Theoretical Computer Science 78 (1991) 3-83
North-Holland

Negation in rule-based database languages: a survey

N. Bidoit*

U.A. 410 du CNRS, Laboratoire de Recherche en Informatique, Bat. 490, 91405 Orsay Cedex, France

Abstract

Bidoit, N., Negation in rule-based database languages: a survey, Theoretical Computer Science 78 (1991) 3-83.

This paper surveys and compares different techniques investigated in order to integrate negation in rule-based query languages. In the context of deductive databases, a rule-based query is a logic program. The survey focuses on the problem of defining the declarative semantics of logic programs with negation. The declarative semantics of logic programs with negation based on fixpoint techniques, based on three-valued logic and based on non-monotonic logics are presented for positive logic programs, (locally) stratifiable logic programs and unstratifiable logic programs. The expressive power of rule-based query languages is examined.

1. Introduction

During the last decades, fundamental work has been done in order to develop extensions of the theory of relational databases [36]. Undoubtedly, the success of the relational database model lies in its simplicity. Data are represented by elementary tables or collections of facts. Manipulation of data is performed by means of basic operations on tables like selecting rows of a table, selecting columns of a table, merging or combining two tables, ..., adding or deleting a row in a table. Although the simplicity of the relational model had led to the development of a real database technology, and commercial relational database management systems are, limitations of the relational theory have soon been recognized both at the level of data representation and at the level of data manipulation. Several directions have been followed, from the non-first normal form database model [1, 52, 100], to the object-oriented database model [88, 13, 12], through the semantic database model [58], each of them being developed to overcome the deficiencies of the relational theory.

* Partially supported by the PRC-BD3.

One of the major extensions investigated, the theory of deductive databases, has emerged, very early [54], from the use of the mathematical logic paradigm. Mathematical logic offers a precise and uniform formalism to study many database problems. The main point is that mathematical logic provides both a representation language and an inference mechanism. The historical development of deductive databases is quite interestingly presented in [86], and [55] provides a survey of the application of logic for studying query languages, integrity constraints, query optimization, data dependencies and database design in the context of conventional databases and deductive databases. Reiter [99] also presents a nice introduction to the domain.

The naive introductory definition of a deductive database usually given is that of a database in which new facts may be derived from facts that were explicitly introduced and from general laws also contained in the database. Indeed that definition is quite insufficient to help to distinguish a conventional database from a deductive database. The relational algebra provides a mechanism (the view mechanism) for deriving new facts from the facts stored in the database, doesn't it? Anyway, this definition at least reveals that, in the context of deductive databases, data are represented, like in the context of relational database, by collection of facts. In the first case, facts are elementary formulas, in the second case, facts are elements of a table. So now we ought to try to make clear the frontier between conventional databases and deductive databases. As a matter of fact, the frontier can be drawn at the level of data manipulation. A deductive database offers more powerful "deductive capabilities" (that is a more powerful data manipulation/view mechanism) than a conventional database.

For instance, it is rather well known that the transitive closure of a relation is not definable by a relational algebraic (or relational calculus) expression, although it is a very natural inference to make. Examples of transitive closure query ranges from the famous Ancestor query to more practical problems in graph theory. The logic approach to databases, through its inference mechanism, provides a direct solution which overcomes in an elegant manner the inability of the relational model to express transitive closure of relations.

Obviously, the contribution of the logical approach to database theory is not to be reduced to an increase of the expressive power of the query language, though it is actually its more visible feature.

Intuitively, at the syntactical level, a deductive database is specified by a set of simple first order formulas. The assumption that function symbols do not occur in a deductive database is usually made. A partition of the deductive database may be used to distinguish between relations explicitly defined and relations defined in terms of the first ones. The first relations are called extensional, the second ones are called intentional. This partition is essentially useful to present and study implementation issues of deductive database systems. Although implementation of deductive database systems is of prime importance, we shall not address this issue in the current paper. Thus in the following, a deductive database is viewed, at the

syntactical level, as a set of laws, some being elementary, that is facts, other being slightly more complex first order formulas.

This presentation of a deductive database already suggests a strong relationship between deductive databases and logic programming. Indeed, foundations of deductive databases and logic programming are closely related [35, 80, 81]. Also, the reader should not be surprised about the fact that we use equally the terms "deductive database" and "logic program" (even though the use of logic program is abusive because of the implicit restriction to function free programs).

The application of mathematical logic tools to characterize the relations specified by a deductive database (or to define the declarative semantics of logic programs) is not straightforward.

Two first (minor) features of logic databases are identified as the *unique name assumption* which states that individuals with different names are different, and the *domain close axiom* which states that there are no other individuals than those in the database. The third (major) feature of logic databases lies in the so-called *Closed World Assumption* and roughly speaking concerns the way negation is treated.

Making the Closed World Assumption corresponds to the choice of an *incomplete* representation of real world situations; only positive (true) information is specified. From a database point of view, this choice can be motivated by "common sense" and performance. For instance in order to define a property, it seems more natural to give the list of individuals satisfying the property than to give both the list of entities satisfying the property and the list of entities that fails to satisfy the property. Thus the Closed World Assumption simplifies tremendously the representation of data. Now, while data description is incomplete, the Closed World Assumption entails a second fundamental principle; complete knowledge of the world situation described is assumed. Intuitively, this means that, although the definition of a property is limited to the list of individuals satisfying it, one should be able to say for any individual whether a property holds.

In the context of relational databases, the Closed World Assumption comes for free (although one should pay attention to write *safe* calculus expressions). From a mathematical logic point of view, it is probably unnecessary to recall that first order logic does not allow one to infer negative facts from a set of positive facts for example. Thus extralogical mechanisms need to be introduced in order to treat "database negation".

The Closed World Assumption determines a major distinction between mathematical logic and logic databases. While first order logic is monotonic (that is adding a formula to a theory has the effect of strictly increasing the set of formulas that can be inferred), logic databases are non-monotonic. Adding a new positive fact to the representation of a logic database has a "side effect": it entails that the negation of this fact cannot be inferred from the database any more.

A very simple case of deductive databases is Horn databases specified by sets of definitie Horn clauses. Roughly speaking, a definite Horn clause is a conditional definition of a single property (relation) whose conditional part is a conjunction of

elementary positive conditions. Horn databases (or positive logic programs) are well understood from a declarative point of view, from a procedural point of view and from a computational point of view.

The relations specified by a Horn database (the declarative semantics of positive logic programs) have been characterized in at least three different ways. From a model theoretic point of view, the deductive database specified by a set of Horn clauses is described by a particular model of the formulas, the minimal Herbrand model [44, 4]. From a proof theoretic point of view, it is described by the theorems derivable from the formulas (and the negation of the sentences not derivable from the formulas) [97]. From an operational point of view, it is given by the least fixpoint of some operator associated with the formulas [44, 4].

Going back to the expressive power issue, positive logic programs allow one to define the transitive closure of a relation (in a way which is close to natural language). However, simple relations like the complement of a relation with respect to another one, are not definable by Horn databases although they are definable by the relational algebra. The idea to introduce negation in the "conditional part" of the formulas specifying intentional relations in order to overcome this deficiency appears to be natural and simple, at first. Nonetheless, extending Horn database (or positive logic program) with negation happens to be not such an easy task.

One of the major problems arising is to characterize the relations intentionally defined by logic programs with negation, or in other words, to define the declarative semantics of general logic programs. The difficulty is due to the implicit extralogical use of negation which makes mathematical logic less a convenient formalism. The extralogical use of negation makes us prefer to call *rules*, the formulas which constitute a logic program.

Declarative semantics gives the meaning of a program in terms of properties and does not involve computation as opposed to procedural semantics which gives the meaning of a program in terms of the execution or evaluation of the program. In other words, declarative semantics is used to formalize *what* we want while procedural semantics is more concerned with *how* to compute it [60]. Providing a declarative semantics of logic programs is obviously of prime importance especially from the database point of view since declarative database query languages is a major issue.

Much recent work has been devoted to incorporate negation in deductive databases and logic programs and various approaches have been proposed from the database community, the logic programming community, as well as the artificial intelligence community (e.g., [3, 94, 74, 15, 115, 8, 66, 53, 47, 69]). This paper is an attempt to present the major solutions proposed to the problem and to compare them. We try to consider the following three criteria.

The first criteria concerns the ability of the declarative semantics of a logic program to reflect its common sense or intended meaning. We should confess that this criteria is a rather fuzzy one.

The second criteria concerns the computational issue. As pointed out in [104], if the only reason for abandoning the clear and well-known concept of classical

negation is the inefficiency of its implementation, it might not be unreasonable to ask for computational tractability of the declarative semantics of logic programs.

Finally, the third criteria is obviously the expressive power of the languages characterized by a given semantics.

1.1. Organization of the paper

The paper consists of a further nine sections. Section 2 contains preliminaries. It gives a brief presentation of basic concepts and notation of first order logic, a syntactical description of logic programs and a minimal set of results on fixpoint theory.

Section 3 can also be considered as a preliminary section and it includes a review of the two major ways to define the declarative semantics of positive logic programs. The presentation of the minimal model semantics and the least fixpoint semantics of positive logic programs is followed by a discussion whose aim is to show some of the problems that arise when negation is introduced and to give a flavor of the different approaches that are later examined.

Sections 4, 5 and 6 are devoted to approaches based on fixpoint techniques.

Stratifiable programs, that is programs in which recursive negation is ruled out, are presented in Section 4. The declarative semantics of stratifiable logic programs is defined in this section by means of iterative fixpoint [3]. Two weaker constraints, local stratification [94] and loose stratification [26], are examined in this section.

The restriction that programs should not contain recursive negation is totally relaxed in Section 5 where however the declarative semantics of a logic program is not forced to "tell everything" about the contents of the relations intentionally defined and also about the contents of the complement of these relations. The well-founded semantics of a logic program is presented in Section 5 [115]. Effective stratification is briefly presented that gives a sufficient condition for logic program to have a "fully" defined meaning with respect to the well-founded semantics [17, 96].

The presentation of inflationary semantics of logic programs is included in Section 6 [8, 66], which does not assume any constraint on the syntax of logic programs.

Sections 7 and 8 are both dedicated to "model theoretic" definitions of the declarative semantics of logic programs. The semantics presented in these sections are based on various forms of non-monotonic logic such as circumscription [83, 73], autoepistemic logic and default logic [98].

Section 7 focuses on the alternative model theoretic definitions of the iterative fixpoint semantics for stratifiable logic programs. In the presentation of these alternative definitions, a particular emphasis is given to the exposition of the perfect model approach [94] and its relationship with circumscription [74].

The exposition of the contribution of non-monotonic logic to define the declarative semantics of logic programs is continued in Section 8 where default logic and autoepistemic logic are shown to provide a very appealing formalism to define the declarative semantics of logic programs [14, 17, 51, 53].

A discussion on the expressive power of the query languages defined by the various semantics reviewed in the paper is carried on in Section 9. While most researcher's attention has been concentrated on the "natural" aspect of the meaning assigned to a logic program, less attention has been paid in general to the expressive power issue (with the exception [8, 63]). Recent work in this area is reported in Section 9.

Finally, Section 10 presents a brief discussion on the aspects of logic programming not developed in the paper. Essentially, Clark's completion approach is discussed, then a quick review of the procedural semantics of logic programs with negation is provided and finally, extensions of positive logic programs not necessarily involving the introduction of negation are suggested.

2. Preliminaries

In the following, we assume that the reader is familiar with symbolic logic and more precisely with propositional logic and first order logic [45, 50]. We also assume that the reader is familiar with notions such as complete lattice, monotonic mapping and fixpoint [79, 75, 108]. However, in order to make the discussion clear, we begin by reviewing some well-known concepts of first order logic and logic programming as well as some elementary results on fixpoint theory. The main notations used throughout the paper are presented in this section.

2.1. First order logic and logic programming

Syntax

Usually, logic programs are syntactically defined as sets of first order formulas, commonly as sets of Horn clauses. However, as the contents of the paper will show, as soon as negation is introduced in logic programs, the semantics associated with the formulas in programs is (more or less) far from first order semantics.

For the sake of clarity (and rigor) and despite some notational overload, we choose here to distinguish logic programming syntax and first order syntax. Roughly speaking, this distinction is made by using the non logical symbols :-, &, *or*, *not* for logic programming languages instead of the first order connectives $\leftarrow$, $\wedge$, $\vee$, $\neg$.

In the following, f.o. is used as an abbreviation for first order and l.p. is used as an abbreviation for logic programming.

A *first order logic* (respectively, *logic programming*) *alphabet* consists of five (respectively, four) classes of symbols (Table 1).

An alphabet is characterized by its set of function symbols, denoted *Fun*, and its set of predicate symbols, denoted *Pred*, the rest of the syntactic symbols being common to all f.o. (resp. l.p.) alphabets. The 0-place function symbols are called *constants*, denoted by $a, b, c \ldots$. The 0-ary predicate symbols are called *propositions*, denoted by $A, B, C \ldots$.

The notion of l.p. *term* (respectively, l.p. *atomic formula*) is identical to the notion of f.o. term (respectively, f.o. atomic formula) and is defined in the standard way.

Table 1

Classes of symbols		f.o. alphabet	l.p. alphabet	Notations
Variables		Infinitely many	Infinitely many	$x\ y\ z$
Functions	0-place n-place	Possibly empty	Non-empty	abc fgh
Predicates	0-ary n-ary	Non-empty	Non-empty	ABC PQR
Connectives		$\leftarrow, \wedge, \vee, \neg$	:-, &, or, not	
Quantifiers		$\forall, \exists$		

A *ground term* (respectively, a *ground atomic formula*) is a term (respectively, an atomic formula) containing no variable symbols.

A f.o. *literal* (respectively, l.p. *literal*) L is either an atomic formula ($P(t)$) and it is then called a positive literal or the negation of an atomic formula ($\neg P(t)$, respectively not $P(t)$) and it is then called a negative literal. A *fact* is a ground literal, a *positive fact* is a ground positive literal and a *negative fact* is a ground negative literal.

For the sake of simplicity, given a set S of f.o. literals (respectively, of l.p. literals) its corresponding set of l.p. literals (respectively, of f.o. literals) is denoted by S itself. The context always makes clear whether a set S of literals is f.o. or l.p. Now, given a set S of literals, pos(S) denotes the subset of positive literals in S and neg(S) denotes the set of negative literals in S. On the other hand, $\neg.S$ denotes $\{\neg L \mid L \in \text{pos}(S)\} \cup \{L \mid \neg L \in \text{neg}(S)\}$ or $\{\text{not } L \mid L \in \text{pos}(S)\} \cup \{L \mid \text{not } L \in \text{neg}(S)\}$.

A *first order language* over a f.o. alphabet (*Fun*, *Pred*) consists of the set of well-formed formulas constructed from the alphabet (*Fun*, *Pred*) in the usual manner.

A *clause* c is a universally quantified well-formed formula of the form $L_1 \vee \cdots \vee L_n$ where the L_i are f.o. literals. If the number of positive literals in the clause c is less or equal to 1, then c is called a *Horn clause*. If the number of positive literals in the clause c is 1, then c is called a *definite clause*.

Given a l.p. alphabet (*Fun*, *Pred*), a *program rule* r is an expression of the form L :- $L_1 \,\&\, \cdots \,\&\, L_n$ with $n \geq 0$, where L is a positive literal and the L_i are l.p. literals, for $i = 1, \ldots, n$.

The literal L is called the *head* of the program rule r and it is denoted by head(r). $L_1 \,\&\, \cdots \,\&\, L_n$ is called the *body* of the rule and body(r) denotes the set of literals $\{L_1, \ldots, L_n\}$. The elements of body(r) are also called *premises* of the rule r.

If $n = 0$, the rule r is written L and called a *unit rule*. Now, if all the premises of r are positive that is if body(r) = pos(body(r)), then r is called a *positive rule*.

A *logic programming language* is the set of program rules constructed from the alphabet (*Fun*, *Pred*). A *logic program* is a set of program rules. In the paper, except where otherwise specified, we assume that logic programs are finite set of rules, and

we assume that the only function symbols in *Fun* are constant symbols. A *positive logic program* is a set of positive program rules.

Given a logic program $\mathscr{P}$, the subset of the rules in $\mathscr{P}$ in which the predicate symbol P occurs in the head is called the *definition of P in* $\mathscr{P}$ and denoted by def(P, $\mathscr{P}$).

The closed instantiation of a program rule r is obtained by substitution of each variable occurring in r by a ground term (or by an element of the Herbrand universe, as explained below). The *instantiation of a logic program* $\mathscr{P}$, denoted *Inst_*$\mathscr{P}$, is the collection of all possible instantiations of each rule in $\mathscr{P}$.

For the purpose of the presentation, we need to associate with a logic program $\mathscr{P}$ a set of f.o. formulas. The *first order notation of a program* $\mathscr{P}$, denoted by $\mathscr{P}_{\text{f.o}}$, is simply the set of first order formulas obtained by replacing the connectives :-, &, not by $\leftarrow$, $\wedge$, $\neg$ in the rules of $\mathscr{P}$.

Finally, in the discussion, the following transformation of logic programs is frequently used. This transformation is similar to the *Davis-Putnam transformation* of a set of clauses [43]. Let $\mathscr{P}$ be a logic program and S be a set of ground atomic literals. Intuitively, DP($\mathscr{P}$, S) is the instantiated program obtained as follows:

(1) remove from *Inst_*$\mathscr{P}$ all the rules having a premise which is in contradiction with S, and

(2) remove from the rules in *Inst_*$\mathscr{P}$ all the premises that belong to S.

Formally, DP($\mathscr{P}$, S) is defined by:

(1) first let $\mathscr{P}_1 = \textit{Inst\_}\mathscr{P} - \{r \mid r \in \textit{Inst\_}\mathscr{P}$ and $\exists L \in \text{body}(r)$, $L \in \neg.S\}$, then

(2) $\text{DP}(\mathscr{P}, S) = \{r \mid r' \in \mathscr{P}_1$, $\text{head}(r) = \text{head}(r')$ and $\text{body}(r) = \text{body}(r') - S\}$.

It may be convenient to represent a logic program by a graph. The *precedence graph* (V, E) associated with a logic program $\mathscr{P}$ is such that:

(1) the set V of vertices is the set of predicate symbols, and

(2) there is one edge from Q to P for each occurrence of Q in a premise of a rule r in $\mathscr{P}$ with head P.

It may be convenient to label the edges of the precedence graph associated with $\mathscr{P}$ in such a way that an edge from Q to P induced by a rule r of $\mathscr{P}$ is positive if Q occurs in a positive premise of r and negative otherwise.

Semantics

The semantics of a logic program is usually defined by means of particular models of the f.o. notation of the program. We recall below some well-known notions used to define the semantics of first order logic. The presentation essentially concentrates on Herbrand interpretations.

Let $\mathscr{A} = (\textit{Fun}, \textit{Pred})$ be a f.o. language. The Herbrand universe of $\mathscr{A}$ is the set of ground atomic terms constructed from the function symbols in *Fun*. The *Herbrand base* $\mathscr{B}_{\mathscr{A}}$ of $\mathscr{A}$ is the set of ground atomic formulas constructed from the ground terms in the Herbrand universe of $\mathscr{A}$ and from the predicate symbols in *Pred*. When the language is understood, the Herbrand base is just denoted by $\mathscr{B}$.

For Herbrand interpretations, the domain is the Herbrand universe and the assignment of functions is (roughly speaking) the identity. Thus a *Herbrand interpre-*

tation can be simply represented by a subset of the Herbrand base. Herbrand interpretations represented by subset of the Herbrand base are denoted by bold possibly subscripted I ($\boldsymbol{I}, \boldsymbol{I}_1, \boldsymbol{I}_2 \ldots$).

For the sake of the discussion, we need to introduce another representation of Herbrand interpretation. Intuitively, an Herbrand interpretation I as represented above gives the set of ground atomic formulas true for I. Implicitly, ground atomic formulas not in I are false for I. An alternative way to represent an Herbrand interpretation I is to give both ground atomic formulas true for I and the negation of ground atomic formulas false for I. Thus a Herbrand interpretation can also be represented by a subset of $\mathscr{B} \cup \neg.\mathscr{B}$. In the following, in order to distinguish the first representation from the second one, we call Herbrand interpretation an interpretation represented by a subset of $\mathscr{B}$ and *completed Herbrand interpretation* an interpretation represented by a maximally consistent subset of $\mathscr{B} \cup \neg.\mathscr{B}$. It should be clear for the reader that the subset $\boldsymbol{I}$ of $\mathscr{B}$ and the subset $\boldsymbol{I} \cup \neg.(\mathscr{B} - \boldsymbol{I})$ of $\mathscr{B} \cup \neg.\mathscr{B}$ are two distinct notations for the same Herbrand interpretation. Completed interpretations are denoted by cursive, possibly subscripted I ($\mathscr{I}, \mathscr{I}_1, \mathscr{I}_2, \ldots$).

The set of all possible Herbrand interpretations (i.e. $2^{\mathscr{B}}$) is denoted by **Int** and the set of all possible completed Herbrand interpretations (i.e. the subset of complete and consistent sets in $2^{\mathscr{B} \cup \neg.\mathscr{B}}$) is denoted *C_Int*.

The truth of a formula for an interpretation I and the notion of Herbrand model of a set of formulas are defined in a standard way. $I \models f$ means I satisfies f. Models are denoted by bold/cursive, possibly subscripted $\boldsymbol{M}$ ($\boldsymbol{M}, \boldsymbol{M}_1, \boldsymbol{M}_2, \ldots, \mathscr{M}, \mathscr{M}_1, \mathscr{M}_2, \ldots$).

A Herbrand partial interpretation is a partial truth valuation of the ground atomic formulas in the Herbrand base. Thus a *Herbrand partial interpretation* $\mathscr{I}$ is represented as a consistent subset of $\mathscr{B} \cup \neg.\mathscr{B}$. In fact, a partial interpretation $\mathscr{I}$ can be viewed as a three-valued logic interpretation in the following way: if a ground atomic formula belongs to $\text{pos}(\mathscr{I})$, its truth value is true; if a ground atomic formula belongs to $\neg.\text{neg}(\mathscr{I})$, its truth value is false; and otherwise its truth value is undetermined. We use the same notational convention for a partial interpretation as for a completed interpretation. The set of all possible partial Herbrand interpretations (i.e. the subset of consistent sets in $2^{\mathscr{B} \cup \neg.\mathscr{B}}$) is denoted *Partial*. Note that completed interpretations are special cases of partial interpretations, i.e. that we have: $C\_Int \subseteq Partial$.

2.2. Complete lattice, monotonic mappings and fixpoints

Let S be a set and R be a binary relation on S. (S, R) is a *complete lattice* if R is a *partial order* and if the *least upper bound* of X, denoted by $\text{lub}(X)$, and the *greatest lower bound* of X, denoted by $\text{glb}(X)$, exist for each subset X of S.

Assume from now on that $(S, \leqslant)$ is a complete lattice. Let $T: S \to S$ be a mapping. The mapping T is *monotonic* iff $s \leqslant s'$ entails $T(s) \leqslant T(s')$ for each pair s, s' in S. The mapping T is *continuous* if $T(\text{lub}(X)) = \text{lub}(T(X))$ for each directed subset X where X is directed if every finite subset of X has an upper bound in X.

Let $s \in S$, s is a *fixpoint* of T iff $T(s) = s$. An element s of S is a *least fixpoint* of T, denoted lfp(T), iff s is a fixpoint of T and $\forall s' \in S,\ T(s') = s' \Rightarrow s \leqslant s'$. In a symmetrical manner an element s of S is a *greatest fixpoint* of T, denoted gfp(T), iff s is a fixpoint of T and $\forall s' \in S,\ T(s') = s' \Rightarrow s' \leqslant s$.

The following result has been established by Knaster and Tarski:

> If $(S, \leqslant)$ is a complete lattice and T is a monotonic mapping defined on S then T has a least fixpoint and a greatest fixpoint.

The transfinite sequences $T\uparrow\alpha$ and $T\downarrow\alpha$ associated with the lattice $(S, \leqslant)$ and the mapping T are defined by:

(1) $T\uparrow 0 = T(\bot)$ where $\bot$ is the greatest lower bound of S,
$T\uparrow\alpha = T(T\uparrow\alpha - 1)$ for α ordinal successor, and
$T\uparrow\alpha = \text{lub}(\{T\uparrow\beta \mid \beta < \alpha\})$ for α a limit ordinal.

(2) $T\downarrow 0 = T(\top)$ where $\top$ is the least upper bound of S,
$T\downarrow\alpha = T(T\downarrow\alpha - 1)$ for α ordinal successor, and
$T\downarrow\alpha = \text{glb}(\{T\downarrow\beta \mid \beta < \alpha\})$ for α a limit ordinal.

Another interesting result follows:

> If T is monotonic then $\text{lfp}(T) = T\uparrow\alpha$ for some ordinal α and $\text{gfp}(T) = T\downarrow\alpha$ for some ordinal α.

The least ordinal α such that $\text{lfp}(T) = T\uparrow\alpha$ is called the closure ordinal of T. Now we also have that:

> If T is continuous, the closure ordinal of T is below[2] ω.

The analogous result does not hold for the least ordinal α such that $\text{gfp}(T) = T\downarrow\alpha$.

Note that the set of Herbrand interpretations **Int** with set inclusion $\subseteq$ is a complete lattice. The set of completed interpretations *C_Int* with the partial order $\leqslant$ induced by the inclusion of the positive part of interpretation (i.e. $\mathcal{I} \leqslant \mathcal{J}$ iff $\text{pos}(\mathcal{I}) \subseteq \text{pos}(\mathcal{J})$) is isomorphic to (**Int**, $\subseteq$) and thus is a complete lattice.

The set of partial interpretations *Partial* together with set inclusion is a semi-complete lattice (only directed subsets of *Partial* have a glb). Let *Inconst* be a new symbol that is intuitively meant to represent all inconsistent sets of literals or in other words the complement of *Partial* in $2^{\mathcal{B} \cup \neg \mathcal{B}}$. Now consider the set *Partial* $\cup$ {*Inconst*} with the partial order $\leqslant$ defined as:

> set inclusion $\subseteq$ on *Partial*, and by $\mathcal{I} \leqslant$ *Inconst* for each $\mathcal{I}$ in *Partial*.

Then, (*Partial* $\cup$ {*Inconst*}, $\leqslant$) is a complete lattice. Intuitively, adding *Inconst* to *Partial* serves to get a top element. In the following, when saying that *Partial* is a complete lattice, we mean (*Partial* $\cup$ {*Inconst*}, $\leqslant$) is a complete lattice.

[2] By "below", we mean here less than or equal to.

3. Declarative semantics of positive logic programs

The purpose of this section is twofold. First, we briefly review the model theoretic and the fixpoint semantics of positive logic programs. Secondly, we present the problems that arise when one tries to directly extend the definitions of the declarative semantics proposed for positive programs to logic programs with negation allowed in the body of rules. These problems are described for both the model theoretic approach and the fixpoint approach. The presentation also serves to motivate the different proposals further reviewed in the paper.

3.1. Declarative semantics of positive logic programs

Minimal model semantics

The model theoretic and the fixpoint semantics of positive logic programs have been presented and extensively discussed in the literature [4, 75, 44]. A nice discussion on these semantics can be found in [60]. Other equivalent approaches to define the declarative semantics of positive logic programs such as [97], based on the closed world inference rule [72], based on circumscription, and such as [18, 19], based on default logic are not presented here.

Following the model theoretic approach, a logic program is viewed as a first order formula and its meaning is captured by the set of atomic ground formulas that are logical consequences of the program. In other words, the meaning of a logic program is captured by the set of ground atomic formulas (facts) true in all models of (the f.o. notation of) the program. Particular models are considered, namely Herbrand models.

In the following, by a model it is always meant a Herbrand model (model = Herbrand model). As a matter of fact, most of the discussion that follows would be incorrect for general models.

The class of programs considered here, positive programs, leads to a simple and nice characterization of the ground atomic formulas that belong to all models of a program. The first order notation $\mathscr{P}_{\text{f.o.}}$ of a positive logic program $\mathscr{P}$ is equivalent to a set of definite clauses. For a set $\mathscr{P}_{\text{f.o}}$ of definite clauses, [44] shows that the intersection of all models of $\mathscr{P}_{\text{f.o.}}$ is a model of $\mathscr{P}_{\text{f.o.}}$. Note that it is not in general true that the intersection of the models of a set of well-formed formulas is a model of this set of formulas.

Thus following the model theoretic approach, the meaning of a positive program is captured by the least model (intersection of all models) of its f.o. notation.

For the sake of further discussion, we introduce the notion of minimal model. A comprehensive study of minimal models is provided in [27].

Definition 3.1 (*Minimal Herbrand model*). Let $\mathscr{F}$ be a set of f.o. formulas. A Herbrand interpretation $\boldsymbol{M}$ is a minimal model of $\mathscr{F}$ iff $\boldsymbol{M}$ is a model of $\mathscr{F}$ and for each $\boldsymbol{M}'$ such that $\boldsymbol{M}'$ is a model of $\mathscr{F}$, $\boldsymbol{M}' \subseteq \boldsymbol{M}$ entails $\boldsymbol{M} = \boldsymbol{M}'$.

The notion of a minimal model is more general than the notion of "intersection of models" although these two notions coincide for sets of definite clauses. Given a set $\mathscr{F}$ of formulas, the intersection of all models of $\mathscr{F}$ may not be a model of $\mathscr{F}$; however, $\mathscr{F}$ may have minimal models.

Consider for example the clause $A \vee B$. It has two minimal models: the first one contains A and does not contain B; the other one contains B and does not contain A. The intersection of all models of the clause $A \vee B$ is the empty interpretation which is not a model of $A \vee B$.

Theorem 3.2 (Van Emden and Kowalski [44]). *If $\mathscr{F}$ is a set of definite clauses then $\mathscr{F}$ has a unique minimal model, or equivalently, the intersection of all models of $\mathscr{F}$ is a model of $\mathscr{F}$.*

The declarative semantics of positive logic programs is simply defined.

Definition 3.3 (*Minimal model semantics of positive logic programs*). Let $\mathscr{P}$ be a positive logic program. The canonical model (the declarative semantics) of $\mathscr{P}$ is the minimal Herbrand model of $\mathscr{P}_{\text{f.o.}}$.

The definition is illustrated by the program presented in the introduction and used to represent a graph and its transitive closure.

Example 3.4. Let us assume that we want to represent a graph

(1) with vertex a, b and c;

(2) with arcs between a and b, b and a, c and a; and

(3) the transitive closure of the graph.

