Logic Programming Revisited: Logic Programs as Inductive Definitions

MARC DENECKER and MAURICE BRUYNOOGHE
Katholieke Universiteit Leuven
and
VICTOR MAREK
University of Kentucky

Logic programming has been introduced as programming in the Horn clause subset of first order logic. This view breaks down for the negation as failure inference rule. To overcome the problem, one line of research has been to view a logic program as a set of iff-definitions. A second approach was to identify a unique *canonical*, *preferred* or *intended* model among the models of the program and to appeal to *common sense* to validate the choice of such model. Another line of research developed the view of logic programming as a non-monotonic reasoning formalism strongly related to Default Logic and Auto-epistemic Logic.

These competing approaches have resulted in some confusion about the declarative meaning of logic programming. This paper investigates the problem and proposes an alternative epistemological foundation for the *canonical model* approach, which is not based on common sense but on a solid mathematical information principle. The thesis is developed that *logic programming can be understood as a natural and general logic of inductive definitions*. In particular, *logic programs with negation represent non-monotone inductive definitions*. It is argued that this thesis results in an alternative justification of the well-founded model as the unique intended model of the logic program. In addition, it equips logic programs with an easy to comprehend meaning that corresponds very well with the intuitions of programmers.

Categories and Subject Descriptors: D.1.6 [**Programming Techniques**]: Logic Programming; D.3.1 [**Programming Languages**]: Formal Definitions and Theory—*Semantics*; F.3.2 [**Logics And Meanings Of Programs**]: Semantics of Programming Languages—*Algebraic approaches to semantics*; F.4.1 [**Mathematical Logic And Formal Languages**]: Mathematical Logic—*Computational logic*; *Logic and constraint programming*; I.2.3 [**Artificial Intelligence**]: Deduction and Theorem Proving—*Logic programming*; *Nonmonotonic reasoning and belief revision*; I.2.4 [**Artificial Intelligence**]: Knowledge Representation Formalisms and Methods

General Terms: Languages, Theory

Author's address: M. Denecker and M. Bruynooghe, Department Computer Science, Celestijnenlaan 200A, B 3001 Heverlee, Belgium. V. Marek, Computer Science Department, University of Kentucky, Lexington, KY 40506-0046,USA.
Research supported by the project GOA/98/08 and the research network Declarative Methods in Computer Science (both funded by the Flemish government). The third author acknowledges a partial support of the NSF grant IRI-9619233.

Additional Key Words and Phrases: Epistemological Foundations, Inductive Definitions

1. INTRODUCTION

Logic programming has its roots in the investigations of the resolution principle [Robinson 1965], an inference method for first order logic. Restricting the first order theories to Horn theories consisting of definite clauses and a definite goal, one could design proof procedures that avoid many of the redundancies showing up in the search spaces of the more general theorem provers of those days. Moreover, one could give a procedural reading to the definite clauses that corresponds to the strategy followed by SLD-proof procedures as explained in the seminal Kowalski [1974] paper. Meanwhile, the group of Colmerauer developed a programming language along the same lines and called it Prolog [Colmerauer et al. 1973] as abbreviation for *PROgrammation en LOGique*.

Many researchers were attracted by the new paradigm: application programmers by the ability to program at a, until then, unprecedented level of abstraction; implementors by the challenge to design and develop efficient implementations; theoreticians by the opportunity to analyze a paradigm rooted in logic.

Originally, logic programming was often summarized as *programming in a subset of first order logic*. Specifically, this subset is the Horn logic, based on Horn theories, that is theories consisting of clauses with at most one positive literal. Despite the fact that this view is still wide-spread, it broke down soon after Logic Programming originated. The introduction of the *negation as failure* rule raised the following dilemma to the Logic Programming community:

—On the one hand, the negation as failure inference rule was *unsound* with respect to the declarative reading of a program as a first order Horn theory [Clark 1978].

—On the other hand, negation as failure derived conclusions with a strong common sense appeal and turned out to be very useful and natural in many practical situations.

The way out was either to drop the negation as failure rule or to strengthen the interpretation of logic programs as Horn theories. The multiple and natural applications of negation as failure resulted in choosing the second option. What at the start seemed to be a hack became a feature. As Przymusinski [1989b] expressed it later, the Logic Programming community decided that "*we really do not want classical logic semantics for logic programs. ... We want the semantics of a logic program to be determined more by its common sense meaning.*". This raised the following fundamental question: what is this common sense meaning and how can we provide a formal semantics for it? The search for an answer to this question started in the late seventies and was intensively pursued until the early nineties. These investigations resulted in a complex and heterogeneous landscape of Logic Programming.

With respect to definite programs (i.e. programs without negation), the question was soon settled. While logicians [Smullyan 1968] knew for long time that consistent Horn theories possess a least Herbrand model, van Emden and Kowalski [1976] showed the existence of the least Herbrand model as the least fixpoint of a

monotone operator, the immediate consequence operator. A few years later, Reiter [1978] showed that the least Herbrand model was the unique intended interpretation of a Horn program augmented with the common sense reasoning principle of the Closed World Assumption. The least Herbrand model is now widely accepted as the intended interpretation of a definite logic program.

With respect to programs with negation, things turned out to be much more complex. There seemed to be different common sense ways in which a logic program could be interpreted. This resulted in three major research directions.

Clark [1978] proposed to interpret a logic program as a first order theory, called the *completion* of the program. It consists of a set of logical *iff-definitions* of which the rules of the programs represent only the *if*-parts, augmented with a theory that axiomatizes the unification. Although this approach resulted in a large body of research, including a three valued completion semantics for programs with negation [Fitting 1985], a basic shortcoming of it is that it fails to capture the intuitive meaning, even in the case of definite programs. A notorious example is the transitive closure program. The unique intended interpretation of this program is its least Herbrand model. However, the completed theory can have also other models. In fact, every fixpoint of the van Emden-Kowalski operator is a model of the completion.

The *canonical model*, *standard* or *preferred* model approach is the second major research direction. The idea is to select one model among the Herbrand models as the *intended model*. The justification for the chosen model is typically based on the appeal to common sense, i.e. on what the reader naturally expects to be the meaning of the program. The approach was initiated by Reiter [1978] for definite programs. Later, the canonical model approach was extended to larger classes of programs. It started with work on the perfect model semantics for stratified programs [Apt et al. 1988; Van Gelder 1988], which was extended to locally stratified [Przymusinski 1988] and weakly stratified [Przymusinska and Przymusinski 1990] programs. This direction culminated in the well-founded semantics which defines a unique (possibly 3-valued) model for all normal programs [Van Gelder et al. 1991].

A third major direction was motivated by the research in Non-monotonic Reasoning. The idea was introduced by Gelfond [1987], who proposed to interpret *failure to prove* literals *not p* as epistemic literals *I do not know p* and represented them by the modal literal $\neg Kp$ in auto-epistemic logic (AEL) [Moore 1985]. In this embedding, a logic programming rule:

```
p :-  q, not r
```

is interpreted as the following AEL formula:

$$p \leftarrow q \wedge \neg Kr$$

Marek and Truszczyński [1989] proposed a similar embedding in Default Logic [Reiter 1980] which maps the above rule to the default:

$$\frac{q : \neg r}{p}$$

In this view, Logic Programming is seen as a restricted form of default logic or auto-epistemic logic. This approach resulted in stable semantics of logic programs

[Gelfond and Lifschitz 1988] and was the foundation for Answer Set Programming [Niemelä 1999; Marek and Truszczyński 1999].

It is easy to see that the above approaches are based on different common sense interpretations of Logic Programming. Consider for example the definite program $\{\mathtt{p} : -\mathtt{q}\}$. In the completion, stable and well-founded semantics, its unique model is the empty set $\{\}$.

—Under completion semantics, the meaning of the program is given by the theory $\{q \leftrightarrow false, p \leftrightarrow q\}$ which entails the falsity of p and q. The same holds for the canonical model views which all coincide for this program.

—Interpreted as an answer set program, its meaning is given by the unique answer set $\{\}$. Since an answer set is to be interpreted as a first order theory consisting of literals, the meaning of this answer set program is given by the empty first order theory and entails neither $\neg p$, nor $\neg q$, nor even $p \leftarrow q$. This interpretation matches with the embedding of the program in default logic. The unique default extension of the default

$$\frac{q :}{p}$$

is the (deductive closure of the) empty first order logic theory.

This example illustrates that "the" common sense meaning of logic programs does not exist; in fact a number of different intuitions exist. The existence of multiple "common sense" meanings of logic programming is responsible for the complex landscape of Logic Programming semantics. Consequently, *common sense* gives little hope for defining a generally accepted single semantics. In view of this multiplicity of viewpoints, we need to find other, more solid information principles that can serve as an epistemological foundation for Logic Programming.

The goal of this paper is to propose such an alternative epistemological foundation for logic programming. It is not based on a common sense principle but on a solid mathematical information principle[1]. The *thesis* is developed that *logic programming can be understood as a natural and general logic of inductive definitions*. In this view, *logic programs represent definitions; logic programs with recursion represent inductive definitions*. In particular, viewing logic programs as inductive definitions yields a solid justification for the well-founded model as the unique intended model of a logic program. Thus, our work provides an epistemological foundation for the well-founded model as the *canonical* model of a logic program. Moreover, it equips logic programs with an easy to comprehend meaning that corresponds very well with the intuitions of programmers.

The main argument for the thesis comes from the comparison of Logic Programming with studies of inductive definitions in mathematical logic. Such a comparison shows a strong congruence between these studies and Logic Programming at the knowledge theoretical, syntactical, semantical and complexity-theoretical level. In particular, this paper compares definite logic programs with positive and monotone Inductive Definitions, and programs with negation with two approaches for

[1]With the term "information principle" we mean a semantic principle, disconnected from any particular inferential mechanism.

generalized *non-monotone* inductive definitions, Inflationary Inductive Definitions and Iterated Inductive Definitions. Moreover, it is pointed out that there are natural types of inductive definitions that can be represented by logic programs that have no counterpart in mathematical logic studies of inductive definitions. It is argued therefore that Logic Programming under well-founded semantics can make an original knowledge-theoretical contribution to the formal study of inductive definitions and can help to improve our understanding of what non-monotone inductive definitions are.

We believe that appealing to the reading of logic programs as inductive definitions provides a much stronger justification for the intended model than appealing to common sense; it explains why the intended model has a common sense appeal.

Our paper is structured as follows. Sections 2 and 3 offer brief overviews of the syntax and semantics of Logic Programming and of Inductive Definitions. These sections define the necessary background for the main arguments in the text, the comparison of both areas in Section 4. In section 5 we discuss the implications of our view. We conclude in Section 6.

2. A BRIEF OVERVIEW OF LOGIC PROGRAMMING SYNTAX AND SEMANTICS

We assume familiarity with basic syntactical and semantical concepts of classical logic and logic programming [Lloyd 1987]. A logical alphabet Σ consists of variables, constants, function symbols and predicates. The first order logical language based on Σ is the set of all well-formed first order formulas using symbols of Σ. Terms are defined in the usual inductive process from constants and variables of the language by application of function symbols. Atoms are formulas of the form $\mathtt{p}(\mathtt{t_1}, .., \mathtt{t_n})$ where $\mathtt{p}$ is a predicate symbol and $\mathtt{t_1}, \ldots, \mathtt{t_n}$ are terms; literals are atoms or their negation. The Herbrand-universe HU is the set of all ground terms. The Herbrand base HB is the set of all ground atoms.

