Extended Logic Programming and Knowledge Representation

Jürgen Dix

University of Koblenz-Landau
Dept. of Computer Science

and

Gerhard Brewka

University of Leipzig
Dept. of Computer Science

I. Introduction

- Some History
- Nonmonotonicity in KR
- How this Lecture is organized

1 Some History

- **1956:** Dartmouth Workshop, “Artificial Intelligence”
- **1962:** GPS (Newell/Simon)
- **1965:** Resolution (Robinson)
- **1973:** PROLOG (Colmerauer et. al.)
- **1980:** Nonmonotonic Reasoning (Reiter, McCarthy, McDermott/Doyle)
- **1986:** Workshop on DDB and LP (Minker)
- **since 1991:** Series of LPNMR conferences (see http://www.uni-koblenz.de/ag-ki/lpnmr97/)

2 Nonmonotonicity in KR

- **Situation Calculus:** McCarthy/Hayes
- **Frame-Problem:** Describe non-changes efficiently
- **Default Logic**, **Autoepistemic Logic**, **Circumscription**
- **Complexity:** Due to consistency checks: not even r.e. But very compact problem description.

3 How this Lecture is organized

Monday:	Chapter I: Introduction
	Chapter II: Definite Programs
Tuesday:	Chapter III: Adding Default Negation
Wednesday:	Chapter IV: Adding Explicit Negation
Thursday:	Chapter V: Adding Preferences
Friday:	Chapter VI: Adding Disjunction
	Chapter VII: What Do We Want?

II. Definite Logic Programs

- Top-Down
- Bottom-Up
- Herbrand Models and underlying Language
- Why going beyond Definite Programs?
- What is a Semantics?

Definition II.0.1 (Definite Logic Program)
A definite *logic program consists of a finite number of* rules *of the form*

$$\boxed{A \leftarrow B_1, \ldots, B_m},$$

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

Definition II.0.2 (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.

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 s.t.

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

In fact, this problem is decidable and if the two atoms are *unifiable* we can indeed compute a *most general unifier*, called mgU (see [Llo87]). An mgU $\boxed{\Theta}$ for A and B is a substitution defined on all variables ocurring in both A and B such that

$$A\Theta \text{ is identical to } B\Theta.$$

How are our programs related to classical predicate logic?
Can we simply translate rules into classical formulae via

$$\boxed{B_1 \wedge \ldots \wedge B_m \quad \supset \quad A}?$$

Principle II.0.3 (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* $\neg A$.

1 Top-Down

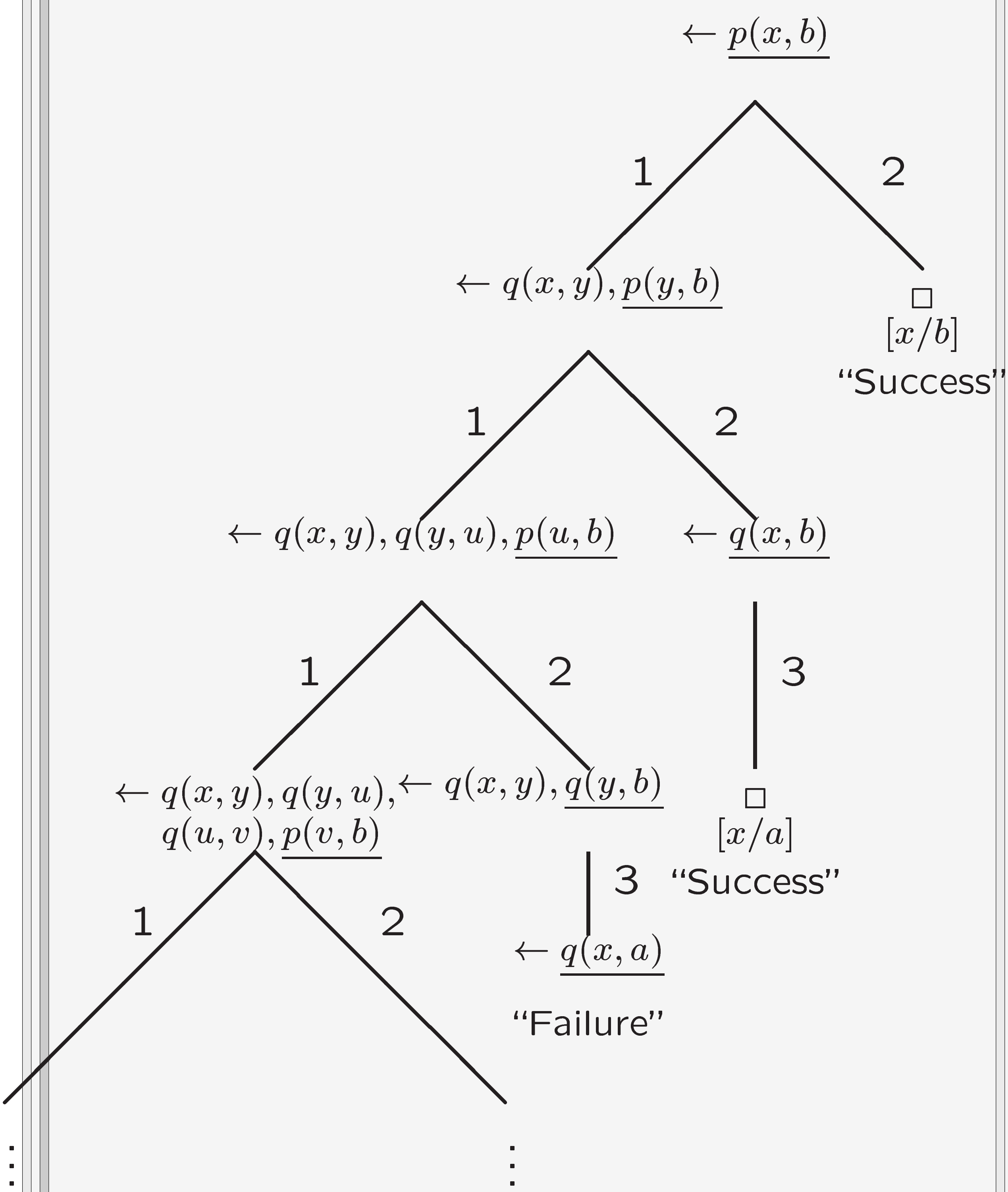
Example II.1.1 (SLD-Resolution)

