Negation in Logic Programs: Is It Necessary Two Connectives?

Pablo R. Fillottrani Guillermo R. Simari

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

e-mail: ccfillo@criba.edu.ar

Abstract

Negative information can be represented in several classes of logic programs. These approaches were first motivated by the search for an appropriate declarative semantics for negation as failure. Recently, some formalizations of monotonic negation were also introduced in logic programming in the form of "classical" or "strong" negation. In view of such a diversity of semantics for negation in logic programs, Dix [7, 5, 6] proposed a method for classifying and characterizing them.

In this paper we present an analysis of these approaches from the viewpoint of knowledge representation. We compare them with nonmonotonic formalisms such as default logic, circumscription, and autoepistemic logic, identifying some problems that are not present in these logics. Then we suggest some alternatives, considering Dix's formal properties. Finally we discuss the effect of allowing only one negative connective in the syntax of logic programs, whose semantics is defined by the programmer.

Keywords: `logic programming, nonmonotonic reasoning, negation, negation as failure`

Negation in Logic Programs: Is It Necessary Two Connectives?

1 Introduction

Negation in logic programs dates back to the introduction of the *negation as failure* *not* operator by the first Prolog interpreters [4]. Its clear procedural semantics: "*not* A is provable if and only if A is not provable" make it a useful and popular programming tool for most applications. Unfortunately this characterization depends on the way each interpreter proves A. Since SLD-resolution is parameterized by the selection rule and the computation rule [20], each implementation results in a different meaning to *not*. A logic program with *not* may be correct in one interpreter and have errors in other.

In view that the goal of logic programming is to be a declarative programming language, the strong dependence of the meaning of *not* on particular procedures was unacceptable. This problem motivated in late eighties and early nineties an active research work to develop a declarative characterization for *not*. Its outcome was a whole family of formal definitions trying to capture the essence of negation as failure into mathematical devices. Several survey articles [2, 25, 7] describe the results in this field. Dix in [5, 6] propose some abstract properties in a general framework so as to compare these proposals. The answer sets semantics [11, 12] and the well-founded model semantics [26] are generally accepted as the most adequate formalization of this type of negation.

Even though answer sets and well-founded model provide acceptable solutions to the presented problem, none of them is accurate for all application domains. Moreover, it has been shown that general negation as failure is not always the right semantics for representing negative information. Sometimes negative literals need to be explicitly deduced from monotonic rules, just like positive literals. Sometimes other specific inference rule is needed. As a consequence several other types of negation have been formalized in logic programs, and classes of programs including two o more negative connectives have been defined [13, 15, 23, 29].

All these semantical proposals have in common the property of being precisely characterized, but this is not enough for being declarative. Declarativeness of a formal device also requires other conditions, such as understandability and composability, that are mostly subjective and then not easy captured by mathematical definitions. The objective of this paper is to analyze these different formalizations of negative information in logic programming from the point of view of declarative knowledge representation. We propose a general method to unify several semantics definitions under the same global label of "negative knowledge" with no prerequisites of formal properties, and argue it improves the compliance to the declarative criterion. We first present in section 2 the most common definitions of negation in logic programming. Then we consider in section 3 their relation to the syntax and inference rules. We present the declarative criterion, and evaluate the presented

semantics in section 4. Even they have a close relationship with nonmonotonic formalisms like circumscription [18] and default logic [24] and these formalisms fulfill the declarative criterion, logic programming semantics do not quite follow it. Finally in section 5 we propose a syntactical way of unifying several formalizations of negative information in logic programs, and argue it is more declarative than having separate connectives for each semantics.

2 Negation in Logic Programming

We will briefly introduce in this section the semantics for negation in logic programming that are most commonly used. Answer sets and well-founded model are two formalizations for negation as failure; strong negation is an additional connective for explicitly derived negative information. We will follow the notation presented in [19]

In logic programming the language describes the set of statements that can be true or false. We assume the traditional definitions of terms and atoms in first order logics.

Definition 2.1 Let us consider an alphabet formed with finite sets of predicate and function symbols, where propositions and constants are 0-ary predicates and functions respectively. A *normal language* $\mathcal{L}$ is the set of ground atoms in this alphabet.

