On the Range of Logics for Logic Programming

Alexander Bochman
Computer Science Department
Holon Institute of Technology, Israel

Abstract

We explore the range of propositional logics suitable for reasoning about logic programs under the stable semantics, starting with the logic of here-and-there as a primary representative. It will be shown, however, that there are other potential logics in the range. Still, all such logics are based on essentially the same semantics, so their differences are largely due to choice of the underlying language. Our representation suggests a more tolerant answer to the question 'What is the Logic of Logic Programming?', as well as some further expressive opportunities in using logic programs as a general knowledge representation formalism.

Introduction

Logic programming has been born with an idea that the language of classical logic can be used directly as a programming language preserving at the same time the declarative meaning of the usual logical connectives. Thus, the rules of positive (Horn) programs can also be seen as ordinary logical formulas with the usual interpretation of conjunction and implication. Moreover, this classical declarative meaning can be safely extended to positive disjunctive programs, with an additional and quite desirable effect that any classical formula becomes reducible to a logic program.

It is well known, however, that the correspondence between program rules and classical logic breaks down in the presence of negation as failure. The latter cannot be interpreted as a classical negation, and hence relevant program rules cannot be viewed as classical logical formulas. Moreover, the resulting logic programming formalism acquires some novel, *nonmonotonic* features which make it distinct from any traditional logical formalism. As a matter of fact, it is these nonmonotonic features that make logic programming a successful competitor of classical logic for problems of artificial intelligence requiring knowledge representation and reasoning.

Facing the discrepancy between logic programming and classical logic, one possible approach that has been pursued amounted to viewing the formalism of logic programs as an independent and self-subsistent knowledge representation formalism that is completely determined by the (restricted) syntax of logic programs, coupled with their (nonmonotonic) semantics. Actually, we will see later that such a view can be given a firm logical justification. However, as we have argued in more details elsewhere (see, e.g., (Bochman 2011)), a more fruitful approach to a nonmonotonic formalism emerges from representing it as a two-layered system composed of some underlying (monotonic) logic coupled with a nonmonotonic mechanism of choosing intended models. This idea is applicable to logic programming as well. In fact, a first successful realization of this view has been developed by David Pearce (see (Pearce 1997; 2006)) who has represented logic programming rules as plain logical formulas in a certain extension of intuitionistic logic called the logic of here-and-there. The logic of here-and-there (HT) is a full-fledged propositional logic with counterparts of all the usual classical connectives, and it can be used for representing and analyzing logic programs of a most general kind (see (Lifschitz, Tang, and Turner 1999; Cabalar and Ferraris 2007)). Moreover, it has been shown in (Lifschitz, Pearce, and Valverde 2001) that two logic programs are strongly equivalent (that is, interchangeable in any larger program without changing the stable semantics) if and only if they are equivalent as formulas in HT. This result has shown, in effect, that HT is a fully adequate logic for reasoning with logic programs under the stable semantics.

Granted the above results, the main question we are going to deal with in this study is whether HT is the only logic adequate for logic programs under the stable semantics, or there are other possibilities? Our answer to this question will be two-fold. On the one hand, it will be shown that there are other logics that are equally suitable for this task. On the other hand, however, we will see that all these logics are based on the same sematic framework as the models of HT, so the differences between them amount to the choice of logical connectives allowed for representing available information. In particular, it will be shown that there exists a unique maximal logic that subsumes both HT and its possible alternatives, a logic that is *expressively complete* for the underlying semantic models.

The Structural Logic of Logic Programs

Though the logic of here-and-there is a certain extension of intuitionistic logic, it is also, and most importantly, a particular three-valued logic (also called Gödel's three-valued logic). In fact, its three-valued character is determined already by the very choice of the models of HT as pairs (H, T) of interpretations (= sets of atoms) such that $H \subseteq T$ (see (Pearce 2006) for details). In order to see this, we can invoke a general interpretation of four-valued reasoning suggested long ago by Nuel Belnap in (Belnap 1977). According to the latter, any four-valued interpretation can be represented as a pair of *independent* ordinary interpretations assigning, respectively, truth and falsity to propositions, allowing thereby propositions to be not only true or false, but also neither true nor false, or both true and false. In this setting the four truth-values $\top, \mathbf{t}, \mathbf{f}, \bot$ can be identified with the four *subsets* of the set of classical truth-values $\{t, f\}$, namely with $\{t, f\}$, $\{t\}$, $\{f\}$ and $\emptyset$, respectively. Thus, $\top$ means that a proposition is both true and false (i.e., contradictory), $\mathbf{t}$ means that it is 'classically' true (that is, true without being false), $\mathbf{f}$ means that it is classically false, while $\bot$ means that it is neither true nor false (undetermined). Accordingly, any four-valued interpretation ν is uniquely determined by a pair of ordinary classical assignments, corresponding, respectively, to assignments of truth and falsity to propositions:

$$\nu \vDash A \quad \text{iff} \quad t \in \nu(A) \qquad\qquad \nu \dashv\!\!\!\!\models A \quad \text{iff} \quad f \in \nu(A)$$

By the above representation, a four-valued reasoning in general can be seen as reasoning about truth and falsity of propositions, the only distinction from classical reasoning being that the assignments of truth and falsity are independent of each other. Moreover, a three-valued setting appropriate for our subject of stable logic programming can be obtained just by restricting the set of four-valued interpretations to *consistent* interpretations that do not assign the value $\top$ to propositions. In this setting, propositions cannot be both true and false, though they still can be undetermined. From the point of view of the truth and falsity assignments, the restriction amounts to the requirement that truth ($\nu \vDash A$) implies non-falsity ($\nu \not\dashv A$), and it can be easily seen that the pairs of assignments ($\vDash, \not\dashv$) under this restriction correspond precisely to HT-models (H, T).

