A General Theory of Confluent Rewriting Systems for Logic Programming and its Applications

Jürgen Dix*
The University of Manchester
Department of Computer Science
Oxford Road, Manchester M13 9PL, UK
dix@cs.man.ac.uk
Mauricio Osorio
Universidad de las Americas
Departamento de Ingenieria en Sistemas Computacionales
Sta. Catarina Martir, Cholula
72820 Puebla, Mexico
josorio@cca.pue.udlap.mx
Claudia Zepeda
Benemerita Universidad Autonoma de Puebla,
Escuela de Computacion,
75570 Puebla, Mexico
claus@solarium.cs.buap.mx

Abstract

Recently, Brass and Dix showed (*Journal of Automated Reasoning* **20(1)**, 1998) that the wellfounded semantics WFS can be defined as a confluent calculus of transformation rules. This lead not only to a simple extension to disjunctive programs (*Journal of Logic Programming* **38(3)**, 1999), but also to a new computation of the wellfounded semantics which is *linear* for a broad class of programs. We take this approach as a starting point and generalize it considerably by developing a general theory of *Confluent LP-Systems* $\mathcal{CS}$. Such a system $\mathcal{CS}$ is a rewriting system on the set of all logic programs over a fixed signature $\mathcal{L}$ and it induces in a natural way a canonical semantics. Moreover, we show four important

*This paper was finished while the first author stayed at the University of Maryland, College Park from January to October 1999.

applications of this theory: *(1) most of the well-known semantics are induced by confluent LP-systems, (2) there are many more transformation rules that lead to confluent LP-systems, (3) semantics induced by such systems can be used to model aggregation, (4) the new systems can be used to construct interesting counterexamples to some conjectures about the space of well-behaved semantics.*

AMS Classification: 03B70, 68P15, 68T27.
Keywords: Databases, Logic in Artificial Intelligence, Theory of Data, Aggregation, Logic Programming, Nonmonotonic Semantics.

1 Introduction

In this article we present a new and powerful method to not only characterize well-known semantics for logic programs (*LP semantics* but also to define new LP semantics (suitable to model aggregation). Roughly speaking, a semantics for a class of logic programs determines the set of derivable literals for each program P. Our overall method of determining such a semantics is to rewrite the program P according to certain rewrite rules. We arrive at a normalform of the original program from which we can immediately read off the derivable literals. Suitable sets of rewrite rules correspond to different normalforms and thus to different semantics.

To capture this idea of a unique normalform, we develop a general theory of *Confluent LP-Systems* $\mathcal{CS}$: these are rewriting systems based on a confluent set of transformation rules (transforming logic programs into logic programs) that induce in a canonical way a semantics SEM.

Thus we combine methods from rewriting with logic programming technology to get a powerful framework for investigating the semantics of logic programs. The starting point of our theory is an idea introduced in [BD98a] for disjunctive logic programs: *determine a set of rewriting rules that is confluent and that computes the semantics in a canonical way.* These rules transform programs into *simpler* programs. *Confluence* and *termination* guarantee that every program has associated with it a *normalform.* This normalform then induces a semantics and a simple and efficient method to answer queries with respect to this semantics[1].

It is important to note that we do not stick to a *particular semantics* but we develop a general theory of *Confluent LP Systems* $\mathcal{CS}$ and show some general lemmas governing this theory. We then consider various applications of our theory:

Known Semantics: We show that most of the classical semantics such as Fitting's 3-valued version of Clark's completion, the wellfounded semantics WFS and Schlipf's extension WFS$^+$ can be defined as confluent LP-systems in a natural way.

[1] This is what justifies our claim that our rules transform programs into *simpler* programs.

Aggregation: Modelling aggregates in semantics such as WFS is a nontrivial problem. Although a full discussion of aggregation is out of the scope of this article, we show how suitable confluent systems can be defined to solve some of the problems with aggregates. The problem is that clauses of the form $a \leftarrow \neg a$ are treated as *undefined* under WFS. As we show here this is particurlarly bad when aggregate predicates are considered: In that case a in the above rule should be treated as *true*. We solve this problem by introducing special transformation rules (motivated by aggregation) and adding them to the system corresponding to WFS. This leads to several interesting confluent LP-systems. Some of them are polynomial-time computable and also satisfy *rationality* ([Dix95a]). For a more complete elaboration of aggregation, we refer to [OJ99, OJP99].

Well-Behaved Semantics: A line of research begun in [Dix95a, Dix95b] and followed by [BD97, BD99] introduced the notion of a *well-behaved* semantics and aims at a classification of well-behaved semantics according to other, clearly formulated declarative properties. In particular, there are three conjectures in [Dix95a, Dix95b] (Conjectures 7.20, 7.21 and 7.22) stating that there are no well-behaved semantics satisfying certain properties other than the currently known semantics. We show that one of these conjectures does not hold by constructing a suitable confluent LP-system, the induced semantics of which constitutes a counterexample.

An advantage of our approach is its declarative nature. The transformation rules express natural conditions and different applications might ask for different selections of such rules. We show that the naturally induced semantics of a confluent LP-system is in fact the *weakest* semantics satisfying all our transformations.

The first proposal to provide a convincing declarative semantics to NAF was given in [Cla78] and is called *Clark's completion*. The main idea is that, to deduce negative information from a normal program, we could "complete" the program by adding the only-if halves of the definitions of the predicate symbols[2]. Clark's completion is obtained as our weakest LP-system $\mathcal{CS}_0$. However, it is now accepted that Clark's completion is often too weak and does not always capture the intended meaning of logic programs necessary, e.g. Knowledge Representation tasks ([BG94, BD98b]). Therefore stronger semantics are needed. The two main competing approaches are the *wellfounded* ([VGRS88, VGRS91]) and the *stable* semantics ([GL88, GL90]).

WFS has received great interest since its introduction, because it has several desirable properties: it is always consistent and every program possesses exactly one 3-valued model. In addition, this model is computable in polynomial-time. In contrast to this, the stable semantics is not always consistent and answering a query ψ for a program P does not depend on the ψ-relevant part of P (call-graph of P wrt. ψ): This implies that no *goal-directed* computation is possible. In fact, as shown in [DM94a, DM94b], this is an inherent feature of the stable

[2] For the details, see [Llo87].

semantics and can not be easily *defined away*. In addition, computing stable models is NP-hard and therefore can not be done in polynomial-time (unless the polynomial hierarchy collapses). Our definitions and results are given for propositional programs only, to ensure termination. But we believe that if we drop termination and only retain confluence, we can get similar results for general programs with variables (under the assumption that the normalform exists). This has been shown in [DS98] for a particular system, namely for D-WFS: the disjunctive counterpart of the wellfounded semantics.

This paper is the first detailed exposition (including full proofs) of our overall approach. Partial and preliminary results have been published as extended abstracts (without proofs) in [DO97a, DO97b, DO99]. Most of the results given in this paper can be easily generalized to disjunctive programs (due to the work in [BD98a, BD99, DS98]), but this requires some technical changes that we do not want to incorporate here.

Structure of this Article

Our article is structured as follows. After giving some background information of the concepts involved (Section 2), we introduce in Section 3 our notion of a *confluent LP-system* and develop a general theory about such systems, independent of particular sets of transformation rules. Our main result in this section is that a simple property that we call *partial distribution* implies *rationality*. In Section 4 we consider how the traditional semantics, such as Clark's completion, WFS and WFS$^+$ fit in our overall framework and can be defined in a natural way via confluent LP-systems. Section 5 provides some motivation to extend WFS in order to model aggregation in logic programming. In this section we motivate and collect various rules that are important for modelling aggregation. In Section 6 we consider a particular approach for defining semantics suited for modelling aggregation: We define suitable *provability closures*, similar to the one used for WFS$^+$. A more thorough investigation of the rules given in Section 5 is done in Section 7, where we introduce several confluent LP-systems that define new rational and polynomial-time computable semantics suitable for aggregation. Finally, in Section 8 we observe that confluent LP-systems allows us to define a semantics that integrates Clark's completion into WFS. We conclude with Section 9.

2 Background

We first collect some definitions from logic in general. Then we review some terminology from Logic Programming. We define the notion of SEM$_{min}$, stating the minimal requirements any logic programming semantics should have. Finally we provide the notion of a Rewriting System and we shortly review the notion of a *well-behaved* semantics as considered in [Dix95b].

2.1 Notions from Logic

A signature $\mathcal{L}$ is a finite set of elements that we call *atoms*. Classical propositional calculus can be defined over the set of well formed formulas defined using a signature and the two logical connectives $\neg$ and $\Rightarrow$; the rule of *modus pones* and the three axiom schemas:

(a1) $\alpha_1 \Rightarrow (\alpha_2 \Rightarrow \alpha_1)$,

(a2) $(\alpha_1 \Rightarrow (\alpha_2 \Rightarrow \alpha_3)) \Rightarrow ((\alpha_1 \Rightarrow \alpha_2) \Rightarrow (\alpha_1 \Rightarrow \alpha_3))$,

(a3) $\neg\neg\alpha_1 \Rightarrow \alpha_1$.

It is also possible to split **(a3)** in:

(b1) $(\neg\alpha_1 \Rightarrow \alpha_1) \Rightarrow \alpha_1)$, and

(b2) $\neg\alpha_1 \Rightarrow (\alpha_1 \Rightarrow \alpha_2)$.

It is not hard to show that **(a1)**, **(a2)** and **(a3)** constitute a sound and complete system for classical logic. Alternatively one can also choose needs **(a1)**, **(a2)**, **(b1)** and **(b2)**. We assume here that any formal system has the unique rule of inference of *Modus Ponens* and a variable set of axiom schemata. We denote by:

1. $\vdash_m$ provability in the formal system with just **(a1)** and **(a2)**,
2. $\vdash_{m1}$ provability in the formal system with just **(a1)**, **(a2)** and **(b1)**,
3. $\vdash_{m2}$ provability in the formal system with just **(a1)**, **(a2)** and **(b2)**,
4. $\vdash_{m12}$ provability in the formal system with just **(a1)**, **(a2)**, **(b1)**, **(b2)**.

It is well known that

- $\vdash_{m2}$ is sound with respect to intuitionistic logic, and
- $\vdash_{m12}$ corresponds to provability in full propositional logic.

A *literal* is an atom or the negation of an atom a that we denote by $\neg a$. Given a set of atoms $\{a_1, \ldots, a_n\}$, we write $\neg\{a_1, \ldots, a_n\}$ to denote $\{\neg a_1, \ldots, \neg a_n\}$.

Finally, we write MOD(T) to denote the set of all models of the theory T.

2.2 Notions from Logic Programming

The most important notion is that of a normal program rule

$$a : -l_1, \ldots, l_n,$$

where a is an atom and each l_i is a literal. We also use the notation

$$a \leftarrow \mathcal{B}^+,\ \neg\mathcal{B}^-,$$

where $\mathcal{B}^+$ contains all the positive body atoms and $\mathcal{B}^-$ contains all the negative body atoms. We sometimes use body(C) to denote $\mathcal{B}^+ \cup \neg\mathcal{B}^-$. A normal program is a finite set of normal program rules. If body(C) $= \emptyset$ then we say that the rule is a fact that we denote by $a \leftarrow$ (or just by a). Sometimes we will consider the logical constants **t** and **f** with their intended interpretation.

Let $\mathrm{Prog}_{\mathcal{L}}$ be the set of all normal propositional programs with atoms from $\mathcal{L}$. By $\mathcal{L}_P$ we understand it to mean the signature of P, i.e. the set of atoms that occur in P. We use $\models$ to denote the consequence relation for classical first-order logic. A (partial) interpretation based on a signature $\mathcal{L}$ is a disjoint pair of sets $\langle I_1, I_2\rangle$ such that $I_1 \cup I_2 \subseteq \mathcal{L}$. A partial interpretation is total if $I_1 \cup I_2 = \mathcal{L}$. Given two interpretations $I = \langle I_1, I_2\rangle$, $J = \langle J_1, J_2\rangle$, we set $I \leq_k J$ *if, by definition,* $I_i \subseteq J_i$, $i = 1, 2$. Clearly $\leq_k$ is a partial order. We may also see an interpretation $\langle I_1, I_2\rangle$ as the set of literals $I_1 \cup \neg I_2$. When we look at interpretations as sets of literals then $\leq_k$ corresponds to $\subseteq$. A general semantics SEM is a function on $\mathrm{Prog}_{\mathcal{L}}$ which associates with every program a partial interpretation. Given a signature $\mathcal{L}$ and two semantics SEM_1 and SEM_2, we define $\mathrm{SEM}_1 \leq_k \mathrm{SEM}_2$ if for every program $P \in \mathrm{Prog}_{\mathcal{L}}$, $\mathrm{SEM}_1(P) \leq_k \mathrm{SEM}_2(P)$. What are the minimal requirements we want to impose on a semantics? Certainly we want the to have the requirement that facts, i.e. rules with empty bodies, treated as being true. Dually, if an atom does not occur in any head, then its negation should be true. This gives rise to the following definition, which will play an important role later.

