscription is a method of computing the closure of a theory by restricting its models to those that have minimal extensions of some of the predicates and functions. Since its first formulation by McCarthy [McC80], it has taken on

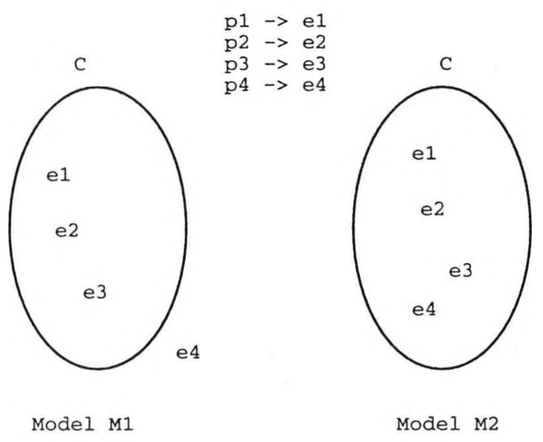


Figure 5: Two Models of a Theory with the Same Valuation.

several different forms, including domain circumscription [McC79] (minimizing the elements in the universe of models), and the most popular and useful version, parallel predicate circumscription [McC80, McC86, Lif85] which we present here.

Although circumscription was originally presented as a schema for adding more formulas to a theory (just as Clark's completion does), here we describe it in terms of restricting the models of the theory. This view leads to the generalization of circumscription by model preference theories, and is more useful analytically in relating circumscription to other nonmonotonic formalisms. More detailed references to circumscription can be found in Lifschitz' excellent survey article [Lif94].

Choose a language $\mathcal{L}$, and let P be the set of predicate symbols that we are interested in minimizing, and Z another set of predicate symbols whose interpretation we allow to vary across compared models. For example, if we wish to minimize the number of cannibals, we would let $P = \{C\}$, and Z be all other predicate symbols (the importance of Z will be indicated later). Suppose A is a theory containing the statements $C(p_1)$, $C(p_2)$, and $C(p_3)$, but no other assertions using C. Then every model of A will have at least the individuals referred to by p_1, p_2, and p_3 with property C. Now consider two models with the same valuation function from terms to individuals, as in Figure 5. In model M_1, the extension of the predicate C includes just the three individuals e_1, e_2, and e_3. In model M_2 there is a fourth individual, e_4, who is a cannibal. Circumscription would prefer M_1 to M_2, since the extension of C in M_1 is a proper subset of its extension in M_2. Under appropriate assumptions (that these terms refer to different individuals), circumscription would yield the result $\neg C(p_4)$, which is not present in the original theory.

Let $A(P, Z)$ be a first-order sentence containing the symbols P and Z. Circumscription prefers models of $A(P, Z)$ that are minimal in the predicates P, assuming that these models have the same interpretation for all symbols not in P or Z. A may contain predicates other than P and Z; these are called the *fixed symbols.*

To state this more formally, let M_1 and M_2 be two models of $A(P, Z)$. $|M|$ is the universe of model M, and $M[\![K]\!]$ is the interpretation of the symbol K in M. Then

Definition A.9 (Minimal Models)

$$M_1 \leq^{P;Z} M_2 \textit{ iff } \begin{cases} 1. & |M_1| = |M_2|. \\ 2. & M_1[\![K]\!] = M_2[\![K]\!] \textit{ for all } K \textit{ not in } P, Z. \\ 3. & M_1[\![P_i]\!] \subseteq M_2[\![P_i]\!] \textit{ for all } P_i \in P. \end{cases}$$

$\leq^{P;Z}$ is a preorder relation (reflexive and transitive) on models, but not necessarily a partial order, since it is not antireflexive. We define the strict order $M_1 <^{(P;Z)} M_2$ as $M_1 \leq^{P;Z} M_2$ and not $M_2 \leq^{P;Z} M_1$. The preferred models of $A(P, Z)$ are those that are minimal according to the strict ordering.

References

[AB90] K. R. Apt and Howard A. Blair. Arithmetic Classification of perfect Models of stratified Programs. *Fundamenta Informaticae*, XIII:1–17, 1990. Addendum in Vol. XIV, pages 339-344, 1991.

[AB94] K. R. Apt and Roland N. Bol. Logic Programming and Negation: A Survey. *Journal of Logic Programming*, 19-20:9–71, 1994.

[AP95] Jose Julio Alferes and Luiz Moniz Pereira. An argumentation theoretic semantics based on non-refutable falsity. In J. Dix, L. Pereira, and T. Przymusinski, editors, *Nonmonotonic Extensions of Logic Programming*, LNAI 927, pages 3–22. Springer, Berlin, 1995.