A general syntactic counterpart of the above binary representation of four-valued reasoning can be given in terms of biconsequence relations suggested in (Bochman 1998a). Biconsequence relations are sets of *bisequents* - rules of the form $a : b \Vdash c : d$, where a, b, c, d are finite sets of propositions. The intended interpretation of such rules is

"If all propositions from a are true and all propositions from b are false, then one of the propositions from c is true, or one of the propositions from d is false".

Bisequents are required to satisfy the following *structural rules*:

$$\frac{a : b \Vdash c : d}{a' : b' \Vdash c' : d'}, \quad \text{if } a \subseteq a', b \subseteq b', c \subseteq c', d \subseteq d'. \qquad \text{(Monotonicity)}$$

$$A : \Vdash A : \qquad \text{(Positive Reflexivity)}$$

$$: A \Vdash : A \qquad \text{(Negative Reflexivity)}$$

$$\frac{a : b \Vdash A, c : d \quad A, a : b \Vdash c : d}{a : b \Vdash c : d} \qquad \text{(Positive Cut)}$$

$$\frac{a : b \Vdash c : A, d \quad a : A, b \Vdash c : d}{a : b \Vdash c : d} \qquad \text{(Negative Cut)}$$

A biconsequence relation can be seen as a 'doubled' version of an ordinary sequent calculus reflecting the independence of the truth and falsity assignments. As in the latter, we can extend the notion of a bisequent to arbitrary (infinite) sets of propositions by accepting the following *compactness condition*:

$$u : v \Vdash w : z \quad \text{iff} \quad a : b \Vdash c : d, \qquad \text{(Compactness)}$$

for some finite sets a, b, c, d such that $a \subseteq u$, $b \subseteq v$, $c \subseteq w$ and $d \subseteq z$.

In what follows, $\overline{u}$ will denote the complement of the set u of propositions. Then the following definition describes 'canonical models' of biconsequence relations.

Definition. A pair of sets of propositions (u, v) is a *bimodel* of a biconsequence relation $\Vdash$ if

$$u : \overline{v} \nVdash \overline{u} : v.$$

The above condition is equivalent to the following requirement stating that bimodels are closed with respect to the rules of a biconsequence relation.

If $a : b \Vdash c : d$ and $a \subseteq u, b \subseteq \overline{v}$, then either $c \cap u \neq \emptyset$ or $d \cap \overline{v} \neq \emptyset$.

Now, any bimodel (u, v) can be identified with a four-valued interpretation by taking u to be the set of true propositions and v the set of propositions that are not false. Then the following Representation Theorem can be used to show that biconsequence relations are adequate for their intended four-valued interpretation.

Theorem 1 ((Representation Theorem)). *$a : b \Vdash c : d$ iff, for any bimodel (u, v), if $a \subseteq u$ and $b \subseteq \overline{v}$, then either $c \cap u \neq \emptyset$ or $d \cap \overline{v} \neq \emptyset$.*

Now, a crucial point for our subsequent discussion amounts to the fact that logic programming rules of a general form

$$\mathbf{not}\, D_1 \vee \ldots \vee \mathbf{not}\, D_k \vee C_1 \vee \ldots \vee C_l \leftarrow A_1 \wedge \ldots A_m \wedge \mathbf{not}\, B_1 \wedge \cdots \wedge \mathbf{not}\, B_n$$

can be directly represented as bisequents

$$A_1, \ldots, A_m : B_1, \ldots, B_n \Vdash C_1, \ldots, C_l : D_1, \ldots, D_k.$$

and vice versa, any bisequent in a language without connectives can be viewed as a logic programming rule. In fact, practically all known semantics for logic programs can be described directly in the framework of biconsequence relations. Moreover, it has been shown in (Bochman 1998b; 1998c) that biconsequence relations in their full (four-valued) generality provide an adequate logical framework for a very broad range of semantics suggested for logic programs, including well-founded and partial stable semantics.

For the particular case of logic programs under the stable semantics, however, the formalism of biconsequence relations should be strengthened by adding further structural rules.

A biconsequence relation will be called *consistent* if it satisfies the following structural rule:

Consistency $\qquad A : A \Vdash$

Consistency corresponds to the semantic requirement that $u \subseteq v$, for any bimodel (u, v). This constraint reduces, in effect, the formalism of biconsequence relations to a three-valued setting. It provides also a precise syntactic counterpart of the constraint $H \subseteq T$ for HT-models.

