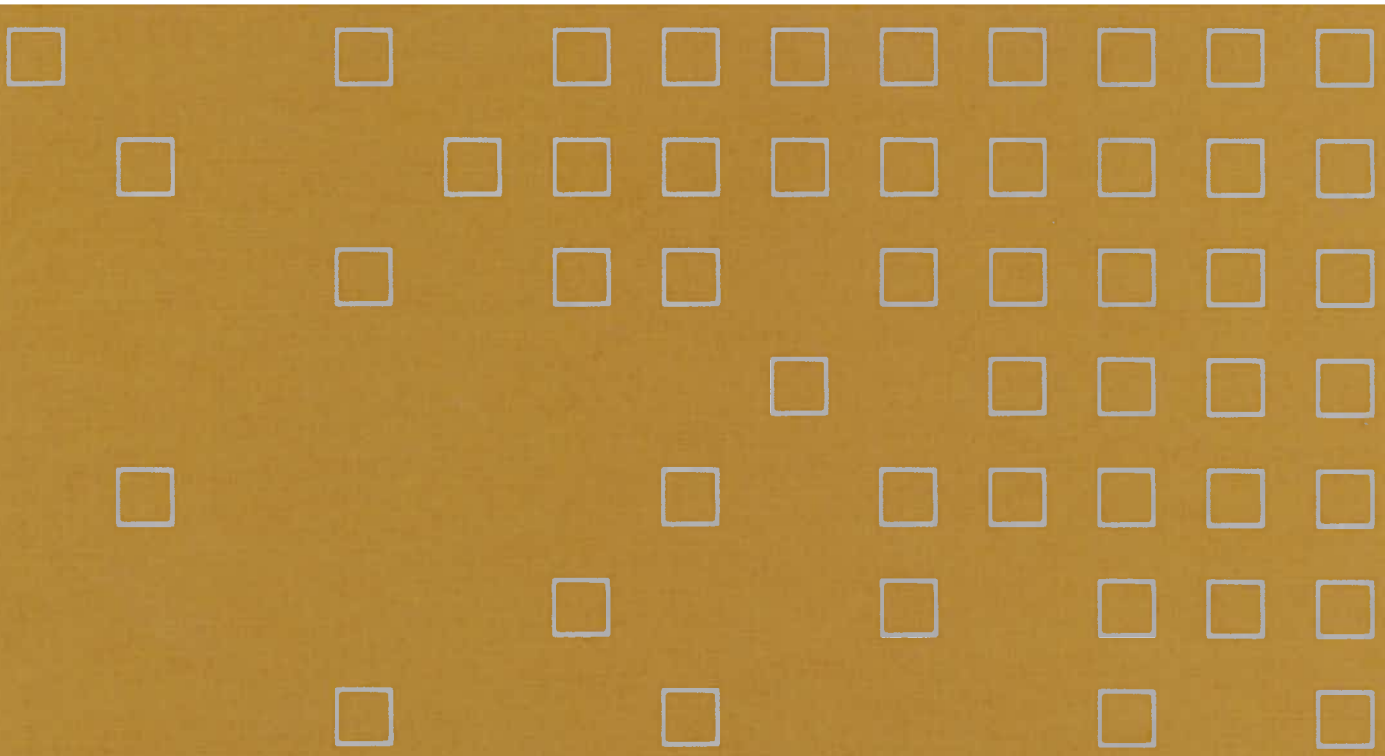

Centrum voor Wiskunde en Informatica

Centre for Mathematics and Computer Science

D. Turi

Logic programs with negation: classes, models, interpreters

Computer Science/Department of Software Technology Report CS-R8943 October

1989

Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

D. Turi

Logic programs with negation: classes, models, interpreters

Computer Science/Department of Software Technology Report CS-R8943 October

---

The Centre for Mathematics and Computer Science is a research institute of the Stichting Mathematisch Centrum, which was founded on February 11, 1946, as a nonprofit institution aiming at the promotion of mathematics, computer science, and their applications. It is sponsored by the Dutch Government through the Netherlands Organization for the Advancement of Research (N.W.O.).

Logic Programs with Negation: Classes, Models, Interpreters

Daniele Turi
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands
and
Department of Computer Science, University of Pisa,
Corso Italia 40, 56100 Pisa, Italy

The aim of this survey is to place in a homogeneous framework disparate notions and results arising from the introduction of negation in logic programming.

1980 Mathematics Subject Classification (1985): 68Q55, 68T15, 68T30.
1986 CR Categories : F.4.1, F.3.2.

BACKGROUNDS

The reader of this survey is assumed to be familiar with Positive Logic Programming, that is, the combination of Horn clause logic with SLD-resolution. The material covered by the first three chapters of LLOYD [Ld] or by the first two of its extended and revised edition ([Ld1]) provide a sufficient background for the understanding of this work. The same holds for the first three chapters of APT [A]. Basic definitions like those of *atom, literal, program clause, head and body of clause, query, Herbrand base, Herbrand interpretation, immediate consequence operator, least Herbrand model, success set, substitution, most general unifier,* are therefore assumed as already given.

To us, a *positive* program is what in LLOYD [Ld]-[Ld1] is called *definite* program and in APT [A] simply *program*. That is, a program whose clauses contain only positive literals (both in the head and in the body). Similarly, a *positive query* is a query with only positive literals. Instead, a *logic program with negation* —or more simply *general* program— is a program whose clause bodies are conjunctions of literals, rather than of atoms. (Recall that a literal is either a atom (*positive literal*) or an atom preceded by **not** (*negative literal*)). Notice that in [Ld1] (but not in [Ld] nor in [A]) the word *general* program defines a wider class of programs, while our general programs are called *normal* programs there. When not otherwise specified a *query* may contain negative literals.

Throughout the paper we use LP instead of Logic Programming and AI instead of Artificial Intelligence.

SYNOPSIS

A preliminary overview of this survey's structure is given by the Table of Contents, placed at the end of the paper. Here we analyse it in detail.

Chapter 1 is a brief introduction.

Chapter 2 presents the first of the two interpreters we give: *SLDNF-resolution*.

Chapter 3 consists of four sections. In the first we examine the semantics of Positive LP and we introduce Clark's Equational Theory (CET) and the Domain Closure Axioms (DCA) independently from *completion* and *cwa*. Starting from the so-called *universal query problem*, Przymusinski's minimal model semantics (*MIN-semantics*) is also given. Afterwards, the new declarative semantics for positive programs allowing a complete characterization of the SLD-resolution computed answers (*s-semantics*) is introduced. We show (and prove) its direct relationship with the MIN-semantics,

Report CS-R8943
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

suggesting that the prefix 's-' should be read as 'supra' instead of 'subset'. In the following section the *closed world assumption* (*cwa*) is presented, first for positive programs and then enlarged for general ones. The *generalized cwa* is also given. In the next section we move from cwa to *circumscription*, using CET and DCA to show their close relationship. This is followed by a detailed account of *completion*. Again, first we treat positive programs only, presenting completion as a 'computable' variant of circumscription. Then we turn to general programs and we introduce a *three-valued logic* to avoid inconsistencies. Kunen's characterization of the three-valued logical consequences of completion closes the chapter.