Definition 2.1 ($\mathrm{SEM}_{\mathbf{min}}$)
For any program P we define $\mathrm{HEAD}(P) = \{a |\ a \leftarrow \mathcal{B}^+,\ \neg\mathcal{B}^- \in P\}$ — the set of all head-atoms of P. We also define

$$\mathrm{SEM}_{\min}(P) = \langle P^{true}, P^{false}\rangle, \text{ where}$$

$$P^{true} := \{p |\ p \leftarrow \in P\}, \quad P^{false} := \{p |\ p \in \mathcal{L}_P \backslash \mathrm{HEAD}(P)\}.$$

Why do we use the symbol $:-$ or $\leftarrow$ in a logic program, and not the more intuitive material implication $\Rightarrow$ from classical logic? The reason is that semantics of logic programs give "$\leftarrow$" a different meaning. For example in classical logic the formula "$\neg a \Rightarrow a$" is equivalent to "a". But adding the clause "$a \leftarrow \neg a$" to a logic program usually changes the semantics drastically and is not equivalent to just adding the fact "a". Nevertheless, it often makes sense to view a logic program as a classical theory. To this end, we have to translate such a program P, formulated in the language defined above, into the language of classical logic.

Definition 2.2 (Translation $\mathbf{tr}(P)$)
For any clause C of the form $a \leftarrow l_1, \ldots, l_n$, we write $\mathbf{tr}(C)$ to denote the well formed formula associated with C, i.e. the formula $l_1 \Rightarrow (l_2 \Rightarrow \cdots \Rightarrow l_n \Rightarrow a) \ldots)$. For a program P, we write $\mathbf{tr}(P)$ to denote the conjunction $\bigwedge_{C \in P} \mathbf{tr}(C)$.

2.3 Rewriting Systems and Well-behaved Semantics

Definition 2.3 (Rewriting System)
An *abstract rewriting system* is a pair $\langle S, \rightarrow \rangle$ where $\rightarrow$ is a binary relation on a given set S. Let $\rightarrow^*$ be the reflexive, and transitive closure of $\rightarrow$. When $x \rightarrow^* y$ we say that x *reduces to* y. An *irreducible* element is said to be in *normalform*. We say that a rewriting system is

noetherian: if there is no infinite chain $x_1 \rightarrow x_2 \rightarrow \cdots \rightarrow x_i \rightarrow x_{i+1} \rightarrow \ldots$, where for all $i$the elements x_i and x_{i+1} are different,

confluent: if whenever $u \rightarrow^* x$ and $u \rightarrow^* y$ then there is a z such that $x \rightarrow^* z$ and $y \rightarrow^* z$,

locally confluent: if whenever $u \rightarrow x$ and $u \rightarrow y$ then there is a z such that $x \rightarrow^* z$ and $y \rightarrow^* z$.

In a noetherian and confluent rewriting system, every element x reduces to a unique normalform that we denote by $norm(x)$.

Given Definition 2.1 what further properties should a semantics satisfy? For example a semantics should be closed under the rules of the given program, i.e. it should be a partial model of the program. There are many more such properties and we will briefly discuss some of them without giving a precise definition. Interested readers may consult [Dix95a, Dix95b] where all these properties are investigated at length. If an atom a is contained in SEM(P) and b is contained in SEM($P \cup \{a\}$), then b should already belong to SEM(P): we call this property *cut*. Also, to do efficient query-answering, answering a query with respect to a program should only depend on the call-graph of the program with respect to the query: we call this property *relevance*. Another useful property is that the introduction of new names (introducing a new atom a_{new} by a rule $a_{new} \leftarrow$ body) should be a conservative extension, i.e. should not affect the semantics of the other literals (we call it *weak PPE*). Finally, we want that tautological clauses (clauses that contain the same atom in their head and in their body) can be removed without changing the semantics (*tautology*). A semantics having all these properties is called *well-behaved* and this notion was studied in detail in [Dix95b]:

Definition 2.4 (Well-Behavedness)
A semantics satisfying *relevance*, *weak partial evaluation*, *tautology* and *cut* is called well-behaved.

One of the main results of [Dix95b] is that WFS is the *weakest* well-behaved semantics under the $\leq_k$ ordering.

The following principles are not considered part of the definition of a well-behaved semantics, but are candidates for a stronger definition of a well behaved semantics, see [BD00, BDK97, Dix95a, Dix95b].

The principle of **GPPE** allows us to replace any positive occurrence of an atom A by its associated defining rules, without any restrictions[3]. The principle of *cautious monotonicity* states that, for every program P and every finite set of ground atoms A and every set of finite literals R, if $A \subseteq \mathrm{SEM}(P)$ and $R \subseteq \mathrm{SEM}(P)$ then $R \subseteq \mathrm{SEM}(P \cup A)$ [Dix95b]. We say that a semantics is *cumulative* if it is *cautious monotonic* and satisfies *cut*. A semantics SEM is *rational* if for every atom a that is not false in $\mathrm{SEM}(P)$ implies $\mathrm{SEM}(P) \leq \mathrm{SEM}(P \cup \{a\})$.

3 Non-Monotonic Confluent LP-Systems

The main concept on which our notion of a LP-system is based, is the concept of a *transformation* rule.

Definition 3.1 (Basic Transformation Rules)
A transformation rule is a binary relation on $\mathrm{Prog}_{\mathcal{L}}$. The following transformation rules are called *basic*. Let a program $P \in \mathrm{Prog}_{\mathcal{L}}$ be given.

RED$^+$: This transformation can be applied to P, if there is an atom a which does not occur in HEAD(P). **RED$^+$** transforms P to the program where all occurrences of $\neg a$ are removed.

RED$^-$: This transformation can be applied to P, if there is a rule $a \leftarrow \in P$. **RED$^-$** transforms P to the program where all clauses that contain $\neg a$ in their bodies are deleted.

SUB: This transformation can be applied to P, if P contains two clauses $a \leftarrow \mathrm{body}_1$, and $a \leftarrow \mathrm{body}_2$, where $\mathrm{body}_1 \subseteq \mathrm{body}_2$. **SUB** transforms P to the program where the clause $a \leftarrow \mathrm{body}_2$ has been removed.

Although these rules are not really *functions* on $\mathrm{Prog}_{\mathcal{L}}$ (e.g. **RED$^-$** is only determined if an occurrence of a certain rule is distinguished), they induce a set of operators on $\mathrm{Prog}_{\mathcal{L}}$ as we will show. An operator, denoted as *op*, is a function over the set of programs that transforms a program P to a program P^{op} as follows. If C_1, C_2 are clauses, g a literal and f an atom, then *op* can be of type

Red$^+$, then P^{op} is the reduction of P with respect to C_1 and g.
We write $\langle \mathbf{RED}^+, C_1, g \rangle$.

Red$^-$, then P^{op} is the reduction of P with respect to C_1 and g.
We write $\langle \mathbf{RED}^-, C_1, g \rangle$.

SUB, then P^{op} is the application of subsumption of C_1 to C_2.
We write $\langle \mathbf{SUB}, C_1, C_2 \rangle$ (C_2 is the clause to be deleted).

[3] **GPPE** will be formally defined in Section 4.

If the transformation can not be applied then the operator behaves as the identity function. On the other hand, if $P \neq P^{op}$, where P is program and op an operator, we say that op is executed.

Example 3.2 (Operators) If P is $\{c \leftarrow \neg d\}$ and $op1$ is $\langle \mathbf{RED}^+, \mathrm{c} \leftarrow \neg d, \neg d \rangle$ then P^{op1} is $\{c\}$. If $op2$ is $\langle \mathbf{RED}^+, \mathrm{c} \leftarrow \neg d, \neg c \rangle$ then $P^{op2} = P$. So, in the second case $op2$ is not executed.

Below we will intoduce several transformations that induce operators in this way. Obviously, the just mentioned transformations are among the minimal requirements that a *well-behaved* semantics should have (see [Dix95b]). From now on, when we speak of a semantics SEM, we understand that SEM is invariant under the transformations $\mathbf{RED}^+$, $\mathbf{RED}^-$ and $\mathbf{SUB}$. Given a program P and a list of operators *ops*, we define the application of *ops* to P, denoted as P^{ops}, as follows (using PROLOG list notation):

$$\begin{array}{lcl} P^{[]} & := & P \\ P^{[op|ops]} & := & (P^{op})^{ops}. \end{array}$$

We define the size of *ops* as the number of elements of *ops*. We are now ready to introduce our main notion:

Definition 3.3 (Confluent LP-system $\mathcal{CS}$)
A *confluent LP-system* $\mathcal{CS}$ over the signature $\mathcal{L}$ is a pair

$$\langle \{op_i \mid i = 1, \ldots, n\}, \rightarrow \rangle$$

that satisfies the following conditions:

1. $\{op_i \mid i = 1, \ldots, n\}$ is a finite set of transformation rules on $\mathrm{Prog}_{\mathcal{L}}$. By abuse of language we often view them as (computable) operators as we explained before.

2. $\langle \mathrm{Prog}_{\mathcal{L}}, \rightarrow \rangle$, where $\rightarrow$ is the union of all the transformation rules in $\mathcal{CS}$, is a *noetherian* and *confluent* rewriting system.

3. If $P \rightarrow P_1$ then $\mathrm{SEM}_{\min}(P) \leq_k \mathrm{SEM}_{\min}(P_1)$.

We denote the uniquely determined normalform of a program P with respect to the LP-system $\mathcal{CS}$ by $norm_{\mathcal{CS}}(P)$. Every LP-system $\mathcal{CS}$ induces a semantics $\mathrm{SEM}_{\mathcal{CS}}$ as follows:

$$\mathrm{SEM}_{\mathcal{CS}}(P) := \mathrm{SEM}_{\min}(norm_{\mathcal{CS}}(P))$$

When there is no ambiguity about the given confluent LP-system we drop the subscripts. Observe that $\mathrm{SEM}_{\min}$ is a semantics as defined in section 2.

3.1 Basic Results

The following theorem shows very clearly the declarative character of the induced semantics of a *confluent LP-system* $\mathcal{CS}$:

Theorem 3.4 ($\mathbf{SEM}_{\mathcal{CS}}(P)$ and weakest Semantics)
Let $\mathcal{CS}$ be a confluent LP-system *over the signature $\mathcal{L}$. If there exists a weakest semantics SEM that satisfies all the transformation rules of $\mathcal{CS}$, then this semantics coincides with the semantics induced by $\mathcal{CS}$.*

Note that a weakest semantics need not neccessarily exist: the transformations need not be closed under intersections. In such cases, there exist several incomparable semantics, but not a weakest. This is a particular important point if we are in a situation where a lot of different queries are posed to a program P under a certain semantics SEM. Instead of starting special procedures that take the query into account, we *compile* the program, which means to compute the normalform. Once the normalform is computed, queries can be answered immediately. In general compiled programs may be very large, but for our proposed semantics in Section 7 this can not happen.

Lemma 3.5
For every confluent LP-system $\mathcal{CS}$ *and every program P such that every step (required to obtain P^{ops} from P) is executed we have: $SEM_{min}(P) \leq_k SEM_{min}(P^{ops})$.*

Proof. Straightforward induction on the size of *ops* using condition 3 of the definition of *LP-systems*. **Q.E.D**

Corollary 3.6
For every program P of a confluent LP-system $\mathcal{CS}$*: $SEM_{min}(P) \leq_k SEM_{\mathcal{CS}}(P)$.*

Proof. Let *ops* be such that $norm_{\mathcal{CS}}(P) = P^{ops}$. Then we have

$$\mathrm{SEM}_{\mathcal{CS}}(P) = \mathrm{SEM}_{\min}(norm_{\mathcal{CS}}(P)) = \mathrm{SEM}_{\min}(P^{ops}).$$

By Lemma 3.5, we get $\mathrm{SEM}_{\min}(P) \leq_k \mathrm{SEM}_{\mathcal{CS}}(P)$ as desired. **Q.E.D**

The following condition is also desirable in logic programming.

Definition 3.7 (3-value based)
A semantics SEM is called *3-value based* if for every program P the partial interpretation $\mathrm{SEM}(P)$ is a 3-valued model of P.

In 3-value based semantics, it can not happen that the body of a rule evaluates to *true* in $\mathrm{SEM}(P)$ while the head of this rule evaluates to *false* or to *undefined*. This is a very mild restriction and all semantics proposed in the literature satisfy it. The interested reader might ask about our underlying notion of a *3-valued model* of P. First of all, we translate P into the language of classical logic by considering $\mathbf{tr}(P)$. Then, we are considering $\Rightarrow$ as the Kleeene connective: an implication "body $\Rightarrow$ head" is true not only if

1. the body is false, or
2. the body is true and the head is true, but also if
3. both the body and head are undefined.

For the details of 3-value based semantics see [Dix95a, BD97, BD99, BDNP99].

3.2 Rationality

A semantics SEM is called *rational*, if for every atom a and program P the following holds

$$\neg a \notin \mathrm{SEM}(P) \;\; \text{implies} \;\; \mathrm{SEM}(P) \leq_k \mathrm{SEM}(P \cup \{a \leftarrow\}).$$

What is rationality good for? Sometimes, this property implies some nice computational properties. For example, rationality implies cautious monotonicity:

$$a \in \mathrm{SEM}(P) \;\; \text{implies} \;\; \mathrm{SEM}(P) \leq_k \mathrm{SEM}(P \cup \{a \leftarrow\}).$$

As all semantics satisfy the cut property:

$$a \in \mathrm{SEM}(P) \;\; \text{implies} \;\; \mathrm{SEM}(P \cup \{a \leftarrow\}) \leq_k \mathrm{SEM}(P),$$

we have the following property for cumulative semantics:

$$a \in \mathrm{SEM}(P) \;\; \text{implies} \;\; \mathrm{SEM}(P) = \mathrm{SEM}(P \cup \{a \leftarrow\}).$$

This tells us the following. Suppose, our overall goal is to compute $\mathrm{SEM}(P)$. Suppose also that in the course of deriving elements of $\mathrm{SEM}(P)$ we have already derived an atom a. Cautious monotonicity tells us, that we can safely assume a without changing the semantics! Thus we can make use of a and perhaps simplify the program.