A biconsequence relation will be called *regular* if it satisfies the structural rule:

Regularity If $b : a \Vdash a : b$, then $: a \Vdash : b$.

Regularity restricts the binary semantics to a *quasi-reflexive* semantics in which, for every bimodel (u, v), (v, v) is also a bimodel. Note that this constraint is also implicit in the semantics of the logic of here-and-there that identifies, in effect, bimodels of the form (T, T) with ordinary 'classical' models.

It has been shown in (Bochman 2005) that for consistent biconsequence relations, Regularity can be replaced with the following structural rule

(C-Regularity) $$\frac{A, a : b \Vdash : d}{a : b \Vdash : A, d}$$

It turns out that biconsequence relations that are both consistent and regular constitute a maximal structural logic adequate for logic programs under the stable semantics. This fact can be demonstrated by showing that logical equivalence with respect to such biconsequence relations coincides with strong equivalence for logic programs. Thus, the following result has been proved in (Bochman 2005) (see also (Bochman and Lifschitz 2011) for a more direct description):

Theorem 2. *Two logic programs are strongly equivalent with respect to the stable semantics if and only if they determine the same consistent and regular biconsequence relation.*

In other words, logic programs Π_1 and Π_2 are strongly equivalent if and only if each program rule of Π_2 is derivable from Π_1 using the structural rules of consistent regular biconsequence relations, and vice versa. The above representation results unanimously suggest that consistent regular biconsequence relations constitute the ultimate structural logic of logic programs under the stable semantics.

A most important consequence of the above representation for our present study is that reasoning about program rules

$$\begin{aligned}\mathbf{not}\, D_1 \vee \ldots \vee \mathbf{not}\, D_k \vee C_1 \vee \ldots \vee C_l \leftarrow \\ A_1 \wedge \ldots A_m \wedge \mathbf{not}\, B_1 \wedge \cdots \wedge \mathbf{not}\, B_n\end{aligned}$$

does *not* depend on the interpretation of the connectives occurring in them (namely conjunction $\wedge$, disjunction $\vee$, negation **not** and implication $\leftarrow$) as *logical* connectives of some logic. Instead, such connectives can be safely viewed simply as suggestive punctuation marks (like commas or parentheses) that determine the structure of the program rule, a structure that is more concisely represented by the corresponding bisequent

$$A_1, \ldots, A_m : B_1, \ldots, B_n \Vdash C_1, \ldots, C_l : D_1, \ldots, D_k.$$

This representation provides, in effect, a solid formal basis for the possibility of viewing logic programming as a 'logically independent' knowledge representation formalism. One of the positive effects of this independence is that the formalism of logic programming acquires freedom of developing its own representation language and constructs that might be more suitable for the problems and tasks it deals with.

Despite all said above, the above structural representation of logic programs leaves some questions unanswered. To begin with, the use of the logical connectives in program rules bears significant heuristic value in the process of transforming 'raw' information about a problem at hand into program rules. In fact, the language of classical logic is pervasive today in informal descriptions of many domains of interest. Moreover, the original logic programming language, Prolog, freely admits compositions of the logical operators, so it treats them essentially as logical connectives. All this obviously creates an incentive for a sound and reasonable extension of the language for logic programs to compound logical formulae. Of course, the fact remains that the paradise of full classical logic is irrevocably lost for representation and reasoning with logic programs. Still, it is worth to inquire how much of it could be preserved in the logic programming reality.

The Logic of Here-and-There

To begin with, note that the formalism of biconsequence relations, unlike the majority of other formalisms for many-valued reasoning, does not depend on a particular choice of four-valued or three-valued connectives. In fact, any such connective is expressible in it via introduction and elimination rules as in ordinary sequent calculi, the only distinction being that we have to write a pair of introduction rules and a pair of elimination rules corresponding to two premise sets and two conclusion sets, respectively. Moreover, just as in the classical sequent calculus, these introduction and elimination rules can be used for reducing any bisequent in the extended language to a set of basic bisequents, that is bisequents that involve propositional atoms only. In what follows, we will illustrate this formalization on the logic of here-and-there.

The language of the logic of here-and-there is based on four propositional connectives $\{\wedge, \vee, \rightarrow, \neg\}$[1]. These connectives are definable in the framework of truth and falsity assignments as follows:

$$\begin{gathered}\nu \vDash A \wedge B \quad \text{iff} \quad \nu \vDash A \text{ and } \nu \vDash B \\ \nu \Dashv A \wedge B \quad \text{iff} \quad \nu \Dashv A \text{ or } \nu \Dashv B \\ \nu \vDash A \vee B \quad \text{iff} \quad \nu \vDash A \text{ or } \nu \vDash B \\ \nu \Dashv A \vee B \quad \text{iff} \quad \nu \Dashv A \text{ and } \nu \Dashv B \\ \nu \vDash \neg A \quad \text{iff} \quad \nu \Dashv A \\ \nu \Dashv \neg A \quad \text{iff} \quad \nu \not\Dashv A \\ \nu \vDash A \rightarrow B \text{ iff } \text{ if } \nu \vDash A, \text{ then } \nu \vDash B, \text{ and if } \nu \Dashv B, \text{ then } \nu \Dashv A \\ \nu \Dashv A \rightarrow B \quad \text{iff} \quad \nu \not\Dashv A \text{ and } \nu \Dashv B\end{gathered}$$

[1]It is known, however, that disjunction $A \vee B$ is definable using the rest of the connectives, namely as $((A \rightarrow B) \rightarrow B) \wedge ((B \rightarrow A) \rightarrow A)$.

