Negation-as-failure considered harmful

Pablo R. Fillottrani
Departamento de Ciencias e Ingeniería de la Computación
Universidad Nacional del Sur
Av. Alem 1253, 8000 Bahía Blanca, Argentina
prf@cs.uns.edu.ar

Abstract

In logic programs, negation-as-failure has been used both for representing negative information and for providing default nonmonotonic inference. In this paper we argue that this twofold role is not only unnecessary for the expressiveness of the language, but it also plays against declarative programming, especially if further negation symbols such as strong negation are also available. We therefore propose a new logic programming approach in which negation and default inference are independent, orthogonal concepts. Semantical characterization of this approach is given in the style of answer sets, but other approaches are also possible. Finally, we compare them with the semantics for logic programs with two kinds of negation.

Keywords: negation as failure, logic programming, knowledge representation

1 INTRODUCTION

The utility of a language as a tool for practical development of knowledge representation systems is grounded on a simple syntax with intuitive semantics, and efficient proof procedures. So, in order to keep a broad scope of users and applications, the language should be both powerful and simple at the same time. The absence of any of these conditions produces languages with theoretical interest, but difficult to use in real applications. We think the original success of Logic Programming was due to the fulfillment of these requirements, so any extension must also preserve them.

Negation-as-failure partially satisfied these properties at the beginning, despite its lack of declarative meaning in all programs. The semantics supported for stratified programs is intuitive, and general enough for extended use. Semantics for nonmonotonic reasoning, such as the stable models or the well founded semantics, filled the gap and provided an elegant characterization valid for all programs. They also helped in understanding the close relationship between "not" and nonmonotonic reasoning formalisms, such as default logic, circumscription or autoepistemic logics. As soon as this relationship became clear, it was evident that negation-as-failure plays a dual rôle in the language of logic programs: it is both a negative connective, and a default, nonmonotonic rule of inference. As a negative connective, it represents negative information, and as a default rule of inference it allows to draw conclusions from the absence of facts.

The fact that one symbol holds both these functions is not helping in the simplicity of the knowledge representation language, since all negative information must be nonmonotonically inferred, and nonmonotonic inference rules must be defined with only negative consequences. None of the nonmonotonic logics previously mentioned share this property with logic programs. This is a restriction on the expressive power of the language, and forces complex ways of representing knowledge such as the introduction of new positive names for negative facts.

Therefore, the negation-as-failure mechanism was shown to be inadequate for representing explicitly negative information [10, 23, 13, 26]. Extended logic programs were introduced with the possibility of referring to another negation symbol "¬", called strong, explicit, or classical negation. Therefore, syntax and semantics of logic programs have been extended to reflect the distinction between asserting something false, and denying something true.

However, the presence of two negative symbols goes against the necessary simplicity for logic programming languages. When using negative information to infer new facts, it is necessary to evaluate if negation-as-failure or strong negation is the right choice. Moreover, most proposed semantics for extended programs do not relate both meanings, yielding two completely independent semantics for negative connectives.

In this paper we argue that in the presence of strong negation, negation-as-failure is no longer necessary for representing negative information. Consequently, negation-as-failure remains in logic programs only as a nonmonotonic inference rule, which is better referred to independently of the way negative and positive information is represented. Formalisms like default logic, circumscription, or autoepistemic logic can be taken as models in this sense. We propose a syntax of logic programs without negation-as-failure, but with strong negation and non-monotonic inference rules. The semantics associated with this approach is given in the style of answer set semantics, but other approaches such as well founded models are also possible.

The structure of the paper is as follows: in Section 2, we perform a more detailed analysis of the above mentioned problems, mostly attributable to the lack of distinction between negation and nonmonotonic inference in logic programs. In Section 3 we propose to extend the influence of nonmonotonic reasoning into logic programming, not only to provide a semantics for negation-as-failure, but also to induce a framework of logic programs where negation and default reasoning are independent concepts. In Section 4 we apply this approach to some examples and show some semantic properties. Finally, in Section 5 we compare it with known semantics for extended logic programs with two kinds of negation.

2 "NOT" REVISITED

We consider in this section the language of extended logic programs from the viewpoint of the requirements for useful knowledge representation tools, identifying some problems and analyzing their causes. In this sense, we will shortly review the concept of declarative language and show that negation-as-failure can be considered harmful for the declarativeness of logic programs, in a similar way Dijkstra [5] analyzed the GOTO statement in the context of structured programming.

Informally, a declarative programming language is one that specifies *what* is computed, instead of *how* it is done. Baral and Gelfond [3] consider McCarthy to be the first advocate for representing knowledge in a declarative way:

> "... expressing information in declarative sentences is far more modular than expressing it in segments of computer programs or in tables. Sentences can be true in a much wider context than specific programs can be used. The supplier of a fact does not have to understand much about how the receiver functions or how or whether the receiver will use it. The same fact can be used for many purposes, because the logical consequences of collections of facts can be available."

In our view, the key point is to establish a clear distinction between the supplier of a fact, and its receiver or user. The receiver does not know how the fact is produced, nor does the supplier know how it is used.

