Negation as Finite Failure is Paraconsistent

P. Mascellani
Mathematics Department
Siena University
via del Capitano 15, Siena, Italy

Abstract *Paraconsistent logics are generally considered somewhat esoteric. Moreover, someone argued that they simply not exist, because paraconsistent negations are not negations. The aim of this work is to provide some valid reasons to reject both these assumptions.*

Negation as finite failure (NAF) is the standard way to compute negation, used, for instance, by all the known (to me) Prolog implementations. Despite its well-known drawbacks, it is the only effective way to compute negation in logic programming. Moreover, none has ever argued that NAF is not a "negation", in the proper sense, although it is not a "classical negation".

It is quite simple to show that NAF exhibits paraconsistent behaviors, and this is yet another way to show that paraconsistent negations can be "true negations". Moreover, this implies that studies on paraconsistency are not so esoteric as they can appear at a first sight: for instance, they can provide the logics community with a clean definition of what a "negation" is.

Keywords: Non-Classical Logic, Logic Programming, Many-Valued Logic, Deductive Databases

1 Introduction

Logic programming is a way to do computations using logics or, from another point of view, is a way to provide a computational behavior to logics. The latter point of view has not been deeply investigated yet, but we think that it can help in finding the answers to some open questions of logic studies.

One of these question is the definition of negation: although almost everyone has an intuitive notion of what it is, it is still a controversial question. In recent years, this question has been raised by studies on "paraconsistent" logics; these esoteric logics (someone, see for instance [15], pretends that they doesn't exist) try to investigate "negation" operators that doesn't obey to the principle that from a contradiction everything can be deduced ("Ex falso sequitur quod libet"). The problem is that a mess of unary operators that do not obey this principle has been defined in classical and modern logics, but it is hardly sustainable that all these operators are "negations".

Needless to say, there is not a general agreement on what a "negation" is.

Instead of trying to give our definition, we start from an empirical fact: negation has been widely investigated in logic programming, mainly because classical negation has the awful characteristic of being not computable, and several definitions of negation have been proposed in order to give a computable behavior to negation. At the end of this process, negation as finite failure, has become the standard way to compute negation for logic programs. What is important to note here, is that none, as long as we know, has ever argued that negation as finite failure is not a negation.

The following step, is to investigate nega-

tion as finite failure from the point of view of its consistency: quite surprisingly, negation as failure exhibits a paraconsistent behavior. Moreover, it is possible to study the properties of the paraconsistent logic corresponding to negation as finite failure.

1.1 Plan of the paper

In the sequel of this section we briefly introduce some basic concept about logic programs. The next two section are also introductory: in Section 2, the definition of negation as finite failure is given and briefly studied; in Section 3, the same is done concerning some different definitions of paraconsistent negation.

In Section 4, we face the question of whether negation as finite failure is, or is not, a paraconsistent negation. In Section 5, some interesting properties of negation as finite failure are investigated, in order reach a deeper understanding and to classify it.

In Section 6, some final remarks are given and some future developments are proposed.

The work ends with an essential bibliography.

1.2 Preliminaries

Throughout this paper the standard notation of Lloyd [10] and Apt [2] is used. A (general) *logic program* is a set of (general) Horn clauses, i.e. a set of pairs $A \leftarrow Ls$, where A, called the *head* of the clause, is an atomic formula and Ls, called the *body*, is a possibly empty sequence of literals, which, in turn, are possibly negated atomic formulae. Clauses with empty bodies are called *facts*.

In logic programming, terms beginning with an uppercase character are considered logic variables, while terms beginning with lowercase characters are considered constants or function symbols (constants are simply function symbols with arity 0).

The *Herbrand universe* is the set of all the terms that can be build using a countable number of different constants and functional symbols of every arity. Given a logic program (or theory) T, we denote with $ground(T)$, the set of all the clauses of T, instantiated with every possible term of the Herbrand universe.

A *query* is a sequence of literals. Given a program T and a query Q, the standard process of computing the most general substitutions θ such that $T \models \theta Q$, if a such substitution exists, is called *SLD-resolution*; this process produces *SLD-derivations*. If an SLD-derivation ends with a suitable substitution it is called a *refutation*, otherwise is a *failure*.

2 Negation as finite failure

Many efforts have been produced in order to give a computable behavior to negation in the context of logic programming. The first attempt, was to establish that *everything that cannot be proven true, is false'*; however, this "closed word assumption" (CWA) is clearly non-decidable, hence, the statement has been weakened to the "negation as finite failure" (NAF) one: *every query for which every SLD-resolution finitely fails is false.*

NAF is computable (semi-decidable, for the sake of precision), but, of course, fails to say something about some formulae (those that produce infinite SLD-resolutions). In order to take this into account, computer scientists have switched from the original two-valued to many-valued logics. In particular, a quite satisfactory environment, has been found in 3-valued logic.