Even semantics that do not satisfy cumulativity, such as the stable semantics, may satisfy a related property. In fact, as first been noticed for the non-disjunctive case of the stable semantics by Schlipf (see [Dix95b, Theorem 3.7]), the following holds:

If $a \in WFS(P)$ then P and $P \cup \{a \leftarrow\}$ have identical stable models.

Now this property can be efficently used for computing stable models. This means that it always pays off to first compute the wellfounded semantics (which can be done in polynomial-time, as opposed to computing stable models, which is one level higher in the polynomial hierarchy) and to use the results to simplify the program.

In fact, this is one of the techniques employed in [NS96] which is currently the fastest method to compute stable models (see also [DFN00]). Therefore if we know a semantics is rational, it is also cumulative and satisfies in particular the above property.

Definition 3.8 (Partial Distribution)
A *confluent LP-system* $\mathcal{CS}$ satisfies *partial distribution* if for every *op*, P and a such that a $\in$ HEAD(P^{op}) and P^{op} is executed, then $(P \cup \{a\})^{op} = P^{op} \cup \{a\}$.

Lemma 3.9
For every partial distributive confluent LP-system $\mathcal{CS}$ *the following is true. Let* $a \in HEAD(P^{ops})$*, where ops is any list of operators such that every step in* P^{ops} *is executed. Then:* $(P \cup \{a\})^{ops} = P^{ops} \cup \{a\}$.

Proof. Straightforward induction on the size of *ops*. **Q.E.D**

Rationality is a nice property for a semantics SEM, especially if used in logic programming. The following result is important because it shows that a relatively simple condition, namely partial distribution on $\mathcal{CS}$ suffices to ensure rationality of the induced semantics.

Theorem 3.10 (SEM$_{\mathcal{CS}}$ of Partial distributive $\mathcal{CS}$ is rational)
Every partial distributive confluent LP-system $\mathcal{CS}$ *induces a rational semantics* $SEM_{\mathcal{CS}}$*, where cut holds.*

Proof. First, we prove rationality. Let *ops* be such that $\mathrm{SEM}(P) = \mathrm{SEM}_{\min}(P^{ops})$. If $\neg\, a \notin$ SEM(P) then $a \in \mathrm{HEAD}(P^{ops})$. By Lemma 3.9, $P^{ops} \cup \{a\} = (P \cup \{a\})^{ops}$, and since $\mathrm{SEM}_{\min}(P^{ops}) \leq_k \mathrm{SEM}_{\min}(P^{ops} \cup \{a\})$, we get that $\mathrm{SEM}_{\min}(P^{ops}) \leq_k \mathrm{SEM}_{\min}((P \cup \{a\})^{ops})$. By Corollary 3.6 $\mathrm{SEM}_{\min}((P \cup \{a\})^{ops}) \leq_k \mathrm{SEM}((P \cup \{a\})^{ops})$. Therefore $\mathrm{SEM}(P) \leq_k \mathrm{SEM}((P \cup \{a\})^{ops})$ **(1)**. By confluence $norm((P \cup \{a\})^{ops}) = norm(P \cup \{a\})$ that implies $\mathrm{SEM}((P \cup \{a\})^{ops}) = \mathrm{SEM}(P \cup \{a\})$ **(2)**. Therefore, by **(1), (2)** $\mathrm{SEM}(P) \leq_k \mathrm{SEM}(P \cup \{a\})$.

Now, we prove cut. We need to prove:

$$a \in \mathrm{SEM}(P) \text{ and } b \in \mathrm{SEM}(P \cup \{a\}) \text{ then } b \in \mathrm{SEM}(P)$$

Let us assume partial distribution, $b \notin \mathrm{SEM}(P)$ and $a \in \mathrm{SEM}(P)$. We need to prove $b \notin \mathrm{SEM}(P \cup \{a\})$. Let *ops* be such that $\mathrm{SEM}(P) = SEM_{min}(P^{ops})$. Since $a \in \mathrm{SEM}(P)$ then $a \in HEAD(P^{ops})$ and every step in P^{ops} is executed. By partial distribution $(P \cup \{a\})^{ops} = P^{ops} \cup \{a\}$ and since $a \in \mathrm{SEM}(P)$ we know that $a \in P^{ops}$ and so $P^{ops} \cup \{a\} = P^{ops}$, therefore $(P \cup \{a\})^{ops} = P^{ops}$. Thus, P^{ops} is the normalform of P but also of $P \cup \{a\}$. So, $b \notin \mathrm{SEM}(P)$ implies that $b \notin \mathrm{SEM}(P \cup \{a\})$ as desired. **Q.E.D**

However, the other direction of the last theorem does not hold, as the following example shows.

Example 3.11 (Rational Sem. induced by $\mathcal{CS}$ but not partial distr.) Let the system $\mathcal{CS}$ consist of just the following rule P reduces to $(P \setminus \{a \leftarrow \neg a\}) \cup \{a\}$ *if, by definition,* $\{a \leftarrow \neg a\}) \in P$ and a does not occur in $P \setminus \{a \leftarrow \neg a\}$.

It is easy to check confluence and rationality. However, partial distribution fails: take the program P consisting solely of $a \leftarrow \neg a$. Then this program reduces to $\{a\}$. But $P \cup \{a\}$ is irreducible.

Some points are worth mentioning:

1. Thanks to *confluence* and *termination*, our transformation rules have both a *declarative* and an *operational* meaning. From the declarative point of view, they tell us that our semantics is closed under the given transformation rule. From an operational point of view the transformations are computable functions that can be applied to *simplify* the program. When we have derived the normalform of the program, then computing its semantics is immediate.

2. Rationality is also an interesting property of semantics in logic programming. Confluence plus *partial distribution* implies rationality. It is straightforward to see that most of our semantics (except WFS^0 and CWFS) satisfy partial distribution. Therefore we know that these semantics are rational.

3. The nature of simple transformations, confluence and termination allows us to prove many properties by induction on the number of steps needed to compute the normalform.

4 Clark's Completion, WFS and WFS^+

In this section we show that the 3-valued version of Clark's completion semantics (introduced by Fitting in [Fit85, Fit86]), WFS ([VGRS88, VGRS91]) and WFS^+ (an extension of WFS introduced independently by [Dix92] and [Sch92]) are induced by *confluent LP-systems*.

To this end, we introduce the following transformation rules:

GPPE: *(Generalized Principle of Partial Evaluation)* Suppose P contains $a \leftarrow \mathcal{B}^+, \neg\mathcal{B}^-$ and we fix an occurrence of an atom $g \in \mathcal{B}^+$ different from a. Then we can replace $a \leftarrow \mathcal{B}^+, \mathcal{B}^-$ by the n clauses $(i = 1, \ldots, n)$

$$a \leftarrow (\mathcal{B}^+ \backslash \{g\}) \cup {B_i}^+,\ \neg\mathcal{B}^- \cup \neg {B_i}^-$$

where $g \leftarrow {B_i}^+,\ \neg {B_i}^- \in P$, (for $i = 1, \ldots, n$) are all clauses with g in their heads. If no such clauses exist, we simply delete the former clause.

TAUT: *(Tautology)* Suppose P contains a rule C which has the same atom in its head and in its positive body. Then we can remove this rule.

LC: *(Logical Consequence)* Suppose $P \vdash a$[4] for an atom a. Then we can add the rule $a \leftarrow$ to P.

[4] Here we view P as a classical propositional theory and $\models$ as the classical entailment relation. To be technically precise, we have to use the translation $\mathbf{tr}(P)$ defined in Definition 2.2 and write: $\mathbf{tr}(P) \vdash a$.

Definition 4.1 (Systems $\mathcal{CS}_0$, $\mathcal{CS}_1$, $\mathcal{CS}_2$)
Let $\mathcal{CS}_0$ be the *LP-system* which contains, besides the basic transformation rules, the rule **GPPE**. Let $\mathcal{CS}_1$ be $\mathcal{CS}_0$ enlarged by **TAUT**. Finally, let $\mathcal{CS}_2$ be $\mathcal{CS}_1$ plus the rule **LC**.

4.1 Fittings comp$_3$ and Wellfounded Semantics

By using results of [Dix95a, BD98a] we get

Theorem 4.2 (Classifying comp$_3$, WFS)

1. *Fitting's semantics comp$_3$ is the weakest 3-value based semantics satisfying **GPPE**. It is induced by the* confluent LP-system $\mathcal{CS}_0$.

2. *The wellfounded semantics is the weakest 3-value based semantics satisfying **GPPE** and **TAUT**. It is induced by the* confluent LP-system $\mathcal{CS}_1$.

Proof. The confluence result for $\mathcal{CS}_0$ is obtained from [BD98a, Theorem4.3] by just restricting to non-disjunctive programs. It is easy to see that comp$_3$ satisfies all these transformations. Therefore

$$\text{comp}_3(P) = \text{comp}_3(norm_{\mathcal{CS}_0}(P)).$$

We now have to check whether $\text{SEM}_{\mathcal{CS}_0}(P) = \text{comp}_3(P)$, where P is completely reduced with respect to $\mathcal{CS}_0$. This is done by comparing the least fixpoint of Fitting's operator Φ_P with $\text{SEM}_{\min}(P)$. In fact, when no more applications can be applied to P, then a straightforward induction shows that $lfp(\Phi_P) = \text{SEM}_{\min}(P)$, where "=" denotes equality between 3-valued models. The confluence result for $\mathcal{CS}_0$ is an immediate corollary of [BD98a, Theorem 4.3]. **Q.E.D**

Example 4.3 (Computing WFS) We consider the program P_1 and reduce it as follows:

$$\begin{array}{l} p \\ q \leftarrow \neg p \\ q \leftarrow t, \neg p \\ s \leftarrow \neg q \\ q \leftarrow r \\ r \leftarrow q \end{array} \rightarrow_{\mathbf{SUB}} \begin{array}{l} p \\ q \leftarrow \neg p \\ s \leftarrow \neg q \\ q \leftarrow r \\ r \leftarrow q \end{array} \rightarrow_{\mathbf{RED^-}} \begin{array}{l} p \\ s \leftarrow \neg q \\ q \leftarrow r \\ r \leftarrow q \end{array}$$

In the next step, we can apply **GPPE** to one of the two last rules to get:

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

Now we can delete the resulting tautology by the application of **TAUT** and then use $\mathbf{Red}^+$

$$\begin{array}{llll} p & & & \\ s & \leftarrow & \neg q & \\ r & \leftarrow & q & \end{array} \quad \rightarrow_{\mathbf{RED}^+} \quad \begin{array}{llll} p & & & \\ s & & & \\ r & \leftarrow & q & \end{array}$$

Finally applying **GPPE** to the last one, we get to $norm_{\mathcal{CS}_1}(P_1)$:

$$\begin{array}{l} p \\ s \end{array}$$

Thus, the wellfounded semantics of P_1 is:

$$\mathrm{WFS}(P_1) = \{p, s, \neg q, \neg t, \neg r\}$$

Let us note that although the $\mathcal{CS}_1$ system has the nice property of confluence (and termination), its computational properties are not that efficient. In fact, computing the normalform is exponential, whereas it is known that the WFS can be computed in quadratic time. Recently an equivalent confluent LP-system was defined, where the normalform can be computed in *quadratic* time (see [BZF97, BDFZ98]). The idea is to restrict the **GPPE** rule and to replace it by two simple instances plus an additional check for an unfounded set (Loop detection):

Success: Suppose that P includes a fact a and a clause $q \leftarrow \mathrm{body}$ such that $a \in \mathrm{body}$. Then we replace the clause $q \leftarrow \mathrm{body}$ by $q \leftarrow \mathrm{body} \setminus \{a\}$.

Failure: Suppose that P contains a clause $q \leftarrow \mathrm{body}$ such that $a \in \mathrm{body}$ and $a \notin HEAD(P)$. Then we erase the given clause.

Loop: We say that P_2 results from P_1 by $Loop_A$ if, by definition, there is a set A of atoms such that

1. for each rule $a \leftarrow \mathrm{body} \in P_1$, if $a \in A$, then $\mathrm{body} \cap A \neq \emptyset$,
2. $P_2 := \{a \leftarrow \mathrm{body} \in P_1 : \mathrm{body} \cap A = \emptyset\}$,
3. $P_1 \neq P_2$.

We call the resulting system $\mathcal{CS}_{1'}$ (**RED**$^+$+**RED**$^-$+**Success**+**Failure**+**Loop**). At first sight, **Failure** looks as a special case of **Loop**: simply set A in **Loop** to $\{a\}$ where $a \notin HEAD(P)$. But **Loop** forces us to remove all occurrences of a failed atom at once: **Failure** is more liberal. But it is true that a block of **Failure** applications can be simulated by **Loop**.

Theorem 4.4 (Elimination of Failure, [Zep97])
*Let P be a program. Let P' the normalform of P w.r.t. the $\mathcal{CS}_{1'}$ system, then P' can also result from P by the same system but without **Failure**.*

4.2 WFS$^+$

The WFS$^+$-semantics (introduced independently by [Dix92] and [Sch92]) is an extension of WFS.