In a logical system, this problem is translated into drawing the boundary between logic and control in Kowalski's [12] famous equation $algorithm = logic + control$. In this sense, John Lloyd [20] proposed that a program is declarative if it may be considered as a formal theory and its results may be obtained as deductions from the theory. This is a broad criterion that all formal systems satisfy as long as they specify a language and a proof theory. But the main advantage of declarativeness is being modular, *i.e.* allowing the theory to be composed with theories, possibly written by other programmers.

Let us then define the components of a formal theory from such logic programs. A *rule* is a clause of the form

$$L_0 \leftarrow L_1, \ldots, L_m, not\ L_{m+1}, \ldots, not\ L_n \tag{1}$$

where all $L_i, 0 \leq i \leq n$ are literals (ground atoms A, or strong negated atoms $\neg A$). A *logic program* is then a set of rules, and a *query* is a finite set of literals. It is clear that literals belong to the language of the formal theory, since they bear truth values and are deduced by the inference rules. Rules are generally not considered as members of the language. Thus, we consider rules of the form (1) as inference rules of the formal theory determined by a specific logic program.

Traditionally, these rules have been split into two parts: head and body. The head, literal L_0 in (1), is where the truth value of the literal is defined. The program containing a rule with a literal in its head may be regarded as the *supplier* of the fact, according to McCarthy's use of the word. On the other hand, in the body of the rule is where facts are *used*. We have not yet established which are the facts in this part since they depend on the interpretation of "not". However, since we want the language to be declarative, these facts must not express the way in which they are inferred. Note that a query is the other possible place where facts can be used.

The status of "not" is still to be determined. In principle, there are two possible alternatives. It is possible to consider rules where "not" appears as nonmonotonic inference rules, similar to the ones in default logic. By doing so, rules of the form (1) are read as "if $L_1, \ldots, L_m$ are true, and there is not enough evidence for $L_{m+1}, \ldots, L_n$, then L_0 is true". If this is the case, then the language only contains literals with explicit negation and "not" has nothing to do with negation. Moreover, "$not\ A$" and "$not\ \neg A$" do not have truth value; instead, they express conditions in the meta-language, and so should not be part of queries. Otherwise, "not" can be treated as a connective. Here, the language is formed by extended literals (literals, and applications of "not" to literals), queries may contain "not", and rules of the form (1) are monotonic inference rules with the usual meaning "if $L_1, \ldots, not\ L_n$ are true then L_0 is true".

Both alternatives are feasible. In fact, one of the first proposals for the semantics of logic programs with two kinds of negation has been proposed as a translation of the logic program into a default logic theory [11]. However, it is not possible to easily extend this translation to other semantics, and in order to be coherent with its use, a change in the syntax of "not" is necessary. General use suggests "$not\ A$" as being a negative fact, assigning the values `true` or `false` just as in any other formula in the language.

Even though "$\neg$" and "not" are two connectives in the alphabet of the language, they are very special ones. The primitive function of a connective is to build a new formula from smaller ones. This syntactical view does not require a truth functional definition of the meaning of the new formula from the truth values of its components. Nevertheless, traditional connectives in propositional logics such as "$\vee$", "$\wedge$", etc. do have clear truth tables. If a formal theory satisfies this property, then it is possible to replace one of the components in a formula by any other formula with the same truth value. However, this is not the case for the two negative connectives in logic programming; truth for negated literals are not a direct consequence of the truth of its components. Furthermore, in the case of "$\neg$", most semantic characterizations only specify a sort of consistency condition when both "A"

and "$\neg A$" are provable.

Another problem is that these negative connectives cannot be nested, which is not actually a problem for "$\neg$" since it is easy to change syntax and semantics of logic programs to allow nested explicit negation. However, "$not\ not\ A$" does not have a clear negative semantics, particularly a declarative one. An approach to assign semantic to these expressions is given in [19]. If we read "$not\ A$" as "A is not known" or "A is not believed", the problem does not appear. In fact, general usage of this connective suggests this interpretation as it follows from the example below [10].

Example 2.1 A college is awarding scholarships to its students. Examination scores are represented by predicates $\texttt{highGPA}(\cdot)$ and $\texttt{fairGPA}(\cdot)$, and $\texttt{selected}(\cdot)$ represents those who where chosen.

$$\begin{array}{lcl}
\texttt{selected}(X) & \leftarrow & \texttt{highGPA}(X) \\
\texttt{selected}(X) & \leftarrow & \texttt{minority}(X), \texttt{fairGPA}(X) \\
\neg\texttt{selected}(X) & \leftarrow & \neg\texttt{fairGPA}(X), \neg\texttt{highGPA}(X) \\
\texttt{interview}(X) & \leftarrow & not\ \texttt{selected}(X), not\ \neg\texttt{selected}(X)
\end{array}$$

The last rule in the program shows a body where an atom appears both negatively and positively. Indeed, if "not" represents negation, then the rule is meaningless. ∎

The real semantics of these occurrences of "not" is "not known", as in Moore's autoepistemic modal operator [22]. The representation of negative information by "not" should avoid this confusion, even if it cannot be nested.

From the semantical point of view, another serious problem arises. Even when both are negative connectives, there is no connection between "$\neg$" and "not" in most semantics. The following examples show this fact. Example 2.2 presents an atom A such that in the well-founded semantics $\neg A$ is $\texttt{true}$ and $not\ A$ is $\texttt{false}$. Example 2.3 shows the opposite situation.