A *definite rule* is of the form `a :- B` where `a` is an atom and `B` a conjunction of atoms. A *normal rule* can also have negative literals in the body `B`. Note that we use the rule operator `:-` to distinguish rules from classical logic implications. A *definite* (respectively: *normal*) *program* is a set of definite (respectively: normal) rules. A normal program P is called *stratified* [Apt et al. 1988; Van Gelder 1988] if it can be split in a sequence of n_P strata $(P_i)_{0 \leq i < n_P}$ such that for each predicate symbol $\mathtt{p}$, there exists a unique natural number i_p called the level of $\mathtt{p}$ such that for each rule $C = \mathtt{p}(\mathtt{t_1}, .., \mathtt{t_n})\mathtt{:-B} \in P$, it holds that (1) $C \in P$ iff $C \in P_{i_p}$, (2) if predicate symbol $\mathtt{q}$ occurs in a positive literal of `B` then $i_q \leq i_p$ and (3) if predicate symbol $\mathtt{q}$ occurs in a negative literal of `B`, then $i_q < i_p$. P_i is called the i'th stratum of P.

Local stratification generalises the concept by considering the *grounding* of the program: the possibly infinite propositional[2] logic program, denoted $ground(P)$, consisting of all rules that can be obtained by substituting all variables of a rule by ground terms. A normal program P is called *locally stratified* [Przymusinski 1988] if there is a possibly infinite ordinal number n_P and the grounding of P can be split in a sequence of n_P strata $(P_i)_{0 \leq i < n_P}$ such that for each atom $\mathtt{p}$, there exists a unique *ordinal number* i_p called the level of $\mathtt{p}$ such that for each rule

[2]Ground atoms are considered propositions in the corresponding propositional system.

$C = \texttt{p:-B} \in \mathit{ground}(P)$, it holds that (1) $C \in \mathit{ground}(P)$ iff $C \in P_{i_p}$, (2) if atom q occurs in a positive literal of B then $i_q \leq i_p$ and (3) if atom q occurs in a negative literal of B, then $i_q < i_p$.

As usual in logic programming, we will use the grounding of a program P rather than the program itself to provide the meaning of the program[3].

We now give an overview of the semantics of Logic Programming. The theory outlined here is the algebraic approach to Logic Programming semantics based on operators in lattices, mostly due to Fitting [1985], Gelfond and Lifschitz [1988], Przymusinski [1990] and Fitting [1991], Fitting [1993]. To make this paper self-contained we will introduce the main concepts of this approach.

The lattice $\mathcal{TWO} = \{\mathbf{f}, \mathbf{t}\}$ is ordered by the natural ordering $\leq$ with $\mathbf{f} \leq \mathbf{t}$. This defines a complete lattice ordering of $\mathcal{TWO}$. We will also consider the lattice $\mathcal{FOUR}$ that consists of elements $\bot, \top, \mathbf{f}_4, \mathbf{t}_4$. There are two natural lattice orderings in $\mathcal{FOUR}$. Namely the *truth* ordering $\leq_t$ where $\mathbf{f}_4 \leq_t \bot, \mathbf{f}_4 \leq_t \top, \bot \leq_t \mathbf{t}_4, \top \leq_t \mathbf{t}_4$ and the *knowledge* ordering $\leq_k$ where $\bot \leq_k \mathbf{f}_4, \bot \leq_k \mathbf{t}_4, \mathbf{f}_4 \leq_k \top, \mathbf{t}_4 \leq_k \top$. Each element has its inverse $\bot^{-1} = \bot, \top^{-1} = \top, {\mathbf{f}_4}^{-1} = \mathbf{t}_4, {\mathbf{t}_4}^{-1} = \mathbf{f}_4$. $\mathcal{THREE}$ is the restriction of $\mathcal{FOUR}$ to $\bot, \mathbf{f}_4, \mathbf{t}_4$. Note that $\mathbf{t}_4, \mathbf{f}_4$ have no least upperbound with respect to $\leq_k$ in $\mathcal{THREE}$, hence $\leq_k$ is not a complete lattice ordering in $\mathcal{THREE}$.

To define semantics for logic programs we will need to discuss *interpretations*. Those are defined as mappings from the Herbrand base HB of the program into the set of truth values: two-valued interpretations map atoms into $\mathcal{TWO}$, three-valued interpretations into $\mathcal{THREE}$ and four-valued interpretations into $\mathcal{FOUR}$. The orderings $\leq$ in $\mathcal{TWO}$, and $\leq_t$ and $\leq_k$ in $\mathcal{FOUR}$ lift to interpretations. So, for two-valued interpretations, $I \leq J$ holds if $I(a) \leq J(a)$ for each atom a. The orders $\leq$ and $\leq_t$ define complete lattice orderings in the corresponding sets of two-, three- and four-valued interpretations; the order $\leq_k$ is a complete lattice ordering of four-valued interpretations but not of three-valued interpretations.

We will use a slightly different representation for three-valued and four-valued interpretations which will allow to simplify the formalization of the semantics. It is based on the fact that $\mathcal{FOUR}$ can be defined alternatively as the product lattice of $\mathcal{TWO}$. Namely, we can define $\bot = (\mathbf{f}, \mathbf{t}), \top = (\mathbf{t}, \mathbf{f}), \mathbf{f}_4 = (\mathbf{f}, \mathbf{f}), \mathbf{t}_4 = (\mathbf{t}, \mathbf{t})$. In this representation, the orders $\leq_t, \leq_k$, and the inverse in $\mathcal{FOUR}$ are generated by the simple laws: $(v, w) \leq_t (v_1, w_1)$ iff $v \leq v_1$ and $w \leq w_1$; $(v, w) \leq_k (v_1, w_1)$ iff $v \leq v_1$ and $w \geq w_1$; $(v, w)^{-1} = (w^{-1}, v^{-1})$. Note that $\mathcal{THREE}$ is the set of tuples (v, w) such that $v \leq w$.

With this representation in mind, it is easy to see that there is a one-to-one correspondence between three- and four-valued interpretations v and pairs (I, J) of two-valued interpretations, namely, for each symbol p, $v(p) = (I(p), J(p))$. Thus, a four-valued interpretation can also be defined as a pair $\langle I, J\rangle$ of two-valued interpretations. Three-valued interpretations correspond to pairs $\langle I, J\rangle$ such that $I \leq J$. The orders are then defined by: $\langle I, J\rangle \leq_t \langle I_1, J_1\rangle$ iff $I \leq I_1$ and $J \leq J_1$; $\langle I, J\rangle \leq_k \langle I_1, J_1\rangle$ iff $I \leq I_1$ and $J \geq J_1$. If we view the two-valued interpretations I and J in the four-valued interpretation $\langle I, J\rangle$ as sets of true atoms, then the set

[3]Using the grounding of a program boils down to restricting models to Herbrand interpretations. In section 5.4, we briefly discuss the effect of this restriction and the extension of the semantics to general interpretations.

$I \cap J$ identifies the atoms that are true in $\langle I, J\rangle$; the set $HB \setminus (I \cup J)$ defines the false atoms; the set $J \setminus I$ those that are undefined and finally, the set $I \setminus J$ those that are inconsistent.

A useful and natural way of interpreting a three-valued interpretation is as an *approximation* of two-valued interpretations. A three-valued interpretation $\langle I, J\rangle$ approximates every two-valued interpretation I' such that $I \leq I' \leq J$. Here I is an underestimate of I', whereas J is an overestimate. The knowledge ordering of approximations corresponds to the intuition of a tighter, more precise approximation. As we will show, the four-valued semantics of logic programs can be considered as a computation of a sequence of improving approximations until some fixpoint is reached. The sequence is obtained by iterating some operator that takes an approximation and *refines* it by deriving a better approximation consisting of a greater underestimate and lower overestimate. While we are really interested in approximations (i.e. three-valued interpretations), the considerations of the bilattice of four-valued interpretations considerably simplifies the arguments because the set of three-valued interpretations does not form a lattice under the ordering $\leq_k$, whereas the set of four-valued interpretations does.

Now we are ready to discuss the operators in the lattices of interpretations. The first operator is the *immediate consequence operator* T_P defined by van Emden and Kowalski [1976]. There exists three versions of it: the two-valued operator denoted T_P was defined by van Emden and Kowalski [1976]; the three-valued version Φ_P was introduced by Fitting [1985] and the four-valued version was introduced by Fitting [1991] and will be denoted by $\mathcal{T}_P$. These operators can be defined uniformly in the following way. Let I be any two-valued interpretation (respectively three-valued, four-valued interpretation). We define $J = T_P(I)$ (respectively $\Phi_P(I), \mathcal{T}_P(I)$) so that for each atom `a`, J(`a`) is computed in two steps:

(a) Compute the truth value[4] of the body `B` of each clause `a:- B` with respect to I.

(b) Take the maximum[5] of the values computed in point (a). This is J(`a`).

Obviously, $\mathcal{T}_P$ generalises Φ_P, which in turn generalises T_P; so, for a three-valued interpretation I, $\mathcal{T}_P(I) = \Phi_P(I)$.

The operator T_P is $\leq$-monotone if P is a definite program, but not in general. The Knaster-Tarski theorem [Tarski 1955] asserts that every monotone operator in a complete lattice possesses a fixpoint, that the fixpoints themselves form a complete lattice, and that the least fixpoint can be computed by iteration of the operator starting at the least element of the lattice. Thus in case of a definite program P, T_P has a unique least fixpoint called the least Herbrand model [van Emden and Kowalski 1976]. But in the general case of normal programs, there is no guarantee that T_P has a least fixpoint; it may have no fixpoint at all or multiple minimal ones. Fixpoints of T_P have been identified as Herbrand models of the completion of P [Apt and Emden 1982].

[4]In the case of three- and four-valued interpretations, the truth value of `B` is the $\leq_t$-minimum of the truth values of the literals in `B`.

[5]In the case of three- and four-valued interpretations, the maximum with respect to $\leq_t$ is to be computed.

The operators Φ_P and $\mathcal{T}_P$ can be defined equivalently on pairs of two-valued interpretations. It can be easily verified that $\mathcal{T}_P$ maps a pair $\langle I, J\rangle$ to a pair $\langle I_1, J_1\rangle$ such that for each atom $\mathtt{a} \in HB$, $\mathtt{a}$ is true in I_1 if a rule `a :- B` can be found such that each positive literal in `B` is true in I and each negative literal in `B` is true in J; `a` is true in J_1 if a rule `a :- B` can be found such that all positive literals of `B` are true with respect to J and all negative literals of `B` are true with respect to I. This observation is interesting for two reasons. First, it defines the three- and four-valued operators in terms of standard 2-valued truth arithmetic. Second, it illuminates the way $\mathcal{T}_P$ computes new approximations. Assume that we have obtained a pair $\langle I, J\rangle$ that approximates some intended but so far unknown interpretation I'. The operator $\mathcal{T}_P$ produces a new approximation $\langle I_1, J_1\rangle$ of I': I_1 is obtained by underestimating all literals in the bodies of rules, hence by evaluating positive literals with respect to I and negative literals with respect to J. I_2 is obtained by overestimating all literals. Thus, I_1 and J_1 provide a new under- and overestimate.

Given this intuition, it is now easy to see that better approximations produce (via $\mathcal{T}_P$) yet better approximations. That is, if $\langle I, J\rangle \leq_k \langle I', J'\rangle$ then $\mathcal{T}_P(\langle I, J\rangle) \leq_k \mathcal{T}_P(\langle I', J'\rangle)$. In other words $\mathcal{T}_P$ is $\leq_k$-monotone. Moreover, for approximations $\langle I, J\rangle$ (that is when $I \leq J$), the value $\mathcal{T}_P(\langle I, J\rangle)$ is also an approximation. The effect of the first of these two facts is that the Knaster-Tarski theorem is applicable and so the operator $\mathcal{T}_P$ possesses a least fixpoint (in the ordering $\leq_k$). The second fact implies that the least fixpoint computation generates more and more precise approximations and the fixpoint is also an approximation. In other words, the least fixpoint has no inconsistent atoms. Hence, it is also the least fixpoint of the three-valued immediate consequence operator. Since a similar construction has been applied by Kleene in his fixpoint theorem for partial recursive functions and by Kripke in his famous paper on truth, the least fixpoint of the operator $\mathcal{T}_P$ is often called the *Kripke-Kleene* fixpoint [Fitting 1985].