[AP96] Jose Julio Alferes and Luiz Moniz Pereira. *Reasoning with Logic Programming*. LNCS. Springer, Berlin, 1996. to appear.

[APP96] Jose Julio Alferes, Luiz Moniz Pereira, and Teodor Przymusinski. Strong and Explicit Negation in Non-Monotonic Reasoning and Logic Programming. In L.M. Pereira and E. Orlowska, editors, *JELIA '96, to appear*. Springer, 1996.

[Apt90] K. R. Apt. Logic programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. B*, chapter 10, pages 493–574. Elsevier Science Publishers, 1990.

[Ara96] Chandrabose Aravindan. An abductive framework for negation in disjunctive logic programming. In L. M. Pereira and E. Orlowska, editors, *Joint European workshop on Logics In AI*. LNAI, Springer-Verlag, 1996. (to appear).

[Bar75] Jon Barwise. *Admissible Sets and Structures*. Springer, 1975.

[BD93] Roland N. Bol and L. Degerstedt. Tabulated resolution for well-founded semantics. In *Proc. Int. Logic Programming Symposium'93*, Cambridge, Mass., 1993. MIT Press.

[BD95a] Stefan Brass and Jürgen Dix. A General Approach to Bottom-Up Computation of Disjunctive Semantics. In J. Dix, L. Pereira, and T. Przymusinski, editors, *Nonmonotonic Extensions of Logic Programming*, LNAI 927, pages 127–155. Springer, Berlin, 1995.

[BD95b] Stefan Brass and Jürgen Dix. Characterizations of the Stable Semantics by Partial Evaluation. In A. Nerode, W. Marek, and M. Truszczyński, editors, *Logic Programming and Non-Monotonic Reasoning, Proceedings of the Third International Conference*, LNCS 928, pages 85–98, Berlin, June 1995. Springer.

[BD95c] Stefan Brass and Jürgen Dix. D-WFS: A Confluent calculus and an Equivalent characterization. Technical Report TR 12/95, University of Koblenz, Department of Computer Science, Rheinau 1, September 1995.

[BD95d] Stefan Brass and Jürgen Dix. Disjunctive Semantics based upon Partial and Bottom-Up Evaluation. In Leon Sterling, editor, *Proceedings of the 12th Int. Conf. on Logic Programming, Tokyo*, pages 199–213. MIT Press, June 1995.

[BD96] Stefan Brass and Jürgen Dix. Characterizing D-WFS: Confluence and Iterated GCWA. In L.M. Pereira and E. Orlowska, editors, *JELIA '96, to appear*. Springer, 1996.

[BDG88] J.L. Balcázar, I. Díaz, and J. Gabarró. *Structural Complexity I.* Springer-Verlag, Berlin, 1988.

[BDNP96] Stefan Brass, Jürgen Dix, Ilkka Niemelä, and Teodor. C. Przymusinski. A Comparison of Static Semantics and D-WFS. Technical Report TR 2/96, University of Koblenz, Department of Computer Science, Rheinau 1, February 1996.

[BDP96] Stefan Brass, Jürgen Dix, and Teodor. C. Przymusinski. Characterizations and Implementation of Static Semantics of Disjunctive Programs. Technical Report TR 4/96, University of Koblenz, Department of Computer Science, Rheinau 1, February 1996.

[BED92] Rachel Ben-Eliyahu and Rina Dechter. Propositional Semantics for Disjunctive Logic Programs. In K. R. Apt, editor, *LOGIC PROGRAMMING: Proceedings of the 1992 Joint International Conference and Symposium*, Cambridge, Mass., November 1992. MIT Press.

[BF91a] Nicole Bidoit and Christine Froidevaux. General logical Databases and Programs: Default Logic Semantics and Stratification. *Information and Computation*, 91:15–54, 1991.

[BF91b] Nicole Bidoit and Christine Froidevaux. Negation by Default and unstratifiable logic Programs. *Theoretical Computer Science*, 78:85–112, 1991.

[BF94] P. Baumgartner and U. Furbach. Model Elimination without Contrapositives and its Application to PTTP. *Journal of Automated Reasoning*, 13:339–359, 1994. Short version in: Proceedings of CADE-12, Springer LNAI 814, 1994, pp 87–101.

[BFN96] P. Baumgartner, U. Furbach, and I. Niemelä. Hyper Tableaux. In *JELIA 96*. European Workshop on Logic in AI, Springer, LNCS, 1996. (Long version in: *Fachberichte Informatik*, 8–96, Universität Koblenz-Landau).