Let the program P_{SLD} consist of the following three clauses

$$
\begin{array}{llll}
(1) & p(x,z) & \leftarrow & q(x,y), p(y,z) \\
(2) & p(x,x) & & \\
(3) & q(a,b) & &
\end{array}
$$

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.


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

Obviously, there are three different sorts of branches, namely

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

Theorem II.1.2 (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* Θ *wrt* Q
(*i.e.* $\exists\sigma : Q\tau\sigma = Q\Theta$.)

Note that not any correct answer is computed,
only the most general one is.

The main feature of SLD-Resolution is its *Goal-Orientedness*.

2 Bottom-Up

Example II.2.1 (T_P)
Given a definite program P let $\boxed{T_P}$: $2^{B_P} \longmapsto 2^{B_P}$; $\mathcal{I} \longmapsto T_P(\mathcal{I})$

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

It turns out that $\boxed{T_P}$ is monotone and continuous: so that

Theorem II.2.2 (T_P and M_P)
$M_P = \boxed{T_P}\!\uparrow^{\omega} = lfp(\boxed{T_P})$.

This approach is especially important in Databases, where the underlying language does not contain function symbols (DATALOG). In addition, rules of the form

$$\boxed{p \leftarrow p}$$

do not make any problems.

Principle II.2.3 (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.

Two main Shortcomings of Bottom-Up:

- no Goal-orientedness,
- Many facts are recomputed again and again.

3 Herbrand Models

Why are we considering all models? Doesn't it make sense to look only at models generated by the underlying language?

Definition II.3.1 (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: $a = t^{\mathcal{I}}$.*

Herbrand models $\mathcal{I}$ are models satisfying UNA, DCA and: a term "$f(t_1, \ldots, t_n)$" is interpreted in $\mathcal{I}$ as "$\boxed{f^{\mathcal{I}}}(\boxed{t_1^{\mathcal{I}}}, \ldots, \boxed{t_n^{\mathcal{I}}})$".

For universal theories T and existential formulae ϕ the following holds

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

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

4 Why going beyond?

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: ψ implies ϕ.

How can we formulate such a statement as a logic program? The most natural way is to use negation "*not*"

$$\phi \leftarrow \psi, \boxed{\textit{not}}\, ab$$

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

Example II.4.1 (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:

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

together with some particular facts, like $bird(Tweety)$ and $penguin(Sam)$.

For the query "$make\_top(Tweety)$" we expect the answer "yes" while for "$make\_top(Sam)$" we expect the answer "no".

Example II.4.2 (Temporal Projection: Yale-Shooting)
We distinguish between three sorts 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.

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 $\boxed{wait}$, $\boxed{load}$ and $\boxed{shoot}$) 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.

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

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

Such a straightforward formalization leads in most versions of classical nonmonotonic logic to the unexpected result, that Fred is not neccesarily dead. We expect to derive

$$holds(alive, res(load, s_0))$$

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

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?*

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.

Definition II.5.1 (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:

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

Formally, we can associate to any semantics SEM in the sense of Definition II.5.1 two entailment relations

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

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

When there is only one model, we will omit the outer brackets and write (instead of $\text{SEM}(P) = \{M\}$)

$$\text{SEM}(P) = M.$$

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

III. Adding Default Negation

- Negation-as-Finite-Failure
- Negation-as-Failure
- Wellfounded Semantics: WFS
- Stable Semantics: STABLE
- Complexity and Expressibility

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 SLDNF-Resolution.

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.

1 Negation-as-Finite-Failure

How should we handle default-atoms?

- If we reach "*not* A" as a subgoal, 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.

Principle III.1.1 (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").*

SLDNF handles properly our examples II.4.1 and II.4.2.

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.