Each member of a normal language is supposed to have a truth value assigned by the program. Note that there are no connectives in the language, just only atoms. Therefore, every interpretation of the language is consistent.

A logic program is a set of rules with the objective of specifying the truth value of all the elements in a language. Even though these rules were historically associated with well-formed formula in first order logic [20], they should be better considered as inference rules [27]. Since they do not belong to the language, no truth value is assigned to them.

Definition 2.2 If $\mathcal{L}$ is a normal language, then a *normal program* in $\mathcal{L}$ is a set of rules

$$A \rightarrow BPos \cup not \;\; BNeg \tag{1}$$

such that $BPos$ and $BNeg$ are finite subsets of $\mathcal{L}$, and $A \in \mathcal{L}$. Whenever a normal program is specified without referring to the language, we will take that formed with the alphabet consisting of the symbols appearing in the program. In a rule of the form 1 A is called the *head*, and $BPos \cup \; /BNeg$ the *body* of the rule. If $BPos = BNeg = \emptyset$, *i.e.* the rule has no atom in its body, we will call it a *fact*. If Π is a normal program such that $BNeg = \emptyset$ for all its rules, then we will call Π a *basic normal program*.

The informal meaning of the program can be described as follows: if an atom A occurs in the head of a rule whose body is satisfied then A will be assigned the truth value `true`; conversely if all rules containing A in the head are such that their body is violated, then A will be assigned the truth value `false`. Each particular formalization of this intuitive meaning solves the problem of determining when the body of the rule is "violated".

Basic normal programs, *i.e.* those where all negative bodies are empty, have a clear and uncontroversial semantics. In this case it is equivalent to consider a rule as an inference rule or as propositional well formed formula. We will first introduce this semantics, and then extend it to the class of all normal programs.

Definition 2.3 Let Π be a basic normal program, and $S \subseteq \mathcal{L}$. S is said to be *closed under* Π if for all rules $A \leftarrow BPos$ such that $BPos \subseteq S$, then $A \in S$ holds. The *set of consequences of* Π is the smallest set of literals closed under Π, and it is noted $Cn(\Pi)$.

It can be proved that the set $Cn(\Pi)$ is well defined, *i.e.* it always exists and it is unique. Taking $Cn(\Pi)$ as the semantics of a basic normal program Π results in the same meaning as if Π is considered as a set of propositional formulas in classical logic.

Unfortunately the clear semantics of basic normal programs cannot be extended to all normal program. As previously mentioned, the presence of negation as failure in the body of a rule originated several formalizations. We present now the concept of answer set [11], also called stable model for programs without strong negation. Answer sets are characterized by first defining a transformation of a normal program into a basic normal program, and then applying $Cn(\cdot)$ (definition 2.3) to the transformed program.

Definition 2.4 Let Π be a normal program and $S \subseteq \mathcal{L}$. The *reduct of* Π *relative to* S is the basic program obtained from Π by

- deleting each rule $A_0 \leftarrow BPos \cup not\, BNeg$ such that $BNeg \cap S \neq \emptyset$.
- replacing each remaining rule $A_0 \leftarrow BPos \cup not\, BNeg$ by $A_0 \leftarrow BPos$.

This program will be noted by Π^S.

Intuitively the reduct or a normal program Π relative to a set S is the result of interpreting as `true` all atoms in S, and discharging or modifying the rules in Π according to this interpretation. To be an answer set, S should not be arbitrary taken. It must be justified by the same transformation.

Definition 2.5 Let Π be a normal program, and $S \subseteq \mathcal{L}$. Then S is an *answer set of* Π if it satisfies the equation $S = Cn(\Pi^S)$.

Since its formalization is an equation, a fixed normal program may have zero, one or more answer sets. The answer sets semantics considers the consequences of a

normal program the intersection of all answer sets of the program. It may be a very strong condition when the program has several answer sets, and it may be a weak condition if it has none.

The well founded semantics [26] tries to overcome the previous difficulty by requiring a three-valued model. In case the program does not clearly define the meaning of an atom, this atom is assigned the truth value `undefined` instead of being forced to `true` or `false` as in an answer set.