Definition 4.5 (WFS$^+$)
Let P be a program, $M_0 := \emptyset$ and $M_{i+1} := \mathbf{T}(\mathrm{WFS}(P \cup M_i))$ where $\mathbf{T}(I)$ stand for the set of all atoms which are true in all models of P that are consistent with I. Let

$$\mathrm{WFS}^+(P) := \mathrm{WFS}(P \cup M_j),$$

where j is the least natural number such that $M_j = M_{j+1}$.

Definition 4.6 (Supraclassicality, [Dix95a])
A semantics SEM satisfies supraclassicality *if, by definition,* for every program P, if $P \models a$ then $a \in SEM(P)$, where a is an atom.

Theorem 4.7 (Classifying WFS$^+$)
The WFS$^+$ semantics is the weakest 3-value based semantics satisfying ***Success***, ***Failure***, ***Loop***, ***TAUT*** *and* ***LC****. It is induced by the* confluent LP-system *consisting of exactly these transformations (plus the basic ones introduced in Section 3).*

Proof. It has been shown in [Dix95a] that WFS$^+$ is the weakest extension of WFS satisfying Cut and **LC**. Our transformation rule **LC** corresponds quite closely to Supraclassicality. Cut is implied by **RED**$\pm$. Because of the results of [BDFZ98], **Success**, **Failure**, **Loop** and **TAUT** suffice to reconstruct WFS. Since WFS$^+$ satisfies all these transformations, it follows that WFS$^+$ is the weakest 3-value based semantics satisfying **Success**, **Failure**, **Loop**, **TAUT** and **LC**. The confluence result follows from the confluence of the rules **Success**, **Failure**, **Loop**, **TAUT**, **RED**$\pm$ and **SUB** (proved in [BZF97]) and showing that the calculus remains locally confluent when **LC** is added. This is done by checking carefully the interaction of **LC** with the other rules. **Q.E.D**

The reason we gave up **GPPE** is not only for computational purposes. As has been shown in [DM94a] **GPPE** does not hold for WFS$^+$. Let us consider the programs:

$$\begin{array}{lllll} P: & a & \leftarrow & \neg b \\ & b & \leftarrow & \neg a \\ & x & \leftarrow & a \\ & x & \leftarrow & b \end{array} \qquad\qquad \begin{array}{lllll} P': & a & \leftarrow & \neg b \\ & b & \leftarrow & \neg a \\ & x & \leftarrow & \neg a \\ & x & \leftarrow & \neg b \end{array}$$

By applying **GPPE** twice (to replace a in the third and b in the fourth rule), we get P'. But $\mathrm{WFS}^+(P) = \{x\}$ while $\mathrm{WFS}^+(P') = \emptyset$. We point out that the WFS for both P and P' is $\emptyset$, which is considered as a shortcoming of WFS. A semantics stronger than WFS but weaker than WFS$^+$ is introduced in [DO97c].

The idea of its definition is to keep the original program and adding more and more atoms to it. This defines a sequence of sets of atoms $\mathcal{A}_i$. These atoms are obtained by applying our old transformations to the program from the last step. Here is the exact definition:

Definition 4.8 (WFS^2, [DO97c])
The extended wellfounded semantics, denoted as $\mathrm{WFS}^2(P)$, is defined through the following increasing sequence: $(\mathcal{A}_0 = \emptyset)$

$$\begin{array}{ll} \mathcal{A}_{i+1} & := T(P \cup \mathcal{A}_i) \\ T(P) & := \{a : \ a \leftarrow \quad \text{or } a \leftarrow \neg a \text{ belongs to } norm_{cs_1}(P)\} \end{array}$$

There is the least number n_0 such that $\mathcal{A}_{n_0} = \mathcal{A}_{n_0+1}$ (note that $\mathcal{A}_i \subseteq \mathcal{A}_{i+1}$ and $\mathrm{size}(norm_{cs_1}(P + \mathcal{A}_{i+1})) < \mathrm{size}(norm_{cs_1}(P + \mathcal{A}_i))$ for $\mathcal{A}_i \neq \mathcal{A}_{i+1}$).
We define $\mathrm{WFS}^2(P) := \mathrm{WFS}(P \cup \mathcal{A}_{n_0})$.

Example 4.9 (WFS^2) We consider the program P_{example} :

$$\begin{array}{lll} a & \leftarrow & \neg a \\ p & \leftarrow & a \end{array}$$

By definition , $A_0 = \emptyset$.

1. Using the transformations (in this case only **GPPE**)

   $$norm_{cs_1}(P_{\text{example}}) = \{a \leftarrow \neg a, p \leftarrow \neg a\}$$

   thus $A_1 = T(P_{\text{example}} \cup A_0) = \{a\}$.

2. Using again the transformations for $P_{\text{example}} \cup A_1$:

   $$\begin{array}{lll} a & & \\ a & \leftarrow & \neg a \\ p & \leftarrow & a \end{array}$$

   we have $norm_{cs_1}(P_{\text{example}}) \cup A_1 = \{a, p\}$ and $A_2 = T(P_{\text{example}} \cup A_1) = \{a, p\}$.

3. Now using them for $P_{\text{example}} \cup A_2$:

   $$\begin{array}{lll} a & & \\ p & & \\ a & \leftarrow & \neg a \\ p & \leftarrow & a \end{array}$$

   and so: $norm_{cs_1}(P_{\text{example}}) \cup A_2 = \{a, p\}$ and: $A_3 = T(P_{\text{example}} \cup A_2) = \{a, p\}$.

Therefore, $n_0 = 2$ is the least number such that $A_1 = A_2$. Thus

$$\mathrm{WFS}^2(P_{\mathrm{example}}) := \mathrm{WFS}(P_{\mathrm{example}} \cup A_2) = \{a, p\}$$

The connection between WFS^2 and partial-order programming is considered in [DO97c]. In this article we show that there is a simpler characterization of WFS^2 based on confluent systems (Theorem 7.5).

5 Aggregation and set-grouping

This section provides a particularly important application area for our confluent LP-systems: *aggregate functions*. We describe the problem of aggregation and show that in order to come up with a meaningful semantics, we have to extend the wellfounded semantics. It turns out that quite simple transformation rules (like *Supraclassicality* introduced in Definition 4.6) are suitable to do this. While this section is devoted to *motivating* such rules, the next section will present the framework of *provability closures* in order to investigate these rules more deeply and integrate them into our confluent LP-systems. *Set-grouping* and *aggregation* are powerful operations of practical interest in database query languages.

An aggregate operation is a function that maps a set to some value, e.g., the *maximum* or *minimum* in the set, the *cardinality* of this set, the *summation* of all its members, and similar operations. Since aggregate operations are typically non-monotonic in nature, recursive programs making use of aggregate operations can not be naturally expressed via monotonic first order logic. As pointed out recently (see [AMTW99] and [Dix98]) there is currently no clear semantics or efficient semantics of constructs like `setof` or similar aggregates. We have done some research on this topic in [OJP99, OJ99] and we have experimental evidence to make us think that our results, as described in this paper, will help to tackle this problem. The idea of expressing *aggregation programs* via *logic programs with some form of negation-as-failure* has been considered several times, see [VG92, OJ99]. The reason for this is that negation-as-failure is a well-known non-monotonic operation that has been investigated extensively in the logic programming field during the last 20 years.

The main idea is to consider a specification involving aggregation as a *macro* for a *normal program*. Note that the translation of a program with aggregation will usually contain *negations*. Such a translation helps us to understand on a declarative level, the connection between various closely related constructs. This approach has been used in the literature to explain new constructs; Sacca/Zaniolo's LDL1 *choice construct* provides but one example of this nature ([SZ91]). Here, the choice construct is modelled by taking the stable semantics as the underlying semantics of the translated normal program.

In [VG92], the authors mainly investigated monotonic aggregation and used the WFS as the underlying semantics. We build on their approach, and expect to contribute to the understanding of the nature of aggregation, set-grouping,

negation-as-failure, and the relationship between them. To this end, our aimis to use the WFS$^+$ semantics as a suitable underlying semantics for the translated programs. There are also other approaches. A language SuRE has been developed in [Jay92, JJ94, JOM95, OJ99, Moo97] with the aim of expressing set-of and aggregation in a natural way. The operational semantics of SuRE do not need to resort to the translation-approach mentioned above, because it is possible to devise a computational model for the large class of the so-called cost-monotonic partial-order programs [Moo97, OJP99, OJN99].

Let us now discuss briefly the set-grouping and aggregation problems and illustrate them with some simple examples. The following example is about set-grouping and is taken from [OJ99]. It is a simple monotonic example of a reflexive-transitive closure. For a full discussion we refer to [OJP99] where the use of partial-order clauses is explained. For a type "type" we denote by "{type}" the powerset 2^{type}, i.e. the set of all subsets consisting of elements of type "type".

Example 5.1 (Transitive and Reflexive Closure) In the following the value of the function r at x, written $r(x)$, is intended to be the set consisting of all nodes that can be reached from x using the *edge*-relation. The notation $r(X) \geq Z$ is intended to formalize "the set Z is a subset of $r(X)$".

TYPE		$1, 2, 3 : \text{node}$
PREDICATE		$edge(\text{node}, \text{node})$
FUNCTION		$r(\text{node}) \rightarrow \{\text{node}\}$
$r(X) \geq S$	$\leftarrow$	$single(X, S)$
$r(X) \geq Z$	$\leftarrow$	$edge(X, Y), r(Y) = Z$

The program defines $x \in r(y)$ *if and only if* $(y, x) \in edge^*$, where $edge^*$ is the transitive and reflexive closure of *edge*. We assume here that *single* is a built-in predicate with the intended meaning of: $single(X, S)$ is true *if, by definition,* S is the singleton set $\{X\}$.

To translate the program we do the following: replace in the program every occurrence of $f(\bar{t})$ *rel* t' by the atom $f_{rel}(\bar{t}, t')$, where f is a functional symbol, $\bar{t}$ is a list of terms, t' is a term, and *rel* is either $=$, or $\geq$. We assume that f_{rel} is a new predicate symbol. The TYPE declarations are also translated in a similar way. The clauses of the translated program are:

TYPE		$1, 2, 3 : \text{node}$
PREDICATE		$edge(\text{node}, \text{node}), r_{\geq}(\text{node}, \{\text{node}\}), r_{=}(\text{node}, \{\text{node}\}),$
		$r_{>}(\text{node}, \{\text{node}\}), single(\text{node}, \{\text{node}\}),$
		$lub(\{\text{node}\}, \{\text{node}\}, \{\text{node}\}), \{\text{node}\} > \{\text{node}\}$
$r_{\geq}(X, Y)$	$\leftarrow$	$single(X, Y)$
$r_{\geq}(X, Z)$	$\leftarrow$	$edge(X, Y), r_{=}(Y, Z)$

The above program makes use of two (unrelated) predicates: $r_{\geq}$, $r_{=}$. We need to include a set of clauses to explicitly relate them as follows:

$$\begin{array}{lcl} r_{\geq}(X, s) & & \\ r_{\geq}(Z, S) & \leftarrow & r_{\geq}(Z, S1), S1 > S \\ r_{\geq}(Z, S) & \leftarrow & r_{\geq}(Z, S1), r_{\geq}(Z, S2), lub(S1, S2, S) \\ r_{=}(Z, \mathrm{C}) & \leftarrow & r_{\geq}(Z, \mathrm{C}), \bigwedge_{\mathrm{C}<S1\leq s123} \neg r_{\geq}(Z, S1) \end{array}$$

The last "rule" needs explanation: we include a clause of this form for every constant symbol C representing an element of the codomain of r.

The type restrictions are used only to ensure that the predicates *lub* and $>$ can be defined by a finite datalog extensional database. For instance, considering *lub* we can declare:

$$lub(s, s_{12}, s_{12}), lub(s_{12}, s_{13}, s_{123}), \ldots$$

Here, s represents the empty set, s_{12} represents the $\{1, 2\}$ set ... We only need eight constant symbols to represent all the required sets. Note also that

$$\bigwedge_{S<S1\leq s_{123}} \neg r_{\geq}(Z, S1)$$

represents a formula of a conjunction of literals.

Now consider the following EDB with respect to *edge*:

$$\mathrm{EDB}_1 = \{edge(1, 2), edge(2, 3)\}.$$

Note that ground(P) is finite. It turns out that WFS defines a total model that agrees with the intended model. In this model $r_{=}(1, s_{123})$ is true. STABLE as well as WFS$^+$ define the intended model.

Here is another example:

$$\mathrm{EDB}_2 = \mathrm{EDB}_1 \cup \{edge(1, 1)\}.$$

We expect to keep the same semantics and this indeed is true for WFS, STABLE and WFS$^+$.

Finally, consider

$$\mathrm{EDB}_3 = \mathrm{EDB}_1 \cup \{edge(2, 1)\}.$$

We expect to keep a total semantics, where $r_{=}(1, s_{123})$ is true. That is, the intended semantics of this program should not change drastically. Lack of space precludes working out this example in full detail. But it can be easily verified, that WFS defines most of the atoms as undefined and that the stable semantics is inconsistent. However WFS$^+$ defines only one model, which is the intended model.

The following example is taken from [VG92], where Van Gelder classifies it as non-monotonic and claims that it makes intuitive sense.