Floundering

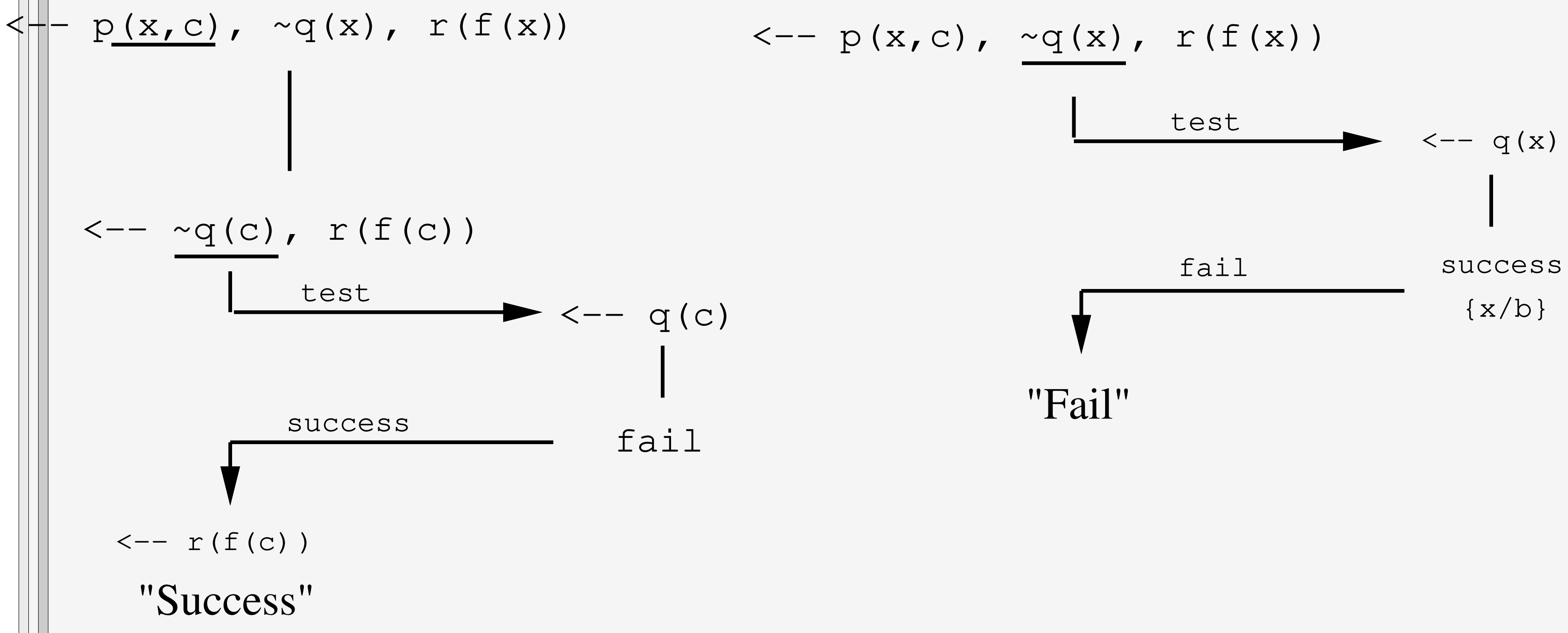


Figure 1: The Floundering-Problem

2 Negation-as-Failure

SLDNF answers quite easily our requirements of a semantics SEM (stated explicitly in Definition II.5.1).

Principle III.2.1 (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.

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.

Example III.2.2 (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 $reachable(a)$*"!*

Partial Evaluation

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 are definitions for $reachable(a)$ and $reachable(b)$. So we can replace the body atoms of $reachable$ by their definitions:

$$\boxed{reachable(a)} \leftarrow edge(a,b), \boxed{edge(b,a), reachable(a)}$$
$$\boxed{reachable(b)} \leftarrow edge(b,a), \boxed{edge(a,b), reachable(b)}$$

that can both be eliminated by applying Principle II.2.3.

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 II.0.3 both atoms should be considered false.

Definition III.2.3 (GPPE)
A semantics SEM satisfies GPPE if the following transformation does not change SEM:
Replace a rule $A \leftarrow \mathcal{B}^+ \wedge \textit{not}\ \mathcal{B}^-$ where $\mathcal{B}^+$ contains an atom B by the rules

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

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

$$\begin{aligned} B &\leftarrow \textit{not}\ E \\ B &\leftarrow D, \textit{not}\ C' \\ A &\leftarrow B, \textit{not}\ C \end{aligned} \quad \mapsto \quad \begin{aligned} A &\leftarrow D, \textit{not}\ C', \textit{not}\ C \\ A &\leftarrow \textit{not}\ E, \textit{not}\ C \end{aligned}$$

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.*

Example III.2.4 (COMP vs. NMR)

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

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

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

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

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

Principle III.2.5 ($\boxed{\textbf{Subsumption}}$)
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

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

$$\boxed{\begin{array}{lll} A & \leftarrow & D, \textit{not}\ E, \textit{not}\ F \\ A & \leftarrow & D, \textit{not}\ F \end{array}} \quad \mapsto \quad \boxed{\begin{array}{lll} A & \leftarrow & D, \textit{not}\ F \end{array}}$$

3 Wellfounded Semantics: WFS

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: $\mathsf{SEM}_1(P) \models l$ implies $\mathsf{SEM}_2(P) \models l$.
I.e. all (default-) atoms derivable from SEM_1 with respect to P are also derivable from SEM_2.

