Weak Properties of Circumscriptive Logic Programming

Pablo R. Fillottrani Guillermo R. Simari

Grupo de Investigación en Inteligencia Artificial
Departamento de Ciencias de la Computación
Universidad Nacional del Sur
Av. Alem 1253
(8000) Bahía Blanca – Argentina

e-mail: ccfillo@criba.edu.ar

Abstract

This is the second in a series of two papers in which we intend to study the formal properties of the semantics of Circumscriptive Logic Programs [6, 7]. The first one [8] was devoted to the *strong properties* [3], so called after its counterparts in nonmonotonic consequence relations [13, 20]. In this work we apply to this semantics the *weak properties* [4], specifically defined an extension of these principles to extended logic programs (they were originally defined for normal logic programs) and prove that the semantics of Circumscriptive Logic Programs is *well-behaved* [4, 5] in the sense it satisfies all "reasonable" principles.

1 Introduction

Declarative semantics for normal logic programs (logic programs with negative literals in the body of the clauses) have been introduced since the first implementation of PROLOG. Being logic programming a declarative paradigm, the operational characterization of PROLOG's *not* operator was an unacceptable contradiction. Clark's *predicate completion* [2] and Reiter's *closed world assumption* [25] were the first declarative semantics intended to solve this problem. But their definitions have some drawbacks, thus motivating some extensions [9, 14]. In parallel, the field of *nonmonotonic reasoning* became an active area of research in Artificial Intelligence, and thus provided further intuitions for representing negation-as-failure in a declarative way

[17, 22, 24]. Semantics such as the *stable models* [10] or the *well-founded* models are exemplar of the several semantics [26, 23] inspired by formalisms in nonmontonic reasoning.

In view of such a diversity of semantics for negation in logic programs, Dix [5, 3, 4] proposed a method for classifying and characterizing them. Inspired in similar work in nonmonotonic reasoning investigated by Gabbay, Makinson, Kraus, Lehmann and Magidor [13, 20], he introduced a nonmonotonic entailment relation "$\mid\!\sim$" for normal logic programs and study its properties under each of the semantics. These principles are classified into two types: *strong properties* are adaptations of those of nonmonotonic reasoning and belief dynamics [1], such as Cumulativity, Rationality, Cut, and Cautious Monotony; *weak properties* reflect the specific idea of negation-as-failure in logic programming. Dix also introduced the notion of *well-behaviour*, a subset of these properties all reasonable semantics should satisfy, and showed several of the semantics are not well-behaved. The reason of these irregularities is that, in order to improve a given semantics, some additional mechanism is included in top of its definition and this extension is not correctly handled in all programs. Well-behaviour thus provides the minimal conditions under which a semantics should be defined.

Negation in normal logic programs is then nonmonotonic under all proposed semantics. Although it formalizes the intuitions of the logic programming community, it has been shown that it is not enough for all knowledge representation applications [11, 12, 27]. Sometimes it is necessary to consider explicit negative information, and thus a monotonic negation operator is needed. *Extended logic programs*, defined in [11, 12], are logic programs with two kinds of negation: negation-as-failure as previously considered, and a new operator of *strong negation*. This strong negation operator satisfies monotonicity and resembles the negation in classical predicate calculus. Although there is a big difference with its semantics[1], some authors originally called it "classical negation". In [8] we proposed *circumscriptive logic programs*, an approach that includes both types of negations in a unified semantics based on pointwise circumscription [16]. In circumscriptive logic programs the programmer is able to specify the type of negation to apply to individual literals. If negation-as-failure is to be applied to all literals, then the semantics coincides with *stable models*. Otherwise, we can have in the same program one negative literal under strong negation, and other negative literal under negation-as-failure.