Example 5.2 (First Party Invitation Problem, [VG92])

TYPES		$a, b, c : person$
		$0, 1, 2, 3, 4, 5, 6, 7, 8, 9 : digit$
PREDICATE		$accept(person)$
FUNCTIONS		$positive_sum(\{pair(person, digit)\})$
		$rel_all(person) \rightarrow \{pair(person, digit)\}$
$accept(X)$	$\leftarrow$	$re_all(X) = S, positive_sum(S)$
$rel_all(X) \geq S$	$\leftarrow$	$single(pair(Y, C), S), accept(Y), compatible(X, Y, C)$

The above clauses express the criterion that a person will attend a party if she has sufficient compatibility with the rest of the people coming to the party. We express this criterion by requiring that the sum of all compatibilities, positive and negative, to be nonnegative. We omit the definition *positive_sum* and we assume that its intended meaning is to add a second argument, S, to every element, and to verify that the result is ≥ 0.

Consider the database of facts:

$$compatible(a, b, 1), compatible(a, c, 1), compatible(b, a, 1), compatible(b, c, 1).$$

The intended semantics of the program (as suggested in [VG92]) defines all these terms $accept(a)$, $accept(b)$ and $accept(c)$ to be true. That is exactly what we get with WFS$^+$. We refer to `http://gente.pue.udlap.mx/$^\sim$josorio` for some more examples.

6 Semantics Based on Provability Closures

In this section we introduce the notion of *provability closures* in order to

1. provide a better understanding of some non-monotonic semantics in terms of inference systems,
2. model some aggregation problems with semantics of low complexity (we define WFS^{m1}, a semantics weaker than WFS$^+$ and of lower complexity),
3. get a better understanding of the space of well-behaved semantics: we refute one of the conjectures in [Dix95a].

Our main idea is to map logic programs into classical logical theories (where the "$\leftarrow$" in logic programs gets the usual meaning of material implication $\Rightarrow$) and to consider closures of these theories. Thus we are working with Definition 2.2 and considering certain extensions of $\mathbf{tr}(P)$ instead of P. Some of the derivable atoms in these theories are declared to be *entailed* by the original program P that we started with.

Definition 6.1 (Provability Closure Properties, [DO97b])
We introduce the following provability closure properties (for atoms a):

***Weakclassicality*:** if $\mathbf{tr}(P) \vdash_m a$ then $a \in \mathrm{SEM}(P)$.

***Classicality*$_1$:** if $\mathbf{tr}(P) \vdash_{m1} a$ then $a \in \mathrm{SEM}(P)$.

***Classicality*$_2$:** if $\mathbf{tr}(P) \vdash_{m2} a$ then $a \in \mathrm{SEM}(P)$.

***Supraclassicality*:** if $\mathbf{tr}(P) \vdash_{m12} a$ then $a \in \mathrm{SEM}(P)$.

The idea of the next definition is to keep the original program and adding more and more atoms to it. This defines a sequence of sets of atoms $\mathcal{A}_i$. These atoms are obtained by applying our old transformations to the program from the last step. Here is the exact definition:

Definition 6.2 (WFS^q, [DO97b])
Let q be either m, m1, m2 or m12. The *extended wellfounded* semantics with respect to q, denoted as $\mathrm{WFS}^q(P)$, is defined through the following increasing sequence: $(\mathcal{A}_0 = \emptyset)$

$$\begin{array}{ll} \mathcal{A}_{i+1} & := T_q^P(\mathrm{WFS}(P \cup \mathcal{A}_i)) \\ T_q^P(I) & := \{a : \ a \text{ an atom, } \mathbf{tr}(P \cup I) \vdash_q a\} \end{array}$$

There is the least number j such that $\mathcal{A}_j = \mathcal{A}_{j+1}$.
We define $\mathrm{WFS}^q(P) := \mathrm{WFS}(P \cup \mathcal{A}_j)$.

It should be clear that WFS^{m1} is weaker than WFS^+. The program P_g defined above is a witness of this claim, since $\mathrm{WFS}^+(P_g)=\{x\}$ but $\mathrm{WFS}^{m1}(P_g)=\emptyset$. From this fact and the construction of each semantics it follows:

Theorem 6.3 (Some Relations among Semantics, [DO97b])
$WFS(P) = WFS^m(P)$, $WFS^{m12}(P) = WFS^+(P)$ and $WFS^{m1}(P) <_k WFS^+(P)$

Proof. To prove $\mathrm{WFS} = \mathrm{WFS}^m$ it suffices to show that WFS already satisfies *Weakclassicality*. Suppose $\mathbf{tr}(P) \vdash_m a$, where a is an atom. Then according to Theorem A.7 in the appendix, a can be proved from $\mathbf{tr}(P)$ in $\vdash_m$ without using any axioms, i.e. using only hypotheses from $\mathbf{tr}(P)$ and modus ponens. But then $a \in \mathrm{WFS}(P)$ because WFS is closed under program rules. The proof of $\mathrm{WFS}^{m12}(P) = \mathrm{WFS}^+(P)$ is obtained immediately by using Theorem 4.7 since $\vdash_{m12}$ corresponds to $\models$. Clearly $\mathrm{WFS}^{m1}(P) \leq_k \mathrm{WFS}^+(P)$ follows by construction and to see that the inequality is strict, just consider the following example:

$$\begin{array}{lcl} p & \leftarrow & a \\ p & \leftarrow & \neg a \\ a & \leftarrow & \neg b \\ b & \leftarrow & \neg a \end{array}$$

Clearly $\mathrm{WFS}^{m1} = \emptyset$, while $\mathrm{WFS}^{m12} = \{p\}$. **Q.E.D**

The following theorem can be proved in the same lines as [Dix95a, Theorem 4.9] about WFS^+:

Theorem 6.4 (Classifying WFS^{m1})
WFS^{m1} is the weakest extension of WFS that satisfies Cut and Classicality$_1$. *In addition WFS^{m1} is rational, well-behaved, and is induced by the* confluent LP-system $\mathcal{CS}_{1'}$ + Classicality$_1$.

It is thus obvious that WFS^{m1} refutes Conjecture 7.20 in [BDK97] which states: There are no well-behaved and rational semantics other than WFS, WFS$^+$, and WFS$'$.

Our next result is a general incompatibility theorem. It says that there are no well-behaved semantics satisfying both cumulativity and **GPPE**.

Theorem 6.5 ([DO97b])
For well-behaved semantics satisfying Classicality$_1$, *Cumulativity and **GPPE** are incompatible.*

Proof. Let SEM be a well-behaved semantics. Let us consider

$$\begin{array}{lllll} P: & a & \leftarrow & \neg a & \qquad P': \quad a \leftarrow \neg a \\ & b & \leftarrow & \neg a & \qquad\quad\ \ b \leftarrow a \end{array}$$

Note that $a \in \mathrm{SEM}(P)$ if *Classicality$_1$* holds. If **GPPE** holds, then also $a \in \mathrm{SEM}(P')$ (because P and P' are ***GPPE-equivalent***: any semantics satisfying **GPPE** will consider P and P' equivalent, because applying **GPPE** to P' just gives P). Because of **GPPE**-equivalence of P and P' we also have $b \in \mathrm{SEM}(P')$ iff $b \in \mathrm{SEM}(P)$. Because of well-behavedness, $\mathrm{SEM}(P') = \{a, b\}$ (SEM must be a model of the second rule) and thus $\mathrm{SEM}(P) = \{a, b\}$. Now $\mathrm{SEM}(P \cup \{a\}) = \{a, \neg b\}$, because well-behaved semantics extend WFS and $\mathrm{WFS}(P \cup \{a\}) = \{a, \neg b\}$. Cautious monotonicity therefore fails. **Q.E.D**

It is therefore obvious that WFS^{m1} does not satisfy **GPPE**. Consider again our Example 5.1 and let us call it P. Note that $r_{\geq}(2, a23) \in \mathrm{WFS}(P)$. Let S be the following set of clauses.

$$\begin{array}{lll} r_{=}(1, a123) & \leftarrow & r_{\geq}(1, a123) \\ r_{\geq}(1, a23) & \leftarrow & r_{=}(2, a23) \\ r_{\geq}(2, a123) & \leftarrow & r_{=}(1, a123) \\ r_{=}(2, a23) & \leftarrow & \neg r_{\geq}(2, a123) \\ r_{\geq}(1, a123) & \leftarrow & r_{\geq}(1, a23) \end{array}$$

1. Let $P' := P \cup \{r_{\geq}(2, a23)\}$. It is easy to verify that $\mathbf{tr}(P') \vdash_{m1} \alpha$ for every $\alpha \in S$.

2. Moreover $S \vdash_{m1} r_{\geq}(2, a123)$. So $\mathbf{tr}(P') \vdash_{m1} r_{\geq}(2, a123)$ and finally note that $\mathrm{WFS}(P \cup \{r_{\geq}(2, a123)\})$ defines the intended (total) model. A similar situation occurs in Example 5.2. So, WFS^{m1} defines the intended total model of both programs.

7 New Semantics that extend WFS

In this section we show how our transformation rules can be extended by new rules that still define confluent LP-systems.

As we have just mentioned, the motivation of these rules is the overly sceptical approach of WFS, and also because we noticed that to express aggregation it is useful to consider that $a \leftarrow \neg a$ is equivalent to a.

Since the WFS$^+$ semantics satifies the **LC**-rule, it certainly has this property. But the **LC**-rule lifts the complexity of WFS$^+$ to the second level of the polynomial hierarchy (WFS is quadratic but a check for classical entailment is NP-complete) and thus most probably not polynomial. To solve this problem, we introduced WFS^{m1} that appears simpler to compute than WFS$^+$. In this section we define two semantics that are polynomial and that try to stay close to WFS^{m1}. We conjecture that one of our semantics (WFS$^{LLC'}$) is equivalent to WFS^{m1}.

We first introduce a weaker form of **LC**:

Definition 7.1 (Local Logical Consequence LLC)
By the application of **LLC** (*local logical consequence*) to a program P that only contains one clause with head a, namely an LC-clause $a \leftarrow \neg a \ \in P$, we mean the transformation of P which simply removes every occurrence of $\neg a$ in P.

We define $\mathcal{CS}_3$ as the LP-system $\mathcal{CS}_1$ plus the rule **LLC**. It seems that for some aggregation functions, this transformation already gives the desired results. Here is one of our main theorems.

Theorem 7.2 ($\mathcal{CS}_3$ is a confluent LP-System)
$\mathcal{CS}_3$ is a confluent LP-System. Its induced semantics, called WFS^0, is 3-value based, but not rational.

Proof. This calculus is noetherian, because the LP-system $\mathcal{CS}_1$ has this property and **LLC** (the only additional transformation) only reduces the program and does not introduce any new positive body-atoms.

To prove confluence, it is enough to prove local confluence. Let us use $\rightarrow_{wfs^1}$ to denote the reduction relation in this system. Let $x \rightarrow_{wfs^1} y$ and $x \rightarrow_{wfs^1} z$. We have to prove that there exists w, such that $y \rightarrow^*_{wfs^1} w$ and $z \rightarrow^*_{wfs^1} w$. Let $x \rightarrow_{wfs} y$ and $x \rightarrow_{wfs} z$, then by local confluence of $\langle \mathrm{Prog}_{\mathcal{L}}, \rightarrow_{wfs} \rangle$ the result holds. So without loss of generality, we have to check the case:

$$\text{Let } x \rightarrow_{wfs^1} y \text{ and } z = x^{op1} \text{ where } op1 \text{ is of type } \mathbf{LLC}.$$

1. If $x \rightarrow_{wfs^1} y$ by a **TAUT**, or a **SUB** operation $op2$ then it is easy to check that $op1$ and $op2$ commute.

2. If $op2$ is a **RED**$^+$ operation, then applying **LLC** to y leads to the same program as applying **RED**$^+$ and then **LLC** again to z.

3. If $op2$ is a $\mathbf{RED}^-$ operation then applying $\mathbf{RED}^-$ to z leads to the same program as applying **LLC** and then $\mathbf{RED}^-$ again to y.

4. If $op2$ is a **GPPE** operation then again $op1$ and $op2$ commute.

To see that it is 3-value based it is enough to remember that it is known that WFS is 3-value based and then is not hard to check that **LLC** preserves this property. To see that it is not rational consider the program P':

$$\begin{array}{lll} a & \leftarrow & \neg a \\ b & \leftarrow & \neg a \end{array}$$

Now $\mathrm{WFS}^0(P') = \{a, b\}$ but $\mathrm{WFS}^0(P' \cup \{a\}) = \{a, \neg b\}$. **Q.E.D**

Note that the same counterexample to rationality (in the above proof) is a counterexample for partial distribution for WFS^0.

7.1 Stronger Versions of LLC

Definition 7.3 (LLC′)
Let a be an atom that occurs negatively in the in a program P and also appears in the head of some rule. Let P_1 be the program that results from P by removing $\neg a$ from every clause of P. Let $\mathbf{Success}^*$ denote the reflexive and transitive closure of the relation **Success**. Suppose that P_1 relates to P_2 by $\mathbf{Success}^*$ and $a \in P_2$. In this case, we add $a \leftarrow$ to P.

We defined $\mathbf{LLC}'$ having in mind the following consideration:

$$\mathbf{tr}(P) \cup \{\neg a\} \models a \textit{ if and only if } \mathbf{tr}(P) \models \mathbf{tr}(a \leftarrow \neg a).$$

But $\mathbf{tr}(a \leftarrow \neg a) = (\neg a \Rightarrow a) \Rightarrow a$ is a theorem in propositional classical logic and by *Modus Ponens* we can infer a. Therefore, this rule is sound in classical propositional logic. We mentioned that SuRE programs can be translated to normal programs such that the declarative semantics of a SuRE program corresponds to the WFS^+ of the translated normal program. But we have recently noted, [DO97c, DO97a], that the full power of WFS^+ is not always needed: a restricted form that corresponds to $\mathbf{LLC}'$ suffices.

