DATENBANKSYSTEME L 3325

Proceedings of the Eighth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems

acm PRESS

March 29-31, 1989

Philadelphia, Pennsylvania

416 132 931 800 14

Special Interest Group for Automata and Computability Theory
(SIGACT)

Special Interest Group for the Management of Data
(SIGMOD)

Special Interest Group for Artificial Intelligence
(SIGART)

Table of Contents

Logic Programming as Constructivism: A Formalization and its Application to Databases

François Bry

ECRC, Arabellastr. 17, 8000 München 81, West Germany
uucp: ...!pyramid!ecrcvax!fb

ABSTRACT *The features of logic programming that seem unconventional from the viewpoint of classical logic can be explained in terms of constructivistic logic. We motivate and propose a constructivistic proof theory of non-Horn logic programming. Then, we apply this formalization for establishing results of practical interest. First, we show that 'stratification' can be motivated in a simple and intuitive way. Relying on similar motivations, we introduce the larger classes of 'loosely stratified' and 'constructively consistent' programs. Second, we give a formal basis for introducing quantifiers into queries and logic programs by defining 'constructively domain independent' formulas. Third, we extend the Generalized Magic Sets procedure to loosely stratified and constructively consistent programs, by relying on a 'conditional fixpoint' procedure.*

1. Introduction

Though close to conventional reasoning, logic programming departs from classical logic in two respects. First, it confines reasoning to limited kinds of deductions. In particular, indefinite statements like disjunctive or existential formulas cannot be derived from logic programs. Second, logic programming draws unconventional inferences by interpreting negation as failure. However, despite of non-classical features, logic programming appears rather natural. Moreover, its unconventional reasoning features seem intuitively founded.

In this paper, we propose a constructivistic rationalization of logic programming. Constructivism is a school in logic that tries to reestablish certain parts of mathematics in more intuitive ways. There are many constructivistic theories. Some retain classical reasoning and confine it to certain types of deduction. Others rely on unconventional inference principles. Logic programming does both. We show that constructivism is surprisingly close to logic programming. The features of logic programming that are unconventional from the classical viewpoint find immediate constructivistic explanations.

A number of formalizations of logic programming have already been proposed. Chandra and Harel [CH 85], Apt, Blair, and Walker [A* 88], and Van Gelder [VGE 88] express the semantics of Horn and non-Horn programs in terms of conventional logic models and fixpoint operators, following van Emden and Kowalski [vEK 76]. In [GR 84, GAB 85], Gabbay and Reyle propose to extend Prolog with non-classical, hypothetical implications. In [GS 86], Gabbay and Sergot advocate for replacing negation as failure by the classical logic treatment 'negation as inconsistency'. Fitting [FIT 85] relies on a three-valued logic for formalizing the behaviour of logic programs that either fail, or succeed, or fall into infinite backtracking. A modal logic interpretation of negation as failure is described by Gabbay in [GAB 86], etc. The different readings contribute to enlighten various aspects of logic programming.

The resemblance between logic programming and constructivistic logic has already been noticed by Bojadziev. In a short article [BOJ 86], he gives a constructivistic interpretation of Horn programs and negative goals. Remarks with constructivistic flavour can be found in most studies devoted to negation in logic programming. However, we do not know any previous proposal to interpret non-Horn logic programs in constructivistic terms and to exploit this interpretation. We show that a constructivistic reading of logic programming answers the question of the declarative semantics of non-Horn programs in a simple and natural manner. In addition, we apply this reading to solving practical problems of various kinds.

In this paper, because of space limitations, we do not give proofs and we consider function-free logic programs. However, the constructivistic rationalization of logic programming we introduce here applies also to logic programs with functions. In particular, it gives very intuitive explanations of necessary requirements such as well-foundedness or local stratification [PRZ 88a, PRZ 88b]. The proofs, a treatment of logic programs with functions, and connected results can be found in the full version [BRY 88a] of this paper.

The first part of this paper proposes a constructivistic axiomatic system, which we call Causal Predicate Calculus (CPC), as a proof-theoretic formalization of non-Horn logic programs. In order to establish the factual decidability of CPC, we extend the fixpoint procedure for Horn programs [vEK 76] into a proof procedure for CPC, which we call 'conditional fixpoint', by introducing some conditional reasoning. We prove the equivalence between the proof-theoretic reading of non-Horn programs with CPC and the model-theoretic one by Apt, Blair, Walker [A* 88] and Van Gelder [VGE 88].

The second part of this paper is devoted to applying the constructivistic axiomatization of logic programming. We prove results of practical consequence in three concerns: For motivating the syntactical restrictions imposed on logic programs in simple and intuitive manners; for extending logic programs with new features; and for proving results on certain database query evaluation methods.

More precisely, we show that stratification [A* 88, VGE 88] and local stratification [PRZ 88a, PRZ 88b] are sufficient conditions of 'constructive consistency', i.e., consistency in CPC. We introduce the class of 'loosely stratified' programs. This property, which is less stringent than stratification and local stratification, appears to be more convenient for practical use. Like stratification but unlike local stratification, loose stratification can be checked without rule instantiation. We establish, for stratified programs, the equivalence between the proof-theoretic formalization with CPC and the model-theoretic one proposed in [A* 88, VGE 88].

We then consider queries with quantifiers. We introduce the concept of 'constructive domain independence' (cdi) as a proof-theoretic counterpart to the model-theoretic notion of 'domain independence' studied by Fagin [FAG 80] and proposed by Kuhns [KUH 67] under the name of 'definiteness'. The new concept 'constructive domain independence' refines and formally motivates syntactical properties previously considered, such as 'safety' introduced by Ullman [ULL 80], 'range-restriction' due to Nicolas [NIC 81], or 'allowedness' investigated by Clark [CLA 78], Lloyd and Topor [LT 86], and Shepherdson [SHE 88]. It gives a logical, constructivistic explanation of the need to keep ordered certain conjunctions in logic programs, a feature traditionally considered non-logical and procedural. As opposed to the classical domain independence, the constructive domain independence is a decidable and syntactically recognizable property. It therefore constitutes a practical basis for introducing quantifiers into logic programs and queries.

In logic, proofs are declaratively defined, i.e., proofs are considered independently from any proof procedure. The definition of CPC induces a declarative definition of constructive proofs. We make use of this definition and of the conditional fixpoint procedure for extending in a quite simple manner the Generalized Magic Sets procedure [BC* 86, BR 87] - a proof procedure for recursive logic programs also proposed under the name of Alexander procedure [R* 86] - to constructively consistent non-Horn programs. More precisely, we show that, although the rewritings of the

Generalized Magic Sets procedure compromise stratification, they preserve constructive consistency. This gives rise to apply the conditional fixpoint procedure to evaluate the rewritten programs.

The paper is organized as follows. Section 1 is this introduction. Following [vEK 76], Section 2 shortly summarizes how both model theory and proof theory convey the declarative semantics of logic programming. Section 3 gives a brief outline of the principles of constructivism. In Section 4, we develop the Causal Predicate Calculus (CPC) for formalizing logic programming. We apply this formalism to practical problems in Section 5 . In Section 6 we summarize the main results of the paper and indicate directions for further research.

2. Model-theoretic and proof-theoretic semantics

Given certain axioms, mathematical logic distinguishes between two complementary issues: The interpretation of the axioms by some classes of mathematical structures, and the construction of proofs from the axioms. The first issue is called model theory, the second proof theory. Intuitively, model theory is concerned with the study of the "world(s)" described by the axioms while proof theory is devoted to the techniques of inferring new properties from those explicitly stated by the axioms.