We need to consider the alphabet consisting of the constants a, b, and c, the binary predicates Arc and tc_Arc. We intend to specify the above data by the following positive logic program:

$$\text{Graph} = \left\{ \begin{array}{l} \text{Arc}(a, b), \\ \text{Arc}(b, a), \\ \text{Arc}(c, a), \\ \text{tc\_Arc}(x, y) \text{ :- } \text{Arc}(x, y), \\ \text{tc\_Arc}(x, y) \text{ :- } \text{Arc}(x, z) \,\&\, \text{tc\_Arc}(z, y). \end{array} \right\}$$

For instance, the Herbrand interpretation containing all elements of the Herbrand base, that is the interpretation given by

$$\left\{ \begin{array}{l} \text{Arc}(a, a), \text{Arc}(a, b), \text{Arc}(a, c), \\ \text{Arc}(b, a), \ldots, \\ \vdots \\ \text{tc\_Arc}(c, a), \text{tc\_Arc}(c, b), \text{tc\_Arc}(c, c) \end{array} \right\}$$

is a model of (the f.o. notation of) Graph. Obviously, it is not a minimal model. The declarative semantics of Graph is given by its unique minimal model, namely:

$$M_{\text{Graph}} = \left\{ \begin{array}{l} \text{Arc}(a, b),\ \text{Arc}(b, a),\ \text{Arc}(c, a), \\ \text{tc\_Arc}(a, b),\ \text{tc\_Arc}(b, a),\ \text{tc\_Arc}(c, a), \\ \text{tc\_Arc}(a, a),\ \text{tc\_Arc}(b, b),\ \text{tc\_Arc}(c, b). \end{array} \right\}$$

We should emphasize here, as it is done in [60], that the least model semantics of positive logic programs is an alternative formulation of the closed world assumption [97]. Given a logic program $\mathscr{P}$, we abusively say that

- A formula is true for $\mathscr{P}$ iff this formula is satisfied by the minimal model of (the f.o. notation of) $\mathscr{P}$. For instance, a positive fact is true for $\mathscr{P}$ iff it simply belongs to the minimal model of $\mathscr{P}$.
- A formula is false for $\mathscr{P}$ iff this formula is not satisfied by the minimal model of $\mathscr{P}$. For instance, a positive fact is false for $\mathscr{P}$ iff it does not belong to the minimal model of $\mathscr{P}$.

An interesting correspondence does exist between the set of positive facts true for a positive logic program $\mathscr{P}$ and the set of positive facts which are logical consequences of the f.o. notation $\mathscr{P}_{\text{f.o.}}$ of $\mathscr{P}$:

(1) A positive fact A is true for $\mathscr{P}$ iff A is a logical consequence of $\mathscr{P}_{\text{f.o.}}$ ($\mathscr{P}_{\text{f.o}} \models A$).

This correspondence allows one to make use of resolution techniques in order to evaluate positive queries.

The same relationship *does not* hold for negative facts true for $\mathscr{P}$. A negative fact $\neg A$ is true for $\mathscr{P}$ iff $\neg A$ is satisfied by the minimal model of $\mathscr{P}_{\text{f.o}}$, that is, iff A does not belong to the minimal model of $\mathscr{P}_{\text{f.o.}}$. Thus:

(2) A negative fact $\neg A$ is true for $\mathscr{P}$ iff A *is not* a logical consequence of $\mathscr{P}_{\text{f.o.}}$ ($\mathscr{P}_{\text{f.o.}} \nvdash A$).

It is well-known that $\mathscr{P}_{\text{f.o.}} \nvdash A$ does not entail $\mathscr{P}_{\text{f.o.}} \models \neg A$. Indeed, $\mathscr{P}_{\text{f.o}} \nvdash A$ as soon as one of the models of $\mathscr{P}_{\text{f.o.}}$ satisfies $\neg A$ while it is necessary that all models of $\mathscr{P}_{\text{f.o}}$ satisfy $\neg A$ in order to conclude that $\mathscr{P}_{\text{f.o.}} \models \neg A$.

Of course, the set of negative facts that are logical consequences of $\mathscr{P}_{\text{f.o.}}$ is included in the set of negative facts true for $\mathscr{P}$, that is, $\mathscr{P}_{\text{f.o}} \models \neg A$ entails $\mathscr{P}_{\text{f.o.}} \nvdash A$. However note that since $\mathscr{P}$ is a positive logic program and thus $\mathscr{P}_{\text{f.o.}}$ is equivalent to a set of definite clauses, the set of negative facts implied by $\mathscr{P}_{\text{f.o.}}$ is *empty*.

Clearly, (2) is the model theoretic formulation of the Closed World Assumption. In [97], the Closed World Assumption is formalized by $\mathscr{P}$ infers $\neg A$ iff $\mathscr{P}_{\text{f.o}} \nvdash A$.

Fixpoint semantics

The fixpoint semantics is based on a somewhat different and more operational view of logic programs. The program is viewed as a set of rules and (positive) facts together with some basic operation for applying rules to facts in order to generate new facts. The semantics of a program is given by means of the facts obtained by iterative application of the rules of the program to facts, starting with an empty set of facts [60].

The key point here is to define what is meant by "apply rules to facts", and make sure that, roughly speaking, the iterative application of this basic operation terminates.

For positive logic programs "apply rules to facts" is defined by means of an operator, called immediate consequence operator, associated with the program. Given a set of positive facts, the immediate consequence operator simply generates the heads of the rules whose bodies are *satisfied* by the given set of positive facts.

Because the programs considered are positive programs, the immediate consequence operator is monotonic (and continuous). This ensures (a) the existence of a least fixpoint, (b) the termination of the iterative application of the operator and (c) the correspondence between the least fixpoint and the set of facts obtained by iterative application of the operator. Formally, we have:

Definition 3.5 (*Immediate consequence operator*). Let $\mathscr{P}$ be a logic program. The immediate consequence operator associated with $\mathscr{P}$, denoted $T^{\models}_{\mathscr{P}}$, is the mapping on **Int** defined by

$$T^{\models}_{\mathscr{P}}(I) = \{\text{head}(r) \mid r \in \mathit{Inst\_}\mathscr{P} \text{ and } \forall L \in \text{body}(r)\, I \models L\}, \text{ for } I \in \mathbf{Int}.$$

The immediate consequence operator associated with a positive logic program satisfies the following property.

Theorem 3.6 (Van Emden and Kowalski [44], Apt and Van Emden [4]). *If $\mathscr{P}$ is a positive logic program then $T^{\models}_{\mathscr{P}}$ is monotonic and continuous.*

The above result entails (see preliminaries) that
(1) $T^{\models}_{\mathscr{P}}$ has a least fixpoint $\text{lfp}(T^{\models}_{\mathscr{P}})$,
(2) $\text{lfp}(T^{\models}_{\mathscr{P}})$ is equal to $T^{\models}_{\mathscr{P}}\uparrow\alpha$ for some ordinal α, and
(3) the closure ordinal (the ordinal α such that $\text{lfp}(T^{\models}_{\mathscr{P}}) = T^{\models}_{\mathscr{P}}\uparrow\alpha$) is below ω.
The fixpoint semantics of positive logic programs is defined by:

Definition 3.7 (*Least fixpoint semantics of positive logic program*). Let $\mathscr{P}$ be a positive logic program. The fixpoint semantics of $\mathscr{P}$ is the least fixpoint of $T^{\models}_{\mathscr{P}}$ or equivalently, the limit of the sequence $T^{\models}_{\mathscr{P}}\uparrow\alpha$.

Let us now illustrate the fixpoint approach using the Graph example.

Example 3.4 (*continued*). Let us consider the logic program Graph and apply iteratively the immediate consequence operator associated with it.

(1) First iteration

$$T^{\models}_{\text{Graph}}\uparrow 0 = T^{\models}_{\text{Graph}}(\emptyset) = \{\text{Arc}(a, b),\ \text{Arc}(b, a),\ \text{Arc}(c, a)\}.$$

Note that $T^{\models}_{\text{Graph}}\uparrow 0$ is equal to the set of positive facts in the logic program Graph.

(2) Second iteration

$$T^{\models}_{\text{Graph}}\uparrow 1 = T^{\models}_{\text{Graph}}(T^{\models}_{\text{Graph}}\uparrow 0)$$

$$= \left\{ \begin{array}{l} \text{Arc}(a, b),\ \text{Arc}(b, a),\ \text{Arc}(c, a), \\ \text{tc\_Arc}(a, b),\ \text{tc\_Arc}(b, a),\ \text{tc\_Arc}(c, a). \end{array} \right\}$$

Note that $T^{\models}_{\text{Graph}}\uparrow 1 = T^{\models}_{\text{Graph}}\uparrow 0 \cup \{\text{tc\_Arc}(a, b), \text{tc\_Arc}(b, a), \text{tc\_Arc}(c, a)\}$.

(3) Third iteration

$$T^{\models}_{\text{Graph}}\uparrow 2 = T^{\models}_{\text{Graph}}(T^{\models}_{\text{Graph}}\uparrow 1)$$

$$= \left\{ \begin{array}{l} \text{Arc}(a, b),\ \text{Arc}(b, a),\ \text{Arc}(c, a), \\ \text{tc\_Arc}(a, b),\ \text{tc\_Arc}(b, a),\ \text{tc\_Arc}(c, a), \\ \text{tc\_Arc}(a, a),\ \text{tc\_Arc}(b, b),\ \text{tc\_Arc}(c, b). \end{array} \right\}$$

Note that $T^{\models}_{\text{Graph}}\uparrow 2 = T^{\models}_{\text{Graph}}\uparrow 1 \cup \{\text{tc\_Arc}(a, a), \text{tc\_Arc}(b, b), \text{tc\_Arc}(c, b)\}$

(4) Fourth iteration

$$T^{\models}_{\text{Graph}}\uparrow 3 = T^{\models}_{\text{Graph}}(T^{\models}_{\text{Graph}}\uparrow 2) = T^{\models}_{\text{Graph}}\uparrow 2.$$

Thus, the least fixpoint semantics of the positive logic program Graph is the least fixpoint of $T^{\models}_{\text{Graph}}$, that is $T^{\models}_{\text{Graph}}\uparrow 2$. Note here that the fixpoint semantics of Graph coincides with its minimal model semantics.

We would like to insist here on the fact that the fixpoint semantics has a strong computational aspect: "the fact that we have a least fixedpoint simply means that we have abstracted a computational process" [60].

For positive logic programs, the model-theoretic semantics and the fixpoint semantics coincide. On the one hand, this justifies the use of the least fixpoint semantics as a declarative semantics for positive programs. On the other hand, this proves that the minimal model semantics is "reasonable" since a constructive alternative definition is provided.

Theorem 3.8 (Apt and Van Emden [4], Van Emden and Kowalski [44]). *Let $\mathscr{P}$ be a positive logic program. The minimal model semantics and the fixpoint semantics of $\mathscr{P}$ are identical.*

Remark 3.9. It is important to note that during the iterative application of $T^{\models}_{\mathscr{P}}$, there is no manipulation of negative facts. The iteration starts with the empty Herbrand interpretation, that is an empty set of positive facts. Applying the immediate consequence operator to a set of facts, namely here a set of positive facts, produces instantiated heads of rules. Instantiated heads of rules are positive facts.

Indeed, for positive logic programs, the fixpoint semantics can be defined in terms of the following operator.

Definition 3.10 (*Set membership immediate consequence operator*). Let $\mathscr{P}$ be a logic program. The (set membership) immediate consequence operator associated with $\mathscr{P}$, denoted by $T^{\in}_{\mathscr{P}}$, is the mapping on **Int** defined by

$$T^{\in}_{\mathscr{P}}(I) = \{\text{head}(r) \mid r \in \mathit{Inst\_}\mathscr{P} \text{ and } \forall L \in \text{body}(r), L \in I\}.$$

It is important to note that:

Theorem 3.11. *For any logic program $\mathscr{P}$ (not necessarily positive logic program), the immediate consequence operator $T^{\in}_{\mathscr{P}}$ is monotonic and continuous.*

This result follows from the fact that applying $T^{\in}_{\mathscr{P}}$ to an interpretation I comes down to applying $T^{\models}_{\mathscr{P}'}$ to I where $\mathscr{P}'$ is the positive logic program obtained by removing from $\mathscr{P}$ all rules having some negative premises.

Moreover, we have for any positive logic program:

Lemma 3.12. *If $\mathscr{P}$ is a positive logic program, then* $\text{lfp}(T^{\models}_{\mathscr{P}}) = \text{lfp}(T^{\in}_{\mathscr{P}})$ *and the fixpoint semantics of $\mathscr{P}$ is equal to the least fixpoint of $T^{\in}_{\mathscr{P}}$.*

Thus, for positive logic programs, logical consequence ($\models$) can be replaced in the definition of $T^{\models}_{\mathscr{P}}$ by set membership ($\in$). This substitution preserves the definition of the fixpoint semantics of positive logic programs.

Now, we would like to emphasize the fact that the definition of the declarative semantics of positive logic programs by means of the least fixpoint of the operator $T^{\models}_{\mathscr{P}}$, i.e. by means of the limit of the sequence $T^{\models}_{\mathscr{P}}\uparrow\alpha$ can be viewed as a process of two ordered phases.

The first phase takes care of the generation of positive facts true for $\mathscr{P}$. It consists of the iterative application of $T^{\models}_{\mathscr{P}}$.

The second phase (totally hidden) takes care of the generation of negative facts true for $\mathscr{P}$. Negative facts true for $\mathscr{P}$ are obtained from the positive facts true for $\mathscr{P}$ and computed during the first phase, by complementation in the Herbrand Base.

The completed interpretation $\mathscr{M}$ associated with the canonical model $M = \text{lfp}(T^{\models}_{\mathscr{P}})$ of the positive logic program $\mathscr{P}$, is given by $\mathscr{M} = M \cup \neg.(\mathscr{B}_{\mathscr{P}} - \text{lfp}(T^{\models}_{\mathscr{P}}))$.

The second part of this section exposes the problems that arise when negative premises are introduced in program rules. First we shall briefly recall the motivation for introducing negation in logic programs.

Let us consider the positive logic program Graph of Example 3.4 and say that we want to isolate the new arcs in tc_Arc that is the arcs in tc_Arc that are not given in the "relation" Arc. Let us introduce a new predicate symbol New_Arc.

New_Arc is not definable (intentionally) by a positive logic program. Introducing negative premises allows us to write the following definition for New_Arc:

$$\text{New\_Arc}(x, y) \text{ :- } \text{tc\_Arc}(x, y) \,\&\, \text{not Arc}(x, y)$$

and the intended semantics of the logic program Graph augmented with the above rule is given by M_{Graph} plus the facts New_Arc(a, a), New_Arc(b, b) and New_Arc(c, b).

From a procedural viewpoint, the semantics of negation in logic programs is close to the well-known Negation as Failure procedure (SLDNF) [35]. SLDNF is a top-down evaluation procedure extending SLD-resolution. The extension concerns the notion of a proof of an elementary negative goal which is defined as the failure to obtain a proof of the corresponding positive goal.

For our example, failure to prove Arc(a, a) provides an SLDNF-proof of $\neg$Arc(a, a) and thus an SLDNF-proof of New_Arc(a, a). The same holds for New_Arc(b, b) and New_Arc(c, b). Now, the proof of Arc(a, b) fails the SLDNF-proof of $\neg$Arc(a, b) and thus the SLDNF-proof of New_Arc(a, b). The same holds for New_Arc(b, a) and New_Arc(c, a).

In the context of databases, introducing negation in logic programs aims to provide the ability to express set difference or set complement.

3.2. Generalizing the fixpoint approach: the problems

Let us now examine general logic programs, that is logic programs in which negative literals are allowed in the body of rules. Before presenting the formal arguments that rule out the use of the immediate consequence operator for defining the declarative semantics of logic programs, we provide some rather intuitive reasons which lead to the very same conclusion and which motivate the contents of the next sections.

The immediate consequence operator has been introduced as a way to "apply rules to facts". Intuitively, for a general logic program, because negative premises may occur in the body of rules, one expects that "applying rules to facts" involves an explicit use of negative facts.

Let us examine how the two operators $T^{\models}_{\mathscr{P}}$ and $T^{\in}_{\mathscr{P}}$ deal with negative premises of rules. The main thing to keep in mind is that both operators are defined on **Int** and produce positive facts exclusively. This partially explains their inadequacy to define the declarative semantics of logic programs with negation.

Let us assume that we want to use the limit of the sequence $T^{\models}_{\mathscr{P}}\uparrow\alpha$ in order to provide a constructive semantics of a logic program $\mathscr{P}$. For the moment, the reader is asked to forget or ignore the formal reason which makes our assumption and discussion formally incorrect (the immediate consequence operator associated with a general logic program is non-monotonic). This assumption is made for the purpose of the intuitive motivation of the further presentation.

Consider for example the simple propositional program $\mathscr{P}_0 = \{A, B \text{ :- not } A\}$. Applying $T^{\models}_{\mathscr{P}_0}$ to the empty Herbrand interpretation produces the set of positive

facts $\{A, B\}$. Intuitively, the proposition A has been derived from the first unit rule of the program while the proposition B has been derived from the second rule of the program using the fact that the empty Herbrand interpretation *satisfies* the negative literal not A. Loosely speaking, the empty interpretation is intended to represent the "starting point" for computing the semantics of the program $\mathscr{P}$. Thus, normally it should represent the fact that we know nothing about truth values of facts (positive and negative ones) defined by $\mathscr{P}$. However here, the definition of the immediate consequence operator in terms of $T^{\models}_{\mathscr{P}}$ definitely gives to the empty interpretation a rather different intuitive meaning: the empty interpretation represents a state of "knowledge" in which *every positive fact is false.* In order to express the immediate consequence operator $T^{\models}_{\mathscr{P}}$ using set membership, we have to write:

$$T^{\models}_{\mathscr{P}}(\boldsymbol{I}) = \left\{ \text{head}(r) \;\middle|\; \begin{array}{l} r \in \text{Inst\_}\mathscr{P}, \\ \forall L \in \text{pos}(\text{body}(r)),\ L \in \boldsymbol{I}, \text{ and} \\ \forall L \in \text{neg}(\text{body}(r)),\ \neg L \notin \boldsymbol{I} \end{array} \right\}$$

Rewriting the definition of $T^{\models}_{\mathscr{P}}$ as above emphasizes the way heads of rules are derived by application of this operator to some set $\boldsymbol{I}$ of positive hypotheses. The head of a rule r in the instantiation of $\mathscr{P}$ is generated iff r satisfies the following two conditions:

(1) each positive premise of r belongs to the set of hypotheses, and
(2) the positive counterpart of each negative premise of r *does not* belong to the set of hypotheses.

Condition (2) clearly means that negative premises of rules are assumed to be "true" by default to get their positive counterpart explicitly in the set of hypotheses. In our example, it is clear that B is derived by default to find A in the hypotheses represented by the empty interpretation. The reason why B is derived is not that $\neg$A has been previously inferred.

This remark leads to the idea that the set membership immediate consequence operator $T^{\in}_{\mathscr{P}}$ may be more adequate.

Using $T^{\in}_{\mathscr{P}}$, in order to be considered "true", positive premises as well as negative premises of rules are required to belong explicitly to the hypotheses. In this setting, as expected, the intuitive meaning of the empty interpretation corresponds to knowing nothing about the truth of facts. As a matter of fact, for the program $\mathscr{P}_0$, applying $T^{\in}_{\mathscr{P}_0}$ to the empty Herbrand interpretation produces the set of positive facts $\{A\}$. However, the set membership immediate consequence operator solves only one part of the problem: negative premises are not inferred to be true "by default". The second part of the problem remains and is crucial: the immediate consequence operator is unable to derive negative facts. This is showed by the following example.

Let us delete the first rule of the previous program and thus consider the program formed by the unique rule B :- not A. Applying $T^{\in}_{\mathscr{P}}$ to the empty Herbrand interpretation produces an empty set of facts. Intuitively, the empty interpretation (either viewed as every positive fact is false, or viewed as nothing is either true or false)

does not correspond to the intended meaning of the logic program $\{B$:- not $A\}$. Clearly, the intended meaning of this program is captured by the Herbrand interpretation $\{B\}$ or by the completed Herbrand interpretation $\{\neg A, B\}$. Moreover, the empty interpretation is not a model of the logical notation $\{B \leftarrow A\}$ of the program $\{B$:- not $A\}$.

We now turn to formal arguments that makes the immediate consequence operator unusable for defining, in a constructive way, the declarative semantics of logic programs with negation. It is well known that:

Lemma 3.13. *Let $\mathscr{P}$ be a logic program. The operator $T^{\models}_{\mathscr{P}}$ is not monotonic.*

It suffices to consider the program $\{B$:- not $A\}$ and the two interpretations $I_1 = \emptyset$ and $I_2 = \{A\}$. We have that: $I_1 \subseteq I_2$ but $T^{\models}_{\mathscr{P}}(I_1) = \{B\} \not\subseteq T^{\models}_{\mathscr{P}}(I_2) = \emptyset$.

The major consequences of Lemma 3.13 are that:

(1) $T^{\models}_{\mathscr{P}}$ may or may not have fixpoints,

(2) if $T^{\models}_{\mathscr{P}}$ has fixpoints it may or may not have a least fixpoint,

(3) in the "good" case where $T^{\models}_{\mathscr{P}}$ has a least fixpoint, it may not be equal to the limit of the sequence $T^{\models}_{\mathscr{P}} \uparrow \alpha$.

Some simple examples below illustrate these remarks.

Example 3.14. (1) $T^{\models}_{\mathscr{P}}$ may have a least fixpoint. Consider the logic program $\mathscr{P}_1 = \{A, C$:- A & not $B\}$. The immediate consequence operator $T^{\models}_{\mathscr{P}_1}$ has a least fixpoint given by the Herbrand interpretation $M_1 = \{A, C\}$. Note first that M_1 is a (minimal) Herbrand model of $\mathscr{P}_{1_{fo}}$. Secondly, M_1 is the limit of the sequence $T^{\models}_{\mathscr{P}_1} \uparrow \alpha$. Now, it can also be useful to notice that A, C and $\neg B$ may be infered by SLDNF.

(2) $T^{\models}_{\mathscr{P}}$ may have fixpoints but no least fixpoint. Consider the logic program $\mathscr{P}_2 = \mathscr{P}_1 \cup \{B$:- $B\}$. Note that the immediate consequence operator $T^{\models}_{\mathscr{P}_2}$ has two fixpoints given by the two following Herbrand interpretations: $M_2 = \{A, C\}$ and $M_2' = \{A, B\}$. But the immediate consequence operator $T^{\models}_{\mathscr{P}_2}$ associated with $\mathscr{P}_2$ has no least fixpoint. Note also that both M_2 and M_2' are (minimal) Herbrand models for $\mathscr{P}_{2_{fo}}$. The limit of the sequence $T^{\models}_{\mathscr{P}_2} \uparrow \alpha$ is the set of facts M_2. Finally, the only fact that may be infered by SLDNF is the positive fact A. Note that, viewed as a "production" rule, the program rule B :- B adds no information to the program $\mathscr{P}_1$. We call this rule a "ghost rule". Thus we could reasonably expect the logic programs $\mathscr{P}_1$ and $\mathscr{P}_2$ to have the same declarative semantics. The immediate consequence operator fails to match our expectation because it behaves differently on the logic programs $\mathscr{P}_1$ and $\mathscr{P}_2$.

(3) $T^{\models}_{\mathscr{P}}$ may not have fixpoints. Consider the logic program $\mathscr{P}_3 = \{A$:- not $A\}$. The immediate consequence operator $T^{\models}_{\mathscr{P}_3}$ does not have a fixpoint (thus a fortiori does not have a least fixpoint). Because $T^{\models}_{\mathscr{P}_3}(\emptyset) = \{A\}$ and $T^{\models}_{\mathscr{P}_3}(\{A\}) = \emptyset$, we have $T^{\models}_{\mathscr{P}_3} \uparrow \omega = \{A\}$ is the unique model of $\mathscr{P}_{3_{fo}}$ (Note that the sequence $T^{\models}_{\mathscr{P}_3} \uparrow \alpha$ does not converge). Not surprisingly, nothing can be infered from $\mathscr{P}_3$ by SLDNF.

(4) $T^{\models}_{\mathscr{P}}$ may have a least fixpoint that is not reachable by $T^{\models}_{\mathscr{P}}\uparrow\alpha$. Consider the program $\mathscr{P}_4 = \{A$:- not C, B :- not A, C :- not A & not $B\}$. The immediate consequence operator $T^{\models}_{\mathscr{P}_4}$ has a unique fixpoint and thus a least fixpoint given by the Herbrand interpretation $\{A\}$. This unique fixpoint is one of the (minimal) Herbrand models of $\mathscr{P}_{4_{f.o.}}$. While we are in an apparently "good" case where $T^{\models}_{\mathscr{P}_4}$ has a unique fixpoint, the sequence $T^{\models}_{\mathscr{P}_4}\uparrow\alpha$ does not converge to this model. Because $T^{\models}_{\mathscr{P}_4}(\emptyset) = \{A, B, C\}$ and $T^{\models}_{\mathscr{P}_4}(\{A, B, C\}) = \emptyset$, $T^{\models}_{\mathscr{P}_4}\uparrow\omega = \{A, B, C\}$ which is not equal to the least fixpoint of $T^{\models}_{\mathscr{P}_4}$. Of course, nothing can be computed from $\mathscr{P}_4$ by SLDNF.

We now examine formally the reason that makes the set membership immediate consequence operator $T^{\in}_{\mathscr{P}}$ unsatisfactory for defining the declarative semantics of logic programs.

Lemma 3.15. *Let $\mathscr{P}$ be a logic program. The least fixpoint* $\mathrm{lfp}(T^{\in}_{\mathscr{P}})$ *(i.e. the limit of the sequence $T^{\in}_{\mathscr{P}}\uparrow\alpha$) associated with $\mathscr{P}$ may not be a (Herbrand) model of $\mathscr{P}_{f.o.}$.*

It suffices to consider the program $\{B$:- not $A\}$. The least fixpoint associated with this program is the empty interpretation. However the empty interpretation is not a model of the f.o. formula $B \leftarrow \neg A$.

The main weakness of the operator $T^{\in}_{\mathscr{P}}$ is that it completely ignores the rules in $\mathscr{P}$ having negative premises and treats these rules as "ghost rules".

Recall that, considering a general logic program $\mathscr{P}$ and the positive logic program $\mathscr{P}'$ obtained by removing all rules with negative premises from $\mathscr{P}$, the least fixpoint of the operator $T^{\in}_{\mathscr{P}}$ associated with $\mathscr{P}$ is equal to the least fixpoint of the operator $T^{\models}_{\mathscr{P}'}$ associated with the *positive* logic program $\mathscr{P}'$.

3.3. Generalizing the model theoretic approach: the problems

Problems analogous to the ones encountered with the fixpoint approach arise in the model theoretic framework. More precisely, the minimal model approach does not provide a satisfactory declarative semantics for programs with negation.

Briefly, an illustration of the problem can be made by considering the logic program $\mathscr{P}_1 = \{A, C$:- A & not $B\}$ of Example 3.14. The (f.o. notation of the) program $\mathscr{P}_1$ has two minimal models, namely the fixpoints $M_1 = \{A, C\}$ and $M'_1 = \{A, B\}$ of $T^{\models}_{\mathscr{P}_1}$. Let us recall that among the minimal models M_1 and M'_1, the one which represents the intended meaning of $\mathscr{P}_1$ is M_1.

Thus the minimality condition is not sufficient for selecting the "good" model representing the meaning of a logic program with negation. The question that arises immediately is how the "good" minimal model capturing the meaning of a logic program can be characterized. The main thing to note is that properties of the f.o. notation of a program are unable to overcome the problem. For instance, consider the program $\mathscr{P}'_1 = \{A, B$:- A & not $C\}$. The f.o. notation $\{A, B \leftarrow A \wedge \neg C\}$ of $\mathscr{P}'_1$ is logically equivalent to the f.o. notation $\{A, C \leftarrow A \wedge \neg B\}$ (because, $(B \leftarrow A \wedge \neg C) \leftrightarrow (C \leftarrow A \wedge \neg B) \leftrightarrow (C \vee \neg A \vee B)$). It follows that M_1 and M'_1 are the two minimal

models of (the f.o. notation of) the program $\mathscr{P}_1'$. While M_1 captures the meaning of the program $\mathscr{P}_1$, the minimal model that is intended to represent the meaning of $\mathscr{P}_1'$ is M_1'. "What complicates the matter is that the choice of $M_{\mathscr{P}}$[3] is apparently not invariant to logically equivalent transformations of $\mathscr{P}$" [74].

The criterion applied in order to select M_1, respectively M_1', among M_1 and M_1' as describing the semantics of $\mathscr{P}_1$, respectively the semantics of $\mathscr{P}_1'$, relies on properties induced by the syntax of $\mathscr{P}_1$, respectively by the syntax of $\mathscr{P}_1'$. Recall that a clear distinction has been made earlier between logic programming languages and first order languages. This distinction, which can be cumbersome, is necessary in order to be able to state in a formally correct fashion that $\mathscr{P}_1$ and $\mathscr{P}_1'$ are distinct programs and thus (may) have distinct meanings.

The notion of a supported model has been introduced in [3] as a criterion for selecting models among minimal models. The very same notion has been independently introduced in [18] where it is called a causal or justified model.[4] In [18], the motivation for defining supported models is given by the search for a model theoretic formalization of Clark's negation as failure inference rule [35].

Intuitively, an interpretation M is supported by a logic program $\mathscr{P}$ if positive facts true for M can be "produced" from some rule r in $\mathscr{P}$ and from the positive and negative facts true for M. An elegant definition of supported model is provided in [3], which not surprisingly makes use of the immediate consequence operator $T_{\mathscr{P}}^{\models}$ associated with $\mathscr{P}$.

Definition 3.16 (*Supported model of a logic program*). Let $\mathscr{P}$ be a logic program. Let M be a Herbrand interpretation. A model M of $\mathscr{P}_{\text{f.o.}}$ is a supported model for $\mathscr{P}$ iff $M \subseteq T_{\mathscr{P}}^{\models}(M)$.

Remark 3.17. Shepherdson [104] provides the following simple and pertinent remark:

- M is a model of $\mathscr{P}_{\text{f.o.}}$ iff $T_{\mathscr{P}}^{\models}(M) \subseteq M$ and consequently,
- M is a supported model for $\mathscr{P}$ iff M is a fixpoint of $T_{\mathscr{P}}^{\models}$.