Definition 2.6 Let Π be a normal program. The function $\gamma_\Pi : 2^{\mathcal{L}} \longrightarrow 2^{\mathcal{L}}$ is defined for every $S \subseteq \mathcal{L}$ as

$$\gamma_\Pi(S) = Cn(\Pi^S)$$

The *well-founded model of* Π is the three-valued model $\langle \text{FIX} \uparrow \gamma^2_\Pi, \text{FIX} \downarrow \gamma^2_\Pi \rangle$. This model is noted $\mathcal{M}_\Pi$.

It can be proved that the interpretation introduced in definition 2.6 is well defined, and that it is indeed a three-valued model of Π. The well founded is thus defined for all normal programs, in contrast with answer sets. Nevertheless, it is weak for some cases. Applications like satisfiability planning where the truth value of an atom in a fixed model is needed, may require the answer set semantics.

Negation as failure is rather a way to interpret an atom within the proof of other, than a way to prove the negation of the atom. No negative information can be deduced from normal programs. Indeed its consequences under the answer set and the well founded semantics, are both sets of atoms. The view of negation as failure as a form of proving negative information is discussed in the following section. Emerging from some applications, an explicit form of negation was introduced in logic programs [13, 15]. This "strong negation" is completely orthogonal with negation as failure, as it is shown by the fact that all previous semantics definition can be extended only be a change in the language.

Definition 2.7 Let us consider an alphabet formed with finite sets of predicate and function symbols together with the unary connective $\neg$. A *language* $\mathcal{L}$ is the set of ground literals in this alphabet, *i.e.* the set of all A and $\neg A$ where A is a ground atom. If A is a ground atom in this alphabet, then A and $\neg A$ are called *complimentary literals*.

Normal languages only contain atoms; languages contain atoms and their negations. We introduce new names to logic programs with two types of negation.

Definition 2.8 If $\mathcal{L}$ is a language, then a *program* in $\mathcal{L}$ is a set of rules

$$L \leftarrow BPos \cup not\ \ BNeg \tag{2}$$

such that $BPos$ and $BNeg$ are finite subsets of $\mathcal{L}$, and $L \in \mathcal{L}$. If $BNeg = \emptyset$ then the program is called a *basic program*.

Note that this definition is a translation of definition 2.2 with the new type of language.

The meaning of programs is defined in a similar way to that of normal programs, both answer sets and well founded models. Semantical definitions 2.4, 2.5 and 2.6 are translated literally with programs instead of normal programs. The only necessary change is to introduce a consistency condition to the consequences of a basic program in definition 2.2.

Definition 2.9 Let $S \subseteq \mathcal{L}$. S is *consistent* if it does not contain a pair of complimentary literals A and $\neg A$. S is *logically closed* if it is consistent or $S = \mathcal{L}$. Let Π be a program. The *set of consequences of* Π is the smallest set of literals closed under Π and logically closed, noted $Cn(\Pi)$.

If a set of elements of the language is inconsistent, then it cannot be considered as the consequences of a program unless it is the whole language. This primitive consistency condition is replaced by other conditions in [28, 1].

3 Inference Rules and Connectives

We will analyze in this section the properties of both negation as failure and strong negation, considering logic programming as a formal theory [5, 6]. A connective is a symbol that from one or more elements of the language constructs a new one. Therefore each member of a language, often called formulæ, is either an atom or may be split into smaller components. This syntactical view does not require a truth functional definition of the meaning of the new formulæfrom the truth value of its components. Nevertheless traditional connectives in propositional logics such as $\vee$, $\wedge$, $\supset$, $\neg$ and $\equiv$ do have clear truth tables. If a formal theory satisfies this property, then it is possible to replace in a formula one of its components by any other formula with the same truth value.

In logic programming,it is clear from definition 2.1 that normal languages do not contain connectives and composed formula. The symbol $\rightarrow$ is not treated as a connective since it is not possible to consider rules as formula in the language. The reason for this is twofold: logic programming rules have no reasonable truth value assignment, and cannot be nested (see although [22] for other types of implications). Rules are better treated as inference rules that allow to prove a formula from the proof of others. A fixed normal logic program is therefore a particular formal theory whose axioms are the atoms appearing in the heads of the facts.