We will now discuss the other two operators important for our investigations.

Let I be a two-valued interpretation. The *Gelfond-Lifschitz* reduct, P^I of the propositional program P is obtained in two steps.

(a) We eliminate from P all clauses C such that the body of C contains a literal $\neg a$ false in I (i.e. a is true in I);

(b) in the remaining clauses, we eliminate all negative literals in the bodies. Note that negative literals in these clauses are true in I.

The program P^I is a definite program and so it possesses a least model N. This interpretation N is the value of the two-valued *Gelfond-Lifschitz* operator GL_P on I [Gelfond and Lifschitz 1988].

The larger the I, the less clauses are being left in P^I, and so it follows that the operator GL_P is *anti-monotonic*, that is $I_1 \leq I_2$ implies $GL_P(I_2) \leq GL_P(I_1)$. The fixpoints of GL_P, if they exist, are called the *stable* models of P [Gelfond and Lifschitz 1988]. When P is a program with variables, the stable models of P are stable models of the grounding of P.

The Gelfond-Lifschitz operator has been generalised to three-valued interpretations by Przymusinski [1990] and to four-valued interpretations by Fitting [2001]. Here we provide a simplified but equivalent definition of these operators presented

in [Denecker et al. 2000]. The four-valued operator $\mathcal{GL}_P$ is defined on the bilattice of four-valued interpretations as follows:

$$\mathcal{GL}_P(\langle I, J\rangle) = \langle GL_P(J), GL_P(I)\rangle.$$

As was the case with $\mathcal{T}_P$, also the operator $\mathcal{GL}_P$ can be understood as an operator for refining approximations. Assume that we obtained an approximation $\langle I, J\rangle$ of the intended but unknown interpretation I' such that $I \leq I' \leq J$. A common intuition about I' is that true atoms in I' should be supported, that is they should be provable from the false atoms in I'. This intuition indicates how to revise the approximation $\langle I, J\rangle$. The new overestimate for I' is computed by fixing the truth values of the negative literals in the bodies of P by some safe overestimation, and then performing a fixpoint computation using the resulting definite program. A safe overestimate of the negative literals *not* p is given by interpreting p by the current underestimate I. Analogously, the new underestimate is obtained by fixing the truth values of the negative literals *not* p in rules by a safe underestimation, and performing the fixpoint computation. A safe underestimation of *not* p is obtained by interpreting p by the current overestimation J.

The anti-monotonicity of the operator GL_P implies two important properties of $\mathcal{GL}_P$ analogous to those of the operator $\mathcal{T}_P$. First, the operator $\mathcal{GL}_P$ is monotone with respect to the ordering $\leq_k$. Second (which, in fact was outlined above, when we discussed the intuition for $\mathcal{GL}_P$), $\mathcal{GL}_P$ maps approximations to approximations. Consequently, just like in case of the operator $\mathcal{T}_P$, we find that the Knaster-Tarski theorem is applicable in case of $\mathcal{GL}_P$, and so $\mathcal{GL}_P$ possesses a $\leq_k$-least fixpoint. Moreover, this least fixpoint is an approximation (as defined above).

The least fixpoint of $\mathcal{GL}_P$ is called the *well-founded* model of P. The well-founded model happens to be a fixpoint of the operator $\mathcal{T}_P$. Consequently, the well founded model is $\leq_k$-greater than the Kripke-Kleene fixpoint.

The well-founded model was originally defined by Van Gelder, Ross, and Schlipf [1991] using a different construction. Its characterization as the least fixpoint of the three-valued Gelfond-Lifshitz operator is due to Przymusinski [1990]. The underlying algebraic structure of the product lattice of interpretations, the role of the four-valued generalization of the van Emden-Kowalski operator T_P and of the algebraic structure of the three-valued and four-valued versions of the Gelfond-Lifschitz operator have been presented by Fitting [1993], [Fitting 2001] and [Denecker et al. 2000].

3. A BRIEF OVERVIEW OF INDUCTIVE DEFINITIONS IN MATHEMATICAL LOGIC

3.1 Monotone induction

The study of induction can be defined as the investigation of a class of effective construction techniques in mathematics. There, sets are frequently defined through a constructive process of iterating some recursive *recipe* that adds new elements to the set given that one has established the presence or absence of other elements in the set. Such a recipe corresponds naturally to an operator on sets (mapping any set S to the set obtained by applying the recipe to elements of S). The set defined by the inductive definition can be obtained through some iterated application of this operator until a fixpoint is reached. Consequently the study of inductive definitions is closely related to the study of operators and their fixpoints [Aczel 1977].

Originally, mathematical logicians focused on *monotone* inductive definitions. When an operator Γ is monotone (i.e. $R \subseteq R'$ implies $\Gamma(R) \subseteq \Gamma(R')$), it follows from the Knaster-Tarski theorem that Γ possesses a least fixpoint. This set can be characterized either in a non-constructive way as the intersection of all sets that are closed under Γ (i.e. $\Gamma(S) \subseteq S$) or in a constructive way as the limit of the increasing sequence obtained by iterated applications of Γ. For this reason, Tarski's least fixpoint theory of monotone operators [Tarski 1955] can be considered as the algebraic theory of monotone induction.

Applications of monotone induction are frequent in mathematics. Typical examples are sets closed under some operation. For instance, the subgroup $G(B)$ generated by a set B of elements in a group $\langle G, ., (\cdot)^{-1}\rangle$ is defined as the least subset $S \subseteq G$ such that $B \subseteq S$ and for each $x, y \in S : x.y^{-1} \in S$. Other examples are the definitions of terms and formulas of logic, or the deductive closure $Cn(T)$ of a logic theory T — the least set of formulas containing T and closed under application of all inference rules of logic.

Let $\mathcal{L}$ be a language of predicate calculus with predicate symbols $p_1, \ldots, p_k$ and one additional n-ary relational symbol p. Let $\varphi[x_1, \ldots, x_n]$ be a formula of $\mathcal{L}$ with n free variables $x_1, \ldots, x_n$. Let us fix an interpretation of the symbols $p_1, \ldots, p_k$ by $\langle A, R_1, \ldots, R_k\rangle$. Here A fixes the domain and interpretation of constant and function symbols and R_i is an n-ary relation interpreting p_i. We say that φ is *monotone* (in relational symbol p) if for all interpretations S_1 and S_2 of symbol p, such that $S_1 \subseteq S_2$ and for all tuples of domain elements $d_1, \ldots, d_n$

$$\langle A, R_1, \ldots, R_k, S_1\rangle \models \varphi[d_1, \ldots, d_n] \text{ implies } \langle A, R_1, \ldots, R_k, S_2\rangle \models \varphi[d_1, \ldots, d_n].$$

Given such interpretation $\langle A, R_1, \ldots, R_k\rangle$ and the monotone formula φ, we can define an operator $\Gamma_\varphi : A^n \to A^n$ by

$$\Gamma_\varphi(R) = \{\langle d_1, \ldots, d_n\rangle : \langle A, R_1, \ldots, R_k, R\rangle \models \varphi[d_1, \ldots, d_n]\}.$$

The operator Γ_φ is monotone, thus it possesses a least fixpoint S. The fixpoint S possesses the property:

$$\{\langle d_1, \ldots, d_n\rangle : \langle A, R_1, \ldots, R_k, S\rangle \models \varphi[d_1, \ldots, d_n]\} = S.$$

By the Knaster-Tarski theorem, $S = \Gamma_\varphi^\beta(\emptyset)$ for a least ordinal β. The sets Γ_φ^α for $\alpha \leq \beta$ are called *levels*. The ordinal β is called the *length* of the recursion.

The Knaster-Tarski theorem tells us that when we deal with a *monotone* inductive definition then a highly non-constructive definition of a fixpoint (defined as the intersection of a large, possibly non-denumerable, family of sets) can be turned into a constructive one (iterate the operator until the fixpoint is reached; when the universe is denumerable, the fixpoint will be reached at a denumerable ordinal). The logical theory of inductive sets in mathematical logic studies the complexity of sets that are inductively definable, the complexity of levels, and the length of the process to reach the fixpoint. Pioneering work in this area was done by Kleene and Spector.

Spector [1961] discussed the question of monotone inductive definability of sets of integers and sets of elements of the Baire space N^N (that is sets of number-theoretic functions). Spector announced that recursively enumerable sets are precisely those definable by positive existential inductive definitions (that is positive formulas with-

out universal quantifiers). Moreover the length of induction is at most ω, that is the fixpoint is reached in ω steps. Even earlier, Kleene [1955] studied so-called Π^1_1 sets of natural numbers[6]. Spector [1961] noticed that Kleene's results imply that all Π^1_1 sets are one-to-one reducible to a set defined inductively by a Π^0_1 positive inductive definition, that is a positive inductive definition where the defining formula φ is of the form $\forall n\psi$ and ψ does not contain quantifiers. The length of induction is, however, ω_1^{CK} where ω_1^{CK} is the least ordinal that is not the type of a recursive well-ordering. Even the Π^1_1 positive inductive definitions (that is those with the formula φ being Π^1_1) do not increase the complexity of the fixpoint; it is still Π^1_1. Spector found the exact bounds on the complexity classes of the levels of monotone inductive definitions. An abstract version of Spector's results is given in Aczel [1977].

The fundamental study of the *abstract* version of the results of Kleene and Spector was performed by Moschovakis [1974] (see also [Aczel 1977], [Barwise 1977]).

3.2 Extensions for non-monotone induction

In mathematical logic, there exists two very different extensions of the above framework to deal with non-monotonic forms of induction.

Moschovakis [1974] considered a scheme where formulas φ are not necessarily monotone. Operators associated to such definitions are non-monotone and may have no fixpoints or multiple minimal fixpoints. To avoid this problem he modified the definition of level, by adding the previously defined level, that is by setting

$$S_{\alpha+1} = S_\alpha \cup \Gamma_\varphi(S_\alpha)$$

(and $S_\lambda = \bigcup_{\alpha<\lambda} S_\alpha$ for limit λ).

It is easy to see that the sequence of levels is increasing and has a limit S_φ which we call the *inflationary fixpoint* defined by φ. Note that $S_\varphi = S_\varphi \cup \Gamma_\varphi(S_\varphi)$ or equivalently $\Gamma_\varphi(S_\varphi) \subseteq S_\varphi$. In other words, S_φ is a *pre-fixpoint* of Γ_φ. In general, it is not a fixpoint of Γ_φ, and it is not even a minimal pre-fixpoint. As an example, consider the predicate $p(x)$ defined by the formula $x = a \wedge \neg p(b) \vee x = b \wedge \neg p(a)$. The set defined by this formula is $\{a, b\}$. It is not a fixpoint and is strictly larger than both fixpoints ($\{a\}$ and $\{b\}$) of this formula.

Moschovakis called this type of definitions *non-monotone inductive definitions*. Later, Gurevich and Shelah [1986] called them *inflationary* inductive definitions.