Theorem III.3.1 (WFS)
There exists the weakest semantics satisfying our four principles Elimination of Tautologies, Subsumption, Reduction, and GPPE. This semantics is called wellfounded semantics WFS.

Theorem III.3.2 (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 strongly terminating for fair sequences: 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

$$\begin{aligned} WFS(P) = \quad & \{A : A \leftarrow \quad \in res(P)\} \\ \cup \quad & \{\text{not } A : A \text{ is in no head of } res(P)\} \end{aligned}$$

Example III.3.3 (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

- $\boxed{wins(x)} \leftarrow$ *move_from_to(x,y)*, $\boxed{\text{not } wins(y)}$

meaning that

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

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

Assume we also have the facts

$$move\_from\_to(a,b), move\_from\_to(b,a), move\_from\_to(b,c).$$

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).

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

4 Stable Semantics: STABLE

The stable semantics associates to any program P a set of 2-valued models. STABLE also satisfies

Principle III.4.1 (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.

Contradiction: $A \vee B \quad \leftarrow \quad C, D, \mathit{not}\ C$

Theorem III.4.2 (STABLE)
There exists the weakest semantics satisfying our five principles Elimination of Tautologies, Subsumption, Reduction, GPPE and Elimination of Contradictions.

The underlying idea of STABLE is that any atom in an intended model should have a definite reason to be true or false. Gelfond-Lifschitz transformation: for a program P and a model $\boxed{N} \subseteq B_P$ we define

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

where $rule := A \leftarrow B_1, \ldots, B_n, \textit{not}\ C_1, \ldots, \textit{not}\ C_m$ is transformed as follows

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

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 III.4.3 (STABLE)

N *is called a* stable *model*[a] *of* P *iff* $M_{P^N} = N$.

[a] Note that we only consider Herbrand models.

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 $N \models l$.
- If WFS(P) is two-valued, then WFS(P) is the unique stable model.

For Example III.3.3 we have two stable models: $\{wins(a), wins(c)\}$ and $\{wins(b), wins(c)\}$ and therefore

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

Example III.4.4 (Reasoning by cases)

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

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:

$$\text{STABLE}(p \leftarrow \textit{not}\ p) = \emptyset.$$

- STABLE does not allow for a goal-oriented implementation.

Example III.4.5 (STABLE is not Goal-Oriented)

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

$P_{rel(a)}$ 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.

- Can we make STABLE consistent by simply building consistency in?
  Yes!

- Can we make STABLE relevant by simply building Relevance in?
  No!

Example III.4.6 (Modified STABLE and Relevance)

P :	$P_{rel(x)}$:	$P_{rel(y)}$:
$a \leftarrow \text{not } b$	$a \leftarrow \text{not } b$	$a \leftarrow \text{not } b$
$b \leftarrow \text{not } a$	$b \leftarrow \text{not } a$	$b \leftarrow \text{not } a$
$x \leftarrow a, \text{not } x$	$x \leftarrow a, \text{not } x$	$y \leftarrow b, \text{not } y$
$y \leftarrow b, \text{not } y$		
$w \leftarrow \text{not } x, \text{not } y$		

Both not x *and* not y *are true in* P *(exactly one stable model exist for* $P_{rel(x)}$ *and* $P_{rel(y)}$*). Therefore* w *should also be true, but it is not, because* P *does not have stable models.*

5 Complexity and Expressibility

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

	Expressibility *1. ord. prog.* *(no functions)*
$\mathbf{M_P}$ *(P is Horn)*	$\subset$**IND** (thus $\subset$**P**)
COMP	$=$**co-NP**
$\mathbf{COMP_3}$	$=$**IND** (thus $\subset$ **P**)
STABLE	$=$**co-NP**
REG-SEM	$=\Pi_2^P$
WFS	$=$**IND** (thus $\subset$**P**)
WFS′	$=$**IND** (thus $\subset$**P**)
$\mathbf{WFS^+}$	$=$**IND** (thus $\subset$**P**)

Efficient Computation of WFS

One possibility to compute WFS is to just apply the confluent calculus of transformations described in Theorem III.3.2. However, there are two details worth noticing:

- **Fairness**,
- **Complexity**.

Fairness: Already the loop "$A \leftarrow A$" shows a problem with termination.

GPPE can be applied infinitely often without leading to the residual program.

Complexity:

The residual program is in general *exponential* in the size of the original program!

Consider

$$\begin{array}{lcl} p(0) & \leftarrow & \\ p(X) & \leftarrow & p(Y), succ(Y,X), \textit{not}\ q(Y) \\ p(X) & \leftarrow & p(Y), succ(Y,X), \textit{not}\ r(Y) \\ q(X) & \leftarrow & succ(X,\_), \textit{not}\ q(X) \\ r(X) & \leftarrow & succ(X,\_), \textit{not}\ r(X) \end{array}$$

together with the facts $succ(0,1), \ldots succ(n-1,n)$. The residual program contains for every $p(i)$ exactly 2^i-many rules.