In [8] we study strong properties of circumscriptive logic programs. The main results are that its semantics is cumulative, and satisfies a restricted version of rationality. We continue this work in this paper by applying the weak properties of Dix' framework. First, we extend the definition of these weak principles to extended logic programs, *i.e.* to include strong negation. After that we analyze them in the context of circumscriptive logic programs. *Relevance* state that the addition of new independent information to a program does not change the meaning of old literals. *Reduction* formalizes the idea that if a literal is not in the head of a clause, then the semantics must consider it false. The principle of *Partial Evaluation* says that any semantics should assign the same meaning to a program and a partial evaluation of it. It has two variants according to the definition of a partial evaluation of a program. *Modularity*

[1]Mainly, it is not contrapositive and does not satisfy the principle of the excluded third.

enables us to compute the semantics of a program by considering certain "subprograms". Finally, *Isomorphism* and *Transformation* state that equivalent programs should have the same meaning. We show that some versions of all these principles are satisfied by the semantics of circumscriptive logic programs. In the other cases, we analyze the reasons of its failure. Nonetheless, considering results in this and the previous paper, we are able to prove that this semantics is well-behaved as defined by Dix [5, 4]

The paper is organized as follows. First, we review the syntax and semantics of circumscriptive logic programs. In section 3 we extend the definitions of the weak properties to extended and circumscriptive logic programs. Finally, using results from section 3 and from [8] we prove in section 4 its well-behaviour. As a corollary we can say that circumscriptive semantics when applied to definite programs coincide with its least Herbrand model, when applied to stratified normal programs it is equivalent to its supported model, and when applied to normal programs with all literals minimized it is equivalent to the well-founded semantics.

2 Circumscriptive Logic Programming

In this section we present the basic ideas of Circumscriptive Logic Programs [6], starting with syntactic considerations. We first introduce some common notions we will use in the following definitions.

Definition 2.1 Let $\mathcal{L}$ be a first order language without equality. With $\mathrm{Lit}_{\mathcal{L}}$ we denote the set of all ground literals in $\mathcal{L}$. L is a *positive* literal if $L = A$, being A an atom of the language. In case L is not a positive literal, then we say it is a *negative* literal. If $L \in \mathrm{Lit}_{\mathcal{L}}$ then the *complementary* literal $\overline{L}$ is the literal

$$\overline{L} = \begin{cases} \neg A & \text{if } L = A, \textit{ i.e.} \text{ it is a positive literal} \\ A & \text{if } L = \neg A, \textit{ i.e.} \text{ it is a negative literal} \end{cases}$$

It is clear that $\overline{\overline{L}} = L$.

Now, we define two distinct sorts of logic programs.

Definition 2.2 An *extended clause* in $\mathcal{L}$ is a clause of the form:

$$L_0 \leftarrow L_1, \ldots, L_n \tag{1}$$

where all $L_i, 0 \leq i \leq n$ are literals in $\mathrm{Lit}_{\mathcal{L}}$. If $n = 0$ the extended clause is called a *fact*. An *extended logic program* is a set of extended clauses.

Suppose the alphabet of the language $\mathcal{L}$ contains a unary predicate symbol $\mathtt{min}(\cdot)$ and one n-ary function symbol $f_p(\cdot)$ associated with each n-ary predicate $p(\cdot)$. Then we say that a *circumscriptive logic program* P in a language $\mathcal{L}$ is a set of extended clauses in $\mathcal{L}$ that includes the fact

$$\mathtt{min}(f_{\mathtt{min}}(X)) \leftarrow \tag{2}$$

The only difference between extended logic programs and normal logic programs [19] is the possible occurrence of negative literals in the head of the clauses. Negative literals in extended logic programs can be interpreted under the negation as failure or the strong negation semantics. Circumscriptive logic programs do not generalize extended logic programs in the sense they form a bigger class of programs. They are just extended logic programs which give special meanings to some of the symbols in the language. Following PROLOG's notation, we will write in this paper predicate and function symbols Starting with a lowercase letter and variable symbols with an uppercase letter. Predicate `min` and functions f_p will be used for specifying the *circumscriptive policy* in a logic program, to avoid the use of second order logic. We will write $\texttt{min}(p(t))$ for $\texttt{min}(f_p(t))$ being its difference clear from the context.

Definition 2.3 Let $X \subseteq \text{Lit}_{\mathcal{L}}$. We call X *logically closed* if it is equal to $\text{Lit}_{\mathcal{L}}$ or it does not contain a pair of complementary literals. If P is an extended logic program, the we say that X is *closed under* P if, for every clause $L_0 \leftarrow L_1, \ldots, L_n$ with $n \geq 0$, then $\{L_i, 1 \leq i \leq n\} \subseteq X$ implies that $L_0 \in X$. Then we define $Cn(P)$ to be the smallest subset of $\text{Lit}_{\mathcal{L}}$ which is both logically closed and closed under P.