Example 3.18. The definition is briefly illustrated using the program $\mathscr{P}_1$ of Example 3.14. The models of $\mathscr{P}_1$ are $M = \{A, B, C\}$, $M_1 = \{A, C\}$ and $M_1' = \{A, B\}$. The only supported model of $\mathscr{P}_1$ is M_1 which is the model capturing the intended meaning of $\mathscr{P}_1$. Because there exists no rule in $\mathscr{P}_1$ with head C, M and M_1' are not supported for $\mathscr{P}_1$. Note also that the only supported model for $\mathscr{P}_1' = \{A, B \text{ :- } A \,\&\, \text{not } C\}$ is M_1' which is the model intended to capture the semantics of $\mathscr{P}_1'$. Recall that the two programs $\mathscr{P}_1$ and $\mathscr{P}_1'$ have equivalent f.o. notations.

As motivated in [19], the notion of a supported model is not sufficient to capture the intended meaning of logic programs. Indeed, the notion of a supported model

[3] The good model.
[4] In the following, we shall use the terminology of [3], that is supported model.

does not subsume the notion of a minimal model. As a consequence, the "pure" supported model semantics does not generalize the minimal model (or, equivalently, the least fixpoint) semantics for positive logic programs. An example follows to illustrate this remark.

Example 3.19. Consider the logic program $\mathscr{P} = \{A \text{ :- } B,\ B \text{ :- } A\}$. It admits two first order models, namely $M = \emptyset$ and $M' = \{A, B\}$. Both models are supported models for $\mathscr{P}$. On the other hand, the meaning associated with $\mathscr{P}$ with respect to the minimal model semantics (or equivalently, with respect to the least fixpoint semantics) is given by the empty model M.

In [19], the notion of a positivist model is introduced, combining the two notions of a minimal model and of a supported model. Positivist models of logic programs are investigated in [19] in the general case, that is, no hierarchical condition on the syntax of the logic programs of the kind discussed in [3] is introduced.

Definition 3.20 (*Positivist model of a logic program*). Let $\mathscr{P}$ be a logic program. Let M be a Herbrand interpretation. A model M of $\mathscr{P}_{\text{f o.}}$ is a positivist model for $\mathscr{P}$ iff M satisfies the following two properties:

(1) M is a minimal model of $\mathscr{P}_{\text{f.o.}}$, and

(2) M is a supported model for $\mathscr{P}$.

Remark 3.21. Attention should be paid to the way positivism combines minimalism and supportedness. In fact, examples are given in [19] which show that the family of positivist models of a logic program (models which enjoy both minimality among the f.o. models of $\mathscr{P}$ and supportedness for $\mathscr{P}$) is not identifiable with the family of minimal models among the supported models for $\mathscr{P}$. One of these examples is given below.

Example 3.22. Let us consider the program $\mathscr{P} = \{A \text{ :- } A,\ A \text{ :- } B,\ A \text{ :- not } C,\ C \text{ :- } A \,\&\, \text{not } B\}$. The unique model of $\mathscr{P}_{f\,o.}$ which is a supported model for $\mathscr{P}$ is $M = \{A, C\}$. Thus, M is the least supported model of $\mathscr{P}$. Notice that $M' = \{C\}$ is one of the minimal models of $\mathscr{P}_{\text{f.o}}$ (the other one is $\{A, B\}$) and is less than (included in) M. This entails that $\mathscr{P}$ admits no positivist model.

The main result of [19] establishes the correspondence between first order models of the completion of a logic program and the supported models of the program. Soundness of the negation as failure algorithm [35] is proved with respect to the positivist models of the program.

Unfortunately, it appears that minimality and supportedness are not always sufficient criteria for selecting the "good" model of a logic program. The introduction of rules of the form $P(x) \text{ :- } P(x)$ in a logic program has the effect of neutralizing the impact of supportedness on the predicate P.

Example 3.23. The program $\mathscr{P}_2 = \mathscr{P}_1 \cup \{B :\text{-} B\}$ of Example 3.14 gives a simple illustration of some "undesirable" behavior of the positivist semantics. The f.o. notation of $\mathscr{P}_2$ is logically equivalent to the f.o. notation of $\mathscr{P}_1$. Thus, $\mathscr{P}_1$ and $\mathscr{P}_2$ have the same minimal models, namely $M_1 = \{A, C\}$ and $M_1' = \{A, B\}$. Both M_1 and M_1' are supported models for $\mathscr{P}_2$. To show that $M_1' = \{A, B\}$ is a supported model for $\mathscr{P}_2$, one makes use of the "ghost rule" $B :\text{-} B$.

Although supportedness does not provide a completely satisfactory criterion for selecting the pertinent model(s) of a logic program, supportedness should be regarded as a desirable property of the intended meaning of a logic program because a supported model "is able to reproduce itself with a certain natural transformation" [115]. In the current context, the natural transformation is the immediate consequence operator.

In [3], the semantical notions of minimality and supportedness are combined with some syntactical restrictions of a hierarchical nature on the logic programs, providing a model theoretic semantics for the stratifiable programs (which is presented in Section 7).

Two main goals are pursued when defining the declarative semantics of logic programs:

(1) interpreting negation (as close as possible) like complementation,

(2) providing, if possible a constructive definition.

The first goal is of a semantic nature while the second one is of a computational nature. Both should be related to the two fundamental principles of [104]: "Even if the practicing logic programmer does regard the written text of the program as its declarative meaning, we feel that in order to be true to the basic aims of logic programming two fundamental principles should be observed.

(1) The semantics of negation should be clear and easily intelligible. That is, the naive programmer should be able to understand the full meaning of what he writes.

(2) The syntax should be computable. That is, at least in theory, an automatic proof procedure should exist. Indeed if the only reason for abandoning the clear and well-known concept of classical negation is the inefficiency of its implementation, we might not unreasonably ask for a complete proof procedure to be feasibly implementable."

In the following sections, we present some contributions to defining the declarative semantics of logic programs and try to measure them against the above two goals.

4. Fixpoint semantics of stratifiable logic programs

In this section, we focus our attention on the different attempts to exhibit a "good" class of logic programs. By "good" logic programs is meant logic programs that are sufficiently simple to raise no discussion about their intended meaning. The problem is of course to define a class of programs as large as possible.

In the previous section, we saw that the semantics of a positive program $\mathscr{P}$ may be constructively defined by means of the least fixpoint of the immediate consequence operator associated with $\mathscr{P}$. It has been suggested that this constructive definition can be seen as a process of two phases, the first phase being dedicated to the iterative derivation of positive facts, the second (hidden) phase consisting of the derivation of the negative facts by complementation.

This view of the fixpoint semantics leads naturally to the idea that, a "safe" way to add negation is to consider a logic program as an ordered sequence of logic programs where the use of negation is restricted at each level to apply exclusively on predicates defined in programs of lower levels. Intuitively, this ordered structure of logic programs implies a very simple way for "evaluating" its semantics. Starting from the lowest level program, the least fixpoint semantics gives for each predicate P defined at each level, the positive facts and the negative facts related to P. So that if, at some level, the definition of P makes use of the negation of the predicate Q, the definition of Q belongs to a subprogram of lower level and thus its semantics has already been "evaluated". This means that the set of negative facts related to Q is available for the evaluation of the positive facts related to P.

This notion of ordered programs is well known as stratified programs. It is used first in [30] in order to generalize the class of Horn clause queries (queries expressed by positive logic program). Stratification becomes popular with the work of [3, 91, 94, 111].

In [30], it is shown that positive logic programs express exactly the queries representable by a fixpoint applied to a positive existential formula (see Section 9 on the expressive power of logic programs). Thus not all first order queries are expressible as positive logic programs. Ways of adding negation to logic programs are examined. The first attempt to extend the class of queries specified by positive logic programs is very simple.

In the alphabet, two classes of predicate symbols are distinguished: terminal predicates and non-terminal predicates. Using the database terminology, terminal predicates correspond to relations explicitly stored in the database and non-terminal predicate symbols correspond to relations intentionally defined by rules. In queries (logic programs), negation is allowed among the premises of rules as long as it applies to terminal predicate symbols. This class of queries corresponds to the class of semi-positive programs [3] defined below.

Definition 4.1 (*Semi-positive logic program*). A logic program $\mathscr{P}$ is semi-positive iff the set of predicate symbols occurring in negative premises of rules in $\mathscr{P}$ and the set of predicate symbols occurring in head of non-unit rules in $\mathscr{P}$ are disjoint.

The logic program $\mathscr{P}_1$ of Example 3.14 is semi-positive because the proposition B does not occur in any head of rules. The logic programs $\mathscr{P}_2$, $\mathscr{P}_3$ and $\mathscr{P}_4$ of Example 3.14 are not semi-positive.

A somewhat more illustrative example is proposed below.

Example 4.2. Let us consider the language induced by the constant symbols Mary, John, Peter, Eva, and by the unary predicate symbols Businessman, Mathematician, Computerscientist, and Avoids_Math. The following program intends to say that people who are businessman and not mathematician avoid mathematics. The "relations" Businessman, Mathematician and Computerscientist are defined "extensively".

$$\text{Math} = \left\{ \begin{array}{l} \text{Businessman(John), Businessman(Mary),} \\ \text{Mathematician(Mary),} \\ \text{Computerscientist(Peter),} \\ \text{Avoids\_Math}(x) \text{ :- Businessman}(x) \text{ \& not Mathematician}(x) \end{array} \right\}$$

The intended semantics of the logic program Math is very easy to exhibit. The predicates Businessman, Mathematician and Computerscientist are terminal predicates. The intended semantics assigned by Math to these predicates is simply given directly by the unit rules related to each of them (because these unit rules are ground). Thus Math defines the three relations:

Businessman
John
Mary

Mathematician
Mary

Computerscientist
Peter

Now, the relation Avoids_Math defined by Math is the difference between the relation Businessman and Mathematician, that is:

Avoids_Math
John

First, note that the (f.o. notation of the) logic program Math has two minimal models: one contains the facts represented above in the relation Businessman, Mathematician, Computerscientist and Avoids_Math, the other one contains the facts represented above in the relations Businessman, Mathematician and Computerscientist plus the fact Mathematician(John). The first of these models (the one that corresponds to our intention) is a supported model for Math while the second one is not.

Secondly, note that the immediate consequence operator associated with Math has a least fixpoint, namely the model corresponding to our intention, and that this least fixpoint can be "computed" by means of the sequence $T^{\models}_{\text{Math}} \uparrow \alpha$.

Intuitively, for a semi-positive logic program, predicates that occur in negative premises of rules have no intentional definition in the program. Thus, intuitively,

the intended semantics assigned by the program to these predicates is the set of facts related to these predicates and that belong to the program itself. As a consequence, the "evaluation" of negative premises is immediate and does not require any intermediate "evaluation".

The simplicity of these programs entails that it is "safe" to use the immediate consequence operator in order to constructively describe the semantics of a semi-positive logic program. Formally,

Lemma 4.3 (Apt et al. [3]). *If $\mathscr{P}$ is a semi-positive program then $T^{\models}_{\mathscr{P}}\uparrow\omega$ is a fixpoint of $T^{\models}_{\mathscr{P}}$.*

The notion of stratified logic program is a straightforward generalization of semi-positive program. Roughly speaking, a stratified logic program is a sequence of semi-positive logic programs. The definition of stratified logic programs is presented here in a slightly different manner than in [3, 91, 94, 111].

Definition 4.4 (*Stratified logic program*). Let C be a set of function symbols. A stratified logic program $\mathscr{P}^*$ is a (possibly infinite) sequence $(\mathscr{P}_n, \mathit{Pred}_n)_{(n\geq 1)}$ such that:

(1) $\mathit{Pred}_i \neq \emptyset$ and Pred_i and Pred_j are pairwise disjoint, for $i \neq j$,
(2) $\mathscr{P}_i$ is a semi-positive logic program defined over the language $(C, \bigcup_{j=1}^{i} \mathit{Pred}_j)$
(3) the set of predicate symbols occurring in the head of rules in $\mathscr{P}_i$ is included in Pred_i, i.e. $\{\text{head}(r) \mid r \in \mathscr{P}_i\}$ is defined over (C, Pred_i).

Each $(\mathscr{P}_i, \mathit{Pred}_i)$ is called a stratum of $\mathscr{P}^*$.

Example 4.5. (1) The sequence of semi-positive programs

$$\mathscr{P}_1^* = (\emptyset, \{B\})(\{A, C \text{ :- } A \,\&\, \text{not } B\}, \{A, C\})$$

is a stratified logic program.

(2) The sequence of semi-positive programs

$$\mathscr{P}_1^{*\prime} = (\{A, C \text{ :- } A \,\&\, \text{not } B\}, \{A, B, C\})$$

is a stratified logic program.

(3) The sequence of semi-positive programs

$$\mathscr{P}_2^* = (\{A, B \text{ :- } B\}, \{A, B\})(\{C \text{ :- } A \,\&\, \text{not } B\}, \{C\})$$

is a stratified logic program.

Note that in Definition 4.4, while Pred_i are always non-empty sets of predicates, $\mathscr{P}_i$ may be empty programs. Stratifiable logic programs are simply defined as follows.

Definition 4.6 (*Stratifiable logic program*). A logic program $\mathscr{P}$ over (C, Pred) is stratifiable iff there exists a stratified logic program $\mathscr{P}^* = (\mathscr{P}_n, \mathit{Pred}_n)_{(n\geq 1)}$ such that $\mathscr{P} = \bigcup_{j=1}^{\infty} \mathscr{P}_j$ and $\mathit{Pred} = \bigcup_{j=1}^{\infty} \mathit{Pred}_j$. $\mathscr{P}^*$ is then called a stratification for the logic program $\mathscr{P}$.

In [75], stratifiable logic programs are defined in an equivalent way by means of a level mapping. "A logic program is stratifiable if it has a level mapping such that, in every program rule r, the level of the predicate symbol of every positive literal in the body is less than or equal to the level of the predicate symbol of head(r), and the level of the predicate symbol of every negative literal in the body is (strictly) less than the level of the predicate in head(r)."

Infinite stratified logic programs have been introduced for the purpose of the presentation of locally stratifiable logic programs provided later in this section. Thus until that presentation, it is understood that we only consider finite stratified programs.

Given a *stratifiable* logic program $\mathscr{P}$, it is easy to verify that there always exists a finite *stratified* logic program $\mathscr{P}^*$ "equal" to $\mathscr{P}$.

Example 4.7. (1) The logic program $\mathscr{P}_1$ of Example 3.14 is semi-positive thus obviously it is stratifiable. Also $\mathscr{P}_1^*$ and $\mathscr{P}_1^{*\prime}$ provided in Example 4.5 are stratified programs "equal" to $\mathscr{P}_1$.

(2) Although it is not semi-positive, the logic program $\mathscr{P}_2$ of Example 3.14, is stratifiable. Note that the sequence of semi-positive logic programs $\mathscr{P}_2^*$ given in Example 4.5 is a stratified logic program "equal" to $\mathscr{P}_2$.

(3) The logic programs $\mathscr{P}_3$ and $\mathscr{P}_4$ of Example 3.14 are not stratifiable.

Note that more than one stratified logic program may be in correspondence with a single stratifiable logic program.

Given a stratified logic program $\mathscr{P}^* = (\mathscr{P}_n, \mathit{Pred}_n)$, we abusively refer to $\mathscr{P}$ as $\bigcup_{j=1}^{n} \mathscr{P}_j$ and to Pred as $\bigcup_{j=1}^{n} \mathit{Pred}_j$. With the above notational convention, we have that:

- given a predicate symbol $\mathscr{P}$ in Pred, $\mathrm{def}(P, \mathscr{P}) \subseteq \mathscr{P}_j$ for some j, and
- given a rule r in $\mathscr{P}_i$ and a predicate P that occurs in a negative premise of r, $\mathrm{def}(P, \mathscr{P}) \subseteq \bigcup_{j=1}^{i-1} \mathscr{P}_j$.

From the definition, this implies that there exists $j < i$ such that $\mathrm{def}(P, \mathscr{P}) \subseteq \mathscr{P}_j$.

The second point above formally expresses the condition that predicates must be completely defined "before" they can be used negatively.

Given a stratified program $\mathscr{P}^* = (\mathscr{P}_1, \mathit{Pred}_1), (\mathscr{P}_2, \mathit{Pred}_2), \ldots, (\mathscr{P}_n, \mathit{Pred}_n)$, the declarative semantics of $\mathscr{P}$ is obviously obtained by iteratively applying the "two phases" mechanism offered by the immediate consequence operator for constructing the canonical model of positive logic programs. Intuitively, we have:

- $\mathscr{P}_1$ is a semi-positive logic program. Predicates occurring in negative premises of rules in $\mathscr{P}_1$ do not have intentional definitions in the full program $\mathscr{P}$. The semantics of $\mathscr{P}_1$ is perfectly understood and given by the completed Herbrand interpretation $\mathscr{M}_1$ where $\mathrm{pos}(\mathscr{M}_1) = T_{\mathscr{P}_1}^{\models} \uparrow \omega$ and $\mathrm{neg}(\mathscr{M}_1) = \neg.(\mathscr{B} - \mathrm{pos}(\mathscr{M}_1))$.
- Now, let us examine the logic program $\mathscr{P}_2$ while keeping in mind that the semantics of $\mathscr{P}_1$ is known. Because predicates occurring negatively in the body of rules of $\mathscr{P}_2$ must have their definition in $\mathscr{P}_1$ and because the semantics of $\mathscr{P}_1$ is known,

intuitively the logic program $\mathscr{P}_2$ can be cleaned up of negative premises, moreover it can be cleaned up of predicates in $Pred_1$. In order to get rid of negation in $\mathscr{P}_2$, it suffices to remove from $\mathscr{P}_2$ the rules having some negative premise not satisfied by $\mathscr{M}_1$ and to delete negative premise satisfied by $\mathscr{M}_1$. This comes down to replacing $\mathscr{P}_2$ by $\mathrm{DP}(\mathscr{P}_2, \mathrm{neg}(\mathscr{M}_1))$. If one wants to get rid of predicates in $Pred_1$, it suffices to replace $\mathscr{P}_2$ by $\mathrm{DP}(\mathscr{P}_2, \mathscr{M}_1)$. In either case, the remaining program is a positive logic program whose semantics is perfectly understood.

Another way to convince the reader (if necessary) that negation in $\mathscr{P}_2$ does not raise problems is to notice that, at this step, predicate symbols in $Pred_1$ play the role of terminal predicate symbols and predicate symbols in $Pred_2$ play the role of non-terminal symbols.

Loosely speaking, the semantics of the stratified program $\mathscr{P}$ is obtained by iteration from $\mathscr{P}_1$ to $\mathscr{P}_n$ of the process sketched for $\mathscr{P}_1$ and $\mathscr{P}_2$, collecting at each step i the definition of the predicates in $Pred_i$.

Formally, we have:

Definition 4.8 (*Iterative fixpoint semantics*). Let $\mathscr{P}^* = (\mathscr{P}_i, Pred_i)_{(1 \leqslant i \leqslant n)}$ be a stratified logic program. The iterative fixpoint of P^* is the Herbrand interpretation $M^*_{\mathscr{P}} = M_n$ where the sequence $M_1, \ldots, M_n$ is defined by:

$$M_1 = T^{\models}_{\mathscr{P}_1} \uparrow \omega(\emptyset), \qquad M_{i+1} = T^{\models}_{\mathscr{P}_{i+1}} \uparrow \omega(M_i) \cup M_i, \quad \text{for } i \geqslant 1.$$

In the above definition, the "derivation" of negative facts is once again totally hidden. However, the reader should be aware that while "computing" the semantics of stratified programs, deduction of negative facts is not delayed after complete deduction of positive facts but comes in between production of positive facts for each stratum of the logic program. Production of negative facts is performed by complementation at each step. We now propose a toy example to illustrate the notion of stratifiable program and iterative fixpoint semantics. The example is taken from [94].

Example 4.9. Consider the logic program:

$$\mathscr{P} = \left\{ \begin{array}{l} \text{Businessman(Iacocca)}, \\ \text{Physicist(Einstein)}, \\ \text{Avoids\_Math}(x) \text{ :- Businessman}(x) \text{ \& not Good\_Mathematician}(x). \end{array} \right\}$$

The program $\mathscr{P}$ is stratifiable. One of the stratified programs equal to $\mathscr{P}$ is $\mathscr{P}_1$, $\mathscr{P}_2$ where

$$\mathscr{P}_1 = \emptyset \text{ over } Pred_1 = \{\text{Good\_Mathematician}\},$$

$$\mathscr{P}_2 = \mathscr{P} \text{ over } Pred_1 \cup \{\text{Businessman, Physicist, Avoids\_Math}\}.$$

The iterative fixpoint of $(\mathscr{P}_1, Pred_1)(\mathscr{P}_2, Pred_2)$ is M_2 where

$M_1 = \emptyset$, and

$M_2 = \{\text{Businessman(Iacocca), Physicist(Einstein), Avoids\_Math(Iacocca)}\}$

Another stratified program equal to $\mathscr{P}$ is $\mathscr{P}'_1, \mathscr{P}'_2$ where

$$\mathscr{P}'_1 = \left\{\begin{array}{l}\text{Businessman(Iacocca)}\\ \text{Physicist(Einstein)}\end{array}\right\} \text{ over } Pred'_1 = \left\{\begin{array}{l}\text{Businessman,}\\ \text{Physicist,}\\ \text{Good\_Mathematician.}\end{array}\right\}$$

$\mathscr{P}'_2 = \{\text{Avoids\_Math}(x) \text{ :- Businessman}(x) \text{ \& not Good\_Mathematician}(x)\}$

over $Pred'_1 \cup \{\text{Avoids\_Math}\}$.

The iterative fixpoint of $(\mathscr{P}'_1, Pred'_1)(\mathscr{P}'_2, Pred'_2)$ is M'_2 where

$M'_1 = \{\text{Businessman(Iacocca), Physicist(Einstein)}\}$, and

$M'_2 = \{\text{Businessman(Iacocca), Physicist(Einstein), Avoids\_Math(Iacocca)}\}$.

Recall that more than one stratified logic program may be in correspondence with a single *stratifiable* logic program. Thus in order to define the semantics of a stratifiable logic program, we need first to present the following property.

Theorem 4.10 (Apt et al. [3]). (Independence of the stratification). *Let* $\mathscr{P}^* = (\mathscr{P}_j, Pred_j)_{(1 \leq j \leq n)}$ *and* $\mathscr{P}^{*\prime} = (\mathscr{P}'_j, Pred'_j)_{(1 \leq j \leq m)}$ *be two stratified logic programs. If* $\bigcup_{j=1}^{n} \mathscr{P} = \bigcup_{j=1}^{m} \mathscr{P}'$ *and* $\bigcup_{j=1}^{n} Pred_j = \bigcup_{j=1}^{m} Pred'_j$ *then the iterative fixpoint of* $\mathscr{P}^*$ *is equal to the iterative fixpoint of* $\mathscr{P}^{*\prime}$.

We are now ready to properly define the iterative fixpoint semantics for stratifiable logic programs. The above property justifies the following definition.

Definition 4.11 (*Iterative fixpoint semantics of stratifiable logic programs*). Let $\mathscr{P}$ be a stratifiable logic program. The iterative fixpoint (semantics) of $\mathscr{P}$ is the iterative fixpoint of any of the stratified logic program "equal" to $\mathscr{P}$.

Obviously, the iterative fixpoint semantics generalizes the fixpoint semantics for positive programs (in the sense that the two semantics match for positive logic programs). It is interesting to note that:

Theorem 4.12 (Apt et al. [3]). *Let* $\mathscr{P}$ *be a stratifiable logic program. Let the Herbrand interpretation* $M_{\mathscr{P}}$ *be the iterative fixpoint of* $\mathscr{P}$.

(1) $M_{\mathscr{P}}$ *is a Herbrand model of* $\mathscr{P}_{\text{f.o.}}$,
(2) $M_{\mathscr{P}}$ *is a minimal model of* $\mathscr{P}_{\text{f.o.}}$, *and*
(3) $M_{\mathscr{P}}$ is a supported model of $\mathscr{P}$.

These properties have been exhibited in [3] in order to support the claim that the iterative fixpoint of $\mathscr{P}$ is "natural". The proof of the above results can be found in [3]. The properties (1) and (2) of Theorem 4.12 entail that the iterative fixpoint of a stratifiable logic program is a minimal fixpoint of the immediate consequence operator $T^{\models}_{\mathscr{P}}$ and also a minimal model of Comp($\mathscr{P}$) [104].

Before discussing the limitation of the stratification constraint, note that stratification enjoys the following property.

Lemma 4.13. *Let $\mathscr{P}$ be a logic program. It can be checked in polynomial time whether $\mathscr{P}$ is a stratifiable logic program.*

Roughly speaking, checking stratifiability of a logic program can be reduced to checking acyclicity of the precedence graph associated with the program. A polynomial algorithm that checks whether a (finite) logic program is stratifiable is provided in [3].

Stratifiable logic programs appear to be a natural and useful class of logic programs. Moreover, stratifiable programs are more expressive than positive programs. This last point will be discussed in Section 9. However, it also appears that stratification is too strong a constraint. Surprisingly, relaxing stratification has been first motivated by allowing function symbols in logic programs. Although the "constructive" requirement is not satisfied in such a context, local stratification is a formally interesting and very natural generalization of stratification. Local stratification has been introduced by [94]. The following presentation slightly differs from, but is equivalent to the one in [94].

Definition 4.14 (*Locally stratifiable logic program*). Let $\mathscr{P}$ be a logic program (possibly with function symbols). $\mathscr{P}$ is locally stratifiable iff the *propositional* logic program *Inst_*$\mathscr{P}$ is stratifiable. The iterative fixpoint of $\mathscr{P}$ is the lower upper bound of $\{M_i \mid i \geqslant 1\}$ where the M_i are defined as in Definition 4.8 for a stratified logic program equal to *Inst_*$\mathscr{P}$.

Note that here *Inst_*$\mathscr{P}$ can be viewed as a propositional program over the Herbrand base $\mathscr{B}$. Recall also that propositions are 0-ary predicate symbols, thus the definition of stratified program perfectly applies to the propositional case. Because function symbols are allowed, *Inst_*$\mathscr{P}$ may be an infinite set of rules and the stratifiable logic program equal to *Inst_*$\mathscr{P}$ may be an infinite sequence of propositional semi-positive logic programs.

Example 4.15. The following well-known logic program defines even numbers. The language is formed of the constant symbol 0, the 1-place function symbol suc and the unary predicate symbol even.

$$Even\_F = \left\{ \begin{array}{l} \text{even}(0) \\ \text{even}(\text{suc}(x)) \text{ :- not even}(x) \end{array} \right\}$$

The logic program *Even_F* is not stratifiable. However, it is locally stratifiable. The stratified logic program equal to the instantiation of *Even_F* is the infinite sequence $(Even\_F_i)_{(i \geq 0)}$ defined by:

$$Even\_F_0 = \{\text{even}(0)\} \quad \text{and,}$$

$$Even\_F_i = \{\text{even}(\text{suc}^i(0)) \text{ :- not even}(\text{suc}^{i-1}(0))\}, \text{ for } i > 0,$$

where $\text{suc}^0(0) = 0$ and $\text{suc}^i(0) = \text{suc}(\text{suc}^{i-1}(0))$.

The iterative fixpoint of *Even_F* is the lower upper bound of $\{\{\text{even}(0)\}, \ldots \{\text{even}(0), \text{even}(\text{suc}^2(0)), \ldots, \text{even}(\text{suc}^{2i}(0))\}, \ldots\}$ that is $\{\text{even}(\text{suc}^{2i}(0)) \mid i \geq 0\}$.

Lemma 4.16. *Let $\mathcal{P}$ be a logic program over a language whose function symbols are constant symbols only. If $\mathcal{P}$ is not stratifiable and if no constant symbol occurs in head of non-unit rules in $\mathcal{P}$ then $\mathcal{P}$ is not locally stratifiable.*

Intuitively speaking, Lemma 4.16 says that a logic program without function symbols is locally stratifiable if it is already in its instantiated form or almost.

This result shows that the contribution of local stratification is restricted to logic programs with function symbols. The main limitation of local stratification resides in the difficulty of checking whether a logic program is locally stratifiable.

Theorem 4.17 (Cholak [33]). *Checking whether a logic program (with function symbols) is locally stratifiable is undecidable.*

Intuitively, checking local stratifiability can be reduced to checking the existence of an infinite path in a (possibly "infinite") graph. This is implied by the fact that a logic program $\mathcal{P}$ is locally stratifiable iff the priority relation[5] associated with $\mathcal{P}$ is noetherian [94] where (A, B) is in the priority relation as soon as there is a path from B to A going through a negative edge in the precedence graph associated with the propositional program $Inst\_\mathcal{P}$.

To conclude this section, we present an example of a logic program (without function symbols), which is not locally stratifiable but has a natural intended meaning. This example motivates the search for improvements in the definition of the declarative semantics of logic programs and thus motivates the contents of Sections 5 and 8.

The example below is reproduced from [17].

Example 4.18. Let us consider a logic program that defines even number for a finite subset of the natural numbers, say for the natural numbers from 0 to i. The order

[5] Formally defined in Section 7.

on natural numbers is represented by means of a *relation* SUC (instead of using a function). The program is then written:

$$Even\_R = \left\{ \begin{array}{l} \text{SUC}(0, 1), \\ \text{SUC}(1, 2), \\ \vdots \\ \\ \text{SUC}(i-1, i), \\ \text{even}(0), \\ \text{even}(x) \text{ :- SUC}(y, x) \ \& \ \text{not even}(y). \end{array} \right\}$$

The program *Even_R* is obviously not stratifiable. It is neither locally stratifiable because its instantiation contains, for instance, the rule even(0) :- SUC(0, 0) & not even(0). No meaning can be assigned to this program with the tools presented in this section. However it is clear that this simple program has a clear intended meaning captured by the following interpretation: {SUC(0, 1), SUC(1, 2), ..., SUC($i-1$, i), even(0), even(2), ..., even($2j$)}. Intuitively, the negative recursion appearing in the instantiation of *Even_R* is not raising problems because it involves facts that are obviously "false". For instance, the recursion in even(0) :- SUC(0, 0) & not even(0) is not "dangerous" because SUC(0, 0) is false.

Programs that are unstratifiable are discussed in the next section and in Section 8.