Chapter 4 is devoted to the completeness of SLDNF-resolution wrt the completion of some opportune classes of programs and queries. In section 4.1 the problem of *floundering* is faced, giving a hint of what Chan's *constructive negation* is and introducing *allowedness* to state the first of the three completeness results in the survey. After the next section, where the basic notion of *dependency graph* is given, we devote three sections to the definition of some classes of programs consistent under completion (in two-valued logic): *hierarchical*, *stratified* and *call-consistent* programs. Section 4.6 provides all of them with a straightforward and powerful extension. Notice that stratified programs (section 4.4) are exactly in the middle of the survey. They are a turning point, indeed. The chapter ends with the second completeness result, after that the definition of *strictness* is given.

Chapter 5 is again divided in four sections, suggesting a symmetry with chapter 3. Similar themes are indeed treated, only shifted from positive to stratified programs, which turn out to be the class of general programs most related to the positive ones. As a consequence, most of the notions presented in the first three sections of chapter 3 (completion is left out) are mimicked in the first three sections here. *Perfect models, iterated cwa* and *prioritized circumscription* are the equivalents for stratified programs of minimal models, cwa and circumscription for positive ones. The last section of the chapter presents an interesting extension of perfect models, which is based on a natural generalization of the negation as failure rule.

Chapter 6 is dedicated to *SLS-resolution,* an interpreter for stratified programs which brings us to the third and last completeness result in the survey.

Chapter 7 closes the paper, bringing together and classifying in two research lines its essential elements.

Warning Our bibliographic references consist mainly of those pointers following the title of each section. They refer to the main sources for the material in that section and to them we refer the reader for all missing proofs. Little attention is paid to attribute theorems and definitions to their original authors. We apologize for this.

1. Introduction

1.1. Negation in Logic Programming

First of all what negation does **not** mean in LP: there is a positive logic program which emulates the Universal Turing Machine (section 4 of Apt [A]). Therefore, negation is not needed to increase the expressive power of LP. Moreover, though mathematical logic provides the language to LP, in general the representation of mathematical knowledge is not the object of application for LP, thus negation does not need to be correlated to the logical one either.

Actually, negation was introduced in LP by Clark [C] in order to allow negative information to be deduced from positive logic programs. In that context a logic program was regarded as a *deductive database* (unit clauses $\equiv$ explicit database, implicative clauses $\equiv$ implicit database), thus as an elementary form of *representation of knowledge for common sense reasoning.*

Suppose all a student of English knows about nouns in plural form is that they end with 's'. When

asked about 'formulas' he will correctly conclude that it is the plural of 'formula'. Though 'formulae' would lead him to error, making use of his basic −incomplete− notions he may reach a stage at which latin nouns are classified as *abnormal* with respect to the plural.

The above is an example of common sense reasoning. Its peculiar elements −*incompleteness* and *non-monotonicity*− are both present: even though knowledge of the English language could be virtually complete, in general the incompleteness of common sense reasoning −rather than to individual ignorance− is due to the natural limitations of our immanent world. Non-monotonicity is related to such incompleteness in the sense that previously drawn conclusions may be invalidated when the set of available informations becomes less incomplete.

Such imperfection (incompleteness and non-monotonicity) is totally reflected in LP's negation −a tool to deduce (presumably) useful information which may otherwise not be deducible by means of pure logic.

1.2. Classes, Models and Interpreters

Positive LP enjoys an admirable completeness: every program has a mathematical (*declarative*) meaning which is totally equivalent to the procedural (*operational*) one. That is, in order to understand the result of a computation, no knowledge of the underlying computational mechanism (*interpreter*) is required. Merely by elementary mathematical means it is possible to establish the meaning of a computation.

The practical import of such a property should be clear to anyone who has experience with other programming languages, whose proper use and comprehension always necessitate a deep knowledge of the interpreter. But perhaps even more relevant is the theoretical aspect of the matter. As already mentioned, when extended with negation, LP is an elementary language for knowledge representation. Preserving under such extension the original declarative nature of LP, we would move a first step 'towards a theory of declarative knowledge'.

Against these noble purposes stands the non-monotonicity of negation, which, though necessary, is *in general* in contradiction with our theoretical requirements. This survey's aim is to present some *classes* of both programs and queries which, together with some particular *models* and *interpreters*, exhibit the same completeness properties of Positive LP.

The ideal is to reach the least restricted classes of programs and queries. The means are: a) to increase the computational power of the interpreter, b) to restrict the number of models considered representative for the program, b') to relax the notion of model itself. We shall proceed hierarchically (and historically), starting from trivially complete classes and then enlarging them operating with a), b) and b').

2. INTERPRETERS I

2.1. SLDNF-Resolution ([K1])

SLD-resolution extended by the meta-rule *negation as finite failure to prove (SLDNF-resolution)* is the most classical of the interpreters for logic programs with negation. We shall introduce it here through a very elegant recursive definition, due to Kunen. First some notations: α and β stand for atoms, ϕ, ψ and ξ for conjunctions of literals, σ and θ for substitutions, *yes* stands for the empty substitution (the only possible for ground formulas). By mutual recursion, we define a binary relation R and a monadic predicate F. R stands for 'Returns': $\phi R\sigma$ *holds iff SLDNF-resolution succeed on the query* ϕ *returning* σ *as answer.* F stands for 'Finitely Failed': ϕ *belongs to F iff the interpreter fails, and fails finitely, to give an answer to* ϕ.

Basis of the recursion is the query **true**, for which the interpreter, just like SLD-resolution, returns *yes*:

0 : **true***Ryes.*

For queries with (at least) one positive atom is still SLD-resolution to drive the interpreter. Thus, for

$$\phi = \alpha \wedge \psi$$

a) Success:

$$R^+ \quad \frac{\exists\ \beta \leftarrow \xi \text{ in P s.t. } \sigma = mgu(\alpha,\beta) \text{ and } (\xi \wedge \psi)\sigma R\theta}{\phi R(\sigma\theta \mid \phi)}$$

where $(\sigma\theta \mid \phi)$ is σ composed θ, restricted to the variables of ϕ.

b) Finite Failure:
Here Kunen uses just one rule, but we rather split it in two, in order to emphasize a difference in the use of recursion.

$$F_1^+ \quad \frac{\alpha \text{ does not unify with any head of clause in P}}{\phi \in F}$$

$$F_2^+ \quad \frac{[\forall\ \beta \leftarrow \xi \text{ in P s.t. } \exists\ \sigma = mgu(\alpha,\beta)] \Rightarrow (\xi \wedge \psi)\sigma \in F}{\phi \in F}.$$

To resolve a negative literal $\neg\alpha$ in the query, we could think of just complementing the answer of the interpreter to α. But some care is needed, as the following examples shows:

EXAMPLE 2.1.
Let us consider the program

$$p(a) \leftarrow,$$

$$q(b) \leftarrow$$

and the query

$$\leftarrow \neg\, p(x).$$

Though $p(x)R\{x/a\}$, we cannot infer $\neg p(x) \in F$, because there is an instance of $p(x)$ $-p(b)-$ which belongs to F. □

The way out to this problem is to 'select' a negative literal in the query only if it is ground. Of course, this is not always possible, but then the computation is simply resumed with the query declared *floundered.* (Clearly, since *floundering* is not a logical notion, this implies that we have to impose some opportune restrictions on both programs and queries in order to avoid a 'built-in' form of incompleteness.)