According to this definition $\text{Lit}_{\mathcal{L}}$ is always logically closed and closed under any program P in $\mathcal{L}$. Thus the set $Cn(P)$ always exists. This set of literals is interpreted as the "consequences" of an extended logic program, and these consequences are only determined by the clauses of the program and, in case of inconsistency, by the set $\text{Lit}_{\mathcal{L}}$.

Definition 2.4 Let P be an extended logic program. We say that P is *inconsistent* if $Cn(P) = \text{Lit}_{\mathcal{L}}$, and *consistent* otherwise.

If a program P is consistent, then it can be shown that $Cn(P)$ is the smallest set closed under P. Otherwise, P is inconsistent and $Cn(P) = \text{Lit}_{\mathcal{L}}$.

Circumscriptive logic programs allow negative literals to be inferred by the rule of "negation as failure". In order to define their semantics, we introduce a process that gives an interpretation to every literal to which we can apply negation as failure. The resulting program does not contain "negation as failure", and then it can be seen as an extended logic program with its consequences already defined by $Cn(\cdot)$.

Definition 2.5 Let P be a circumscriptive logic program, and $X \subseteq \text{Lit}_{\mathcal{L}}$. The *reduct of* P *relative to* X is the extended program obtained from P by applying the following rules:

- deleting each clause $L_0 \leftarrow L_1, \ldots, L_n$ such that there exists $i, 1 \leq i \leq n$ with $\texttt{min}(\overline{L_i}) \in X$ and $\overline{L_i} \in X$.
- replacing each remaining clause $L_0 \leftarrow L_1, \ldots, L_n$ such that there exists $i, 1 \leq i \leq n$ with $\texttt{min}(\overline{L_i}) \in X$, by the clause $L_0 \leftarrow L_1, \ldots, L_{i-1}, L_{i+1}, \ldots, L_n$.

This program is noted by P^X.

The program P^X does not contain literals L such that $\texttt{min}(L) \in X$. Intuitively, we can think that X is the circumscriptive policy that specify which literals contain negation as failure. The first rule in the definition eliminates from the program all clauses that use inconsistent literals with negation as failure. The second rule simply eliminates from the rest of the clauses all literals with negation as failure, since they can be assumed to be true.

Finally, to define the consequences of a circumscriptive logic programs we must consider the least and greatest fixpoints of the following operator γ_P.

Definition 2.6 For any circumscriptive program P, the function $\gamma_P : 2^{\text{Lit}_\mathcal{L}} \longmapsto 2^{\text{Lit}_\mathcal{L}}$ is defined for any $X \subseteq \text{Lit}_\mathcal{L}$ as the set $Cn(P^X)$.

This function γ_P is anti-monotone for every circumscriptive logic program P, *i.e.* if $X \subseteq Y$ then $\gamma_P(Y) \subseteq \gamma_P(X)$. It follows that γ_P^2 is monotone.

Definition 2.7 Let $L \in \text{Lit}_\mathcal{L}$ and P a circumscriptive logic program. We say that L is *well-founded* relative to P if L belongs to the least fixpoint of γ_P^2, and that L is *unfounded* relative to P if L does not belong to the greatest fixpoint of γ_P^2. L is called a *minimized literal* if $\texttt{min}(L)$ is a well-founded literal, and L is called a *default literal* if $\overline{L}$ is a minimized literal and $\overline{L}$ is an unfounded literal. The *circumscriptive consequences of* P, noted by $\mathcal{M}_P$, is the least logically closed set that contains all well-founded and default literals of P.

If P is a definite program, then M_P denotes its minimal Herbrand model. We can consider P as a circumscriptive logic program, and then the minimal Herbrand model coincides with the set of circumscriptive consequences of P. This is because all literals in M_P are well-founded, and there exists no default literal in P. In case all positive literals are minimized, then all unfounded literals are default literals and this semantics is the same as the well-founded semantics.

