Lecture Notes in Computer Science

Edited by G. Goos and J. Hartmanis

474

D. Karagiannis (Ed.)

Information Systems and Artificial Intelligence: Integration Aspects

First Workshop
Ulm, FRG, March 19–21, 1990
Proceedings

Springer-Verlag
Berlin Heidelberg New York London
Paris Tokyo Hong Kong Barcelona

Contents

Negation in Logic Programming: A Formalization in Constructive Logic

François Bry

ECRC, Arabellastraße 17, D - 8000 München 81, Germany
fb@ecrc.de

ABSTRACT *The conventional formalization of logic programming in classical logic explains very convincingly the basic principles of this programming style. However, it gives no easy or intuitive explanations for the treatment of negation. Logic Programming handles negation through the so-called "Negation as Failure" inference principle which is rather unconventional from the viewpoint of classical logic. Despite its nonclassical nature, this inference principle cannot be avoided in practice. The appropriate application of Negation as Failure requires either syntactical restrictions, or significant changes in the semantics of logic programs. In this article, we defend the thesis that these syntactical restrictions or semantical changes are naturally and simply explained by observing that logic programming in fact implements no more than a constructive fragment of classical logic. Relying on a "Conditional Fixpoint Procedure", we first define a monotonic inference procedure for logic programs with negation that are consistent in this constructive fragment. Then we show how this procedure can be extended into a "Ternary Fixpoint Procedure" for general programs. This fixpoint procedure defines a ternary logic semantics for syntactically unrestricted logic programs. Finally, we argue that the constructive interpretation of logic programming also gives a simple and natural explanation of meta-programming. Relying on this view of meta-programming, we specify different forms of reflective reasoning, in particular default reasoning.*

1. Introduction

Since *Prolog* (*Pro*gramming in *Log*ic) has been developed by A. Colmerauer and R. Kowalski, the concept of "logic programming" refers to a certain type of declarative programming languages. Logic programs are formalized as relationship between terms in a logical theory [vEK76, Kow79]. This view gives a very convincing explanation of the basic principles of declarative programming languages, which is independent from any inference principle. However, this formalization proves to be inappropriate as soon as additional features – language constructs or syntactical restrictions – are considered, that are necessary for practical applications of declarative languages.

Negation is such a construct. Its processing, which could be considered rather unlogical by mathematicians, has led either to restrict the syntax of logic programs, or to formalize their semantics in ternary logics. In this paper, we show that these syntactical restrictions and nonclassical semantics are naturally and intuitively explained by referring to a rather simple fragment of classical logic.

This restricted logic is in fact the original form of mathematical logic. It is called "constructive logic" because it rejects existence proofs that do not explicitly construct the mathematical object, the existence of which is proven. Before Cantor, the restriction to constructive proofs was almost taken for granted. By applying its diagonalization principle, Cantor has shown that an additional number can be defined from any enumerable set of real numbers. Cantor's definition is however not a construction. Therefore his proof has been first strongly criticized. The article [Cal79] provides an easy introduction to this part of mathematics and to its history. Although nonconstructive proofs are nowadays no longer criticized, constructive reasoning has survived in mathematics. Various constructive logics have been developed aside the main stream of conventional mathematics. We do not need to refer to these theories, whose notations and vocabularies are often discouraging. We shall only make use of the original and quite intuitive notion of constructive proof.

The formalization of logic programming in constructive logic is not only useful for investigating negation. It gives also a very natural explanation of meta-programming. From the constructive viewpoint, it is rather natural to define theories or programs, whose variables range over formulas. Since in constructive logic formulas represent themselves but no platonistic objects, it is possible to interpret meta-programs as first-order, constructive theories. This constructive view of meta-programming is especially useful for axiomatizing non-classical logics such as, for example, autoepistemic or default logics. Such axiomatizations can also be used as implementations.

Some authors have already observed the link between logic programming and constructive reasoning. G. Huet suggests in [Hue86] to view Prolog interpreters as proof synthetisers of a sequent calculus – a kind of constructive logic. D. Bojadziev notices similarities between Prolog and constructive reasoning in the short article [Boj86]. Constructive aspects of Prolog are investigated by C. Beckstein in relationship with *Truth Maintenance Systems* in [Bec88]. J.-Y. Girard, Y. Lafont, and P. Taylor suggest in [GLT89] a possible relationship between logic programming and linear logic [Gir87] – a constructive logic recently proposed by J.-Y. Girard. In [Bry89a] the view of logic programming as a constructive fragment of classical logic was used, for the first time as far as we know, to intuitively motivating syntactical restrictions like stratification or ternary semantics. We do not know about any work relying on constructive logic for investigating meta-programming.