Requiring groundness for selected negative literals, we obtain a simmetry between F and R with respect to negation. Indeed, for ground atoms, since the only possible substitution is the empty one, R acts as monadic predicate, just like F does. Therefore, α*Ryes* can be complemented to $\neg\alpha \in F$ and $\alpha \in F$ to $\neg\alpha$*Ryes* without loosing information.

Formally, for

$$\phi = \neg\alpha \wedge \psi,\ \alpha \textit{ ground}$$

a) Success:

$$R^- \quad \frac{\alpha \in F \text{ and } \psi R\sigma}{\phi R\sigma}$$

b) Finite Failure:

$$F^- \quad \frac{\alpha R yes}{\phi \in F}.$$

3. Models I

3.1. Minimal Model Semantics for Positive Programs ([FLMP], [P])

The declarative semantics of Positive LP is the ideal upon which the declarative meaning of logic programs with negation has to be moulded. We shall devote this section to reasses its basic notions under the light of some recent works (Falaschi, Levi, Martelli and Palamidessi [FLMP], Kunen [K], Przymusinski [P]).

The cardinal semantical property of Positive LP is the following completeness theorem: *all the most general substitutions under which a query is* logical consequence *of a program P are exactly all the answers the interpreter (SLD-resolution) can possibly give to the same query when applied to P.*

The notion of logical consequence is in a way non constructive. The models of a program form a *heterogeneous* collection of structures, with no criteria to build them. Gödel's completeness theorem provides a syntactical way to capture the entire notion of logical consequence, but still then no model for the program is actually exhibited.

The universal form of logic programs offers the possibility of concentrating on just a *homogeneous* collection of models, namely the Herbrand one. The reader should be familiar with the following properties of positive logic programs Herbrand models (and interpretations): an interpretation is a set of ground atoms; the collection of such interpretations (together with the usual set inclusion) forms a complete lattice; there always exists a least (Herbrand) model; such least model is both the intersection of all (Herbrand) models and the least fixpoint of the immediate consequence operator.

The last property above amounts to the existence of a declarative semantics for Positive LP based on a *single* and *constructable* model. However, being based on Herbrand's theorem, such model is representative only if we restrict our attention to the class of *existential* queries.

Example 3.1 (Universal Query Problem)

The program P:

$$p(a) \leftarrow$$

has as least (and only) Herbrand model

$$M_P = \{p(a)\}.$$

Therefore,

$$M_P \models \forall x.p(x).$$

In order to be complete, the interpreter should then return *yes* —the empty substitution— as answer to the query $\leftarrow p(x)$. However SLD-resolution, as well as any interpreters based on the unification algorithm, will return $\{x/a\}$ as answer. □

If we extend the program above to include a totally incorrelated clause like $q(b)\leftarrow$, we have that M_P does not imply $\forall x.p(x)$ anymore. This suggests that, in order to overcome the universal query problem, it suffices to extend the language of every program with a new element. A more general and uniform solution (Kunen [K]) is to assume a fixed maximal Herbrand universe for all programs — a universe based on a language containing infinite function symbols for every arity. No program can have a language of greater cardinality and thus the one-element extension above is always included in

it (up to isomorphisms).

Another solution (PRZYMUSINSKI [P]) is to extend the programs with an axiomatization of the unification algorithm itself, solving the problem at its very source. Few natural axioms are then sufficient:

$$U^+ : \quad f(x_1,...,x_n)=f(y_1,...,y_n) \rightarrow x_1=y_1 \wedge \cdots \wedge x_n=y_n$$

for each function f in the program;

$$U^- : \quad f(x_1,...,x_n) \neq g(y_1,...,y_m)$$

for each pair of distinct functions f, g;

$$OA : \quad t(x) \neq x$$

for each term t in wich x occurs.

Of course, in order to interpret "$=$" correctly, we have to add the classical *equality axioms* (EA), i.e. reflexivity, symmetry and transitivity together with

$$x_1=y_1 \wedge \cdots \wedge x_n=y_n \rightarrow f(x_1,...,x_n)=f(y_1,...,y_n)$$

$$x_1=y_1 \wedge \cdots \wedge x_n=y_n \rightarrow [p(x_1,...,x_n) \leftrightarrow p(y_1,...,y_n)]$$

(for every function f and every predicate p in the program).

Axioms U^+, U^- and OA are the *freeness axioms* (FA). Equality and freeness axioms together are called *Clark's Equational Theory* ($CET = FA + EA$). CET's relationship with the unification algorithm is expressed by the following lemma:

LEMMA 3.2. *Let t_1 and t_2 be any terms. Then:*
If t_1 unifies with t_2 then there exists an mgu σ of t_1 and t_2 such that

$$CET \models \forall(t_1=t_2)\sigma ;$$

otherwise

$$CET \models \forall(t_1 \neq t_2). \quad \square$$

Since CET does not refer to relations other than equality (which is interpreted as identity), its interpretations amount merely to interpretations of terms (*pre-interpretations*). Therefore, all interpretations of P+CET can be obtained through the two following stages: i) specify a pre-interpretation J of CET (on the language of P, of course); ii) assign truth values to the predicates of P for all tuples of terms pre-interpreted by J. The resulting interpretation I is said to be *based on J*.

Just like an atom belonging to a Herbrand interpretation is called *ground*, a predicate instantiated with terms pre-interpreted by J could be called *J-ground atom* (and this will be our terminology, too). Indeed, the Herbrand interpretations of a program P are all based on a particular pre-interpretation (we shall call it J_P), which is completely determined by the syntax: *every term corresponds to an unique semantical object and no two terms are mapped on the same element.*

The restriction to J_P can be axiomatized by adding to CET the following *domain closure axiom:*

$$DCA \equiv x=t_1 \vee x=t_2 \vee \cdots$$

where t_1, t_2, $\cdots$ are all the ground terms in the language of the program. Notice that, when function symbols do appear in the program, DCA is an infinite disjunction. On the contrary, the *weak domain closure axiom*

$$WDCA \equiv \exists y_1,...,y_n(x=f_1(y_1,...,y_n)) \vee \cdots \vee \exists y_1,...,y_m(x=f_k(y_1,...,y_m))$$

(where $f_1,...,f_k$ are all the function symbols in the program) is always finite. But we shall later see that WDCA can sometimes be too weak.

Its syntactical nature places J_P in a special position with respect to the other pre-interpretations: *J_P is a model of CET and a pre-interpretation of P with minimal cardinality.* Indeed, it is easy to verify

that every other pre-interpretation of P which is model of CET contains in its domain an isomorphic copy of P's Herbrand universe.

Several notions involving J_P can be naturally generalized to all other models of *CET*. For instance, we can define B_P^J, the *J-base* of P, as the set of J-ground atoms of the program. Also the immediate consequence operator T_P is easily extended to T_P^J: for every interpretation I based on J and for each $\alpha \in B_P^J$

$$\alpha \in T_P^J(I) \quad \textit{iff} \quad \exists\ \sigma \text{ state over J and}$$
$$\exists\ A \leftarrow F \text{ clause of P such that}$$
$$\alpha = A\sigma \text{ and } I \models_\sigma F.$$

Each J defines a complete lattice of interpretations and thus, since T_P^J is monotonic, $T_P^J \uparrow \omega = lfp(T_P^J)$. But the analogy with T_P^J goes further. Given any J model of CET, the intersection of all models of P based on J is the least model of P based on J. Moreover, it is equivalent to the least fixpoint of T_P^J. The set $MIN(P)$ of minimal models of P+CET is then to be obtained collecting each of these least models for every possible J model of CET.