In logic programming, both the model theoretic and the proof theoretic readings are useful - if not necessary - for conveying the semantics attached to sets of axioms. This has been observed by van Emden and Kowalski in [vEK 76]. In order to promote the language of Horn clauses as a programming language, they have investigated on the one hand the close correspondence between denotational semantics of programs and model theory, and between operational semantics and proof theory on the other hand. Some logicians use the word 'semantics' in place of model theory and call 'syntax' the proof theory - see, e.g., [CHU 56]. Instead, we give here to the term 'semantics' the same meaning as in programming language theory.

The denotational semantics of a program describes the objects and structures that are consulted or constructed by the program. The operational semantics provides with a description of the operations performed by the program, without necessarily defining the implemented procedure. Viewing logic as a programming language raises two questions: "What is a proof?" and "How to generate proofs?", i.e., the complementary questions of giving declarative and procedural definitions to the operational semantics.

Despite a fallacious appearance of simplicity, non-Horn programs raise a severe difficulty: Their operational semantics - or underlying proof theory - cannot be defined in classical logic. As opposed to Horn programs, they perform inferences that do not always conform to classical logic and conventional reasoning. For example, the rules $p \leftarrow r \wedge \neg q$ and $q \leftarrow r \wedge \neg p$ are not identically interpreted though equivalent in classical logic. Conveying the same non-classical interpretation of implications, constructivism is appropriate to formalize declaratively the operational semantics of non-Horn programs.

A procedural, proof-theoretic treatment of non-Horn programs has been developed by Lloyd in terms of the SLDNF-resolution proof procedure [LLO 84]. As opposed, the proof-theory we propose here is independent of any procedure. It is declarative and therefore easily applicable to proof procedures that are not based on SLDNF-resolution, e.g., the Generalized Magic Sets [BC* 86, BR 87] or Alexandre procedure [R* 86].

3. Constructivism: An outline

A brief outline of the principles of constructivistic logic is proposed in order to show that it surprisingly resembles logic programming. Refer to [TRO 77] for a detailed overview of constructivism in mathematics.

There is no clear-cut definition of constructivism. According to Quine [QUI 70], constructivism can be broadly described as "intolerance of methods that lead to affirming the existence of things of some sort without showing how to find one". Constructivism does not allow indefiniteness in proofs. It rejects proofs affirming the truth of $F_1 \vee F_2$

without telling which one of F_1 and F_2 holds. Similarly, indefinite existence conveyed by existential quantifications is not constructively provable: A constructive proof of $\exists x\, F[x]$ does exhibit a term satisfying F.

A classical example of non-constructive reasoning is the following proof of the existence of irrational numbers p and q such that p^q is rational:

> $(\sqrt{2})^{\sqrt{2}}$ is either rational or irrational. If it is rational, take $p = q = \sqrt{2}$ which is known to be irrational. Hence, $p^q = (\sqrt{2})^{\sqrt{2}}$ is rational. If $(\sqrt{2})^{\sqrt{2}}$ is irrational, take $p = (\sqrt{2})^{\sqrt{2}}$ and $q = \sqrt{2}$, hence $p^q = 2$ is rational.

This proof is not constructive because it draws consequences from a disjunctive hypothesis - $(\sqrt{2})^{\sqrt{2}}$ is either rational or irrational - which is not based on established facts - the proof does not show whether $(\sqrt{2})^{\sqrt{2}}$ is rational or not. In other words, constructivism rejects excluded middle.

Examples of constructive proofs are easily found in mathematics and computer science: It is a general inclination to prefer constructive proofs to non-constructive ones. The results given in this paper are all constructively established (for not debating on the legitimacy of non-constructed foundations for a constructivistic theory).

It is interesting to recall that the introduction of non-constructive proofs into mathematics led to controversies. In fact, until the end of the 19th century and Cantor's set theory, mathematics was constructivistic. "This is not mathematics. It is theology", said a mathematician about the non-constructive techniques introduced by Cantor [CAL 79]. Though these techniques are now considered as providing "a paradise the mathematicians do not want be driven from", as Hilbert said, contemporary mathematicians revive constructivism, with the aim to provide "realistic" and intuitive motivations to classical results - see, e.g., [BIS 67].

The constructivistic interpretation of disjunctive and existential statements corresponds to the practice in logic programming. Logic programming prevents the derivation of indefinite information by forbidding disjunction and existential quantification in heads of rules. A constructivistic view of logic programming is interesting because it is usually more intuitive to people not trained in formal logic, like most of the database and expert system users.

Assuming an intuitive understanding of the proofs of ground atomic formulas, constructive proofs can be formalized as follows [BRO 54, KRE 65]:

Definition 3.1

A. Closed formulas:

1. A constructive proof of $F_1 \wedge F_2$ consists in a constructive proof of F_1 and a constructive proof of F_2.
2. A constructive proof of $F_1 \vee F_2$ consists in a constructive proof of F_1 or in a constructive proof of F_2.
3. A constructive proof of $F_1 \Rightarrow F_2$ consists in specifying a procedure T which transforms any constructive proof P_1 of F_1 into a constructive proof $T(P_1)$ of F_2.
4. $\neg F$ is defined as $F \Rightarrow$ false.
5. If the variable x ranges over the domain D, a constructive proof of $\forall x\, F[x]$ is a procedure T which, on application to any pair (t,p) of a term t and a constructive proof p that $t \in D$, yields a constructive proof T(t,p) of F[t].
6. If the variable x ranges over the domain D, a constructive proof of $\exists x\, F[x]$ consists in a term t, in a constructive proof of $t \in D$, and then in a constructive proof of F[t].

B. Open formulas:
A constructive proof of an open formula $F[x_1,...,x_n]$ with free variables $x_1, ..., x_n$ ranging over the domain D consists in a tuple $(t_1,...,t_n)$ of terms, in n constructive proofs of $t_i \in D$, and then in a constructive proof of $F[t_1,...,t_n]$.

Though Definition 3.1 seems rather natural, it modifies considerably the notion of proof of an implication. Moreover, it strongly restricts proofs of disjunctions and of quantified expressions.

From a constructivistic viewpoint implications are not "hidden disjunctions". The formulas $p => q$ and $\neg p \vee q$ are not constructively equivalent. The same holds for $r \wedge \neg p => q$ and $r \wedge \neg q => p$. Constructivism is causalistic: Implications are viewed as inferring new information from already proved information, like in logic programming. In constructivistic logic, the formula $\neg p => p$ is considered equivalent to *false*, according to the intuition that it is impossible to transform a proof of $\neg p$ into a proof of p.

From a constructivistic viewpoint, a disjunction $p \vee \neg p$ is not necessarily true, in case both p and $\neg p$ are not constructively provable. Proofs of quantified formulas are considerably constrained. Constructive proofs of quantified expressions reduce to proofs of ground expressions. This corresponds to the logic programming practice.

Note finally that Definition 3.1 induces the concept of 'ordered conjunction'. For example, a constructive proof of an open formula F[x] consists in a constructive proof that a term t belongs to the domain *followed* by a constructive proof of F[t]. The need to keep ordered certain conjunctions in logic programs for avoiding incorrect evaluations and undesirable behaviours is classically viewed as a non-logical, procedural feature. In fact, it can be explained in logic by the restriction to constructive proofs.

Restricting the concept of proof requires in turn either to restrict the axioms, the logical axioms as well as the proper axioms, or to rely on non-classical inference principles. Adopting *modus ponens* - if formulas F_1 and $F_1 => F_2$ hold, then the formula F_2 is provable - imposes for example to reject axioms such as:

A_1: $p => q \vee r$

A_2: $\forall x\ p(x) => \forall y\ q(x,y)$

Indeed, if p is provable, A_1 would induce by *modus ponens* $q \vee r$. Similarly, if p(t) holds, then *modus ponens* permits to derive $\forall y\ q(t,y)$ from A_2.