This article consists of seven sections, the first of which is this introduction. In Section 2, we recall the usual formalization of logic programming without negation. Then, we consider in Section 3 the processing of negation and the difficulties resulting from it. We define a "Conditional Fixpoint Operator" in Section 4. We show how to use this operator for monotonically generating consequences from programs with negation that are constructively consistent. We show how syntactical properties like stratification [ABW88, Van88] or local stratification [Prz88] are easily motivated in constructive logic: They are necessary conditions for constructive consistency. In Section 5, we propose a ternary valued semantics for syntactically unrestricted logic programs. We define a "Ternary Fixpoint Procedure" for this semantics. Finally, we propose an interpretation of meta-programming in constructive logic in Section 6. As examples of this interpretation, we show how reflective inference principles like modal and default logics can be specified as meta-programs. Section 7 is a conclusion.

This article is translated and adapted from the text of a talk given at the 1st Workshop Information Systems and Artificial Intelligence [Bry90].

2. Logic Programming without Negation

The inference principles of logic programming are *modus ponens* and the specialization rule. By *modus ponens,* a formula G is derived from two formulas F and F $\Rightarrow$ G. If c is a constant in the universe under consideration, specialization gives rise to deriving F[c] from a universal formula $\forall$x F[x]. These two inference rules are combined in the resolution principle [Rob65a], the knowledge of which is not necessary for our purposes.

The two inference principles *modus ponens* and specialization can be more or less directly implemented. The simplest approach consists in applying first the specialization rule for eliminating the universal quantifiers. A formula $\forall xyz\ p(x, y) \land q(y, z) \Rightarrow r(x, z)$ – usually written as $r(x, z) \leftarrow p(x, y) \land q(y, z)$ in logic programming – is for example specialized by instantiating its variables in all possible ways over the universe under consideration. This approach which applies the two inference rules specialization and *modus ponens* separately and successively is usually retained in *Truth Maintenance Systems* [Doy79, dK86]. A major drawback is the number of useless instantiated formulas that are, in most cases, generated.

A significant improvement consists in relying on the premise of the clauses – i.e., on $p(x, y) \land q(y, z)$ in the above-mentioned example – for restricting specialization: Only those instances of the clause $r(x, z) \leftarrow p(x, y) \land q(y, z)$ are needed that make the conjunction $p(x, y) \land q(y, z)$ true. Note that no subsequent instanciations are necessary if all variables appear in the premises, i.e., if the clause is *range-restricted.* It is common to describe this improvement for range-restricted clauses as a special type of resolution, *unit hyperresolution* [Rob65b, vEK76]. This approach is called *forward* or *bottom-up* reasoning, for it somehow follows the direction of the implication sign in the formulas or in the clauses. A remarkable efficiency can be achieved in this way, which applies specialization no more than necessary.

Following van Emdem and Kowalski [vEK76], the denotational semantics of a logic program is traditionally defined in terms of bottom-up reasoning. The semantics of a logic program without negation P is defined in terms of the facts that are derivable from P by iterated applications of bottom-up reasoning. A facts which is derivable from a specialization of a clause in P by one application of *modus ponens* is called a *direct consequence* of P. The set of the direct consequences of P is noted T(P).

$T\uparrow^{n}(P)$ denotes the set of facts that are derivable from instances of clauses in P by at most n applications of the *modus ponens* rule. This set is inductively defined as follows:

$$\begin{aligned} T\uparrow^{0}(P) &= P \\ T\uparrow^{n+1}(P) &= T(T\uparrow^{n}(P)) \cup T\uparrow^{n}(P) \end{aligned}$$

The sets of all facts that are derivable from P is:

$$\bigcup_{n \in N} T\uparrow^{n}(P) = T\uparrow^{0}(P) \cup T\uparrow^{1}(P) \cup \dots \cup T\uparrow^{n}(P) \cup \dots$$

It is denoted $T\uparrow^{\omega}(P)$. This set is the fixpoint of the function $T\uparrow$ on the program P.

The function $T\uparrow$ and the computation of its fixpoints are traditionally used for formalizing the semantics of logic programs without negation. Despite of an elegant analogy between model theory in logic and denotational semantics of programs [vEK76], it is worth to notice that the computation of a fixpoint $T\uparrow^{\omega}(P)$ is in principle a forward reasoning procedure.

This procedure has two forms, often called *naïve* and *semi-naïve methods.* The naïve method computes the sets $T\uparrow^{n}(P)$ for increasing values of n. The semi-naïve methods improves the naïve methods by only computing thoses facts of $T\uparrow^{n+1}(P)$ that have at least one premise in

$$T\uparrow^{n}(P) \setminus T\uparrow^{n-1}(P)$$

This aims at restricting the computation of $T\uparrow^{n+1}(P)$ as much as possible to that of

$$T\uparrow^{n+1}(P) \setminus T\uparrow^{n}(P)$$

The operators T and $T\uparrow$ are monotonic on programs without negation: If $P_1 \subseteq P_2$, then $T(P_1) \subseteq T(P_2)$ and $T\uparrow(P_1) \subseteq T\uparrow(P_2)$.