Example 2.2 The following program has no answer set, and its well-founded model is $\langle\{\neg\texttt{p}\}, \emptyset\rangle$.

$$\begin{array}{lcl}
\texttt{p} & \leftarrow & not\ \texttt{p} \\
\neg\texttt{p} & \leftarrow &
\end{array}$$

The well-founded semantics accepts $\neg\texttt{p}$ and sets $not\ \texttt{p}$ as undefined. Technically, the set of consequences under the answer set semantics is the whole language. ∎

Example 2.3 In this example, we will follow the definitions and notations in [18]. Suppose the language contains the constant 0 and the unary function symbol $\texttt{s}(\cdot)$ to represent all natural numbers. The logic program

$$\begin{array}{lcl}
\texttt{even}(0) & \leftarrow & \\
\texttt{even}(\texttt{s}(X)) & \leftarrow & not\ \texttt{even}(X)
\end{array}$$

has a unique answer set $S = \{\texttt{even}(\texttt{s}^n(0)), \text{for all even } n\}$. Therefore, its well founded model is $\langle S, \mathcal{L}\backslash S\rangle$ implying that for example $\neg\texttt{even}(\texttt{s}(0))$ is false, as well as $\texttt{even}(\texttt{s}(0))$. This problem can be solved adding the rule

$$\neg\texttt{even}(X) \leftarrow not\ \texttt{even}(X)$$

but this is the programmer's choice. We consider that the semantics should solve this kind of problems for *all* programs, instead of trusting the programmer to include one additional rule for each predicate symbol in such conditions. Thus, the formalization of the semantics avoids the problem from the beginning. ∎

In answer set semantics, the difference between "not" and "$\neg$" diminishes since it is a two-valued semantics. However, the different connectives are still there, and when applied to programs with "$\neg$" the semantics lack the necessary connection between them.

Even though there exists a close relationship between logic programming semantics and formalizations of nonmonotonic reasoning, in these systems the problem does not arise. Moreover, negation is represented by only one connective and is completely separated from the nonmonotonic rule of inference.

Example 2.4 We represent Example 2.3 in nonmonotonic formalisms. In default logic we need the following default rule D, along with theory $W = \{\texttt{even}(0)\}$,

$$\frac{: \neg\, \texttt{even}(x)}{\neg\texttt{even}(x) \wedge \texttt{even}(\texttt{s}(x))}$$

Then we can prove all intuitive positive and negative facts, like $\neg\texttt{even}(\texttt{s}(0))$ and $\texttt{even}(\texttt{s}(\texttt{s}(0)))$. Note that atoms, literals, and composed formulas can be obtained by default rules. In circumscription we can represent this problem by means of the theory $\{\texttt{even}(0), \forall x(\neg\texttt{even}(x) \supset \texttt{even}(\texttt{s}(x)))\}$ where predicate $\texttt{even}(\cdot)$ is minimized. Apart from the formal theory, circumscription needs to specify which circumscriptive policy is applied. However, the consequences are also formulas in the traditional logic language; there is no connective distinguishing between nonmonotonically and monotonically inferred formulas. Autoepistemic logic, as well as all nonmonotonic modal logics, has a belief modal connective "L". The formula "$not\ A$" in logic programming is associated with "$\neg \mathrm{L}A$". In these systems, the problem of distinguishing between "$\neg A$" and "$\neg \mathrm{L}A$" is also present. But here "$\mathrm{L}A$" has an intended semantics of "A is known" or "A is believed" which makes the distinction between "A" and "$\mathrm{L}A$". ∎

These examples show that negative and positive information can be inferred monotonically or not, and a clear distinction is made between the nonmonotonic semantics and the contingently chosen syntactic representation of a piece of information.

Consider for instance an atom A in a logic program under the well-founded semantics. The syntax allows the four alternatives A, $\neg A$, $not\ A$, $not\ \neg A$, and since the semantics is three valued, nine truth value assignments are possible, which are indeed too many cases for easy understanding. Notice the paradox that an inconsistent program, represented by both A and $\neg A$ being `true`, has $not\ A$ and $not\ \neg A$ as `false` formulas. In particular, if a programmer is trying to use a positive occurrence of the atom, he/she will need to decide which of A, $not\ \neg A$ or $A \wedge not\ \neg A$ is adequate.

Alferes and Pereira [2] suggested that all semantic formalization should satisfy the following *coherence principle*: " if $\neg A$ belongs to the semantics of a program then $not\ A$ must also belong to the same semantics". Obviously, the well-founded semantics does not observe this principle, and they proposed a variant that complies with it. Even though this principle improves the situation, both connectives are still present in this class of programs. We think this dual representation of negation is the original problem. The not is a syntactical device used to refer to the inference rule used to prove the literal. This procedural meaning was the reason for the difficulty in finding its declarative counterpart.