Negation as failure *not* also cannot be nested: *not not A* does not have declarative semantics and does not belong to the language. In spite of this fact, it could be regarded as a kind of connective since *not A* has a clear semantics. For instance if *not A* means "*A* is not known" or "*A* is not believed". the general usage of this connective has no connection with negative information.

Example 3.1 This example appeared in [12]. A college is awarding scholarships to its students. Examination scores are represented by predicates `highGPA`$(\cdot)$ and `fairGPA`$(\cdot)$, and `eligible`$(\cdot)$ represents those elected.

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

The last rule in the program shows a body where an atom appears both negatively and positively. If indeed *not* represents negation, then the rule is meaningless. The real semantics of these occurrences of *not* is "not known", like in modal logics. Negation may or may not be related to the absence of knowledge of the complimentary literal, but this a point to be solved by the supplier of the information and not by the user. □

In fact, it is desirable to use a symmetric representation of negative and positive information. In this way there is no difference whether the programmer chooses positive or negative names for predicates. For example, he/she can represent the information with predicates `normal(·)` or `abnormal(·)`, `internal(·)` or `external(·)` without restricting future references to atoms in these predicates. Clearly, the *not* connective does not satisfy this property.

The complexity of the problem grows when considering languages with explicit negation $\neg$. Although not represented in definition 2.7 for simplicity, it is easy to allow nested formula with $\neg$ and it also has comprehensible semantics. But there is no connection between $\neg$ and *not*, as it is shown in the following examples. In example 3.2 presents an atom A such that in the well-founded semantics $\neg A$ is `true` and *not* A is `false`. Example 3.3 shows the opposite situation.

Example 3.2 The following program has no answer set, and its well-founded model is $\langle\{\neg\texttt{p}\}, \emptyset\rangle$.

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

The well-founded semantics accepts $\neg$p and sets *not* p as undefined. Technically, the set of consequences under the answer set semantics is the whole language. However the previous problem arises: "does *not* p belong to the language?" □

Example 3.3 Suppose the language contains the constant 0 and the unary function symbol `s`$(\cdot)$ to represent all natural numbers.

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

This program 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 `even`(`s`(0))!. This problem can be solved adding the clause

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

but this is a choice of the logic programmer. We consider the semantics should solve this kind of problem for *all* programs instead of trusting the programmer to include one additional rule for each predicate symbol in such condition. The semantic formalization should solve this problem from the beginning. □

In answer set semantics the difference between *not* and ¬ vanishes since it is two-valued. The problem is that this formalization has other undesirable properties, see for example [7], that are not present in well-founded semantics. But then, the well-founded semantics in programs with ¬ lacks the necessary connection between both negative connectives.

Even though there exists a close relationship between logic programming semantics and formalizations of nonmonotonic reasoning, these systems do not introduce this shortcoming. Moreover, negation is representing by only one connective and is completely separated from the nonmonotonic semantics or rule of inference.

Example 3.4 We represent example 3.3 in default logic, circumscription and autoepistemic logic. In default logic we need the following default rule D, together with theory $W = \{\texttt{even}(0)\}$:

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

This default rule establishes that if it is consistent to assume that a number is not even, then it is not even but its immediate successor is. Together with the fact in W is enough to 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 formula can be obtained by default rules.

In circumscription we can represent this problem by the theory

$$\{\texttt{even}(0), \forall x(\texttt{even}(x) \supset \texttt{even}(\texttt{s}(x)))\}$$

where predicate $\texttt{even}(\cdot)$ is minimized. Besides the formal theory, circumscription needs to specify which circumscriptive policy is applied. But the consequences are also formula in the traditional logic language; there is no connective distinguishing between nonmonotonically and monotonically inferred formula.

Autoepistemic logic, as well as all nonmonotonic modal logic, has a belief modal connective L. *not* A in logic programming is associated with $\neg \mathrm{L}A$. In these system 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" that makes the difference between A and $\mathrm{L}A$. □

These examples show that negative and positive information can be inferred monotonically or not. A clear distinction is made between the nonmonotonic semantics and the contingent syntactic representation of a piece of information. We argue the need for declarative knowledge representation of exhibiting this separation in next section.