Brass/Freitag/Zukowski showed how a small modification of the residual program (still satisfying the nice characterization of computing WFS), results in a *polynomial* computation.

Idea: Replace GPPE by *Loop-check* that does not produce an exponential blow up. Here are two instances of GPPE:

Definition III.5.1 (Success and Failure)
The success-transformation can be applied to a program P whenever there is a rule $A \leftarrow$ in P. Success then removes a positive occurrence of A in the body of another rule.

Dually, the failure-transformation can be applied to a program P whenever there is a an atom A which does not occurr in any head of a rule. Failure then removes a rule which contains A positively in its body.

If we forget about GPPE and the tautology rule we get the following interesting result:

Theorem III.5.2 (Linear Complexity)
The calculus of rules consisting of success, failure and reduction (see Definition III.2.1) is also confluent. In addition, it is strongly terminating and the normalform nf(P) *can be computed in linear time. The semantics according to the normalform coincides with the least fixpoint of Fittings operator Φ_P and thus coincides with $comp_3$:*

$$
\begin{aligned}
comp_3(P) = \quad & \{A : \; A \leftarrow \quad \in \boxed{\mathit{nf}(P)}\} \\
\cup \quad & \{\text{not } A : \; A \text{ is in no head of } \boxed{\mathit{nf}(P)}\}
\end{aligned}
$$

The last result is, from the complexity point of view quite promising. But it just computes a weak approximation of the WFS. What is missing is:

Definition III.5.3 (Loop Detection)
The Loop Detection transformation can be applied to a program P whenever there is a set $\mathcal{S}$ of ground atoms such that

- *for each rule in* $A \leftarrow \mathcal{B} \in P$,
  if $A \in \mathcal{S}$ *then* $\mathcal{B} \cap \mathcal{S} \neq \emptyset$.

Loop Detection then removes all rules of P which contain in their body a positive occurrence of an atom in $\mathcal{S}$.

Using our transformations, we get the following algorithm:

1. We first compute the strongly connected components of the program. As is well-known, this can be done in linear time (see eg. Tarjan 72, BerSchFra95).
2. We start with working on the lowest component and propagate the values of atoms to the higher components.
3. In every connected component we do
   (a) apply reduction as well as success and failure until a fixpoint is reached. This fixpoint corresponds exactly to the fixpoint of the Fitting operator Φ_P.
   (b) apply loop-detection, which amounts to compute the maximal unfounded set.

Theorem III.5.4 (Implementation of WFS)
The above described algorithm is provably better than the alternating fixpoint procedure of van Gelder, which already works in $\#At \times |P|$, where $\#At$ is the number of atoms and $|P|$ the size of the program.

Moreover, the algorithm is linear for the class of all programs, where in each strongly connected component the following holds: either all the head-atoms defined in that component depend only positively from each other *or* no predicate defined in that component depends positively on itself *(also called* positive-acyclic*). Note that these two conditions can change from one connected component to the other.*

Example III.5.5 (Algorithm for WFS using SCC's)
We consider the following program, which contains both negative and positive dependencies.

$$
\begin{array}{lllllllll}
p_1 & \leftarrow & q_1 & p_2 & \leftarrow & \text{not}\ q_2 & q_3 & \leftarrow & \text{not}\ w_3 \\
q_1 & \leftarrow & r_1 & q_2 & \leftarrow & \text{not}\ s_2 & w_3 & \leftarrow & \text{not}\ z_3, x_3 \\
r_1 & \leftarrow & q_1, z_3 & s_2 & \leftarrow & t_2 & z_3 & \leftarrow & \\
r_1 & \leftarrow & p_2, \text{not}\ q_3 & t_2 & \leftarrow & \text{not}\ q_3, t_2 & x_3 & \leftarrow & \text{not}\ x_3
\end{array}
$$

We have already split the program into strongly connected components and start with the lowest component, which is positive-acyclic.

1. z_3 gets immediately true, while w_3 gets false and therefore q_3 becomes true and x_3 remains undefined.

2. We propagate these values to the higher components. Note that the next component now also gets positive-acyclic, because $t_2 \leftarrow \textit{not}\ q_3, t_2$ is removed due to propagation of the truthvalue of q_3. Consequently t_2, s_2 and p_2 are false while q_2 is true.

3. The last component is purely positive, so we need the loop detection (linear) to determine that p_1, q_1, r_1 are all false.

VI. Adding Disjunction

- GCWA
- D-WFS
- DSTABLE
- Complexity and Expressibility

In Knowledge Representation it often occurs that we know $A \vee B$ without being sure which of these propositions hold. We can have information that A implies D and B implies D from which we would like to derive that D holds for sure.

Definition VI.0.6 (CWA)

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

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

1 GCWA

GCWA is defined for positive disjunctive programs

$$\boxed{A_1 \vee \ldots \vee A_n \;\leftarrow\; B_1, \ldots, B_m}$$

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

Definition VI.1.1 (GCWA)
The generalized closed world assumption GCWA of P is the semantics given by the set of all minimal Herbrand models:

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

Originally, Minker denoted by GCWA(P) a certain set of negative atoms, but we prefer here to denote by GCWA a semantics in the sense of Definition II.5.1.