Although in [6, 8] we have presented the semantics of circumscriptive logic programs in different way, both are equivalent [6]. This definition will allow an easier proof of the properties, whilst the other is more related to circumscription and the other formalizations of nonmonotonic reasoning in artificial intelligence.

3 Weak Properties for Extended Logic Programs

In this section we will extend the framework for comparing semantics for logic programs. We will first introduce some common concepts that are necessary for the definitions, and then present each of these properties.

We will denote by $SEM(P)$ the set of literals in the language of a logic program P that are inferred under a particular semantics. For example, in the case of circumscriptive logic programs $SEM(P) = \mathcal{M}_P$. Nevertheless this notation can be consider under all known semantics. In this context we say that the *definition* of a literal is the set of clauses in the program that are needed for determining its truth value.

Definition 3.1 If P is an extended logic program and $L \in \text{Lit}_{\mathcal{L}}$ then the *definition of L in P* is the following set of clauses:

$$Def_P(L) = \{L_0 \leftarrow L_1, \ldots, L_n \in P \ : \ L_0 = L \text{ or } L_0 = \overline{L} \text{ or } L_0 = \texttt{min}(L) \text{ or } L_0 = \texttt{min}(\overline{L})\}$$

The following principles were introduced by Dix in [3, 4] for normal logic programs (logic programs whose clauses only allow positive literals in their heads). In order to apply them to circumscriptive logic programs, and in general to extended logic programs, we must first generalize their definitions. For some properties, like isomorphy, this extension is not immediate since it is necessary to take care of the special meanings for some symbols in the language. Other properties, like isomorphy, need not be reformulated.

3.1 Principle of Partial Evaluation

The principle of partial evaluation (PPE) states that a reference to an atom in the body of a rule is equivalent to a reference to the body of the rule from which the atom was derived. Recall that the definition of an atom in a program is the set of rules that have it in their heads. Then, if PPE is satisfied we can replace each occurrence of an atom by the bodies of the rules from its definition.

Definition 3.2 Let P be an extended logic program and $L \in \text{Lit}_{\mathcal{L}}$ such that $\texttt{min}(L)$ and $\texttt{min}(\overline{L})$ do not occur in P. If $Def_P(L) = \{L \leftarrow L_1^i, \ldots, L_{k_i}^i, 1 \leq i \leq s\}$ then let P_L be the program obtained from P by deleting the definition of L and replacing each occurrence of L in a clause

$$L_0 \leftarrow L_1, \ldots, L_j, A, L_{j+1}, \ldots, L_n$$

by the set of rules

$$\{L_0 \leftarrow L_1, \ldots, L_j, L_1^i, \ldots, L_{k_i}^i, L_{j+1}, \ldots, L_n, 1 \leq i \leq s\}$$

Then if $SEM(P_L) = SEM(P) \backslash \{L, \overline{L}\}$ we will say that SEM satisfies the *principle of partial evaluation*.

Note that the literal L does not appear in the program P_L. Then PPE says that the meaning of P is the meaning of P_L plus the meaning of L, allowing to reduce the Herbrand base of the program by eliminating literals L and $\overline{L}$. Any semantics that is sound with respect to a variant of SLD resolution should satisfy this principle.

In order to apply this principle to extended logic programs we allowed the program P_L to be described for negative literals L. The original definition in [4] only applies to positive literals. This can be considered as a generalization of the principle. Alternatively, PPE cannot be applied to literals with negation as failure. So we require in its definition that literals $\texttt{min}(L)$ and $\texttt{min}(\overline{L})$ do not appear in the program. This restriction can be seen as a counterpart of not allowing negative literals in the original definition.

3.2 Relevance

In any semantics the meaning of a literal should not change under the addition or deletion of irrelevant clauses. In other words, the truth value of a literal only depends on the clauses the programmer explicitly states for that literal. This fact is captured by the principle of relevance.

Definition 3.3 Let P be an extended logic program, $L \in \text{Lit}_{\mathcal{L}}$ and $SEM(P) \subset \text{Lit}_{\mathcal{L}}$. We define the *relevant rules of L in P* as the set of clauses in the program that contributes to the definition of L or its complement:

$$Relevant_P(L) = \bigcup_{L' \in Def_P(L)} Def_P(L')$$

In this definition we consider that a literal belongs to a set of clauses if it belongs to the head or to the body of a clause in the set. We then say that SEM satisfies *relevance* if $SEM(P)(L) = SEM(Relevant_P(L))(L)$.