[BFS95] P. Baumgartner, U. Furbach, and F. Stolzenburg. Model Elimination, Logic Programming and Computing Answers. In *Proceedings of IJCAI '95*, 1995. (to appear, Long version in: Research Report 1/95, University of Koblenz, Germany).

[BG94] Ch. Baral and Michael Gelfond. Logic Programming and Knowlege Representation. *Journal of Logic Programming*, 19-20, 1994.

[BM86] R. Barbuti and M. Martelli. Negation as Failure. Completeness of the Query Evaluation Process for Horn Clause Programs with Recursive Definition. *Journal of Automated Reasoning*, 2:155–170, 1986.

[BNNS93] Colin Bell, Anil Nerode, Raymond T. Ng, and V. S. Subrahmanian. Implementing Stable Semantics by Linear Programming. In Luis Moniz Pereira and Anil Nerode, editors, *Logic Programming and Non-Monotonic Reasoning, Proceedings of the Second International Workshop*, pages 23–42, Cambridge, Mass., July 1993. Lisbon, MIT Press.

[BNNS94] Colin Bell, Anil Nerode, Raymond T. Ng, and V. S. Subrahmanian. Mixed Integer Programming Methods for Computing Non-Monotonic Deductive Databases. *Journal of the ACM*, 41(6):1178–1215, November 1994.

[Bon93] Piero Bonatti. Shift-based semantics: general results and applications. Technical Report CD-TR-93-59, Technical University of Vienna, Inst. für Informationssysteme, 1993.

[Bör87] Egon Börger. Unsolvable Decision Problems For Prolog Programs. In Egon Börger, editor, *Computation Theory and Logic*, LNCS 270, pages 37–47, Berlin, 1987. Springer.

[BR91] Catril Beeri and Raghu Ramakrishnan. On the power of magic. *The Journal of Logic Programming*, 10:255–299, 1991.

[Bre94] G. Brewka. Adding priorities and specificity to default logic. In *Proc. JELIA-94, York.* Springer, 1994.

[Bry90] François Bry. Query evaluation in recursive databases: bottom-up and top-down reconciled. *Data & Knowledge Engineering*, 5:289–312, 1990.

[BS91] Chitta Baral and V.S. Subrahmanian. Dualities between Alternative Semantics for Logic Programming and Non-monotonic Reasoning. In Anil Nerode, Wiktor Marek, and V. S. Subrahmanian, editors, *Logic Programming and Non-Monotonic Reasoning, Proceedings of the first International Workshop*, pages 69–86, Cambridge, Mass., July 1991. Washington D.C, MIT Press.

[BS92] Chitta Baral and V.S. Subrahmanian. Stable and Extension Class Theory for Logic Programs and Default Logics. *Journal of Automated Reasoning*, 8, No. 3:345–366, 1992.

[CDLS95] M. Cadoli, F. M. Donini, P. Liberatore, and M. Schaerf. The size of a revised knowledge base. In *PODS '95*, pages 151–162, 1995.

[CDS95a] M. Cadoli, F. M. Donini, and M. Schaerf. Is intractability of non-monotonic reasoning a real drawback? Technical Report RAP.09.95, Dipartimento di Informatica e Sistemistica, Università di Roma "La Sapienza", July 1995. To appear in *Artificial Intelligence Journal.* Short version appeared in *Proc. of AAAI-94*, pages 946–951.

[CDS95b] M. Cadoli, F. M. Donini, and M. Schaerf. On compact representations of propositional circumscription. In *STACS '95*, pages 205–216, 1995. Extended version as RAP.14.95 DIS, Univ. of Roma "La Sapienza", July 1995. To appear in *Theoretical Computer Science.*

[CEG92] Marco Cadoli, Thomas Eiter, and Georg Gottlob. An efficient method for eliminating varying predicates from a circumscription. *Artificial Intelligence Journal*, 54:397–410, 1992.

[Cha88] David Chan. Constructive negation based on the completed database. In *Proc. 1988 Conf. and Symp. on Logic Programming*, pages 111–125, September 1988.

[CKPR73] A. Colmerauer, H. Kanoui, R. Pasero, and P. Roussel. Un système de communication homme-machine en français. Technical report, Groupe de Intelligence Artificielle Universite de Aix-Marseille II, 1973.

[CL89] L. Cavedon and J.W. Lloyd. A Completeness Theorem for SLDNF-Resolution. *Journal of Logic Programming*, 7:177–191, 1989.

[CL95] Stefania Costantini and Gaetano A. Lanzarone. Static Semantics as Program Transformation and Well-founded Computation. In J. Dix, L. Pereira, and T. Przymusinski, editors, *Nonmonotonic Extensions of Logic Programming*, LNAI 927, pages 156–180. Springer, Berlin, 1995.