GCWA is very important because

1. it plays the same role for positive disjunctive programs as the least Herbrand model M_P does for definite programs.

2. most semantics SEM defined for arbitrary disjunctive programs (i. e. with default-negation) can be characterized by reducing them to positive programs and then applying recursively GCWA.

3. in fact, an appropriate procedure iterating GCWA can "implement" such semantics SEM.

Theorem VI.1.2 (Characterization of GCWA)
Let SEM be a semantics satisfying GPPE *and* Elimination of Tautologies.

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

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

b) *If SEM is non-trivial and satisfies in addition*[a] Isomorphy *and* Relevance, *then it* $\boxed{\textit{coincides with GCWA on positive disjunctive programs}}$.

[a] See Chapter VII for the precise definitions of Relevance and Isomorphy.

Example VI.1.3 (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:

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

We expect that Fred is able to make out a cheque and that he is not (completely) disabled.

The problem is to rule out the unintended model where both arms are broken and Fred is disabled.

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

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

by

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

where $\mathcal{A} := \{A_1, \ldots, A_k\}$, $\mathcal{B}^+ := \{B_1, \ldots, B_m\}$,
$\mathcal{B}^- := \{C_1, \ldots, C_n\}$.

All our transformations generalize easily. For example:

Definition VI.2.1 (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:*

$$+\ \begin{array}{lcl} A \vee B & \leftarrow & \text{not } F \\ E & \leftarrow & A, \text{not } F \end{array} \quad \mapsto \quad \begin{array}{lcl} A \vee B & \leftarrow & \\ E & \leftarrow & A \end{array}$$

$$-\ \begin{array}{lcl} A \vee B & \leftarrow & \\ F \vee E & \leftarrow & C, \text{not } A, \text{not } B, \text{not } F \end{array} \quad \mapsto \quad \begin{array}{lcl} A \vee B & \leftarrow & \end{array}$$

Definition VI.2.2 (GPPE)
A semantics SEM satisfies GPPE if the following transformation does not change SEM:
Replace a rule $\mathcal{A} \leftarrow \mathcal{B}^+ \wedge \mathit{not}\ \mathcal{B}^-$ where $\mathcal{B}^+$ contains an atom B by the rules

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

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

$$\begin{array}{l}
B \quad \leftarrow \mathit{not}\ E \\
A' \vee B \leftarrow D, \mathit{not}\ C' \\
A \vee D \leftarrow B, \mathit{not}\ C
\end{array}
\quad \mapsto \quad
\begin{array}{lcl}
A \vee D \vee A' & \leftarrow & D, \mathit{not}\ C', \mathit{not}\ C \\
A \vee D & \leftarrow & \mathit{not}\ E, \mathit{not}\ C
\end{array}$$

Defining a disjunctive version of WFS is easy:

Definition VI.2.3 (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 ([BD95a, BD96]).

Theorem VI.2.4 (Confluent Calculus for D-WFS)
The calculus consisting of our four transformations is confluent and strongly terminating for propositional programs and fair sequences of transformations. 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 $\boxed{res(P)}$. The disjunctive wellfounded semantics of P can be read off from $\boxed{res(P)}$ as follows

$$\psi \in \text{D-WFS}(P) \iff \exists\ \mathcal{A} \subseteq \psi \text{ with } \mathcal{A} \leftarrow true \in \boxed{res(P)} \text{ or }$$
$$\exists\ \textit{not}\ A \in \psi \text{ and } A \notin Head\_atoms(\boxed{res(P)}).$$

Let us see how Example VI.1.3 is handled by D-WFS. Applying GPPE and Reduction gives us the following residual program (we consider just the $Fred$-instantiations):

$$\begin{array}{lcl}
left_use(F) & \leftarrow & \boxed{\text{not}}\ ab(left, F) \\
ab(left, F) \boxed{\vee} right_brok(F) & \leftarrow & \\
right_use(F) & \leftarrow & \boxed{\text{not}}\ ab(right, F) \\
ab(right, F) \boxed{\vee} left_brok(F) & \leftarrow & \\
left_brok(F) \boxed{\vee} right_brok(F) & \leftarrow & \\
make_cheque(F) & \leftarrow & \boxed{\text{not}}\ ab(left, F) \\
make_cheque(F) & \leftarrow & \boxed{\text{not}}\ ab(right, F)
\end{array}$$

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.

3 DSTABLE

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

Definition VI.3.1 (DSTABLE)
N is called a stable *model*[a] *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 III.4.3 — its generalization to disjunctive programs is obvious).

[a]Note that we only consider Herbrand models.

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.

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

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.

Theorem VI.3.2 (Characterization of DSTABLE)
Let SEM be a semantics satisfying GPPE, Elimination of Tautologies, *and* Elimination of Contradictions.
Then: $\boxed{\mathit{SEM}(P) \subseteq \mathit{STABLE}(P)}$.

Moreover, DSTABLE is the weakest semantics satisfying these properties.