In order to avoid the generation of useless facts and to restrict as much as possible the generation to relevant facts, so-called backward or *top-down* reasoning methods are applied. It is a quite intuitive fact, that forward reasoning cannot take into account the constants occurring in queries. This inference technique indeed considers the queries at the very end. As opposed backward reasoning methods start from the queries. They can therefore propagate the constants occurring in queries.

Backward reasoning is close to the so-called problem reduction reasoning paradigm. For example, in order to prove a fact p(a), a backward reasoning methods searches for all clauses whose conclusions can be unified with p(a). For example, the clause

$$p(x) \leftarrow q(x) \wedge r(x)$$

would be selected. In contrast, the following clauses would not be retained:

$$\begin{gathered} s(x) \leftarrow t(x) \wedge u(x) \\ p(b) \leftarrow v(x) \end{gathered}$$

The premises of the selected clauses – also called "bodies" – are in turn similarly processed.

A particular form of backward reasoning, based on linear resolution (see, e.g., [Sti86]), gives the principle – or the so-called operational semantics – of the *Prolog* programming language [Llo87]. Logic programming is however neither restricted to linear resolution, nor to backward reasoning. In order to correctly handle recursive programs, a few new procedures have been defined in the last few years, among others the *magic sets method* [BMSU86, BR87] and SLDAL-resolution [Vie89]. It has been recently observed that these methods all implement the same nonlinear backward reasoning principle [Bry89b, Man89].[1] Forward reasoning is for example applied in abstract interpretation [GCS88, BD88].

It is worth noting that logic programming (without negation) cannot prove all theorems of (negation-free) logical theories. From the clauses

$$\begin{gathered} p(a) \\ p(f(x)) \leftarrow p(x) \end{gathered}$$

it is possible to derive the following property:

$$\forall n \in N^* \; p(f^n(x))$$

This universal formulas is however not derived from a logic program, because the induction principle is missing in logic programming.

[1]Some methods *implement* backward reasoning in a language of clauses that are processed forward. This property of the *programming language* used does however not characterize the *implementation.*

Disjunctive formulas like $a \lor b$ cannot be expressed in logic programs. (The logical dependency of a fact a from b or from c can however be expressed by means of two clauses $a \leftarrow b$ and $a \leftarrow c$.) In logic programming, a disjunctive statement

$$a \lor b$$

can only be proven by proving a or by proving b. Classical logic gives rise to other proofs, for example based on the *tertio non datur* or excluded middle principle.

The above-mentioned restrictions are all constructive. The formalization of logic programming in full classical logic is already questionable for programs without negation. An interpretation in constructive logic would be more natural.

3. A Non-Classical Inference Principle: Negation as Failure

In logic programming, a negated formula $\neg F$ is considered proven when F is not provable. This inference principle is called *negation as failure* [Cla78]. This manner to derive negative information can be called a reflective inference principle, for the reasoning system reflects on its own knowledge. It departs from classical logic which treats negative and positive expressions similarly.

The negation as failure principle is rather intuitive. It is often used in every-day life. For example, one concludes that there is no direct flight from Nürnberg to Stuttgart if one finds none in the time table. This reasoning is nonmonotonic: If new flights are created, negative conclusions might be no longer valid.

The negation as failure principle has been introduced for relational databases under the name of *closed world assumption* in the seventies [Rei78]. Because it is natural and also because it gives rise to an efficient backward evaluation of negated expressions, this principle has been retained. It is quite interesting to notice that inference principles that are, like negation as failure, unconventional for mathematicians, but rather intuitive for nonmathematicians, have been proposed within constructive logics.

Negation as failure is very close to default reasoning [Rei80]. One can for example see the condition $r(x)$ in the clause $p(x) \leftarrow q(x) \land \neg r(x)$ as an exception for the negation-free clause $p(x) \leftarrow q(x)$. Since negation as failure is a reflective inference principle, it is also related to modal nonmonotonic logics – e.g., autoepistemic logic [Moo85].

Negation as failure poses problems for several reasons. In classical logic, expressions such as $a \Leftarrow b$ and $a \lor \neg b$ are equivalent. As a consequence, implications like

$$p \Leftarrow r \land \neg q$$

and

$$q \Leftarrow r \wedge \neg p$$

are also equivalent. Are however the following clauses also equivalent?

$$p \leftarrow r \wedge \neg q$$
$$q \leftarrow r \wedge \neg p$$

Obviously not: From a program consisting of the fact r and first clause, p and $\neg q$ are derived. In contrast $\neg p$ and q follow from r and the second clause.

In constructive logic, the formulas $a \Leftarrow b$ and $a \vee \neg b$ are not equivalent. The implication $a \Leftarrow b$ means only that a can be proven as soon as b is provable. If implications are only interpreted in this constructive manner, an implication $a \Leftarrow \neg a$ is always false: a and $\neg a$ cannot be both provable. The constructive interpretation of implications gives a natural explanation, why the two clauses considered above do not have the same effect: They are not constructively equivalent.