Various constructivistic formal systems have been proposed, e.g., [GÖD 58, PRA 65, FIT 69]. Some of them rely on non-classical rules of inference. Others, e.g., [HEY 66], allow classical inference principles and express the constructivistic restriction by constraining the syntax of the axioms. Logic programming does both. It has the classical inference principle *modus ponens* and constrains the syntax of the axioms. It has negation by failure as an unconventional inference principle.

The following syntactical constraints on the axioms guarantee constructivism under *modus ponens:*

- Definiteness:
  No axiom an no conjunct of an axiom is a disjunction. No axiom and no conjunct of an axiom is an existential formula.

  If $F_1 => F_2$ is an axiom or a conjunct of an axiom, then F_2 contains no disjunctions, no implications, and no quantified formulas.

  If $Q_1x_1...Q_nx_n\ F_1 => F_2$ (Q_i denotes either $\forall$ or $\exists$) is an axiom or a conjunct of an axiom, then $Q_i = \forall$ if x_i is free in in F_2, and F_2 contains no disjunctions, no implications, and no quantified formulas.
- Positivity of consequents:
  The consequent F_2 of an implicative conjunct $F_1 => F_2$ or $Q_1x_1...Q_nx_n\ F_1 => F_2$ of an axiom is neither a negated formula, nor a conjunction containing a negated formula.

These conditions are familiar to logic programmers. Note that they do not impose that the axioms are safe [ULL 80], range-restricted [NIC 81], or allowed [LT 86, VGT 87, SHE 88]. They do not preclude axioms that are ground negative literals, or (mutually) recursive axioms, or implicative axioms with negations in their premisses.

Lemma 3.1
A formula satisfying the conditions of definiteness and of positivity of consequents is of one of the following types:

- Implicative formula

  $$F_1 => F_2$$

  where F_1 is a closed formula and F_2 is a ground atom or a conjunction of ground atoms.
- Quantified implicative formula

  $$Q_1x_1...Q_nx_n\ F_1 => F_2$$

  where $Q_i = \forall$ if x_i is a free variable in F_2, and where F_2 is an atom or a conjunction of atoms.
- Ground literal.

- Conjunction of formulas of the above-mentioned types.

In the rest of the paper, we shall make use of the following, slightly extended definition of a rule, that allows negations, quantifiers and disjunctions in bodies of rules.

Definition 3.2
A rule is an expression of the form

$$A[x_1,\dots,x_n,z_1,\dots,z_p] \leftarrow F[x_1,\dots,x_n,y_1,\dots,y_m]$$

where the head of the rule

$$A[x_1,\dots,x_n,z_1,\dots,z_p]$$

is an atom in which the x_is and the z_js are free and where the body of the rule

$$F[x_1,\dots,x_n,y_1,\dots,y_m]$$

is a formula in which the x_is and the y_js are free.

It denotes the implicative formula:

$$\forall x_1 \dots \forall x_n \forall y_1 \dots \forall y_m \forall z_1 \dots \forall z_p$$
$$F[x_1,\dots,x_n,y_1,\dots,y_m] \Rightarrow A[x_1,\dots,x_n,z_1,\dots,z_p]$$

A rule is a Horn rule if its body does not contain atoms with negative polarity. A fact is a ground atom.

Proposition 3.1
A set of axioms satisfying the conditions of definiteness and of positivity of consequents is constructively equivalent to a set of rules and ground literals.

For the sake of simplicity, we shall assume in the sequel that axioms satisfying the conditions of definiteness and of positivity of consequents are always rules or ground literals. By Proposition 3.1 there is no loss of generality.

4. The Causal Predicate Calculus

Though imposing many of the syntactical restrictions of logic programs, the conditions of definiteness and of positivity of consequents, or equivalently the restriction to facts and rules, do not suffice to formalize non-Horn logic programming. Logic programming conforms in addition to the following principles:

1. Negation as failure principle: $\neg F$ holds if F is not provable.
2. Domain closure principle: Variables range over the terms occurring in the axioms or in provable facts.
3. Decidability principle: Facts are effectively decidable, i.e., a procedure that decides whether a fact is provable or not exists and is known.

The following axiomatic system expresses these principles in constructivistic logic. We call it Causal Predicate Calculus (CPC). It formalizes the operational semantics of non-Horn logic programs independently from any proof procedure.

Upper case characters denote formulas. The symbol '&' denotes ordered conjunction: F & G means that the proof of F has to precede that of G. Proofs have to be understood according to Definition 3.1. Legal inferences are expressed as usual with the symbol '|-'.

- Inference principles:
  1. *modus ponens*
  2. negation as failure

- Axiom schemata:
  1. $\neg F \wedge F \vdash$ *false*
  2. $\neg F \Rightarrow F \vdash$ *false*
  3. $F \vdash F \vee G$
  4. $G \vdash F \vee G$
  5. $F \wedge G \vdash F$
  6. $F \wedge G \vdash G$
  7. $dom(t)\ \&\ F[t] \vdash \exists x\, F[x]$
  8. $\neg(\exists x\, \neg F[x]) \vdash \forall x\, F[x]$
  9. $\forall x\, F[x] \vdash F[t]$ (t free for x in F)

- Conditions on the proper axioms:
The proper axioms are rules or ground literals.

- Domain axioms:
For each n-ary predicate p occurring in a proper axiom, there are n axioms (i = 1, ..., n):

$$dom(x_i) \leftarrow p(x_1,\dots,x_i,\dots,x_n)$$

- Finiteness Principle:
All proofs are finite.

The first axiom schema and the finiteness principle are usually not made explicit. They are implicitly assumed in all axiomatic systems. Here, we make them explicit for two reasons. First, we would like to emphasize that *false* is provable in constructivistic logic not only with Schema 1 but also with Schema 2, as opposed to classical logic. Second, the finiteness principle induces severe restrictions on logic programs with functions [BRY 88a].

We shall call 'logic program' a finite set of rules and ground facts. Given a logic program LP, its domain, noted 'dom(LP)', is by definition the set of terms occurring in dom-facts that are constructively provable in CPC with proper axioms LP. The domain of a logic program is a subset, possibly strict, of its Herbrand universe. Therefore, the domains of function-free logic programs are finite. It follows that universally quantified and negated formulas can be decided in finite time in any function-free logic program.

In CPC disjunctive statements like $p \vee \neg p$ are true, thanks to negation as failure. Logic programs are CPCs, but not all CPCs are logic programs since CPCs may have negative literals as axioms. Horn programs are consistent since neither Schema 1 nor Schema 2 can apply. Similarly, Schema 1 is irrelevant to non-Horn logic programs.

Provided one knows that the proper axioms are consistent, e.g., because of their syntactical structure, then the axiom schemata 1 and 2 are useless. They are usually omitted by logicians who always assume consistency of the proper axioms. They are needed - at least for theoretical reasons - in logic programming and databases where such assumptions cannot always be made. In Section 5.1, we show that the properties 'stratification' and 'local stratification' ensure consistency of logic programs, thus permitting to discard Schema 2.

According to the definition of a rule and to the schemata 7 and 8, the rule

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

would be evaluated like the rule

$$p(x) \leftarrow dom(x) \ \& \ [\neg q(x) \wedge r(x)]$$

This is inefficient since 'r(x)' is a more restricted range for x. In Section 5.2 and in [BRY 88b], we show how to avoid the domain predicates.

We conclude this section by introducing a proof procedure, which we call 'conditional fixpoint', in order to establish the factual decidability of CPC with function-free axioms. The procedure relies on a 'conditional immediate consequence' operator T_c which we define first.