DSTABLE is stronger than D-WFS as can be seen from Example VI.1.3. 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 (in contrast to D-WFS) can make out a cheque.

4 Complexity and Expressibility

	Complexity	
	1. ord. prog. (with functions)	*prop. prog. (no variables)*
GCWA (*P is positive*)	A: $\mathbf{\Sigma^0_1}$-compl. *not* A: $\mathbf{\Pi^0_2}$-compl.	A: **co-NP**-compl. *not* A: $\mathbf{\Pi^P_2}$-compl.
WGCWA (*P is positive*)	A: $\mathbf{\Sigma^0_1}$-compl. *not* A: $\mathbf{\Pi^0_1}$-compl.	A: **co-NP**-compl. *not* A: **linear** in $\lvert P\rvert$
PERFECT (*P is stratified*)	**arithm.**-compl.	$\mathbf{\Pi^P_2}$-compl.
WPERFECT (*P is stratified*)	**arithm.**-compl.	$\mathbf{\Pi^P_2}$-compl.
DSTABLE	$\mathbf{\Pi^1_1}$-compl. over $\mathbb{N}$	$\mathbf{\Pi^P_2}$-compl.
D-WFS	$\mathbf{\Pi^1_1}$-compl. over $\mathbb{N}$	$\mathbf{\Pi^P_2}$-compl.

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

VII. What Do We Want?

- What is the Best Semantics?
- Implementations

1 What is the Best Semantics?

In the early 90ies, a whole zoo of different semantics came up. People were not satisfied with how some particular examples were handled by existing semantics. They modified the definition a bit, and created a new semantics, often with even less convincing behaviour than before the modification.

- Small modifications in technical definitions have often totally unforeseeable global effects, even if they seem locally well motivated.

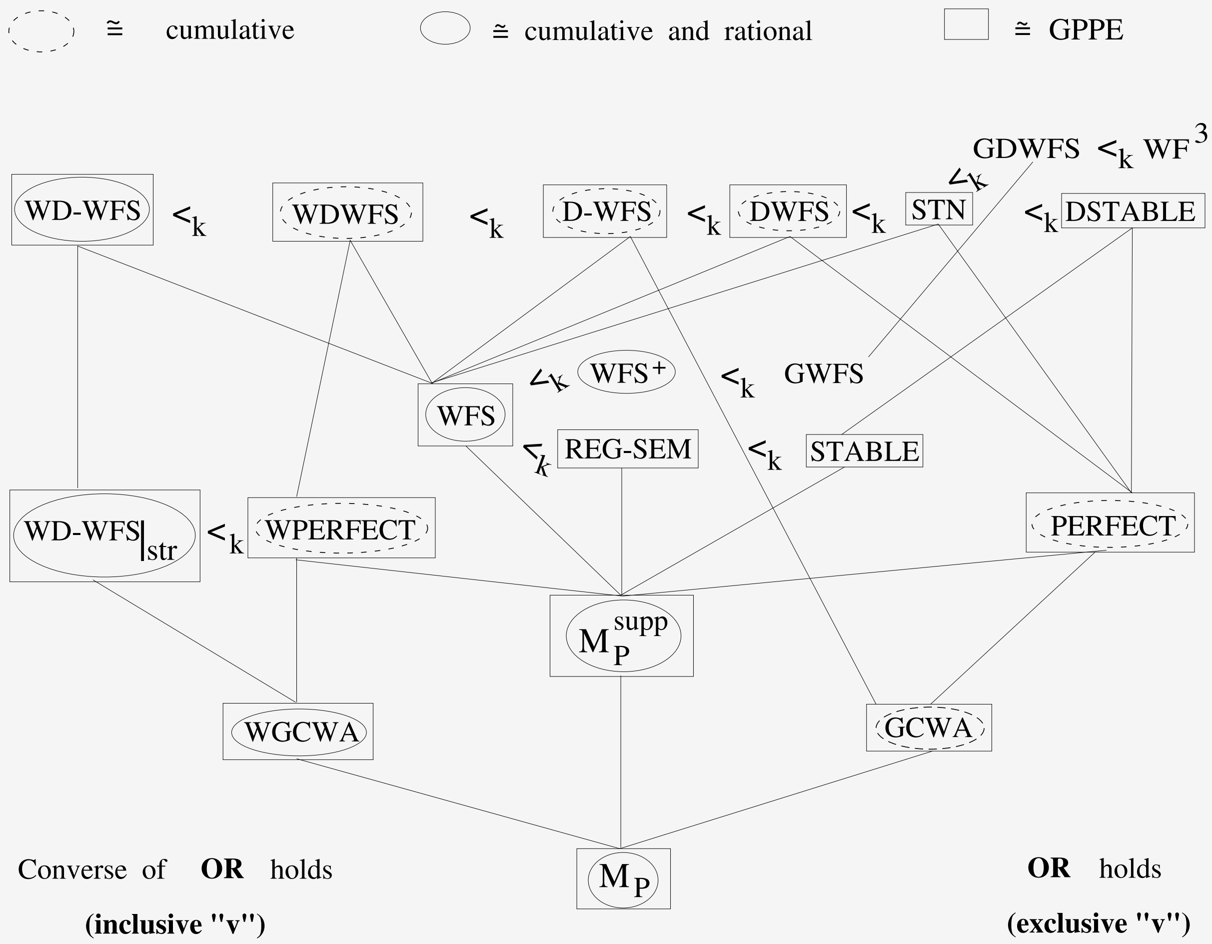
- **Abstract Properties:**
  - Cumulativity: If $\alpha \mathrel{|\!\sim} \beta$ then: $\alpha \mathrel{|\!\sim} \gamma$ **iff** $\alpha + \beta \mathrel{|\!\sim} \gamma$.
  - Rationality: If $\alpha \mathrel{|\!\not\sim} \neg\beta$ and $\alpha \mathrel{|\!\sim} \gamma$ then $\alpha + \beta \mathrel{|\!\sim} \gamma$.