Although the iterative fixpoint semantics, in general, does not apply directly to loosely stratifiable programs, we insert here a presentation of loose stratification [26] because, loose stratification and local stratification coincide for function free logic programs. One of the nice features of loose stratification is that it does not require to consider the instantiation of programs.

The definition of loose stratification uses a form of precedence graph enriched with information (unifiers) concerning the condition under which one atom depends on another one. The graph associated with a program is defined as follows.

Definition 4.19 (*Adorned dependency graph*). Let $\mathscr{P}$ be a logic program (possibly with function symbols). Let us put in V a representative of each type of atomic formulas occurring in $\mathscr{P}$ (for instance, if $P(x, y)$ and $P(z, y)$ occur in $\mathscr{P}$, they have the same representant, say $P(x_1, x_2)$ in V). It is also assumed that, in V, two distinct atomic formulas have no variables in common.

The adorned dependency graph associated with $\mathscr{P}$ is the directed graph (V, G) where $\text{At}_1 \rightarrow_{\sigma}^{\text{sign}} \text{At}_2$ is an arc in G iff there exists a rule r in $\mathscr{P}$ and a most general unifier τ such that:

- $\text{At}_1\tau = \text{head}(r)\tau$,
- sign is + if $\text{At}_2\tau$ occurs positively in $\text{prem}(r)\tau$,
  sign is − if $\text{At}_2\tau$ occurs negatively in $\text{prem}(r)\tau$, and
- σ is the restriction (possibly empty) of τ to the variables occurring in At_1 and At_2.

Loose stratification relies on adorned dependency graph as follows.

Definition 4.20 (*Loosely stratifiable logic program*). A logic program $\mathscr{P}$ is loosely stratifiable iff the adorned dependency graph associated with $\mathscr{P}$ contains no path $\mathrm{At}_1 \rightarrow_{\sigma_1}^{\mathrm{sign}_1} \mathrm{At}_2 \rightarrow_{\sigma_2}^{\mathrm{sign}_2} \mathrm{At}_3 \ldots \mathrm{At}_n \rightarrow_{\sigma_n}^{\mathrm{sign}_n} \mathrm{At}_{n+1}$ such that:

(1) there exists $i \in \{1 \ldots n\}$ such that $\mathrm{sign}_i = -$, and

(2) there exists a unifier σ which is more general than σ_i for each $i \in \{1 \ldots n\}$ such that $\mathrm{At}_{n+1}\sigma = \mathrm{At}_1\sigma$.

Example 4.21. (1) Consider the logic program *Even_F* presented in Example 4.15. The only arc in the adorned dependency graph associated with *Even_F* is $\mathrm{Even}(s(y)) \rightarrow_{[y|x]}^{+} \mathrm{Even}(x)$. Thus, because $\mathrm{Even}(s(x))$ and $\mathrm{Even}(x)$ cannot be unified, *Even_F* is loosely stratifiable.

(2) Now consider the logic program *Even_R* of Example 4.18. The vertices of the adorned dependency graph associated with *Even_R* are: Even(0), Even(z), Suc(x, y), SUC(0, 1), SUC(1, 2), and SUC($i-1, i$). We have a positive arc from Even(z) to Suc(x, y) with unifier $[z|y]$, a negative arc from Even(z) to itself with empty unifier, etc. The negative arc from Even(z) to itself implies that *Even_R* is not loosely stratifiable.

It is claimed in [26] that, for function free logic programs, local stratification and loose stratification coincide, showing that, for function free logic programs, local stratification is independent of the Herbrand instantiation of the program.

In the case of programs with functions, it is said that loose stratification relaxes local stratification. The relationship between loose stratification and local stratification is not detailed here. It is investigated in [25].

5. Well-founded semantics

This section is devoted to the presentation of the well-founded semantics of logic programs. The approach chosen here is to leave the programmer totally free of writing any program he wants but to abandon the idea that "at a minimum, a semantics for a logic program must supply an assignment of truth values to ground atomic formulas" [47].

Thus, in this section, no syntactical restriction is required on the logic programs (one is allowed to write unstratifiable logic program like the program *Even_R* of Example 4.18). The meaning of logic programs is captured by means of a partial truth assignment on the Herbrand base. The declarative semantics of a logic program may not tell whether a fact is true or whether it is false. Indeed, the concepts presented in this section are based on three valued logic. In order to avoid the formal presentation of Kleene's three valued logic [62] that may discourage the non-specialist reader, we present the approach using fixpoint techniques.

The two proposals presented here can be both motivated by the discussion provided in Section 3 and more precisely by the part of the discussion exhibiting the inadequacy of the immediate consequence operator $T_{\mathscr{P}}^{\in}$ to capture the declarative

semantics of logic programs with negation. Recall that, loosely speaking, the set membership immediate consequence operator restores to the empty interpretation its intuitive natural meaning, that is "no truth value is assigned to facts". However, as showed in Section 3, the operator $T^{\in}_{\mathscr{P}}$ suffers from its inability to derive negative facts.

The two proposals presented here share the following features:

(1) the intended meaning of a logic program is given by a partial interpretation of the f.o. notation of the program, and

(2) the set membership immediate consequence operator $T^{\in}_{\mathscr{P}}$ is combined with another operator whose aim is to overcome the problem of "producing" negative facts.

The second proposal, the well-founded semantics [115] which is presented here, can be viewed as a generalization of the first one, the weak well-founded semantics[6] [48]. The operator defined in [115] to generate negative facts is more powerful than the one defined in [48]. The presentation of these two semantics focuses on the introduction and definition of these operators called, respectively, *weak unfounded operator* and *unfounded operator.*

5.1. The weak well-founded semantics

Intuitively, the basic idea is to derive negative literals corresponding to facts that do not have a definition (extensional as well as intentional) in the logic program. This idea is very natural and simple, however its immediate application leads to a poor generalization.

Definition 5.1 (*The weak unfounded operator*). Let $\mathscr{P}$ be a logic program. The weak unfounded operator associated with $\mathscr{P}$, denoted by $w\_U_{\mathscr{P}}$, is a mapping from *Partial* into itself defined by

$$w\_U_{\mathscr{P}}(\mathscr{I}) = \left\{ \neg B \;\middle|\; \begin{array}{l} B \in \mathscr{B}, \text{ and} \\ \forall r \in \mathit{Inst\_}\mathscr{P}\ (\text{head}(r) = B \Rightarrow \exists L \in \text{body}(r) \text{ and } \neg L \in \mathscr{I}). \end{array} \right\}$$

Given a logic program $\mathscr{P}$, the weak unfounded operator $w\_U_{\mathscr{P}}$ is combined with the immediate consequence operator $T^{\in}_{\mathscr{P}}$ to constructively define the semantics of logic programs. The two operators $T^{\in}_{\mathscr{P}}$ and $w\_U_{\mathscr{P}}$ are both iteratively applied to the empty partial interpretation.

- Thus at the first iteration, the set of positive facts Pos_1 generated by applying $T^{\in}_{\mathscr{P}}$ to the empty interpretation is the set of positive facts that belong to $\mathscr{P}$. The set of negative facts Neg_1 generated by applying $w\_U_{\mathscr{P}}$ to the empty partial interpretation is the set of facts having no definition in the instantiation of $\mathscr{P}$ ($\forall r \in \mathit{Inst\_}\mathscr{P}$ $\text{head}(r) = B \Rightarrow \exists L \in \text{body}(r)$ and $\neg L \in \mathscr{I}$ with $\mathscr{I} = \emptyset$ entails that there is

[6] No particular name is given to the semantics proposed in [48]. The choice of a name here is related to the organization of the presentation.

no rule r in $Inst\_\mathscr{P}$ such that head$(r) = B$ and conversely). Now given the partial interpretation $\mathscr{I}_1 = \text{Pos}_1 \cup \text{Neg}_1$ obtained from the first iteration, the program $Inst\_\mathscr{P}$ can be simplified in a natural manner by deleting each rule which has at least one premise inconsistent with the partial interpretation $\mathscr{I}_1$ and by deleting in the remaining rules the premises that do belong to the partial interpretation $\mathscr{I}_1$. Let us call this set of rules $Inst\_\mathscr{P}_1$. We have that $Inst\_\mathscr{P}_1 = \text{DP}(\mathscr{P}, \mathscr{I}_1)$.

- Then, at the second iteration, applying $T^{\in}_{\mathscr{P}}$ to the partial interpretation $\mathscr{I}_1$ generates the set of positive facts that belong to the transformed program $Inst\_\mathscr{P}_1$, and applying $w\_U_{\mathscr{P}}$ to the partial interpretation $\mathscr{I}_1$ generates the negative facts corresponding to facts having no definition in $Inst\_\mathscr{P}_1$. And so on....

The example below is followed by the formal definition of the weak well-founded semantics of a logic program.

Example 5.2. (1) Consider the logic program $\mathscr{P}_1$ given in Example 3.14. First consider the empty partial interpretation. Note that B has no definition in $\mathscr{P}_1$. Thus we have

$$T^{\in}_{\mathscr{P}_1}(\emptyset) = \{A\}, \qquad w\_U_{\mathscr{P}_1}(\emptyset) = \{\neg B\}.$$

Consider now the partial interpretation given by the first step of "derivation", i.e. $\mathscr{I}_1 = \{A, \neg B\}$. Note that $\text{DP}(\mathscr{P}_1, \mathscr{I}_1) = \{A, C\}$ and

$$T^{\in}_{\mathscr{P}_1}(\mathscr{I}_1) = T^{\in}_{\text{DP}(\mathscr{P}_1, \mathscr{I}_1)}(\emptyset) = \{A, C\} \quad \text{and}$$

$$w\_U_{\mathscr{P}_1}(\mathscr{I}_1) = w\_U_{\text{DP}(\mathscr{P}_1, \mathscr{I}_1)}(\emptyset) = \{\neg B\}.$$

The partial interpretation $\{A, \neg B, C\}$ obtained by the second step of iteration is a complete interpretation. The weak well-founded semantics of the program $\mathscr{P}_1$ is given by the completed interpretation $\{A, \neg B, C\}$.

(2) Consider now the logic program $\mathscr{P}_2$ given in Example 3.14. First consider the empty partial interpretation. Note that each proposition occurs in the head of some rule in $\mathscr{P}_2$. Then, we have

$$T^{\in}_{\mathscr{P}_2}(\emptyset) = \{A\} \quad \text{and} \quad w\_U_{\mathscr{P}_2}(\emptyset) = \emptyset.$$

Consider now the partial interpretation given by the first step of "derivation", i.e. $\mathscr{I}_1 = \{A\}$. Note that $\text{DP}(\mathscr{P}_2, \mathscr{I}_1) = \mathscr{P}_2$ and so we have

$$T^{\in}_{\mathscr{P}_2}(\mathscr{I}_1) = \{A\} \quad \text{and} \quad w\_U_{\mathscr{P}_2}(\mathscr{I}_1) = \emptyset.$$

The weak well-founded semantics of the program $\mathscr{P}_2$ is given by the partial interpretation $\{A\}$. Clearly, the weak unfounded operator is unable to detect that the unique rule B :- B defining B is a "ghost rule" and thus that there is no effective definition for B in $\mathscr{P}_2$.

(3) Consider the program *Even_R* of Example 4.15. The weak well-founded semantics of this program happens to coincide with its intended meaning presented in Example 4.15. Indeed, at the first iteration, $w\_U_{Even\_R}$ produces the negative facts $\neg\text{SUC}(0, 0), \neg\text{SUC}(1, 0) \ldots, \neg\text{SUC}(i, i)$ and the transformation of the instantiation

of *Even_R* by DP with these negative facts plus the positive fact even(0) is a logic program without negative cycles or in other words is a stratifiable logic program.

Example 5.2 illustrates by (3) the ability of the weak well-founded semantics to give a meaning, the intended one, to programs such as *Even_R*. The example also illustrates by (2) the weakness of the approach in the sense that introduction in a program of "ghost rules" of the kind B :- B has a side effect on the semantics associated with the program.

Example 5.2 also allows one to see the different way by which negative facts are "derived" according to the iterative fixpoint semantics and according to the weak well-founded approach.

In the first case, "derivation" of negative facts is guided by the stratified structure of the program and is performed at each meta-iteration. The main thing to keep in mind is that in that framework derivation of negative facts is performed by complementation and thus it strongly depends on the derivation of positive facts.

In the case of the weak well-founded approach, negative facts are generated at each iteration and this process is guided by the syntax of the program. More importantly, one should notice that at each iteration, derivation of negative facts is totally independent of derivation of positive facts. Complementation is not needed in this framework in order to generate negative facts. This will be formally discussed later.

Note that a partial interpretation $\mathcal{I}$ may be inconsistent with the interpretation obtained by applying $w\_U_{\mathcal{P}}$ to $\mathcal{I}$. Consider the (positive) logic program $\{A$:- $B\}$ and the partial interpretation $\mathcal{I} = \{\neg A, B\}$, then $w\_U_{\mathcal{P}}(\mathcal{I}) = \{\neg A, \neg B\}$ is inconsistent with $\mathcal{I}$.

However, it is easy to check that:

Theorem 5.3 (Fitting [48]). *Let $\mathcal{P}$ be a logic program. The weak unfounded operator $w\_U_{\mathcal{P}}$ is monotonic.*

Definition 5.4 (*Weak well-founded semantics*). Let $\mathcal{P}$ be a logic program. The weak well-founded model of $\mathcal{P}$ is the least fixpoint of the operator $(T^{\in}_{\mathcal{P}} \cup w\_U_{\mathcal{P}})$ associated with $\mathcal{P}$ or equivalently, it is the limit of the sequence of partial interpretations $(T^{\in}_{\mathcal{P}} \cup w\_U_{\mathcal{P}})\uparrow\alpha$.

Considering finite logic programs without function symbols leads to the weak well-founded model of a program by at most ω application of $(T^{\in}_{\mathcal{P}} \cup w\_U_{\mathcal{P}})$.

The next remark to make here is that the weak well-founded semantics does not generalize the fixpoint (or minimal model) semantics for positive logic programs in the following sense. Although the weak well-founded semantics is able to associate a meaning to all programs and in particular to positive logic programs, the weak well-founded model of a positive logic program may not be a completed interpretation and thus it may not be isomorphic to the fixpoint semantics of the program. This is not surprising in view of the fact that originally the weak well-founded model

of a logic program was defined to be the three-valued minimal model of Clark's completion of the program. An example follows.

Example 5.5. Consider the positive logic program $\{A \text{ :- } A\}$. Its weak well-founded model is the partial empty interpretation. Note that the completion of this program $\{A \leftrightarrow A\}$ neither proves A nor $\neg A$. However, the fixpoint semantics of this positive logic program is the completed interpretation $\{\neg A\}$.

However, for a positive logic program $\mathscr{P}$, there exists a natural correspondence on the one hand, between the positive part $\text{pos}(\mathscr{M})$ of its weak well-founded model $\mathscr{M}$ and the least fixpoint of $T^{\models}_{\mathscr{P}}$, and on the other hand between the negative part $\text{neg}(\mathscr{M})$ of $\mathscr{M}$ and the greatest fixpoint of $T^{\models}_{\mathscr{P}}$.

Theorem 5.6 (Fitting [48]). *Let $\mathscr{P}$ be a positive logic program. Then for all ordinal α,*

(1) $T^{\models}_{\mathscr{P}}\uparrow\alpha = \text{pos}((T^{\in}_{\mathscr{P}} \cup w\_U_{\mathscr{P}})\uparrow\alpha)$,

(2) $T^{\models}_{\mathscr{P}}\downarrow\alpha = \text{neg}((T^{\in}_{\mathscr{P}} \cup w\_U_{\mathscr{P}})\uparrow\alpha)$.

In fact part (1) of Theorem 5.6 easily follows from Lemma 3.12 which says that $T^{\models}_{\mathscr{P}}\uparrow\alpha = T^{\in}_{\mathscr{P}}\uparrow\alpha$. The merit of Theorem 5.6 is to clearly exhibit the fundamental difference that exists between the fixpoint semantics for positive programs (or the iterative fixpoint for stratified logic programs) and the weak well-founded semantics. The difference resides in the way negation is treated. As already underlined, fixpoint semantics treats negation via complementation whereas the weak well-founded semantics treats negation via the greatest fixpoint of $T^{\models}_{\mathscr{P}}$. Thus the difference between these two semantics is simply an instance of the well-known "Herbrand gap" discussed in [75]. As discussed in [47], the well-founded approach does not solve the computability problem because while it is known that, for a positive logic program, the least ordinal α such that $T^{\models}_{\mathscr{P}}\uparrow\alpha$ is the least fixpoint of $T^{\models}_{\mathscr{P}}$ is below ω, the same result does not hold for $T^{\models}_{\mathscr{P}}\downarrow\alpha$ and the greatest fixpoint of $T^{\models}_{\mathscr{P}}$. Apt and Van Emden [4] provide a logic program (with function symbols)[7] whose immediate consequence operator is not down continuous.

5.2. The well-founded semantics

The well-founded semantics generalizes the weak well-founded semantics in the following way. The unfounded operator defined in order to generate negative facts is more powerful than the weak unfounded operator in the sense that the set of negative facts produced by applying the unfounded operator to a partial interpretation is larger than the set of negative facts obtained by applying the weak unfounded operator to the same interpretation. An intuitive presentation of the unfounded operator follows its formal definition.

Definition 5.7 (*The unfounded operator*). Let $\mathscr{P}$ be a logic program, let $\mathscr{I}$ be a partial

[7] The program contains the rules: $P(a)$:- $P(x)$ & $Q(x)$, $P(s(x))$:- $P(x)$, $Q(b)$, $Q(s(x))$:- $Q(x)$.

interpretation and $\mathscr{F}$ be a subset of $\mathscr{B}$. $\mathscr{F}$ is an unfounded set of $\mathscr{P}$ with respect to $\mathscr{I}$ iff, given any f in $\mathscr{F}$,

$$\left[(r \in Inst\_\mathscr{P} \text{ and } \text{head}(r) = f) \Rightarrow \left\{ \begin{array}{l} \exists L \in \text{body}(r), \neg L \in \mathscr{I} \\ \text{or} \\ \exists L \in \text{body}(r),\ L \in \mathscr{F} \end{array} \right\} \right]$$

The unfounded operator associated with $\mathscr{P}$, denoted by $U_{\mathscr{P}}$, is the mapping on *Partial* defined by $U_{\mathscr{P}}(\mathscr{I}) = \{\neg B \mid B \in \text{GUS}_{\mathscr{P},\mathscr{I}}\}$, where $\text{GUS}_{\mathscr{P},\mathscr{I}}$ is the greatest unfounded set of $\mathscr{P}$ with respect to $\mathscr{I}$.

Unfounded sets are closed under set union which makes the greatest unfounded set of $\mathscr{P}$ equal to the union of all unfounded sets of $\mathscr{P}$.

The intuition underlying the notion of unfounded sets with respect to a partial interpretation is the following. First of all, let us see the partial interpretation $\mathscr{I}$ as a set of assumptions. Now, given a positive fact A, A belongs to $\text{GUS}_{\mathscr{P},\mathscr{I}}$ means that for any rule in $\mathscr{P}$ that could be used in order to derive A (that is, any rule with head A), attempts to activate the rule entail making some assumption explicitly in contradiction with the assumptions contained in $\mathscr{I}$. As such, there is no reason to believe, under the assumption $\mathscr{I}$, that A can be derived from $\mathscr{P}$.

It should be noted that the definition of unfounded sets provided here, as well as in [15] is an inductive definition. Alternative constructive definitions have recently been proposed in [17, 96, 112] and shall be presented later in this section.

As for the weak well-founded approach, the unfounded operator $U_{\mathscr{P}}$ is combined with the immediate consequence operator $T^{\in}_{\mathscr{P}}$ to "evaluate" (assuming unfounded sets are defined in a constructive manner) the semantics of logic programs. In the "evaluation" of the well-founded semantics of the logic program $\mathscr{P}$, the operators $T^{\in}_{\mathscr{P}}$ and $U_{\mathscr{P}}$ are both iteratively applied to the empty partial interpretation.

The example below is followed by the formal definition of the well-founded semantics of a logic program.

Example 5.8. (1) Consider the logic program $\mathscr{P}_1$ given in Example 3.14. Because the unfounded operator $U_{\mathscr{P}_1}$ "subsumes" the weak unfounded operator $w\_U_{\mathscr{P}_1}$, it is not surprising to have that the weak well-founded semantics of $\mathscr{P}_1$ given by the completed interpretation $\{A, \neg B, C\}$ coincides with the well-founded semantics of $\mathscr{P}_1$.

$$T^{\in}_{\mathscr{P}_1}(\emptyset) = \{A\}, \qquad U_{\mathscr{P}_1}(\emptyset) = w\_U_{\mathscr{P}_1}(\emptyset) = \{\neg B\},$$
$$T^{\in}_{\mathscr{P}_1}(\{A, \neg B\}) = \{A, C\}, \qquad U_{\mathscr{P}_1}(\{A, \neg B\}) = \{\neg B\}.$$

(2) The well-founded model of the program *Even_R* of Example 4.18 is also equal to the weak well-founded model of *Even_R*. Recall that the logic program *Even_R* is unstratifiable.

(3) As for the logic program $\mathscr{P}_2$ given in Example 3.14, the well-founded semantics strictly subsumes the weak unfounded semantics : truth values of the kind True or

False are not assigned neither to B nor to C by the weak well-founded semantics; the well-founded truth values of B and C are respectively false and true.

Note that the greatest unfounded set of $\mathscr{P}_2$ with respect to the empty interpretation is $\{B\}$ and thus, we have

$$T^{\in}_{\mathscr{P}_2}(\emptyset) = \{A\}, \qquad U_{\mathscr{P}_2}(\emptyset) = \{\neg B\},$$

$$T^{\in}_{\mathscr{P}_2}(\{A, \neg B\}) = \{A, C\}, \qquad U_{\mathscr{P}_2}(\{A, \neg B\}) = \{\neg B\}.$$

Note that under the well-founded semantics the programs $\mathscr{P}_1$ and $\mathscr{P}_2$ are equivalent. This equivalence is natural if one considers the rule B :- B as a "ghost rule", that is, as a rule adding no information to $\mathscr{P}_1$.

(4) Consider now the program $\mathscr{P}_4$ given in Example 3.14. The well-founded model of $\mathscr{P}_4$ is a partial model. Note that the greatest unfounded set of $\mathscr{P}_4$ with respect to the empty interpretation is the empty set of facts. This implies that the well-founded model of $\mathscr{P}_4$ is the empty partial interpretation. In other words, the meaning of $\mathscr{P}_4$ is defined but "unknown".

We now formally present the well-founded semantics for logic programs.

Theorem 5.9 (VanGelder et al. [115]). *Let $\mathscr{P}$ be a logic program. The unfounded operator $U_{\mathscr{P}}$ associated with $\mathscr{P}$ is monotonic.*

Definition 5.10 (*The well-founded semantics*). Let $\mathscr{P}$ be a logic program. The well-founded model of $\mathscr{P}$ is the least fixpoint of the operator $(T^{\in}_{\mathscr{P}} \cup U_{\mathscr{P}})$ associated with $\mathscr{P}$ or equivalently, it is the limit of the sequence of partial interpretations $(T^{\in}_{\mathscr{P}} \cup U_{\mathscr{P}})\uparrow\alpha$.

We now focus our attention on a constructive definition of unfounded sets which leads to constructive definitions of the well-founded semantics for logic programs.

In [17], the notion of a potentially founded set of facts is introduced which is the dual of the notion of unfounded sets. As a matter of fact, given a logic program $\mathscr{P}$, a potentially founded set is defined in [17] as the complement (with respect to the Herbrand base) of the greatest unfounded set of $\mathscr{P}$ with respect to the *empty partial interpretation.* The notion of potentially founded facts is defined below with respect to partial interpretations (not necessarily empty ones).

Definition 5.11 (*Potentially founded set*).[8] Let $\mathscr{P}$ be a logic program, let $\mathscr{I}$ be a partial interpretation and let $\mathscr{F}$ be a subset of $\mathscr{B}$. $\mathscr{F}$ is a potentially founded set of $\mathscr{P}$ with respect to $\mathscr{I}$ iff,

$$[\exists r \in \mathit{Inst\_}\mathscr{P} \text{ such that } \forall L \in \text{pos}(\text{body}(r)), (L \in F, \text{ or } \neg L \notin \mathscr{I})]$$

$$\Rightarrow \text{head}(r) \in \mathscr{F}.$$

[8] This definition has been proposed in a revised version of [17].

Intuitively, potentially founded facts are facts that can be derived from the rules of the program while declaring true all negative premises as soon as these additional assumptions are not in contradiction with the interpretation $\mathcal{I}$.

Potentially founded sets of $\mathcal{P}$ with respect to $\mathcal{I}$ are closed under intersection. We are naturally interested in the *smallest potentially founded* set of $\mathcal{P}$ with respect to $\mathcal{I}$ (intersection of all potentially founded sets of $\mathcal{P}$ with respect to $\mathcal{I}$), denoted by $\mathrm{SPF}_{\mathcal{P},\mathcal{I}}$ for which [17] provides the following constructive definition.

Lemma 5.12. *Let $\mathcal{P}$ be a logic program and $\mathcal{I}$ be a partial interpretation. Consider the sequence $(\mathrm{SPF}_i)_{(i\geq 0)}$ of sets of facts defined by*

$$\mathrm{SPF}_0 = \{\mathrm{head}(r) \mid r \in \mathit{Inst\_}\mathcal{P}\ \mathrm{pos}(\mathrm{body}(r)) = \emptyset \text{ and } \forall L \in \mathrm{body}(r) \neg L \notin \mathcal{I}\},$$

$$\mathrm{SPF}_{i+1} = \{\mathrm{head}(r) \mid r \in \mathit{Inst\_}\mathcal{P}\ \mathrm{pos}(\mathrm{body}(r)) \subseteq \mathrm{SPF}_i \text{ and } \forall L \in \mathrm{body}(r) \neg L \notin \mathcal{I}\}.$$

Let us denote SPF_∞ the first element SPF_i of the sequence satisfying $\mathrm{SPF}_i = \mathrm{SPF}_{i+1}$. Then $\mathrm{SPF}_{\mathcal{P},\mathcal{I}} = \mathrm{SPF}_\infty$.

Another equivalent way to define SPF_∞ is provided by:

(1) first, consider the logic program $\mathcal{P}'$ obtained by removing from $\mathcal{P}$ the rules with at least one premise inconsistent with $\mathcal{I}$, i.e.

$$\mathcal{P}' = \{r \mid r \in \mathit{Inst\_}\mathcal{P} \text{ and } \exists L \in \mathrm{body}(r) \neg L \in \mathcal{I}\},$$

(2) now, consider the positive logic program $\mathcal{P}_{\mathrm{pos}}$ obtained by removing the negative premises in the rules of $\mathcal{P}'$, i.e.

$$\mathcal{P}_{\mathrm{pos}} = \{r \mid \exists r' \in \mathcal{P}'\ \mathrm{head}(r) = \mathrm{head}(r') \text{ and } \mathrm{body}(r) = \mathrm{pos}(\mathrm{body}(r'))\}.$$

Then, SPF_∞ is simply the least fixpoint of the immediate consequence operator associated with the positive logic program $\mathcal{P}_{\mathrm{pos}}$($\mathrm{SPF}_\infty = \mathrm{lfp}(T^{\models}_{\mathcal{P}_{\mathrm{pos}}}) = \mathrm{lfp}(T^{\in}_{\mathcal{P}_{\mathrm{pos}}})$).

The notion of potentially founded sets has been introduced as the dual notion of unfounded sets. The next result confirms that the greatest unfounded set of $\mathcal{P}$ with respect to $\mathcal{I}$ can be defined in terms of the smallest potentially founded set of $\mathcal{P}$ with respect to $\mathcal{I}$ as follows.

Theorem 5.13 (Bidoit and Froidevaux [17]). *Let $\mathcal{P}$ be a logic program and $\mathcal{I}$ be a partial interpretation. Then $\mathrm{GUS}_{\mathcal{P},\mathcal{I}} = \mathcal{B} - \mathrm{SPF}_{\mathcal{P},\mathcal{I}}$.*

A constructive definition of the well-founded model of a logic program follows immediately from the initial definition of a well-founded model, the theorem above and the constructive definition of a smallest potentially founded set.

In fact the constructive definitions of a well-founded model proposed in [17, 112, 96] all differ slightly from this straightforward approach. We review below the proposals in [17, 112].

Until now, we have characterized two kinds of (sets of) propositions, namely unfounded and potentially founded propositions. It is very natural to introduce a third kind of propositions that are called founded and that intuitively correspond to facts that can be derived from the rules in a program and the facts in a partial interpretation without making any additional assumptions.

Definition 5.14 (*Founded set*). Let $\mathscr{P}$ be a logic program and let $\mathscr{I}$ be a partial interpretation. The set of founded facts of $\mathscr{P}$ with respect to $\mathscr{I}$, denoted by $F_{\mathscr{P},\mathscr{I}}$ is defined by $F_{\mathscr{P},\mathscr{I}} = \mathrm{lfp}(T^{\in}_{\mathrm{DP}(\mathscr{P},\mathscr{I})})$.

The constructive definition of a well-founded model provided in [17] is based on:

- on the one hand, the notion of founded facts defined above and the notion of unfounded facts as defined by Lemma 5.12 and Theorem 5.13 (the particularity of the approach is that these two notions are restricted to the case of empty partial interpretation),
- on the other hand, a simple iterative transformation of logic programs that, given a logic program, constructs an equivalent[9] logic program.

The underlying idea is quite simple. Founded facts (respectively, unfounded facts) of $\mathscr{P}$ with respect to the empty partial interpretation are true (respectively, false) for $\mathscr{P}$. More formally, it is shown in [17] that $F_{\mathscr{P},\emptyset} \cup \neg.\mathrm{GUS}_{\mathscr{P},\emptyset} \subseteq \mathscr{M}$ where $\mathscr{M}$ is the well-founded model of $\mathscr{P}$.

Thus, as soon as we get the sets of founded and unfounded facts of $\mathscr{P}$ with respect to the empty partial interpretation, the logic program $\mathscr{P}$ can be transformed into a "simpler" equivalent logic program as follows:

- rules with premise $\neg f$ (resp. with premise f), where f is a founded fact (resp. unfounded fact), can be deleted from $\mathscr{P}$, and
- premises f (resp. $\neg f$) where f is founded (resp. unfounded) can be removed from the rules in $\mathscr{P}$.