[Cla78] K. L. Clark. Negation as Failure. In H. Gallaire and J. Minker, editors, *Logic and Data-Bases*, pages 293–322. Plenum, New York, 1978.

[CS90] Jan Chomicki and V.S. Subrahmanian. Generalized Closed World Assumption is Π_2^0-Complete. *Information Processing Letters*, 34:289–291, 1990.

[CS93] Marco Cadoli and Marco Schaerf. A Survey of Complexity Results for Non-Monotonic Logics. *Journal of Logic Programming*, 17:127–160, 1993.

[CSW95] Weidong Chen, Terrance Swift, and David S. Warren. Efficient Top-Down Computation of Queries under the Well-Founded Semantics. *Journal of Logic Programming*, 24(3):219–245, 1995.

[CW89] David Chan and Mark Wallace. An Experiment with programming using pure Negation. Technical Report TR, ECRC, July 1989.

[CW93] Weidong Chen and David S. Warren. A Goal Oriented Approach to Computing The Well-founded Semantics. *Journal of Logic Programming*, 17:279–300, 1993.

[CW95] Weidong Chen and David S. Warren. Computing of Stable Models and its Integration with Logical Query Processing. *IEEE Transactions on Knowledge and Data Engineering*, 17:279–300, 1995.

[CW96] Weidong Chen and David S. Warren. Tabled Evaluation with Delaying for General Logic Programs. *Journal of the ACM*, 43(1):20–74, January 1996.

[DC90] Hendrik Decker and Lawrence Cavedon. Generalizing syntactic properties which ensure that SLDNF-Resolution is complete and flounder-free. Technical report, ECRC Munich, January 1990.

[DF96] J. Dix and U. Furbach. The DFG-Project DisLoP on Disjunctive Logic Programming. *Computational Logic*, 2:89–90, 1996.

[DG84] W.F. Dowling and J.H. Gallier. Linear Time Algorithms for Testing the Satisfiability of Propositional Horn Formulae. *Journal of Logic Programming*, 1:267–284, 1984.

[DGM94] Jürgen Dix, Georg Gottlob, and Viktor Marek. Causal Models for Disjunctive Logic Programs. In Pascal Van Hentenryck, editor, *Proceedings of the 11th Int. Conf. on Logic Programming, S. Margherita Ligure*, pages 290–302. MIT, June 1994.

[DGM96] Jürgen Dix, Georg Gottlob, and Viktor Marek. Reducing disjunctive to non-disjunctive semantics by shift-operations. *Fundamenta Informaticae*, forthcoming, 1996.

[Dix91] Jürgen 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, Cambridge, Mass., July 1991. Washington D.C, MIT Press.

[Dix92a] Jürgen Dix. A Framework for Representing and Characterizing Semantics of Logic Programs. In B. Nebel, C. Rich, and W. Swartout, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the Third International Conference (KR '92)*, pages 591–602. San Mateo, CA, Morgan Kaufmann, 1992.

[Dix92b] Jürgen Dix. Classifying Semantics of Disjunctive Logic Programs. In K. R. Apt, editor, *LOGIC PROGRAMMING: Proceedings of the 1992 Joint International Conference and Symposium*, pages 798–812, Cambridge, Mass., November 1992. MIT Press.

[Dix95a] Jürgen Dix. A Classification-Theory of Semantics of Normal Logic Programs: I. Strong Properties. *Fundamenta Informaticae*, XXII(3):227–255, 1995.

[Dix95b] Jürgen Dix. A Classification-Theory of Semantics of Normal Logic Programs: II. Weak Properties. *Fundamenta Informaticae*, XXII(3):257–288, 1995.

[Dix95c] Jürgen Dix. Semantics of Logic Programs: Their Intuitions and Formal Properties. An Overview. In Andre Fuhrmann and Hans Rott, editors, *Logic, Action and Information – Essays on Logic in Philosophy and Artificial Intelligence*, pages 241–327. DeGruyter, 1995.

[DM92] Jürgen Dix and Martin Müller. Abstract Properties and Computational Complexity of Semantics for Disjunctive Logic Programs. In *Proc. of the Workshop W1, Structural Complexity and Recursion-theoretic Methods in Logic Programming, following the JICSLP '92*, pages 15–28. H. Blair and W. Marek and A. Nerode and J. Remmel, November 1992. also available as Technical Report 13/93, University of Koblenz, Department of Computer Science.

[DM93] Jürgen Dix and Martin Müller. 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, Cambridge, Mass., July 1993. Lisbon, MIT Press.