3 LOGIC PROGRAMS WITH DEFAULT POLICIES

Circumscription is a general term that designates a whole family of nonmonotonic reasoning formalisms which have the minimization of a semantic concept as a distinctive feature (see Lifschitz [17] for a review). Nonmonotonicity in these logics originates from the fact that minimization is always done when it is consistent to do so. Therefore, new facts may block the assumption of previously accepted conclusions. Each theory in these logics is specified by a collection of first order formulas

together with a particular *circumscriptive policy*, a set of rules defining the way in which minimization must be applied. A second order logic formula is constructed from these two elements, which determines the semantics of the theory.

In the logic programming language that is defined in this section we introduce, within logic programs, features of two special circumscriptive logics: pointwise circumscription [15] and circumscriptive theories [16]. In pointwise circumscription, minimization is taken to be at the level of ground atoms, rather than at the level of predicates or formulas as it is in the general case. Circumscriptive theories are characterized by the insertion of a circumscriptive policy within the same language of the theory. These two features are incorporated into the definition of *logic programs with default policies*.

Due to the fact that the syntax of these programs should include the circumscriptive policy, one or more second order predicates must be present in the alphabet.

Definition 3.1 Let $\sigma = \langle \mathcal{V}, Func, Pred, Pred2 \rangle$ be a signature (set of variable, function, first order predicate, and second order predicate symbols respectively). The set $\mathbf{Lit}(\sigma)$ of all its literals is called a *circumscriptive language* if $Pred2 = \{\texttt{def}\}$. A circumscriptive language will be noted by $\mathbf{Lit}_{CIRC}(\sigma)$.

Then a circumscriptive language is just an ordinary language for logic programs, except for the fact that it includes a special "second order" predicate $\texttt{def}(\cdot)$. This predicate will be used in the definition of minimization policies among atoms, which in turn will be interpreted by the semantics to implement default reasoning. The circumscriptive policy, *i.e.* the set of all atoms containing predicate $\texttt{def}(\cdot)$, may be considered the extension of a classic second order predicate. However, since there is no quantifier and the semantics will interpret variables as shorthand for subsumed rules, then the language is not as expressive as (and does not have the computational problems of) second order logic.

Logic programs with circumscriptive policies are then basic programs (monotonic programs with strong negation) using a circumscriptive language.

Definition 3.2 Let $\mathbf{Lit}_{CIRC}(\sigma)$ be a circumscriptive language. A *logic program with default policy* ($\mathbf{lpdp}$) is a set of basic rules in $\mathbf{Lit}_{CIRC}(\sigma)$.

Note that the syntax of these rules is the same as that of traditional rules without negation-as-failure

$$L_1 \leftarrow L_2, \ldots, L_n$$

The only requirement is that some of the literals might refer to predicate $\texttt{def}(\cdot)$. In the same way, the syntax of circumscriptive theories [16] is the same as that of first order logic.

Every basic program Π can be considered as an $\mathbf{lpdp}$ taking its signature σ without second order predicates, and extending it by including $\texttt{def}(\cdot)$ in a signature σ'. Therefore, when referring to a $\mathbf{lpdp}$ we assume the language and signature are those implicitly determined by the program.

The semantics of an $\mathbf{lpdp}$ will minimize those ground literals $\bar{L}$ such that $\texttt{def}(L)$ is a consequence in the program. Truth minimization implies that literal L will be assumed by default whenever it is consistent to do so. It will be the only nonmonotonic rule of inference in the program, and it can be applied to positive or negative literals.

For the formal characterization, we need to introduce the concept of the set of default literals for a program with respect to a given answer set. Recall that $\mathbf{Cn}(\Pi)$ is the set of (monotonic) consequences of Π, *i.e.* the minimal set closed both logically and under the rules of the program.

Definition 3.3 Let Π be an $\mathbf{lpdp}$, and C a set of literals in its language. The *set of default literals of* Π *with respect to* C is the set

$$\mathrm{Def}_{\Pi}(C) := \{L : \mathtt{def}(L) \in \mathbf{Cn}(\Pi) \wedge \bar{L} \notin C\} \tag{2}$$

C is called the *consistency basis* of $\mathrm{Def}_{\Pi}(C)$. The *minimization policy* of Π is the set of atoms in $\mathbf{Cn}(\Pi)$ with predicate $\mathtt{def}(\cdot)$.

This means that L is a default literal if $\mathtt{def}(L)$ belongs to $\mathbf{Cn}(\Pi)$, and $\bar{L}$ does not belong to C. Then, the set of default literals depends on the minimization policy of the program, and a given consistency basis. From the semantic point of view, this set is important because it contains all literals that are assumed to be true by default. Traditional rules determine monotonic inferences; circumscriptive policies and the consistency basis determine nonmonotonic ones.

We will now introduce the answer set semantics for a $\mathbf{lpdp}$. The answer sets definition for extended logic programs [18] constitutes an extension of the stable model semantics [9] definition only for negation as failure. The idea is that, for a set of literals (or only atoms in the original case) to be the consequences of a program, it is necessary to satisfy all negation as failure occurrences in it. However, every literal in the set must have some justification for being in it. In other words, the set should satisfy at the same time a *completeness* property, *i.e.* it should contain all the rules' consequences, and a *soundness* property, *i.e.* no other literal should be contained. The negation-as-failure characteristics make these conditions mutually dependent. Technically, this is solved by means of an equation: the sets that satisfy the equation are answer sets of the $\mathbf{lpdp}$, or stable models of the extended logic program.