$MIN(P)$ enjoys homogeneity and constructivity, but in a lower degree than the least Herbrand model. Though the various pre-interpretations share the same framework, they are still largely undetermined. On the other hand, $MIN(P)$ has the same representative power as the entire collection of models of P. *The completeness result at the beginning of this section still holds when in place of* "logical consequence of P" *we put* "true in all minimal models of P+CET".

The major shortcoming of the $MIN(P)$ semantics is in the violation of the single model approach which makes the least Herbrand model semantics so attractive, despite of the restrictions it imposes. A solution to this problem arises from the work of FALASCHI, LEVI, MARTELLI AND PALAMIDESSI [FLMP].

$MIN(P)$ is the result of embedding the unification algorithm into the syntax. Lifting such embedding from the syntactical level to the semantical one we can *abstract* the infinite collection $MIN(P)$ into a *single* structure —a 'supra-model' (*s-model*) for P. To this purpose we need to allow variables to appear in the interpretations (*s-interpretations*).

EXAMPLE 3.3.

The s-interpretation

$$I = \{\ p(f(x))\ \}$$

stands for the whole family of interpretations

$$\{\ p(f(t))\ \}_{t \in D}\ ,$$

where D is any given domain for I.

For $P = p(f(x)) \leftarrow$, I is a synthetic representation of $MIN(P)$. □

The *Herbrand s-universe* of a program P is simply the Herbrand universe of the language of P extended with infinite variables. Since in LP variables are always closed (by the implicit universal quantification of each clause), a distinction between them is necessary only inside atoms: $p(x,y)$ is equivalent to $p(x,z)$, but different from $p(x,x)$. Variables are thus grouped in equivalence classes and B_P^S, the *Herbrand s-base* of P, is built up from the Herbrand s-universe modulo such equivalence. S-Interpretations are then *subsets* (this is the original meaning of the prefix *s*, but we definitely prefer *supra*) of the Herbrand s-base. All Herbrand interpretations' properties still hold for the s- extension: complete lattice, immediate s-consequence operator, least s-model both as intersection and as least fixpoint.

The definition of s-model requires a reformulation of the notion of truth (comp. principle (b') of

section 1.2):

$$I \models_S A \qquad \textit{iff} \quad \exists\, \alpha \in I \text{ such that } \alpha \text{ is more general than } A$$

$$I \models_S A \leftarrow A_1,\ldots,A_n \quad \textit{iff} \quad \forall\, \alpha_1,\ldots,\alpha_n \in I$$
$$\exists\, \theta = \mathrm{mgu}((A_1,\ldots,A_n),\ (\alpha_1,\ldots,\alpha_n)) \Rightarrow A\theta \in I.$$

(By "$\in$" we intend here a membership which takes variables redenominations into account, ie such that $p(x,y) \in \{ p(x,z)\}$ and $p(x,y) \notin \{ p(x,x)\}$. For any two atoms $-$or vectors of atoms$-$ α_1 and α_2 we say that α_1 is *more general* than α_2 when there exists a substitution θ s.t. $\alpha_1\theta = \alpha_2$.)

It is important to recall that it is **not** our purpose to introduce an alternative logic. We simply aim to provide a synthetic way of constructing entire families of models for P+CET. Indeed, our next step is to define the *immediate s-consequence operator* $-$the elementary unit of s-model construction:

$$\alpha \in T_P^S(I) \quad \textit{iff} \quad \exists\, A \leftarrow A_1,\ldots,A_n \text{ in } P$$
$$\exists\, \alpha_1,\ldots,\alpha_n \in I \quad \exists\, \theta \text{ such that}$$
$$\theta = \mathrm{mgu}((A_1,..,A_n),\ (\alpha_1,\ldots,\alpha_n))$$
$$\alpha = A\theta$$

Every model in $MIN(P)$ can be obtained as 'projection' of $T_P^S{\uparrow}\omega$, the least Herbrand s-model of P. By 'projection' we intend here a *single* state application to an opportune redenomination of the s-model, equivalent to the original one (for the definition of *state* see APT [A]). The redenomination, which we shall denote by "$\star$", is necessary in order to pass at once from a set like $\{ p(x) \}$ to $\{ p(a), p(b) \}$. (In this particular case $\{ p(x) \}^*$ could be $\{ p(x), p(y) \}$ with projecting state σ: $\sigma(x) = a$ and $\sigma(y) = b$.)

THEOREM 3.4. *Let P be a positive logic program.*

$$\forall\, J \text{ model of CET } \exists\, \rho \text{ over J such that } \quad T_P^J{\uparrow}\omega = (T_P^S{\uparrow}\omega)^*\rho\,.$$

PROOF.

By induction.

0)

$$T_P^J{\uparrow}0 = \varnothing = T_P^S{\uparrow}0$$

$n \rightarrow n+1$)

$$\exists\, \rho \text{ over J s.t. } T_P^J{\uparrow}n = (T_P^S{\uparrow}n)^*\rho$$

$$\alpha \in T_P^J{\uparrow}n+1 \Rightarrow \exists\, A \leftarrow A_1,\ldots,A_k \text{ in P}$$
$$\exists\, \sigma \text{ over J s.t.}$$
$$\alpha = A\sigma \text{ and } (A_1,\ldots,A_k)\sigma \subseteq T_P^J{\uparrow}n$$

$$(A_1,\ldots,A_k)\sigma \subseteq T_P^J{\uparrow}n \Rightarrow \exists\, \alpha_1,..,\alpha_k \in (T_P^S{\uparrow}n)^* \text{ s.t.}$$
$$(A_1,\ldots,A_k)\sigma = (\alpha_1,\ldots,\alpha_k)\rho$$
$$\Rightarrow \exists\, \theta = \mathrm{mgu}((A_1,\ldots,A_k),\ (\alpha_1,\ldots,\alpha_k)) \quad [\text{by J} \models \text{CET}]$$
$$\Rightarrow A\theta \in T_P^S{\uparrow}n+1 \text{ and}$$
$$\exists\, \gamma \text{ over J s.t. } A\theta\gamma = \alpha$$

$$\Rightarrow \forall\, \alpha \in T_P^J{\uparrow}n+1 \quad \exists \alpha' \in T_P^S{\uparrow}n+1 \text{ and } \exists\gamma \text{ over J s.t.} \quad \alpha = \alpha'\gamma$$

Applying a proper "$\star$" and consequently combining the various γ's we have that

$$\exists\ \pi \text{ over J s.t. } T_P^J \uparrow n+1 \subseteq (T_P^S \uparrow n+1)^\star \pi\ .$$

The other side of the equivalence can be obtained with similar arguments. □

The representative power of the 's-semantics' is even greater than the $MIN(P)$ semantics'. The strongest completeness result for Positive LP, i.e. completeness without requiring substitutions to be most general at the declarative level, can be stated in terms of s-semantics: *the whole set of substitutions under which a query is true in the least Herbrand s-model of P is equivalent to the set of answers the interpreter can give to the query (when applied to P).*

To summarize, the s-semantics couples constructivity, homogeneity and single structure of the Herbrand semantics with the greatest representative power. Moreover, the presence of variables in the s-models allows the interpretation of the programs to be completely independent from the possible domains and contexts in which programs can be merged (*modularity*).