[DM94a] Jürgen Dix and Martin Müller. An Axiomatic Framework for Representing and Characterizing Semantics of Disjunctive Logic Programs. In Pascal Van Hentenryck, editor, *Proceedings of the 11th Int. Conf. on Logic Programming, S. Margherita Ligure*, pages 303–322. MIT, June 1994.

[DM94b] Jürgen Dix and Martin Müller. Partial Evaluation and Relevance for Approximations of the Stable Semantics. In Z.W. Ras and M. Zemankova, editors, *Proceedings of the 8th Int. Symp. on Methodologies for Intelligent Systems, Charlotte, NC, 1994*, LNAI 869, pages 511–520, Berlin, 1994. Springer.

[DM94c] Jürgen Dix and Martin Müller. The Stable Semantics and its Variants: A Comparison of Recent Approaches. In L. Dreschler-Fischer and B. Nebel, editors, *Proceedings of the 18th German Annual Conference on Artificial Intelligence (KI '94), Saarbrücken, Germany*, LNAI 861, pages 82–93, Berlin, 1994. Springer.

[DN95] Lars Degerstedt and Ulf Nilsson. Magic Computation of Wellfounded Semantics. In J. Dix, L. Pereira, and T. Przymusinski, editors, *Nonmonotonic Extensions of Logic Programming*, LNAI 927, pages 181–204. Springer, Berlin, 1995.

[Dra94] Wlodzimierz Drabent. What is failure? A constructive approach to negation. *Acta Informatica*, 1994. forthcoming.

[Dun92] P. M. Dung. On the relations between stable and wellfounded semantics of logic programs. *Theoretical Computer Science*, 105:7–25, 1992.

[EG93] Thomas Eiter and Georg Gottlob. Propositional Circumscription and Extended Closed World Reasoning are Π_2^P-complete. *Theoretical Computer Science*, 144(2):231–245, Addendum: vol. 118, p. 315, 1993, 1993.

[EGLS96] T. Eiter, G. Gottlob, J. Lu, and V. S. Subrahmanian. Computing Non-Ground Representations of Stable Models. Technical report, University of Maryland, 1996.

[EGM93] Thomas Eiter, Georg Gottlob, and Heikki Mannila. Expressive Power and Complexity of Disjunctive DATALOG. In *Proceedings of Workshop on Logic Programming with Incomplete Information, Vancouver Oct. 1993, following ILPS' 93*, pages 59–79, 1993.

[Fag93] F. Fages. Consistency of Clark's completion and existence of stable models. *Methods of Logic in Computer Science*, 2, 1993.

[Fit85] Melvin C. Fitting. A Kripke-Kleene Semantics of logic Programs. *Journal of Logic Programming*, 4:295–312, 1985.

[FLMS93] J. A. Fernändez, J. Lobo, J. Minker, and V.S. Subrahmanian. Disjunctive LP + Integrity Constraints = Stable Model Semantics. *Annals of Mathematics and Artificial Intelligence*, 8(3-4), 1993.

[FM95] J. A. Fernändez and J. Minker. Computing Perfect Models of Disjunctive Stratified Databases. *Annals of Mathematics and Artificial Intelligence*, 1995. to appear.

[GJ79] M.R. Garey and D.S. Johnson. *Computers and Intractability*. W.H. Freeman and Company, San Francisco, 1979.

[GL88] Michael Gelfond and Vladimir Lifschitz. The Stable Model Semantics for Logic Programming. In R. Kowalski and K. Bowen, editors, *5th Conference on Logic Programming*, pages 1070–1080. MIT Press, 1988.

[GL91] Michael Gelfond and Vladimir Lifschitz. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*, 9:365–387, 1991. (Extended abstract appeared in: Logic Programs with Classical Negation. *Proceedings of the 7-th International Logic Programming Conference, Jerusalem*, pages 579-597, 1990. MIT Press.).

[Gor93] T. F. Gordon. *The Pleadings Game: An Artificial Intelligence Model of Procedural Justice*. PhD thesis, TU Darmstadt, 1993.

[GPSK95] G. Gogic, C. Papadimitriou, B. Selman, and H. Kautz. The Comparative Linguistics of Knowledge Representation. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence*, pages 862–869, Montreal, Canada, August 1995. Morgan Kaufmann Publishers.

[Gur88] Y. Gurevich. Logic and the Challenge of Computer Science. In E. Börger, editor, *Trends in Theoretical Computer Science*, chapter 1. Computer Science Press, 1988.