Definition 3.4 Let Π an $\mathbf{lpdp}$, and S a set of literals in its language. S is called an *answer set of* Π if it satisfies

$$S = \mathbf{Cn}(\Pi \cup \mathrm{Def}_{\Pi \cup S}(S)) \tag{3}$$

Equation 3 assures that if S is an answer set then the default literals generated by S, with consistency basis S, have as consequence in Π the same set S. The answer set thus provides a certain "stability" to the program since it generates each of its members, and contains all of the generated elements. The difference with the equation for extended programs is that the extended literals are interpreted through the reduct of a program. On the other hand, in equation (3) default literals are generated outside the traditional rules in program Π, and then incorporated as facts in order to obtain the consequences of the program. The program reduct disappears as default literals are introduced.

In a similar manner, it is possible to introduce the well founded model semantics for $\mathbf{lpdp}$'s or to include rules in the style of Default Logic instead of the circumscriptive logic [8]. In the next section we present several examples for $\mathbf{lpdp}$'s, and we discuss some properties of this semantics.

4 PROPERTIES AND EXAMPLES

We will show an example in which positive and negative default conclusions are possible. This emphasizes the symmetry of our approach with respect to positive and negative data, in contrast to the standard attachment of a "negative" context to negation-as-failure.

Example 4.1 Suppose we know that every train T is passing through some stations, but we do not know in advance in which of these stations it stops. When the train is announced, information about the stops is given in the shortest possible way. If the train stops in many stations, only information for non-stopping stations is given. If the train stops in a small number of stations, only information for

stopping stations is given. The following program formalizes the complete reasoning about stopping and non-stopping stations for a train in this line, beginning with the information given in an announcement.

1. $\texttt{line\_station}(T, \texttt{lavis}) \leftarrow \texttt{train}(T)$
2. $\texttt{line\_station}(T, \texttt{mezzocorona}) \leftarrow \texttt{train}(T)$
3. $\texttt{line\_station}(T, \texttt{salorno}) \leftarrow \texttt{train}(T)$
4. $\texttt{line\_station}(T, \texttt{egna}) \leftarrow \texttt{train}(T)$
5. $\texttt{def}(\neg\texttt{stops}(T, X)) \leftarrow \texttt{line\_station}(T, X), \texttt{stops}(T, Y)$
6. $\texttt{def}(\texttt{stops}(T, X)) \leftarrow \texttt{line\_station}(T, X), \neg\texttt{stops}(T, Y)$

The first four rules define the stations in the line, and the last two formalize the default reasoning. For instance, rule 5 says that if we know that the train stops in some station, then it will be assumed by default that the train does not stop in others. Note that this default is not applied if the conclusion is contradictory with some already known fact; therefore, it is not necessary to include conditions such as $X \neq Y$ in the body of these rules. If the announcement is

$$\texttt{train}(\texttt{r123}), \texttt{stops}(\texttt{r123}, \texttt{mezzocorona})$$

then there is only one answer set of the above program together with these facts which contains the following extension of the $\texttt{stop}(\cdot, \cdot)$ predicate

$$\{\texttt{stop}(\texttt{r123}, \texttt{mezzocorona}), \neg\texttt{stop}(\texttt{r123}, \texttt{lavis}), \\ \neg\texttt{stop}(\texttt{r123}, \texttt{salorno}), \neg\texttt{stops}(\texttt{r123}, \texttt{egna})\}$$

For a new announcement

$$\texttt{train}(\texttt{r124}), \neg\texttt{stops}(\texttt{r123}, \texttt{lavis})$$

then the answer set contains

$$\{\texttt{stop}(\texttt{r124}, \texttt{mezzocorona}), \neg\texttt{stop}(\texttt{r124}, \texttt{lavis}), \\ \texttt{stop}(\texttt{r124}, \texttt{salorno}), \texttt{stops}(\texttt{r124}, \texttt{egna})\}$$

In case we only know that $\texttt{train}(\texttt{r125})$, then no positive nor negative information about stops is inferred from the program. ∎

The same kind of problems represented in extended logic programs need several combinations of "not" and "$\neg$" in the bodies of the rules. This fact affects the objectives mentioned in Section 2, and obviously make the understanding of such programs more difficult.

In order to formalize this reasoning, let S be an answer set for an $\mathbf{lpdp}$ P. Let us call a literal L *supported* in S for Π iff there is some rule in Π with L in the head, and a body included in S; a literal L is a *default* literal in S for Π if $\texttt{def}(L)$ is supported, and $\bar{L} \notin S$. Intuitively, a supported literal can exhibit some justification to be included in the semantics. Then the following result can be proved.

Proposition 4.2 *Let* Π *be an* $\mathbf{lpdp}$ *with a consistent answer set* S*. Then, for every literal* $L \in S$ *either* L *or* $\texttt{def}(L)$ *is supported in* S *for* Π

It is not possible for a similar property to hold for extended logic programs. There is no rule in an extended program for justifying extended literals of the form $not\, L$. Therefore, not only are these literals unsupported, but the rules that use these literals in their body cannot justify their conclusions. In fact, these definitions can be used to characterize answer sets for $\mathbf{lpdp}$'s.