Now, there is a systematic procedure that allows us to transform the above semantic definitions into introduction and elimination rules for the corresponding connectives in the framework of biconsequence relations (see (Bochman 2005) for details). Just as in the classical case, these rules are easily discernible from the above definitions given the intended interpretation of the premises and conclusions of a bisequent.

For our case of HT, these rules are as follows:

Rules for conjunction

$$\frac{a : b \Vdash c, A : d \quad a : b \Vdash c, B : d}{a : b \Vdash c, A \wedge B : d}$$

$$\frac{a, A, B : b \Vdash c : d}{a, A \wedge B : b \Vdash c : d}$$

$$\frac{a : b, A \Vdash c : d \quad a : b, B \Vdash c : d}{a : b, A \wedge B \Vdash c : d}$$

$$\frac{a : b \Vdash c : d, A, B}{a : b \Vdash c : d, A \wedge B}$$

Rules for disjunction

$$\frac{a, A : b \Vdash c : d \quad a, B : b \Vdash c : d}{a, A \vee B : b \Vdash c : d}$$

$$\frac{a : b \Vdash c, A, B : d}{a : b \Vdash c, A \vee B : d}$$

$$\frac{a : b \Vdash c : d, A \quad a : b \Vdash c : d, B}{a : b \Vdash c : d, A \vee B}$$

$$\frac{a : b, A, B \Vdash c : d}{a : b, A \vee B \Vdash c : d}$$

Rules for negation $\neg$

$$\frac{a : b \Vdash c : d, A}{a : b \Vdash \neg A, c : d} \qquad \frac{a : A, b \Vdash c : d}{a : b \Vdash c : \neg A, d}$$

$$\frac{a : b, A \Vdash c : d}{a, \neg A : b \Vdash c : d} \qquad \frac{a : b \Vdash c : A, d}{a : b, \neg A \Vdash c : d}$$

Rules for implication

$$\frac{a : b, A \Vdash c : d \quad a, B : b \Vdash c : d \quad a : b \Vdash c, A : d, B}{a, A \to B : b \Vdash c : d}$$

$$\frac{a, A : b \Vdash c, B : d \quad a : b, B \Vdash c : d, A}{a : b \Vdash c, A \to B : d}$$

$$\frac{a : b, B \Vdash c : A, d}{a : b, A \to B \Vdash c : d}$$

$$\frac{a : b, A \Vdash c : d \quad a : b \Vdash c : d, B}{a : b \Vdash c : d, A \to B}$$

As in the classical sequent calculus, the above rules (applied bottom-up) allow us to reduce any bisequent in the language of HT to a set of bisequents containing atomic propositions only.

Moreover, the language of HT has an important additional property, namely, it allows us to transform any bisequent into a propositional formula. More precisely, the above rules can be used for verifying that any bisequent $a : b \Vdash c : d$ in the language of HT is equivalent to the following bisequent containing a single formula in the conclusions:

$$\Vdash \bigwedge(a \cup \neg b) \to \bigvee(c \cup \neg d) :,$$

where $\neg b$ denotes the set $\{\neg A \mid A \in b\}$. Due to this possibility, any set of bisequents can be represented as a *propositional theory* in HT. Actually, it can be easily verified that, under this correspondence, the above introduction and elimination rules for the connectives of HT correspond precisely to the reduction rules for HT-formulas described in (Cabalar, Pearce, and Valverde 2005).

Summing up the above descriptions and results, the logic of here-and-there fulfils the main desiderata for a sound extension of logic programs to a full-fledged logic. More precisely, it determines a logic which defines the key connectives occurring in logic programming rules as logical connectives, while securing at the same time mutual reducibility of program rules and propositional formulas.

Desiderata for a Logic of Logic Programming

The logic of here-and-there has established a certain standard about what could be expected from a potential logic that fulfils the role of logic for logic programming. In this section we will investigate the essential ingredients of this role and requirements imposed by it. On the way we will also single out the range of alternatives that are open for complying with these requirements.

An initial and most basic requirement for such a logic stems from the fact that reasoning about logic programming rules is determined by the three-valued semantics of HT-models. This naturally suggests that any potential logic of this kind should also have a semantic interpretation in terms of such models, which implies, in turn, that it should be some three-valued logic. Accordingly, the choice of a logic reduces, in effect, to the choice of the language, namely to the choice of logical connectives allowable in the three-valued setting. Recall also that any choice of this kind will already secure that any formula or rule of such a logic will be reducible to a logic program.