[IKH92] Katsumi Inoue, M. Koshimura, and R. Hasegawa. Embedding negatin-as-failure into a model generation theorem prover. In Deepak Kapur, editor, *Automated Deduction — CADE-11*, number 607 in LNAI, Berlin, 1992. Springer.

[Imi91] T. Imielinski. Incomplete Deductive Databases. *Annals of Mathematics and Artificial Intelligence*, 3:259–294, 1991.

[JdL92] Catholijn Jonker and G.R.Renardel de Lavalette. A tractable algorithm for the wellfounded model. Technical Report Logic Group Preprint Series No. 746, Utrecht University, Dept. of CS, 1992.

[Joh90] D.S. Johnson. A catalog of complexity classes. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A. Algorithms and Complexity, pages 67–161. 1990.

[KNS94] Vadim Kagan, Anil Nerode, and V. S. Subrahmanian. Computing Definite Logic Programs by Partial Instantiation. *Annals of Pure and Applied Logic*, 67:161–182, 1994.

[KNS95] Vadim Kagan, Anil Nerode, and V. S. Subrahmanian. Computing Minimal Models by Partial Instantiation. *Theoretical Computer Science*, 155:157–177, 1995.

[Kon88] Kurt Konolige. Partial Models and Non-Monotonic Reasoning. In J. Richards, editor, *The Logic and Aquisition of Knowledge.* Oxford Press, 1988.

[Kow74] R.A. Kowalski. Predicate logic as a programming language. In *Proceeedings IFIP' 74*, pages 569–574. North Holland Publishing Company, 1974.

[KSS91] David B. Kemp, Peter J. Stuckey, and Divesh Srivastava. Magic Sets and Bottom-Up Evaluation of Well-Founded Models. In Vijay Saraswat and Kazunori Ueda, editors, *Proceedings of the 1991 Int. Symposium on Logic Programming*, pages 337–351. MIT, June 1991.

[Kun87] Kenneth Kunen. Negation in Logic Programming. *Journal of Logic Programming*, 4:289–308, 1987.

[Kun89] Kenneth Kunen. Signed Data Dependencies. *Journal of Logic Programming*, 7:231–245, 1989.

[Lif85] Vladimir Lifschitz. Computing Circumscription. In *Proceedings of the International Joint Conference on Artificial Intelligence, Los Angeles, California*, pages 121–127, 1985.

[Lif94] V. Lifschitz. *Circumscription*, pages 297–353. Clarendon, Oxford, 1994.

[Lif96] V. Lifschitz. Foundations of declarative logic programming. In G. Brewka, editor, *Principles of Knowledge Representation.* CSLI publishers, Studies in Logic, Language and Information, 1996.

[Llo87] John W. Lloyd. *Foundations of Logic Programming.* Springer, Berlin, 2nd edition, 1987.

[LMR92] Jorge Lobo, Jack Minker, and Arcot Rajasekar. *Foundations of Disjunctive Logic Programming.* MIT-Press, 1992.

[Lud91] Bertram Ludäscher. CNF-Prolog: A Meta-Interpreter for Chan's Constructive Negation, Implementation. Technical report, Master Thesis, Karlsruhe University (in german), 1991.

[McC79] John McCarthy. First order theories of individual concepts and propositions. In B. Meltzer and D. Michie, editors, *Machine Intelligence 9*, pages 120–147. Edinburgh University Press, Edinburgh, 1979.

[McC80] John McCarthy. Circumscription — a form of nonmonotonic reasoning. *Artificial Intelligence*, 13(1–2), 1980.

[McC86] John McCarthy. Applications of circumscription to formalizing commonsense knowledge. *Artificial Intelligence*, 28, 1986.

[MD93] Martin Müller and Jürgen 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, Cambridge, Mass., July 1993. Lisbon, MIT Press.

[Min82] Jack Minker. On indefinite databases and the closed world assumption. In *Proceedings of the 6th Conference on Automated Deduction, New York*, pages 292–308, Berlin, 1982. Springer.

[Min88] Jack Minker. *Foundations of Deductive Databases.* Morgan Kaufmann, 95 First Street, Los Altos, CA 94022, 1st edition, 1988.

[Min93] Jack Minker. An Overview of Nonmonotonic Reasoning and Logic Programming. *Journal of Logic Programming, Special Issue*, 17, 1993.

[MMP88] Paolo Mancarella, Simone Martini, and Dino Pedreschi. Complete logic Programs with domain-closure Axiom. *Journal of Logic Programming*, 5:263–276, 1988.

[Mos74] Y. N. Moschovakis. *Elementary Induction on Abstract Structures.* North-Holland, 1974.