Proposition 4.3 *Let* Π *be an* $\mathbf{lpdp}$ *with a consistent answer set* S*. Then,* $S = \{L : L$ *is supported or* L *is default in* S *for* $\Pi\}$

Naturally, several other properties of the answer set semantics for extended logic programs also hold for answer sets in **lpdp**'s. For example, there are **lpdp**'s with no, one, and several answer sets.

In view of the diversity of semantics for negation-as-failure in logic programs, Dix [6, 4] proposed a method for classifying and characterizing them. Inspired in similar work in nonmonotonic reasoning [14, 21], he introduced a nonmonotonic entailment relation for normal logic programs and studied its properties under each of the semantics. These principles are classified into two types: *strong properties* are adaptations of those from nonmonotonic reasoning and belief dynamics, such as cumulativity, rationality, cut, and cautious monotony; *weak properties* reflect the specific idea of negation-as-failure in logic programming.

The answer set semantics for extended programs does not satisfy most strong properties like cumulativity or rationality, and the same holds for the given semantics for **lpdp**'s. In fact, this allows us to give the name of "answer set" to this semantics. However, there are some weak properties like *reduction* that do not hold for **lpdp**'s even if they are satisfied in extended programs. Reduction removes those literals that are facts in the program from the bodies of all other rules. The idea is to interpret those literals as `true`, and simplifying the rules that refer to them. In the original version [4] for normal logic programs, reduction has two effects: one for positive literals, and the other for negative literals. Positive literals are considered as true with the explained meaning, but negative literals in these programs are interpreted with negation-as-failure semantics. So the effect of reducing a program by a negative literal is to consider the complementary literal as false, and eliminating from the program all rules that have the complement in the head. This is no longer adequate for **lpdp**'s, because negative literals can also be in the head of basic rules, and non-monotonic inference can also be applied to positive literals. In short, we can say **lpdp**'s do not satisfy reduction because negation is no longer the result of only nonmonotonic inferences.

5 COMPARISON WITH RELATIONED WORK

As it was mentioned in the introduction, negation-as-failure preceded explicit negation in its incorporation into a logic programming framework. The first proposals that included the possibility to express both types of negation [11, 10, 13] did so under the semantics of the stable models. As it was shown in several examples, this semantics has the property of preferring those models that are "as much two-valued as possible". In other words, it maximizes the truth value of literals, according to the ordering $\leq_k$. Therefore, the non orthogonal problems are reduced, since it is often the case that when $\neg L$ belongs to the semantics so does $not\ L$.

On the other hand, the well founded model semantics only assigns a truth value of true or false to a literal if it is safe to do so in every situation. The fact that a default literal is consistent with the set of consequences is not enough justification to include it. When explicit negation was added to this framework [7, 24], the lack of semantic connection between $\neg L$ and $not\ L$ was worsened. Literals can be negated in one way or the other, but both negations are completely independent, as it is shown in the following example.

Example 5.1 Let Π be the following extended program

1. $\mathtt{a} \leftarrow not\ \mathtt{b}$
2. $\mathtt{b} \leftarrow not\ \mathtt{a}$
3. $\neg\mathtt{a} \leftarrow$

The well founded model of Π is the set $\{\neg\mathtt{a}, not\ \neg\mathtt{b}\}$, where `a` and `b` have undefined truth value. In spite of the fact that $\neg\mathtt{a}$ is true, $not\ \mathtt{a}$ is undefined and this does not allow the application of rule 2. In this case strong negation is not "strong" enough to generate weak negation as failure.

Note that Π has only one stable model, $S_1 = \{\neg\mathtt{a}, \mathtt{b}\}$, so both credulous and sceptic semantics coincide. In this case the relation between $\neg\mathtt{a}$ and $not\ \mathtt{a}$ is forced by the inconsistent assumption. In

this way, however, the semantics assigns a preference of rule 2 over rule 1. ∎

In view of these facts, Alferes and Pereira [1, 2] presented a new version of the well founded model semantics, called WFSX, such that the coherence principle (see section 2) is enforced. In order to simplify the presentation, we will not discuss here the precise definitions for the construction of the model according to WFSX. It is based on the traditional well founded model ($\gamma_{\Pi}(\cdot)$ operator); then, those literals that are necessary for the satisfaction of the coherence principle are added, a new closure is calculate, and a fixpoint of the resulting operator is obtained. The following program shows the result of this semantics.

Example 5.2 Let Π be the extended program from example 5.1. Then the set

$$\{\neg \mathtt{a}, \mathtt{b}, not\ \mathtt{a}, not\ \neg \mathtt{b}\}$$

is the set of consequences in the WFSX semantics. ∎

The WFSX semantics, as it is shown in the previous example, considers "not" as a connective in the language. Therefore it has an extensional meaning, at the same level as "$\neg$". Besides, WFSX has two fundamental characteristics that distinguish it from most of the other semantics for programs with two types of negation.

- Symmetry in the treatment of positive and negative literals. The fact that a literal is positive or negative doesn't have an impact on the set of consequences of a semantics, and therefore on the truth value assigned. Many times the sign of a literal depends on the election of the name for the predicate, such as for example $\mathtt{guilty}(X)$ or $\neg\mathtt{innocent}(X)$. In consequence, the programmers will to use one or another version makes a difference in most semantics [13]. The WFSX semantics amends these characteristics if we do not distinguish both types of negations.