A very different account of non-monotone induction is found in Iterated Inductive Definitions (IID's). These were first introduced in [Kreisel 1963] and later studied in [Feferman 1970; Martin-Löf 1971; Buchholz et al. 1981]. Aczel [1977] formulates the intuition of Iterated Inductive Definitions in the following way. Given a mathematical structure M_0 fixing the interpretation of the function symbols and some set of interpreted predicates, a positive or monotone inductive definition defines one or more new predicates in terms of M_0. The definition of these new predicates may depend positively or negatively on the interpreted predicates. Once the interpretation of the defined symbols p is fixed, M_0 can be extended with these interpretations, yielding a new interpretation M_1. On top of this structure, again

[6]These are sets of integers definable as $\{m : \forall f \exists n\ R(m, n, f(n))\}$ where R is a recursive relation (quantifier $\forall f$ ranges over all number-theoretic functions, that is elements of the Baire space N^N).

new predicates may be defined in the similar way as before. The definition of these new predicates may now depend positively or negatively on the defined predicates p as their interpretation is now fixed by M_1. This modular principle can be iterated arbitrarily many times, yielding a possibly transfinite sequence of positive inductive definitions.

Though the intuition is simple, it is not straightforward to see how this idea is implemented in IID-approaches. [Feferman 1970; Buchholz et al. 1981] investigate IID's encoded in an IID-form, a single FOL formula of the form $F[n, x, Q, P]$[7], and expresses its semantics in a circumscription-like second order formula. The problem is that this encoding is extremely tedious and blurs the simple intuitions behind this work. For more details on the encoding of iterated inductive definitions, we refer the reader to [Denecker 1998].

Inflationary inductive definitions and Iterated Inductive definitions are not equivalent and are based on very different intuitions. At present, there is no standard well-motivated treatment of non-monotone inductive definitions.

3.3 Discussion of non-monotone induction

When evaluating the two different approaches to non-monotone induction, the question arises which of them has an empirical basis in mathematical practice. The specific question is: can we find inductive constructions in mathematics that use a *recipe* that adds new elements to the constructed relations based on the established *absence* of other elements in the relation (and hence modeling a non-monotone operator)? And if such applications can be found, do such constructions correspond to inflationary induction or to iterated induction, or can we find both types?

Non-monotone iterated induction occurs frequently in the context of *inductive definitions over a well-founded set*. A well-founded set is a partial order without infinite descending chains $x_0 > x_1 > x_2 > \ldots$. Equivalently, it is a partial order such that each subset contains a minimal element. Such orderings are also called Noetherian orderings. Inductive definitions of this kind describe the membership of an element, say a, of the defined predicate X in this domain in terms of the presence (or absence) of strictly smaller elements in the defined predicate. Thus, to check if an element a belongs to X we need to check some properties of *predecessors* of a. By applying this definition recipe to the minimal elements and then iterating it for higher levels, the defined predicate can be completely constructed. Consequently, this type of definition correctly and fully defines a predicate, even when it is non-monotone.

The following example illustrates this principle. We can define an even natural number n by induction on the natural numbers:

—$n = 0$ is even;

—if n is not even then $n + 1$ is even; otherwise $n + 1$ is not even.

[7]The meaning of the variables is as follows: x is a candidate element of the set defined by the IID, n is an ordinal number, Q is a set of elements and P a set of elements defined in the n^{th} stratum. Roughly speaking, $F[n, x, Q, P]$ is true when x belongs to the n^{th} stratum and x can be obtained from the set P and the restriction of the set Q to the elements defined in strata $m < n$ by an application of a rule of the n^{th} stratum.

Representing this definition in the same style as monotone inductions yields the following non-monotone formula defining *even* in the language of arithmetic:

$$x = 0 \vee \exists y.x = s(y) \wedge \neg even(y) \tag{1}$$

It turns out that the set of even numbers is the unique fixpoint of the operator associated to this formula. Equivalently, the set of even numbers is correctly characterized with respect to the natural numbers by the following recursive iff-definition:

$$\forall x.even(x) \leftrightarrow x = 0 \vee \exists y.x = s(y) \wedge \neg even(y)$$

It is easy to see that the inflationary approach applied on this formula does not yield the intended set of even numbers. Indeed, applying the operator to the empty set produces the set of all natural numbers, which is necessarily the inflationary fixpoint.

On the other hand, it is natural to consider this definition as an iterated inductive definition if we split up the definition in a sequence of definitions compatible with the order on the natural numbers. The following list depicts the splitting of the definition in small definitions each defining a single atom:

$$\begin{array}{lll} (0) & even(0) := & true. \\ (1) & even(1) := & \neg even(0) \\ (2) & even(2) := & \neg even(1) \\ & & \ldots \\ (n+1) & even(n+1) := & \neg even(n) \\ & & \ldots \end{array}$$

It is clear, at least intuitively, that the iterated induction correctly constructs the predicate *even*.

Although intuitively correct, it is unfortunate that in the IID approach this inductive definition cannot be encoded by the simple formula (1) defined above. Instead, it must be encoded by a rather complex formula in which the level of the defined atoms are explicitly encoded. This formula is[8]:

$$n = 0 \wedge x = 0 \vee \exists m.(n = s(m) \wedge x = s(m) \wedge \neg Q(m)) \tag{2}$$

For more details on this we refer to [Buchholz et al. 1981] and [Denecker 1998].

Notice that induction over a well-founded order is frequently non-monotone. A common example is that of *rank* of an element in a well-founded order $\langle P, \preceq \rangle$. The rank of an element x of P is defined by transfinite induction as the least ordinal which is a strict upper-bound of the ranks of elements $y \in P$ such that $y \prec x$.

Formally, let $F[x, n]$ denote the following formula expressing that n is a larger ordinal than the ranks of all elements $y \prec x$:

$$\forall y, n'.(y \prec x \wedge rank(y, n') \rightarrow n' < n)$$

Intuitively, $rank(x, n_x)$ is represented by the following formula:

$$F[x, n_x] \wedge \forall n.(F[x, n] \rightarrow n_x \leq n)$$

[8]Note that there is no positive induction involved, so only the variable Q representing the elements of lower strata occurs (see the discussion in footnote 7).

Note that *rank* occurs negatively in this definition (it occurs as a condition in the implication in the first conjunct) and that the associated operator is non-monotone. As in the case of *even*, the meaning of this definition cannot be obtained via inflationary induction[9]. Instead, iterated induction is required. In the context of a well-founded structure, *rank* is also correctly described by the corresponding iff-definition:

$$\forall x, n_x.rank(x, n_x) \leftrightarrow F[x, n_x] \wedge (\forall n.F[x, n] \rightarrow n_x \leq n)$$

Possibly the most important application of this form of induction is the definition of the *levels* of a monotone operator in Tarski's least fixpoint theory. Given an operator O in a complete lattice $\langle L, \bot, \top, \leq \rangle$, Tarski defines the levels of the operator O by transfinite induction:

—$O^0 = \bot$
—$O^{\alpha+1} = O(O^\alpha)$
—$O^\alpha = lub(\{O^{\alpha'} | \alpha' < \alpha\})$ if α is a limit ordinal.

This induction defines a function mapping ordinal numbers from some pre-selected segment γ of ordinals to the lattice L. The function is defined by transfinite iterated induction in the well-founded order of the segment of ordinals. It is of interest to see if this definition is monotone or non-monotone. It is not straightforward to see this due to the use of the functional notation and of the higher order *lub* function. Therefore, consider the following reformulation using a predicate notation. We introduce the binary predicate $level_O$ such that $level_O(\alpha, x)$ iff $O^\alpha = x$. Using this notation, one could represent the above inductive definition by the following formula:

$$\begin{cases} \alpha = 0 \wedge x = \bot \vee \\ \exists \alpha', y.(\alpha = \alpha' + 1 \wedge x = O(y) \wedge level_O(\alpha', y)) \vee \\ limit(\alpha) \wedge x = lub(\{y | \exists \beta < \alpha : level_O(\beta, y)\}) \end{cases}$$

The operator Γ associated to this definition is an operator on the power-set of the cartesian product $\gamma \times L$. It is easy to see that even if O is monotone, Γ is non-monotone. This is due to the rule describing the predicate $level_O$ at limit ordinals and the fact that *lub* has a non-monotone behaviour with respect to $\subseteq$. Indeed, take some limit ordinal α and two sets $S \subseteq S' \subseteq \gamma \times L$. If $(\alpha, x) \in \Gamma(S)$, then x is the *lub* of the set $\{y | \exists \beta < \alpha : (\beta, y) \in S\}$, but in general not of the set $\{y | \exists \beta < \alpha : (\beta, y) \in S'\}$. Hence (α, x) does not in general belong to $\Gamma(S')$.

The above definition still contains the higher order function *lub* but can be expressed as a first order inductive definition which (necessarily) contains negative occurrences of the defined predicate $level_O$:

$$\begin{cases} \alpha = 0 \wedge x = \bot \vee \\ \exists \alpha', y.(\alpha = \alpha' + 1 \wedge x = O(y) \wedge level_O(\alpha', y)) \vee \\ \left(\begin{array}{ll} limit(\alpha) \wedge & (1) \\ \forall \beta, v.(\beta < \alpha \wedge level_O(\beta, v) \rightarrow x \geq v) \wedge & (2) \\ \forall z.(\forall \beta, v.(\beta < \alpha \wedge level_O(\beta, v) \rightarrow z \geq v)) \rightarrow x \leq z & (3) \end{array} \right) \end{cases}$$

[9]The inflationary fixpoint assigns rank 0 to all elements.

In this formula, (2) expresses that x is an upper bound; (3) expresses that x is less or equal than upperbounds. Note that $level_O$ has a negative occurrence in (2).

Induction on a well-founded order defines elements in terms of strictly earlier elements. This excludes that such definitions contain positive (or negative) loops. Iterated induction generalizes this form of induction by allowing positive loops. An example illustrating this principle is the definition of a *stable theory* [Moore 1985]. A stable theory extends the notion of closure $Cn(T)$ of a first order theory T and represents the known formulas of a first order theory T expressible in the language of modal logic. It can be defined through the following fixpoint expression:

$$S = Cn(T \cup \{KF : F \in S\} \cup \{\neg KF : F \notin S\})$$

Alternatively, Marek [1989] gives a definition by iterated induction, based on the standard inference rules and two additional inference rules:

$$\frac{\vdash F}{KF} \qquad \frac{\nvdash F}{\neg KF}$$

The first expresses that if we can infer F, then we can infer KF; the second that if we cannot infer F, then we can infer $\neg KF$. Note that the second rule is non-monotone. The iterated induction proceeds as follows: first $Cn(T)$ is computed, using the classical inference rules on first order formulas; next the two new inference rules are applied, and the extended set is again closed for all modal formulas without a nested modal operator. This can be iterated for formulas of increasing nesting of modal operators until a fixpoint is reached in ω steps. This process constructs the unique stable theory of T.

Notice that the iterated inductive definition of a stable theory is *not* a definition in a well-founded set. Indeed, for any pair of logically equivalent formulas ϕ and ψ, there is a sequence of inference steps leading from ψ to ϕ and vice versa. Hence, ψ belongs to the stable theory if ϕ belongs to it and vice versa. Hence, formulas and inference rules cannot be well-ordered in a way that the conditions of all inference rules are strictly less than the derived formula. This definition is a simple example of an *inductive definition in a well-founded semi-order*[10] $\leq$ in which membership of a domain element a in a defined relation X is defined in terms of the *presence* of domain elements $b \leq a$ in X and in terms of *absence* of domain elements $b < a$ in X.