Let $\mathcal{CS}_4$ be the LP-system $\mathcal{CS}_{1'}$ plus the rule $\mathbf{LLC}'$.

Theorem 7.4 ($\mathcal{CS}_{1'}$ + $\mathbf{LLC}'$ is confluent)
$\mathcal{CS}_4$ is a confluent LP-System. Its induced semantics, that we call $WFS^{LLC'}$, is 3-value based, well-behaved and rational.

Proof. It is obvious that the system is noetherian. So, to prove confluence it suffices to prove local confluence. The only case that deserves certain consideration is to check **Loop** versus $\mathbf{LLC}'$. Note however that these operators commute. Rationality follows since it is easy to check that the LP-System satisfies partial distribution.

It remains to show well-behavedness. The *cut* property is obvious; *Weak partial evaluation* follows from **Success** and **Loop**. We therefore have to show *relevance*. Although we have not given the precise formulation of *relevance* (answering a query ψ with respect to program P should only depend on the call graph P/ψ) we can easily sketch a proof. As we have a confluent system and all our transformations in $op \in \mathcal{CS}_4$ respect the call graph construction, i.e.

$$P/\psi^{op} = P^{op}/\psi.$$

The result follows. **Q.E.D**

Theorem 7.5 ($\mathbf{WFS}^2 = \mathbf{WFS}^{LLC'}$)
For every program, $WFS^2(P) = WFS^{LLC'}(P)$.

Proof. We write $P \vdash_{\text{LLC'}} P'$ to denote that P adds a by using the LLC' transformation. We write LLC'$^+$ to denote the transitive closure of LLC'. We use SEM$^*(P)$ to denote an auxiliarly semantics defined through the following increasing sequence: $(\mathcal{A}_0^* = \emptyset)$

$$\begin{array}{ll} \mathcal{A}_{i+1}^* & := T^*(P \cup \mathcal{A}_i^*) \\ T^*(P) & := \{a : \ a \leftarrow \quad \text{belongs to } norm_{cs_{1'}}(P)\} \cup \{a : \ norm_{cs_{1'}}(P) \vdash_{LLC'} a\} \end{array}$$

There is the least number n_0 such that $\mathcal{A}_{n_0}^* = \mathcal{A}_{n_0+1}^*$. We define

$$\text{SEM}^*(P) := \text{WFS}(P \cup \mathcal{A}_{n_0}^*).$$

We are going to use SEM* to prove that SEM$^*(P) = $ WFS$^2(P)$ and SEM$^*(P) = $ WFS$^{LLC'}(P)$ from which we conclude WFS$^2(P) = $ WFS$^{LLC'}(P)$.

To prove SEM$^*(P) = $ WFS$^2(P)$ we have to prove that WFS$(P \cup \mathcal{A}_{n_0}^*) = $ WFS$(P \cup \mathcal{A}_{n_0})$. So, we only need to check that $\mathcal{A}_{n_0}^* = \mathcal{A}_{n_0}$ which is straightforward by induction on the index n_0 using:

$$a \leftarrow \neg a \text{ belongs to } norm_{cs_1}(P) \textit{if and only if } norm_{cs_{1'}}(P) \vdash_{\text{LLC'}} a.$$

To prove SEM$^*(P) = $ WFS$^{\text{LLC'}}(P)$, note that we can reduce P to $norm_{cs_{1'}}(P)$ and after that we can apply the transformation LLC'$^+$ to get a program P_1. We can continue doing this, now on program P_1, until we get to $norm_{cs_4}(P)$, i.e.,

$$\begin{array}{l} P \mapsto_{cs_{1'}}^* norm_{cs_{1'}}(P) \mapsto_{\text{LLC'}^+} P_1 \mapsto_{cs_{1'}}^* \ \cdots \ norm_{cs_{1'}}(P_{m-1}) \\ \mapsto_{\text{LLC'}^+} P_m \mapsto_{cs_{1'}}^* \ \cdots \ norm_{cs_4}(P), \end{array}$$

and from that we get WFS$^{\text{LLC'}}(P)$. Finally, we check by induction on i that $P \cup \mathcal{A}_i^* \mapsto_{cs_{1'}} P_i$. Thus, SEM$^*(P) = $ WFS$(P \cup \mathcal{A}_{n_0}^*) = $ WFS$^{\text{LLC'}}(P)$. **Q.E.D**

Definition 7.6 (Contra)
Let C be a clause of the form $a \leftarrow$ body, where both $b \in$ body and $\neg b \in$ body. We define the transformation Contra as the one that deletes the clause C from the program P.

The motivation for *Contra* comes from the following observations. Clearly, the rule is a theorem in intuitionistic propositional logic and therefore in classical logic. The power of a theory should not be affected if we add or delete theorems from the proper axioms of the theory (the proper axioms correspond to our programs). In addition, the stable semantics satisfies *Contra*. In fact, as proved in [BD97], *Contra* is the only difference between WFS and STABLE (in the sense that adding *Contra* to the system defining WFS implies that the intended models are exactly the stable models).

Theorem 7.7 (WFS + Contra)
The LP-system given by $\mathcal{CS}_{1'}$ plus the rule Contra is confluent. In addition its induced semantics is rational but not 3-value based.

Proof. It is obvious that the system is noetherian. To prove that the system is confluent we note that $\mathcal{CS}_{1'}$ is known to be confluent. Thus, we only need to check Contra is locally confluent with respect to the other operators.

RED$^+$: Either $P^{[\mathbf{RED}^+,\mathbf{Failure}]} = P^{\mathbf{Contra}}$ or $P^{[\mathbf{RED}^+,\mathbf{Contra}]} = P^{\mathbf{Contra}}$.

RED$^-$ Either Contra commutes with **RED$^-$** or $P^{\mathbf{RED}^-} = P^{\mathbf{Contra}}$.

Success: Either Success commutes with **Contra** or $P^{[\mathbf{Success},\mathbf{RED}^-]} = P^{\mathbf{Contra}}$.

Loop: Either **Loop** commutes with **Contra** or $P^{[\mathbf{Contra},\mathbf{Loop}_A]} = P^{\mathbf{Loop}_A}$.

With respect to rationality it is again clear that partial distribution is satisfied. To show that the LP-system is not 3-valued, consider the following counterexample. Let P be the program consisting of $b \leftarrow \neg b$ and $a \leftarrow b, \neg b$. Then SEM(P) = $\{\neg a\}$. The induced 3-valued interpretation assigns false to a and undefined to b. But this is clearly not a model of P (due to the second clause).
Q.E.D

WFS has been criticized for its inability to do reasoning by cases. In several cases, this is a reason to prefer STABLE over WFS, see for instance [BG94, Dix95b]. The following rule allows us to add a weak form of this kind of reasoning to WFS, while still preserving the polynomial-time computability property of the semantics.

Definition 7.8 (Clauses Weak-Cases related)
Given two clauses C_1 and C_2, we say that C_1 and C_2 are weak-cases related *if, by definition,* the following condition holds: C_1 is of the form $a \leftarrow \text{body}_1$, C_2 is of the form $a \leftarrow \text{body}_2$, body_1 is the disjoint union of body and $\{l_1\}$, body_2 is the disjoint union of body and $\{l_2\}$, l_1 and l_2 are complementary literals, and a does not occur in l_1.

Definition 7.9 (T-Weak-Cases)
Let P be a program and suppose the following condition holds: $C_1 \in P$, $C_2 \in P$, C_1 is weak-cases related to C_2, C_1 is of the form $a \leftarrow \text{body}_1$, C_2 is of the form $a \leftarrow \text{body}_2$. Let $\text{body} := \text{body}_1 \cap \text{body}_2$. Assume in addition that if C_3 weak-cases relate to C_1 then $C_3 = C_2$, and if C_4 weak-cases relate to C_2 then $C_4 = C_1$ for any pair of clauses C_4 and C_3.

Then the T-Weak-Cases transformation replaces the clauses C_1 and C_2 in P by the single clause $a \leftarrow \text{body}$.

Example 7.10 (T-Weak-Cases) Suppose P is:

$$\begin{array}{lll} a & \leftarrow & b, c \\ a & \leftarrow & \neg b, c \end{array}$$

Then T-Weak-Cases transforms P to:

$$a \leftarrow c$$

Now consider the program P

$$\begin{array}{lll} a & \leftarrow & b, c \\ a & \leftarrow & \neg b, c \\ a & \leftarrow & \neg b, \neg c \end{array}$$

Then T-Weak-Cases can not be applied.

Theorem 7.11 (WFS + T-Weak-Cases)
The LP-system $\mathcal{CS}_5$ given by $\mathcal{CS}_{1'}$ plus the rule Weak-Cases is confluent. In addition its induced semantics (called WFS^{WK}) is rational and 3-value based.

Proof. It is clear that the LP-systems is noetherian. It suffices to check T-Weak-Cases versus the other operators.

- Either **T-Weak-Cases** commutes with $\mathbf{RED}^+$ or $P^{\mathbf{T\text{-}Weak\text{-}Cases}} = P^{[\mathbf{RED}^+, \mathbf{Failure}]}$.
- Either **T-Weak-Cases** commutes with $\mathbf{RED}^-$ or $P^{[\mathbf{RED}^-, \mathbf{Success}]} = P^{\mathbf{T\text{-}Weak\text{-}Cases}}$.
- Either **T-Weak-Cases** commutes with **Success** or $P^{[\mathbf{Success}, \mathbf{RED}^-]} = P^{\mathbf{T\text{-}Weak\text{-}Cases}}$ or $P^{[\mathbf{Success}, \mathbf{Success}, \mathbf{T\text{-}Weak\text{-}Cases}]} = P^{[\mathbf{T\text{-}Weak\text{-}Cases}, \mathbf{Success}]}$
- Either $\mathbf{Loop}_A$ commutes with **T-Weak-Cases** or $P^{[\mathbf{Loop}_A, \mathbf{RED}^+]} = P^{[\mathbf{T\text{-}Weak\text{-}Cases}, \mathbf{Loop}_A]}$.
- **T-Weak-Cases** with itself commutes.

- **T-Weak-Cases** and **Failure**: They commute or
  $P^{[\mathbf{Failure},\mathbf{Failure}]} = P^{[\mathbf{T\text{-}Weak\text{-}cases},\mathbf{Failure}]}$ or
  $P^{[\mathbf{Failure},\mathbf{RED}^-]} = P^{[\mathbf{T\text{-}Weak\text{-}cases}]}$.

Again partial distribution is satisfied and so the system is rational. Since WFS is 3-value based and Weak-Cases preserves this property, the resulting system is also 3-value based. **Q.E.D**

An interesting aspect about the LP-systems is that we can start treating them as objects to perform operations over them. For instance, it makes sense to define the union of two systems $\mathrm{LP}_1 := \langle ops_1, \rightarrow_1 \rangle$ and $\mathrm{LP}_2 := \langle ops_2, \rightarrow_2 \rangle$ as $\langle ops_1 \cup ops_2, \rightarrow_1 \cup \rightarrow_2 \rangle$. We denote the operation by $\mathrm{LP}_1 \cup \mathrm{LP}_2$. A simple sufficient condition to ensure that $\cup$ defines a confluent LP-system is the following:

Theorem 7.12 (Constructing new confluent LP-systems)
Let $LP_1 := \langle ops \cup ops_1, \rightarrow \rangle$ and $LP_2 := \langle ops \cup ops_2, \rightarrow \rangle$ be two confluent LP-systems and $LP_3 := LP_1 \cup LP_2$. Suppose in addition that LP_3 is noetherian and that for every $op_1 \in ops_1$ and every $op_2 \in ops_2$, op_1 and op_2 commutes. Then LP_3 is a confluent LP-system.

We say that op_1 and op_2 commute, if, by definition, for every program P such that P reduces to P_1 by op_1 and P reduces to P_2 by op_2 then P_2 reduces to P' by op_1 and P_1 reduces to P' by op_2.

If in addition LP_1 and LP_2 satisfy partial distribution then LP_3 is also rational.

Proof. The conditions ensures that LP_3 is locally confluent. The termination property is given and thus LP_3 is a confluent LP-system. If LP_1 and LP_2 satisfy partial distribution, so does LP_3 since this property is local to each operator. But partial distribution implies rationality. **Q.E.D**

From the last theorem it follows immediately that:

Theorem 7.13 ($\mathcal{CS}_6 := \mathcal{CS}_4 \cup \mathcal{CS}_5$)
Let $\mathcal{CS}_6 := \mathcal{CS}_4 \cup \mathcal{CS}_5$. The $\mathcal{CS}_6$ is a confluent LP-system. It induces a 3-valued and rational semantics that we call $WFS^{WK+LLC'}$.

Example 7.14 ($\mathbf{WFS}^{WK+LLC'}$) Let P be:

$$\begin{array}{lcl} a & \leftarrow & \neg b, \neg c \\ a & \leftarrow & b, \neg c \\ c & \leftarrow & a \\ b & \leftarrow & \neg d \\ d & \leftarrow & \neg b \end{array}$$

Then $\mathrm{WFS}^{WK+LLC'}(P) = \{c, \neg a\}$. Notice that STABLE is inconsistent.