[MR95] Jack Minker and Carolina Ruiz. Computing stable and partial stable models of extended disjunctive logic programs. In J. Dix, L. Pereira, and T. Przymusinski, editors, *Nonmonotonic Extensions of Logic Programming*, LNAI 927, pages 205–229. Springer, Berlin, 1995.

[MRT92] Wiktor Marek, Arcot Rajasekar, and Mirek Truszczyński. Complexity of Computing with Extended Propositional Logic Programs. In *Proc. of the Workshop W1, Structural Complexity and Recursion-theoretic Methods in Logic Programming, following the IJCSLP '92*, pages 93–102. H. Blair and W. Marek and A. Nerode and J. Remmel, November 1992.

[Mül92] Martin Müller. Examples and Run-Time Data from KORF, 1992.

[Nie96a] I. Niemelä. Implementing circumscription using a tableau method. In *Proceedings of the European Conference on Artificial Intelligence*, Budapest, Hungary, August 1996. John Wiley. To appear.

[Nie96b] I. Niemelä. A tableau calculus for minimal model reasoning. In *Proceedings of the Fifth Workshop on Theorem Proving with Analytic Tableaux and Related Methods*, pages 278–294, Terrasini, Italy, May 1996. Springer-Verlag.

[NNS91] Anil Nerode, Raymond T. Ng, and V.S. Subrahmanian. Computing Circumscriptive Deductive Databases. CS-TR 91-66, Computer Science Dept., Univ. Maryland, University of Maryland, College Park, Maryland, 20742, USA, December 1991.

[NS96] Ilkka Niemelä and Patrik Simons. Efficient implementation of the well-founded and stable model semantics. In *Proceedings of the Joint International Conference and Symposium on Logic Programming*, Bonn, Germany, September 1996. To appear.

[Odi89] P. Odifreddi. *Classical Recursion Theory.* North-Holland, 1989.

[PA92] L.M. Pereira and J.J. Alferes. Well founded semantics for logic programs with explicit negation. In *Proc. 10th European Conference on Artificial Intelligence, Vienna*, 1992.

[PAA93] L. M. Pereira, J. N. Aparício, and J. J. Alferes. Non-Monotonic Reasoning with Logic Programming. *Journal of Logic Programming*, 17:227–264, 1993.

[Pap94] C.H. Papadimitriou. *Computational Complexity.* Addison-Wesley, 1994.

[Poo85] D. Poole. On the comparison of theories: Preferring the most specific explanation. In *Proc. IJCAI-85, Los Angeles*, 1985.

[Pra93] H. Prakken. *Logical Tools for Modelling Legal Argument.* PhD thesis, VU Amsterdam, 1993.

[Prz91a] Teodor Przymusinski. Semantics of Disjunctive Logic Programs and Deductive Databases. Technical report, Department of Computer Science, University of California at Riverside, November 1991.

[Prz91b] Teodor Przymusinski. Stationary Semantics for Normal and Disjunctive Logic Programs. In C. Delobel, M. Kifer, and Y. Masunaga, editors, *DOOD '91, Proceedings of the 2nd International Conference*, Berlin, December 1991. Muenchen, Springer. LNCS 566.

[Prz95] Teodor Przymusinski. Static Semantics For Normal and Disjunctive Logic Programs. *Annals of Mathematics and Artificial Intelligence*, Special Issue on Disjunctive Programs, 1995. to appear.

[Rei78] Raymond Reiter. On closed world data bases. In Hervé Gallaire and Jack Minker, editors, *Logic and Data Bases*, pages 55–76, New York, 1978. Plenum.

[Rei80] Raymond Reiter. A Logic for Default-Reasoning. *Artificial Intelligence*, 13:81–132, 1980.

[RLM89] Arcot Rajasekar, Jorge Lobo, and Jack Minker. Weak Generalized Closed World Assumption. *Journal of Automated Reasoning*, 5:293–307, 1989.

[RN95] Stuart Russel and Peter Norvig. *Artificial Intelligence — A Modern Approach.* Prentice Hall, New Jersey 07458, 1995.

[Ros89] Kenneth A. Ross. The well-founded semantics for disjunctive logic programs. In *Proceedings of the first International Conference on Deductive and Object Oriented Databases, Kyoto, Japan*, pages 1–22, 1989.

[Ros92] Kenneth A. Ross. A procedural semantics for well-founded negation in logic programs. *Journal of Logic Programming*, 13:1–22, 1992.

[RT88] Kenneth A. Ross and Rodney A. Topor. Inferring negative Information from disjunctive Databases. *Journal of Automated Reasoning*, 4:397–424, 1988.