Definition 2.1 Let T be a logic program with negation in the body of the clauses. Then:

1. A *3-valued (Herbrand) interpretation* I is a pair (I^+, I^-), where I^+ and I^- are

disjoint subsets of the Herbrand base of T.

2. $I \models L$ iff $\begin{cases} L = A \text{ and } A \in I^+ \text{ or} \\ L = \neg A \text{ and } A \in I^- \end{cases}$

3. $I \models Ls$, where Ls is a sequence of literals (possibly negated atomic formulae), iff, for every $L \in Ls$, $I \models L$.

   $I \models \neg Ls$, iff exists $L \in Ls$, such that $I \models \neg L$ (where $\neg\neg A$ is considered as A).

4. The *immediate consequence operator* of T is defined as follows:

$$\Phi_T(I) = (J^+, J^-)$$

   where J^+ is the set:

$$\{A \mid \exists A \leftarrow Ls \in ground(T) : I \models Ls\}$$

   and J^- is the set:

$$\{A \mid \forall A \leftarrow Ls \in ground(T) : I \models \neg Ls\}$$

5. A 3-valued interpretation M is a *model* of T iff $\Phi_T(M) \subseteq M$. □

Under this framework, it is straightforward to prove the following.

Theorem 2.2 *Let T be a logic program and M_T its least 3-valued model. Then: M_T is the least fix-point of Φ_T.* □

However, the restriction of Definition 2.1 that the positive and the negative part of a 3-valued interpretation have to be disjoint, is unnecessary and can be dropped without any consequence on Theorem 2.2 (see [11]).

2.1 Bottom-up computations

An alternative way to compute a logic program, is the so called "bottom-up" computation, mainly used in deductive databases. Given an atomic formula a:

$$T \vdash a \text{ iff } \Phi_T^i(\emptyset, \emptyset) \models a \text{ for some } i$$

Theorem 2.2 prove that SLD-resolution and the bottom-up resolution are equivalent.

3 Paraconsistent negations

Not every unary logical operator can be considered a "negation"; however, it not clear which conditions must a negation operator fulfill. By the way, this is maybe one important topic that studies on paraconsistency can clarify.

Many logicians agree, at least, on some *negative* criteria, i.e. criteria that distinguish classical negation from paraconsistent negations. Discussions on this topic are outside the scope of this work; we only mention that these criteria can be syntactical as well as semantical.

In the sequel, we will use the following definitions. From the syntactical point of view:

Definition 3.1 A negation operator is *paraconsistent* iff, for some theory T and formulae a, b:

$$T, a, \neg a \nvdash b$$

□

Sometimes, a different definition of paraconsistent negation is also used.

Definition 3.2 A negation operator is *paraconsistent* iff, for some theory T and formula a:

$$T \vdash \neg(a \wedge \neg a) \tag{1}$$

does not hold. □

However, Definition 3.2 is not equivalent to Definition 3.1; moreover, Definition 3.2 is not "pure", in the sense that it relies with another logical operator, namely the "$\wedge$" operator.

From the semantical point of view, we can say:

Definition 3.3 A model M is *trivial* iff, for every formula b:

$$M \models b$$

A "negation" operator is *paraconsistent* iff exist a non-trivial model M and a formula a such that:

$$M \models a \text{ and } M \models \neg a$$

□

4 Paraconsistency of NAF

If none ever argued that NAF is not a "negation", the question is: "is it a paraconsistent negation"? Let we apply the above definitions:

- Definition 3.1 cannot be expressed as a logic program.
- Definition 3.2 is fulfilled considering an atom a which give raise to an infinite SLD-resolution or; in other words, if there is no i for which either $\Phi_T^i(\emptyset, \emptyset) \models a$ or $\Phi_T^i(\emptyset, \emptyset) \models \neg a$; however, this does not seem to catch the intuitive meaning of paraconsistency (it seems to be related with the excluded middle principle).
- Definition 3.3 requires that $a \in M^+$ and that $a \in M^-$, which is impossible, by Definition 2.1, since $M^+ \cap M^- = \emptyset$.

Hence, the only criterion saying that NAF is paraconsistent is the more controversial. However, the request of disjointness of the positive and the negative part of a 3-valued interpretation, has been shown unnecessary[1] and, in some sense, damaging; for instance, it is impossible to denote a trivial model. In [11] is shown that, if we follow this suggestion and drop this restriction, we obtain an equivalent logic.