3.2. Closed World Assumption ([GMN], [P], [S])

Single model semantics may be used to implicitely represent information which, though not formally entailed, can be regarded as being meant by the programs (comp. section 1.1). For positive programs this amounts not only to a quantitative (for finite domains) but also to a qualitative expressive improvement: negative information is introduced.

The least Herbrand model semantics constitutes a classical example of such *non monotone* representation:

EXAMPLE 3.5.

Let us consider the following program P, defining the set of even natural numbers:

$$even(0) \leftarrow,$$

$$even(s^2(x)) \leftarrow even(x).$$

Its Herbrand base is

$$B_P \equiv \{even(0), even(s(0)), ..., even(s^n(0)), ...\}$$

$$= \{even(s^n(0)) : n \in \mathbf{N}\}$$

while its least Herbrand model is

$$M_P \equiv \{even(0), even(s^2(0)), ..., even(s^{2n}(0)), ...\}$$

$$= \{even(s^{2n}(0)) : n \in \mathbf{N}\}.$$

Now,

$$N_P \equiv B_P - M_P = \{even(s(0)), even(s^3(0)), ..., even(s^{2n+1}(0)), ...\}$$

$$= \{even(s^{2n+1}(0)) : n \in \mathbf{N}\}$$

can be consistently (and naturally) assumed to be P's false atoms set. □

In order to be effective, this representation has to be transposed from the declarative level to the operational one. The completeness theorem for Herbrand semantics ensures equivalence between M_P and success set of P. Thus N_P corresponds to the union of P's infinite and finite failure sets. Here a computational problem arises. Both success (R) and finite failure (F) sets of any positive program are semidecidable. Thus, recalling the existence of a positive program which emulates the Universal Turing Machine (section 1.1), the existence of an interpreter complete with respect to the 'M_P-N_P'

semantics would imply that the Halting Problem be decidable.

Anyway, such negative result should not be taken too drastically. First of all, most of infinite computations are recognizable by rather trivial loop-checking algorithms; thence, a reasonable large class of programs can be expected to have a decidable halting problem. Second, if we analize the role of LP with respect to its implementations, we see that the complete theoretical framework is just an idealization of 'real-life' approximations, which sacrifice completeness to computational effectiveness (PROLOG's depth-first selection rule, for instance). Therefore, it makes sense to define an interpreter based on *negation as failure* (not to be confused with *finite* failure) and to claim completeness for it. However, we shall postpone such definition until section 6, when we shall introduce it in a more general context than positive programs, namely *stratified* programs.

The expression in a logical theory of the preference for the least Herbrand model has been accomplished in two different ways. One, *circumscription* will be the subject of section 3.3. The other, *closed world assumption* is what we are going to show here.

The idea behind the closed world assumption is rather trivial: if we explicitely add to a positive program P the negation of all atoms in N_P together with the axioms CET + DCA, the models of the resulting extended program all collapse in the original program's least Herbrand model.

EXAMPLE 3.6.

Referring to the program in the previous example, we would have:

$$CET + DCA,$$

$$even(0) \leftarrow,$$

$$even(s^2(x)) \leftarrow even(x),$$

$$\neg even(s(0)),$$

$$\neg even(s^3(0)),$$

$$\ldots$$

$$\neg even(s^{2n+1}(0)),$$

$$\ldots \qquad \square$$

Notice that, in order to preserve completeness for universal queries, the infinitary DCA in the example above is not replaceable with the finitary WDCA: since for the program above

$$WDCA \equiv [x = 0 \vee \exists y.x = s(y)]$$

there is a model of the program which together with $\mathbb{N}$ has in the domain a copy of $\mathbb{Z}$ in which nothing is even; thus, the query

$$\forall x(even(x) \vee even(s(x)))$$

is not satisfied in such model.

Our definition of *closed world assumption* for a positive progam P will then be:

$$cwa(P) \equiv P + CET + DCA + \{\neg\alpha\colon \alpha \in N_P\}$$

$$= P + CET + DCA + \{\neg\alpha\colon \alpha \text{ ground atom}, M_P \not\models \alpha\}.$$

In order to have a definition to apply also to general programs, since a ground atom is logical consequence of a positive program iff it is true in its least Herbrand model, we rewrite what above as

$$cwa(P) \equiv P + CET + DCA + \{\neg\alpha\colon \alpha \text{ ground atom}, P \not\models \alpha\}.$$

However, while for positive programs it is a solid, though theorical, structure to deduce negative

information, the closed world assumption is for general programs just a fragile framework, veined by inconsistencies.

EXAMPLE 3.7.
The program P: $p \leftarrow \neg q$ is logically equivalent to $p \vee q$. Since $p \vee q$ does not imply p nor q, we have that

$$cwa(P) \equiv CET \cup DCA \cup \{p \vee q\} \cup \{\neg p, \neg q\}$$
$$= CET \cup DCA \cup \{(p \vee q) \wedge \neg(p \vee q)\}$$

—which is clearly inconsistent. □

Notice that SLDNF-resolution does not interpret $p \leftarrow \neg q$ as $p \vee q$: it rather succeeds on the query $\leftarrow p$ and fails on the query $\leftarrow q$. It is thus clear that, when we base the closure of the programs upon the notion of first-order logical consequence, we go too far from the actual behaviour of any —even theorical— interpreter for LP's negation. For positive programs we have equivalence with the least Herbrand model and it is this property which in general gives consistence to cwa:

THEOREM 3.8. *P general program. Then:*

$cwa(P)$ is consistent *iff* there exists the least Herbrand model of P. □

In our example above, we do not have a least model. We rather have two minimal models: $\{p\}$ and $\{q\}$. Recalling the previous section where minimal model semantics has been shown to be a natural generalization of least Herbrand model semantics, it makes sense to reformulate *cwa* in terms of minimal models. We define then the *generalized closed world assumption* ([M]) as:

$$gcwa(P) \equiv P + CET + DCA + \{\neg\alpha: \alpha \text{ ground atom}, H\text{-}MIN(P) \not\models \alpha\}$$

where $H\text{-}MIN(P)$ stands for the set of minimal *Herbrand* models of P. Although by definition gcwa is always consistent, we still have not found a completely satisfactory closure, as the following example shows:

EXAMPLE 3.9.
To P as in the previous example, gcwa does not add any information. Indeed,

$$H\text{-}MIN(P) = \{p\} \cup \{q\}$$

—the whole Herbrand base of P. This is in contrast with SLDNF-resolution, which, as already mentioned, treats q as false atom. □

In order to find a satisfactory form of closure for general programs, we should then look for something in between cwa and gcwa, operating a restriction to a certain class of minimal models (comp. principle (b) of section 1.2). Once more, from the program in Example 3.7 we get the right idea: the 'proper' model for the SLDNF-computations of P is $\{p, \neg q\}$ —the closure based on the minimal model $\{p\}$.

But let us resume here our discussion. Again, we shall return to it in connection with *stratified* programs —a class which is semantically characterized by a *preference* for a particular minimal model, the so-called *perfect* model. By the way, $p \leftarrow \neg q$ is a stratified program.