Another difficulty caused by the negation as failure principle is the nonmonotonicity of the operator T. Consider the program $P = \{p \leftarrow \neg q, q \leftarrow r, r\}$. Since $q \notin P$ and $r \in P$, we have $T(P) = \{p, q\}$. The derivation of p is however questionable because it relies on $\neg q$. The later derivation of q contradicts this assumption. The operator T therefore is not suited to defining the denotational semantics of programs with negation. In the next section, we propose a monotonic fixpoint procedure for such logic programs.

4. Monotonic Reasoning on Programs with Negation

In order to restore monotonic reasoning on programs with negation, one has to delay the evaluation of negated expressions. Instead of generating facts, as does the operator T, we propose to collect the negative premises for later evaluations.

The facts a and b and the clause $c \leftarrow a \wedge b \wedge \neg c$ give for example rise to deriving the clause $c \leftarrow \neg c$. From the two following clauses

$$a \leftarrow b \wedge \neg c$$
$$b \leftarrow \neg d$$

one can derive the clause $a \leftarrow \neg c \wedge \neg d$. Nonground clauses are appropriately instanciated and similarly processed.

We first illustrate on an example the operator based on this principle. Then, we recall the formal definition from the article [Bry89a]. Since the new operator generates a kind of conditional facts, we call it "Conditional Fixpoint Operator" and denote it T_c.

Let P be the following logic program:

$p(x) \leftarrow q(x) \wedge \neg t(x) \wedge \neg r(x)$	$s(a)$	
$q(x) \leftarrow s(x) \wedge \neg t(x)$	$s(b)$	$u(b)$
$r(x) \leftarrow s(x) \wedge \neg u(x)$		$u(c)$

During a first phase, the following clauses – or conditional facts – are generated, until a fixpoint is reached.

$T_c \uparrow^1(P)$: P
$q(a) \leftarrow \neg t(a)$
$q(b) \leftarrow \neg t(b)$
$r(a) \leftarrow \neg u(a)$
$r(b) \leftarrow \neg u(b)$

$T_c \uparrow^2(P)$: $T_c \uparrow^1(P)$
$p(a) \leftarrow \neg t(a) \wedge \neg r(a)$
$p(b) \leftarrow \neg t(b) \wedge \neg r(b)$

Then, in a second phase, these clauses are simplified. The literals $\neg t(a)$ and $\neg t(b)$ are eliminated, since there are neither facts nor clauses defining t. Similarly, $\neg u(a)$ is eliminated. The r(a) and the clause $p(a) \leftarrow \neg r(a)$ result from these first eliminations. Since r(a) has been already obtained, this clause is useless: We eliminate it as well. Pursuing this elimination process results in the following set of facts:

$$\{q(a), q(b), r(a), p(b)\}$$

The definition of the T_c operator given below makes use of a few notations. Given a literal or a conjunction of literals B, 'pos(B)' will denote the conjunction of positive literals in B or, if this is the case, the only positive literal in B. Similarly, 'neg(B)' will denote the conjunction of negative literals or the single negative literal in B. If B does not contain any positive literal (negative literal, resp.) 'pos(B)' ('neg(B)', resp.) is defined as 'true'. A clause, the body of which is 'true' or a conjunction of negative literals, will be called a "conditional fact".

Definition 1:
Let P be a logic program. A conditional fact

$$C: A\sigma \leftarrow \text{neg}(B\sigma)\ C_1 \wedge \ldots \wedge C_i \wedge \ldots \wedge C_n$$

is a conditional consequence of P – i.e., $C \in T_c(P)$ – if:
1. There is a clause $(A \leftarrow B)$ in P
2. There is a substitution σ instanciating the variables in $(A \leftarrow B)$ with ground terms of the language of P
3. $\text{pos}(B)\sigma = \text{true}$
or $\text{pos}(B) = A_1 \wedge \ldots \wedge A_i \wedge \ldots A_n$ $(n \geq 1)$ and for all $i \in \{1, \ldots, n\}$ there is a clause $A_i \leftarrow C_i$ in P, or $C_i = \text{true}$ and A_i is a fact in P.

Since negative literals are not evaluated during the computation of $T_c(P)$, we have:

Proposition:
The operator $T_c(P)$ is monotonic. It has a unique fixpoint.

The Conditional Fixpoint Procedure is defined as follows:

Definition 2:
Let P be a logic program (without function symbols). The Conditional Fixpoint Procedure consists of two successive phases:
1. The fixpoint $T_c(P)$ is first computed.
2. The set $T_c(P)$ is reduced to a set of facts by applying the following rewriting rules:

$$\begin{array}{llll}
(F \leftarrow \text{true}) \rightarrow F & & \text{true} \wedge F & \rightarrow F \\
F \wedge \text{true} \quad \rightarrow F & & F \wedge \text{false} & \rightarrow \text{false} \\
\text{false} \wedge F \quad \rightarrow \text{false} & & (F \leftarrow \text{false}) & \rightarrow \Lambda \\
 & \neg F & \rightarrow \text{true} &
\end{array}$$

if F is neither a fact, nor the conclusion of a clause.