It is certainly much easier to find applications of iterated induction than of inflationary induction. The applications of inflationary induction, e.g. in [Moschovakis 1974], tend to be for defining highly abstract concepts in set theory. Although inflationary induction is expressive [Moschovakis 1974; Kolaitis and Papadimitriou 1991; Gurevich and Shelah 1986], it turns out to be very difficult to use it to encode even simple concepts. This is illustrated by Van Gelder [1993] with a discussion of the definition of the complement of the transitive closure of a graph. This concept can be defined easily by an iterated definition with 2 levels: at the first level, the transitive closure is defined; at the second level, the complement is defined as the

[10]This concept extends well-founded order. A semi-order $\leq$ is a reflexive and transitive relation. Let $x < y$ denote that $x \leq y$ and $y \not\leq x$. Then $\leq$ is a well-founded semi-order if there is no infinite descending chain $x_0 > x_1 > x_2 > ...$

negation of the transitive closure. On the other hand, it was considered as a significant achievement when a (function-free) solution was found using inflationary induction. Van Gelder [1993] adds: "Presumably, in a practical language, we do not want expression of such simple concepts to be significant achievements!".

The cause for this may lay in the weakness of the characterization of the inflationary fixpoint. A positive feature of inflationary semantics is its simple and elegant mathematics. A negative property is that the set characterized by inflationary induction, though unique, apparently has rather weak mathematical properties. The inflationary fixpoint is not a fixpoint of the semantic operator of the definition, only a pre-fixpoint and not even a minimal one. The property of being just a pre-fixpoint seems too weak to be useful. Notice that in all above applications of non-monotone induction, the intended sets are fixpoints of the operator of the inductive definition.

Let us summarize this discussion. Which form of non-monotone induction has an epistemological foundation in mathematical practice? In the case of inflationary induction, while we don't exclude that it exists, we are not aware of it. For Iterated Induction, we showed that such a basis exists. However, the current logics of iterated induction impose an awkward syntax which makes them unsuitable for practical use. To their defence, we must say that IID's were never intended for practical use but rather for constructive analysis of mathematics. But it is a natural and modular principle. As will be argued below, logic programming builds on the same principle and, from an epistemological point of view, contributes by offering a more general and much more elegant formalization of this principle.

4. INDUCTIVE DEFINITIONS AS AN EPISTEMOLOGICAL FOUNDATION FOR LOGIC PROGRAMMING

4.1 Definite programs - monotone induction

The relationship between logic programs and inductive definitions is already apparent in many standard prototypical logic programming examples. Recall the following programs:

```
list([]).
list([X|Y]) :- list(Y).

member(X,[X|T]).
member(X,[Y|T]):- member(X,T).

append([],T,T).
append([X|Y],T,[X|T1]):-append(Y,T,T1).

sorted_list([]).
sorted_list([X]).
sorted_list([X,Y|Z]):-X<Y,sorted_list([Y|Z]).

arc(a,a).
arc(b,c).
connected(X,Y) :- arc(X,Y).
connected(X,Y) :- arc(X,Z), connected(Z,Y).
```

These programs are natural representations of inductive definitions of the concepts. Interpreting them as inductive definitions provides a justification for deducing that the atoms `member(a,[b,c])`, `append([a,b],[c,d],[a,d])` as well as `sorted_list([1,3,2])` are false, facts which could not be justified by interpreting these programs as Horn theories. Indeed, only positive facts can be deduced from a Horn theory.

At the syntactical level, there is a close relationship between the way inductive definitions are represented in logic programs and in mathematical logic. In particular, the mathematical logic form corresponds exactly to the right hand side of the completed definition of the predicate. For example, the *completed definition* [Clark 1978] of the `member`-program is:

$$\forall x, y.member(x, y) \leftrightarrow \exists z.y = [x|z] \vee \exists z, t.y = [z|t] \wedge member(x, t)$$

The right hand side of the equivalence is the formula that inductively defines the `member` relation. Thus, (finite) definite logic programs correspond to (a subclass of) positive existential inductive definitions.

Also at the semantical level, there is congruence between semantical methods in mathematical logic and in Logic Programming. Aczel [1977] gives an overview of three equivalent mathematical principles for describing the semantics of a (positive) inductive definition. They are equivalent with the way the least Herbrand model semantics of definite logic programs can be defined:

—the least set or least Herbrand model definition.

—the least fixpoint characterization.

—The model can be expressed also as the interpretation in which each atom has a *proof tree*[11]. Also this formalization has been used in Logic Programming, e.g. in [Denecker and De Schreye 1993].

The Logic Programming community devoted considerable attention to the study of the complexity and expressivity issues of definite logic programs. Not surprisingly, the results thus obtained resemble those found by Spector. Andréka and Németi [1978] found that definite (Horn) programs compute the same sets as positive existential inductive definitions, i.e. recursively enumerable sets (for the case of Herbrand interpretations, this result has been established already by Smullyan [1968]). That is, for a given recursively enumerable set S there is a normal program P_S such that the language of P_S contains a predicate $sol/1$ and a function symbol $s/1$ and $S = \{n : sol(s^n(0)) \in M_P\}$ where M_P is the least Herbrand model of P_S. In other words, the *least* fixpoint of the operator T_P allows for the computation of all recursively enumerable sets. But the converse is also true - sets computed by definite programs are recursively enumerable.

4.2 Stratified programs - iterated inductive definitions

Consider now the following examples of stratified or locally stratified logic programs:

```
% Using list/1, sorted_list/1
unsorted_list(L):-list(L), not sorted_list(L).
```

[11] We will discuss those in detail below.

```
% Using connected/2
disconnected(X,Y):-node(X), node(Y), not connected(X,Y).

% Using  person/1, man/1
woman(X):-person(X),not man(X).

even(0).
even(s(X)):- not even(X).
```

These are clearly examples of iterated definitions. There is an obvious correspondence between (locally) stratified logic programs under perfect model semantics [Apt et al. 1988; Van Gelder 1988; Przymusinski 1988] and Iterated Inductive Definitions.

Let P be a stratified (or locally stratified) program with stratification $(P_i)_{0\leq i<n_P}$. Let D_i be the set of all symbols that are defined in P_i. Then the perfect model of P is the union M_{n_P} of the sequence of Herbrand models $(M_i)_{1\leq i\leq n_P}$:

—M_1 is the least Herbrand model of P_0;

—M_{n+1} is the least Herbrand model of P_n such that the restriction of M_{n+1} to the symbols in $\bigcup_{i\leq n} D_n$ is M_n.

—In case when n_P is infinite, for a limit ordinal λ, M_λ is the union of the increasing sequence $(M_i)_{1\leq i<\lambda}$.

Though at the intuitive and semantical level, (locally) stratified logic programming and iterated inductive definition formalisms are analogous, there are substantial differences at the level of the syntactical sugar (and thus in their availability for programming). In the IID formalisms, a possibly transfinite sequence of positive inductive definitions is encoded in one (often quite complex) finite iterated induction formula. As the above examples, in particular the *even* program, illustrate (locally) stratified logic programs offer a much more simple and elegant syntax to represent inductive definitions. Yet, as will be argued in the next section, also this formalism imposes severe disadvantages.

The expressivity of the class of stratified programs has been studied by Apt and Blair [1990]. Specifically, they have shown that the Andréka-Németi-Smullyan result can be lifted in a very natural way. Namely, the stratified programs with n strata, $n \geq 1$ compute precisely all Σ^0_{n+1} sets in the Kleene-Mostowski hierarchy[12].

Thus the programs with n strata are complete for Σ^0_{n+1} sets of integers, and stratified programs compute *precisely* arithmetic sets. This result was significant for the following reasons. On one hand it pinpointed the expressive power of a natural class of programs. On the other hand it demonstrated that normal programs go

[12]The formulas of the form $\exists k_1 \forall k_2 \exists k_3 ... R$ (with $n-1$ alternations of quantifiers), where R has no quantifiers, are called Σ^0_n -formulas. Sets with the definition of the form $\{n : \varphi(n)\}$ where, φ is a Σ^0_n formula, are called Σ^0_n sets of natural numbers. Notice that in Π^1_1 definitions defined above the quantifier over f was a function-theoretic quantifier. Here there are only number quantifiers. The classification of sets of natural numbers defined by Σ^0_n formulas and dually, by Π^0_n formulas is called the Kleene-Mostowski hierarchy.

beyond the generally accepted class of computable sets[13].

REMARK 4.1. *It is interesting to note that the inflationary fixpoint construction resurfaced in the context of logic programming, more precisely in the context of database investigations of logic programs with negation. Kolaitis and Papadimitriou [1991] advocate the use of the inflationary fixpoint as the semantics of normal programs. It is easy to see that in* none *of the above programs with negation, the inflationary fixpoint corresponds to the perfect model and with what most logic programmers would consider as the intended interpretation. For example, the inflationary fixpoint of* `even` *is the set of natural numbers; that of* `disconnected` *the total binary relation of nodes, that of* `unsorted_list` *the set of all lists, etc..*

4.3 A critique of syntactic stratification

A problem with stratification is that stratifiability of a program or definition is broken even by the most innocent syntactic changes. The following variant of the `even` program illustrates this. Assume that we introduce the predicate `successor/2` to represent the successor relation. In what is essentially an innocent linguistic variant of the even program defined in the previous section, we can write down the following definitions for `successor/2` and `even/1`:

```
successor(s(X),X).
even(0).
even(Y):- successor(Y,X),not even(X).
```

This variant program is not longer locally stratified due to the presence of rule instances of the form:

```
even(m) :- successor(m,m), not even(m).
```

This simple example is just one out of a broad class of simple transformations that transform a stratified logic program into an unstratified logic program. A detailed study of semantics-preserving transformations has been conducted by Brass and Dix [1999] who showed that several classes of semantics can be characterized in these terms.

Another familiar example is the vanilla meta-interpreter [Bowen and Kowalski 1982] which consists of the following rules:

```
demo(true).
demo((P,Q)):-demo(P),demo(Q).
demo(P):-atomic(P),clause(P:-Q),demo(Q).
demo(not P):- not demo(P).
```

This program induces a transformation of a normal program to the vanilla meta-program consisting of the above definition of `demo` augmented with the `clause`

[13] [Blair et al. 1995] generalized the Apt-Blair result for the case of locally stratified programs and the hyperarithmetical hierarchy. Further, Schlipf [1995b] proved that a complete Π^1_1 set can be defined using the well-founded model (this result generalizes [Blair et al. 1995] result mentioned above). A further relationship between the set of all stable models of a normal program and effectively closed subsets of the Baire space has been established by Marek, Nerode, and Remmel [1994]. Finally, Ferry [1994] characterized the family of stable models of a normal program in terms of the inverse-Scott topology of Cantor space.

representation of the program. This transformation transforms *any* normal program into a non-stratifiable program [Martens and De Schreye 1995]. For example, for any atom `p`, the grounding contains the following unstratifiable rules:

```
demo(p):- atomic(p),clause(p:-not p), demo(not p).
demo(not p):- not demo(p).
```

Consequently, syntactical restrictions such as stratification or local stratification are untenable in the sense that they cannot lead to robust formalisms for the representation of inductive definitions. At the same time, the above examples show that also general, syntactically unstratifiable logic programs can still be interpreted as inductive definitions.

However, dropping the stratifiability constraint introduces several problems at the semantical level. In IID and stratified logic programming, the construction of the formal semantics of a definition is strongly based on the explicit stratification. Such base does not longer exist in the unstratified case. Consequently, alternative semantic techniques are needed to characterize the model of a generalized inductive definition. A second problem is that for some logic programs, in particular those with recursion through negation, the interpretation as inductive definitions breaks down. This problem is considered in section 5.1.