Remark. It should be noted that the above requirement excludes, in effect, some weaker logics that may still be adequate for logic programs on other counts. Thus, it has been shown in (de Jongh and Hendriks 2003) that a weakest intermediate logic that characterizes strong equivalence for logic programs is the logic of weak excluded middle WEM (known also as KC), which is obtained by augmenting intuitionistic logic with the axiom $\neg A \vee \neg\neg A$. As has been noted in (Bochman and Lifschitz 2011), this means, in particular, that WEM does not differ from HT on the level of flat (non-nested) logic programs. Still, the semantics of WEM is more general than that of HT, with a side effect that complex logical formulas in WEM are not reducible, in general, to logic programs. This latter fact can be viewed, however, as a sign that the logic WEM goes beyond the logical paradigm behind logic programs.

The stable semantics of logic programs implicitly imposes a further plausible constraint on the set of admissible three-valued connectives. Recall that stable models (equilibrium models in the terminology of HT) are defined as models of the form (H, H) that satisfy some further conditions. In such models, truth coincides with non-falsity, so they determine a purely classical, two-valued logical setting. Accordingly, it is natural to require that admissible connectives of a relevant three-valued logic should behave as ordinary classical connectives in this setting. This requirement amounts to a restriction of three-valued connectives to connectives that give classical truth-values when their arguments receive classical values $\mathbf{t}$ or $\mathbf{f}$. We will call such connectives *conservative* three-valued connectives in what follows.

A final constraint on a potential logic can be formulated as a requirement that the language of such a logic should contain logical counterparts of the connectives that occur in program rules, namely conjunction, disjunction, negation and implication.

In fact, it can be safely claimed that both the conjunction and disjunction of HT constitute natural three-valued counterparts of logical conjunction and disjunction that satisfy practically all the properties of the corresponding classical connectives (such as commutativity, associativity, idempotence and distributivity). Granted this, we can restrict our search of connectives to negation and implication.

Negations: Gödel versus Lukasiewicz

The negation connective should serve as a logical counterpart of the negation-as-failure operator $\mathbf{not}$ in logic programs. Consequently, it cannot satisfy all the properties of classical negation. In particular, it should not satisfy the principle of excluded middle $A \vee \mathbf{not}\, A$, since in the logic programming setting a propositional atom may be neither proved, nor rejected (witness the program $p \leftarrow \mathbf{not}\, p$). Nevertheless, the crucial question is what are the properties of classical negation that could be preserved in this setting?

As a matter of fact, the requirements stated earlier do not leave us with too many options. First, a three-valued negation connective $\mathbf{N}$ should behave as the classical negation on the classical truth-values $\mathbf{t}$ and $\mathbf{f}$, that is, it should satisfy

$$\mathbf{N}(\mathbf{t}) = \mathbf{f} \quad \text{and} \quad \mathbf{N}(\mathbf{f}) = \mathbf{t}.$$

Second, in order to be a logical counterpart of $\mathbf{not}$, it should satisfy the condition that a bisequent $a : A, b \Vdash c : d$ must be equivalent to $a, \mathbf{N}A : b \Vdash c : d$, while $a : b \Vdash c : A, d$ should be equivalent to $a : b \Vdash c, \mathbf{N}A : d$. This latter constraint completely determines the corresponding truth assignment for $\mathbf{N}$ as the following semantic requirement:

$$\nu \vDash \mathbf{N}A \quad \text{iff} \quad \nu \dashv A.$$

It turns out that there are precisely *two* three-valued connectives that satisfy the above requirements. The first one is the negation $\neg$ of HT. The second one is the well-known *Lukasiewicz's negation* that we will denote here by $\sim$. It has the following natural semantic definition:

$$\nu \vDash {\sim}A \quad \text{iff} \quad \nu \dashv A \qquad\qquad \nu \dashv {\sim}A \quad \text{iff} \quad \nu \vDash A.$$

This fully symmetric description says that ${\sim}A$ is true whenever A is false, while ${\sim}A$ is false if and only if A is true.

A syntactic characterization of this negation is provided by the following introduction and elimination rules:

Rules for $\sim$

$$\frac{a, A : b \Vdash c : d}{a : {\sim}A, b \Vdash c : d} \qquad \frac{a : A, b \Vdash c : d}{a, {\sim}A : b \Vdash c : d}$$

$$\frac{a : b \Vdash c, A : d}{a : b \Vdash c : {\sim}A, d} \qquad \frac{a : b \Vdash c : A, d}{a : b \Vdash c, {\sim}A : d}$$

Both negations of Gödel and Lukasiewicz satisfy the two de Morgan laws with respect to disjunction and conjunction. Still, the main difference between them is that Lukasiewicz's negation satisfies the Double Negation Law:

$${\sim}{\sim}A \equiv A,$$

while $\neg$ satisfies only the Triple Negation Law

$$\neg\neg\neg A \equiv \neg A.$$

As a consequence of the Double Negation Law, conjunction and disjunction become inter-definable in the presence of $\sim$:

$$\begin{aligned} A \wedge B &\equiv {\sim}({\sim}A \vee {\sim}B) \\ A \vee B &\equiv {\sim}({\sim}A \wedge {\sim}B). \end{aligned}$$

As a final attractive feature of Lukasiewicz's negation we should also mention that it sanctions the reduction rules of the well-known Lloyd-Topor transformation of logic programs with respect to conjunction and disjunction (see (Lloyd and Topor 1984)).