The reduction phase of the Conditional Fixpoint Procedure always terminates, as soon as the program under consideration is constructively consistent [Bry89a]. This condition prevents that a clause, the premise of which is the negation of the conclusion – e.g., $p(a) \leftarrow \neg p(a)$ – is derivable. In the next section, we show the procedure of Definition 2 can be adapted to also handle logic programs that are constructively inconsistent.

The extension of Definition 2 to logic programs with function symbols is possible. It requires however more complex notations. For the sake of simplicity, we do not give it here.

The operator T_c does not perform nonmonotonic deductions. However, consequences of a program P can still become invalid in a program extending P with additional facts or clauses.

During the last few years, various classes of syntactically restricted logic programs with negation have been proposed, among others hierarchical, stratified [ABW88, Van88], and locally stratified [Prz88] programs. The constructive formalization of logic programming gives a rather simple motivation for these syntactical restrictions: There are necessary conditions for constructive consistency [Bry89a]. Previously, these syntactical restrictions were only procedurally motivated.

5. Ternary Logic for Unrestricted Programs

The fixpoint semantics of logic programs with negation described in the previous section in terms of the operator T_c does not take into account logic programs that are constructively inconsistent. In practice, it is however often desirable to accept such programs by somehow "ignoring" their inconsistent parts. In this section, we propose a semantics for general programs which formalizes this intention of "overseing" inconsistencies.

This semantics is defined by slightly modifying the reduction phase of the Conditional Fixpoint Procedure. Clauses such as $p(a) \leftarrow \neg p(a)$ that are false from the constructive viewpoint are simply eliminated, and the facts that are "defined" by such clauses - $p(a)$ in case of $p(a) \leftarrow \neg p(a)$ - are marked as "unknown". They are thus distinguished from facts without any definition. The negation as failure principle is maintained by interpreting as "false" those facts that have neither consistent, nor inconsistent definitions. We first illustrate this modification of the Conditional Fixpoint Procedure on an example.

Let p be the following logic program:

$$\begin{array}{lll} p \leftarrow a & q \leftarrow \neg p & a \\ p \leftarrow q & r \leftarrow \neg r & \end{array}$$

From P the following marked facts are derived: a: true, p: true, r: unknown, and q: false. This semantics is called ternary or 3-valued, for it has three truth values. The following inequalities express, in the usual formalism of ternary logics, the precedences between the truth values:

$$\text{true} \geq \text{unknown} \geq \text{false}$$

A fact is not assigned the truth value "false" as soon as it can be proven - i.e., marked "true" - or marked "unknown". Similarly, as soon as a fact is proven, it is removed from the facts marked "unknown".

We call "Ternary Fixpoint Procedure" the reasoning informally described above. Like the Conditional Fixpoint Procedure, it is based on the T_c operator. It is formally defined as follows:

Definition 3:
Let P be a logic program (without function symbols). The Ternary Fixpoint Procedure consists of three successive phases:
1. The fixpoint $T_c \uparrow(P)$ is first computed.
2. The set $T_c \uparrow(P)$ is then reduced to a set of marked facts by applying the following rewriting rules, where F denotes an atom:

$$\begin{array}{llll}
F & \rightarrow (F\text{: true}) & (F \leftarrow \text{true}) & \rightarrow (F\text{: true}) \\
\text{true} \wedge F & \rightarrow F & F \wedge \text{true} & \rightarrow F \\
F \wedge \text{false} & \rightarrow \text{false} & \text{false} \wedge F & \rightarrow \text{false} \\
(F \leftarrow \text{false}) & \rightarrow \Lambda & & \\
(F \leftarrow \neg F) & \rightarrow (F\text{: unknown}) & &
\end{array}$$

$\neg F \rightarrow$ true
if F is neither a fact, nor a fact marked "true", nor the conclusion of a clause in P

3. Finally, all facts that are neither marked "true", nor "unknown" are marked "false"; moreover "unknown" marks are removed from facts also marked "true".

It is worth noting that ternary logics are constructive logics. Ternary logics for unrestricted logic programs have been defined in various ways, for example in [Fit85, GL88, VRS88, BF90, Van89]. They do not significantly differ from the above-defined semantics. Moreover, the Ternary Fixpoint Procedure can easily be adapted to reflect these differences. Some authors make use of similar concepts as the T_c operator, which was introduced in [Bry89a].

Like the Conditional Fixpoint Procedure, and in the same way, the Ternary Fixpoint Procedure can be extended to logic programs with function symbols.