4 Declarativeness of Logic Programs

Informally a declarative programming language is one that specify *what* is computed, instead of *how* it is done. Baral and Gelfond [3] attribute to McCarthy being 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.

Then the point is to establish what properties a declarative language must satisfy, *i.e.* where is the line between logic and control in Kowalski's [14] famous equation

$$algorithm = logic + control$$

John Lloyd [21] 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 logic systems satisfy as long as they have a proof theory. But in order to the theory to be modular, *i.e.* to be composed with other theories possible written by other programmers, the logic should have widely comprehensible semantics.

One way to achieve this goal is to define a language similar to natural language whose semantics follows common sense reasoning. Logic programming falls short from this point since its syntax is simpler than full first order logic, and programs are sets of rules with definite semantics. The efficiency of its proof methods is another advantage. Even though, this is not enough. Negation is necessary, but its semantics is controversial. We think that having two separate negative connectives is a false step towards a declarative programming language, because the type of negation must be decided each time negative information is used. The asymmetry between positive and negative information makes decisive the supplier's choice of names for predicates, and the user is not allowed to freely combine the connectives. Moreover, the difference between both types of negation is not reflected in natural language unless negation as failure is understood as a modal operator of knowledge or belief. If this is the case it should not be called "negation" at all.

Consider for instance an atom A in a program under the well-founded semantics. The syntax allows the four alternatives A, $\neg A$, $not\, A$, $not\, \neg A$, so nine truth value assignments are possible considering the first two alternatives as three-valued. They are indeed too much cases for an easy understanding. The table in figure 1 shows all these cases. Notice the paradox that an inconsistent program, represented by the first line, has some `false` formula. For example 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.

A	$\neg A$	$not\ A$	$not\ \neg A$
true	true	false	false
true	undefined	false	undefined
true	false	false	true
undefined	true	undefined	false
undefined	undefined	undefined	undefined
undefined	false	undefined	true
false	true	true	false
false	undefined	true	undefined
false	false	true	true

Figure 1: Possible truth value assignments in programs with two negative connectives under the well-founded semantics.

Alferes and Pereira suggested in [1] that all semantic formalization should satisfy the following *coherence principle*

> *if* $\neg A$ *belongs to the semantics of a program* Π *then* $not\ A$ *must also belong to the semantics of* Π

Obviously the well-founded semantics does not observe this principle, and they proposed a variant that complies with it. Even though this principle simplifies the situation, both connectives are still present in this class of programs. We think this dual representation of negation is the original problem. *not* is a syntactical device to refer to the inference rule used to prove the literal. This procedural meaning was the reason of the difficulty in finding its declarative counterpart. We propose in next section a class of logic programs with a clear distinction between the nonmonotonic inference rule "assume A in the absence of a proof for its complimentary" and the negative connective $\neg$.

5 Circumscriptive Logic Programming

In section 3 we showed that nonmonotonic formalisms, unlike logic programming, represent negation with one connective unrelated with their nonmonotonic semantic devices. Next we propose a similar syntactic treatment for negation in logic programming, arguing it is more declarative than traditional *not* and $\neg$ connectives. No semantic requirements are present for this negation, allowing even varying meaning for each ground literal. The supplier of the negative information specifies this meaning, whilst the user only knows its negative nature but not how it is proved.

In order to present this system we will apply two techniques originally proposed for circumscription: circumscriptive theories [17] and pointwise circumscription [16]. Circumscriptive theories introduce circumscriptive policy as formula in the language, instead of being a metatheoretical specification as in traditional circumscription. In

this way it is possible to reason about this policy in the same theory. Pointwise circumscription is a variant of predicate circumscription in which the policy is specify individually for each ground literal. In predicate circumscription each predicate has a policy valid for all its literals. For instance, it is possible to apply negation as failure to some cases and explicit negation to others.

The basic change from definitions in section 2 is that policy literals can now be specified in the language. In order to do so, we include in the alphabet a kind of second order predicate $\texttt{min}(\cdot)$ where $\texttt{min}(L)$ has the intuitive meaning of "L is a literal where negation as failure can be applied", or in circumscriptive terminology "L is a minimized literal". The language will be now the set of all normal literal joined with all these policy literals.