While both are true for WFS, neither Rationality nor Cumulativity are true for STABLE. But the following restricted property holds and is used in Niemelae/Simons's implementation of STABLE:

If $a \in WFS(P)$ then: $\mathit{Stable}(P) = \mathit{Stable}(P \cup \{a\})$.

The same Property holds for D-WFS and D-Stable!

- **Other Properties:**
  - Goal-Orientedness (Relevance),
  - Modularity (Compositionality),
  - Partial Evaluation.



GWFS, GDWFS, WF^3 : **PPE** does not hold

DSTABLE , STABLE : **Relevance** does not hold

STN : **Modularity** does not hold

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.	•	•	•	•	•
$\mathcal{STN}$	[Prz91]	Dis.	•	•	•	•	•
STATIC	[Prz95]	Dis.	•	•	•	•	•
D-WFS	[BD95b]	Dis.	•	•	•	•	•
DWFS	[Dix92]	Dis.	•	•	•	•	•
Strong WFS	[Ros92]	Dis.	—	—	•	—	•
WD-WFS	[BD95b]	Dis.	—	•	•	—	•
WDWFS	[Dix92]	Dis.	—	•	•	—	•

2 Implementations

A web-page on LP/NMR systems has been created and is maintained continuously:

http://www.uni-koblenz.de/ag-ki/LP/

Non-Disjunctive:

- Model-generation TP (Inoue),
- Explicit Negation (Pereira et. al.),
- Fast Computation of Stable Models, handles 100000 models (Niemelä, Simons).

Non-Disjunctive:

- Implementation of WFS: XSB-System, based on SLG-Resolution (Warren et. al.).
  - Handles first-order programs and in addition it Constitutes a complete PROLOG System.
  - XSB implements Success, Failure, Reduction, Loop-Detection using Tabling in low level data-structures (WAM-level).
  - Termination is not guarantueed in general (undecidable) but in restricted cases (non-floundering Datalog). A finite propositional residual program is computed: it is the same as introduced in Chapter III.

Disjunctive:

- Model-generation TP (Inoue),
- DisLog-System (Seipel): Minker's semantics,
- System at TU Vienna (Leone/Gottlob): just started. Based on complexity analysis.

Disjunctive:

- DisLoP (Dix/Furbach): Programming System for D-WFS, DSTABLE, STATIC.
  - Taking advantage of a Tableau-Theorem Prover for classical predicate logic: Polynomial Space implementation.
  - Query-answering for pos.-disj. programs: definite answers are preferred.
  - Abductive Methods are also incorporated.
  - Implementation of first-order D-WFS on its way: Dix/Stolzenburg, Special Issue of JLP (1998): Combining CLP and NMR. Also work of Sakama/Seki, JLP(1997) 32(3)*229-245.

What is the problem of extending D-WFS?

Example VII.2.1 (Duplicate Occurrences and GPPE)
In the program Φ we apply GPPE and obtain $GPPE_B(\Phi)$.

$$\Phi: \quad A \leftarrow B \qquad GPPE_B(\Phi): \quad A \vee B \leftarrow$$
$$B \vee B \leftarrow \qquad\qquad\qquad B \vee B \leftarrow$$

$GPPE_B(\Phi)$ is irreducible, but equivalent to the program $A \vee B \leftarrow$, $B \leftarrow$, which reduces to $B \leftarrow$ only. Therefore we can derive B but not A—we can even derive $\neg A$—, if we apply our normalform-characterisation given in Theorem VI.2.4.

But the original program is certainly equivalent to $A \leftarrow B$, $B \leftarrow$, which reduces to $A \leftarrow$, $B \leftarrow$, and therefore not only B but also A is derivable.

Solve this problem with using constraints!

References

[BD95a] 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.

[BD95b] 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 J.J. Alferes and E. Orlowska, editors, *Logics in Artificial Intelligence (JELIA '96)*, LNCS 1126, pages 268–283. Springer, 1996. (Extended version will appear in the *Journal of Automated Reasoning* in 1997.).

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

[Dix92] 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.

[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.).

[Llo87] John W. Lloyd. *Foundations of Logic Programming*. Springer, Berlin, 1987. 2nd edition.

[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.

[Prz91] 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*, 14:323–357, 1995.

[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.

[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.