Using this semantics for logic programs, we can conclude, using the semantic criterion, that NAF is paraconsistent. For instance, we can have the following model:

$$M = (\{a, b\}, \{a\})$$

It is straightforward to observe that M is not trivial ($M \not\models \neg b$); hence, it is paraconsistent ($M \models a$ and $M \models \neg a$).

How is it that the semantic criterion leads to a result different from the syntactic one? This contradiction is only apparent: indeed, we simply cannot express the syntactic criterion of Definition 3.1, and this does not imply that the criterion is not fulfilled. Concerning Definition 3.2, it is clearly unfulfilled, but many paraconsistentists think that it is not the right definition of paraconsistent negation, but only an interesting property (they say that a paraconsistent logic for which (1) holds is a *full* paraconsistent logic (see, for instance, [6]).

The conclusion is that all the applicable criteria show that NAF is paraconsistent.

4.1 Syntactical criterion revisited

Is it possible to modify something in our framework, in order to check the paraconsistency of NAF also against the syntactic criterion? Let us think at the bottom-up computation of logic programs: it is widely used in deductive databases, where the logic program is generally divided into an "intensional" database, containing the rules (the real program, in some sense), and the "extensional" database, containing only facts (the data).

If we admit that the data can be "controversial", i.e. that the extensional database contains both positive and negative facts, we can apply the criterion and, once again, find a paraconsistent behavior.

Example 4.1 Consider the program:

$$a \leftarrow$$

[1] Apparently, such request has been made only to avoid risks of inconsistencies

$\neg a \leftarrow$
$b \leftarrow$

that can also be viewed as an extensional database. Obviously, it is impossible to derive, for instance, $\neg b$, or c. □

In order to be more precise and to compute the consequences of the program of Example 4.1, we need to give a definition of immediate consequence operator for this kind of programs:

Definition 4.2 Consider a logic program T; the *immediate consequence operator* T is defined as follows: $\Phi_T(I) = (J^+, J^-)$, where J^+ is the set:

$$\{A \mid \exists A \leftarrow Ls \in ground(T) : I \models Ls\}$$

and J^- is the set:

$$\left\{ A \mid \begin{array}{c} \forall A \leftarrow Ls \in ground(T) : I \models \neg Ls \\ \text{or} \\ \exists \neg A \leftarrow Ls \in ground(T) : I \models Ls \end{array} \right\}$$

□

Now we can observe that the minimum model of the program of Example 4.1 (restricted to the functional symbols and constants that appear in it) is $(\{a, b\}, \{b\})$, hence, we cannot derive $\neg b$ from it. Once again, a paraconsistent behavior. It should be noted that Definition 4.2 can be adopted also if we consider clauses with bodies or, in other words, if we add an intensional part to the database.

5 Properties of NAF

It is well-known that there are properties that cannot hold together in a paraconsistent logic: for instance, a paraconsistent logic cannot be full, adjunctive, involutive and self-extensional. In order to investigate the properties of NAF, we need to define them:

Table 1: AND truth table

$\wedge$	H	T	U	F
H	H	T	U	F
T	T	T	U	F
U	U	U	U	F
F	F	F	F	F

Table 2: OR truth table

$\vee$	H	T	U	F
H	H	H	H	H
T	H	T	T	T
U	H	T	U	U
F	H	T	U	F

Definition 5.1 A paraconsistent logic is:

- *full* if, for every theory T and formula a:
$$T \vdash \neg(a \wedge \neg a)$$
- *self-extensional* if the following (replacement theorem) holds in it: if $T, a \vdash b$ and $T, b \vdash a$, then $T \vdash c$ iff $T \vdash c$ after that a has been replaced by b in T and c;
- *truth-functional* if can be described by a finite matrix. □

We already showed that NAF leads to a non-full paraconsistent logic. This logic is also clearly self-extensional, as we can see both thinking at the SLD-resolution and at the bottom-up computation. Concerning truth-functionality, this logic can be described by the 4-valued truth-functional matrices of Tables 1, 2 and 3.

In which we can intuitively explain the values as T for *true*, F for *false*, U for *undefined*, and H for *hyper-defined*.