In presence of non-Horn rules, the immediate consequence operator T [vEK 76] is non-monotonic [A* 88, VGE 88]. We restore monotonicity with T_c by introducing some conditional reasoning. Instead of facts, conditional statements are obtained by delaying the evaluation of negative literals. Consider for example the rule

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

If a fact q(a) holds, delayed evaluation of $\neg r(a)$ yields the conditional statement

$$p(a) \leftarrow \neg r(a)$$

T_c is the immediate consequence operator that generates facts from Horn rules, and conditional statements from non-Horn rules.

We make use of the following notations in the definition of T_c. Given a conjunction of literals B, we shall denote by 'pos(B)' ('neg(B)', resp.) the conjunction of all positive (negative, resp.) literals in B. If there is no positive (negative, resp.) literals in B, then pos(B) (neg(B), resp.) reduces to *true*. We shall call 'conditional statement' a ground rule the body of which is a negative literal or a conjunction of negative literals and of *true*.

Definition 4.1
The conditional immediate consequence $T_c(LP)$ of a logic program LP is the set of all ground rules

$$H\sigma \ \leftarrow \ neg(B\sigma) \wedge C_1 \wedge ... \wedge C_n$$

that verify the conditions:

- $(H \leftarrow B) \in LP$
- σ is a substitution of terms in dom(LP) for variables in the rule $H \leftarrow B$
- $pos(B\sigma) = A_1 \wedge ... \wedge A_n$ $(n \geq 0)$ and for each $i = 1, ..., n$ either there is a conditional statement $A_i \leftarrow C_i$ in LP, or $C_i = true$ and $A_i \in$ LP.

We recall that an operator Γ is said to be monotonic if:

$$\forall S_1 \forall S_2 \quad S_1 \subseteq S_2 \quad \Rightarrow \quad \Gamma(S_1) \subseteq \Gamma(S_2)$$

We shall use the notations:

$$\begin{aligned}
\Gamma\uparrow 0(S) &= S \\
\Gamma\uparrow(n+1)(S) &= \Gamma(\Gamma\uparrow n(S)) \cup \Gamma\uparrow n(S) \quad (n \in \mathbf{N}) \\
\Gamma\uparrow\omega(S) &= \cup_{k \in N} \Gamma\uparrow k(S)
\end{aligned}$$

In other words, $\Gamma\uparrow 1(S)$ denotes the set S augmented by the conditional immediate consequences that are computable from S.

Finally, we recall that a least fixpoint of an operator Γ is by definition a set $\Gamma\uparrow n(S)$ ($n \in N^*$) such that:

$$\begin{aligned}
\Gamma\uparrow\omega(S) &= \Gamma\uparrow n(S) \\
\Gamma\uparrow\omega(S) &\neq \Gamma\uparrow(n-1)(S)
\end{aligned}$$

Lemma 4.1
The operator T_c is monotonic. It has a unique least fixpoint.

We define the 'conditional fixpoint' procedure for function-free logic programs. In [BRY 88a], we define it for logic programs with functions.

Definition 4.2
Let LP be a function-free logic program. The conditional fixpoint procedure performs in two successive phases:

1. The fixpoint $T_c\uparrow\omega(LP)$ is computed.
2. $T_c\uparrow\omega(LP)$ is reduced by recursively applying the following four rewriting rules:

$$\begin{aligned}
(F \leftarrow \mathit{true}) &\rightarrow F \\
\mathit{true} \wedge F &\rightarrow F \\
F \wedge \mathit{true} &\rightarrow F \\
\neg A &\rightarrow \mathit{true}
\end{aligned}$$

if A is neither a fact,
nor the head of a rule

In Section 5.2, we give syntactical conditions that permit not to explicitly refer to dom(LP) for the computation of $T_c\uparrow\omega(LP)$ during the first phase of the conditional fixpoint procedure.

The rewriting system which defines the reduction phase is bounded and confluent [HUE 80]. Therefore, one verifies easily that the reduction phase always terminates.

Note that the reduction phase yields a set of ground atoms. It is inspired of a proof procedure for propositional calculus due to Davis and Putnam [DP 60] - see also [CL 73, pp. 63-66]. Indeed, conditional statements are ground, by definition of T_c. The last rewriting rule of Definition 4.2 expresses the negation by failure principle.

If a logic program LP contains function symbols, then its domain and the least fixpoint $T_c\uparrow\omega(LP)$ might be infinite. In such a case, it is not possible to perform the reducing rewritings after the computation of the fixpoint $T_c\uparrow\omega(LP)$ is completed. Instead, the generation of conditional statements and their reduction have to be intertwined by level of term nesting. This is possible provided that the program is Nötherian, a property defined in [BRY 88a] that ensures that logic programs with functions obey the finiteness principle. The conditional fixpoint procedure for Nötherian programs is defined in [BRY 88a].

The proof of the completeness of the conditional fixpoint procedure makes use of the following property.

Lemma 4.2
The formulas $F \wedge G \Rightarrow H$ and F constructively imply $G \Rightarrow H$.

Proposition 4.1
The conditional fixpoint procedure decides facts in non-Horn, function-free logic programs.

Note that *false* $\in T_c\uparrow\omega(LP)$ if and only if LP is constructively inconsistent. Properties ensuring constructive consistency are investigated in the next section.

5. Applications

This section is devoted to applications of the constructivistic view of logic programming. We apply it first for motivating the restrictions imposed on function-free logic programs in an intuitive manner, then for giving a constructivistic basis to domain independent evaluations. Finally, we apply constructivism for extending the Generalized

Magic Sets procedure to constructively consistent non-Horn programs.

5.1. Constructive consistency and stratification

In this section, we first show that the properties 'stratification' [A* 88, VGE 88] and 'local stratification' are sufficient conditions of constructive consistency. Then, we introduce the class of 'loosely stratified' logic programs. This property, which is less stringent than stratification and local stratification, seems to be more convenient for practical applications. Finally, we show that the proof-theoretic formalization of logic programs and the model-theoretic treatment proposed in [A* 88] and [VGE 88] are equivalent.

We consider in this section rules whose bodies are conjunctions of literals or single literals, as in [A* 88, VGE 88, PRZ 88b].

We shall use the word 'proof' with the meaning of 'proof in CPC'. Given a logic program LP, we shall call 'proof in LP' a proof in CPC with set of proper axioms LP. We first give a characterization of proofs.

Proposition 5.1
Let LP be a logic program. Let F be a fact.

A proof of F in LP is F itself if $F \in LP$ or a ground tree structure $F \leftarrow P$ such that :

- There exist a rule $H \leftarrow B$ in LP and a substitution σ such that $H\sigma = F$.
- P is a proof of $B\sigma$ in LP.

Assume that $F \notin LP$. A proof of $\neg F$ in LP is *true* if no head of a rule in LP unifies with F. Else it is a ground tree structure $\neg F \leftarrow P$ such that:

- The rules in LP whose heads unify with F are $H_i \leftarrow B_i$ with substitutions σ_i $(i = 1, ..., n)$
- P is a proof of $\wedge_{i=1}^{i=n} \neg B_i\sigma_i$ in LP.

Definition 5.1
Let $L \leftarrow P$ be a proof of a ground literal L in a logic program LP. Let F be a fact occurring positively (negatively, resp.) in P. L is said to depend positively (negatively, resp.) on F in LP.

Proposition 5.2
A logic program LP is constructively consistent if and only if no fact depends negatively on itself in LP.

Proposition 5.2 gives a very intuitive, equivalent condition of constructive consistency of a logic program. This condition has been proposed and intuitively motivated by Deransart and Ferrand in [DF 87]. A similar intuition motivates the property 'sup-stratification' proposed by Bidoit and Froidevaux in [BF 88].