Although being a perfectly reasonable property, relevance is not satisfied by the semantics of the stable models, the most popular semantics for normal logic programs. The following program [4] proves this fact.

Example 3.4 Let $P_1 = \{a \leftarrow \neg b,\ b \leftarrow \neg a,\ p \leftarrow \neg p\}$, and $P_2 = P_1 \cup \{p \leftarrow \neg a\}$. P_1 has no stable model, but P_2 has the stable model $\{b,\ p\}$. This implies that the meaning of b in this semantics depends on the addition of the irrelevant clause $p \leftarrow \neg a$.

Evidently we should avoid this strange behaviour by requiring relevance to be satisfied. For circumscriptive logic programs we must include in the definition of a literal L those literals in clauses for $\texttt{min}(L)$. The following example shows it is necessary.

Example 3.5 Let $P_3 = \{\min(a) \leftarrow c, b \leftarrow \neg a, c \leftarrow\}$. Suppose the clause $c \leftarrow$ is not included in the definition of b, then it follows that the circumscriptive semantics does not satisfy relevance.

3.3 Reduction

In order to define the principle of reduction, we must first characterize a reduction operation in logic programs that is in close relation with definition 2.5.

Definition 3.6 Let P be an extended logic program and $M \subseteq \text{Lit}_{\mathcal{L}}$. Then the *reduction of P by M*, noted by $P^{\div M}$, is the program $P^M \backslash Def(M)$.

This reduction eliminates all literals from M in the program P, by interpreting them true in every occurrence. The difference between the definition 2.5 and this definition is that in the former we only give interpretation to literals in the body of the clauses, whilst in the latter we also eliminate all occurrences in the head of the clauses. Note also that the program P^M is not necessarily definite.

The principle of reduction generalizes a very intuitive property: adding literals to a program P does not change the meaning in P of the rest of the literals.

Definition 3.7 Let P be a logic program and $M \subseteq \text{Lit}_{\mathcal{L}}$. A semantics SEM satisfies the principle of *reduction* if and only if $SEM(P \cup M) = SEM(P^{\div M}) \cup M$.

All known semantics for normal logic programs satisfy this property, since it is a very weak one. It only assures that explicit facts are considered true in every occurrence in the program.

3.4 Modularity

The principle of modularity enables us to split a logic program into two subprograms or "modules". One module contains the definition of a literal, and the other module the rest of the program. Modularity assures that the semantics of the whole program P does not change if we give an interpretation of the second part according to the semantics of the first. This principle has a close relationship with PPE, reduction and relevance. In fact, several deducible results have been shown [5].

Definition 3.8 Let P be a logic program, $P = P_1 \cup P_2$, $B_{P_1} \cap B_{P_2} = \{A\}$ and $P_2 = Relevant_P(A)$. Then SEM satisfies *modularity* if $SEM(P) = SEM(P_1^{SEM(P_2)} \cup P_2)$.

If the principle of modularity holds for a logic program, then it is possible to apply the process of "splitting" described in [18]. This allows us to reduce the Herbrand universe of a program by a set of several predicates, instead of only one predicate as PPE states.

3.5 Isomorphy

It is a very reasonable property that equivalent logic programs should have the same semantics. The problem is to formalize this notion of "equivalence" between logic programs. One possibility is to consider two logic programs equivalent if one is obtained from the other by renaming each predicate and function constant. This property is called isomorphy.

Definition 3.9 Let P and Q be two logic programs, and I an isomorphism between B_P and B_Q. The principle of *isomorphism* states that $I(SEM(P)) = SEM(Q)$.

In other words, the semantics of a program does not depend on the name given to functions and predicates. In order to keep this property for circumscriptive logic programs we need a further condition on the isomorphism $I()$: $I(\texttt{min}) = \texttt{min}$ and $I(f_p^P) = f_{I(p)}^Q$ for all predicates p in the language. This restriction to the notion of isomorphism is due to the fact that circumscriptive logic programs assign a special meaning to the predicate $\texttt{min}()$ and the functions f_p^P in P and f_p^Q in Q.