[Sac93] Domenico Sacca. The Expressive Power of Stable Models For DATALOG Queries with Negation. In *Proceedings of Workshop on Logic Programming with Incomplete Information, Vancouver Oct. 1993, following ILPS' 93*, pages 150–162, 1993.

[Sak89] Chiaki Sakama. Possible Model Semantics for Disjunctive Databases. In Won Kim, Jean-Marie Nicolas, and Shojiro Nishio, editors, *Deductive and Object-Oriented Databases, Proceedings of the First International Conference (DOOD89)*, pages 1055–1060, Kyoto, Japan, 1989. North-Holland Publ.Co.

[Sch90] 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.

[Sch92] John S. Schlipf. A Survey of Complexity and Undecidability Results in Logic Programming. In H. Blair, W. Marek, A. Nerode, and J. Remmel, editors, *Proceedings of the Workshop on Complexity and Recursion-theoretic Methods in Logic Programming, following the JICSLP'92.* informal, 1992.

[She88a] John C. Shepherdson. Language and Equality Theory in Logic Programming. Pm-88-08, School of Mathematics, University of Bristol, School of Mathematics, University Walk, August 1988.

[She88b] John C. Shepherdson. Negation in Logic Programming. In Jack Minker, editor, *Foundations of Deductive Databases*, chapter 1, pages 19–88. Morgan Kaufmann, 1988.

[She91] John C. Shepherdson. Unsolvable Problems for SLDNF-Resolution. *Journal of Logic Programming*, 10:19–22, 1991.

[SI93] Chiaki Sakama and Katsumi Inoue. Negation in Disjunctive Logic Programs. In D. Warren and Peter Szeredi, editors, *Proceedings of the 10th Int. Conf. on Logic Programming, Budapest*, Cambridge, Mass., July 1993. MIT Press.

[SS95] Chiaki Sakama and Hirohisa Seki. Partial Deduction of Disjunctive Logic Programs: A Declarative Approach. In *Logic Program Synthesis and Transformation – Meta Programming in Logic*, LNCS 883, pages 170–182, Berlin, 1995. Springer.

[Stä94] Robert F. Stärk. Input/output dependencies of normal logic programs. *Journal of Logic and Computation*, 4(3):249–262, 1994.

[Tar55] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.

[THT86] Richmond Thomason, Jeff Horty, and D. S. Touretzky. A Calculus for Inheritance in Monotonic Semantic Nets. Research Note CMU-CS 86-138, Carnegie Mellon, 1986.

[Tou86] D. S. Touretzky. *The Mathematics of Inheritance.* Research Notes in Artificial Intelligence. Pitman, London, 1986.

[TS86] H. Tamaki and T. Sato. OLD Resolution with Tabulation. In *Proceedings of the Third International Conference on Logic Programming, London,* LNAI, pages 84–98, Berlin, June 1986. Springer.

[TTH91] D. S. Touretzky, R. H. Thomason, and J. F. Horty. A skeptic's menagerie: Conflictors, preemptors, reinstaters, and zombies in nonmonotonic inheritance. In *Proc. 12th IJCAI, Sydney,* 1991.

[Ull89a] Jeffrey D. Ullman. Bottom-up Beats Top-down for Datalog. In *Proc. of the Eight ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, Philadelphia, Pennsylvania,* pages 140–149. ACM Press, March 1989.

[Ull89b] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems, Vol. 2.* Computer Science Press, Rockville, 1989.

[vEK76] M.H. van Emden and R.A. Kowalski. The semantics of predicate logic as a programming language. *JACM,* 23:733–742, 1976.

[vGRS88] Allen van Gelder, Kenneth A. Ross, and J. S. Schlipf. Unfounded Sets and well-founded Semantics for general logic Programs. In *Proceedings 7th Symposion on Principles of Database Systems,* pages 221–230, 1988.

[vGRS91] Allen van Gelder, Kenneth A. Ross, and John S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM,* 38:620–650, 1991.

[Vor91] Martin Vorbeck. CNF-Prolog: A Meta-Interpreter for Chan's Constructive Negation, Theory. Technical report, Master Thesis, Karlsruhe University (in german), 1991.

[Wit91a] Cees Witteveen. Partial Semantics for Truth Maintenance. In J. van Eijck, editor, *Logics in AI,* LNAI 478, Berlin, 1991. Springer.

[Wit91b] Cees Witteveen. Skeptical Reason Maintenance is Tractable. In J. Allen, R. Fikes, and B. Sandewall, editors, *Proceedings of the second Conference on Principles of Knowledge Representation and Reasoning, Cambridge, Massachusetts,* pages 570–581. Morgan Kaufmann, 1991.