6. Application: Reflective Reasoning

Constructive logic is not platonistic. Constructive reasoning does not refer to any meaning of the formulas, but to the formulas themselves. A frequent example of this way of thinking is the hypothesis that distinct constants denote distinct objects. This hypothesis is called *Unique Name Axiom* in database theory. In logic programming, this hypothesis is conveyed through the restriction to Herbrand models. It is questionable, whether human thinking is restricted in a similar way. We shall not try to answer this philosophical issue. We just notice that constructive reasoning is well suited to artificial intelligence: A computer just constructs.

From the constructive viewpoint, it is rather natural to define logic programs with variables ranging over formulas. Such programs are called "meta-programs" [SS86b]. They are often seen as second-order theories. The constructive viewpoint, however, suggests another formalization. Since the variables of meta-programs represent formulas, but no mathematical objects defined by these formulas, meta-programs can be formalized in first-order logic – see in particular [End72], pp. 281-289.

Meta-programming is a mean to overcome restrictions of logic programming, for it gives rise to specifying within logic programming, reasoning principles different from that of logic programming. We illustrate this point by giving some examples of meta-programs that implement some forms of reflective and nonmonotonic reasoning.

Logic programming does not give rise to deriving universal formulas. The following meta-program specify an evaluation of universal formulas in implicative form. The specification is sound and complete for restricted universal quantification [Bry89c].

$$\text{forall}(x, y \Rightarrow z) \quad \leftarrow \quad \neg\,(y \wedge \neg\, z)$$

Let F be the formula $\forall x\ p(x) \Rightarrow q(x)$ and P the program $\{p(a), p(b), q(a), q(b), q(c)\}$. When y is instanciated with p(x), and z with q(x), a backward evaluation of forall(x, y $\Rightarrow$ z) binds the variable x successively with all values in p-facts. For each instantiation of x, q(x) is checked. It is worth noting, that this way to prove universal formulas is constructive.

We show by means of a simple example how a form of autoepistemic reasoning can be easily implemented by a logic meta-program.

Let A, B, and C be three agents. assume that A believes all what B believes, provided that C does not believe it. Assume also that B believes everything provable. Assume finally that C believes everything simple. We assume that an information is simple if it is provable without applying *modus ponens.*

$$\begin{aligned}
&\text{believes}(A, x) \leftarrow \text{believes}(B, x) \wedge \neg\, \text{believes}(C, x)\\
&\text{believes}(C, x) \leftarrow \text{fact}(x)\\
&\text{believes}(B, x) \leftarrow \text{proven}(x)\\
&\text{believes}(C, x) \leftarrow \text{simple}(x)
\end{aligned}$$

$$\begin{aligned}
&\text{simple}(F_1 \wedge F_2) &&\leftarrow \text{simple}(F_1) \wedge \text{simple}(F_2)\\
&\text{simple}(F) &&\leftarrow \text{fact}(F)
\end{aligned}$$

Finally, we propose a meta-program for default reasoning. General rules are assumed to be stored is facts by means of a predicate "clause". For example,

$$\text{clause}(\text{flies}(x) \leftarrow \text{bird}(x))$$

means that birds normally fly. Exceptions to general rules are assumed to be expressed by using an "except" predicate. Since penguins are birds, but do not fly, we have:

$$\text{except}(\text{flies}(\text{penguin}))$$

The following meta-program implements default reasoning:

$$\begin{aligned}
&\text{proved}(x) &&\leftarrow \text{clause}(x \leftarrow y) \wedge \text{proved}(y) \wedge \neg\, \text{except}(x)\\
&\text{proved}(x_1 \wedge x_1) &&\leftarrow \text{proved}(x_1) \wedge \text{proved}(x_2)\\
&\text{proved}(x) &&\leftarrow \text{fact}(x)
\end{aligned}$$

From the following object-program

```
clause(flies(x) ← bird(x))
    bird(crow)
    bird(penguin)
    except(flies(penguin))
```

the above-defined meta-program derives only the fact flies(crow).

It is worth noting that exceptions – i.e., "except" facts – can as well be defined by means of clauses. These clauses in turn can admit exceptions. Processing such "nested exceptions" by means of the Conditional or Ternary Fixpoint Procedures results in the hierarchical reasoning suggested by Poole in [Poo98] for default reasoning systems. Thus, logic meta-programming is very convenient to easily specifying – and implementing – sophisticated default logics.

We illustrate the notion of "nested exceptions" on an example. Cars must stop if the traffic light is red. This rule however does not apply to ambulance cars provided they are in service. This last restriction – i.e., "in service" – may be viewed as an exception to the derogatory rule for ambulance cars. In the formalism of the above-defined meta-program, this example can be represented as follows:

```
      stop             ← red
except(stop)           ← ambulance
except(except(stop))   ← after-service
```