This "simpler" logic program is nothing else than the logic program $\mathrm{DP}(\mathscr{P}, F_{\mathscr{P},\emptyset} \cup \neg.\mathrm{GUS}_{\mathscr{P},\emptyset})$. This program is denoted by $\mathrm{EFF}_{\mathscr{P}}$ in the following.

If the program $\mathrm{EFF}_{\mathscr{P}}$ is not trivial (is not a set of facts), we can iterate the process by computing the set of founded and unfounded facts of $\mathrm{EFF}_{\mathscr{P}}$ with respect to the empty interpretation. Thus consider the sequence $\mathrm{EFF}_{\mathscr{P}}{\downarrow}i_{(i \geq 0)}$ defined by

$$\mathrm{EFF}_{\mathscr{P}}{\downarrow}0 = \mathscr{P} \quad \text{and} \quad \mathrm{EFF}_{\mathscr{P}}{\downarrow}(i+1) = \mathrm{EFF}_{\mathrm{EFF}_{\mathscr{P}}{\downarrow}i} \quad \text{for } i > 0.$$

Intuitively, the $(i+1)$th element of the sequence contains less rules than the ith element as well as the rules remaining in the $(i+1)$th element of the sequence contain less premises than the rules in the ith element. Thus, at some point a program is obtained that contains a minimal number of rules and these rules are "as small as possible".

In [17], it is shown that:

[9] With respect to the well-founded semantics and also with respect to the default semantics presented in Section 8.

Theorem 5.15. *Let $\mathscr{P}$ be a logic program.*

(1) *There exists i such that* $\mathrm{EFF}_{\mathscr{P}}\downarrow i+1 = \mathrm{EFF}_{\mathscr{P}}\downarrow i$. *Let us denote* $\mathrm{EFF}_{\mathscr{P}}\downarrow\omega$[10] *the logic program* $\mathrm{EFF}_{\mathscr{P}}\downarrow i$ *where i is the smallest integer such that* $\mathrm{EFF}_{\mathscr{P}}\downarrow i+1 = \mathrm{EFF}_{\mathscr{P}}\downarrow i$.

(2) *The well-founded model $\mathscr{M}$ of $\mathscr{P}$ is equal to the union of the greatest unfounded set of* $\mathrm{EFF}_{\mathscr{P}}\downarrow\omega$ *with respect of the empty partial interpretation and of the founded set of* $\mathrm{EFF}_{\mathscr{P}}\downarrow\omega$ *with respect to the empty partial interpretation. That is* $\mathscr{M} = \neg.\mathrm{GUS}_{\mathrm{EFF}_{\mathscr{P}}\downarrow\omega,\emptyset} \cup F_{\mathrm{EFF}_{\mathscr{P}}\downarrow\omega,\emptyset}$.

The aim of [17] is to define a class of logic programs larger than the class of (locally) stratifiable programs and having a "natural" intended meaning. Intuitively, a "good" logic program is a program which can be transformed into an equivalent stratifiable logic program. Formally, effectively stratifiable logic programs are defined in [17] by:

Definition 5.16 (*Effectively stratifiable logic program*). Let $\mathscr{P}$ be a logic program (over a language containing no function symbols other than constant symbols). $\mathscr{P}$ is effectively stratifiable iff there exists i such that the logic program $\mathrm{EFF}_{\mathscr{P}}\downarrow i$ is locally stratifiable.

Note that in the definition above $\mathrm{EFF}_{\mathscr{P}}\downarrow i$ is a finite program and it is equal to its instantiation. Thus, in this particular case local stratification can be checked in polynomial time.

The next result formalizes the fact that effectively stratifiable programs have a natural intended meaning.

Theorem 5.17 (Bidoit and Froidevaux [17]). *Let $\mathscr{P}$ be a logic program. The two following assertions are equivalent*:

(1) *$\mathscr{P}$ is effectively stratifiable,*

(2) *the well-founded model of $\mathscr{P}$ is total* (*i.e. is a completed interpretation*).

The following example aims to illustrate the definition of potentially founded facts, founded facts, the notion of effective stratification and the related results.

Example 5.18. Let us consider the logic program *Even_R* presented in Example 4.18. Table 2 shows the smallest potentially founded set, the greatest unfounded set and the founded set of *Even_R* with respect to the empty partial interpretation. Note that the smallest potentially founded set is equal to the least fixpoint of the immediate consequence operator associated with the positive logic program:

$$Even\_R\_pos = \left\{ \begin{array}{l} \mathrm{SUC}(0,1), \mathrm{SUC}(1,2), \ldots, \mathrm{SUC}(i-1,i), \\ \mathrm{even}(0), \\ \mathrm{even}(x) \text{ :- } \mathrm{SUC}(y,x). \end{array} \right\}$$

[10] This notation does not refer to a greatest fixpoint.

Table 2

Potentially founded facts	Unfounded facts	Founded facts
SUC(0, 1)	SUC(0, 0)	SUC(0, 1)
SUC(1, 2)	SUC(0, 2)	SUC(1, 2)
. . .	. . .	. . .
SUC($i-1$, i)	SUC(0, i)	SUC($i-1$, i)
even(0)	. . .	even(0)
even(1)	SUC(i, i)	
. . .		
even(i)		

Actually, the founded set contains the facts in the least fixpoint of the set membership operator associated with *Even_R*.
Let us call $\mathcal{I}_1$ the partial interpretation formed by the founded and unfounded sets of *Even_R* with respect to the empty partial interpretation. $\mathcal{I}_1$ can be used in order to simplify the (instantiation of) the logic program *Even_R*. The program obtained is

$$\mathrm{EFF}_{Even\_R} = \left\{ \begin{array}{l} \mathrm{SUC}(0,1), \mathrm{SUC}(1,2), \ldots, \mathrm{SUC}(i-1,i), \\ \mathrm{even}(0), \\ \mathrm{even}(1) \text{ :- not even}(0), \\ \\ \mathrm{even}(2) \text{ :- not even}(1), \\ \ldots \\ \mathrm{even}(i) \text{ :- not even}(i-1). \end{array} \right\}$$

Note that the logic program EFF_{Even_R} is stratifiable and thus the logic program *Even_R* is effectively stratifiable. The well-founded semantics of *Even_R* is equal to the well-founded semantics of EFF_{Even_R} and to the iterative least fixpoint semantics of EFF_{Even_R}.

Table 3 gives the smallest potentially founded set, the greatest unfounded set and the founded set of EFF_{Even_R} with respect to the empty partial. Note that the greatest unfounded set of EFF_{Even_R} with respect to the empty partial interpretation corresponds to the greatest unfounded set of the initial program *Even_R* with respect to the partial interpretation $\mathcal{I}_1$.

The only changes between Tables 2 and 3 is that the fact even(1) has moved from potentially founded to unfounded. No change is to be noticed in Founded Facts.

Let us call $\mathcal{I}_2$ the partial interpretation formed by the founded and unfounded facts of EFF_{Even_R} with respect to the empty partial interpretation.

Once again, we can simplify the logic program EFF_{Even_R} using $\mathcal{I}_2$. The new logic program has a smallest potentially founded set (respectively, greatest unfounded set) with respect to the empty partial interpretation equal to the preceding ones. The only change that occurs at this step is the introduction of the fact even(2) in the Founded Facts.

Table 3

Potentially founded facts	Unfounded facts	Founded facts
SUC(0, 1)	SUC(0, 0)	SUC(0, 1)
SUC(1, 2)	SUC(0, 2)	SUC(1, 2)
...	...	...
SUC($i-1$, i)	SUC(0, i)	SUC($i-1$, i)
even(0)	...	even(0)
even(2)	SUC(i, i)	
...	even(1)	
even(i)		

The courageous reader is free to check that iterating the process outlined here leads to the computation of the well-founded semantics of the logic program *Even_R*.

The notion of an effectively stratifiable program has been independently proposed in [96], where it is called a dynamically stratifiable logic program. Indeed, an interesting constructive definition of a well-founded model is also presented there.

In a different but related context (the methodology of software development in the framework of logic programming), [40] defines a condition analogous to effective stratification through the notion of well-founded semi-proof trees.

The term "dynamic" is particularly appropriate. Stratification and local stratification are syntactical constraints and can be checked directly on the logic program. They are independent of the data (facts) contained in the program. On the contrary, effective (or dynamic) stratification is strongly dependent on the data in the program: the facts in the program are needed in order to check whether that program is dynamically stratifiable.

Checking whether a logic program is effectively (or dynamically) stratifiable may lead, in the worst case, to computing the well-founded model of that program.

The next example illustrates this remark.

Example 5.19. Consider the following logic program:

$$\mathscr{P} = \left\{ \begin{array}{l} \text{SUC}(0, 1), \\ \text{SUC}(1, 0), \\ \text{even}(x) \text{ :- SUC}(y, x) \ \& \ \text{not even}(y). \end{array} \right\}$$

Note that this program is the program *Even_R* in which facts about the relation SUC have been changed. Now, note also that $\text{EFF}_{\mathscr{P}}{\downarrow}\omega$ is the logic program

$$\left\{ \begin{array}{l} \text{SUC}(0, 1), \\ \text{SUC}(1, 0), \\ \text{even}(0) \text{ :- SUC}(1, 0) \ \& \ \text{not even}(1), \\ \text{even}(0) \text{ :- SUC}(0, 1) \ \& \ \text{not even}(0), \end{array} \right\}$$

which is not locally stratifiable. Thus $\mathscr{P}$ is not effectively stratifiable. As a matter of fact, the well-founded model of $\mathscr{P}$ is the partial interpretation {SUC(0, 1), SUC(1, 0), ¬SUC(0, 0), ¬SUC(1, 1)} which gives no truth value to even(0) and even(1).

We now expose the constructive definition of a well-founded model proposed in [112]. Interestingly, [112] makes use of the notions of founded set and of its complement. The particularity of this approach is to consider negative partial interpretations. A new operator is introduced.

Definition 5.20. Let $\mathscr{P}$ be a logic program. The operator $\tilde{S}_{\mathscr{P}}$ associated with $\mathscr{P}$ is defined from $2^{\neg.\mathscr{B}}$ into $2^{\neg.\mathscr{B}}$ by $\tilde{S}_{\mathscr{P}}(\mathscr{I}) = \neg(\mathscr{B} - F_{\mathscr{P},\mathscr{I}})$ where $\mathscr{I}$ is a set of negative literals.

Intuitively, in [112] the operator $\tilde{S}_{\mathscr{P}}$ is used in order to generate the negative part neg($\mathscr{M}$) of the well-founded model $\mathscr{M}$ of a logic program. An intuitive description of the behavior of $\tilde{S}_{\mathscr{P}}$ is helpful to understand the next definition and result.

Remark 5.21. Let $\mathscr{P}$ be a logic program and let $\mathscr{M}$ be the well-founded model of $\mathscr{P}$. The set of positive facts not occurring (either positively or negatively) in $\mathscr{M}$ is denoted by unknown $(\mathscr{M}) = \mathscr{B} - (\text{pos}(\mathscr{M}) \cup \neg.\text{neg}(\mathscr{M}))$. Now let us consider a set of negative literals $\mathscr{I}$. Two complementary cases are interesting to examine:

(1) Assume that $\mathscr{I}$ is a subset of neg($\mathscr{M}$). Then $\tilde{S}_{\mathscr{P}}(\mathscr{I})$ happens to be a superset of neg($\mathscr{M}$) ∪ ¬.unknown($\mathscr{M}$).

(2) Assume now that $\mathscr{I}$ is a superset of neg($\mathscr{M}$) ∪ ¬.unknown($\mathscr{M}$). Then $\tilde{S}_{\mathscr{P}}(\mathscr{I})$ happens to be a subset of neg($\mathscr{M}$).

So, intuitively speaking, the sequence $\mathscr{I}_0 = \emptyset$, $\mathscr{I}_1 = \tilde{S}_{\mathscr{P}}(\mathscr{I}_0), \ldots, \mathscr{I}_{i+1} = \tilde{S}_{\mathscr{P}}(\mathscr{I}_i)$ obtained by iteratively applying $\tilde{S}_{\mathscr{P}}$ starting with the empty set of negative facts alternates in the following sense (note that trivially $\mathscr{I}_0 = \emptyset$ is a subset of neg($\mathscr{M}$)):

(1) each even element of the sequence is an underestimation of neg($\mathscr{M}$), and

(2) each odd element of the sequence is an overestimation of neg($\mathscr{M}$) ∪ unknown($\mathscr{M}$).

Moreover,

(1) the subsequence of even elements of $\mathscr{I}_{i(i\geq 0)}$ is an increasing sequence, and

(2) the subsequence of odd elements of $\mathscr{I}_{i(i\geq 0)}$ is a decreasing sequence.

Assume that the sequence stabilizes for $2k$, that is $\mathscr{I}_{2k+2} = \mathscr{I}_{2k}$ and $\mathscr{I}_{2(k+1)+1} = \mathscr{I}_{2k+1}$. Then, on the one hand $\mathscr{I}_{2k}$ is a (maximal) underestimation of neg($\mathscr{M}$) and on the other hand $\mathscr{I}_{2k+1}$ is a (minimal) overestimation of neg($\mathscr{M}$) ∪ ¬.unknown($\mathscr{M}$). In other words, one expects that $\mathscr{I}_{2k} = \text{neg}(\mathscr{M})$, $\mathscr{I}_{2k+1} = \text{neg}(\mathscr{M}) \cup \neg.\text{unknown}(\mathscr{M})$ and moreover that $\mathscr{M} = \mathscr{I}_{2k} \cup (B - \neg.\mathscr{I}_{2k+1})$.

This is formally stated in [112] by:

Definition 5.22 (*Alternating fixpoint of a logic program*). Let $\mathscr{P}$ be a logic program. The alternating fixpoint of $\mathscr{P}$ is the least fixpoint of the monotonic operator $A_{\mathscr{P}}$ defined from $2^{\neg\,\mathscr{B}}$ into $2^{\neg.\mathscr{B}}$ by $A_{\mathscr{P}}(\mathscr{I}) = \tilde{S}_{\mathscr{P}}(\tilde{S}_{\mathscr{P}}(\mathscr{I}))$ where $\mathscr{I}$ is a set of negative facts.

Theorem 5.23 (Van Gelder [112]). *Let $\mathscr{P}$ be a logic program and let $\mathscr{A}^-$ be the least fixpoint of $A_{\mathscr{P}}$. Let $\mathscr{A}^+ = F_{\mathscr{P},\mathscr{A}^-}$. The well-founded model of $\mathscr{P}$ is equal to $\mathscr{A}^- \cup \mathscr{A}^+$.*

The relationship between the alternative operator and the notion of smallest potentially founded set highlights the relationship between the constructive definition of well-founded model in [17] and the one in [112].

Lemma 5.24. *Let $\mathscr{P}$ be a logic program and $\mathscr{I}$ be a (negative) partial interpretation.*

(1) $F_{\mathscr{P},\tilde{S}_{\mathscr{P},\mathscr{I}}} = \mathrm{SPF}_{\mathscr{P},\mathscr{I}}$, *and thus*

(2) $A_{\mathscr{P}}(\mathscr{I}) = S_{\mathscr{P}}(S_{\mathscr{P}}(\mathscr{I})) = \mathrm{GUS}_{\mathscr{P},\mathscr{I}}$.

It is important to note that although it is hidden in the inductive definition of the unfounded operator, complementation is needed in order to compute the greatest unfounded set of $\mathscr{P}$ with respect to $\mathscr{I}$. This appears very explicitly in [17] as well as in [112]. This remark also applies to [96] although indirectly. In [96], the inverse inclusion is considered between the negative parts of partial interpretations and thus the least fixpoint which gives the negative part of the well-founded model of a logic program is computed by iterative application of some operator starting with the Herbrand Base.

The end of this section is devoted to the minimality and supportedness properties of the well-founded semantics. Of course, since the well-founded model of a logic program is a partial model, these properties need to be slightly modified.

Given a logic program $\mathscr{P}$ and a partial interpretation $\mathscr{I}$ of (the f.o. notation) $\mathscr{P}$:

(1) $\mathscr{I}$ is a (partial) minimal model of (the f.o. notation of) $\mathscr{P}$ iff there exists a model of $\mathscr{P}_{\text{f.o.}}$ which is an extension of $\mathscr{I}$ and a minimal model of $\mathscr{P}_{\text{f.o.}}$,

(2) $\mathscr{I}$ is a (partial) supported model of $\mathscr{P}$ iff $\mathrm{pos}(\mathscr{I}) \subseteq T^{\models}_{\mathscr{P}}(\mathscr{I})$.

Then, we have:

Lemma 5.25. *Let $\mathscr{P}$ be a logic program. Then*:

(1) *the well-founded model of $\mathscr{P}$ is a (partial) minimal model of $\mathscr{P}_{\text{f.o.}}$,*

(2) *the well-founded model of $\mathscr{P}$ is a (partial) supported model of $\mathscr{P}$, and moreover,*

(3) *the well-founded model $\mathscr{M}$ of $\mathscr{P}$ satisfies the property*: $\mathrm{pos}(\mathscr{M}) \subseteq T^{\in}_{\mathscr{P}}(\mathscr{M})$.

6. Inflationary semantics

In this section, we present the inflationary semantics of logic programs. Inflationary (or cumulative) fixpoints are investigated in [57] and its application to the definition of the semantics of logic programs is studied in [8, 66]. The motivation given in [8] for introducing and studying this semantics as a "new and appealing" semantics for logic programs with negation is different from the one given in [66].

In [8, 7, 9] *database transformations*, that is database queries and database updates, are investigated. A variety of database languages, that is query languages and update

languages, are proposed and analyzed in detail at a fundamental level. In particular, different families of languages are characterized. For instance, the choice of a non-deterministic semantics versus a deterministic semantics is discussed.

Roughly speaking, in the context of logic programs, deterministic fixpoint semantics corresponds to "apply ALL rules" at once, while non-deterministic semantics corresponds rather to "apply ONE rule" at a time. These two different ways of "evaluating" logic programs can be illustrated on programs with negation.

Example 6.1. Consider the logic program

$$\mathscr{P} = \begin{cases} A \text{ :- not } B \\ B \text{ :- not } A \end{cases} \begin{matrix} (r_1) \\ (r_2) \end{matrix}$$

(1) Deterministic "evaluation" of the inflationary (cumulative) semantics of $\mathscr{P}$: As usual, let us consider the empty Herbrand interpretation and let us "fire", at once, all rules that apply, that is all rules whose premises are satisfied by the empty interpretation. Here, both rules (r_1) and (r_2) have their premise satisfied by the empty interpretation. This leads, at once, to the two facts A and B. The intermediate evaluation of the semantics of $\mathscr{P}$ is $\{A, B\}$. Given the set of facts $\{A, B\}$, we proceed in the same way. No facts are produced because neither (r_1) nor (r_2) can be activated. Thus no fact can be added to the intermediate semantics $\{A, B\}$. Thus the deterministic inflationary semantics of $\mathscr{P}$ is given by the Herbrand interpretation $\{A, B\}$.

(2) Non-deterministic "evaluation" of the inflationary semantics of $\mathscr{P}$: As above, let us consider the empty interpretation and let us choose, non-deterministically, one rule of $\mathscr{P}$ that applies, that is one rule whose premises are satisfied by the empty interpretation.

- Let us choose the rule r_1. The fact produced by r_1 is A and thus the intermediate evaluation of the semantics of $\mathscr{P}$ is $\{A\}$. Given the set of facts $\{A\}$, we proceed in the same way. No rule can be activated and thus no fact can be added to the previous intermediate semantics. Thus a non-deterministic semantics of $\mathscr{P}$ is given by the Herbrand interpretation $\{A\}$.
- Now let us choose the rule r_2. It is easy to check that, in this case, the non-deterministic interpretation "evaluated" is $\{B\}$ which gives another non-deterministic inflationary semantics of $\mathscr{P}$.

In [8], connections between procedural database languages and declarative database languages are exhibited that suggest how explicit control can be used in conjunction with declarative languages.

Concerning logic programs with negation (Datalog with negation), the inflationary fixpoint semantics is introduced in [8] as a deterministic, declarative (and strongly safe) update language.

In [66], the interest taken in inflationary fixpoint semantics is essentially motivated by exhibiting "compelling complexity-theoretic obstacles" that lead to inefficient

implementation of classical fixpoint semantics (based on the immediate consequence operator $T^{\models}_{\mathscr{P}}$). Let us briefly present these results here.

First of all, a tight correspondence is established between problems in NP and logic programs with negation. The result in [66] uses the connection between computability and second order formula established by [46]. Shortly, a database, that is considered to be a Herbrand interpretation of the language $\mathscr{L}$ in our context, is NP computable iff it is definable by an existential second order formula over $\mathscr{L}$. The first result[11] of [66] implies that:

Theorem 6.2. *If M is a* NP*-computable database, then there exists a logic program $\mathscr{P}$ with negation such that the immediate consequence operator associated with $\mathscr{P} \cup M$ has a fixpoint.*

In conclusion, the existence of a fixpoint for $T^{\models}_{\mathscr{P}}$ is a NP-complete problem. A similar connection is established between program and the class US (unique solution) whose prototypical problem is UNIQUE SAT (given a Boolean formula does it have a unique satisfying assignment?).

This second result can be summarized by: the existence of a unique fixpoint for $T^{\models}_{\mathscr{P}}$ where $\mathscr{P}$ is a logic program with negation is a US (unique solution) problem. Uniqueness of fixpoint for $T^{\models}_{\mathscr{P}}$ is a desirable situation since obviously it implies the existence of a least fixpoint for $T^{\models}_{\mathscr{P}}$.

Thus it is not surprising that the existence of a least fixpoint for $T^{\models}_{\mathscr{P}}$ is harder than US. In [66], this problem is shown to range between the class US and the class $\mathrm{FO}_{\mathrm{NP}}$ (first order with NP-oracles) of problems. This technical result is not detailed here. We limit ourselves to provide the prototypical $\mathrm{FO}_{\mathrm{NP}}$ problem given in [66]. Given a graph $G = (V, E)$, is there an edge $E(x, y)$ such that if this edge is removed then the resulting graph is 3-colorable but not Hamiltonian?

Inflationary semantics of logic programs is introduced in [66] as a natural extension of the standard fixpoint semantics for positive logic programs with the advantage of overcoming the computational obstacles of standard fixpoint semantics of logic programs with negation.

A common motivation links both presentations [8, 66] which can be summarized by: developing an extension of Datalog (the language corresponding syntactically to positive logic programs with the least fixpoint semantics) with increased expressive power. Indeed, logic programs with inflationary semantics leads to first order+ fixpoint queries [8, 66]. Results on the expressive power of logic programming languages are provided in Section 9.

Inflationary semantics of logic programs is a natural technical extension of the classical fixpoint semantics. From this point of view, this apparoach can be compared with the iterative fixpoint semantics for stratifiable logic programs.

[11] We do not present Theorem 1 of [66] because it would require to make an explicit separation between database predicate symbols and non-database predicate symbols.

However, whereas the iterative fixpoint semantics is defined for the class of stratifiable logic programs exclusively and thus is unable to assign a meaning to all logic programs with negation, inflationary semantics has the advantage that it gives a meaning to all logic programs. From this point of view, the inflationary approach can be compared with the well-founded semantics.

The analogy is limited by the following. While the well founded semantics of a logic program presented in Section 5 is a partial Herbrand interpretation (or a three-valued interpretation), the inflationary semantics of a logic program is always a Herbrand interpretation (bivalued interpretation).

Recall that the well founded model of a logic program is a total Herbrand interpretation iff that program is effectively (or dynamically) stratifiable. Indeed, the analogy between the well founded semantics and the inflationary semantics collapses completely when one compares the meaning assigned by each of these approaches to an effectively stratifiable program.

The main point as we will see later, is that, while the inflationary approach is concerned with the computational (and thus implementation) and the expressive power issues, it is not so concerned with the "common sense"[12] issue.

We now proceed to the formal presentation of the inflationary semantics of logic programs. The operator applied in order to generate new facts is simply the cumulative operator obtained from the immediate consequence operator $T^{\models}_{\mathscr{P}}$. In the following, this operator is called the inflationary immediate consequence operator. It should be pointed out that the inflationary immediate consequence operator is applied to Herbrand interpretations. Thus, we are back in the situation where the empty interpretation represents an assignment of each truth value of positive facts to false.

Definition 6.3 (*The inflationary immediate consequence operator*). Let $\mathscr{P}$ be a logic program. The inflationary immediate consequence operator associated with $\mathscr{P}$, denoted by $\text{Inf\_}T^{\models}_{\mathscr{P}}$, is the mapping defined on **Int** by

$$\text{Inf\_}T^{\models}_{\mathscr{P}}(I) = I \cup T^{\models}_{\mathscr{P}}(I) \text{ for } I \in \mathbf{Int}.$$

The operator $\text{Inf\_}T^{\models}_{\mathscr{P}}$ is not monotonic. It suffices to consider the logic program $\{A \text{ :- not } B\}$ and the two interpretations $I_1 = \emptyset$ and $I_2 = \{B\}$. Although $I_1 \subseteq I_2$, $\text{Inf\_}T^{\models}_{\mathscr{P}}(I_1) = \{A\} \not\subseteq \text{Inf\_}T^{\models}_{\mathscr{P}}(I_2) = \{B\}$. However, the inflationary immediate consequence operator associated with a logic program $\mathscr{P}$ is inflationary that is:

$$I \subseteq \text{Inf\_}T^{\models}_{\mathscr{P}}(I) \text{ for } I \in \mathbf{Int}.$$

This property ensures the existence of a unique fixpoint for $\text{Inf\_}T^{\models}_{\mathscr{P}}$ where $\mathscr{P}$ is a logic program (over a language without function symbols other than constant

[12] The natural, intended meaning of programs.

symbols). Moreover, the fixpoint of $\text{Inf\_}T^{\vDash}_{\mathscr{P}}$ is of the form $\text{Inf\_}T^{\vDash}_{\mathscr{P}}\uparrow i$ for some $i \geqslant 0$ where the sequence of interpretations $(\text{Inf\_}T^{\vDash}_{\mathscr{P}}\uparrow i)_{i\geqslant 0}$ is defined as usual.

The inflationary semantics of $\mathscr{P}$ is naturally defined by:

Definition 6.4 (*The inflationary semantics of logic programs*). Let $\mathscr{P}$ be a logic program. The inflationary model of $\mathscr{P}$ is the fixpoint of the inflationary immediate consequence operator $\text{Inf\_}T^{\vDash}_{\mathscr{P}}$ associated with $\mathscr{P}$, or equivalently, is the limit of the sequence of interpretations $\text{Inf\_}T^{\vDash}_{\mathscr{P}}\uparrow\alpha$.

We insist on the fact that the inflationary approach assigns a meaning to all logic programs.

The definition is illustrated below by some examples which show that at the opposite of the previously presented approaches, the inflationary semantics of logic programs is not concerned with the "common sense" meaning of negation. In order to make easier the comparison between the inflationary semantics and the well founded semantics, we choose to give the completed interpretation associated with the inflationary model of each program considered.

Example 6.5. (1) Consider the program $\mathscr{P}_1$ of Example 3.14. Recall that this program is semi-positive. The inflationary semantics of $\mathscr{P}_1$ is given by the completed interpretation $\{A, \neg B, C\}$.

(2) Consider the logic program $\mathscr{P}_2 = \mathscr{P}_1 \cup \{B \text{ :- } B\}$ of Example 3.14. Recall that this program is stratifiable. The inflationary semantics of $\mathscr{P}_2$ is given by the completed interpretation $\{A, \neg B, C\}$. Note that the programs $\mathscr{P}_1$ and $\mathscr{P}_2$ are equivalent under the inflationary semantics.

(3) The inflationary semantics of the program $\mathscr{P}_3 = \{A \text{ :- not } A\}$ is given by the completed interpretation $\{A\}$ which is the unique model of $\mathscr{P}_{3_{fo}}$.

(4) The inflationary semantics of the program $\mathscr{P}_4 = \{A \text{ :- not } C,\ B \text{ :- not } A,\ C \text{ :- not } A \,\&\, \text{not } B\}$ is given by the completed interpretation $\{A, B, C\}$ which is a model of $\mathscr{P}_{4_{fo}}$ but neither a minimal model of $\mathscr{P}_{4_{fo}}$ nor a fixpoint of $T^{\vDash}_{\mathscr{P}_4}$.

(5) Let us now consider the program *Even_R* of Example 4.18 which is supposed to define even numbers of a finite set of numbers. The inflationary semantics of this program is given by the completed interpretation

$$\left\{\begin{array}{l}\text{SUC}(0,1),\ \text{SUC}(1,2),\ \text{SUC}(2,3),\\ \neg\text{SUC}(0,0),\ \neg\text{SUC}(0,2),\ldots,\neg\text{SUC}(i,i),\\ \text{Even}(0),\ \neg\text{Even}(1),\ \text{Even}(2),\ldots,\text{Even}(i).\end{array}\right\}$$

The inflationary semantics of the program *Even_R* does not correspond to the intended definition of even numbers.

Note that the logic program *Even_R* can be easily modified in such a way that its inflationary semantics corresponds to the definition of even numbers. In order to do so, we need to introduce a unary predicate *Reached* that will intuitively be

used to delay the production of certain facts. The modified program follows:

$$Even\_Inf = \left\{ \begin{array}{l} \text{even}(0), \\ \text{SUC}(0, 1), \\ \text{SUC}(1, 2), \\ \quad \vdots \\ \text{SUC}(i-1, i), \\ \text{even}(x) \text{ :- suc}(y, x) \text{ \& not even}(y) \text{ \& reached}(y), \\ \text{reached}(x) \text{ :- even}(x), \\ \text{reached}(x) \text{ :- suc}(y, x) \text{ \& reached}(y). \end{array} \right\}$$

Then

$$\text{Inf\_}T^{\models}_{Even\_Inf}(\emptyset) = \{\text{even}(0), \text{suc}(0, 1), \text{SUC}(1, 2), \ldots, \text{SUC}(i-1, i), \text{reached}(0)\} = \boldsymbol{I}_1,$$

$$\text{Inf\_}T^{\models}_{Even\_Inf}(\boldsymbol{I}_1) = \boldsymbol{I}_1 \cup \{\text{reached}(1)\} = \boldsymbol{I}_2,$$

$$\text{Inf\_}T^{\models}_{Even\_Inf}(\boldsymbol{I}_2) = \boldsymbol{I}_2 \cup \{\text{even}(2), \text{reached}(2)\} = \boldsymbol{I}_3,$$

$$\text{Inf\_}T^{\models}_{Even\_Inf}(\boldsymbol{I}_3) = \boldsymbol{I}_2 \cup \{\text{reached}(3)\} = \boldsymbol{I}_4 \ldots\ldots$$

The inflationary model of *Inf_Even* corresponds to the "intended" definition of even numbers.