3.6 M_P extension

Definite programs (programs without negation) have a well-established semantics determined by its least Herbrand model, which it is also the intersection of all its models. If P is a logic

program, then M_P is its least Herbrand model. The principle of M_P extension preserves this semantics when applied to definite programs.

Definition 3.10 Let P be a definite program. Then a semantics SEM satisfies the principle of M_P *extension* if and only if $SEM(P) = M_P$.

Thus extending the class of programs on which the semantics is defined should not change the meaning of definite programs. This definition has no changes with that in [4].

3.7 Transformation

Transformation is a property that models the idea that derivation without negative literals should follow the rules of the programs. We need first to define the transformation of a logic program.

Definition 3.11 Let P be a logic program, A an atom in its language and P^+ (respectively P^-) be the program obtained from P by canceling all negative literals (resp. canceling all clauses containing a negative literal). Then a semantics SEM satisfies the principle of *transformation* if and only if the following holds:

1. if $\neg A \in SEM(P^+)$, then $SEM(P \cup \{\neg A\}) \subseteq SEM(P)$.
2. if $A \in SEM(P^-)$, then $A \in SEM(P)$.

This principle, together with those introduced before, is necessary to recover the well-founded models semantics from all the above defined properties.

4 Well-behaviour of Circumscriptive Logic Programs

In this section we apply these weak principles to circumscriptive logic programs. We first the notion of well-behaviour of a semantics, introduced in [5, 4], and then analyze the semantics of circumscriptive logic programs.

Definition 4.1 A semantics for logic programs is called *well-behaved* if it satisfies relevance, reduction, modularity, isomorphy, M_P-extension, transformation and cut.

Well-behaviour of any semantics is based on the satisfaction of the properties defined in section 3, and the strong property *Cut*. These conditions are indeed weak if they are individually considered, since most semantics satisfies the majority of them. Taking them as a whole can be considered as an axiomatic framework that captures the intuitions under nonmonotonic and strong negation in logic programming.

Now we prove that circumscriptive logic program satisfy PPE, relevance and reduction. Then we will show that the rest of the properties follows from these and from some results in [5]. We will then be able to state that circumscriptive semantics is a well-behaved semantics. All proof are given in a schematic way, in order to avoid an excess of technical details.

Lemma 4.2 *If P is a circumscriptive logic program, then $\mathcal{M}_P$ satisfies PPE.*

Proof Let P be a circumscriptive logic program and $L \in \mathrm{Lit}_{\mathcal{L}}$ in the conditions of definition 3.2, we must show that $\mathcal{M}_P = \mathcal{M}_{P_L}$. We first prove by transfinite induction that the well-founded and the unfounded sets of literals in P and P_L are the same except of $L, \overline{L}$. Recall that these sets are the least fixpoint and the complement of the greatest fixpoint of γ_P and γ_{P_L}. It is necessary in this step to require that negation as failure is not applied neither in L nor in $\overline{L}$. This implies the following facts, being L' a literal such that $L' \neq L, \overline{L}$,

$$L' \text{ is well-founded in } P \text{ if and only if it is well-founded in } P_L \tag{3}$$

and also

$$L' \text{ is unfounded in } P \text{ if and only if it is unfounded in } P_L. \tag{4}$$

Then to prove the equivalence of the semantics of P and P_L we must consider two cases: L' is a well-founded literal and L' is an unfounded literal. The first case follows immediately from remark (3). To prove the second case, we must recall from definition 2.7 that if L' is a default literal in P or in P_L, then $\mathtt{min}(L')$ is well-founded and $\overline{L'}$ is an unfounded literal. The first case is covered by remark (3), and the second by remark (4). ∎

Lemma 4.3 *If P is a circumscriptive logic program, then $\mathcal{M}_P$ satisfies relevance.*

Proof Also for relevance, we must prove the equivalence of the semantics of the programs P and $Relevant_P(L)$ for all literals L. Again this is done by proving by transfinite induction that $Cn(P)$, restricted to literals in $Relevant_P(L)$, and $Cn(Relevant_P(L))$ coincide for all literals L. In this sense it is necessary to include in $Relevant_P(L)$ not only those clauses that define L or $\overline{L}$, but also those that define $\mathtt{min}(L)$ and $\mathtt{min}(\overline{L})$. ∎