Instead of refering to predicates – "stop" and "except(stop)" in the previous example – exceptions can also refer to clauses. This requires a slight modification of the formalism. This is easily achieved by, for example, naming the clauses and modifying the "proved" meta-program. We assume that a clause x ← y is uniquely assigned an identifier z. We also assume that clauses named in this way can be retrieved using a binary predicate "clause":

$$\text{clause}(z,\ x \leftarrow y)$$

The following meta-program handles exceptions to clauses:

$$\begin{array}{lll} \text{proved}(x) & \leftarrow & \text{clause}(z,\ x \leftarrow y) \wedge \text{proved}(y) \wedge \neg\ \text{except}(z) \\ \text{proved}(x_1 \wedge x_1) & \leftarrow & \text{proved}(x_1) \wedge \text{proved}(x_2) \\ \text{proved}(x) & \leftarrow & \text{fact}(x) \end{array}$$

Doing so, different exceptions can be given for "stop", depending on the stop-rule under consideration. Such a refined representation give rises, for example, to express that there is no derogation to stopping at level crossing when a train arrives. The following clauses formalize this example:

```
clause(1,       stop             ← red)
clause(2,       stop             ← level-crossing ∧ train)
clause(3, except(1)              ← ambulance)
clause(4, except(except(1)) ← after-service)
```

Clearly, some applications might require to handle both, exceptions to clauses as well as exceptions to predicates, depending on the predicates. We leave specifying the corresponding meta-program to the reader, for it is quite easy.

Because it gives rise to easily refining various deduction schemes, we argue that meta-programming is very convenient for defining and implementing complex reasoning, especially nonmonotonic reasoning systems.

7. Conclusion

In this article, we have proposed a particular interpretation of logic programs. According to this interpretation, logic programming departs from modern mathematics. It restricts reasoning to certain inference rules, in a manner which corresponds to a primitive form of mathematical logic, constructive logic. It provides us with rather natural explanations for several features of logic programming.

In particular, it explains very naturally syntactical restrictions such as "stratification" and "local stratification" that were proposed for overcoming difficulties resulting from the non-monotonicity of negation as failure. The constructive interpretation of logic programs also provides us with a convenient framework for defining a ternary semantics for syntactically unrestricted programs.

The constructive viewpoint also suggests a formalization of meta-programming. By means of a few examples of meta-programs, we have shown how logic programming is closely related to other non-monotonic logics. We have shown how meta-programming gives rise to elegant specifications of various logics that depart from the very semantics of logic programming. These specifications are as well implementations. Thus, although it implements a restricted, constructive fragment of classical logic, logic programming gives rise, through meta-programming, to overcoming its own limitation.

We argue that it is preferable to specify refined forms of reasoning as logic meta-programs, instead of developing new formalisms or new implementation paradigms, for the following reasons. First, constructive reasoning – the paradigm of logic programming – is very natural and intuitive. Second, the meta-programming approach gives rise to easily combining into one system, different forms of reasoning. Finally, program transformation techniques – in particular partial evaluation, see, e.g., [SS86a] – have been developed for automatically generating efficient implementations from specifications in form of logic meta-programs.

Acknowledgement

This research was partly supported by the European Community in the framework of the Basic Research Action *Compulog* No. 3012.

References

[ABW88] K. R. Apt, H. A. Blair, and A. Walker. *Foundations of Deductive Databases and Logic Programming*, chapter Towards a Theory of Declarative Knowledge. Morgan Kaufmann, Los Altos, Calif., 1988.

[BD88] M. Bruynooghe and D. De Schreye. Tutorial Notes for: Abstract Interpretation in Logic Programming. Research report, University of Leuven, 1988. Invited tutorial of the 5th Int. Conf. on Logic Programming (ICLP), Seattle, 1988.

[Bec88] C. Beckstein. *Zur Logik der Logik-Programmierung – Ein konstruktiver Ansatz*. Informatik-Fachbericht 199. Springer-Verlag, 1988.

[BF90] N. Bidoit and C. Froidevaux. Negation by Default and Unstratifiable Logic Programs. *Theoretical Computer Science*, 1990. To appear.

[BMSU86] F. Bancilhon, D. Maier, Y. Sagiv, and J. Ullamn. Magic Sets and other Strange Ways to Implement Logic Programs. In *Proc. 5th ACM SIGMOD-SIGACT Symp. on Principles of Database Systems (PODS)*, 1986.

[Boj86] D. Bojadziev. A Constructive View of Prolog. *Journal of Logic Programming*, 3(1):69–74, 1986.

[BR87] C. Beeri and R. Ramakrishnan. On the Power of Magic. In *Proc. 6th ACM SIGMOD-SIGACT Symp. on Principles of Database Systems (PODS)*, 1987.

[Bry89a] F. Bry. Logic Programming as Constructivism: A Formalization and its Application to Databases. In *Proc. 8th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems (PODS)*, Philadelphia, Penn., March 1989.

[Bry89b] F. Bry. Query Evaluation in Recursive Databases: Bottom-up and Top-down Reconciled. In *Proc. 1st Int. Conf. on Deductive and Object-Oriented Databases (DOOD)*, Kyoto, Japan, Dec. 1989. The complete version of this article will appear in the special issue of *Data & Knowledge Engineering* devoted to the DOOD '89 Conference.