- Existence of a semantic connection between the two negative connectives. It solves the problems of Section 2 concerning the negative relationship. Thus it allows to combine both connectives, and the use of negative connectives can be simplified.

The WFSX semantics provides a solution to the problem of semantic connection between negative connectives. However, this formalism doesn't solve other problems, like the non-homogeneous syntax and the poor adaptability to changes in the program. We consider all these problems related by the same cause: the bond of non-monotonic inference and a negative connective. The WFSX semantics does not break this bond. Instead, it attacks one of its effects.

The relationship between the WFSX semantics and **lpdp**'s is that both have a similar motivation. Alferes and Pereira's proposal redefines the well founded process to include those consequences that were intuitively missing. On the other hand, **lpdp** tries to maintain the well known inferences of answer sets and well founded models, but makes a complete revision of the representation of the information and the inference rules in a program. The idea behind it is that both the answer set and the well founded model semantics are sufficiently expressive mechanisms to represent most of the practical problems in which you expect to apply a logic programming system. Therefore, it was preferred to change the programming style to separate negation from nonmonotonic inference. This decision not only permits to solve the problem of the semantic connection, but it also establishes a more elegant syntax and allows the resulting logic programs to have more extendibility and composition properties, as it was shown in the previous section.

In **nmlp**'s the use of non-monotonic inference is protected by the language: once it is applied, no clues are left to indicate its application. The representation of information is transparent to the inference procedure. Since the underlying programs in the WFSX semantics are still extended programs,

every reference to "not" has the modal flavor that makes it incompatible with negation. Further references to explicit negation might not be in concordance with this style. When negation is independent of the inference rules, combinations, extensions, and restrictions of the original program can be more declaratively specified.

Other related works with the proposed framework are Wagner's proposal for two negations in [26, 25] and Poole's Theorist system [**?**]. Wagner's work is in the same line as the WFSX semantics. It presents a semantics of logical programs with two negative connectives, "$\sim$" and "$-$" which are semantically related. However, both semantic characterizations differ from negation-as-failure. Besides, the connection is carried out at the level of predicates, *e.g.*, all the atoms that contain a predicate `father` are such that the negation $\sim$ `father`(X, Y) implies $-$`padre`(X, Y), a kind of selective coherence principle in the sense of WFSX. For **lpdp**'s, the only negation connective can be inferred by monotonic or non-monotonic inference rules, and this fact is not carried out in the representation. In these programs, it is also possible to make nonmonotonic inference only for certain ground atoms (`def`(`father`(`charles`, `susan`)), instead of applying it to the whole extension of the predicate.

On the other hand, Poole proposes in Theorist a practical implementation of a diagnosis system based on the PROLOG language. It uses a syntax of logic programming such that it incorporates a kind of default rule as an additional type of inference rule. The syntax of these rules is similar that of default rules, but the semantics is different. The nonmonotonic inference is carried out by means of abduction, that is, trying to generate explanations for a given atom. There is no connection in the system among the abducible literals and the possible references to explicit negation.

6 CONCLUSIONS

In this paper we presented some problems in the integration of negation-as-failure and strong negation into logic programs. Their origin is the fact that negation-as-failure is both a negative symbol and a nonmonotonic inference rule. We proposed an approach where negation-as-failure is not present in the language, and nonmonotonicity is introduced in the form of default policies. Then, we showed that it is possible to recast the answer set semantics and recover similar properties that hold for extended programs. Thus, we showed in section 4 that our approach has the same expressivity as answer set semantics for extended programs, with the advantage that the described shortcomings do not show up making the language more declarative and amenable to interoperability. Another advantage of this embedding is that existing answer set provers, like DLV or SMODELS, can be used to compute the semantics of **nmlp**. In this sense, a DLV front-end processor is being developed.

We presented the semantics of **nmlp** based on an extension of answer set semantics, but this is not essential. Well founded model semantics can also be used to characterize a weaker **nmlp** semantics, as shown in [8]. We consider other logic programming extensions, like action languages, constraint programming, are also possible in this framework.

We conclude that negation-as-failure is not necessary for the semantics of logic programming, or answer set programming. Furthermore, since it also makes the understanding of extended programs more difficult because of the presence of two negation symbols without semantic connection, it is preferable to remove it. In this way, we are encouraging simplicity while maintaining the expressiveness of the language.

REFERENCES

[1] José Júlio Alferes and Luís Moniz Pereira. On logic program semantics with two kinds of negation. In Krzysztof R. Apt, editor, *Proceedings of the 1992 Joint International Conference*

and Symposium on Logic Programming, pages 574–588, Washington, DC, 1992. MIT Press.

[2] José Júlio Alferes and Luís Moniz Pereira. *Reasoning with logic programming*, volume 1111 of *Lecture Notes in Computer Science*. Springer Verlag, 1996.

[3] Chitta Baral and Michael Gelfond. Logic programming and knowledge representation. *Journal of Logic Programming*, 19:73–148, 1994.