Lemma 4.4 *If P is a circumscriptive logic program, then $\mathcal{M}_P$ satisfies reduction.*

Proof It is necessary to prove that $\mathcal{M}_{P \cup M} = \mathcal{M}_{P^M} \cup M$. We first observe that by inclusion [8] $M \subseteq \mathcal{M}_{P \cup M}$. Also, by the definition 3.6 $\mathcal{M}_{P^M} \subseteq \mathcal{M}_{P \cup M}$. Then $\mathcal{M}_{P^M} \cup M \subseteq \mathcal{M}_{P \cup M}$.

To prove the other way we consider a literal $L \in \mathcal{M}_{P \cup M}$ such that $L \notin M$. By transfinite induction we prove that if L is well-founded in $P \cup M$ then it is well-founded in P^M, and L is unfounded in $P \cup M$ then is unfounded in P^M. From these two facts it follows that $\mathcal{M}_{P \cup M} \subseteq \mathcal{M}_{P^M}$. ∎

We are now in conditions to establish the main result of this work, the well-behaviour of the circumscriptive semantics for circumscriptive logic programs. Previous lemmas proved the main properties, we will use some results in [5] and [8] to prove modularity. The rest of the principles follows easily from the definitions. Observe that when we apply lemma 5.20 from [5] we need an adaptations to our definitions, which is immediate.

Theorem 4.5 *For all circumscriptive logic programs P, $\mathcal{M}_P$ is a well-behaved semantics.*

Proof From lemmas 4.2, 4.3, and 4.4 we know that $\mathcal{M}_P$ satisfies PPE, relevance and reduction. From theorem 4.3 in [8] we know that this semantics also satisfies *cut* and *cautious monotony*. Although cautious monotony is not required for a semantics being well-behaved, we need it to apply lemma 5.20 in [4] (or its reformulation according to our framework) to obtain modularity. M_P-extension is immediately satisfied as argued above. The property of isomorphy requires the restrictions mentioned in definition 3.9. If an isomorphism follows them, then any other semantics considerations in $\mathcal{M}_P$ are independent of the underlying language.

Therefore to be able to prove the well-behaviour of this semantics, we only need transformation. Let $\neg A$ be a negative literal such that $\neg A \in \mathcal{M}_{P^+}$. Then $\neg A$ is a well-founded literal in P^+. Every well-founded literal in $\mathcal{M}_{P\cup\{\neg A\}}$ must also be well-founded in P, since the positive literal A cannot be inferred from $\neg A \in \mathcal{M}_{P^+}$. And every unfounded literal in $P \cup \{\neg A\}$ must also be unfounded in P, since the positive literal A cannot be inferred from $\neg A \in \mathcal{M}_{P^+}$. The proof for the other condition is symmetric with the previous one. ∎

5 Conclusions

Several semantics for normal logic programs have been proposed in recent years. Each one supported by more or less "intuitions", and with no general agreement about which are the right intuitions. Structural properties were introduced by Dix in [5] to solve this problem. Every semantics is associated with an entailment relation, and these properties describe its behaviour. This principles are classified into two types: strong properties and weak properties. Certain subset of these properties can be used to rule out strange behaviour of some formalisms. especially in the case of negation as failure.

In [6] we presented circumscriptive logic programs in order to provide a unifying semantics for negation as failure and strong negation in logic programming. The semantics for these programs is based in a preference relation, like the one that characterizes circumscription [21, 16, 15], but defined over sets of literals instead of models. We claim that this semantics captures both kinds of negations, so we checked if it satisfies the Dix' framework. The first step was to generalize strong and weak properties that were defined for normal logic programs, to the class of extended logic programs. For example, properties like PPE and isomorphy, require some changes in their formulations. Finally, we proved that circumscriptive semantics satisfies all principles needed to be a *well-behaved* semantics as defined in [5]. As a consequence. this semantics coincide in three classes of programs with well-known semantics: in definite programs, with the least Herbrand model; in stratified programs, with the supported model; and in normal programs, with the well-founded models. In these last two cases only negation as failure is applied; in definite programs there is no negation.