Consider again our Example 5.1. After grounding the program and applying some transformations we get to a program which includes the following rules:

$$\begin{array}{lcl} r_{=}(1,a123) & \leftarrow & r_{\geq}(1,a123) \\ r_{\geq}(1,a23) & \leftarrow & r_{=}(2,a23) \\ r_{\geq}(2,a123) & \leftarrow & r_{=}(1,a123) \\ r_{=}(2,a23) & \leftarrow & \neg r_{\geq}(2,a123) \\ r_{\geq}(1,a123) & \leftarrow & r_{\geq}(1,a23) \end{array}$$

Thanks to the $\mathbf{LLC}'$ transformation we can conclude that `r≥(2,a123)` is true (which is undefined under WFS). With this new fact, the transformations now allow us to obtain the intended (total) model as desired. A similar situation occurs to Example 5.2. So, $\text{WFS}^{LLC'}$ defines the intended total model of both programs.

We finish this section with a conjecture:

Conjecture 7.15 ($\mathbf{WFS}^{LLC'} = \mathbf{WFS}^{m1}$)
We claim that $WFS^{LLC'} = WFS^{m1}$.

8 Approximating the Stable Semantics

In this section, we study the integration of comp and WFS and propose a new semantics CWFS. CWFS has the following nice features: it is stronger than WFS but "less inconsistent" than STABLE and it can be used to model aggregation.

Before giving the precise construction and in order to make this paper self-contained, we need to review the STABLE semantics and Clark's completion semantics (comp).

A model M of a program P is called *supported*, *if, by definition,* for every atom $a \in M$ there exists a clause whose consequent is a and whose antecedent is true in M. A model M of P is called *well-supported*, *if, by definition,* it is supported and there exists a wellfounded partial ordering $\leq$ on M such that for any atom $a \in M$ there exists a clause in P with consequent a, and for every positive literal b in the antecedent of the clause, we have $b < a$. For any normal program P, the well-supported models of P are defined to be the *stable models* of P. The *scenario stable* semantics is taken as the set of the stable models. The *sceptical stable* semantics is taken as the set of literals true in all stable models. We take here the last approach as our definition for the stable semantics. We also consider here that if a program has no stable models its semantics infers every literal.

Definition 8.1 (Definition of def and sup)
Let a be an atom in $\mathcal{L}_P$. By the definition of a in P, denoted by def(a), we mean the set of clauses: $\{a \leftarrow \text{body} \in P\}$. We define

$$\text{sup}(a) := \begin{cases} \mathbf{f} & \text{if def}(a) = \emptyset \\ \text{body}_1 \vee \cdots \vee \text{body}_n & \text{if def}(a) = \{a \leftarrow \text{body}_1, \ldots, a \leftarrow \text{body}_n\}^5 \end{cases}$$

Definition 8.2 (comp(P) ([Cla78]))
For any program P, we define comp(P) as the classical theory $\{a \Leftrightarrow \sup(a) : a \in \mathcal{L}_P\}$.

A nice connection between STABLE, comp and WFS is the following:

Theorem 8.3 (STABLE in terms of $\mathcal{CS}_1$ and comp, [BD95])
For a program P, $STABLE(P) = MOD(comp(norm_{\mathcal{CS}_1}(P)))$.

As shown in [BD95, BDP99], this relation opens the way to compute STABLE by doing hyperresolution on $norm_{\mathcal{CS}_1}$(P).

Theorem 8.4 (Weak Cautious Monotonicity of STABLE)
Let P be a program without positive body literals which has STABLE models. Suppose a does not occur in the body of any clause of P then $a \in STABLE(P)$ implies: "$b \in STABLE(P)$ if and only if $b \in STABLE(P \cup \{a\})$."

Proof. For programs without positive literals,

$$\text{MOD(STABLE}(P)) = \text{MOD(comp}(P)),$$

where MOD(P) is the set of two-values classic models of P. It is easy to verify that with the given syntactic restrictions, MOD(comp(P)) = MOD(comp$(P \cup \{a\})$). **Q.E.D**

We now present CWFS which is a semantics that combines WFS and comp in a suitable way. Roughly speaking, CWFS uses the WFS power to derive negative literals and the comp power to derive positive literals. It is known that CWFS lies in between WFS and STABLE (Theorem 8.10). The construction of CWFS is given thanks to confluent systems and it helps to understand better the relationship among comp, WFS and STABLE.

In [OJ98] it is wrongly claimed that the system $\mathcal{CS}_{1'}$ + **TAUT** is confluent. Here is a counterexample:

$$\begin{array}{lcl} a & & \\ a & \leftarrow & a, b \\ b & \leftarrow & \neg c \\ c & \leftarrow & \neg b \end{array}$$

If we apply **Success** the program reduces to :

$$\begin{array}{lcl} a & & \\ a & \leftarrow & b \\ b & \leftarrow & \neg c \\ c & \leftarrow & \neg b \end{array}$$

However, if we apply **TAUT** to the original program it reduces to

$$\begin{array}{lll} a & & \\ b & \leftarrow & \neg c \\ c & \leftarrow & \neg b \end{array}$$

Both programs are different but irreducible! This example shows that in general, it is not possible to just add **TAUT** and retain the confluence property.

8.1 CWFS

Definition 8.5 (CWFS(P))
Let P be a normal program. We define comp-WFS(P) := comp($norm_{\mathcal{CS}_{1'}}(P)$), where comp is defined over $\mathcal{L}_P$. A model of comp-WFS(P) is called an intended model of P with respect to CWFS. The *basic CWFS semantics* of P is defined as the unique intended model of P with respect to CWFS.

When there are either many or an absence of intended models, we say that the program is inconsistent. The *scenario CWFS semantics* of P is defined as the set of intended models of P. Finally, the *sceptical CWFS semantics* of comp-WFS(P), denoted as CWFS(P), is defined as

$$\{l : \text{comp-WFS}(P) \vdash l, \text{where } l \in \mathcal{L}_P\}$$

Note that the sceptical CWFS is inconsistent only when there is no intended model at all. In this case it proves every literal. (But we could redefine this notion, and state that in this case "no literal are derived.") Unless stated otherwise, we assume the sceptical view of the CWFS semantics.

Theorem 8.6 (Closure Properties of CWFS)
CWFS is closed under each of the following transformations: ***Loop, RED$^-$, RED$^+$, Success, Failure****.*

Lemma 8.7
For every normal program P, such that $P \rightarrow^* P_1$, where $\rightarrow$ is the transformation relation in $\mathcal{CS}_{1'}$, M being a stable model of P implies that M is a stable model of P_1.

Proof. See [BZF97, BDFZ98]. **Q.E.D**

The following is an important property of CWFS.

Theorem 8.8 (STABLE Models are Intended Models for CWFS)
For every normal program P, if M is a stable model of P then M is an intended model of P with respect to CWFS.

Proof. Suppose that M is a stable model of P, then by Lemma 8.7, M is a stable model of $norm_{\mathcal{CS}_{1'}}(P)$. Therefore M is a well-supported model of $norm_{\mathcal{CS}_{1'}}(P)$ and clearly M is a supported model of $norm_{\mathcal{CS}_{1'}}(P)$. M is also a model of $\text{comp}(norm_{\mathcal{CS}_{1'}}(P))$, which by definition is an intended model of P with respect to CWFS, as desired. **Q.E.D**

The following theorems assume the sceptical approach of the given semantics.

Theorem 8.9 (CWFS is consistent in more Cases than STABLE)
For every normal program P, STABLE(P) is consistent implies that CWFS(P) is consistent, but the converse is false.

Proof. That STABLE(P) is consistent implies that CWFS(P) is consistent (a direct consequence of Theorem 8.8). To see that the converse is false just take the following example:

$$\begin{array}{lcl} c & \leftarrow & c \\ b & \leftarrow & a \\ a & \leftarrow & b \\ a & \leftarrow & \neg b, \neg c \end{array}$$

Note that $\{a, b, \neg c\}$ is the unique CWFS model of P while STABLE is inconsistent. **Q.E.D**

Theorem 8.10 (CWFS is in between WFS and STABLE)
For all programs P: WFS(P) $<$ CWFS(P) $<$ STABLE(P).

Proof. We first prove that given P, $\text{WFS}(P) \leq_k \text{CWFS}(P)$. Let a be an atom such that $a \in \text{WFS}(P)$, then $a \in norm_{\mathcal{CS}_{1'}}(P)$ and so $\sup(a) = \mathbf{t} \vee \alpha$ for some formula α (or $\sup(a)$=$\mathbf{t}$). Therefore $a \in \text{CWFS}(P)$. Now consider the literal $\neg a$ such that $\neg a \in \text{WFS}(P)$, so $a \notin \text{HEAD}(norm_{\mathcal{CS}_{1'}}(P))$, and so $\sup(a) = \mathbf{f}$ and so $\neg a \in \text{CWFS}(P)$. In addition, the following example shows that this inequality is strict.

$$\begin{array}{lcl} a & \leftarrow & \neg b \\ b & \leftarrow & \neg a \\ p & \leftarrow & a \\ p & \leftarrow & b \end{array}$$

WFS defines $\emptyset$ but CWFS defines $\{p\}$. $\text{CWFS}(P) \leq_k \text{STABLE}(P)$ is a direct consequence of Theorem 8.8 (STABLE models are intended models w.r.t. CWFS). That the inequality is strict is a direct consequence of last Theorem 8.9 plus the convention that inconsistency implies that every literal is provable. We remind the reader that the example given in the proof of Theorem 8.9 is a witness of this claim. **Q.E.D**

Consider again the aggregation issue. Both Examples 5.1 and 5.2 work fine under CWFS. But there is a more natural translation that also defines the intended model of our examples. We explain our translation with Example 5.1. The first step of the translation is the same, and we get:

TYPE		$1, 2, 3 : \text{node}$
PREDICATES		$r_{\geq}(\text{node}, \{\text{node}\}), r_{=}(\text{node}, \{\text{node}\}), r_{>}(\text{node}, \{\text{node}\})$
		$\mathit{single}(\text{node}, \{\text{node}\}), \mathit{lub}(\{\text{node}\}, \{\text{node}\}), \{\text{node}\} > \{\text{node}\}$
		$\mathit{edge}(\text{node}, \text{node}),$
$r_{\geq}(X, Y)$	$\leftarrow$	$\mathit{single}(X, Y)$
$r_{\geq}(X, Z)$	$\leftarrow$	$\mathit{edge}(X, Y), r_{=}(Y, Z)$

The above program makes use of two (unrelated) predicates: $r \geq$, $r_{=}$. As before, we need to include a set of clauses to explicitly relate them. But now, we add the following set of clauses:

$$
\begin{array}{lcl}
r_{\geq}(X, \text{s}) & & \\
r_{\geq}(Z, S) & \leftarrow & r_{\geq}(Z, S1), S1 > S \\
r_{\geq}(Z, S) & \leftarrow & r_{\geq}(Z, S1), r_{\geq}(Z, S2), \mathit{lub}(S1, S2, S) \\
r_{=}(Z, S) & \leftarrow & r_{\geq}(Z, S), \neg r_{>}(Z, S) \\
r_{>}(Z, S) & \leftarrow & r_{\geq}(Z, S1), S1 > S
\end{array}
$$

Recall that s is a constant symbol that is interpreted as the empty set. Note that we extended the language by adding a new predicate symbol $r_{>}$. We also use the same EDB for $>$ and lub.

Consider again the extensional database EDB_3. Note that the ground instantiation of P is finite. The relevant clauses of the new translated program of Example 5.1 are:

$$
\begin{array}{lcl}
r_{\geq}(X, Y) & \leftarrow & \mathit{scons}(X, \text{s}, Y) \\
r_{\geq}(X, Z) & \leftarrow & \mathit{edge}(X, Y), r_{=}(Y, Z) \\
r_{\geq}(X, \text{s}) & & \\
r_{\geq}(Z, S) & \leftarrow & r_{\geq}(Z, S1), S < S1 \\
r_{\geq}(Z, S) & \leftarrow & r_{\geq}(Z, S1), r_{\geq}(Z, S2), \mathit{lub}(S1, S2, S) \\
r_{=}(Z, S) & \leftarrow & r_{\geq}(Z, S), \neg r_{>}(Z, S1) \\
r_{>}(Z, S) & \leftarrow & r_{\geq}(Z, S1), S < S1
\end{array}
$$

Consider again the extensional database EDB_3. Note that the ground instantiation of P is finite.

1. WFS defines most of the atoms as undefined and the stable semantics is inconsistent.

2. WFS^{+} defines a total model where $r_{=}(1, A)$ is false for every constant symbol A. That is, r does not behave as a total function.

3. However CWFS defines only one model, which is the intended model. CWFS also defines the intended model for Example 5.2 under this new translation.

9 Conclusion

We introduced the general notion of a *confluent LP-system*. Such a system is based on certain transformation rules and induces a semantics SEM in a natural way. Thanks to *confluence* and *termination*, our transformation rules have both a declarative and an operational meaning. From the declarative point of view, they tell us that our semantics is closed under the given transformation rule. From an operational point of view the transformations are computable functions that can be applied to *simplify* the program and derive a *normalform*. When we have derived the normalform, then computing the semantics is immediate. In [BD99, BDFZ98] it is shown how these transformations induce effective bottom-up computation procedures.

We developed in this paper a general theory of confluent LP-systems (see Theorems 3.4, 3.10, 6.5, 7.12). We defined a new interesting property, *partial distribution*, and related it to *rationality*, a well-known property in nonmonotonic reasoning.