4.4 Normal programs - general non-monotone inductive definitions

This section presents and argues the main thesis of this paper, that the well-founded semantics of logic programming [Van Gelder et al. 1991] provides a more general and more robust formalization of the principle of iterated inductive definition that applies beyond the stratified case. Under this semantics, logic programming can be naturally seen as a generalized non-monotone inductive definition logic not suffering from the aforementioned limitations imposed by syntactic stratification. The arguments below are based on and extend the discussion in [Denecker 1998].

First, the well-founded semantics is a conservative extension of the perfect model semantics; the well-founded model of a (locally) stratified program is its perfect model. Second, many transformations of the type illustrated in the previous section which may transform stratified into unstratified programs, preserve the well-founded model — see [Brass and Dix 1999].

The third argument is based on the analogy between the well-founded semantics and the semantic principle used in IID and stratified logic programming. Przymusinski [1989a] showed that each logic program P has a *dynamic* stratification $(P_i)_{0 \leq i < n_P}$ such that the well-founded model can be obtained by an iterated least model construction. In particular, P_i consists of all rules `p :- B` of P such that i is the least ordinal for which `p` is not undefined in the level $i+1$ of $\mathcal{GL}_P$. Then the well-founded fixpoint can be obtained by an iterated process of extending a 3-valued interpretation defining the atoms of level $< i$ by extending it with the least model of P_i.

Below is an alternative attempt to show the deep structural similarities in the way the perfect model and the well-founded model are constructed. The well-founded semantics formalizes the same intuition of iterated induction but implements them in a superior, more robust and syntax independent way. To illustrate this, let us compare the formalizations. A stratified program P can be split up in a (possibly

transfinite) sequence $(P_i)_{0 \leq i < n_P}$ of definitions P_i of a subset D_i of the atoms. If we fix the meaning of the already defined atoms, each P_i is a monotone definition. The perfect model is the limit M_P of the sequence $(M_i)_{0 < i \leq n_P}$ where each M_{i+1} is obtained by applying the positive inductive definition P_i on M_i. Each M_i approximates M_P and gives the correct truth values on all atoms of $\bigcup_{j<i} D_i$. The role of the stratification in this process is to *delay* the use of some part of the definition until enough information is available to safely apply the positive induction principle on that part of the definition.

The same idea could be implemented in a different way, without relying on an explicit syntactical partitioning of the definition. As in perfect model semantics, the model could be obtained as the limit of a sequence of gradually more refined interpretations (monotonically increasing with respect to the knowledge ordering $\leq_k$ defined in section 2). But rather than approximating by 2-valued interpretations of sub-alphabets, partial interpretations can be used; they also define the truth value of a subset of the atoms. Rather than extending at each level i the given interpretation M_i by applying the positive definition P_i, M_i is extended by applying an operator that implements the positive induction principle. This operator takes as input a partial interpretation I representing well-defined truth values for a subset of atoms, and derives an extended partial interpretation defining the truth values of other atoms that can be derived by positive induction. Definition of truth values of atoms for which not enough information is available is delayed.

The key challenge in the above enterprise is to define an operator that embodies the principle of positive induction in the context of definitions with negation. In [Denecker 1998], it is argued that the multi-valued Gelfond-Lifschitz operator $\mathcal{GL}_P(\cdot)$ of section 2 is an answer to this problem. Below we give an alternative definition for this operator based on proof-trees; this formalization shows very clearly the correspondence with positive induction. The definition is restricted to the three-valued case; this suffices for our purposes: the approximations of the well-founded model computed during fixpoint computation are three-valued.

Let P be a ground program. We assume that each atom occurs as the head of a rule, and that each rule has a non-empty body. To obtain this, it suffices to add to the program the rule `p:-f` for each atom `p` with the empty definition and to transform every atomic rule `p.` to the rule `p:-t`. This preprocessing allows for a more uniform treatment.

DEFINITION 4.2. *A proof-tree $\mathcal{T}$ for an atom* `p` *in a normal program P is a tree labeled with literals such that:*

- `p` *is the root of $\mathcal{T}$;*
- *each non-leaf node is an atom* `q`*; its direct descendants are the literals in the body B of some rule* `q:- B` *of P;*
- *each leaf is either* $\mathbf{t}, \mathbf{f}$ *or a negative literal.*
- *there are no infinite branches.*

This definition formalizes the notion of a candidate proof. Note that the leaves of proof-trees of a definite program P are all $\mathbf{t}$ or $\mathbf{f}$. The least model of a definite program can be characterized as the set of all atoms that have a proof-tree without $\mathbf{f}$ among the leaves.

The intuition of the positive induction operator can be expressed as follows.

Assume that we have constructed a partial interpretation I which assigns correct truth values to a subset of atoms as defined by P. We can extend I in the following way. Assume that an atom p has a proof-tree with only true leaves w.r.t. I: either **t** or negative literals `not q` where $I(\mathtt{q}) = \mathbf{f}$. In that case, it is justified to extend I by assigning **t** to p. On the other hand if each proof-tree for p contains a false leaf (either **f** or a negative literal `not q` where $I(\mathtt{q}) = \mathbf{t}$), then it is impossible to prove p no matter how I is further extended; consequently, it is justified to extend I by assigning **f** to p. All other atoms have at least one proof-tree without false leaves and at least one undefined leaf `not q` and no proof-tree with only true leaves; the computation of the truth value of such an atom must be delayed until all leaves of one of its proof trees are known to be true or all proof-trees are known to contain at least one false leaf.

The above intuition is formalized as follows:

DEFINITION 4.3. *The* Positive Induction Operator $\mathcal{PI}_P$ *maps partial interpretations I to I' such that for each atom* p *:*

—$I'(\mathtt{p}) = \mathbf{t}$ *if* p *has a proof-tree with all leaves true in I.*

—$I'(\mathtt{p}) = \mathbf{f}$ *if each proof-tree of* p *has a false leaf in I;*

—$I'(p) = \mathbf{u}$ *otherwise, i.e. if no proof-tree of p has only true leaves but there exists at least one without false leaves.*

It is straightforward to see that $\mathcal{PI}_P$ is monotone (w.r.t. $\leq_k$): indeed, if $I \leq_k J$, each proof tree with only true leafs in I has only true leaves in J; each proof tree with a false leaf in I has a false leaf in J. Consequently, if p is true or false in $\mathcal{PI}_P(I)$, then it has the same truth value in $\mathcal{PI}_P(J)$.

Given a partial interpretation I, $\mathcal{PI}_P$ computes the truth value of all atoms that can be obtained by applying monotone induction starting from I; $\mathcal{PI}_P$ delays the computation of the truth value of all other atoms. Thus, iterating the operator $\mathcal{PI}_P$ corresponds to the process of iterating monotone induction.

PROPOSITION 4.4. *$\mathcal{PI}_P$ coincides with $\mathcal{GL}_P$ on 3-valued interpretations.*

PROOF. We begin by showing that our proposition is true for 2-valued interpretations.

Let I be an arbitrary 2-valued interpretation. First, we show that if an atom p is true in $\mathcal{PI}_P(I)$, then p is true in $GL_P(I)$.

Consider the set S of all proof-trees of the program P with only true leaves in I, with a root false in $GL_P(I)$. We must show that this set is empty. It is straightforward to see that the collection of proof-trees of P is a well-founded order under the subtree relation. That is, each non-empty set of proof-trees contains a minimal element. Consequently, S contains a minimal element $\mathcal{T}$ with a root p false in $GL_P(I)$. At the top level of $\mathcal{T}$, some rule `p :- B` of P is used such that (1) $\mathcal{T}$ comprises a strict subtree without false leaves for each atom q in B and (2) all negative literals in B are true in I. From (1) and the minimality of $\mathcal{T}$ it follows that each q is true in $GL_P(I)$. From (2) it follows that P^I contains the rule `p :- B'` where B′ is obtained from B by eliminating all negative literals. Consequently, applying T_{P^I} on $GL_P(I)$ yields p. This is a contradiction, because $GL_P(I)$ is a fixpoint of T_{P^I}. This proves the $\Rightarrow$.

For the opposite direction, assume that atom p is true in $GL_P(I)$. We construct a proof-tree for p by induction on the levels of the operator T_{P^I}. Assume that for some ordinal α, each atom of level $\beta < \alpha$ has a proof-tree without false leaves. Let p be an atom of level α. Then for some rule `p :- B` of P^I, each atom in B belongs to a level $\beta < \alpha$ and by the induction hypothesis, it has a proof-tree. By construction of P^I, there exists a rule `p :- B'` of P such that B' extends B with negative literals that are true in I. Obviously, this rule and the proof-trees of the atoms in B can be used to construct a proof-tree for p.

Finally, we extend the argument to the general 3-valued interpretations. Given an arbitrary 3-valued interpretation $I = \langle I_1, I_2 \rangle$ (i.e. $I_1 \leq_k I_2$). Recall that $\mathcal{GL}_P(I) = \langle GL_P(I_2), GL_P(I_1) \rangle$, and let $\mathcal{PI}_P(I) = \langle J_1, J_2 \rangle$.

Note that an atom p is true (resp. false) in $\mathcal{PI}_P(I)$ iff it is true in J_1 (resp. false in J_2). Vice versa a literal `not q` is true (resp. false) in I iff it is true in I_2 (resp. false in I_1). Therefore, a proof-tree of P has only true leaves in I iff it has only true leaves in I_2, and has a false leaf in I iff it has a false leaf in I_1. Consequently, $J_1 = \mathcal{PI}_P(I_2)$ and $J_2 = \mathcal{PI}_P(I_1)$. Since $\mathcal{PI}_P$ and GL_P coincide on the 2-valued interpretations, the argument is complete. □

This proposition shows that $\mathcal{GL}_P$ is an operator performing monotone induction, and that the well-founded model is the model obtained by iterating monotone induction. This, together with our discussion of Iterated Inductive Definitions, shows that the well-founded semantics is an alternative formalization of iterated induction.

4.5 Conclusions

The well-founded model of a stratified program coincides with the perfect model and is preserved by transformations that destroy syntactic stratification. Beyond the class of stratified programs, we have pointed to the strong structural resemblances between IID and perfect model construction and the way the well-founded model is constructed. We find essentially the same ingredients:

—Computation by $\leq_k$-monotonically increasing sequence of approximating partial interpretations.

—Delaying computation of truth values of atoms for which no sufficient information is available.

—Deriving truth values by monotone induction.

The superiority of the well-founded model construction lies in the fact that there is no need for an a priori splitting of the program in different levels. The Positive Induction Operator $\mathcal{PI}_P$ looks at the complete program and derives truth values whenever sufficient information is available.

Consequently, we postulate the *thesis* that the well-founded semantics formalizes the principle of non-monotone iterated induction. This thesis is about the relation between a mathematical theory and an empirical reality, in particular the notion of (general) inductive definition as found in mathematics. Such a thesis of course cannot be formally proven; it is a thesis of a similar nature as e.g. Church's thesis.

5. DISCUSSION

5.1 Total definitions

An aspect that we have ignored so far is that for some programs, the well-founded model is partial. Consider for example the following program which is a formalization of the barber's paradox.

```
shaves(b,X) :- citizen(X), not shaves(X,X).
citizen(a).
citizen(b).
...
```

The well-founded model of this program is partial and does not define the truth value of the atom `shaves(b,b)`. The reason is the *recursion through negation*. If `shaves(b,b)` is false then the rule body is true and one can infer that `shaves(b,b)` is true; however the rule by which that atom is inferred is then no more applicable and the support for its truth is lost.