The following result shows that 'stratification' and 'local stratification' are sufficient conditions of constructive consistency. We do not recall here the definition of these properties and refer to [A* 88, VGE 88] and to [PRZ 88a, PRZ 88b], respectively.

Corollary 5.1
Stratified and locally stratified logic programs are constructively consistent.

<u>Logic Program:</u>

$p(x) \leftarrow q(x,y) \wedge \neg p(y)$
$q(a,1)$

<u>Herbrand Saturation:</u>

$p(a) \leftarrow q(a,a) \wedge \neg p(a)$
$p(a) \leftarrow q(a,1) \wedge \neg p(1)$
$p(1) \leftarrow q(1,a) \wedge \neg p(a)$
$p(1) \leftarrow q(1,1) \wedge \neg p(1)$
$q(a,1)$

Fig. 1

The converse of Lemma 5.1 does not hold. For example, the logic program of Figure 1 is constructively consistent but neither stratified, nor locally stratified. It is not stratified because the rule defining p contains a negated p-atom in its body. It is not locally stratified since its Herbrand saturation contains instances of a rule in the body of which the head atom appears negatively.

The condition of constructive consistency is difficult to apply in practice, because it relies on all possible proofs. Since there are often fewer rules than facts, it is desirable to dispose of sufficient conditions of constructive consistency that depend on the rules only.

The property 'stratification' is such a condition, since it implies the constructive consistency. The property 'local stratification' is fact independent too. However, it relies on the Herbrand saturation of the program under consideration. Therefore, it is in practice as difficult to check as constructive consistency.

We propose another sufficient condition of constructive consistency, which we call 'loose stratification'. Loose stratification is similar to, but less stringent than stratification. Like stratification and local stratification, it does not depend on the facts occurring in the logic program under consideration. As opposed to local stratification, its definition does not depend on the Herbrand saturation.

According to Lemma 1 in [A* 88, p. 97], a logic program LP is stratified if and only if the dependency graph [A* 88] of the rules in LP contains no cycles with negative arcs. For example, the rule

$$p(x) \leftarrow q(x,y) \wedge \neg r(z,x)$$

induces two arcs in the dependency graph: A positive arc

$$p \rightarrow^{+} q$$

and a negative arc

$$p \rightarrow^{-} r$$

Relying on the very intuition of the dependency graph, we define the 'adorned dependency graph' of a logic program as follows. Instead of predicates, we consider atoms with variable arguments as vertices of the adorned dependency graph. We define an arc between two atoms only if they are unifiable. In addition, we adorn an arc joining an atom A_1 to an atom A_2 with a most general unifier of A_1 and A_2. An arc is assigned a '+' or a '-' sign like in the conventional dependency graph.

Thus, the rule

$$p(x,a) \leftarrow q(x,y) \wedge \neg r(z,x) \wedge \neg p(z,b)$$

yields a positive and a negative arc:

$$p(x_1,a) \rightarrow^{+}_{[x_1/x_2]} q(x_2,x_3)$$

$$p(x_1,a) \rightarrow^{-}_{[x_1/x_4]} r(x_3,x_4)$$

Note that there is no arc from $p(x_1,a)$ to $p(x_3,b)$. Indeed, these atoms do not unify because of the constants a and b. Formally, the adorned dependency graph is defined as follows:

<u>*Definition 5.2*</u>
Let LP be a logic program. Let V be the set of atoms occurring in rules in LP. Assume that V has been rectified such that distinct elements of V do not share variables.

The adorned dependency graph of LP is the directed graph with set of vertices V and with set of arcs A defined as follows.

Given $A_1 \in V$ and $A_2 \in V$, $(A_1 \rightarrow^{s}_{\sigma} A_1) \in A$ if there is a rule $H \leftarrow B \in LP$ and a most general unifier τ such that:

- $A_1\tau = H\tau$
- $s =$ '+' if $A_2\tau$ occurs positively in $B\tau$.
  $s =$ '-' if $A_2\tau$ occurs negatively in $B\tau$.
- σ is the restriction of τ to the variables occurring in A_1 and A_2

We recall that n unifiers $\sigma_1, ..., \sigma_n$ are said to be compatible if there exists a unifier τ which is more general than each σ_i. The definition of 'loose stratification' makes use of this notion and relies on the adorned dependency graph.

<u>*Definition 5.3*</u>
A logic program LP is said to be loosely stratified if the adorned dependency graph of the rules in LP contains no finite chain

$$C:\ (A_1 \rightarrow^{s_1}_{\sigma_1} A_2 \rightarrow^{s_2}_{\sigma_2} A_3 \dots A_n \rightarrow^{s_n}_{\sigma_n} A_{n+1})$$

such that:

- C contains a negative arc, i.e., at least one s_i is '-'.
- the unifiers $\sigma_1, ..., \sigma_n$ adorning the arcs along C are compatible. There is a unifier τ which is more general than each σ_i such that $A_{n+1}\tau = A_1\tau$.

Intuitively, stratification forbids that a fact depends negatively on another fact with the same predicate letter. Loose stratification forbids such a dependence only if the unifiers collected along the rules are compatible. It allows it otherwise. Like stratification, loose stratification depends only on the rules and can be checked without rule instantiation.

Corollary 5.2
Loosely stratified logic programs are constructively consistent.

Stratified programs are loosely stratified, but the converse is false. For example, the program consisting of the rule

$$p(x,a) \leftarrow q(x,y) \wedge \neg r(z,x) \wedge \neg p(z,b)$$

is loosely stratified since constants 'a' and 'b' do not unify, but it is not stratified. The program of Figure 1 is not loosely stratified. The concepts of 'adorned dependency graph' and of 'loose stratification' are inspired of [LEW 85].

For function-free logic programs, loose stratification and local stratification coincide [VIE 88, BRY 88a]. However, this is not the case for logic programs with functions. The relationship between loose stratification and local stratification is investigated more thoroughly in [BRY 88a].

With the following proposition, we establish, for stratified programs, the equivalence between the proof-theoretic formalization with CPC and the model-theoretic one proposed in [A* 88] and [VGE 88].

Proposition 5.3
Let F be a set of facts and R a stratified set of rules. A formula is a theorem of CPC with proper axioms $F \cup R$ if and only if it is satisfied in the natural model of $F \cup R$.

5.2. Constructive domain independence

A constructive proof of an open formula F[x] or of a closed formula $\exists x$ F[x] consists in a proof of 'dom(t)' for some term 't' followed by a proof of F[t]. In this section, we show how to avoid explicit references to the domain predicates in constructive proofs.

By the domain axioms, a proof of 'dom(t)' consists in a proof of a ground fact in which 't' occurs. We shall say that a proof of 'dom(t)' occurs redundantly in a proof P if P consists of a proof of 'dom(t)' and of a proof which implies 'dom(t)'. For example, the proof of 'dom(a)' is redundant in the following proof

$$[dom(a) \leftarrow q(a,b)] \ \& \ [p(a) \leftarrow r(a,b) \wedge s(a)]$$

since p(a) => dom(a) by definition of the predicate 'dom' (Section 4).

Redundant occurences of 'dom' atoms in proofs are characterized by means of the concept of range.