3.3. Circumscription ([L], [L1], [R])

Since the set of ground atoms which are not derivable from a logic program can be infinite, every closure which, like *cwa*, is based on the *explicit* enumeration of all negative (ground) information suffers a major syntactical shortcoming: *non-finite axiomatizability*. In order to avoid such difficulty,

while preserving the semantical idea behind *cwa*, we have to 'capture' the negative information *implicitely*, axiomatizing the notion of minimality. Indeed, we have seen that, when consistent, *cwa* has a single model – the least Herbrand one. Now, since such model is uniquely defined by assigning to each predicate the minimal of its possible extensions, we can achieve the same effect of *cwa* by 'circumscribing' to its narrowest domain each predicate appearing in a program. Formally, if

$$\boldsymbol{p} \equiv p_1, \ldots, p_n$$

are all the predicate symbols appearing in a program P, we call *circumscription* of P the following *second order formula*:

$$circ(P) \equiv P(\boldsymbol{p}) \wedge \forall \boldsymbol{p}' [P(\boldsymbol{p}') \rightarrow \boldsymbol{p} \leqslant \boldsymbol{p}']$$

where $\boldsymbol{p}' \equiv p_1', \ldots, p_n'$ are predicate variables (recall we are in second order) corresponding for arity to $p_1, \ldots, p_n$, and where

$$\boldsymbol{p} \leqslant \boldsymbol{p}' \equiv \bigwedge_{i=1}^{n} \forall \boldsymbol{x} [p_i(\boldsymbol{x}) \rightarrow p_i'(\boldsymbol{x})],$$

for $\boldsymbol{x}$ opportune tuple of (object) variables. What $circ(P)$ literally states is that: *the set of predicate constants* $\boldsymbol{p}$ *has the property* P and *there is no set of predicates satisfying* P *which is 'smaller' than* $\boldsymbol{p}$.

By definition, a program's least Herbrand model uniquely defines the least extension which any set of predicates must have in order to satisfy the program. Thus, with respect to Herbrand pre-interpretations, we have the desired equivalence:

THEOREM 3.10. *If cwa(P) is consistent, then*

$$cwa(P) \Leftrightarrow circ(P) + CET + DCA. \quad \square$$

Since *circumscription* refers to minimality it would be natural to expect equivalence also with *gcwa*. However, this is not the case, as the following example shows:

EXAMPLE 3.11.

The program P:

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

– logically equivalent to $p(a) \vee p(b)$ – has two minimal models: $\{p(a)\}$ and $\{p(b)\}$. $GCWA(P)$ does not bring new information to P, so that the whole Herbrand base of P $-\{p(a), p(b)\}-$ is a model for $gcwa(P)$, too. On the other hand, $circ(P)$ is equivalent to

$$\forall x [p(x) \leftrightarrow x = a] \vee \forall x [p(x) \leftrightarrow x = b]$$

which has for Herbrand models $\{p(a)\}$ and $\{p(b)\}$, but not $\{p(a), p(b)\}$. $\square$

The program in the example above is very similar to $p \leftarrow \neg q$, the program in Example 3.7, and the same consideration as there still holds here: of all the minimal models, only one is the intended. In order to conclude that $\{p(a)\}$ rather than $\{p(b)\}$ is the proper model for the program, we have to reflect on the position of the atoms in the program. In other words, *global* considerations about the program are required. Instead, as defined above, *circumscription* is performed in a *local* manner, each predicate being minimized independently from the others. To overcome this problem, more contextual form of circumscription, *pointwise* and *prioritized circumscriptions*, have been introduced. We shall present them in section 5.3, after having defined the stratified programs.

The previous example also suggests that circumscription be in some cases expressable by a first order formula – a necessary condition, if we are looking for implementations. Unfortunately, the class of programs which enjoys such property is too narrow to be significative. However, the procedure by which most of computable circumscriptions are performed can be successfully generalized

to define a milder form of closure which is tightly related to SLDNF-resolution. It is thus worth to exhibit some significative examples of computable circumscription. We shall treat positive programs only.

i) When a predicate symbol, say p, appears in the body of clauses only, its definition in the program is empty. Thus, under assumption of minimality, its extension will be empty as well. The predicate p is then circumscribed as:

$$\forall x.\neg p(x).$$

ii) A clause like $p(x) \leftarrow q(x)$, for p and q with same arity, states that the extension of q is contained or equal to the one of p. Thus, if that is the only clause defining p, we can circumscribe it as

$$\forall x.p(x) \leftrightarrow q(x)$$

and then substitute q in the uses of p. Notice that such circumscription is correct only if the predicates involved do not *depend* recursively on each other. (On the notion of dependence we shall return later; at the moment intuition should suffice.)

iii) Ground facts can be regarded as special case of implicative clauses. For instance,

$$p(a) \leftarrow$$

is equivalent to

$$p(x) \leftarrow x = a.$$

Actually, "$=$" has arity two, but here we can consider it as monadic predicate "$=a$". Using λ-notation:

$$p(x) \leftarrow \lambda x.x = a.$$

For conjunctions of ground facts we can proceed analogously:

$$p(a) \leftarrow,$$

$$p(b) \leftarrow,$$

equivalent to

$$p(x) \leftarrow \lambda x.(x = a \vee x = b),$$

is then circumscribed as

$$\forall x\ [p(x) \leftrightarrow x = a \vee x = b].$$

3.4. Completion ([A], [K1], [S])

Peculiar to the computable circumscriptions of the previous section is the transformation of sufficient conditions in *necessary* and sufficient, accomplished simply by substituting implications with equivalences. In order to generalize such 'IFF-transformation' to all programs, we have to depart from circumscription, replacing *minimality* with *necessity* as closure's basic principle. Once more, let us proceed with examples.

i) More rules to define a predicate:

$$p(a) \leftarrow q\ ,\ p(b) \leftarrow r \quad \text{become} \quad \forall x\ [\ p(x) \leftrightarrow (x = a \wedge q) \vee (x = b \wedge r)].$$

Notice that we did not assume anymore that predicates in head and body have the same arity.

ii) Variables appearing in the body of clauses:

$$p \leftarrow q(x) \text{ becomes } p \leftrightarrow \exists x.q(x)$$

(recall our basic principle: *necessity*).

iii) Function symbols in non ground heads:

$$p(f(y)) \leftarrow \text{ becomes } \forall x\,[p(x) \leftrightarrow \exists y(x = f(y))].$$

iv) More atoms in the body:

$$p \leftarrow q,r \text{ becomes } p \leftrightarrow q \wedge r.$$

v) Recursion (a classical example):

$$p(s(y)) \leftarrow p(y)\,,\ p(0) \leftarrow \text{ become } \forall x\,[p(x) \leftrightarrow \exists y(x = s(y) \wedge p(y)) \vee x = 0].$$

We are now able to formalize. Let us indicate with Def_P the definition of a predicate p in a program

$$Def_P \equiv \bigwedge_{i=1}^{k} p(t_i) \leftarrow C_i$$

(where if k = 0 then $Def_P = \varnothing$ and where t_i stands for the tuple of terms $t_{i1},\ldots,t_{in}$ (n being the arity of p); C_i stands for a conjunction of atoms).
Then the following two rules suffice to generalize all previous examples:

$$IFF^- \quad \frac{Def_P = \varnothing}{\forall x.\ \neg p(x)}$$

$$IFF^+ \quad \frac{Def_P \equiv \bigwedge_{i=1}^{k} p(t_i) \leftarrow C_i \neq \varnothing}{\forall x\,[p(x) \leftrightarrow \bigvee_{i=1}^{k} \exists(x = t_i \wedge C_i)]}$$

(where x is a tuple of variables and $x = t_i$ stands for $x_1 = t_{i1},\ldots,x_n = t_{in}$).
Of course, "=" is again intended to be supported by CET.