The differences between these two kinds of negation can be attributed, ultimately, to their different historical origins and associated objectives. Thus, Lukasiewicz's negation has emerged as a most natural generalization of the classical negation for a three-valued setting in which the third truth-value was interpreted as 'undetermined' (neither true, nor false). This kind of negation (and its four-valued extension) is widely used in other parts of Logic, for example in relevance logic (see, e.g., (Anderson and Belnap 1975)).

On the other hand, the HT-negation can be viewed as an offspring of an intuitionistic negation in a three-valued setting. As is well-known, the intuitionistic negation does not satisfy the Double Negation Law, for philosophical reasons based on constructivist considerations. Note in this respect that, though natural from the point of view of truth and falsity assignments, the definition of $\sim$ looks very strange in the framework of intuitionistic semantics based on states ordered by a relation of information inclusion. To be more precise, all the connectives in intuitionistic semantics are 'forward-looking': their definitions involve only a current state and its extensions. This implies, in particular, that in an HT-model (H, T), the value of any formula at the 'there' state T (which does not have further extensions) depends only on the values of its components in T. In the framework of truth and falsity assignments, however, this amounts to a restriction of three-valued connectives to *negatively local* ones, namely to connectives for which their falsity is defined only in terms of falsity of their arguments. One of the immediate consequences of this fact is

Lemma. *Lukasiewicz's negation $\sim$ is not definable in the language of HT.*

Despite the above ideological differences, we will argue below that both kinds of negation can be seen as equally valuable knowledge representation tools. In both cases, given also conjunction and disjunction, logic programming rules are reducible to plain inference rules between logical formulas due to the following equivalences:

$$\begin{aligned} a : b \Vdash c : d \ &\equiv \bigwedge(a \cup {\sim}b) \colon \Vdash \bigvee(c \cup {\sim}d) \colon \\ &\equiv \bigwedge(a \cup \neg b) \colon \Vdash \bigvee(c \cup \neg d) \colon \end{aligned}$$

If we want to transform such inference rules into logical formulas, however, we need a logical implication connective.

Rules and Implications

Both logic programming rules and their associated bisequents should be primarily viewed as *inference rules* rather than logical formulas. However, in expressive logical formalisms such as classical or intuitionistic logic, inference rules are reducible to logical formulas due to availability of the corresponding implication connective that satisfies the Deduction Theorem: $a, A \vdash B$ iff $a \vdash A \supset B$. In our case, the HT-implication $\to$ satisfies this property, so, as we already mentioned, it allows us to transform any bisequent into a logical formula of HT.

As has been established in the preceding section, the connectives of conjunction, disjunction and negation allow us to transform any bisequent into a rule of the form $A\colon \Vdash B\colon$, where A and B are logical formulas. Now, an implication connective $\Rightarrow$ will sanction a reduction of such inference rules to formulas if it will satisfy a condition that $A\colon \Vdash B\colon$ is equivalent to $\Vdash A \Rightarrow B\colon$. A semantic counterpart of this condition is the following requirement:

(R) $\nu \vDash A \Rightarrow B$ holds for all admissible valuations ν if and only if $\nu \vDash A$ implies $\nu \vDash B$ for all such valuations.

Now, the following 'classical' definition of truth for implication provides the simplest way of complying with the above requirement:

$$\nu \vDash A \Rightarrow B \quad \text{iff} \quad \nu \nvDash A \text{ or } \nu \vDash B. \qquad \text{(R')}$$

It turns out that there are precisely two conservative three-valued connectives that satisfy the above constraint and coincide with classical implication when restricted to the classical truth-values $\mathbf{t}$ and $\mathbf{f}$. Given (R') as a common condition for the truth assignment, these implications are determined, respectively, by adding one of the following falsity assignments:

$$\begin{aligned} \nu \dashv A \supset B \quad &\text{iff} \quad \nu \vDash A \text{ and } \nu \dashv B \\ \nu \dashv A \supset_0 B \quad &\text{iff} \quad \nu \vDash A \text{ and } \nu \nvDash B. \end{aligned}$$

The first of these implications $\supset$ has been used in (Arieli and Avron 1996). Its corresponding syntactic characterization is provided by the following introduction and elimination rules:

Rules for $\supset$

$$\frac{a : b \Vdash c, A : d \quad a, B : b \Vdash c : d}{a, A \supset B : b \Vdash c : d}$$

$$\frac{a, A : b \Vdash c, B : d}{a : b \Vdash c, A \supset B : d}$$

$$\frac{a, A : b, B \Vdash c : d}{a : b, A \supset B \Vdash c : d}$$

$$\frac{a : b \Vdash c, A : d \quad a : b \Vdash c : d, B}{a : b \Vdash c : d, A \supset B}$$

In contrast to the above two implications, however, the implication of HT does not satisfy the condition (R'); its corresponding condition of truth is more complex (see above). Still, the adequacy of $\to$ follows from the fact that it satisfies (R), namely $\nu \vDash A \to B$ holds for all admissible valuations ν if and only if $\nu \vDash A$ implies $\nu \vDash B$ for all such valuations[2]. Nevertheless, a stronger condition for truth shows itself in more complex reduction rules for HT implication, as compared, for example, with the above rules for $\supset$.