Theorem 6.6. *Let $\mathscr{P}$ be a logic program. The inflationary model of $\mathscr{P}$ is a Herbrand model of $\mathscr{P}_{\text{f.o.}}$.*

It is simple to note that the inflationary semantics generalizes the standard fixpoint semantics for positive logic programs. Indeed for a positive logic program $\mathscr{P}$, the operator $Inf\_T^{\models}_{\mathscr{P}}$ is inflationary and monotonic and the fixpoint of $Inf\_T^{\models}_{\mathscr{P}}$ is equal to the least fixed point of $T^{\models}_{\mathscr{P}}$.

The Example 6.5 above shows two things. On the one hand, it shows that the inflationary model of a logic program is not, in general, a minimal model of (the f.o. notation of) that logic program. It suffices to look at the logic programs $\mathscr{P}_4$ and *Even_R*. On the other hand, Example 6.5 also shows that the inflationary model of a logic program is not, in general, a supported model of that program. Once again, it suffices to consider the logic programs $\mathscr{P}_4$ and *Even_R*. The behavior of inflationary negation steps back from complementation and is uneasy to motivate from an intuitive point of view.

Finally, Example 6.5 suggests that, although the programmer has to write programs in a less natural manner (for instance the rules which define even numbers), there exists a way to express what one intends to define. Indeed, inflationary semantics has an expressive power strictly higher than the expressive power of stratifiable programs with iterative fixpoint and equal to the expressive power of well-founded semantics. The comparative study of the expressive power of logic programming languages for databases will be developed in detail in Section 9.

7. Model theoretic semantics of stratifiable logic programs

This section is devoted to the model theoretic semantics of stratifiable logic programs. A number of approaches have been proposed which all provide definitions equivalent to the iterative fixpoint semantics presented in Section 4.

The first definition presented is based on the notion of minimal and supported models, and it uses the stratified structure of the program. This first model theoretic definition is due to [3]. This definition can be "simplified" in the sense that if one chooses carefully a stratification of the progam, the notion of supportedness is unnecessary. Thus, the second model theoretic definition proposed here uses the notion of minimal model and the stratified structure of the program. This second alternative model theoretic definition is due to [15, 14] and it leads directly to defining the declarative semantics of stratifiable logic program in terms of circumscription [74] and thus in terms of perfect models [94].

Some new notations are required to make the presentation clear.

Notation. Let $\mathscr{A} = (Fun, Pred)$ and $\mathscr{A}' = (Fun, Pred')$ be two languages such that $Pred \subseteq Pred'$. Note that the language $\mathscr{A}'$ can be viewed as an extension of the language $\mathscr{A}$. Recall that $\mathscr{B}_{\mathscr{A}}$ (respectively, $\mathscr{B}_{\mathscr{A}'}$) denotes the Herbrand base associated with $\mathscr{A}$ (resp. $\mathscr{A}'$). Because the language $\mathscr{A}'$ is an extension of the language $\mathscr{A}$, notice that $\mathscr{B}_{\mathscr{A}} \subseteq \mathscr{B}_{\mathscr{A}'}$. Now let $\boldsymbol{M}'$ be a Herbrand interpretation of $\mathscr{A}'$ (i.e. a subset of $\mathscr{B}_{\mathscr{A}'}$). The Herbrand interpretation $\boldsymbol{M}' \cap \mathscr{B}_{\mathscr{A}}$ of $\mathscr{A}$ is denoted $\boldsymbol{M}'_{|Pred}$.

7.1. The iterative positivist semantics

The model theoretic alternative definition of a stratifiable logic program proposed by [3] is based on the notion of positivist models, that is, models statisfying both minimality and supportedness properties. Recall that in Section 4, it has been shown that the iterative fixpoint $\boldsymbol{M}_{\mathscr{P}}$ of a stratifiable logic program $\mathscr{P}$ is a minimal model of $\mathscr{P}_{\text{f.o.}}$ and a supported model for $\mathscr{P}$, thus $\boldsymbol{M}_{\mathscr{P}}$ is a positivist model for $\mathscr{P}$. However, the converse does not hold, that is, given a positivist model $\boldsymbol{M}$ of $\mathscr{P}$, this model $\boldsymbol{M}$ may not be equal to the iterative fixpoint $\boldsymbol{M}_{\mathscr{P}}$ of $\mathscr{P}$. In order to define the declarative semantics of stratifiable logic programs in terms of positivist models, one needs to make use of the stratified structure of the programs as follows.

Definition 7.1 (*The iterative positivist model of a stratified logic program*). Let $\mathscr{P}^* = (\mathscr{P}_1, Pred_1)(\mathscr{P}_2, Pred_2)\ldots(\mathscr{P}_n, Pred_n)$ be a stratified logic program. The iterative positivist model $\boldsymbol{M}$ of $\mathscr{P}^*$ is the Herbrand interpretation of $(Fun, \bigcup_{j=1}^{n} Pred_j)$ such that: for $i = 1 \ldots n$, $\boldsymbol{M}_{|\bigcup_{j=1}^{i} Pred_j}$ is a positivist model of $(\bigcup_{j=1}^{i} \mathscr{P}_i, \bigcup_{j=1}^{i} Pred_i)$.

The notion of iterative positivist model provides an alternative definition of the declarative semantics of stratifiable logic programs.

Theorem 7.2. *Let $\mathscr{P}$ be a stratifiable logic program.*

(1) [3] *Let $\mathscr{P}^*$ be a stratification for $\mathscr{P}$. The iterative fixpoint of $\mathscr{P}^*$ is equal to the iterative positivist model of $\mathscr{P}^*$, and thus*

(2) *the iterative fixpoint of $\mathscr{P}$ is equal to the iterative positivist model of $\mathscr{P}$ (where the iterative positivist model of $\mathscr{P}$ is the iterative positivist model of one of the stratifications of $\mathscr{P}$).*

The proof of this result can be found in [3]. Note that because two distinct stratifications $\mathscr{P}^*$ and $\mathscr{P}^{*\prime}$ of a stratified logic program $\mathscr{P}$ have the same iterative fixpoint, part (1) of Theorem 7.2 implies that they also share the same iterative positivist model. This justifies defining the iterative positivist model of $\mathscr{P}$ as the iterative positivist model of one of the stratifications of $\mathscr{P}$.

As a matter of fact, in [3], Definition 7.1 is given in slightly different terms. A sequence $(\boldsymbol{M}_i)_{1 \leqslant i \leqslant n}$ of Herbrand interpretations are defined by:

$$\boldsymbol{M}_1 = \bigcap \{\boldsymbol{M} \mid \boldsymbol{M} \text{ is a supported model of } \mathscr{P}_1\} \text{ and for } 1 \leqslant i < n,$$

$$\boldsymbol{M}_{i+1} = \bigcap \{\boldsymbol{M} \mid \boldsymbol{M} \text{ is a supported model of } \bigcup_{j=1}^{i+1} \mathscr{P}_j \text{ and } \boldsymbol{M}_{|\bigcup_{j=1}^{i} Pred_j} = \boldsymbol{M}_i\},$$

and the iterative positivist model $\boldsymbol{M}_{\mathscr{P}}$ of $\mathscr{P}$ is defined as $\boldsymbol{M}_n$.

Let us briefly show that Definition 7.1 and the definition in [3] sketched above are equivalent. First notice that:

Although it has been emphasized in Section 3 that, in general, the family of positivist models of a logic program $\mathscr{P}$ cannot be identified with the family of minimal models among supported models for $\mathscr{P}$, for semi-positive logic programs we have:

Lemma 7.3. *If $\mathscr{P}$ is a semi-positive logic program, then $\mathscr{P}$ admits a unique positivist model $\boldsymbol{M}_{\mathscr{P}}$ and $\boldsymbol{M}_{\mathscr{P}} = \bigcap \{\boldsymbol{M} \mid \boldsymbol{M}$ is a supported model of $\mathscr{P}\}$.*

Lemma 7.3 gives a characterization of a class of logic programs, namely semi-positive logic programs, whose positivist model is the least supported model. In fact, the proof of Lemma 7.3 follows directly from the results of [3].

7.2. The iterative minimal model

We continue this presentation by showing that with a careful choice of a stratification, the notion of supportedness is not necessary in order to define the semantics of stratifiable logic programs.

Recall that the intended meaning of a stratifiable logic program $\mathscr{P}$ is defined as the iterative fixpoint (or the iterative positivist model) of some stratification $\mathscr{P}^*$ of $\mathscr{P}$. The choice of a stratification $\mathscr{P}^*$ for $\mathscr{P}$ is not relevant because all stratifications of $\mathscr{P}$ have the same iterative fixpoint (or iterative positivist model).

In the following, we introduce some property of stratifications and utilize this property in order to assign meaning to stratifiable logic programs.

Remark 7.4. If $\mathscr{P}$ is stratifiable then there exists a stratification $\mathscr{P}^* = (\mathscr{P}_1, Pred_1)(\mathscr{P}_2, Pred_2)\ldots(\mathscr{P}_n, Pred_n)$ for $\mathscr{P}$ such that:

(1) $\text{Def}(P, \mathscr{P}) = \emptyset$ entails that $\mathscr{P} \in Pred_1$, and

(2) $\mathscr{P}_1$ is a positive logic program.

Let us briefly show that our remark is correct. Given a random stratification of $\mathscr{P}$ which may not satisfy the two conditions above, it is easy to construct a stratification of $\mathscr{P}$ which does satisfy both conditions. The way one can enforce condition (1) is obvious. Concerning condition (2) and assuming that $\mathscr{P}^* = (\mathscr{P}_1, Pred_1)(\mathscr{P}_2, Pred_2)\ldots(\mathscr{P}_n, Pred_n)$ is a stratification of $\mathscr{P}$ that does not satisfy (2), it suffices to split $\mathscr{P}_1$ in two programs. The first program $\mathscr{P}_1'$ is the subset of positive rules of $\mathscr{P}_1$, the second $\mathscr{P}_1''$ is the remaining rules in $\mathscr{P}_1$. The stratified logic program obtained by concatenation of $\mathscr{P}_1'$ and $\mathscr{P}_1''$ to $(\mathscr{P}_2, Pred_2)\ldots(\mathscr{P}_n, Pred_n)$ is a stratification of $\mathscr{P}$ that does satisfy (2).

The first stratum $(\mathscr{P}_1, Pred_1)$ of a stratified program satisfying conditions (1) and (2) may be such that the program $\mathscr{P}_1$ is empty (but this is not a new situation).

In the following, a stratification for $\mathscr{P}$ (resp. a stratified program $\mathscr{P}^*$) satisfying conditions (1) and (2) is called a *strict stratification* for $\mathscr{P}$ (resp. a *strictly stratified program*).

Definition 7.5 (*Iterative minimal model*, Bidoit and Hull [19]). Let $\mathscr{P}^* = (\mathscr{P}_1, Pred_1)(\mathscr{P}_2, Pred_2)\ldots(\mathscr{P}_n, Pred_n)$ be a strictly stratified logic program. The iterative minimal model M of $\mathscr{P}$ is the Herbrand interpretation of $(Fun, \bigcup_{j=1}^{n} Pred_j)$ such that for $i = 1\ldots n$, $M_{|\bigcup_j^i Pred_j}$ is a minimal model of $(\bigcup_{j=1}^{i} \mathscr{P}_j, \bigcup_{j=1}^{i} Pred_j)$.

The notion of iterative minimal model of strictly stratified programs provides an alternative definition of the declarative semantics of stratifiable logic programs.

Theorem 7.6 (Bidoit and Froidevaux [14]). *Let $\mathscr{P}$ be a stratifiable logic program and let $\mathscr{P}^*$ be a strict stratification for $\mathscr{P}$. Then, the iterative fixpoint of $\mathscr{P}$ is equal to the iterative minimal model of $\mathscr{P}^*$.*

Proofs of the above result can be found in [14].

7.3. Circumscription

The notion of iterative minimal models of strictly stratified logic programs happens to be a special case of the notion of model of Prioritized Circumscription. The relationship between Prioritized Circumscription and iterative minimal model has been shown in [94] through the relationship between Prioritized Circumscription and perfect models of stratifiable logic programs. Prioritized Circumscription has been introduced by [83] and further investigated by [73]. The model theoretic characterization of Prioritized Circumscription is presented below.

Definition 7.7 (*Model of prioritized circumscription*). Let $\mathscr{A} = (Fun, Pred)$ be a first order language and $\mathscr{F}$ be a set of first order formulas over $\mathscr{A}$. Let $Pred_1 \ldots Pred_n$ be a partition of the set of predicate symbols *Pred*. A model M of $\mathscr{F}$ is a model of Prioritized Circumscription of $\mathscr{F}$ with respect to priorities $Pred_1 > \cdots > Pred_n$ iff for $i = 1 \ldots n$,

$$M_{|\bigcup_{j=1}^{i} Pred_j} \text{ is minimal}^{13} \text{ in the set } \left\{ N_{|\bigcup_{j=1}^{i} Pred_j} \;\middle|\; \begin{array}{l} N \text{ is a model of } \mathscr{F}, \text{ and} \\ M_{|\bigcup_{j=1}^{i-1} Pred_j} = N_{|\bigcup_{j=1}^{i-1} Pred_j}. \end{array} \right\}$$

Intuitively, a model of Prioritized Circumscription of $\mathscr{F}$ with respect to $Pred_1 > \cdots > Pred_n$ is a model that "first" minimizes the truth of facts related to the predicates in $Pred_1$, and then, with the interpretation of the facts related to predicates in $Pred_1$ being fixed, minimizes the truth of facts related to the predicates in $Pred_2 \ldots$.

Definition 7.7. together with the definition of iterative minimal models of strictly stratified logic programs gives a strong intuition of the way that the declarative semantics of stratifiable logic programs can be defined in terms of Prioritized Circumscription.

Theorem 7.8 (Przymusinski [94]). *Let $\mathscr{P}$ be a stratifiable logic program and let $\mathscr{P}^* = (\mathscr{P}_1, Pred_1) \ldots (\mathscr{P}_n, Pred_n)$ be a strict stratification for $\mathscr{P}$. Then, the iterative fixpoint of $\mathscr{P}$ is equal to the model of Prioritized Circumscription of $\mathscr{P}$ with respect to $Pred_1 > \cdots > Pred_n$.*

Note that a strict stratification for $\mathscr{P}$ is needed in Theorem 7.8. The proof of this theorem can be derived from the proof that the perfect model of a stratifiable logic program is equal to the model of Prioritized Circumscription with respect to $Pred_1 > \cdots > Pred_n$ [94]. In fact, in the context of logic programs, the notion of a perfect model is introduced as a conceptually simpler and more natural definition of model of Prioritized Circumscription.

Before presenting the perfect model semantics of logic programs, it should be mentioned that the semantics of stratifiable logic programs has also been defined using another form of circumsciption, namely Pointwise Circumscription [74].

7.4. Perfect model

From a technical point of view, perfect models are defined by deriving from the syntax of the logic program, a relation, called the priority relation, on the elements of the Herbrand base. This priority relation is, in turn, used to derive another relation, called the preferability relation, on the f.o. models of the program. Under certain condition, the preferability relation is a partial order and a perfect model is a minimal model with respect to preferability.

The definition of the priority relation is motivated by the two following principles.

(1) The consequent of a rule r should have strictly lower priority (for minimization) than a negative premise of the rule r.

[13] With respect to set inclusion.

(2) The consequent of a rule r should have priority not higher than a positive premise of the rule r.

Let us try to rephrase the first principle. Consider a propositional rule r with head A and with $\neg B$ in its body. Then one should first minimize the truth of B (which leads us to prefer the models in which B is false and to select these models if some exist) and then minimize the truth of A (which leads us to prefer among the preceding collected models the ones in which A is false). Intuitively, principle (1) is made in the same spirit as the one used to motivate stratification, that is, "predicates must be completely defined before they can be used negatively". The nature of principle (1) is slightly different because it does not express any syntactical constraint on the program. Anyway, it will be shown that principle (1) leads to requiring the (local) stratification of the programs.

Definition 7.9 (*Dependency and priority relation*). Let $\mathscr{P}$ be a logic program over the language $\mathscr{A}$. The dependency relation on $\mathscr{B}_{\mathscr{A}}$ associated with $\mathscr{P}$, denoted by $\leqslant_{\mathscr{P}}$, is the transitive closure of the binary relation

$$\{(A, B) \mid r \in \mathit{Inst\_P},\ \mathrm{head}(r) = A \text{ and } B \in \mathrm{pos}(\mathrm{prem}(r)) \cup \neg.\mathrm{neg}(\mathrm{prem}(r))\}.$$

The priority relation on $\mathscr{B}_{\mathscr{A}}$ associated with $\mathscr{P}$, denoted by $<_{\mathscr{P}}$, is the binary relation $(<_{\mathscr{P}} \circ \leqslant_{\mathscr{P}}) \cup (\leqslant_{\mathscr{P}} \circ <_{\mathscr{P}})$ where

$$<_{\mathscr{P}} = \{(A, B) \mid r \in \mathit{Inst\_P},\ \mathrm{head}(r) = A \text{ and } B \in \mathrm{neg}(\mathrm{prem}(r))\}$$

and

$$\mathrm{Rel} \circ \mathrm{Rel}' = \{(A, B) \mid (A, C) \in \mathrm{Rel} \text{ and } (C, B) \in \mathrm{Rel}'\},$$

with Rel and Rel′ binary relations.[14]

Intuitively, while considering the precedence graph associated with $\mathscr{P}$, we have: $A \leqslant_{\mathscr{P}} B$ if there exists a path from B to A and $A <_{\mathscr{P}} B$ if there exists a path from B to A going through a negative edge.

The priority relation among ground atomic formulas is used to induce a relation on the models of the program as follows.

Definition 7.10 (*Preferability relation and perfect model of a logic program*). Let $\mathscr{P}$ be a logic program. The preferability relation on the set of f.o. models of $\mathscr{P}_{\text{f.o.}}$ associated with $\mathscr{P}$, denoted $\ll_{\mathscr{P}}$, is defined by

$$M \ll_{\mathscr{P}} N \text{ iff } M \neq N \quad \text{and} \quad \forall A \in M - N, \exists B \in N - M\ A <_{\mathscr{P}} B.$$

A model M of $\mathscr{P}_{\text{f.o.}}$ is a perfect model of $\mathscr{P}$ iff there exists no model N of $\mathscr{P}_{\text{f.o.}}$ such that $N \ll_{\mathscr{P}} M$.

[14] The operation $\circ$ defined here is similar to the relational database join operation.

One should notice that preferability may only hold between distinct models of the program and thus is antireflexive by definition.

As it is shown in [94], not all logic programs can be assigned a meaning under the perfect model semantics. In other words, a logic program may not have a perfect model. An example is given below of such a logic program.

Example 7.11. Let us consider the propositional logic program $\mathscr{P} = \{A$:- not B, B:- not $A\}$. Then, we have: $A \leqslant_{\mathscr{P}} B$, $B \leqslant_{\mathscr{P}} A$ and $A <_{\mathscr{P}} B$, $B <_{\mathscr{P}} A$. $\mathscr{P}$ has three f.o. models, namely $M_1 = \{A\}$, $M_2 = \{B\}$ and $M_3 = \{A, B\}$. Thus, $M_1 \lessdot_{\mathscr{P}} M_3$, $M_2 \lessdot_{\mathscr{P}} M_3$, $M_1 \lessdot_{\mathscr{P}} M_2$ and $M_2 \lessdot_{\mathscr{P}} M_1$. Each model of $\mathscr{P}$ has a preferred model, thus $\mathscr{P}$ has no perfect model.

However, it should be noted that:

Lemma 7.12. *A logic program $\mathscr{P}$ has at most one perfect model.*

A class of logic programs having a perfect model has been exhibited in [94]. Unsurprisingly, we have:

Theorem 7.13 (Przymusinski [94]). *If $\mathscr{P}$ is a locally stratifiable logic program then $\mathscr{P}$ has a unique perfect model.*

The proof of this theorem utilizes the fact that, if $<_{\mathscr{P}}$ is noetherian (i.e. if there exits no infinite increasing sequence) then the priority relation $<_{\mathscr{P}}$ and the preferability relation $\lessdot_{\mathscr{P}}$ are (strict) partial orders. As a matter of fact, local stratification entails that $<_{\mathscr{P}}$ is noetherian.

It should be emphasized here that local stratification is a sufficient condition for the existence of a perfect model. However, a logic program may not be stratifiable and still have a perfect model. This is illustrated by the following example.

Example 7.14. Let us consider the propositional program $\mathscr{P} = \{A$:- not $A\}$. We simply have that $A \leqslant_{\mathscr{P}} A$ and $A <_{\mathscr{P}} A$. Anyway, because the preferability relation is antireflexive, the unique f.o. model $M = \{A\}$ of $\mathscr{P}$ is a perfect model. Note that $\mathscr{P}$ is not stratifiable.

Now let us consider the class *Unstrat_Perfect* of logic programs that are unstratifiable but still have a perfect model. Intuitively, this class is "small" which can be explained by the fact that the definition of a perfect model closely follows the definition of stratification. Again from a very intuitive point of view, programs in *Unstrat_Perfect* contain recursive negation in a very restrictive form, that is, of the form $A <_{\mathscr{P}} A$.

As a matter of fact, the class *Unstrat_Perfect* does not include dynamically stratifiable logic programs (which are unstratifiable at the same time). This remark

can be illustrated by considering the logic program *Even_R* of Example 4.18. Recall that *Even_R* is not locally stratifiable. It can be easily checked that none of the models of *Even_R* is perfect.

The need for an extension of the notion of a perfect model has been strongly felt and a first attempt to provide such an extension has been proposed in [92]. More recently, a major and interesting extension has been investigated in [96] which leads us to redefine the well-founded semantics.

To conclude this section, let us now examine the perfect model semantics with respect to minimality and supportedness. In [94], it is shown that:

Theorem 7.15 (Przymusinski [94]). *A perfect model of a logic program $\mathscr{P}$ is a minimal model (with respect to set inclusion) of $\mathscr{P}_{f.o.}$.*

As a consequence, in order to show that a model is perfect, it suffices to show that there exists no minimal model preferable to it.

While perfect models enjoy the minimal model property, a perfect model of a logic program $\mathscr{P}$ may not be a supported model of $\mathscr{P}$. Consider the program $\mathscr{P}$ of Example 7.14 above. Notice that the perfect model M of this program is not a supported model of $\mathscr{P}$. Notice also that this program is not stratifiable.

Of course, perfect models of locally stratifiable programs enjoy supportedness. This directly follows from the equivalence between perfect model semantics and iterative fixpoint semantics for the class of stratifiable logic programs established by [94] and from the equivalence between perfect model semantics and default model semantics for the larger class of locally stratifiable logic programs established by [14].

Theorem 7.16 (Przymusinski [94]). *Let $\mathscr{P}$ be a stratifiable logic program. Then, the iterative fixpoint of $\mathscr{P}$ is equal to the perfect model of $\mathscr{P}$.*

It should be mentioned that the notion of a perfect model has been defined and investigated for disjunctive logic programs (with negation). A disjunctive rule is a rule whose head is a disjunction of positive literal.

In order to define perfect models of disjunctive logic programs, a third principle is added to the two principles stated at the beginning of this subsection:

(3) Predicates occurring in the head of a given rule should have the *same* priority.

Thus in Definition 7.9 the dependency relation associated with $\mathscr{P}$ needs to be completed by the couples (A, B) where A and B are in the head of some rule in $\mathscr{P}$. The definition of perfect model is unchanged.

The notion of stratified programs is also slightly generalized in order to deal with disjunctive programs. Intuitively, it is required that predicates (or positive facts) occurring in the head of a given rule should belong to the same stratum. Of course stratification ensures the existence of at least one perfect model for disjunctive logic program. A disjunctive logic program may have more than one perfect model.

We limit ourself to presenting an example.

Example 7.17 (*Przymusinski* [94]). Let us consider the following disjunctive logic program:

$$\text{Happy} = \left\{ \begin{array}{l} \text{Has\_Vacation(Jones)}, \\ \text{Goto\_Australia}(x) \vee \text{Goto\_Europe}(x) \text{ :- Has\_Vacation}(x), \\ \text{Unhappy}(x) \text{ :- not Goto\_Australia}(x), \\ \text{Unhappy}(x) \text{ :- not Goto\_Europe}(x). \end{array} \right\}$$

Perfect models of a disjunctive logic program are among the minimal models of that program. Here the f.o. notation of Happy has three minimal models, namely:

$M_1 = \{\text{Has\_Vacation(Jones)}, \text{Goto\_Australia(Jones)}, \text{Unhappy(Jones)}\}$,

$M_2 = \{\text{Has\_Vacation(Jones)}, \text{Goto\_Europe(Jones)}, \text{Unhappy(Jones)}\}$,

$M_3 = \{\text{Has\_Vacation(Jones)}, \text{Goto\_Australia(Jones)}, \text{Goto\_Europe(Jones).}\}$

Because $\text{Unhappy} <_{\text{Happy}} \text{Goto\_Australia}$ and $\text{Unhappy} <_{\text{Happy}} \text{Goto\_Europe}$, we have:

$$M_1 \lessdot_{\text{Happy}} M_3 \quad \text{and} \quad M_2 \lessdot_{\text{Happy}} M_3.$$

Moreover, the two models M_1 and M_2 are not comparable. It follows that these two models, M_1 and M_2, are the perfect models of Happy. Finally, it is quite easy to check that the disjunctive program Happy is stratifiable.

8. Default and stable semantics

In this section, we shall discuss two equivalent approaches for defining the declarative semantics of logic programs which are both based on non-monotonic logic. The first approach, called default semantics was proposed in [15, 14] and is based on Reiter's Default logic [98]. The second approach, called stable semantics was proposed in [51, 53] and is based on Moore's Autoepistemic logic [87], which in its turn is based on ideas from [39].

First of all, the reader should be informed that there is nothing very deep in the fact that the stable model semantics and the default model semantics for logic programs are equivalent. This is a straightforward consequence of the equivalence between Autoepistemic logic and Default logic in general. The equivalence has been recently exhibited in [64].

The two approaches follow a quite classical process. A systematic translation of logic programs into default/autoepistemic theories is provided. (This resembles Clark's completion approach where a first order theory is associated to a logic program.) Then the semantics of logic programs is defined in terms of the default/autoepistemic translation of the programs, that is in terms of "extension"/"stable expansion" of the translations. Since our purpose is not a review of non-monotonic logics, we do not detail the foundations of default logic

and autoepistemic logic. Neither do we give details on the translations of the programs into the respective formalism. Such presentations have been provided in [15, 14] concerning the default logic approach and in [51, 53] concerning the autoepistemic approach.

We limit ourself to a brief intuitive presentation of the motivations which lead to investigate default logic for defining the declarative semantics of logic programs. The motivations leading to the use of autoepistemic logic are very similar.

Default logic has been introduced by Reiter [98] in order to formalize default reasoning. Default reasoning is a fundamental component of common sense reasoning and it is a form of non-monotonic reasoning. Default logic allows one to reason about real world situations whose descriptions may be incomplete and provides a way to infer more than what these descriptions allows. These additional inferences are determined by special rules called default rules. Default rules have premises (properties that need to be "proved" in order to activate the rules), justifications (properties that need only to be consistent with all the properties that can be inferred) and of course a consequence.

Indeed, the Closed World Assumption [97] is an instance of default reasoning. Making the Closed World Assumption corresponds to the choice of describing real world situations in an incomplete manner: only positive (true) information is specified. From a database point of view, this choice can be motivated by "common sense". For instance in order to define a property, it seems more natural, to give the list of entities satisfying the property than to give both the list of entities satisfying the property and the list of entities that fails to satisfy the property. Thus the Closed World Assumption simplifies tremendously the representation of data.

Now, while data description is "incomplete", the Closed World Assumption entails a second fundamental principle: complete knowledge of the world situation described is assumed. Intuitively, this means that, although the definition of a property is limited to the list of entities satisfying it, one should be able to say for any entity whether the property holds. It is probably unnecessary to recall that first order logic does not allow one to infer negative facts from a set of positive facts for example. This is where default logic (and default rules) intervene: in order to be able to derive negative information from the incomplete description, one resorts to default rules, called CWA-default rules, of the form: if it is consistent to assume that an entity does not satisfy a property, then infer that this entity does not satisfy the property.

When properties are specified by rules rather than extensively, CWA-default rules are obviously still needed and rules having negative premises have the status of default rules, or in other words, negative premises are treated like justifications.

The use of default logic for defining the declarative semantics of logic programs has been first investigated in [18, 19] where positive disjunctive programs are considered. It is shown there that this approach is equivalent to the Generalized Closed World Assumption [84]. The use of default logic has been extended to logic programs with negation in [15].

For the sake of simplicity, we prefer to give, in the current paper, a definition of stable/default models that does not require the introduction of autoepistemic logic or default logic. A unique formalism is adopted for this presentation. We shall just ask the reader to keep in mind that the notion of a stable model and the notion of a default model originate from ideas developed for non-monotonic and common sense reasoning [39, 98] and from the observation that the Closed World Assumption is a form of default reasoning.

We proceed now to the formal presentation of the stable/default semantics of logic programs. First we need to draw the reader's attention to the fact that the stable/default semantics of a logic program is given by some model of (the first order notation of) the program and that models are required to be represented by completed Herbrand interpretation.

Definition 8.1 (*Stable/default model of logic programs*). Let $\mathscr{P}$ be a logic program. A stable/default model of $\mathscr{P}$ is a completed model $\mathscr{M}$ such that $\text{pos}(\mathscr{M}) = T^{\in}_{\mathscr{P}}(\mathscr{M})$.

From the point of view of default logic, if we want to motivate the definition of a stable/default model, we would say that a default model $\mathscr{M}$ of a logic program is a model whose negative part $\text{neg}(\mathscr{M})$ leads to compute (by means of $T^{\in}_{\mathscr{P}}$) a set of positive facts $\text{pos}(\mathscr{M})$ consistent with $\mathscr{M}$. Indeed, "$T^{\in}_{\mathscr{P}}(\mathscr{M})$ is a superset of $\text{pos}(\mathscr{M})$" means that positive facts have been produced by $T^{\in}_{\mathscr{P}}$ which are in contradiction with $\text{neg}(\mathscr{M})$. Now, "$T^{\in}_{\mathscr{P}}(\mathscr{M})$ is a subset of $\text{pos}(\mathscr{M})$" means that the CWA is not taken into account because the complement of $T^{\in}_{\mathscr{P}}(\mathscr{M})$ is a superset of (and is inconsistent with) $\text{neg}(\mathscr{M})$.