By all this, we define the *completion* of a program P ([Cl]) as:

$$comp(P) \equiv IFF(P) + CET$$

(where IFF(P) is the result of applying rules IFF^- and IFF^+ to all predicate symbols of P).

As already mentioned (section 3.2), from a computational point of view the complement of the success set of a program (failure set) is of scarce interest. It is rather the *dual* of the success set, the finite failure set, which is significant. The semantical import of completion is in that it is the only closure which suits a complete declarative characterization of finite failure. Medium for such characterization is the immediate consequence operator.

For the success set (P positive program, A ground atom) we have:

α) *A is in the success set of P* $\Leftrightarrow$ $A \in T_P \uparrow \omega$.

β) *I based on J is a model of $P + CET$* $\Leftrightarrow$ $T_P^J(I) \subseteq I$.

γ) $A \in T_P \uparrow \omega \Leftrightarrow P \vdash A$.

Following **duality** we put

$$T_P^J\downarrow 0 = B_P^J$$

$$T_P^J\downarrow(n+1) = T_P^J(T_P^J\downarrow n)$$

$$T_P^J\downarrow\omega = \bigcap_{n<\omega} T_P^J\downarrow n.$$

Since the intersection with the Herbrand base of any $T_P^J\downarrow\omega$ is equivalent to $T_P\downarrow\omega$, the chain above can be reformulated in terms of finite failure and completion as:

α') *A is in the finite failure set of P* $\Leftrightarrow$ $A \notin T_P\downarrow\omega$.

β') *I based on J is a model of comp(P)* $\Leftrightarrow$ $T_P^J(I) = I$.

γ') $A \notin T_P\downarrow\omega$ $\Leftrightarrow$ $comp(P) \models \neg A$.

This should not induce to believe in a total duality between $T_P\uparrow\omega$ and $T_P\downarrow\omega$. In fact, $T_P\downarrow\omega$ is not equivalent to the greatest fixpoint of T_P:

EXAMPLE 3.12.
Let P be

$$p(f(x)) \leftarrow p(x),$$

$$q(a) \leftarrow p(x).$$

Then

$$T_P\downarrow\omega = \{q(a)\}$$

while

$$T_P(T_P\downarrow\omega) = T_P(\{q(a)\}) = \varnothing = gfp(T_P). \quad \square$$

Despite of this gap, the chain α'-β'-γ' is not broken since every non finitely failed SLD-computation (of ground positive queries) gives somehow rise to an interpretation which can be extended to a model for comp(P).

Though completeness of negation as finite failure is a very important result, it should be noticed that it does not imply completeness of SLDNF-resolution, even for positive programs only. The following example clarifies the problem:

EXAMPLE 3.13.
For P:

$$r(a) \leftarrow,$$

$$r(b) \leftarrow r(b),$$

$$r(b) \leftarrow q(a),$$

$$q(a) \leftarrow q(a)$$

the query

$$\leftarrow r(x), \neg q(x),$$

though it leads to infinite computations only, is logical consequence of comp(P). Indeed, for every pre-interpretation J model of CET and every model M of comp(P) based on J, $T_P^J(M) = M$. Thus

$$r(a) \in M$$

$$q(a) \in M \rightarrow r(b) \in M$$

$$q(b) \notin M.$$

Then either $r(a) \wedge \neg q(a)$ or $r(b) \wedge \neg q(b)$ is logical consequence of M, which means

$$comp(P) \models \exists x(r(x) \wedge \neg q(x)). \quad \square$$

If for positive programs completeness seems anyway achievable for a restricted class of queries (comp. section 4.7), for general programs there are much serious flaws in the theory, which often produce inconsistencies:

EXAMPLE 3.14.
The program $p \leftarrow \neg p$ (logically equivalent to an 'innoquous' sentence like $p \vee p$) yields the inconsistent completion:

$$p \leftrightarrow \neg p. \quad \square$$

There are two solutions to this problem. One is simply to forbid programs whose completion is inconsistent. This amounts to define opportune classes of programs and it is the argument of next chapter. Another one, more radical, is to modify the notion of model (principle (b') of section 1.2) in order to overcome inconsistence at its very root. *Three valued models* are those which best suit to this purpose.

The core of such approach is Φ_P^J – the 3-valued extension of the immediate consequence operator. T_P^J has to be abandoned because not anymore monotonic when P is a general program (no semantics could be based on its iterations):

EXAMPLE 3.15.
Let P be

$$p \leftarrow \neg p, \neg q;$$

Then

$$T_P(\varnothing) = \{p\}$$

and

$$T_P(\{p\}) = \varnothing. \quad \square$$

The problem is that T_P^J is defined upon interpretations stated in terms of true atoms only, leaving the false ones *implicitely* defined by the difference with the base B_P^J. For general programs, this is a too weak representation: false atoms, as *explicitely* partecipating to computations, are to be *explicitely* mentioned, too. When we adopt such other representation of interpretations, we can assign – in general – truth values only to a subset of the whole base. The rest can be seen as *unknown* or *undefined* and we can associate to it a *third* truth value (**u**) which intuitively corresponds to infinite computations. Notice that these 3-*valued* interpretations can alternatively be seen as 2-valued *partial* interpretations.

A partial interpretation I based on J can be uniquely defined by two sets of J-ground atoms I_T and I_F, respectively representing true and false atoms in I. We shall then use the following notation:

$$I = (I_T, I_F)$$

and

$$(I_T', I_F') = \Phi_P^J(I_T, I_F)$$

Thus, to define Φ_P^J it suffices to define I_T' and I_F'. Now, since I_T' is the set of J-ground atoms which are immediate consequence of P with respect to I, we have just to report T_P^J's definition. That is, for

each $\alpha \in B_P^J$

$\alpha \in I_T'$ *iff* $\exists\ \sigma$ state over J and

$\exists\ A \leftarrow F$ clause of P such that

$\alpha = A\sigma$ and $I \models_\sigma F$.

On the other hand, for I_F' we can directly make use of rules F_1^+ and F_2^+ used in the definition of SLDNF-resolution (section 2.1). We have then that, for each $\alpha \in B_P^J$

$\alpha \in I_F'$ *iff* $\forall\ \sigma$ state over J and

$\forall\ A \leftarrow F$ clause of P

either $\alpha \neq A\sigma$ or $I \models_\sigma \neg F$.

Truth values in such logic are determined by following rules (SHEPHERDSON [S]):

a) Every connective other than '$\leftrightarrow$' has value **t** (**f**) if it has that value in ordinary 2-valued logic for all possible replacements of **u**'s by **t** or **f**; otherwise it has value **u**.

b) Quantifiers are treated like infinite conjunctions ($\forall$) or disjunctions ($\exists$); therefore, $\forall x.\phi(x)$ is **t** if for all a $\phi(a)$ *is* **t**, it is **f** if there exists an a such that $\phi(a)$ *is* **f** and it is **u** otherwise.

c) $\phi \leftrightarrow \psi$ has value **t** if ϕ and ψ have the same truth value, **f** otherwise.

Rule (c) avoids that formulas like $p \leftrightarrow p$ be interpreted as **u** when p is **u**.