Definition 5.4
Ranges for terms $t_1, ..., t_n$ are recursively defined as follows:

- $P(t_{\sigma(1)}, ..., t_{\sigma(n)})$ is a range for $t_1, ..., t_n$ if P is a predicate and σ a permutation of $\{1, ..., n\}$.
- R_1 & R_2 is a range for $t_1, ..., t_n$ if R_1 is a ranges for $u_1, ..., u_k$ $(k \geq 0)$, R_2 is a ranges for $v_1, ..., v_h$ $(h \geq 0)$, and $$\{t_1...t_n\} = \{u_1...u_k\} \cup \{v_1, ..., v_h\}$$
- $R_1 \vee R_2$ an $R_1 \wedge R_2$ are ranges for $t_1, ..., t_n$ if both R_is are ranges for $t_1, ..., t_n$.
- A term $(H \leftarrow B)$ is a range for $t_1, ..., t_n$ if B is a range for $t_1, ..., t_n$.

Definition 5.5
Let D be an atom with predicate 'dom' and let D & P be a proof.

D is redundant in the proof D & P if P is a range for all terms occurring in D.

The following concept gives rise to avoid the dom-predicates in queries.

Definition 5.6
A formula F is constructively domain independent (cdi) if for all constructive proofs P of F, the proofs of domain facts contained in P are redundant in P.

The following proposition gives a syntactical characterization of constructively domain independent formulas. Note the occurrences of the ordered conjunction '&'.

Proposition 5.4
Constructively domain independent (cdi) formulas are recursively characterized as follows:

- An atom $A[x_1,...,x_n]$ is a cdi formula.
- The conjunction ($\wedge$ or &) of two cdi formulas is a cdi formula.
- The disjunction of two cdi formulas with same free variables is a cdi formula.
- If F_1 is a cdi formula and if F_2 is any formula whose free variables are all free in F_1, then F_1 & F_2 is a cdi formula.
- $\exists x$ F is a closed cdi formula if F is an open cdi formula.
- If F_1 is a cdi formula with free variable x and if F_2 is any formula with no free variable other than x, then $\forall x \; \neg[F_1 \& \neg F_2]$ is a cdi formula.

According to Proposition 5.4 the rule

$$p(x) \leftarrow q(x) \;\&\; \neg r(x)$$

is cdi, while the rule

$$p(x) \leftarrow \neg r(x) \;\&\; q(x)$$

is not. Prolog programmers are used to make variables in negative goals occurring in a preceding positive literal as well, in order to ensure correct runs of programs. Proposition 5.4 gives a logical motivation to this practice.

Given a CPC theory C, let C_{cdi} denote the calculus obtained by removing the domain axioms from C.

Lemma 5.1
Let F[x] be an open formula with free variable x.

If F[x] is a range for x then $\forall x$ F[x] => dom(x) holds.

Proposition 5.5
Let S be a finite set of cdi formulas satisfying the conditions imposed on proper axioms of a CPC. Let C denote the CPC with proper axioms S.

C_{cdi} and C are constructively equivalent.

In [FAG 80], Fagin has studied the model-theoretic notion 'domain independent' proposed by Kuhns [KUH 67] under the name 'definiteness'. Roughly, a formula F is domain independent if its valuation in a model depends only on the extensions of the relations mentioned in F. The class domain independent formulas is not solvable [DIP 69]. However, the constructivistic restrictions imposed on proof implies the solvability of the class of constructively domain independent formulas.

Corollary 5.3
The class of constructively domain independent formulas is a solvable subclass of the domain independent formulas.

Other solvable classes of domain independent formulas have been proposed: Range-restricted formulas [NIC 81], evaluable formulas [DEM 82, VGT 87], and allowed formulas [LT 86, VGT 87, SHE 88]. For each formula in one of these classes it is possible to construct an equivalent cdi formula [BRY 88b].

5.3. The Generalized Magic Sets procedure extends to non-Horn programs

The Generalized Magic Sets procedure [BR 87] - also proposed under the name of Alexander procedure [R* 86] - is a proof procedure for Horn logic programs with recursive axioms. It is not based on SLDNF-resolution [LLO 84]. In order to achieve a good efficiency in presence of huge amounts of facts, it is 'set-oriented'. We show in this section how the concept of constructive proof and the conditional fixpoint procedure permit to extend the Generalized Magic Sets procedure to constructively consistent non-Horn programs. By Corollaries 5.1 and 5.2 the Generalized Magic Sets procedure therefore extends to stratified, locally stratified, and loosely stratified programs.

In order to conform with the definition of the Generalized Magic Sets procedure, we consider in this section - like in Section 5.1 - rules whose bodies are literals or conjunctions. In addition, we assume that they are constructively domain independent (cdi), i.e., rule's bodies are conjunctions, some of them being ordered such that a negative literal with a variable x follows a positive literal containing x.

The Generalized Magic Sets procedure answers a query on a program with rule set R and fact set F by performing three successive steps. First, the rules are specialized into a set R^{ad} of adorned rules. Second, the set of adorned rules R^{ad} is rewritten into a set R^{mg} of rules intended for bottom-up evaluation. Third, the fixpoint of the immediate consequence operator on $R^{mg} \cup F$ is computed.

The rule specialization $R \rightarrow R^{ad}$ of the first step and the rule rewriting $R^{ad} \rightarrow R^{mg}$ of the second step are formally defined in [BR 87]. Here, we recall them on examples.

Adorned rules are obtained by ordering the body literals. The (partial) ordering is chosen for optimally propagating the bindings of variables from the head of the rule backwards. Consider for example the rule:

$$p(x,y) \leftarrow q(x,z) \wedge r(z,y)$$

The ordering q(x,z) & r(z,y) is appropriate in presence of a goal such as '$p(a,z_1)$' since the binding x/a is transmitted to the first body literal. As opposed, the ordering r(z,y) & q(x,z) is preferable for the goal '$p(x_1,a)$'.

In order to permit different orderings depending on the instantiation pattern of the head, adorned predicates are introduced. A binary predicate 'p' for example induces adorned predicates like 'p^{bf}', where 'b' ('f', resp.) denotes a bound (free, resp.) argument. For example, the rule

$$p(x,y) \leftarrow r(z,y) \wedge q(x,z)$$

induces - among others - the adorned rule

$$p^{bf}(x,y) \leftarrow q^{bf}(x,z) \ \&\ r^{bf}(z,y)$$

The adorned rules are specialized forms of the original rules.

According to Proposition 5.4, non-Horn cdi rules contain ordered conjunctions. In order to preserve cdi, the reordering of body literals has to respect the ordered conjunctions. Under this condition, we have:

Proposition 5.6
If R is a set of cdi rules, then the rules in R^{ad} are cdi.

The second step of the Generalized Magic Sets procedure generates from R^{ad} a set R^{mg} of rules of two kinds. First, R^{mg} contains magic rules representing the encountered subgoals in a backward - or top-down - evaluation of the adorned rules. For example, the rule

$$p^{bf}(x,y) \leftarrow q^{bf}(x,z) \ \&\ r^{bf}(z,y)$$

yields three magic predicates 'magic-p^{bf}', 'magic-q^{bf}', and 'magic-r^{bf}'.

It induces the two magic rules

$$\text{magic-}q^{bf}(x,z) \leftarrow \text{magic-}p^{bf}(x,y)$$
$$\text{magic-}r^{bf}(z,y) \leftarrow \text{magic-}p^{bf}(x,y) \ \&\ q^{bf}(x,z)$$

In fact only 'b' variables are kept in magic-predicates: For example, 'magic-$p^{bf}(x,y)$' should be replaced by 'magic-$p^{bf}(x)$'. The magic rules of our example are therefore:

$$\text{magic-}q^{bf}(x) \leftarrow \text{magic-}p^{bf}(x)$$
$$\text{magic-}r^{bf}(z) \leftarrow \text{magic-}p^{bf}(x) \ \&\ q^{bf}(x,z)$$

Queries induce ground magic facts, called seeds. The query 'p(a,x)' induces for example the seed 'magic-$p^{bf}(a)$'.