The definition above can be rewritten as follows.

Lemma 8.2. *The completed interpretation $\mathscr{M}$ is a stable/default model of the logic program $\mathscr{P}$ iff there exists $i \geqslant 0$ such that $\mathscr{M} = \mathscr{M}^i$ where the sequence $(\mathscr{M}^i)_{(0 \geqslant i)}$ is defined by*

$$\mathscr{M}^0 = \text{neg}(\mathscr{M}), \qquad \mathscr{M}^{i+1} = \mathscr{M}^i \cup T^{\in}_{\mathscr{P}}(\mathscr{M}^i).$$

Note that the sequence $(\mathscr{M}^i)_{(0 \geqslant i)}$ defined above always converges even when $\mathscr{M}$ is not a stable/default model of $\mathscr{P}$. It is an increasing sequence of sets of literals thus, its limit is a subset, not necessarily consistent, of $\mathscr{B} \cup \neg.\mathscr{B}$. (For instance if we consider the logic program $\{A$:- not $B\}$ and the completed interpretation $\mathscr{M} = \{\neg A, \neg B\}$, the limit of the sequence associated with $\mathscr{P}$ and $\mathscr{M}$ is the inconsistent set of facts $\{\neg A, A, \neg B\}$. Thus a stable/default model is characterized by the fact that the limit of the sequence is a completed interpretation.

In [53], the definition of a stable model is given in a slightly different and less concise form. The proof of the equivalence of Gelfond's definition and the above definition can be found in [17].

Example 8.3. Consider the logic program *Even_R* presented in Example 4.18 and the completed interpretation $\mathcal{M}$ corresponding to the intended meaning of that program (this intended completed interpretation has already been characterized as the well-founded model of *Even_R*). It is rather immediate to check that $\mathcal{M}^2 = \mathcal{M}$. Thus $\mathcal{M}$ is a stable/default model of *Even_R*. In this particular case, $\mathcal{M}$ is the unique stable/default model of $\mathcal{P}$.

An interesting characterization of the stable/default model has been recently proposed in [112]. This characterization is based on:

(1) the use of the dual representation of Herbrand interpretation as a set of positive facts, that is, a Herbrand interpretation is represented by the set of negative facts corresponding to the ground atomic formulas false in this interpretation.

(2) the use of the operator $\tilde{S}_{\mathcal{P}}$ presented in Section 5.

Theorem 8.4. *A completed model $\mathcal{M}$ of $\mathcal{P}_{\text{f.o.}}$ is a stable/default model of the logic program $\mathcal{P}$ iff* $\text{neg}(\mathcal{M})$ *is a fixpoint of $\tilde{S}_{\mathcal{P}}$.*

The proof is immediate from the definitions of a stable model and from the definition of $\tilde{S}_{\mathcal{P}}$.

As discussed in [14, 16, 17, 53] and as suggested by the above remark, not all logic programs can be assigned a meaning under the stable/default model semantics. In fact, there are two kinds of programs to which the stable/default model semantics is not applicable. The first kind are the programs that have no stable/default model. These programs are called inconsistent. The second kind are the logic programs that do have more than one stable/default model. These programs are called ambiguous. Very simple programs are presented below to illustrate the phenomenon.

Example 8.5. (1) Let us first consider the logic program $\mathcal{P} = \{A \text{ :- not } A\}$. Recall that this program has a unique f.o. model given by the completed interpretation $\{A\}$. $A \notin T^{\in}_{\mathcal{P}}(\{A\})$, thus $\mathcal{P}$ has no stable/default model.

(2) Consider now the program $\mathcal{P} = \{A \text{ :- not } B,\ B \text{ :- not } A\}$. This program has three f.o. models, namely $\mathcal{M}_1 = \{A, \neg B\}$, $\mathcal{M}_2 = \{\neg A, B\}$ and $\mathcal{M}_3 = \{A, B\}$. Note that $\text{pos}(\mathcal{M}_1) = \{A\} = T^{\in}_{\mathcal{P}}(\mathcal{M}_1)$ and $\text{pos}(\mathcal{M}_2) = \{B\} = T^{\in}_{\mathcal{P}}(\mathcal{M}_2)$, while $\text{pos}(\mathcal{M}_3) = \{A, B\} \neq T^{\in}_{\mathcal{P}}(\mathcal{M}_3) = \emptyset$. Thus $\mathcal{P}$ has two stable/default models.

The interest of the stable/default model semantics lies in its applicability to a class of logic programs which is wider than the class of (locally) stratifiable logic programs. Note that the unstratifiable logic program *Even_R* has a unique stable/default model. The class of locally stratifiable logic programs is included in the class of logic programs having a unique stable/default model. The stable/default model semantics generalizes the standard semantics of positive logic programs and the iterative fixpoint semantics of stratifiable logic programs.

Theorem 8.6 (Bidoit and Froidevaux [14], Gelfond and Lifschitz [53]).

(1) *If $\mathscr{P}$ is a locally stratifiable logic program then $\mathscr{P}$ has a unique stable/default model.*

(2) *If $\mathscr{P}$ is a stratifiable logic program then the (unique) stable/default model of $\mathscr{P}$ is equal to the completion of the iterative fixpoint of $\mathscr{P}$.*

(3) *If $\mathscr{P}$ is a locally stratifiable logic program then the (unique) stable/default model of $\mathscr{P}$ is equal to the completion of the perfect model of $\mathscr{P}$.*

The logic program *Even_R* given in Example 4.15 and which defines even numbers, illustrates the applicability of the stable/default semantics to unstratifiable logic programs. This example also gives an indication of the relation between the stable/default model semantics and the well-founded semantics. The unique stable/default model of the program *Even_R* is equal to its well-founded model.

A particular effort has been put into characterizing the class of logic programs that do have a unique stable/default model. Indeed, only subclasses have been exhibited and the following result shows that the problem is a difficult one.

Theorem 8.7 (Bidoit and Froidevaux [17]). *Determining whether a propositional logic program has a stable/default model is a* NP*-complete problem.*

The largest class of programs so far shown to have a unique stable/default model is the class of effectively (or dynamically) stratifiable logic programs.

Theorem 8.8 (Bidoit and Froidevaux [17]). *If $\mathscr{P}$ is effectively stratifiable then $\mathscr{P}$ has a unique stable/default model.*

Recall that checking whether a program is effectively (or dynamically) stratifiable can be done in polynomial time (in the size of the total number of premises in the program) and in the worst case, requires to compute the entire well-founded model of the program.

Although effective stratification ensures the existence and uniqueness of a stable/default model, clearly it is not a necessary condition. The program given in the following example provides a logic program which is not effectively stratifiable and which has a unique stable/default model.

Example 8.9. Consider the logic program $\mathscr{P}_4 = \{A$:- not C, B :- not A, C :- not A & not $B\}$ of Example 3.14. It is immediate to check that $\mathrm{EFF}_{\mathscr{P}_4} = \mathscr{P}_4$ thus because $\mathscr{P}_4$ is not stratifiable, it is not effectively stratifiable. However, $\mathscr{P}_4$ has a unique stable/default model, namely the completed interpretation $\{A, \neg B, \neg C\}$ which as a matter of fact corresponds to the unique fixpoint of $T^{\models}_{\mathscr{P}_4}$. Note that the well-founded model of $\mathscr{P}_4$ is the empty partial interpretation (which can be interpreted as "the well-founded semantics is not able to assign a meaning to the program $\mathscr{P}_4$").

Recall that effective stratification characterizes exactly the class of logic programs whose well-founded model is a completed interpretation (a 2-valued model). In fact there exists a strong relationship between the stable/default model semantics and the well-founded semantics.

Theorem 8.10 (Bidoit and Froidevaux [17], VanGelder et al. [115]).

(1) *If the well-founded model $\mathcal{M}$ of $\mathcal{P}$ is total* (*i.e. is a completed interpretation*) *then $\mathcal{M}$ is the* (*unique*) *stable/default model of $\mathcal{P}$.*

(2) *If $\mathcal{P}$ has a unique stable/default model $\mathcal{M}$ then the well-founded model of $\mathcal{P}$ is included in $\mathcal{M}$.*

Note that the converse of statement (2) does not hold simply because a logic program always has a well-founded model (possibly empty) whereas it may not have a stable/default model.

Finally, it is also worth noticing that:

Theorem 8.11 (Bidoit and Froidevaux [14], Gelfond and Lifschitz [53]). *Let $\mathcal{P}$ be a logic program and let $\mathcal{M}$ be a stable/default model of $\mathcal{P}$. Then*:

(1) *$\mathcal{M}$ is a minimal model of $\mathcal{P}_{\text{f.o.}}$, and*

(2) *$\mathcal{M}$ is a supported model of $\mathcal{P}$.*

Recall here that positivist (minimal and supported) models of a logic program $\mathcal{P}$ are fixpoints of $T^{\models}_{\mathcal{P}}$. Thus the theorem above just says that stable/default models of $\mathcal{P}$ are particular fixpoints of the immediate consequence operator $T^{\models}_{\mathcal{P}}$ associated with $\mathcal{P}$.

To conclude this section, note that the notion of a default model has been defined for disjunctive logic programs. The extended notion of stratification has been used to characterize a class of disjunctive logic programs having (at least) a default model. Unsurprisingly, it has been shown in [14] that the default model semantics and the perfect model semantics coincide for the class of stratifiable disjunctive logic programs.

9. On the expressive power of rule-based query languages

The focus of this section is on the expressive power of the various logic programming semantics defined throughout the paper. In this section, logic programs are viewed as specifying queries and each semantics (iterative fixpoint, well-founded semantics, inflationary semantics) defines a distinct query language.

The problem of defining a "natural" semantics of negation cannot be separated from the problem of defining sufficiently expressive query languages. Recall indeed that the introduction of negation in logic programs has been motivated by the inability of positive logic programs to define, for instance, the complement of the transitive closure. The problem of providing a suitable semantics of negation has

retained most of the researcher's attention, concentrating on the "natural" aspects of the proposed semantics and neglecting the expressive power aspect. It is only recently that the expressive power of these languages has been studied in more detail [2, 8, 7, 66, 63, 112].

9.1. Minimal background

Querying occupies a central place in the history of database technology and constitutes by itself a domain. The literature [37, 11, 28, 29, 32, 61, 110] offers a large place to the theory of database queries.

For the purpose of the presentation, we briefly present classes of computable database queries. The following well-known database query languages are briefly described: First Order queries (FO) [37], Fixpoint queries (FP) [29] and Inflationary Fixpoint queries (IFP) [57].

Intuitively, a query defines a mapping from the set of instances of a (input) database schema into the set of instances of a (output) database schema. The restriction that the output database schema be a single relation schema is frequently stated but not significant for the discussion. A database schema is a pair $(D, \mathscr{R})$ where D is a finite set of values and $\mathscr{R}$ is a finite set of predicate symbols. An instance of a database schema $(D, \mathscr{R})$ is simply a finite relational structure $(D, \boldsymbol{I})$ where $\boldsymbol{I}$ is an "interpretation" of the predicate symbols in $\mathscr{R}$ (the reader can see $\boldsymbol{I}$ as a Herbrand interpretation where the constant symbols are the elements of the domain D and where no other function symbols are allowed).

The two major questions to examine are:

- What does it mean to be a "reasonable" query?
- What queries are expressible in a given query language? (and, are all reasonable queries expressible in the language?)

Clearly, a query cannot be an arbitrary mapping. A *computable query* [28] should satisfy the following basic features:

(1) it is a partial recursive function ρ, and

(2) ρ should map isomorphic input database instances to isomorphic output database instances in order to preserve symmetry among the elements in the database and in order to avoid considering queries whose answer would depend on the implementation of relation instances as ordered sets. Also this allows one to avoid queries inventing values.

A *first order query* is represented by an expression of the form $\{\vec{x} \mid \phi(\vec{x})\}$ where ϕ is a first order formula on the language constructed from the predicate symbols in the database schema. Given an input database instance $(D, \boldsymbol{I})$, the output relation instance produced by the first order query $\{\vec{x} \mid \phi(\vec{x})\}$ is the set of facts out_$p(\vec{c})$ such that $\phi(\vec{c})$ is satisfied by $(D, \boldsymbol{I})$.

In the following, the first order query language is denoted FO.

In the database community, first order queries have a favored place and the expressive power of database query languages are often measured with respect to the expressive power of FO. The notion of complete query language is indeed

defined in terms of "as expressive as FO". The relational algebra is an instance of a complete database query language since it is as expressive as FO [37].

Note that FO defines a strict subclass of computable queries. First order queries are, of course, computable queries. However, it is easy to note that all computable queries are not in FO. The well-known transitive closure query is an instance of a computable query which is not definable by a FO query.

A natural way to increase the expressive power of FO is to add a fixpoint construct. Let P be a predicate symbol not in the database schema (a variable predicate symbol) and let $\phi(\vec{x})$ be a f.o. formula for P (the arity of P is assumed to match the length of $\vec{x}$).

We say that $\phi(\vec{x})$ is positive for P iff each occurrence of P in ϕ is under an even number of negation.

We say that $\phi(\vec{x})$ is monotone for P iff given two input databases (D, I) and (D, I') which only differs on P in such a way that I on P is included in I' on P, then the answer of the f.o. query $\{x \mid \phi(\vec{x})\}$ on (D, I) is included in the answer of the f.o. query $\{x \mid \phi(\vec{x})\}$ on (D, I').

The fact that the formula $\phi(\vec{x})$ for P is positive entails that it is monotone and the monotonicity of $\phi(\vec{x})$ entails that the equation $P(\vec{x}) \leftrightarrow \phi(\vec{x})$ has a least fixpoint on any instance database over the schema containing the predicate symbols in ϕ except P.

The least fixpoint of $\phi(\vec{x})$ on (D, I) is the instance database out_P (over the schema containing the predicate symbol P) where out_P is classically defined by out_$P = \bigcup_{j=1}^{\infty}$ out_P_j, and

(1) out_$p_0 = \emptyset$, and

(2) out_p_{j+1} is the output instance produced by applying the f.o. query $\phi(\vec{x})$ on $(D, I \cup \text{out\_}p_j)$.

Informally, a constructor fp is introduced which, applied to a predicate P and a formula ϕ where ϕ is positive for P, constructs the least fixpoint of the formula $(P(x) \leftrightarrow \phi(\vec{x}))$. This construction is denoted by fp$[P, \phi(\vec{x})]$.

Then *fixpoint formulas* are obtained from the first order constructors and the fixpoint constructor fp. A *fixpoint query* is an expression of the form $\{\vec{x} \mid \phi(\vec{x})\}$ where $\phi(\vec{x})$ is a fixpoint formula.

For example, assume that the arcs of a graph are represented by the binary predicate symbol Arc. The transitive closure of the graph is expressed by the fixpoint query $\{(x, y) \mid \text{fp}[\text{tc\_Arc}, \phi(x, y)]\}$ where

$$\phi(x, y) \text{ is the positive formula } \text{Arc}(x, y) \vee \exists z \quad (\text{tc\_Arc}(x, z) \wedge \text{Arc}(z, y)).$$

In the following the fixpoint query language is denoted by FP. As a matter of fact, FP is a strict subclass of computable queries. It is noted in [32] that: "One capability missing from fixpoint queries is that of counting. For example fixpoint queries cannot tell if the size of a relation is even or odd...". It should be stressed here that the complement of a fixpoint query can be expressed itself as a fixpoint (because finite domains are considered) [59].

Other query languages can be defined using fixpoints. The first "extension" that one can consider is obtained by allowing the application of the fixpoint construct not only to positive formulas (formulas in which the variable predicate symbol occurs only under even number of negations) but to monotonic formulas in general. Although there exists some monotonic first order formula which is not equivalent to any positive first order formula [5], it turns out that adding the fixpoint construct to the first order constructs on monotonic formulas yields the same class of queries as FP (because finite domains are considered) [57]. The problem arising here is that monotonicity is not a decidable property.

The second "extension" consists in allowing any formula (monotonic or not) and adding the inflationary construct to the usual first order constructs. Thus, given an input database instance (D, I), the output relation instance produced by ifp$[P, \phi(\vec{x})]$ is the set of facts out_$p = \bigcup_{j=1}^{\infty}$ out_p_j where:

(1) out_$p_0 = \emptyset$ and

(2) out_p_{j+1} is the *union* of out_p_j and of the output relation produced by applying the query $\phi(\vec{x})$ on $(D, I \cup \text{out\_}p_j)$.

If the formula $\phi(\vec{x})$ is monotone, its least fixpoint coincides with its inflationary fixpoint.

The query language obtained by adding the inflationary fixpoint construct to the first order constructs is denoted by IFP.

Although IFP seems more general than FP because any formula is allowed, once again it turns out that IFP = FP (because finite domains are considered) [57].

9.2. Expressive power of rule based languages for databases

In the following, logic programs denote queries. This is done by partitioning the set of predicate symbols into two sets:

(1) a set of "EDB" predicate symbols which intuitively correspond to relation instances stored in the database, and

(2) a set of "IDB" predicate symbols which are defined by the (non-unit) rules of the program.

Given a logic program $\mathscr{P}$, it is assumed that the EDB predicate symbols occurs only in the body of rules. One of the IDB predicate symbols P is distinguished and roughly speaking is used to collect the answers of the query specified by $\mathscr{P}$. We say that $\mathscr{P}$ is a logic program for P.

Now, given a logic program $\mathscr{P}$ for P and a (input) database instance (D, I), the output relation database (D, J) produced by $\mathscr{P}$ with respect to the "x"-semantics is defined in a natural way: (D, J) is the restriction on the predicate symbol P of the "x"-fixpoint (or "x"-model) of the logic program $\mathscr{P} \cup I$.

For the well-founded semantics, because the well-founded model of a logic program is a partial Herbrand model and thus contains positive and negative literals, the restriction on the predicate P of the well-founded model of $\mathscr{P} \cup I$ designates the positive portion of the literals related to the predicate symbol P.

For example the logic program expressing the transitive closure of a graph where the arcs are represented by the EDB predicate symbol Arc is the logic program *TC_ARC* for the predicate tc_Arc and it contains the two following rules:

- tc_Arc(x, y) :- Arc(x, y),
- tc_Arc(x, y) :- Arc(x, z) & tc_Arc(z, y).

Assume that the database graph contains the following arcs Arc(a, b), Arc(b, a) and Arc(c, a). The answer to the query specified by *TC_ARC* for tc_Arc with respect to the least fixpoint semantics (or with respect to the iterative fixpoint semantics or the well-founded semantics or the inflationary semantics) is the restriction on tc_Arc of the least fixpoint (or of the iterative fixpoint, or the positive part of the well-founded model, or of the inflationary fixpoint) of the program obtained by adding the facts Arc(a, b), Arc(b, a) and Arc(c, a) to the two above rules, that is {tc_Arc(a, b), tc_Arc(b, a), tc_Arc(c, a), tc_Arc(a, a), tc_Arc(b, b), tc_Arc(c, b)}.

A logic programming query language (or rule based query language) is characterized by the type of rules allowed (positive rules, stratifiable programs) and also by the semantics considered. In the following, we consider the following query languages.

- Datalog is the query language obtained by considering positive logic programs together with the standard least fixpoint semantics presented in Section 3.
- $\text{Datalog}^{\text{Neg}}_{\text{Strat}}$ is the query language obtained by considering stratifiable logic programs together with the (standard) iterative fixpoint semantics presented in Section 4 (or any other semantics presented in the paper with the exception of the inflationary semantics).
- $\text{Datalog}^{\text{Neg}}_{\text{Well-f}}$ is the query language obtained by considering logic programs (without any syntactical restriction) together with the well-founded semantics presented in Section 5.
- $\text{Datalog}^{\text{Neg}}_{\text{Infl}}$ is the query language obtained by considering logic programs (without any syntactical restriction) together with the inflationary semantics presented in Section 6.

It is quite easy to note that:

Lemma 9.1. *FO and Datalog are not comparable.*

Simply:

- The complement of a relation with respect to another relation cannot be expressed by a Datalog query. However such a query can be expressed for instance in the relational algebra using the set difference. Since the relational algebra as the same expressive power as FO, such a query is a first order query.
- The transitive closure cannot be expressed by a FO query. This has already been highlighted. However such a query can be expressed by a positive logic program.

Note that the class of conjunctive queries (which are obtained by removing negation $\neg$ and universal quantification $\forall$ from the first order construct in FO) is a strict subclass of Datalog (and of course of FO).

Theorem 9.2. *Datalog is a strict subclass of FP.*

This last result is a consequence of the fact that a Datalog query can be shown to be equivalent to a fixpoint query where the fixpoint construct is applied on a first order formula containing no universal quantification or negation [30, 105]. Thus Datalog queries cannot express FO queries including universal quantification or negation.

Intuitively, it is clear that $\text{Datalog}^{\text{Neg}}_{\text{Strat}}$ has more expressive power than FO and Datalog. An example of a query which can be expressed in $\text{Datalog}^{\text{Neg}}_{\text{Strat}}$ but is not definable in FO or Datalog is the complement of the transitive closure of a graph. This query is expressed by the stratifiable logic program *COMP_TC_ARC* for comp_tc_Arc whose first stratum is the program *TC_ARC* and whose second stratum is formed by the unique simple rule

comp_tc_Arc(x, y) :- not tc_Arc(x, y).

Indeed, the iterative fixpoint of a stratifiable logic program is a Δ^1_1 relation [63]. Thus:

Lemma 9.3. *FO and Datalog are strict subclasses of* $\text{Datalog}^{\text{Neg}}_{\text{Strat}}$.

However, it has been shown in [63] that $\text{Datalog}^{\text{Neg}}_{\text{Strat}}$ cannot express all FP queries. In particular, it cannot express fixpoint queries involving fixpoints over universal quantifiers.

Theorem 9.4 (Kolaitis [63], Dahlaus [38]). $\text{Datalog}^{\text{Neg}}_{\text{Strat}}$ *is a strict subclass of FP.*

A FP query which is not expressible in $\text{Datalog}^{\text{Neg}}_{\text{Strat}}$ is exhibited in [63]: it uses the "game tree" structures of [30]. We prefer to present here a simpler example that has been utilized in [8] for another purpose.

Once again consider a graph whose arcs are represented by the binary predicate symbol Arc(x, y). A node of the graph is a good one if all its incoming arcs originate from other good nodes. The query "finds all good nodes in the graph" is expressed by the following FP query fp[Good, $\phi(x)$] where

$\phi(x)$ is the first order formula $(\forall y\ \text{Arc}(y, x) \rightarrow \text{Good}(y))$.

There exists no stratifiable logic program which expresses this query.

Indeed, the separation of FP and $\text{Datalog}^{\text{Neg}}_{\text{Strat}}$ is tight because every formula of fixpoint logic over a vocabulary having unary predicate symbols only is equivalent to a first order formula on finite structure.

Apt and Blair [2] provides a study of the recursion theoretic complexity of the iterative fixpoint of stratifiable logic programs. This study is carried on in the context

of finitely generated Herbrand universe (at leat one function symbol in the language). The main result is that:

Theorem 9.5 (Apt and Blair [2]).

(1) *If $\mathscr{P}$ is a stratified logic program with n strata, then the iterative fixpoint of $\mathscr{P}$ is in Σ_n^0 and*

(2) *For each $n \geqslant 1$ there exists a stratified logic program $\mathscr{P}$ with n strata whose iterative fixpoint is Σ_n^0-complete.*

The next language to be examined here is of course $\text{Datalog}_{\text{Well-f}}^{\text{Neg}}$. This language allows more general forms of logic programs, indeed any logic program. Therefore, one can expect $\text{Datalog}_{\text{Well-f}}^{\text{Neg}}$ to be more expressive than $\text{Datalog}_{\text{Strat}}^{\text{Neg}}$.

In [112], a transformation of FP queries into $\text{Datalog}_{\text{Well-f}}^{\text{Neg}}$ queries is provided. Conversely, the existence of a representation of $\text{Datalog}_{\text{Well-f}}^{\text{Neg}}$ queries by FP queries is proved. This yields the following interesting result.

Theorem 9.6 (VanGelder [112]). *FP and* $\text{Datalog}_{\text{Well-f}}^{\text{Neg}}$ *have the same expressive power* (*for finite domain*).

To illustrate this result, we present below the logic program for good_node which expresses the "good node" query (with respect to the *positive part* of the well-founded semantics for logic programs). In order to express this query, a new predicate bad(x) is introduced. The program is very simple. It consists of the two following rules:

- good_node(x) :- not bad(x),
- bad(x) :- Arc(x, y) & not good_node(y).

We emphasize the restriction to the positive part of the well-founded model of programs in the above result.

In fact, given a graph I (for instance consider the graph given by Arc(a, b), Arc(b, c), Arc(c, a) and Arc(d, e)):

- The logic program $\mathscr{P}$ used to compute the answer of the "good node" query and obtained by adding I to the two above rules is not necessarily dynamically stratifiable and,
- . As a consequence, its well-founded model is not total (i.e. is not maximally consistent) which means that the "truth" of some facts (for instance here, the truth of good_node(a)) remains unknown.
- More precisely, let us consider the completed answer $\mathscr{J}$ to the "good node" query, the restriction of the well-founded model of $\mathscr{P}$ to the facts (positive and negative) related to the predicate symbol good_node is not equal to $\mathscr{J}$. Here $\mathscr{J} = \{\text{good\_node}(d), \text{good\_node}(e), \neg\text{good\_node}(a), \neg\text{good\_node}(b), \neg\text{good\_node}(c)\}$ and although the positive part pos($\mathscr{J}$) of $\mathscr{J}$ is included in the well-founded model of $\mathscr{P}$, $\mathscr{J}$ itself is not included in the well-founded model of $\mathscr{P}$. The well-founded model of $\mathscr{P}$ contains none of the negative facts in $\mathscr{J}$.
- This leads also to the fact that the rules of $\mathscr{P}$ fail to express the "good node" query with respect to the stable/default semantics. It suffices to note that for our

example $\mathscr{P} \cup \{\text{Arc}(a, b), \text{Arc}(b, c), \text{Arc}(c, a), \text{Arc}(d, e)\}$ has no stable/default model. The above discussion comes as an illustration for the remark from [112]: "This suggests the general rule that the alternating fixpoint partial model [i.e. the well-founded model] captures the negation of positive existential closures (such as the transitive closure), but not the negation of positive universal closures."

As for $\text{Datalog}^{\text{Neg}}_{\text{Well-f}}$, the $\text{Datalog}^{\text{Neg}}_{\text{Infl}}$ query language allows any logic program to specify query. Although inflationary models of logic programs are less "natural" than well-founded models, a prime advantage of $\text{Datalog}^{\text{Neg}}_{\text{Infl}}$ lies in its expressive power.

Theorem 9.7 (Abiteboul and Vianu [8], Kolaitis and Papadimitriou [66]). $\text{Datalog}^{\text{Neg}}_{\text{Infl}}$ *and IFP* $(= FP)$ *have the same expressive power.*

We reproduce here the logic program of [8] that is used to illustrate the simulation of IFP (= FP) by $\text{Datalog}^{\text{Neg}}_{\text{Infl}}$. The following logic program for good_node(x) does express the "good nodes" query (with respect to the inflationary semantics for logic programs).

- $\text{bad}_0(x)$:- $\text{Arc}(x, y)$ & $\text{good}(y)$,
- c_0,
- good_node(x) :- not $\text{bad}_0(x)$ & c_0,
- good_node(x) :- result(x, t),
- good(x, t) :- good_node(x) & good_node(t) & not old(t),
- old(x) :- good_node(x),
- bad(x, t) :- Arc(y, x) & old(t) & not good(y, t),
- $c(t)$:- old(t),
- result(x, t) :- not bad(x, t) & $c(t)$.

As explained in [8], the above inflationary logic program is designed in order to simulate the consecutive iterations of the IFP formula ifp[Good, $\phi(x)$] where

$$\phi(x) \text{ is the first order formula } (\forall y \quad \text{Arc}(y, x) \rightarrow \text{Good}(y)).$$

To conclude this section, note that none of the logic programming languages presented here can express all computable queries defined as the largest set of "reasonable" queries. Query languages able to express all computable queries are studied in [28]. More recently the problem is addressed in [7, 9] in a larger framework. Although a presentation of these languages is out of the scope of this paper, the interested reader should find interesting discussions and results in [7, 8, 9, 32].

10. Concluding remarks

Many aspects and developments of "rule based languages" have not been discussed in the paper which nevertheless are of prime importance.

In this section, we first motivate our choice not to present the Clark's completion approach to defining the declarative semantics of logic programs with negation. Then, we focus our attention on a very important aspect of logic programming with negation, namely the evaluation procedure of logic programs with negation. Finally, we say a few words on the extensions of Datalog (in other words the extensions of positive logic programs) which do not involve (only) adding negation in the premise of program rules.

10.1. (Why not) Clark's completion

The paper includes neither a presentation nor a comparative discussion of Clark's completion. Let us briefly explain this choice. While Clark's completion is a fundamentally important contribution to logic programming, the completion approach focuses essentially on the following specific problem.

A procedural semantics of logic programs with negation is given by means of the negation as failure resolution procedure (SLDNF) in [35]. SLDNF is a top-down evaluation procedure. It is an intuitively simple extension of SLD-resolution which consists in defining the notion of a proof of a negative elementary goal as the failure to obtain a proof of the corresponding positive goal. More precisely, the notion of failure is the reverse notion of success and goal $\leftarrow$ not Q succeeds if the goal $\leftarrow Q$ *finitely* fails while the goal $\leftarrow$ not Q finitely fails if the goal $\leftarrow Q$ succeeds.

SLDNF is an ineffective evaluation procedure because it is a non-deterministic procedure and a sound implementation of SLDNF would require a depth first search strategy. Not all queries may be processed by SLDNF. The impossibility of handling "floundering queries" (negative goals with variables) is a very difficult problem as well as a very important limitation of SLDNF.

Prolog [34, 65, 102] implements a restrictive form of SLDNF in the sense that Prolog's proofs have a specific form determined by a fixed selection rule on literals and by a depth first search strategy on rules.

One of the main problems which retains the attention of the logic programming community is to provide a declarative semantics of logic programs in order to "validate" the procedural SLDNF semantics. This is testified by the vast number of papers addressing this problem [e.g., 35, 48, 68, 76, 77, 75, 90].