Definition 5.1 Let $\mathcal{A}$ be an alphabet such that it includes a special unary predicate $\texttt{min}(\cdot)$. If L is a literal in this alphabet, we say that $\texttt{min}(L)$ is a *policy literal*. The *circumscriptive language* in alphabet $\mathcal{A}$ is the set of all literals and policy literals.

Now if a policy literal $\texttt{min}(L)$ belongs to the consequences of a programs, then negation as failure will be applied to $\overline{L}$.

The resulting programs are simpler than programs in definitions 2.2 and 2.8. Since programs use literals and there is only one negative connective $\neg$, an atom A or its complement $\neg A$ are the only possibilities of formula. But observe that in this case $\texttt{min}(\cdot)$ may be part of the atoms.

Definition 5.2 If $\mathcal{L}$ is a circumscriptive language, then a *circumscriptive program* in $\mathcal{L}$ is a set of rules

$$L \leftarrow Body \tag{3}$$

such that $Body$ is a finite subset of $\mathcal{L}$, and $L \in \mathcal{L}$.

Recall that in section 2 rules of the form (1) and (2) have their bodies separated into a positive and a negative part; rules of the form (3) do not present this discrimination. Thus the latter are simpler to write and understand. Note that sometimes is confusing the fact that a literal $\neg A$ belongs to the *positive* body of a rule.

Different semantics of "negation as failure" can be applied to the minimized literals. Definition 2.5 of answer sets and 2.6 of well-founded models are easily extended to circumscriptive programs as shown in [8]. These extensions observe the Dix's original formal properties [6] of the corresponding semantics, with minor adjustments to the new language. The case of the well-founded semantics was shown in [9, 10]. Moreover, it is possible to apply other semantic definitions to negation and also to combine several of them in the same program. This is a great step towards a declarative programming because the programmer uses negative information no matter how the information was defined. The following examples are translations of those introduced in section 3.

Example 5.3 This is the problem in example 3.1.

$$
\begin{array}{lcl}
\texttt{eligible}(X) & \leftarrow & \texttt{highGPA}(X) \\
\texttt{eligible}(X) & \leftarrow & \texttt{minority}(X), \texttt{fairGPA}(X) \\
\neg\texttt{eligible}(X) & \leftarrow & \neg\texttt{fairGPA}(X), \neg\texttt{highGPA}(X) \\
\neg\texttt{interview}(X) & \leftarrow & \texttt{eligible}(X) \\
\neg\texttt{interview}(X) & \leftarrow & \neg\texttt{eligible}(X) \\
\texttt{min}(\neg\texttt{interview}(X)) & \leftarrow &
\end{array}
$$

The last rule allows the program to deduce that all students not reaching the sufficient condition, but complying to the necessary one, are interviewed. In this case the predicate $\texttt{eligible}(\cdot)$ is incomplete, *i.e.* there are some students who are neither eligible nor not. On the other hand, predicate $\texttt{interview}(\cdot)$ is complete. Observe that both types of predicates are referred with the connective $\neg$, and then the user of $\neg\texttt{eligible}(X)$ or of $\neg\texttt{interview}(X)$ is not worried to specify if the literal was explicitly or implicitly proved. □

Example 5.4 This is the circumscriptive program from problem in example 3.3

$$
\begin{array}{lcl}
\texttt{even}(0) & \leftarrow & \\
\texttt{even}(\texttt{s}(X)) & \leftarrow & \neg\texttt{even}(X) \\
\texttt{min}(\texttt{even}(X)) & \leftarrow &
\end{array}
$$

Since the property of being an even number is exactly defined then we apply negation as failure to all numbers that are not explicitly even. □

6 Conclusions

We argue that if the goal of logic programming is a declarative knowledge representation language, then the presence of two connectives for negation does not conform to it. Every time negative information is referred it is necessary to define which one is adequate. Nonmonotonic reasoning formalization do not present this problem since negation is independent from the nonmonotonic inference constructions.