The second type of rules in R^{mg} are modified versions of the adorned rules. These versions are obtained by inserting magic atoms in the rules of R^{ad} for constraining the instantiations. For example, the rule

$$p^{bf}(x,y) \leftarrow q^{bf}(x,z) \ \&\ r^{bf}(z,y)$$

is rewritten into

$$p^{bf}(x,y) \leftarrow \text{magic-}p^{bf}(x) \ \&\ \text{magic-}q^{bf}(x) \ \&\ q(x,z) \ \&\ \text{magic-}r^{bf}(z) \ \&\ r(z,y)$$

The rewriting $R^{ad} \leftarrow R^{mg}$ can easily be extended to non-Horn rules by processing negative literals like positive ones. For example, the rule

$$p^{bf}(x) \leftarrow q^{bf}(x) \ \&\ \neg r^{bf}(z)$$

induces the same magic atoms and magic rules as does the Horn rule

$$p^{bf}(x) \leftarrow q^{bf}(x) \ \&\ r^{bf}(z)$$

It is therefore rewritten into

$$p^{bf}(x,y) \leftarrow \text{magic-}q^{bf}(x) \ \&\ q^{bf}(x,z) \ \&\ \text{magic-}r^{bf}(z) \ \&\ \neg r^{bf}(z)$$

Assuming this extension of the rewriting $R^{ad} \rightarrow R^{mg}$, we have:

Proposition 5.7
If R^{ad} is a set of cdi rules, then the rules in R^{mg} are cdi.

As it has been often noted, only the first of the two rewritings $R \rightarrow R^{ad} \rightarrow R^{mg}$ preserves stratification. However, we show below that both preserve constructive consistency. By Corollary 5.1 this suffices to conclude to the correctness of the Magic Sets transformation for non-Horn logic programs.

The technique we use for proving that both rewritings preserves constructive consistency, consists in transforming proofs in R^{ad} and in R^{mg} into proofs in R.

Lemma 5.2
A proof P in R^{ad} induces a proof in R by replacing in P the adorned predicates by the corresponding non-adorned predicates. A proof P in R^{mg} is reduced into a proof in R^{ad} by pruning P from proofs of magic atoms.

Proposition 5.8
Let R be a set of rules and F a set of facts. If $R \cup F$ is constructively consistent then $R^{ad} \cup F$ and $R^{mg} \cup F$ are constructively consistent.

The third step of the Generalized Magic Sets procedure, namely the computation of the fixpoint of $R^{mg} \cup F$, can be performed by applying the conditional fixpoint procedure of Section 4. If R and therefore R^{mg} contains function symbols, the conditional fixpoint procedure as defined in [BRY 88a] must be applied.

In [BB* 88], Balbin, Meenakshi, Port, and Ramamohanarao have proposed to modify the Magic Sets rewriting in order to preserve stratification. They define a 'structured' bottom-up procedure applicable to stratified programs. Kerisit proposes a similar method in his PhD thesis [KER 88]. Kerisit's rewriting is simpler than the other one. It generates programs that are not always stratified but satisfy a condition called 'weak stratification'. Kerisit defines a 'layered' bottom-up procedure for weakly stratified programs. The modified rewritings defined in these reports do not seem to extend to non-stratified constructively consistent programs.

It is not clear if an approach always permits better performance than another on stratified programs. Because of its simplicity, the modified rewriting proposed by Kerisit seems to be preferable to the other one. The modified rewritings generate significantly more additional predicates than the Magic Sets rewriting. This certainly increases the complexity of the bottom-up evaluation. The bottom-up procedure can however make benefit from the weak stratification for not delaying the evaluation of negative premisses as long as the conditional fixpoint procedure does.

Other recursive query processing procedures extend to stratified programs as well. Kemp and Topor [KT 88], and independently Seki and Itoh [SI 88] have recently defined such extensions for the twin procedures OLD-resolution with tabulation [TS 86] and QSQR/SLD-resolution [VIE 87]. In [PRZ 89], these procedures have been further extended, relying on a concept of 'dynamic stratification', for processing all logic programs that have a well-founded model.

6. Conclusion

The purpose of this article was twofold. It was first to show that the features of logic programming that seem unconventional from the viewpoint of classical logic can be nicely explained in terms of constructivistic logic. This reading of logic programming is usually more intuitive to people not trained in formal logic. It provides logical foundations for features often considered purely procedural. The second purpose of this paper was to apply the constructivistic formalization of logic programming for establishing new results of practical interest.

We first recalled the complementary roles of model and proof theory for conveying the semantics of logic programs. Then, we showed how constructivism and logic programming are connected. We proposed a constructivistic axiomatic system, the Causal Predicate Calculus (CPC), as a proof-theoretic formalization of non-Horn programs. A bottom-up proof procedure, the conditional fixpoint procedure, was defined for CPC.

Next, we used this formalization of logic programming in order to establish practical results. First, we have given a simple and intuitive motivation for the concepts 'stratification' and 'local stratification': They are sufficient conditions of constructive consistency. Second, we introduced the notion 'constructive domain independence', which gives a logical explanation of the need to 'keep ordered' certain conjunctions in logic programs. It also constitutes a practical basis for introducing quantifiers into logic programs and queries. Finally, we showed how the concept of constructive proof and the conditional fixpoint procedure permit to extend the Magic Sets procedure to constructively consistent non-Horn logic programs.

Being independent from any proof procedure, the constructivistic formalization of logic programming should help in investigating various query evaluation techniques. It is indeed important to also investigate other evaluation strategies than the one of Prolog and SLDNF-resolution.

The constructivistic reading of logic programming seems promising for studying 'logical optimization' techniques. Roughly, we mean methods that translate queries or rules into equivalent expressions, on the basis of logical rules or of integrity constraints. The main problem encountered in defining such rewritings is to control the number of generated expressions. The constructivistic restriction of logical equivalence seems to correspond to useful rewritings. Finally, a constructivistic understanding of logic programming is surely applicable to the generation of intuitive explanations.

7. Acknowledgement

We would like to thank Hervé Gallaire and Jean-Marie Nicolas for their support. We are indebted to them and to Christoph Freytag, Rainer Manthey, and Mark Wallace for helpful comments on an earlier draft.

8. References

[A* 88] K.R. Apt, H.A. Blair, and A. Walker. Towards a theory of declarative knowledge. *Foundations of Deductive Databases and Logic Programming.* Morgan Kaufmann, Los Altos, CA, 1988, pages 89-148.

[BB* 88] I. Balbin, K. Meenakshi, G. Port, and K. Ramamohanarao. *Efficient Bottom-Up Computation of Queries on Stratified Databases.* Technical Report, University of Melbourne, Dep. of Computer Science, 1988.

[BC* 86] F. Bancilhon, D. Maier, Y. Sagiv, and J. Ullman. Magic sets and other strange ways to implement logic programs. In *Proc. 5th ACM SIGMOD-SIGACT Symposium on Principles of Database Systems.* 1986.

[BF 88] N. Bidoit and C. Froidevaux. More on stratified default theories. In *Proc. of the 8th European Conf. on Artificial Intelligence*, pages 492-494. Munich, West Germany, August, 1988.

[BIS 67] E. Bishop. *The Foundations of Constructive Analysis.* McGraw Hill, New York, 1967.

[BOJ 86] D. Bojadziev. A constructive view of Prolog. *Journal of Logic Programming* 3(1):69-74, 1986.

[BR 87] C. Beeri and R. Ramakrishnan. On the power of magic. In *Proc. 6th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 269-283. 1987.

[BRO 54] L.E.J. Brouwer. Points and spaces. *Canadian Journal of Mathematics* 6:1-17, 1954.