Moreover, we showed that most of the well-known semantics for normal logic programs are induced by confluent LP-systems. We also showed by introducing several new transformation rules that the corresponding LP-systems induce interesting semantics which are polynomial time computable, extend WFS and seem suitable for modelling *aggregation* in logic programs.

We were also able to use our theory to get a better picture of the space of well-behaved semantics and to falsify some claims made in related research.

It is quite surprising that the simple notion of a *confluent LP-system* that we introduced here, is so flexible that it allows us to extend recently defined calculi for WFS by new transformations in such a way, that the new system still is confluent.

While the original motivation came from *aggregation* in logic programming, we have shown that rewriting methods can be successfully and elegantly applied in the realm of LP. Many interesting new semantics are just confluent LP-systems.

Acknowledgements

Many thanks to two anonymous referees, who provided us with a lot of helpful remarks and suggestions. Special thanks to John Benton who helped to put this paper into a more readable form and in (almost) correct English. Last, but by no means least, we thank Jose Arrazola, who was involved in many other joint papers: His comments, suggestions and his critical proofreading helped us to put this paper in its final form.

References

[AMTW99] K. R. Apt, V. Marek, M. Truszczynski, and D. S. Warren, editors. *The Logic Programming Paradigm: Current Trends and Future Directions*, LNAI, Berlin, 1999. Springer.

[BD95] 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.

[BD97] Stefan Brass and Jürgen Dix. Characterizations of the Disjunctive Stable Semantics by Partial Evaluation. *Journal of Logic Programming*, 32(3):207–228, 1997. (Extended abstract appeared in: Characterizations of the Stable Semantics by Partial Evaluation *LPNMR, Proceedings of the Third International Conference, Kentucky*, pages 85–98, 1995. LNCS 928, Springer.).

[BD98a] Stefan Brass and Jürgen Dix. Characterizations of the Disjunctive Well-founded Semantics: Confluent Calculi and Iterated GCWA. *Journal of Automated Reasoning*, 20(1):143–165, 1998. (Extended abstract appeared in: Characterizing D-WFS: Confluence and Iterated GCWA. *Logics in Artificial Intelligence, JELIA '96*, pages 268–283, 1996. Springer, LNCS 1126.).

[BD98b] Gerhard Brewka and Jürgen Dix. Knowledge representation with logic programs. In J. Dix, L. Pereira, and T. Przymusinski, editors, *Logic Programming and Knowledge Representation*, LNAI 1471, pages 1–55, Berlin, 1998. Springer. Full version will appear as Chapter 6 in *Handbook of Philosophical Logic*, 2nd edition (2000), Volume 6, Methodologies.

[BD99] Stefan Brass and Jürgen Dix. Semantics of (Disjunctive) Logic Programs Based on Partial Evaluation. *Journal of Logic Programming*, 38(3):167–213, 1999. (Extended abstract appeared in: Disjunctive Semantics Based upon Partial and Bottom-Up Evaluation, *Proceedings of the 12-th International Logic Programming Conference, Tokyo*, pages 199–213, 1995. MIT Press.).

[BD00] Gerhard Brewka and Jürgen Dix. Knowledge representation with extended logic programs. In D. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic, 2nd Edition, Volume 6, Methodologies*, chapter 6. Reidel Publ., 2000. Shortened version also appeared in Dix, Pereira, Przymusinski (Eds.), *Logic Programming and Knowledge Representation*, Springer LNAI 1471, pages 1–55, 1998.

[BDFZ98] Stefan Brass, Jürgen Dix, Burkhard Freitag, and Ulrich Zukowski. Transformation-based bottom-up computation of the well-founded

model. Technical Report TR 15/98, University of Koblenz, Department of Computer Science, Rheinau 1, August 1998. Currently under review for *Journal of Logic Programming*.

[BDK97] Gerd Brewka, Jürgen Dix, and Kurt Konolige. *Nonmonotonic Reasoning: An Overview*. CSLI Lecture Notes 73. CSLI Publications, Stanford, CA, 1997.

[BDNP99] Stefan Brass, Jürgen Dix, Ilkka Niemelä, and Teodor Przymusinski. On the Equivalence of the Static and Disjunctive Well-Founded Semantics and its Computation. *Theoretical Computer Science*, 251, 1999.

[BDP99] Stefan Brass, Jürgen Dix, and Teodor Przymusinski. Computation of the semantics of autoepistemic belief theories. *Artificial Intelligence*, 112(1-2), 1999.

[BG94] Chitta Baral and Michael Gelfond. Logic Programming and Knowlege Representation. *Journal of Logic Programming*, 19-20:73–148, 1994.

[BZF97] Stefan Brass, Ulrich Zukowski, and Burkhardt Freitag. Transformation Based Bottom-Up Computation of the Well-Founded Model. In J. Dix, L. Pereira, and T. Przymusinski, editors, *Nonmonotonic Extensions of Logic Programming*, LNAI 1216, pages 171–201. Springer, Berlin, 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.

[DFN00] Jürgen Dix, Ulrich Furbach, and Ilkka Niemelä. Nonmonotonic Reasoning: Towards Efficient Calculi and Implementations. In Andrei Voronkov and Alan Robinson, editors, *Handbook of Automated Reasoning*. Elsevier-Science-Press, 2000.

[Dix92] 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.

[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.

[Dix98] Jürgen Dix. The Logic Programming Paradigm. *AI Communications*, Vol. 11, No. 3:39–43, 1998. Short version in Newsletter of ALP, Vol. 11(3), 1998, pages 10–14.

[DM94a] 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.

[DM94b] 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.

[DO97a] Jürgen Dix and Mauricio Osorio. On Well-Behaved Semantics Suitable for Aggregation. In Jan Maluszynski, editor, *Logic Programming: Proceedings of the 1997 International Symposium*, page 505. MIT Press, 1997.

[DO97b] Jürgen Dix and Mauricio Osorio. Provability Closures in Logic Programming. In Instituto Politectico Nacional, editor, *Proceedings of the International Symposium on Computer Science in Mexico*, pages 313–319, 1997.

[DO97c] Jürgen Dix and Mauricio Osorio. Towards Well-Behaved Semantics Suitable for Aggregation. Technical Report TR 11/97, University of Koblenz, Department of Computer Science, Rheinau 1, April 1997. Accepted at ILPS '97 as poster paper.

[DO99] Jürgen Dix and Mauricio Osorio. Confluent rewriting systems in non-monotonic reasoning. *Computacion y Sistemas*, Volume II, No. 2-3:104–123, 1999.

[DS98] Jürgen Dix and Frieder Stolzenburg. A Framework to incorporate Nonmonotonic Reasoning into Constraint Logic Programming. *Journal of Logic Programming*, 37(1,2,3):47—76, 1998. Special Issue on *Constraint Logic Programming*, Guest Editors: Kim Marriott and Peter Stuckey.

[Fit85] Melvin C. Fitting. A Kripke-Kleene Semantics of logic Programs. *Journal of Logic Programming*, 4:295–312, 1985.

[Fit86] Melvin C. Fitting. Partial Models and Logic Programming. *Theoretical Computer Science*, 48:229–255, 1986.

[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.

[GL90] Michael Gelfond and Vladimir Lifschitz. Logic Program with Classical Negation. In David H.D. Warren and Peter Szeredi, editors, *Proceedings of the 7th Int. Conf. on Logic Programming*, pages 579–597. MIT, June 1990.

[Jay92] Bharat Jayaraman. Implementation of subset-equational programs. *Journal of Logic Programming*, 11(4):299–324, 1992.

[JJ94] D. Jana and B. Jayaraman. Set constructors, finite sets, and logical semantics. Technical Report TR 94-030, Department of Computer Science, SUNY-Buffalo, August 1994.

[JOM95] B. Jayaraman, M. Osorio, and K. Moon. Partial order programming (revisited). In M. Nivat V.S. Alagar, editor, *Proc. AMAST*, LNCS 936, pages 561–575. Springer-Verlag, 1995.

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

[Moo97] K. Moon. *Implementation of Subset Logic Languages.* PhD thesis, Department of Computer Science, SUNY-Buffalo, 1997.

[NS96] Ilkka Niemelä and Patrik Simons. Efficient Implementation of the Well-founded and Stable Model Semantics. In M. Maher, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 289–303, Bonn, Germany, September 1996. The MIT Press.

[OJ98] Mauricio Osorio and Bharat Jayaraman. Integrating the completion and the well-founded semantics. In Helder Coelho, editor, *Proc. Iberamia '98*, LNAI 1484, pages 230–241. Springer, 1998.

[OJ99] Mauricio Osorio and Bharat Jayaraman. Aggregation and negation-as-failure. *New Generation Computing*, 17(3):255–284, 1999.

[OJN99] Mauricio Osorio, Bharat Jayaraman, and J. C. Nieves. Declarative pruning in a functional query language. In Danny de Schreye, editor, *Proc. ICLP '99.* MIT Press, 1999.

[OJP99] Mauricio Osorio, Bharat Jayaraman, and David Plaisted. Theory of partial-order programming. *Science of Computer Programming*, 34(3):207–238, 1999.

[Sch92] John S. Schlipf. Formalizing a Logic for Logic Programming. *Annals of Mathematics and Artificial Intelligence*, 5:279–302, 1992.

[SZ91] D. Sacca and C. Zaniolo. Partial models and Three-Valued Models in Logic Programs with Negation. In Anil Nerode, Wiktor Marek, and V. S. Subrahmanian, editors, *Logic Programming and Non-Monotonic Reasoning, Proceedings of the first International Workshop*, pages 87–104, Cambridge, Mass., July 1991. Washington D.C, MIT Press.

[VG92] Allen Van Gelder. The Well-Founded Semantics of aggregation. In *Proc. of the 11th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, San Diego, USA*, pages 127–138. ACM Press, 1992.

[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.

[Zep97] Claudia Zepeda. Semanticas Bien Fundamentadas. Technical report, Department of Computer Science, UDLA, December 1997.

A An additional Theorem

Definition A.1
By the system $\vdash_{MP}$ we mean the system with no axioms and modus ponens as its only inference rule.

Definition A.2
A *normal* theory is a set of formulas of the form $\mathbf{tr}(P)$, where P is a normal program. A definite theory is a set of formulas of the form $\mathbf{tr}(P)$, where P is a definite program.

Definition A.3
Let T be a normal theory over the signature Σ. Let Σ_1 a signature such that $|\Sigma| = |\Sigma_1|$ and $\Sigma \cap \Sigma_1 = \emptyset$. Let f be 1-1 function $f : \Sigma \to \Sigma_1$. We define definite(α), where α is a normal formula recursively as follows:

- definite(a) $= a$ if a is an atom,
- definite($\neg a$) $= f(a)$ if a is an atom,
- definite($\alpha_1 \Rightarrow \alpha_2$) = definite($\alpha_1$) $\Rightarrow$ definite(α_2).

We also define definite(Γ), where Γ is a normal theory, as follows:

$$\text{definite}(\Gamma) = \{\text{definite}(C) : C \in \Gamma\}.$$

Lemma A.4
Let T be a normal theory and a an atom.
If $T \vdash_m a$ then definite$(T) \vdash_m a$.

Proof. Let $\alpha_1, \ldots, \alpha_n$ be a proof of "$T \vdash_m a$". We claim that

$$\text{definite}(\alpha_1), \ldots, \text{definite}(\alpha_n)$$

is a proof of "definite$(T) \vdash_m$ definite(a)". Our claim is proved by straightforward induction on the length of the proof. Since definite$(a) = a$ we are done.

Q.E.D

Lemma A.5
Let T be a definite theory and a an atom. If $T \vdash_m a$ then $T \vdash_{MP} a$.

Proof. Suppose that $T \vdash_m a$, then $T \models a$ and so there exists $n \in \mathcal{N}$ such that $a \in T_{P_1} \uparrow^n$, where $P_1 = \mathbf{tr}^{-1}(P)$ (this result is due to van Emden and Kowalski). But then it is easy to verify that $T \vdash_{MP} a$. **Q.E.D**

Lemma A.6
Let definite(P) defined over the signature $\Sigma \cup \Sigma_1$ and $a \in \Sigma$. If definite$(P) \vdash_{MP} a$, then $P \vdash_{MP} a$.

Proof. Let definite$(\alpha_1), \ldots,$ definite(α_n) be a proof of definite$(P) \vdash_{MP} a$. Notice that definite is 1-1 function, so definite^{-1} exists and moreover definite$^{-1}(\alpha \Rightarrow \beta) =$ definite$^{-1}(\alpha) \Rightarrow$ definite$^{-1}(\beta)$. Then we can verify that definite^{-1} preserves modus ponens. From these facts it follows easily by induction that $\alpha_1, \ldots, \alpha_n$ is a proof for $P \vdash_{MP} a$. **Q.E.D**

Theorem A.7
Let P be a normal program, and a an atom. Then $\boldsymbol{tr}(P) \vdash_m a$ implies $\boldsymbol{tr}(P) \vdash_{MP} a$.

Proof. Suppose $\mathbf{tr}(P) \vdash_m a$ then by Lemma A.4, definite$(\mathbf{tr}(P)) \vdash_m a$. Now by Lemma A.5, definite$(\mathbf{tr}(P)) \vdash_{MP} a$. Finally by Lemma A.6, $\mathbf{tr}(P) \vdash_{MP} a$. **Q.E.D**