In further work we can extend this analysis to other generalizations of logic programming, like disjunctive programs. It will be also useful to supplement this semantics, and any other

semantics of strong negation, with some mechanisms of contradiction removal. In all known approaches for this kind of negation, in case of contradiction every literal is inferred. This policy has the disadvantage of ignoring all valid inference in case of contradictory consequences. This fact is reflected in the consistency condition for the principle of relevance.

References

[1] C. E. Alchourrón, P. Gärdenfors, and D. Makinson. On the logic of theory change: contraction functions and their associated revision functions. *Journal of Symbolic Logic*, 50, 1985.

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

[3] J. Dix. A classification theory of semantics of normal logic programs: I. strong properties. *Fundamenta Informaticae*, 1994.

[4] J. Dix. A classification theory of semantics of normal logic programs: II. weak properties. *Fundamenta Informaticae*, 1994.

[5] J. Dix. Semantics of logic programs: Their intuitions and formal properties. In *Logic, Action and Information*. de Gruyter, 1995.

[6] P. R. Fillottrani. Sistemas de razonamiento no monótono y su relación con la semántica de las bases de datos deductivas. Tesis de Magister en Ciencias de la Computación. Universidad Nacional del Sur, Bahía Blanca, Argentina., 1995.

[7] P. R. Fillottrani and G. R. Simari. Circumscriptive logic programming. In *Proceedings of the XIV International Conference of the Chilean Computer Science Society*, Concepción, Chile, 1994.

[8] P. R. Fillottrani and G. R. Simari. Strong properties of circumscriptive logic programming. In *Proceedings CACIC'96, Segundo Congreso Argentino de Ciencias de la Computación.*, San Luis, Argentina, 1996.

[9] M. C. Fitting. A kripke-kleene semantics for general logic programs. *Journal of Logic Programming*, 2, 1985.

[10] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proceedings of the 5th. International Conference on Logic Programming*, Seattle, WA, 1988.

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

[12] R. Kowalski and F. Sadri. Logic programs with exceptions. In *Proceedings of the 7th. International Conference on Logic Programming*, Jerusalem, Israel, 1990.

[13] S. Kraus, D. Lehmann, and M. Magidor. Nonmonotonic reasoning, preferential models and cumulative logics. *Artificial Intelligence*, 44, 1990.

[14] K. Kunen. Negation in logic programming. *Journal of Logic Programming*, 4, 1987.

[15] V. Lifschitz. Computing circumscription. In *Proceedings of the 9th. International Joint Conference on Artificial Intelligence*, Los Angeles, CA, 1985.

[16] V. Lifschitz. Pointwise circumscription. In M. L. Ginsberg, editor, *Readings in Nonmonotonic Reasoning*, pages 179–193. Morgan Kaufmann Publishers, 1987.

[17] V. Lifschitz. On the declarative semantics of logic programs with negation. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programs*, pages 177–192. Morgan Kaufmann Publishers, Los Angeles, CA, 1988.

[18] V. Lifschitz. Foundations of logic programs. In G. Brewka, editor, *Principles of Knowledge Representation*. CSLI Publications, 1996.

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

[20] D. Makinson. General patterns in nonmonotonic reasoning. In D. Gabbay, editor, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume III, pages 35–110. Oxford University Press, 1990.

[21] J. McCarthy. Circumscription - a form of non-monotonic reasoning. *Artificial Intelligence*, 13, 1980.

[22] T. C. Przymusinski. On the declarative semantics of deductive databases and logic programs. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programs*, pages 193–216. Morgan Kaufmann Publishers, Los Angeles, CA, 1988.

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

[24] T. C. Przymusinski. Three-valued nonmonotonic formalisms and semantics of logic programs. *Artificial Intelligence*, 49, 1991.

[25] R. Reiter. On closed world data bases. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 55–76. Plenum Press, New York, NY, 1978.

[26] A. van Gelder, K. Ross, and J. Schlipf. The well-founded semantics for general logic programs. *Journal of the Association for Computing Machinery*, 37, 1990.

[27] G. Wagner. Logic programming with strong negation and inexact predicates. *Journal of Logic Computation*, 1(6), 1991.