Even though several negative semantics are required for a generalized application of logic programs, expressing each one by its own connective is cumbersome and error prone. Based on techniques presented in circumscriptive theories, we proposed a framework for logic programs to solve this problem. In this framework, several definitions for negation may be associated to different literals but this is not reflected in the syntax of the language. Thus no interaction between them is possible.

Some applications of logic programs use negation as failure *not* as an epistemic operator. These applications do not fall within our framework but it is possible to characterize a modal operator of knowledge in it. This will require the introduction of second order predicates in the language, in the same line as $\texttt{min}(\cdot)$. Their semantics may be general for all literals, or pointwise for each ground literal.

References

[1] J. J. Alferes and L. M. Pereira. *Reasoning with logic programming.* Number 1111 in Lecture Notes in Artificial Intelligence. Springer-Verlag, Berlin, 1996.

[2] K. R. Apt and R. Bol. Logic programming and negation: a survey. *Journal of Logic Programming*, 30, 1993.

[3] C. Baral and M. Gelfond. Logic programming and knowledge representation. *Journal of Logic Programming*, 19, 1994.

[4] A. Colmerauer, H. Kanoui, P. Roussel, and R. Passero. Un systeme de communication homme-machine en français. Technical report, Groupe de Recherche en Intelligence Artificielle, Universite d'Aix-Marseille, 1973.

[5] J. Dix. A classification theory of semantics of normal logic programs: I. strong properties. *Fundamenta Informaticae*, 22:227–255, 1995.

[6] J. Dix. A classification theory of semantics of normal logic programs: II. weak properties. *Fundamenta Informaticae*, 22:257–288, 1995.

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

[8] 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.

[9] 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.

[10] P. R. Fillottrani and G. R. Simari. Weak properties of circumscriptive logic programming. In *Proceedings CACIC'97, Tercer Congreso Argentino de Ciencias de la Computación.*, La Plata, Argentina, 1997.

[11] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proceedings of ICLP'88*, Seattle, WA, 1988.

[12] M. Gelfond and V. Lifschitz. Logic programs with classical negation. In *Proceedings of ICLP'90*, Jerusalem, Israel, 1990.

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

[14] R. Kowalski. Predicate logic as a programming language. In *Proceedings of IFIP'74*, 1974.

[15] R. Kowalski and F. Sadri. Logic programs with exceptions. In *Proceedings of ICLP'90*, Jerusalem, Israel, 1990.

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

[17] V. Lifschitz. Circumscriptive theories: a logic-based framework for knowledge representation. In R. H. Thomason, editor, *Philosophical Logic and Artificial Intelligence*, pages 109–159. Kluwer Academic Publishers, Dordrecht, The Netherlands, 1989.

[18] V. Lifschitz. Circunscription. In D. Gabbay, editor, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume III, pages 297–352. Oxford University Press, 1990.

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

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

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

[22] D. Miller. *λ-Prolog: an introduction to the language and its logic*. Department of Computer and Information Science, University of Pennsylvania, draft edition, 1996.

[23] D. Pearce. Reasoning with negative information II: negation, strong negation and logic programming. In *Nonclassical Logic and Information Processing*, number 619 in LNAI, pages 63–79. Springer-Verlag, 1992.

[24] D. Poole. Default logic. In D. Gabbay, editor, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume III, pages 189–215. Oxford University Press, 1990.

[25] H. Przymusinska and T. C. Przymusinski. Semantic issues in deductive databases and logic programs. In A. Banerji, editor, *Formal techniques in artificial intelligence*, pages 321–367. North Holland, Amsterdan, 1990.

[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. Vivid reasoning with negative information. In W. van der Hoek, J. Meyer, Y. Tan, and C. Witteveen, editors, *Non-monotonic Reasoning and Partial Semantics*, Series in Artificial Intelligence, pages 181–205. Ellis Horwood, 1992.

[28] G. Wagner. Reasoning with inconsistency in extended deductive databases. In *Proceedings of IWLPNR'93*, Lisbon, Portugal, 1993.

[29] G. Wagner. *Vivid logic: knowledge based reasoning with two kinds of negation.* Number 764 in Lecture Notes in Artificial Intelligence. Springer-Verlag, Berlin, 1994.