A natural quality criterion for definitions is that they define the truth values of *all* atoms of the defined predicates. This criterion boils down to the requirement that *good* definitions should have a two-valued well-founded model. We call such definitions *total definitions*. When the requirement of stratifiability is dropped, the formalism allows definitions for which this quality criterion does not hold. Partial models point to *bugs* in the definition. The set of undefined atoms identifies exactly the atoms that are ill-defined. For programs with a partial well-founded model, the interpretation as inductive definitions breaks down to some extent.

There seem to be two sensible treatments for definitions that are not total. A rigorous treatment would be to simply consider them as *inconsistent*. In this strict view, we would define that the model of a normal program is the well-founded model if it is total; otherwise the program has no model. The result is a 2-valued logic in which definitions that are not total have no models and entail everything.

The approach to reject partial models and thus to treat non-total programs as *inconsistent* logical theories can be questioned. The problem of such a rigorous position is that it seriously complicates the design of query-answering systems which then not only should compute answers to a query but also check the consistency of the program, that is the fact that the well-founded model is total. The latter is in general an undecidable problem [Schlipf 1995a][14]. Even for programs for which it is feasible to prove that they are total, the cost of doing so could be prohibitive[15]. Moreover, in some complex applications, partial models simply cannot be avoided. An illustration is the theory of truth presented in Fitting [1997]. Fitting uses the well-founded semantics to define the truth predicate and obtains one in which the *liar* paradox ("I am a liar") is undefined ($\bot$) but the *truth sayer* ("I am true") is false.

Hence a more reasonable position is to accept definitions with partial well-founded models. The result is a kind of *paraconsistent* definition logic, i.e. a logic in which definitions with local inconsistencies do not entail every formula. In the context

[14]Schlipf showed that the set of indices of finite programs for which the well-founded semantics is 2-valued forms a Π^1_1-complete set.

[15]But note that showing totality of the program should be done only once, not for every query.

of logics for definitions, 3-valued well-founded semantics offers an answer to an old critique on classical logic, namely that it collapses totally in the case of inconsistency.

5.2 Computational aspects

As discussed in [Apt and Blair 1990], the perfect model and hence the well-founded model is not recursively enumerable for all programs and no effective proof procedure is possible for the general case. However computation is effective for function free logic programs (Datalog). As proven in [Van Gelder 1993], the data complexity of Datalog[16] under the well-founded semantics is polynomial time; this is better than the computational complexity of the stable model semantics [Gelfond and Lifschitz 1988]. For instance the existence problem for stable models of Datalog programs is NP-complete. Similarly, "membership-in-some" problem for such programs is also NP-complete, while the "membership-in-all" problem is co-NP-complete [Marek and Truszczyński 1991].

With the introduction of tabling [Tamaki and Sato 1986] and the development of the SLG resolution procedure [Chen and Warren 1996], more powerful top down proof procedures became available. It is proven in [Chen and Warren 1996] that SLG is sound and search space complete with respect to the well-founded partial model and is polynomial time in case of function free programs.

The impossibility of a complete proof procedure could be considered a drawback with respect to the completion semantics. Indeed, the SLDNF proof procedure [Lloyd 1987] is known to be complete for certain classes of programs under the completion semantics — see [Apt and Bol 1994] for an overview. First, we believe it is more important to use a semantics that corresponds to the intuitive meaning of a program than one for which complete proof procedures exist. Second, despite completeness results for SLDNF, in practice the completeness of SLD(NF) is lost anyway due to the use of the depth-first search strategy of Prolog implementations. So, Prolog programmers are used to (sound but) incomplete proof procedures. In the current systems, incompleteness is caused either by non-termination or by floundering. Users know that they have to reason about this and have developed methodologies to avoid these problems. To some extent, reasoning about the decidability of a class of queries of interest can even be automated. Indeed, as mentioned, decidability of a query is closely related to non-floundering termination of the query. Techniques for analysis of termination of SLD [De Schreye and Decorte 1994] and of SLG [Verbaeten et al. 2001] exist. Floundering can be analyzed by means of abstract interpretation [Cousot and Cousot 1977], more specifically by groundness analysis [Marriott and Søndergaard 1993]. Of course, due to the undecidability results, these techniques cannot be complete.

5.3 Inductive definitions in the absence of complete knowledge

A logic program, which expresses a correct inductive definition has a unique well-founded total model. This presupposes that the programmer, when writing the program, has complete knowledge of the problem domain and can define each predicate of the program.

[16] As defined in [Vardi 1982].

In recent years, several Logic Programming extensions capable of representing incomplete knowledge have been proposed. One is Abductive Logic Programming [Kakas et al. 1992; Denecker 1995], an extension of logic programming by means of abductive reasoning. An abductive logic program is a triple $\langle A, P, IC\rangle$ consisting of a set A of abducible (or open) predicates, a logic program P defining the set of non-abducible predicates and a set IC of classical logic constraints. Another new paradigm is Answer Set Programming [Marek and Truszczyński 1999; Lifschitz 1999; Niemelä 1999]. This approach is based on the stable model semantics [Gelfond and Lifschitz 1988] and is fundamentally different from the view we have developed so far. Under the stable model semantics, a normal program is viewed not as a set of definitions but as a set of rules expressing constraints on the problem domain. Stable models are "possible sets of beliefs that a rational agent might hold" given the rules of the program [Gelfond and Lifschitz 1988].

There are more possibilities, in fact. For instance, the interpretation of a normal program as an inductive definition can and has been adapted to cope with missing knowledge at the predicate level. The approach consists of distinguishing between defined predicates and *open* predicates. The latter are predicates for which the program contains no definition (to be distinguished from predicates with empty definition). We illustrate it with an example for finding the Hamiltonian cycles in a finite directed graph[17]. The unknown Hamiltonian cycle can be expressed by a binary predicate `in/2`; the property that all nodes have to be reachable from a particular node (say node 1) can be defined by a predicate `reachable/1`. This gives the following piece of code:

```
open in/2
node(1).
...
edge(1,2).
...
reachable(U) :- in(1,U).
reachable(V) :- reachable(U), in(U,V).
```

Note that the `reachable/1` predicate depends on the open predicate `in/2`. Given a definition for `in/2`, it is a correct inductive definition and determines a unique model. However not every model (in the language of the program) of `in/2` is a Hamiltonian cycle. As in Answer Set Programming, the set of candidate models has to be constrained. For this task, first order logic is an excellent tool. The constraints that the cycle must pass over the edges and must visit all nodes exactly once can be expressed as the following set of constraints[18]:

```
edge(U,V) <- in(U,V).
V=W <- in(U,V), in(U,W).
U=V <- in(U,W), in(V,W).
reachable(U) <- node(U).
```

[17] The interested reader can find a solution by means of the Answer Set Programming paradigm in [Lifschitz 1999].

[18] We use $\leftarrow$ to stress that these are FOL integrity constraints and not program rules or queries.

Under these constraints, models of `in/2` are restricted to be Hamiltonian cycles. Note that the last constraint does not involve directly the open predicate (but `reachable/` depends on it).

The distinction between definitions and partial knowledge is similar to the distinction in the literature of knowledge representation between *assertional knowledge* and *definitional knowledge* [Reichgelt 1991].

As the example shows, combining inductive definitions of defined predicates in terms of open predicates with FOL formulas gives an expressive language facilitating the declarative formulation of problems. The idea is elaborated in the inductive definition logic presented in [Denecker 2000]. A theory of this logic, called *ID-logic*, consists of a set of FOL formulas and a set of definitions. In models of such theories, the defined predicates of a definition are interpreted by the well-founded model of the definition extending some interpretation of the open predicates. The FOL assertions filter away those well-founded models in which these assertions do not hold. This logic is closely related to and provides the epistemological foundation for abductive logic programming [Kakas et al. 1992] (under extended well-founded semantics [Pereira et al. 1991]), which in turn can be viewed as the study of abductive reasoning in the context of ID-logic.

5.4 Herbrand interpretations versus General interpretations

The use of the grounding of a program as a basis for defining semantics boils down to the use of Herbrand interpretations. The restriction to Herbrand interpretations imposes two assumptions at the knowledge level:

—**Domain Closure**: every element of the domain of discourse is named by at least one ground term.

—**Unique Names**: two different ground terms denote different objects.

These two restrictions imply that there is an isomorphism between the Herbrand Universe and the objects in the problem domain, i.e. that one knows all objects of interest and can distinguish between them. These axioms express complete knowledge of the domain of discourse. However as stated in [Denecker 2000], these restrictions are independent of each other and of the view of logic programs as inductive definitions. The inductive definition logic ID-logic introduced in [Denecker 2000] is based on general interpretations rather than Herbrand interpretations and comprises neither domain closure nor unique names axioms (but both can be expressed in ID-logic).

5.5 Closing the circle: a fixpoint theory for non-monotone operators.

Throughout this paper we have argued that the theory of inductive definitions in mathematical logic provides an epistemological foundation for Logic Programming. Vice versa, Logic Programming can contribute to the study of non-monotone induction in two ways. First, as argued above, the well-founded semantics can be seen as a more general and more robust formalization of the principle of iterated induction. Second, Logic Programming can also contribute to the algebraic theory of induction, namely the fixpoint theory of generalized operators.

Until recently, no fixpoint theory for general (monotone or non-monotone) operators was known that was modeling the principle of iterated induction. Building

on Fitting's work [Fitting 1993] on semantics of Logic Programming in bilattices, [Denecker et al. 2000] developed *Approximation Theory*, an algebraic fixpoint theory for general (monotone and non-monotone) operators in a lattice. This theory defines for each operator O a set of *stable fixpoints* and a unique *well-founded fixpoint* which is a pair (x, y) of lattice elements such that (x, y) approximates each stable fixpoint z (that is $x \leq z \leq y$). This theory has two key properties: (1) it extends Tarski's fixpoint theory in the sense that if the operator O is monotone, then its well-founded fixpoint is the pair (x, x) where x is the least fixpoint of O (in this case, x is also the unique stable fixpoint of O), and (2), the well-founded fixpoint of the immediate consequence operator T_P of a normal program is exactly the well-founded model of P.

In combination with the arguments in section 5.1 that well-founded semantics is a generalized principle of non-monotone induction, we put forward the thesis that Approximation Theory is the natural fixpoint theory of generalized non-monotone induction.

6. CONCLUSIONS

We have revisited the semantics of logic programming. We have developed the thesis that logic programs can be understood as inductive definitions. Elaborating on ideas originally proposed by one of the authors in [Denecker 1998], we have argued that their interpretation as inductive definition gives a more solid epistemological foundation for their canonical models than a reference to common sense. Moreover, this interpretation corresponds to the well-founded semantics. Next, we have shown that this reading of logic programs extends the notion of iterated inductive definitions as studied so far in mathematical logic. Finally, we have elaborated on some of the consequences of this thesis.

We believe that *logic programs as definitions* offer a simple, elegant and powerful conceptualization of logic programming, within reach of comprehension for a broad audience not versed in the literature on semantics of negation and moreover, that it reconciles the semantics of logic programs with the intuitions and expectations of programmers inspired by Kowalski's vision of logic as a programming language.

Acknowledgements

The authors thank Krzysztof R. Apt for his encouragement as well as a number of valuable suggestions. The anonymous referees suggested a number of improvements. The first author also wants to thank Danny De Schreye and Eugenia Ternovska for support and comments on earlier versions of the paper.

REFERENCES

Aczel, P. 1977. An Introduction to Inductive Definitions. In *Handbook of Mathematical Logic*, J. Barwise, Ed. North-Holland Publishing Company, 739–782.

Andréka, H. and Németi, I. 1978. The Generalized Completeness of Horn Predicate Logic as a Programming Language. *Acta Cybernetica 4*, 3–10.

Apt, K. and Blair, H. 1990. Arithmetical classification of perfect models of stratified programs. *Fundamenta Informaticae 13,* 1, 1–17.