In fact, in order to evaluate potential alternative candidates on the role of implication in logic programming, it should be taken into account that implication serves not only for reification of program rules, but also, and most importantly, for encoding conditional assertions and if-then-else constructs. What complicates the analysis in this situation is a real possibility that these two roles may conflict with each other. Namely, some implication connectives may be adequate for encoding program rules as formulas, but are less appropriate for representing conditional claims, and vice versa, there are implication connectives that provide a reasonable formalization for conditional assertions, but cannot transform rules into formulas.

As an example of the latter situation, we can consider yet another implication $A \supset_{LT} B$ defined as ${\sim}A \vee B$. A most attractive feature of this implication is that it is determined by introduction and elimination rules that correspond precisely to *Lloyd-Topor reduction rules* for implication:

Rules for $\supset_{LT}$

$$\frac{a : b, A \Vdash c : d \quad a, B : b \Vdash c : d}{a, A \supset_{LT} B : b \Vdash c : d}$$

$$\frac{a : b \Vdash c, B : d, A}{a : b \Vdash c, A \supset_{LT} B : d}$$

[2]The semantic property of regularity is essential for this equivalence.

$$\frac{a, A : b, B \Vdash c : d}{a : b, A \supset_{LT} B \Vdash c : d}$$

$$\frac{a : b \Vdash c, A : d \quad a : b \Vdash c : d, B}{a : b \Vdash c : d, A \supset_{LT} B}$$

Combined with our preceding results for disjunction, conjunction and negation, we even can state the following

Theorem 3. *The language $\{\wedge, \vee, \sim, \supset_{LT}\}$ provides a sound logical representation of logic programs that satisfies the Lloyd-Topor transformation rules with respect to the stable semantics of logic programs.*

The above result implies that, after all, logic programs under the stable semantics can be given a logical interpretation that satisfies the Lloyd-Topor transformation rules. Still, this interpretation has, however, a serious drawback, namely the implication $\supset_{LT}$ is not adequate for transforming program rules into formulas. This follows already from the fact that it does not satisfy reflexivity $A \Rightarrow A$, whereas logic programs under the stable semantics freely admit program rules $p \leftarrow p$.

A Reconciliation

Facing the above proliferation of potential logics for logic programming, we will suggest in this section a more general approach to the question 'What is a logic of logic programming?'.

Our starting point is based on an observation that all the above alternative logics are based on the same basic three-valued semantics of HT-models. As a result, they are completely determined, in effect, by different choices of conservative three-valued connectives. In fact, all such logics can be viewed simply as alternative *ways of encoding three-valued information.* And just as in the case of classical logic, we can ask whether we can achieve *functional completeness* in our choice of connectives, namely find a set of connectives such that any other connective can be expressed using this set. Fortunately, this can be done.

As we already mentioned, Lukasiewicz's negation $\sim$ is not expressible in the language of the logic of here-and-there. Accordingly, the language of HT is functionally incomplete in the class of all conservative three-valued functions. However, it turns out that adding $\sim$ to the language of HT will already suffice for achieving functional completeness. In fact, the implication $\to$ of HT will already be definable in this extended language, so we will obtain the following

Theorem 4. *The language $\{\wedge, \neg, \sim\}$ is functionally complete for the class of all conservative three-valued functions.*

Proof. Using the HT-negation $\neg$, we can define a unary connective $\mathbf{A}A$ as $\neg\neg A$, and then the result follows from a functional completeness of the set $\{\wedge, \sim, \mathbf{A}\}$, proved in (Bochman 2005). $\square$

Let us denote by LP3 the logic determined by the language $\{\wedge, \neg, \sim\}$. An axiomatic description of this logic is provided by the introduction and elimination rules for conjunction and the two negations, given earlier. Moreover, the above result shows, in effect, that LP3 is a maximal logic for reasoning about logic programs. In this logic we have full freedom in defining any of the connectives that have been mentioned in this study, and even for defining any other three-valued conservative connective that we could find useful in applications. Consequently, program rules can be encoded as propositional formulas in this logic (in a number of ways), though we still have that any formula of LP3 is reducible, in turn, to a logic program. In this sense, the logic LP3 can be viewed as an ultimate logic for logic programs under the stable semantics.

As a final observation, instead of a formulation of LP3 in the language with two 'competitive' negations, we can use a more 'cooperative' description that employs the implication connective $\supset$, described earlier.