[Bry89c] F. Bry. Towards an Efficient Evaluation of General Queries: Quantifier and Disjunction Processing Revisited. In *Proc. ACM-SIGMOD Conf. on Management of Data (SIGMOD)*, Portland, Oreg., May-June 1989.

[Bry90] F. Bry. Negation in logischer Programmierung: Eine Formalisierung in konstruktiver Logik. Research Report IR-KB-72, ECRC, 1990.

[Cal79] A. Calder. Constructive Mathematics. *Scientific American*, 241(4):134–143, 1979.

[Cla78] K. L. Clark. *Logic and Databases*, chapter Negation as Failure. Plenum Press, New York, 1978.

[dK86] J. de Kleer. An Assumption-Based Truth Maintenance system. *Artificial Intelligence*, 28:127–162, 1986.

[Doy79] J. Doyle. A Truth Maintenance system. *Artificial Intelligence*, 24, 1979.

[End72] H. B. Enderton. *A Mathematical Introduction to Logic.* Academic Press, 1972.

[Fit85] M. Fitting. A Kripke-Kleene Semantics for Logic Programs. *Journal of Logic Programming*, 4:295–312, 1985.

[GCS88] J. Gallagher, M. Codish, and E. Shapiro. Specialisation of Prolog and FCP Programs Using Abstract Interpretation. *New Generation Computing*, 6:159–186, 1988.

[Gir87] J.-Y. Girard. Linear Logic. *Theoretical Computer Science*, 50:1–102, 1987.

[GL88] M. Gelfond and V. Lifschitz. The Stable Model Semantics for Logic Programming. In *Proc.* 5th *Int. Conf. and Symp. on Logic Programming (ICLP-SLP)*, Seattle, 1988.

[GLT89] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types.* Cambridge Tracts in Theoretical Computer Science 7. Cambridge University Press, 1989.

[Hue86] G. Huet. *Fundamentals of Artificial Intelligence*, chapter Deduction and Computation. LNCS 232. Springer-Verlag, 1986.

[Kow79] R. Kowalski. *Logic for Problem Solving.* North Holland, 1979.

[Llo87] J. Lloyd. *Foundations of Logic Programming.* Symbolic Computation. Springer-Verlag, 1987. Second Edition.

[Man89] R. Manthey. Can We Reach a Uniform Paradigm for Deductive Query Evaluation? In *Proc.* 3rd *Int. GI Congress "Knowledge Based Systems"*, München, Oct. 1989.

[Moo85] R. C. Moore. Semantical Considerations on Nonmonotonic Logic. *Artificial Intelligence*, 25, 1985.

[Poo98] D. Poole. Explanation and Prediction: An Architecture for Default and Abductive Reasoning. Tech. Report 89-4, Dept. of Comp. Sc., Univ. of British Columbia, 198.

[Prz88] T. C. Przymusinski. *Foundations of Deductive Databases and Logic Programming*, chapter On the Declarative Semantics of Deductive Databases and Logic Programs. Morgan Kaufmann, Los Altos, Calif., 1988.

[Rei78] R. Reiter. *On Closed World Databases*, pages 56–76. Plenum Press, New York, 1978.

[Rei80] R. Reiter. A Logic for Default Reasoning. *Artificial Intelligence*, 13, 1980.

[Rob65a] J. A. Robinson. A Machine-Oriented Logik Based on the Resolution Principle. *Journal of the ACM*, 12(1):23–41, 1965.

[Rob65b] J. A. Robinson. Automated Deduction with Hyper-Resolution. *Journal of Comp. Math.*, 1:227–234, 1965.

[SS86a] S. Safra and E. Shapiro. Meta Interpreters for Real. In *Proc.* 12th *IFIP World Congress*, 1986.

[SS86b] L. Sterling and E. Shapiro. *The Art of Prolog.* MIT Press, 1986.

[Sti86] M. E. Stickel. *An Introduction to Automated Deduction*, pages 75–132. Springer Verlag, Berlin, 1986.

[Van88] A. Van Gelder. *Foundations of Deductive Databases and Logic Programming*, chapter Negation as Failure Using Tight Derivations for General Logic Programs. Morgan Kaufmann, Los Altos, Calif., 1988.

[Van89] A. Van Gelder. The Alternating Fixpoint of Logic Programs with Negation. In *Proc.* 8th *ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems (PODS)*, Philadelphia, Penn., March 1989.

[vEK76] M. van Emden and R. Kowalski. The Semantics of Predicate Logic as a Programming Language. *Journal of the ACM*, 23(4):733–742, 1976.

[Vie89] L. Vieille. Recursive Query Processing: The Power of Logic. *Theoretical Computer Science*, 69(1):1–53, Dec. 1989.

[VRS88] A. Van Gelder, K. A. Ross, and Schlipf J. S. The Well-Founded Semantics for General Logic Programs. In *Proc.* 7th *ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems (PODS)*, Austin, Texas, March 1988.