Apt, K., Blair, H., and Walker, A. 1988. Towards a theory of Declarative Knowledge. In *Foundations of Deductive Databases and Logic Programming*, J. Minker, Ed. Morgan Kaufmann, 89–148.

Apt, K. and Bol, R. 1994. Logic programming and negation: a survey. *Journal of Logic Programming 19-20*, 9–71.

Apt, K. R. and Emden, M. V. 1982. Contributions to the Theory of Declarative Knowledge. *Journal of the ACM 29,* 3, 841–862.

Barwise, J. 1977. *Admissible Sets and Structures*. Springer Verlag.

Blair, H., Marek, V., and Schlipf, J. 1995. The expressiveness of locally stratified programs. *Annals of Mathematics and Artificial Intelligence 15*, 209–229.

Bowen, K. and Kowalski, R. 1982. Amalgamating Language and Metalanguage in Logic Programming. In *Logic Programming*, K. Clark and S.-A. Tärnlund, Eds. Academic Press, 153–172.

Brass, S. and Dix, J. 1999. Semantics of (disjunctive) logic programs based on partial evaluation. *Journal of Logic Programming 40,* 1 (July), 1–46.

Buchholz, W., Feferman, S., and Sieg, W. P. W. 1981. *Iterated Inductive Definitions and Subsystems of Analysis: Recent Proof-Theoretical Studies*. Lecture Notes in Mathematics, vol. 897. Springer-Verlag.

Chen, W. and Warren, D. 1996. Tabled evaluation with delaying for general logic programs. *Journal of the ACM 43,* 1 (January), 20–74.

Clark, K. 1978. Negation as failure. In *Logic and Databases*, H. Gallaire and J. Minker, Eds. Plenum Press, 293–322.

Colmerauer, A., Kanoui, H., Pasero, R., and Roussel, P. 1973. Un système de communication homme-machine en Français. Tech. rep., GIA, Université d'Aix Marseille II, Luminy, France.

Cousot, P. and Cousot, R. 1977. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the Fourth ACM Symp. on Principles of Programming Languages*. 238–252.

De Schreye, D. and Decorte, S. 1994. Termination of logic programs: the never-ending story. *Journal of Logic Programming 19-20*, 199–260.

Denecker, M. 1995. A Terminological Interpretation of (Abductive) Logic Programming. In *International Conference on Logic Programming and Nonmonotonic Reasoning*, V. Marek, A. Nerode, and M. Truszczynski, Eds. Lecture notes in Artificial Intelligence 928. Springer, 15–29.

Denecker, M. 1998. The well-founded semantics is the principle of inductive definition. In *Logics in Artificial Intelligence*, J. Dix, L. Fariñas del Cerro, and U. Furbach, Eds. Lecture Notes in Artificial Intelligence, vol. 1489. Springer-Verlag, 1–16.

Denecker, M. 2000. Extending classical logic with inductive definitions. In *First International Conference on Computational Logic (CL2000)*. Lecture Notes in Artificial Intelligence, vol. 1861. Springer, 703–717.

Denecker, M. and De Schreye, D. 1993. Justification semantics: a unifying framework for the semantics of logic programs. In *Proc. of the Logic Programming and Nonmonotonic Reasoning Workshop*. MIT Press, 365–379.

Denecker, M., Marek, V., and Truszczynski, M. 2000. Approximating operators, stable operators, well-founded fixpoints and applications in nonmonotonic reasoning. In *Logic-based Artificial Intelligence*, J. Minker, Ed. Kluwer Academic Publishers, Boston, Chapter 6, 127–144.

Feferman, S. 1970. Formal theories for transfinite iterations of generalised inductive definitions and some subsystems of analysis. In *Intuitionism and Proof theory*, A. Kino, J. Myhill, and R. Vesley, Eds. North Holland, 303–326.

Ferry, A. 1994. Topological characterizations for logic programming semantics. Ph.D. thesis, University of Michigan.

Fitting, M. 1985. A Kripke-Kleene Semantics for Logic Programs. *Journal of Logic Programming 2,* 4, 295–312.

Fitting, M. 1991. Bilattices and the semantics of logic programming. *Journal of Logic Programming 11,* 2 (August), 91–116.

Fitting, M. 1993. The family of stable models. *Journal of Logic Programming 17,* 2,3&4, 197–225.

Fitting, M. 1997. A theory of truth that prefers falsehood. *Journal of Philosophical Logic 26*, 477–500.

Fitting, M. 2001. Fixpoint semantics for logic programming - a survey. *Journal of Theoretical Computer Science*. To appear.

Gelfond, M. 1987. On Stratified Autoepistemic Theories. In *Proc. of AAAI87*. Morgan Kaufman, 207–211.

Gelfond, M. and Lifschitz, V. 1988. The stable model semantics for logic programming. In *Proc. of the International Joint Conference and Symposium on Logic Programming*. MIT Press, 1070–1080.

Gurevich, Y. and Shelah, S. 1986. Fixed-point Extensions of First-Order Logic. *Annals of Pure and Applied Logic 32*, 265–280.

Kakas, A. C., Kowalski, R., and Toni, F. 1992. Abductive Logic Programming. *Journal of Logic and Computation 2,* 6, 719–770.

Kleene, S. 1955. Arithmetical predicates and function quantifiers. *Transactions Amer. Math. Soc. 79*, 312–340.

Kolaitis, P. G. and Papadimitriou, C. H. 1991. Why not negation by fixpoint. *Journal of Computer and System Sciences 43,* 1, 125–144.

Kowalski, R. 1974. Predicate logic as a programming language. In *Proc. of IFIP 74*. North-Holland, 569–574.

Kreisel, G. 1963. Generalized inductive definitions. Tech. rep., Stanford University.

Lifschitz, V. 1999. Answer set planning. In *Logic Programming, Proc. 1999 Int. Conf. on Logic Programming (ICLP'99)*, D. De Schreye, Ed. MIT Press, 23–37.

Lloyd, J. 1987. *Foundations of Logic Programming*. Springer-Verlag.

Marek, V., Nerode, N., and Remmel, J. 1994. The stable models of predicate logic programs. *Journal of Logic Programming 21,* 3, 129–154.

Marek, V. and Truszczyński, M. 1989. Stable semantics for logic programs and default reasoning. In *Proc. of the North American Conference on Logic Programming and Non-monotonic Reasoning*, E.Lust and R. Overbeek, Eds. 243–257.

Marek, V. and Truszczyński, M. 1991. Autoepistemic Logic. *Journal of the ACM 38,* 3, 588–619.

Marek, V. and Truszczyński, M. 1999. Stable models and an alternative logic programming paradigm. In *The Logic Programming Paradigm: a 25 Years Perspective*, K. Apt, V. Marek, M. Truszczynski, and D. Warren, Eds. Springer-Verlag, pp. 375–398.

Marek, W. 1989. Stable theories in autoepistemic logic. *Fundamenta Informaticae 12,* 2, 243–254.

Marriott, K. and Søndergaard, H. 1993. Precise and efficient groundness analysis for logic programs. *ACM-LOPLAS 2,* 1-4, 181–196.

Martens, B. and De Schreye, D. 1995. Why untyped non-ground meta-programming is not (much of) a problem. *Journal of Logic Programming 22,* 1 (January), 47–99.

Martin-Löf, P. 1971. Hauptsatz for the intuitionistic theory of iterated inductive definitions. In *Proceedings of the Second Scandinavian Logic Symposium*, J. Fenstad, Ed. 179–216.

Moore, R. 1985. Semantical considerations on nonmonotonic logic. *Artificial Intelligence 25,* 1, 75–94.

Moschovakis, Y. N. 1974. *Elementary Induction on Abstract Structures*. North-Holland Publishing Company, Amsterdam- New York.

Niemelä, I. 1999. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence 25,* 3,4, 241–273.

Pereira, L., Aparicio, J., and Alferes, J. 1991. Hypothetical Reasoning with Well Founded Semantics. In *Proc. of the 3th Scandinavian Conference on AI*, B. Mayoh, Ed. IOS Press, 289–300.

Przymusinska, H. and Przymusinski, T. 1988. Weakly perfect model semantics for logic programs. In *Proc. of the Fifth International Conference and Symposium on Logic Programming*, R. Kowalski and K. Bowen, Eds. MIT Press, 1106–1120.

Przymusinska, H. and Przymusinski, T. 1990. Weakly Stratified Logic Programs. *Fundamenta Informaticae 13*, 51–65.

Przymusinski, T. 1988. On the declarative Semantics of Deductive Databases and Logic Programs. In *Foundations of Deductive Databases and Logic Programming*, J. Minker, Ed. Morgan Kaufman, 193–216.

Przymusinski, T. 1989a. Every Logic Program Has a Natural Stratification And an Iterated Least Fixed Point Model. In *Proceedings of the Eight ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*. 11–21.

Przymusinski, T. 1989b. On the declarative and procedural semantics of logic programs. *Journal of Automated Reasoning 5*, 167–205.

Przymusinski, T. 1990. Well founded semantics coincides with three valued Stable Models. *Fundamenta Informaticae 13*, 445–463.

Reichgelt, H. 1991. *Knowledge Representation: an AI Perspecitive*. Ablex Publishing Corporation.

Reiter, R. 1978. On Closed World Data bases. In *Logic and Data Bases*, H. Gallaire and J. Minker, Eds. Plenum Press, New York, 55–76.

Reiter, R. 1980. A logic for default reasoning. *Artificial Intelligence 13*, 81–132.

Robinson, J. 1965. A machine-oriented logic based on the resolution principle. *Journal of the ACM 12,* 1, 23–41.

Schlipf, J. 1995a. Complexity and undecidability results in logic programming. *Annals of Mathematics and Artificial Intelligence 15*, 257–288.

Schlipf, J. 1995b. The expressive powers of the logic programming semantics. *Journal of Computer and System Sciences 51*, 64–86.

Smullyan, R. 1968. *First-Order Logic*, second ed. Springer, New York.

Spector, C. 1961. Inductively defined sets of natural numbers. In *Infinitistic Methods (Proc. 1959 Symposium on Foundation of Mathematis in Warsaw)*. Pergamon Press, Oxford, 97–102.

Tamaki, H. and Sato, T. 1986. OLD resolution with tabulation. In *Third International Conference on Logic Programming*. Lecture Notes in Computer Science, vol. 225. Springer-Verlag, 84–98.

Tarski, A. 1955. Lattice-theoretic fixpoint theorem and its applications. *Pacific journal of Mathematics 5*, 285–309.

van Emden, M. and Kowalski, R. 1976. The semantics of Predicate Logic as a Programming Language. *Journal of the ACM 4,* 4, 733–742.

Van Gelder, A. 1988. Negation as Failure Using Tight Derivations for General Logic Programs. In *Foundations of Deductive Databases and Logic Programming*, J. Minker, Ed. Morgan Kaufmann, 149–176.

Van Gelder, A. 1993. The Alternating Fixpoint of Logic Programs with Negation. *Journal of Computer and System Sciences 47,* 1, 185–221.

Van Gelder, A., Ross, K. A., and Schlipf, J. 1991. The Well-Founded Semantics for General Logic Programs. *Journal of the ACM 38,* 3, 620–650.

Vardi, M. 1982. The complexity of relational query languages. In *14th ACM Symposium on Theory of Computing*. 137–145.

Verbaeten, S., Sagonas, K., and De Schreye, D. 2001. Termination proofs for logic programs with tabling. *ACM Transactions on Computational Logic 2,* 1, 57–92.

Received August 2000; revised February 2001, April 2001; accepted April 2001