By a semantics which validates SLDNF, is meant a semantics leading to establish the soundness and completeness results. Intuitively, soundness establishes that answers obtained by SLDNF evaluation of a query on a logic program are satisfiable by or derivable from the declarative meaning associated with the program. Completeness is the reverse notion. Completeness states that answers satisfiable by or derivable from the declarative meaning of a logic program can be retrieved by SLDNF evaluation. Soundness together with completeness assure a total correspondence between the declarative meaning and the procedural meaning of logic programs. Clark's completion is the main approach investigated for this purpose.

Given a logic program $\mathscr{P}$, its completion Comp($\mathscr{P}$) is constructed by considering a new predicate symbol = for equality, and by considering the first order theory

containing the equality axioms plus the completed definition of each predicate symbol. Intuitively, the completed definition of a predicate symbol is obtained by replacing the symbol :- ("if") by the logical equivalence ↔ ("iff") in the program rules. When a predicate P has no occurrence in the program, the universal closure of $\neg P(x)$ is introduced in Comp($\mathscr{P}$).

The first obstacle concerning soundness and completeness results of SLDNF with respect to completion is that Comp($\mathscr{P}$) may be an inconsistent theory. If Comp($\mathscr{P}$) is inconsistent then anything can be derived from Comp($\mathscr{P}$). In particular anything that can be computed by SLDNF can be derived from Comp($\mathscr{P}$). Hence soundness is trivial and meaningless in this case. On the other hand, it is quite clear that if a query succeeds, it cannot fail at the same time and vice versa. Thus it is rather obvious that SLDNF is not complete (for logic programs with inconsistent Comp($\mathscr{P}$)).

This obstacle cannot be neglected because the problem of deciding whether Comp($\mathscr{P}$) is consistent is recursively undecidable.

For consistent (with respect to Comp) logic programs, SLDNF is sound. For establishing completeness, it is not sufficient to avoid inconsistent programs, it is necessary to introduce some other restrictions, but this time on the type of queries (programs) considered. This is essentially due to the fact that even for a logic program $\mathscr{P}$ with Comp($\mathscr{P}$) consistent, it may arise that a query Q neither succeeds nor fails (but has an infinite evaluation tree) and at the same time Q may be a logical consequence of Comp($\mathscr{P}$).

Restrictions such as *hierarchy* and *allowedness* are introduced in order to overcome that kind of problem and in order to establish the (weak) completeness result.

The "weak" correspondence between the SLDNF procedural semantics of programs and the completion of programs suffers from the following drawback. The completion approach tends to embody, at a declarative level, the undesirable features of SLDNF. At the declarative level, the non-termination of the evaluation of a query Q on a program $\mathscr{P}$ is conveyed by the non-derivability of Q and the non-derivability of $\neg Q$ from Comp($\mathscr{P}$).

It is commonly asserted that when one writes a logic program $\mathscr{P}$ what one really has in mind is Comp($\mathscr{P}$). For instance, if one states that "If I come then I will bring a cake" what one really means is "I will bring a cake iff I come". As it is nicely discussed in [104], this point of view of the programmer's intention is justifiable in simple cases but gets less easy to apprehend when the program is more involved and in particular when the program contains recursive rules.

It is interesting to report here an important result established recently by Kunen [69]. Intuitively, this result says that the transitive closure of a relation is not definable by the completion semantics (and cannot be computed by SLDNF). This result strengthens the remark of [111] that "the usual rules to define transitive closure of a directed graph did not yield the value *false* on pairs of nodes not in the transitive closure." Before presenting an example illustrating this remark, recall that, in the context of a database, rule based languages have been looked at in order to specify

more expressive query languages and specifically to provide languages able to express transitive closure queries.

Consider a simple graph. The existence of an arc between two nodes a and b in the graph is represented by asserting Arc(a, b). The usual rules to define the transitive closure tc_Arc of the relation Arc are:

(a) tc_Arc(x, y) :- Arc(x, y),

(b) tc_Arc(x, y) :- Arc(x, z) & tc_Arc(z, y).

Assume that we have the following arcs in the graph: Arc(a, b), Arc(b, a), Arc(c, a). Let us consider the query $\leftarrow$ tc_Arc(a, c).

(1) The SLDNF evaluation of the query $\leftarrow$ tc_Arc(a, c) enters an infinite loop and fails to provide an answer:

(i) The goal tc_Arc(a, c) unifies with the head of rule (b) and, the first new goal $\leftarrow$ Arc(a, z) & tc_Arc(z, c) is obtained.

(ii) The first literal Arc(a, z) of the goal $\leftarrow$ Arc(a, z) & tc_Arc(z, c) unifies with the fact Arc(a, b) and the second new goal generated is $\leftarrow$ tc_Arc(b, c).

(iii) The goal $\leftarrow$ tc_Arc(b, c) unifies with the head of rule (b) and the new goal $\leftarrow$ Arc(b, z) & tc_Arc(z, c) is obtained.

(iv) The first literal Arc(b, z) of the newly obtained goal $\leftarrow$ Arc(b, z) & tc_Arc(z, c) unifies with the fact Arc(b, a) and it follows that $\leftarrow$ tc_Arc(a, c) is the next goal to be examined. This happens to be the initial goal $\leftarrow$ tc_Arc(a, c).

(2) The completed definition of the predicate symbol tc_Arc defined by the rules (a) and (b) is the following first order formula:

$$\text{tc\_Arc}(x, y) \leftrightarrow (\text{Arc}(x, y) \vee (\exists z \quad (\text{Arc}(x, z) \wedge \text{tc\_Arc}(z, y)))).$$

Neither tc_Arc(a, c) nor $\neg$Arc(a, c) are logical consequences of the completion of the program.

(3) (Bottom-Up Evaluation) Let us evaluate the query $\leftarrow$ tc_Arc(a, c) in a bottom up fashion. Simply, the following facts related to the predicate tc_Arc are generated:

$$\text{tc\_Arc}(a, b), \quad \text{tc\_Arc}(b, a), \quad \text{tc\_Arc}(c, a),$$

$$\text{tc\_Arc}(a, a), \quad \text{tc\_Arc}(b, b), \quad \text{tc\_Arc}(c, b).$$

The fact tc_Arc(a, c) is not among the generated facts thus the naive bottom-up evaluation of $\leftarrow$ tc_Arc(a, c) returns the answer NO ($\neg$tc_Arc(a, c)).

(4) The least (Herbrand) model of the program satisfies $\neg$tc_Arc(a, c).

In conclusion, for the class of positive programs, the above examples and discussion show that, from the database point of view,[15] the completion approach is unsatisfactory. Declarative semantics of logic programs defined in terms of the logical consequences of Comp($\mathscr{P}$) or in terms of 2-valued models of Comp($\mathscr{P}$) as well as in terms of 3-valued models of Comp($\mathscr{P}$) all reflect these shortcoming. Taking the risk to be brutal, we will say that, from the database point of view,

[15] We insist: from the database point of view.

"interesting" results for logic programs with negation are unlikely to be obtained using this approach.

Clark's completion and the works related to it are presented and discussed in detail in [35, 104, 75, 48, 47, 68, 69].

10.2. Evaluation procedures

The previous section on Clark's completion approach and SLDNF introduces the presentation of evaluation procedures for logic programs with negation. The purpose of the paper was essentially to discuss the declarative aspects of rule-based languages with negation. The procedural semantics of logic programs with negation is nevertheless equally important. One could even find it awkward to dissociate these two aspects. On the one hand, in order to be able to *use* a given declarative semantics, a corresponding procedural semantics is necessary. On the other hand, it seems unreasonable to define the computation of programs without knowing what should be computed.

There has been much research into rule-based query evaluation during the past ten years. Most of this work has concentrated on evaluating recursive queries, that is queries specified by positive logic programs. The underlying declarative semantics considered in this context is the minimal model (or least fixpoint) semantics. Various strategies have been proposed which are reviewed in [23]. In general function symbols other than constants are ruled out. Restrictions on the programs considered are made in order to avoid having infinite answers to compute (the use of evaluable predicates combined with free variables in the head of rules which does not appear in the body is a source of unsafeness). The major characteristics of these procedures are: top-down versus bottom-up, and recursive versus iterative. Among the evaluation procedures proposed, let us quote the naive evaluation procedure which is a bottom-up iterative strategy and is a direct "implementation" of the immediate consequence operator, QSQR [113, 114] which is one of the most interesting top-down recursive strategy, the Magic Sets, Counting and Reverse Counting [22, 24, 107] which belong to the class of optimization strategies.

Concerning stratifiable logic programs and the iterative fixpoint semantics, most of the procedures designed for positive logic programs can be easily extended. This is not really surprising, because a stratifiable logic program is a sequence of semi-positive logic programs.

In [3], the notion of "interpreter" is formally defined for logic programs. Problems such as ambiguity and non-computability are identified. Essentially, the contribution of [3] is twofold. It shows that, if there exists an interpreter for a stratifiable logic program, then it "computes" the iterative fixpoint model of that program (interpreter of a stratifiable logic program is not ambiguous). The existence of an interpreter for stratifiable logic programs is proved, the computability of the interpreter is guaranteed for stratifiable logic programs without function symbols other than constants.

Przymusinski [95] introduces SLS-resolution (*Linear Resolution with Selection function for Stratified Programs*) as a natural generalization of SLD-resolution from the class of positive logic programs to the class of stratifiable logic programs. SLS-resolution is a modification of SLDNF-resolution. In this framework, the declarative semantics of logic programs is given by the class of all (*not necessarily Herbrand*) perfect model models of programs [95]. Considering non-Herbrand models is motivated by the principle that "positive information not derivable from the f.o. notation of a program should not be either derivable from the declarative semantics assigned to the program". Note that for SLS-resolution, an infinite branch of an SLS-resolution is regarded as failed. As usual, a goal (query) is failed if each branch of an SLS-tree is (not necessarily finitely) failed. As stated in [95], SLS-resolution can be considered as an "ideal" sound and complete evaluation procedure for stratifiable logic programs, but it is not effective. Let us have a brief look at some "*effective approximations*" of SLS-resolution.

Kemp and Topor [67] propose an extension of Vieille's QSQR/SLD for stratifiable logic programs. Roughly speaking, the extension is "straightforward" and consists in forcing the complete evaluation of subqueries issued from negative (ground) subgoals. The contribution of [67] is to provide an effective query evaluation procedure which is sound and complete.

In the same spirit, let us quote [106] which proposes an extension of OLD-resolution with tabulation [109].

Bry [26] proposes an extension of the Generalized Magic Set optimization strategy in order to deal with (loosely) stratifiable logic programs.

Procedural semantics for unstratifiable is still under study. Przymusinski [96] extends SLS-resolution [95] from the class of stratifiable programs (under the iterative fixpoint model semantics) to the class of all programs (under the well-founded semantics).

Independently, [101] provides a procedural semantics for well-founded programs, called global SLS-resolution, which extends SLS-resolution [95]. As discussed in [101], there are three sources of the non-effectiveness in global SLS-resolution: infinite branches are treated as failed, the SLP-tree for a goal may have an infinite number of branches and if a goal is indeterminate, global SLS-resolution will recurse infinitely through negation.

Ross [101] suggests that developing an effective top-down procedure should provide some form of loop checking to handle the above mentioned problems. Legay [71] started to investigate this direction and proposes an extension of QSQR [113, 114]. Intuitively, QSQR is modified in order to compute in a top-down manner founded and potentially founded facts. The loop checking principle of QSQR (a stack of subqueries) "traps" negative recursion in the same way as it "traps" positive recursion. This method does not avoid the problem of floundering queries and logic programs are restricted to those in which variables in head of rules and in negative premises occur in positive premises.

10.3. Other extensions of Datalog

Extending Datalog (in other words, extending positive logic programs) is not limited to adding negation in the premise of program rules. Interesting extensions, motivated by various database concepts such as complex objects, updates and incomplete information, have been investigated.

In the context of complex object adding sets to rule-based query languages has been investigated by introducing special predicates, data functions or special constructor like groupings [6, 10, 70]. The language COL [6], for instance, integrates sets, data functions and negation; its semantics is based on minimal model; because of sets and data functions, some COL programs may have more than one minimal model; a notion of stratification is used; finally negation can be simulated using data functions.

Adding tuple deletion has been recently investigated by allowing negative consequences in program rules providing an update oriented extension of Datalog [7, 42].

References

[1] S. Abiteboul and N. Bidoit, Non first normal form relations: an algebra allowing data restructuring, *Comput. System Sci.* **33**(3) (1986) 361–393.

[2] K.R. Apt and H.A. Blair, Arithmetic classification of perfect models of stratified programs, Technical Report TR-88-09, University of Texas at Austin, 1988.

[3] K.R. Apt, H. Blair and A. Walker, Towards a theory of declarative knowledge, in: *Proc. Workshop on the Foundations of Deductive Databases and Logic Programming* (1986) 546–628; also in [85].

[4] K.R. Apt and M.H. Van Emden, Contributions to the theory of logic programming, *ACM* **29**(3) (1982) 841–862.

[5] M. Ajtai and Y. Gurevich, Monotone versus Positive, *ACM* **34**(4) (1987) 1004–1015.

[6] S. Abiteboul and S. Grumbach, COL: a logic-based language for complex objects, in: *Internat. Conf. on Extending Data Base Technology* (1988) 271–293.

[7] S. Abiteboul and V. Vianu, Datalog extensions for database queries and updates, Technical Report 900, INRIA, 1988; to appear in *J. Comput. System Sci.*

[8] S. Abiteboul and V. Vianu, Procedural and declarative database update languages, in: *Symp. on Principles of Database Systems, ACM SIGACT-SIGMOD-SIGART* (1988) 240–250; to appear in *J. Comput. System Sci.*

[9] S. Abiteboul and V. Vianu, Fixpoint extensions of first order logic and Datalog-like languages, in: *Proc. Internat. Conf. on Logic in Computer Science* (IEEE, 1989) 71–79.

[10] C. Beeri et al., Sets and Negation in a Logic Database Language (LDL1), in: *Symp. on Principles of Database Systems, ACM SIGACT-SIGMOD* (1987) 21–37.

[11] F. Bancilhon, On the completeness of query languages for relational data base, in: *Symp. Mathematical Foundations of Computer Science* (Springer, Berlin, 1978).

[12] F. Bancilhon, Object oriented database systems, in: *Symp. on Principles of Database Systems, ACM SIGACT-SIGMOD-SIGART* (1988).

[13] J. Banerjee, H.T. Chou, J.G. Garza, W. Kim, D. Woelk, N. Ballou and H.-J. Kim, Data model issues for object oriented applications, in: *ACM Transactions on Database System* (1987).

[14] N. Bidoit and C. Froidevaux, General logic databases and programs: default logic semantics and stratification, Technical Report, LRI, 1987; to appear in *J. Inform. and Comput.*

[15] N. Bidoit and C. Froidevaux, Minimalism subsumes default logic and circumscription, in: *Proc. Conf. on Logic in Computer Science* (IEEE, 1987) 89–97.

[16] N. Bidoit and C. Froidevaux, More on stratified default theories, in: *Proc. European Conf. on Artificial Intelligence* (1988) 492-494.
[17] N. Bidoit and C. Froidevaux, Negation by default and unstratifiable logic programs, Technical Report 437, LRI, 1988; to appear in a special issue of *Theoret. Comput. Sci.* on Research in Deductive Databases.
[18] N. Bidoit and R. Hull, Positivism versus minimalism in deductive databases, in: *Symp. on Principles of Database Systems, ACM SIGACT-SIGMOD-SIGART* (1986) 123-132.
[19] N. Bidoit and R. Hull, Minimalism, justification and non-monotonicity in deductive databases, *Comput. System Sci.* **38**(2) (1989) 290-325.
[20] N. Bidoit, Negation in rule based database languages: a survey (1989).
[21] H.A. Blair, The undecidability of the two completeness notions for negation as failure in logic programming, in: Ed. Van Canegen, ed., *Proc. First Logic Programming Conf.* (1982) 163-168.
[22] F. Bancilhon, D. Maier, Y. Sagiv and J. Ullman, Magic sets and other strange ways to implement logic programs, in: *Symp. on Principles of Database Systems, ACM SIGMOD-SIGACT* (1986).
[23] F. Bancilhon and R. Ramakrishnan, An amateur's introduction to recursive query-processing strategies, in: *Conf. on Management of Data, ACM SIGMOD* (1986) 16-52.
[24] C. Beeri and R. Ramakrishnan, On the power of magic, in: *Symp. on Principles of Database Systems, ACM SIGACT-SIGMOD-SIGART* (1987) 269-283.
[25] F. Bry, Logic programming as constructivism: a formalization and its application to databases, Technical Report, IR-KB-58, ECRC, 1988.
[26] F. Bry, Logic programming as constructivism: a formalization and its application to databases, in: *Symp. on Principles of Database Systems* (ACM, 1989).
[27] G. Bossu and P. Siegel, Saturation, non-monotonic reasoning and the closed world assumption, *Artificial Intelligence* **25** (1985) 13-63.
[28] A. Chandra and D. Harel, Computable queries for relational data bases, *Comput. System Sci.* **25**(2) (1980) 156-178.
[29] A. Chandra and D. Harel, Structure and complexity of relational queries, *Comput. System Sci.* **25**(1) (1982) 99-128.
[30] A. Chandra and D. Harel, Horn clause queries and generalizations, *Logic Programming* **2**(1) (1985) 1-15.
[31] D. Chan, Constructive negation based on the completed database, in: *Proc. Internat. Conf. on Logic Programming* (1988) 111-125.
[32] A.K. Chandra, Theory of database languages, in: *Symp. on Principles of Database Systems, ACM SIGACT-SIGMOD-SIGART* (1988) 1-9.
[33] P. Cholak, Post correspondence problem and Prolog programs, Technical Report, Univ. of Wisc., Madison, 1988.
[34] A. Colmerauer, H. Kanoui, R. Pasero and P. Roussel, Un système de Communication Homme-Machine en Français, Technical Report, Univ. d'Aix-Marseille II, Marseille, 1973.
[35] K.L. Clark, Negation as failure, in: H. Gallaire and J. Minker, eds., *Logic and Database* (Plenum Press, New York, 1978) 293-322.
[36] E.F. Codd, A relational model of data for large shared data banks, *Comm. ACM* **13** (1970) 377-387.
[37] E.F. Codd, Relational completeness of database sublanguages, in: *Data Base Systems* (1972).
[38] E. Dahlaus, Skolem normal forms concerning the least fixpoint, in: E. Borger, ed., *Computation Theory and Logic* (1987); also in Lecture Notes in Computer Science **270** (Springer, Berlin) 101-106.
[39] D. McDermott and J. Doyle, Non monotonic logic 1, *Artificial Intelligence* **13** (1980) 41-72.
[40] P. Deransart and G. Ferrand, A methodological view of logic programming with negation, in: *Actes du 8ème Seminaire de Programmation en Logique de Tregastel* (1989).
[41] W.F. Dowling and J. Gallier, Linear time algorithms for testing the satisfiability of propositional Horn formulae, *Logic Programming* **3** (1984) 267-284.
[42] C. de Maindreville and E. Simon, Modelling a production rule language for deductive databases, in: *Conf. on Very Large Data Bases* (1988).
[43] M. Davis and H. Putnam, A computing procedure for quantification theory, *ACM* **7** (1960) 201-215.
[44] M.H. Van Emden and R.A. Kowalski, The semantics of predicate logic as a programming language, *ACM* **23**(4) (1976) 733-742.
[45] H.B. Enderton, *A Mathematical Introduction to Logic* (Academic Press, New York, 1972).

[46] R. Fagin, Generalized first-order spectra and polynomial time recognizable sets, in: R. Karp, ed., *Proc. Conf. on Complexity of Computations, SIAM-AMS* (1974) 43–73.
[47] M. Fitting and M. Ben-Jacob, Stratified and three-valued logic programming semantics, in: *Proc. Internat. Conf. on Logic Programming* (1988) 1055–1069.
[48] M. Fitting, A Kripke-Kleene semantics for logic programs, *Logic Programming* **4** (1985) 295–312.
[49] M. Falaschi, G. Levi, M. Martelli and C. Palamidessi, A new declarative semantics for logic languages, in: *Proc. Internat. Conf. on Logic Programming* (1988) 993–1005.
[50] J. Gallier, *Logic For Computer Science—Foundations of Automatic Theorem Proving* (Harper and Row, New York, 1986).
[51] M. Gelfond, On stratified autoepistemic theories, in: *Proc. AAAI* (1987) 207–211.
[52] D. Van Gucht and P.C. Fischer, Some classes of multilevel relational structures, in: *Symp. on Principles of Database Systems, ACM SIGACT-SIGMOD* (1986).
[53] M. Gelfond and V. Lifschitz, The stable model semantics for logic programs, in: *Proc. Internat. Conf. on Logic Programming* (1988) 1070–1080.
[54] H. Gallaire and J. Minker, eds., *Logic and Data Bases* (Plenum Press, New York, 1978).
[55] H. Gallaire, J. Minker and J-M. Nicolas, Logic and databases: a deductive approach, *Comput. Surv.* (1984) 151–185.
[56] D.M. Gabbay and M.J. Sergot, Negation as inconsistency, *Logic Programming* **3**(1) (1986) 1–36.
[57] Y. Gurevich and S. Shelah, Fixed point extensions of first order logic, in: *Proc. 26th Symp. on Foundations of Computer Science* (1986) 346–353.
[58] R. Hull and R. King, Semantic data modelling: survey, applications and research issues, *ACM Comput. Surv.* **19**(3) (1987) 201–260.
[59] N. Immerman, Relational queries computable in polynomial time, *Inform. and Control* **68** (1986) 86–104.
[60] J. Jaffar, J. L. Lassez and M.J. Maher, Some issues and trends in the semantics of logic programming, in: Shapiro, ed., *Proc. Internat. Conf. on Logic Programming* (1986); also in Lecture Notes in Computer Science **225** (Springer, Berlin, 1986) 223–240.
[61] P. Kanellakis, Elements of relational theory, in: J. van Leeuwen, ed., *Handbook of Theoretical Computer Science, Vol. B* (Elsevier Science Publishers, Amsterdam, 1990) 1073–1156.
[62] S.C. Kleene, *Introduction to Metamathematics* (Van Nostrand, New York, 1952).
[63] P. Kolaitis, The expressive power of stratified logic programs, submitted to *Inform. and Comput.*
[64] S. Konolige, On the relation between default and autoepistemic logic, *Artificial Intelligence* **35**(3) (1988) 343–382.
[65] R.A. Kowalski, Predicate logic as a programming language, *Inform. Process.* **74** (1974) 574–579.
[66] P. Kolaitis and C. Papadimitriou, Why not negation by fixpoint, in: *Proc. Symp. on Principles of Database Systems, ACM, SIGACT-SIGMOD-SIGART* (1988) 231–239.
[67] D. Kemp and R. Topor, Completeness of a top-down query evaluation procedure for stratified databases, in: *Proc. Internat. Conf. on Logic Programming* (1988) 178–194.
[68] K. Kunen, Negation in logic programming, *Logic Programming* **4** (1987) 289–308.
[69] K. Kunen, Some remarks on the completed database, in: *Proc. Internat. Conf. on Logic Programming* (1988) 978–992.
[70] G.M. Kuper, Logic programming with sets, in: *Symp. on Principle of Database Systems, ACM, SIGACT-SIGMOD* (1988).
[71] P. Legay, Evaluation de requète en Datalog avec negation, Rapport de DEA d'Informatique, LRI, Université Paris XI, 1989.
[72] V. Lifschitz, Closed world databases and circumscription, *Artificial Intelligence* **27** (1985) 229–235.
[73] V. Lifschitz, Computing circumscription, in: *Proc. Internat. Joint Conf. on Artificial Intelligence* (1985) 121–127.
[74] V. Lifschitz, On the declarative semantics of logic programs with negation, in: *Proc. Workshop on the Foundations of Deductive Databases and Logic Programming* (1986) 420–432; also in [85].
[75] J.W. Lloyd, *Foundation of Logic Programming* (Springer, Berlin, 1987).
[76] J.L. Lassez and M.J. Maher, Closures and fairness in semantics of programming logic, *Theoret. Comput. Sci.* **29** (1984) 167–184.
[77] J.L. Lassez and M.J. Maher, Optimal fixedpoints of logic programs, *Theoret. Comput. Sci.* **39** (1985) 15–25.

[78] J. Lobo, J. Minker and A. Rajaseka, Weak completion theory for non-Horn programs, in: *Proc. Internat. Conf. on Logic Programming* (1988) 828-842.
[79] J.L. Lassez, V.L. Nguyen and E.A. Sonenberg, Fixed point theorems and semantics: a folk tail, *Inform. Lett.* **14**(3) (1984) 167-184.
[80] J.W. Lloyd and R.W. Topor, A basis for deductive data base systems, *Logic Programming* **2** (1985) 93-110.
[81] J. W. Lloyd and R.W. Topor, A basis for deductive data base systems II, *Logic Programming* **3** (1986) 55-68.
[82] J. M. Maher, Equivalence of logic programs, in: *Proc. Internat. Conf. on Logic Programming* (1986); also in Lecture Notes in Computer Science **225** (Springer, Berlin) 410-424.
[83] J. McCarthy, Circumscription—a form of nonmonotonic reasoning, *Artificial Intelligence* **13**(1) (1980) 27-39.
[84] J. Minker, On indefinite databases and the closed world assumption, in: *Proc. 6th Conf. on Automated Deduction* (1982); also in Lecture Notes in Computer Science **138** (Springer, Berlin) 292-308.
[85] J. Minker, ed., *Foundations of Deductive Databases and Logic Programming* (Morgan Kaufmann, 1988).
[86] J. Minker, Perspectives in deductive databases, *Logic Programming* **5**(1) (1988) 33-60.
[87] R.C. Moore, Semantics considerations on non-monotonic logic, *Artificial Intelligence* **25** (1985) 75-94.
[88] D. Maier, J. Stein, A. Otis and A. Purdy, Development of an object-oriented management system, in: *ACM OOPSALA* (1988).
[89] A. Marek and M. Truszczynski, Autoepistemic logic, Technical Report, University of Kentucky, 1988.
[90] A. Mycroft, Logic program and many-valued logic, in: *STACS* (1984); also in Lecture Notes in Computer Science **166** (Springer, Berlin) 274-286.
[91] S.A. Naqvi, A logic for negation in database systems, in: *Proc. Workshop on Foundations of Deductive Databases and Logic Programming* (1986) 378-387.
[92] T. Przymusinska and H. Przymusinski, Weakly perfect model semantics for logic programs, in: *Proc. Internat. Conf. on Logic Programming* (1988) 1106-1120.
[93] T. Przymusinska and H. Przymusinski, Semantic issues in deductive databases and logic programs, invited survey article to appear in: A. Barerji, ed., *Sourcebook on the Formal Approaches in Artificial Intelligence* (North-Holland, Amsterdam, 1989).
[94] T. Przymusinska, On the semantics of stratified deductive databases, in: *Proc. Workshop on the Foundations of Deductive Databases and Logic Programming* (1986) 433-443; also in [85].
[95] T. Przymusinska, Perfect model semantics, in: *Proc. Internat. Conf. on Logic Programming* (1988) 1081-1096.
[96] T. Przymusinska, Every logic program has a natural stratification and an iterated fixed point model, in: *Symp. on Principles of Database Systems, ACM, SIGACT-SIGMOD-SIGART* (1989) 11-21.
[97] R. Reiter, On closed world data bases, in: H. Gallaire and J. Minker, eds., *Logic and Data Bases* (Plenum Press, New York, 1978) 55-76.
[98] R. Reiter, A logic for default reasoning, *Artificial Intelligence* **13**(1) (1980) 80-132.
[99] R. Reiter, Towards a reconstruction of relational database theory, in: M.L. Brodie, J.L. Mylopoulos and J.W. Schmidt, eds., *On Conceptual Modelling* (Springer-Verlag, New York, 1984) 163-189.
[100] M.A. Roth, H.F. Korth and A. Silberschatz, Extended algebra and calculus for nested relational databases, *ACM Trans. Database Systems* **13**(4) (1988) 389-417.
[101] A. Ross, A procedural semantics for well founded negation in logic programs, in: *Symp. on Principles of Database Systems* (ACM, 1989) 22-33.
[102] P. Roussel, *PROLOG: Manuel de Référence et d'Utilisation* (Groupe d'Intelligence Artificielle, Marseille, 1975).
[103] T. Sato, Negation and semantics of Prolog programs, in: *Proc. First Internat. Conf. on Logic Programming* (1982) 169-174.
[104] J. Shepherdson, Negation in logic programming, in: J. Minker, ed., *Foundations of Deductive Databases and Logic Programming* (Morgan Kaufmann, 1988) 19-88.
[105] O. Shmueli, Decidability and expressiveness aspects of logic queries, in: *Symp. on Principles of Database Systems, ACM SIGACT-SIGMOD-SIGART* (1987) 237-249.

[106] H. Seki and H. Itoh, A query evaluation procedure for stratified programs under the extended CWA, in: *Proc. Internat. Conf. on Logic Programming* (1988) 196-211.

[107] D. Sacca and C. Zaniolo, On the implementation of a simple class of logic queries for databases, in: *Symp. on Principles of Database Systems, ACM SIGMOD-SIGACT* (1986).

[108] A. Tarski, A lattice theoretical fixpoint theorem and its application, *Pacific J. Math.* **5** (1955) 285-309.

[109] H. Tamaki and T. Sato, OLD resolution with tabulation, in: *Internat. Conf. on Logic Programming* (1986) 84-98.

[110] J.D. Ullman, *Principles of Database and Knowledge Base Systems* (Computer Science Press, 1988).

[111] A. Van Gelder, Negation as failure using tight derivation for general logic programs, in: *Proc. Third IEEE Symp. on Logic Programming* (1986) 137-146; also in [85].

[112] A. Van Gelder, The alternating fixpoint of logic programs with negation, in: *Symp. on Principles of Database Systems, ACM SIGACT-SIGMOD-SIGART* (1989) 1-10.

[113] L. Vieille, Recursive axioms in deductive databases: the query-subquery approach, in: *Internat. Conf. on Expert Database Systems* (1986) 179-193.

[114] L. Vieille, A database-complete proof procedure based on SLD-resolution, in: *Internat. Conf. on Logic Programming* (1987) 74-103.

[115] A. Van Gelder, K. Ross and J.S. Schlipf, Unfounded sets and well-founded semantics for general logic programs, in: *Symp. on Principles of Database Systems, ACM SIGACT-SIGMOD-SIGART* (1988) 221-230.