We can explain these tables with some example:

Example 5.2 The following program `AND`:

Table 3: NOT truth table

a	$\neg a$
T	F
U	U
H	H
F	T

$$a \leftarrow b, c$$

can be used to explain the AND truth-table (Table 1). For instance, if in some model M we have that $M \models b$, $M \models \neg b$, and $M \models c$, we can say that b is *hyper-defined* and c is *true*; the consequence is that $M \models a$, but $M \not\models \neg a$, hence a is *true*, that explains why H$\wedge$T=T. □

Example 5.3 The OR truth-table (Table 2) can be explained in terms of the following "OR" program:

$$a \leftarrow b$$
$$a \leftarrow c$$

considering a model M analogous to the previous, we obtain that $M \models a$ and $M \models \neg a$, hence a is *hyper-defined* (H$\vee$T=H). □

Example 5.4 The NOT truth-table (Table 3 can be explained in terms of the following "NOT" program:

$$a \leftarrow \neg b$$

considering, for instance, a model M in which b is *true*, we obtain that $M \models \neg a$, hence a is *false* ($\neg$T=F). □

6 Conclusions

In [6] Béziau says that no paraconsistent logics "with interesting mathematical properties together with a coherent and intuitive interpretation" has been proposed until now. We think that such an object is already here: it is computational logic. In some sense, it is extremely intuitive that the "ex falso" principle is not computationally "safe" and that it has to be rejected in order to build a computational logic; however, this have not been realized until now.

In this paper, we begin to study this particular kind of logic from the paraconsistency point of view, but we hope that future works will unveil several interesting of its characteristics. Conversely, computational logic can make the paraconsistent logics community aware that the object of its studies not only exists, but is central in theoretical computer science. For instance, in [11] it is shown that the use of paraconsistent models can be useful in logic program verification.

A question that arise from all these studies is: "what is a negation"? If the closest computable approximation of classical negation exhibits a paraconsistent behavior, the question cannot be simply stated saying that the essence of negation is the "ex falso" principle and that paraconsistent negations are not negations.

More studies should be devoted to the relationships between computational logic and many-valued logics. The four-valued logic sketched in this work is closely related to the Belnap's logic (see [3]). However, we think that we have provided e new evidence of its usefulness; moreover, our derivation of the truth-tables is motivated by a well-founded computational model (see also [9]).

Another open field is to build a computational logic system, such as Prolog or Datalog, that exploits these characteristics and allows to compute the consequences of contradictorial theories. Such a system can be useful in every field where computational logic has been already used, such as deductive and inductive reasoning (including databases), data mining, and so on.

References

[1] K.R. Apt and D. Pedreschi. *Reasoning About Termination of Pure Prolog Programs.* Information and computation, 106(1):109–157, 1993.

[2] K.R. Apt. *Logic programming.* In J. van Leeuwen, editor, Handbook of Theoretical Computer Science, volume B, pages 493–574. Elsevier, 1990.

[3] N.D. Belnap. *A Useful Four-valued Logic.* In, Modern Uses of Multiple-valued Logics. D. Reidel, Dordrecht. 1977.

[4] J.Y. Béziau. *Logiques Construites Suivant les M'etodes de da Costa* Logiques et Analyse. 1990.

[5] J.Y. Béziau. *Idempotent Full Paraconsistent Negations are not Algebraizable* Notre Dame Journal of Formal Logic. 1998.

[6] J.Y. Béziau. *Are Paraconsistent Negations Negations?* Second World Congress on Paraconsistency. Juquehy, Brazil, 2000.

[7] N.C.A. da Costa. *Calculs Propositionels pour les Systémes Formels Inconsistants.* Comptes Rendus de l'Académies des Sciences de Paris. 1963.

[8] M. Fitting. *A Kripke-Kleene Semantics for General Logic Programs.* Journal of Logic Programing 2, pages 295–312.

[9] F. Fages P. Ruet. *Combining Explicit Negation and Negation by Failure via Belnap's Logic.* LIENS, 94-15. 1994.

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

[11] P. Mascellani. *Declarative Verification of General Logic Programs.* Student session, ESSLLI-2000. Birmingham UK, 2000.

[12] P. Mascellani and D. Pedreschi. *Proving Termination of Prolog Programs.* In, Proceedings 1994 Joint Conf. on Declarative Programming GULP-PRODE '94, pages 46–61, 1994.

[13] P. Mascellani and D. Pedreschi. *Total Correctness of Prolog Programs.* In F.S. de Boer and M. Gabbrielli, editors, Proceedings of the W2 Post-Conference Workshop ICLP'94. Vrije Universiteit Amsterdam, 1994.

[14] P. Mascellani and D. Pedreschi. *The Declarative Side of Magic.* to appear, 2001.

[15] B. H. Slater. *Paraconsistent Logics?* Journal of Philosophical Logic, 24, 451-454. 1995.

[16] I. Urbas. *Dual-intuitionistic Logic.* Notre Dame Journal of Formal Logic. 1996.