[4] Gerhard Brewka, Jürgen Dix, and Kurt Konolige. *Nonmonotonic reasoning: an overview*. Number 73 in Lecture Notes. CSLI Publications, 1997.

[5] Edsger W. Dijkstra. GOTO statement considered harmful. *Communications of the ACM*, 11(3):147–148, 1968.

[6] Jürgen Dix. Semantics of logic programs: their intuitions and formal properties. In André Fuhrmann and Hans Rott, editors, *Logic, action and information*, pages 227–313. de Gruyter, Berlín–New York, 1994.

[7] Phan Minh Dung and Phaiboon Ruamviboonsuk. Well-founded reasoning with classical negation. In Anil Nerode, V. Wiktor Marek, and V. S. Subrahmanian, editors, *Proceedings of the 1st. International Workshop on Logic Programming and Nonmonotonic Reasoning*, pages 120–132, Washington, DC, 1991. MIT Press.

[8] Pablo R. Fillottrani. Sobre la negación y la inferencia no monótona en la programación en lógica. In *Proceedings CACIC'00, Sexto Congreso Argentino de Ciencias de la Computación*, pages 489–500, 2000.

[9] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In Robert A. Kowalski and Kenneth A. Bowen, editors, *Proceedings of the 5th. International Logic Programming Conference*, Seattle, Washington, 1988. MIT Press.

[10] Michael Gelfond and Vladimir Lifschitz. Logic programs with classical negation. In David H. D. Warren and Peter Szeredi, editors, *Proceedings of the 7th. International Logic Programming Conference*, pages 579–597, Jerusalem, Israel, 1990. MIT Press.

[11] Michael Gelfond and Vladimir Lifschitz. Classical negation in logic programming and disjunctive databases. *New Generation Computing*, 9:365–385, 1991.

[12] Robert A. Kowalski. Predicate logic as a programming language. In Jack L. Rosenfeld, editor, *Proceedings of the IFIP Congress 74*, pages 569–574, Stockholm, Sweden, 1974. North Holland.

[13] Robert A. Kowalski and Fariba Sadri. Logic programs with exceptions. In David H. D. Warren and Peter Szeredi, editors, *Proceedings of the 7th. International Logic Programming Conference*, pages 398–613, Jerusalem, Israel, 1990. MIT Press.

[14] Sarit Kraus, Daniel Lehmann, and Menachem Magidor. Nonmonotonic reasoning, preferential models and cumulative logics. *Artificial Intelligence*, 44:167–207, 1990.

[15] Vladimir Lifschitz. Pointwise circumscription. In Matthew L. Ginsberg, editor, *Readings in nonmonotonic reasoning*, pages 179–193. Morgan Kaufmann Publishers, 1987.

[16] Vladimir Lifschitz. Circumscriptive theories: a logic-based framework for knowledge representation. In Richmond H. Thomason, editor, *Philosophical logic and artificial intelligence*, pages 109–159. Kluwer Academic Publishers, 1989.

[17] Vladimir Lifschitz. Circumscription. In Dov M. Gabbay, C. J. Hogger, and J. A. Robinson, editors, *Handbook of logic in artificial intelligence and logic programming*, volume 3, pages 297–352. Oxford University Press, 1994.

[18] Vladimir Lifschitz. Foundations of logic programs. In Gerhard Brewka, editor, *Principles of knowledge representation*, Studies in Logic, Language and Information, pages 69–127. Cambridge University Press, 1996.

[19] Vladimir Lifschitz, Lappoon R. Tang, and Hudson Turner. Nested expressions in logic programs. *Annals of Mathematics and Artificial Intelligence*, 25:369–389, 1999.

[20] John W. Lloyd. Practical advantages of declarative programming. In *Proceedings of the 1994 Joint Conference on Declarative Programming, GULP-PRODE'94*. Springer Verlag, 1994.

[21] David Makinson. General patterns in nonmonotonic reasoning. In Dov M. Gabbay, C. J. Hogger, and J. A. Robinson, editors, *Handbook of logic in artificial intelligence and logic programming*, volume 3, pages 35–110. Oxford University Press, 1994.

[22] Robert C. Moore. Semantical considerations on nonmonotonic logic. *Artificial Intelligence*, 25:75–94, 1985.

[23] David Pearce. Reasoning with negative information II: hard negation, strong negation and logic programs. In David Pearce and Heinrich Wansing, editors, *Nonclassical logics and information processing*, number 619 in Lecture Notes in Computer Science, pages 63–79. Springer Verlag, 1992.

[24] Teodor C. Przymusinski. Extended stable semantics for normal and disjunctive logic programs. In David H. D. Warren and Peter Szeredi, editors, *Proc. of the 7th. International Logic Programming Conference*, pages 459–477, Jerusalem, Israel, 1990. MIT Press.

[25] Gerd Wagner. Vivid reasoning with negative information. In Wiebe van der Hoek, J.-J. Ch. Meyer, Y. H. Tan, and Cees Witteveen, editors, *Nonmonotonic reasoning and partial semantics*, chapter 7, pages 181–205. Ellis Horwood, 1992.

[26] Gerd Wagner. *Vivid logic: knowledge based reasoning with two kinds of negation*. Number 764 in Lecture Notes in Computer Science. Springer Verlag, 1994.