Note first that an HT-implication $A \to B$ is definable as $(A \supset B) \wedge (\neg B \supset \neg A)$. Moreover, in the presence of $\sim$, we can define also an HT-negation:

$$\neg A \equiv \sim(A \supset \sim(A \supset A)).$$

As a result, we immediately obtain the following

Corollary. *The language $\{\wedge, \supset, \sim\}$ is functionally complete for the class of all conservative three-valued functions.*

Thus, the logic LP3 can also be formulated in the language $\{\wedge, \supset, \sim\}$. As has been established earlier, this language also directly provides all the necessary connectives for a logical representation of logic programs.

Conclusions

It has been shown in this study that the range of logics suitable for reasoning with logic programs under the stable model semantics is determined, in effect, by possible choices of propositional connectives definable in the framework of the HT-semantics. As a matter of fact, the idea that a logical variation in reasoning about logic programs is largely confined to the choice of the underlying logical language can actually be extended far beyond the stable semantics. Thus, it has been shown in (Bochman 1998b; 1998c) that practically all semantics suggested for general logic programs can be viewed as instantiations of the same nonmonotonic construction in different logical languages. Even a causal representation of logic programs, described in (Bochman 2004), can be seen as a particular language choice for basically the same semantic interpretation (see also (Bochman 2005) for a more detailed picture).

The approach sketched in the last section above might be considered as a particular elaboration of the claim that logic programming is a logically independent knowledge representation formalism that is permitted to have its own objectives and expressive means. Basically, we suggest to replace the question "What is a logic of logic programs?" (which does not have a unique answer) with a more appropriate question "What are the logical means that are available in representing information and reasoning about logic programs?". According to this view, the choice of a logic amounts, in effect, to a choice of a language used for encoding information in logic programs. In fact, we even do not have to strive for a maximal choice in this respect (provided by the logic LP3), especially if a restricted language turns out to be more efficient from a computational point of view. In any case, we are dealing just with different ways of representing knowledge and information in the formalism of logic programs.

References

Anderson, A. R., and Belnap, N. D. J. 1975. *Entailment: The Logic of Relevance and Necessity*. Princeton: Princeton University Press.

Arieli, O., and Avron, A. 1996. Reasoning with logical bilattices. *Journal of Logic, Language, and Information* 5:25–63.

Belnap, Jr, N. D. 1977. A useful four-valued logic. In Dunn, M., and Epstein, G., eds., *Modern Uses of Multiple-Valued Logic*. D. Reidel. 8–41.

Bochman, A., and Lifschitz, V. 2011. Yet another characterization of strong equivalence. In Hermenegildo, M., and Schaub, T., eds., *Technical Communications of the 26th Int'l. Conference on Logic Programming (ICLP'10)*, volume 7 of *Leibniz International Proceedings in Informatics (LIPIcs)*, 281–290. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

Bochman, A. 1998a. Biconsequence relations: A four-valued formalism of reasoning with inconsistency and incompleteness. *Notre Dame Journal of Formal Logic* 39(1):47–73.

Bochman, A. 1998b. A logical foundation for logic programming I: Biconsequence relations and nonmonotonic completion. *Journal of Logic Programming* 35:151–170.

Bochman, A. 1998c. A logical foundation for logic programming II: Semantics of general logic programs. *Journal of Logic Programming* 35:171–194.

Bochman, A. 2004. A causal logic of logic programming. In Dubois, D.; Welty, C.; and Williams, M.-A., eds., *Proc. Ninth Conference on Principles of Knowledge Representation and Reasoning, KR'04*, 427–437.

Bochman, A. 2005. *Explanatory Nonmonotonic Reasoning*. World Scientific.

Bochman, A. 2011. Logic in nonmonotonic reasoning. In Brewka, G.; Marek, V. W.; and Truszczynski, M., eds., *Nonmonotonic Reasoning. Essays Celebrating its 30th Anniversary*. College Publ. 25–61.

Cabalar, P., and Ferraris, P. 2007. Propositional theories are strongly equivalent to logic programs. *TPLP* 7(6):745–759.

Cabalar, P.; Pearce, D.; and Valverde, A. 2005. Reducing propositional theories in equilibrium logic to logic programs. In *EPIA*, volume 3808 of *Lecture Notes in Computer Science*, 4–17. Springer.

de Jongh, D. H. J., and Hendriks, L. 2003. Characterization of strongly equivalent logic programs in intermediate logics. *Theory Pract. Log. Program.* 3:259–270.

Lifschitz, V.; Pearce, D.; and Valverde, A. 2001. Strongly equivalent logic programs. *ACM Transactions on Computational Logic* 2:526–541.

Lifschitz, V.; Tang, L. R.; and Turner, H. 1999. Nested expressions in logic programs. *Annals of Mathematics and Artificial Intelligence* 25:369–389.

Lloyd, J., and Topor, R. 1984. Making Prolog more expressive. *Journal of Logic Programming* 3:225–240.

Pearce, D. 1997. A new logical characterization of stable models and answer sets. In Dix, J.; Pereira, L. M.; and Przymusinski, T., eds., *Non-Monotonic Extensions of Logic Programming*, volume 1216 of *LNAI*, 57–70. Springer.

Pearce, D. 2006. Equilibrium logic. *Ann. Math. Artif. Intell.* 47(1-2):3–41.