[BRY 88a] F. Bry. *Logic Programming as Constructivism: A Formalization and its Application to Databases.* Research Report IR-KB-58, ECRC, Dec., 1988. Full version of this paper.

[BRY 88b] F. Bry. *Logical Rewritings for Improving the Evaluation of Quantified Queries.* Research Report IR-KB-56, ECRC, Nov., 1988. Submitted for publication.

[CAL 79] A. Calder. Constructive mathematics. *Scientific American* 241(4):134-143, Oct., 1979.

[CH 85] A.K. Chandra and D. Harel. Horn clause queries and generalizations. *Journal of Logic Programming* 1(1):1-15, 1985.

[CHU 56] A. Church. *Introduction to Mathematical Logic, Vol. 1.* Princeton University Press, Princeton, NJ, 1956.

[CL 73] C.L. Chang and R.C.T. Lee. *Symbolic Logic and Mechanical Theorem Proving.* Academic Press, New York, 1973.

[CLA 78] K.L. Clark. Negation as failure. *Logic and Databases.* Plenum Press, New York, 1978, pages 293-322.

[DEM 82] R. Demolombe. *Syntactical Characterization of a Subset of Domain Independent Formulas.* Technical Report, ONERA-CERT, Toulouse, France, 1982.

[DF 87] P. Deransart and G. Ferrand. An operational formal definition of Prolog. In *Proc. Symp. on Logic Programming*, pages 162-172. San Francisco, CA, 1987.

[DIP 69] R.A. Di Paola. The recursive unsolvability of the decision problem for the class of definite formulas. *Journal of the ACM* 16(2):324-327, Apr., 1969.

[DP 60] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM* 7:201-215, 1960.

[FAG 80] R. Fagin. Horn clauses and database dependencies. In *Proc. 12th Ann. ACM Symp. on Theory of Computing*, pages 123-134. 1980.

[FIT 69] M. Fitting. *Intuitionistic Logic, Model Theory and Forcing.* North-Holland, Amsterdam, 1969.

[FIT 85] M. Fitting. A Kripke-Kleene semantics for logic programs. *Journal of Logic Programming* 4(4):295-312, 1985.

[GÖD 58] K. Gödel. Über eine bisher noch nicht benützte Erweiterung des finiten Standpunktes. *Dialectica* 12:280-287, 1958.

[GAB 85] D.M. Gabbay. N-Prolog: An extension of Prolog with hypothetical implication. II. Logical foundations, and negation as failure. *Journal of Logic Programming* 4(4):251-283, 1985.

[GAB 86] D.M. Gabbay. *Modal provability foundations for negation by failure.* preprint, Dpt of Computing, Imperial College of Sc. and Tech., 1986. Cited in [SHE 88].

[GR 84] D.M. Gabbay and U. Reyle. N-Prolog: An extension of Prolog with hypothetical reasoning. *Journal of Logic Programming* 4(4):318-355, 1984.

[GS 86] D.M. Gabbay and M.J. Sergot. Negation as inconsistency. *Journal of Logic Programming* 1(1):1-35, 1986.

[HEY 66] A. Heyting. *Intuitionism: An Introduction.* North-Holland, New York, 1966.

[HUE 80] G. Huet. Confluent reduction: Abstract properties and applications of term rewriting systems. *Journal of the ACM* 27(4):797-821, Oct., 1980.

[KER 88] J.-M. Kerisit. *La Méthode d'Alexandre: une Technique de Déduction.* PhD thesis, Université de Paris VII, June, 1988. See [KP 88] for a presentation in English.

[KP 88] J.-M. Kerisit and J.-M. Pugin. *Efficient Query Processing on Stratified Databases.* Technical Report, Bull research centre, Mai, 1988. Cited in [KER 88].

[KRE 65] G. Kreisel. Mathematical logic. *Lectures on Modern Mathematics - III.* Wiley, New York, 1965, pages 95-195.

[KT 88] D.B. Kemp and R.W. Topor. Completeness of a top-down query evaluation procedure for stratified databases. In *Proc. 5th Int. Conf. and Symp. on Logic Programming*, pages 179-194. 1988.

[KUH 67] J.L. Kuhns. *Answering Questions by Computers - A Logical Study.* Rand Memo RM 5428 PR, Rand Corp., Santa Monica, Calif., 1967.

[LEW 85] H.R. Lewis. *Cycles of Unifiability and Decidability by Resolution.* Research Report, Aiken Comp. Lab., Harvard Univ., Cambridge, Mass., 1985.

[LLO 84] J.W. Lloyd. *Foundations of Logic Programming.* Springer, Berlin, New York, 1984.

[LT 86] J.W. Lloyd and R.W. Topor. A basis for deductive database systems II. *Journal of Logic Programming* 3(1):55-67, 1986.

[NIC 81] J.-M. Nicolas. Logic for improving integrity checking in relational databases. *Acta Informatica* 18(3):227-253, Dec., 1981.

[PRA 65] D. Prawitz. *Natural Deduction, a Proof Theoretical Study.* Almqvist and Wiksell, Stokholm, 1965.

[PRZ 88a] T.C. Przymusinski. On the semantics of stratified deductive database and logic programs. *Journal of Logic Programming* , 1988. invited paper, to appear.

[PRZ 88b] T.C. Przymusinski. On the declarative semantics of deductive databases and logic programs. *Foundations of Deductive Databases and Logic Programming*. Morgan Kaufmann, Los Altos, CA, 1988, pages 193-216.

[PRZ 89] T.C. Przymusinski. Every logic program has a natural stratification and an iterated least fixed point model. In *Proc. 8th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems*. Philadelphia, Pennsylviana, March, 1989. In this book.

[QUI 70] W.V.O. Quine. *Philosophy of Logic*. Prentice-Hall, Englewood Cliffs, NJ, 1970. cited in [BOJ 86].

[R* 86] R. Rohmer, R. Lescoeur, and J.-M. Kerisit. The Alexander method, a technique for the processing of recursive axioms in deductive databases. *New Generation Computing* 4(3):273-285, 1986.

[SHE 88] J.C. Shepherdson. Negation in logic programming. *Foundations of Deductive Databases and Logic Programming*. Morgan Kaufmann, Los Altos, CA, 1988, pages 19-88.

[SI 88] H. Seki and H. Itoh. A query evaluation method for stratified programs under the extended CWA. In *Proc. 5th Int. Conf. and Symp. on Logic Programming*, pages 195-211. 1988.

[TRO 77] A.S. Troelstra. Aspects of constructive mathematics. *Hanbook of Mathematical Logic*. North-Holland, Amsterdam and New York, 1977, pages 973-1052.

[TS 86] H. Tamaki and T. Sato. OLD resolution with tabulation. In *Proc. 3rd Int. Conf. on Logic Programming*, pages 84-98. London, UK, 1986.

[ULL 80] J. Ullman. *Principle of Database Systems*. Computer Sc. Press, Rockville, MD, 1980.

[vEK 76] M. van Emden and R. Kowalski. The semantics of predicate logic as a programming language. *Journal of the ACM* 23(4):733-742, Oct., 1976.

[VGE 88] A. Van Gelder. Negation as failure using tight derivations for general logic programs. *Foundations of Deductive Databases and Logic Programming*. Morgan Kaufmann, Los Altos, CA, 1988.

[VGT 87] A. Van Gelder and R.W. Topor. Safety and correct translation of relational calculus formulas. In *Proc. 6th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 313-327. San Diego, CA, 1987.

[VIE 87] L. Vieille. A database-complete proof procedure based on SLD-resolution. In *Proc. 4th Int. Conf. on Logic Programming*, pages 74-103. Melbourne, 1987.

[VIE 88] L. Vieille. Unpublished note. Nov., 1988.