With such rules the program $p \leftrightarrow \neg p$ has a 3-valued model, which assings to p (and thus to $\neg p$) truth value **u**. More in general, *every general program admits a 3-valued model for its completion.*

Φ_P^J inherits from T_P^J most of its properties, like, for instance, models as fixpoints:

THEOREM 3.16. *P general program.* $I = (I_T, I_F)$ *partial interpretation based on J. Then*

I is a (3-valued) model of comp(P) iff $\Phi_P^J(I) = I$. □

But also like the lattice structure (based now on the semilattice $\mathbf{u} < \mathbf{t}$, $\mathbf{u} < \mathbf{f}$), which, together with the monotonicity of Φ_P^J, enables to characterize for each J the least of these fixpoints iterating the operator from the bottom ($\varnothing$). That is, for every program P, there exists an ordinal α for which

$$\Phi_P^J \uparrow \alpha = lfp(\Phi_P^J).$$

Unfortunately, α is in general greater than ω: if we analize the definition of Φ_P^J, we see that for every positive program P

$$\Phi_P^J \uparrow \alpha = (T_P^J \uparrow \alpha,\ B_P^J - T_P^J \downarrow \alpha).$$

Therefore, recalling Example 3.12 even under the very strong restriction to positive programs and Herbrand pre-interpretations,

$$\Phi_P \uparrow \omega \neq lfp(\Phi_P).$$

This loss of compactness shows that completeness cannot be reached via fixpoints of Φ_P^J. However, $\Phi_P \uparrow \omega$ —even though it is not a fixpoint— is enough representative for the notion of logical consequence of comp(P):

THEOREM 3.17. ([K]) *Let* ϕ *be a query for a general program P. Then:*

$comp(P) \models_3 \phi$ *iff* ϕ is **t** in $\Phi_P^* \uparrow n$. □

When marked with "*" Φ_P is intended to be based on a language which has infinite predicate and function symbols for every arity (comp. section 3.1).

Notice that none of the $\Phi_P^* \uparrow n$ nor $\Phi_P^* \uparrow \omega$ need to be a model for comp(P).

4. Classes

4.1. Allowedness ([A], [Ch], [K1])

When SLDNF-resolution has to evaluate a (sub-) query in which all literals are both negative and non-ground it does not proceed any further: it flounders (see section 2.1). This in order to ensure effectiveness to the interpreter. Indeed, while positive literals in the body of clauses are (implicitely) existentially quantified, negative literals transform variables implicit quantification from existential to universal, requiring the whole universe to be inspected. When variables in negative literals are not processed (like in SLDNF-resolution) derivations are always downward directed – converging, cycling or floundering but never diverging.

Declaratively, such resources' limitation hasn't any equivalent:

Example 4.1.

Let P be

$$p(a) \leftarrow p(a),$$

$$q(b) \leftarrow .$$

Then $comp(P) \models \neg p(b)$ and thus $comp(P) \models \exists x. \neg p(x)$. But SLDNF-resolution, though succeding on $\leftarrow \neg p(b)$, flounders on $\leftarrow \neg p(x)$. □

Thence, completeness may be stated in terms of completion only for those programs and queries which do not flounder. Unfortunately, the class of non-floundered programs + queries (it does not make sense to refer just to programs or just to queries) happens to be undecidable. Indeed, the existence of a positive program which emulates the Universal Turing Machine relates floundering to the halting problem. It suffices to add to each query for such program the literal $\neg p_{new}(x_{new})$, with both predicate and variable not appearing in the program nor in the query. Since both program and queries are positive the related SLDNF-computation will not flounder iff it is infinite.

A decidable class of programs and queries which is free from floundering is the so-called *allowed* one. It is based on the consideration that variables appearing in negative literals of a query ought to appear also in some of its positive literals, otherwise they can not be grounded during computations. Recursively, also subqueries produced by matching with program clauses have to respect the same condition. Formally, for a general program P and a general query ϕ, we say that ϕ is *allowed* iff all its variables appear in some of its positive literals, and that P is *allowed* iff every variable of a clause in P appears in some positive literals in the body of that clause. The second definition implies that unit clauses (facts) in an allowed program are always ground. Thence, all answer substitutions given from that program to allowed queries will be ground. A too strong restriction, indeed. Current research is devoted to define more significant decidable and non-floundered classes.

A different approach to the problem is to be found in Chan [Ch], who suggests to adopt a more '*constructive*' approach to negation as finite failure, so to allow (a form of) bounding between variables to take place also in negative literals. More information is gathered and floundering never occurs (comp. principle (a) of section 1.2). Some examples of constructive negation:

Example 4.2.

The program

$$p(a) \leftarrow,$$

under constructive negation, answers to the query

$$\leftarrow \neg p(x)$$

with

$[x \neq a]$. □

EXAMPLE 4.3.

$p \leftarrow \neg q(x)$

$q(f(x)) \leftarrow q(x)$

$q(0) \leftarrow$

$\leftarrow p$

true. □

EXAMPLE 4.4.

Replacing the first clause in Example 2 with

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

we have

$[x \neq 0,\ x \neq f(0),\ x \neq f^2(0),\ \ldots]$.

('diverging'). □

EXAMPLE 4.5.

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

$p(x,x) \leftarrow$

$\leftarrow q(x,y,z)$

$[x \neq y]$. □

Such 'default' treatment of the variables in negative literals is in line with the epistemological considerations of section 1.1.

The problem with constructive negation is that it may lead to diverging computations – that is, what floundering was meant to avoid. However, constructive negation **subsumes** classical negation as (finite) failure so that we can restrict to the same classes of non-floundering programs and queries, but also, and more significantly, we may try to define classes of programs and queries which do not diverge under constructive negation. This is an area for new research and virtually everything has still to be done.

In TURI [T] we make use of constructive negation to extend s-interpretations to general programs modelling. Here we proceed with SLDNF-resolution and allowedness. In such terms, a completeness theorem for general programs is possible, when we consider last section's 3-valued logic. From Theorem 3.17 we have:

THEOREM 4.6. ([K1]) *Let P a general program and ϕ a general query. If P and ϕ are allowed then*

$comp(P) \models_3 \forall \phi\sigma \Rightarrow \phi R \sigma$

$comp(P) \models_3 \neg\exists\phi \Rightarrow \phi \in F$. □

4.2. Global Dependencies ([A], [K1])

The completeness result of theorem 4.6 is *partial* not only because stated in terms of 3-valued models, but also because it can be seen as starting point to establish a 2-valued (*total*) completeness result. To this purpose it suffices to look for classes of programs with consistent completion (i.e. with at least one 2-valued model for their completion) and then for classes of queries which do not violate such consistence. This methodology is due to KUNEN [K1] and we believe it is an enlightening approach to the semantics of logic programs with negation.

In order to be of any interest for the programmer, a class of programs should be easy to define and to recognize. The syntactical structure – the *form*– should suffice to determine whether a program belongs to a class or not. Now, the *form* of a program is related to the *consistence* of its completion by the way (ground) atoms *depend on* each other in the program. We shall call this dependence *local* as opposed to the dependence between predicate symbols, which we name *global*. The analysis of the latter leads to a smaller class of programs than the former does, but the results at the global level extend very naturally at the local one. Thence, we shall concentrate on global dependencies.

To formalize the notion of global dependency we may make use of graphs. Every program P defines a directed and signed graph (*dependency graph*) in the following way: a) the nodes of the graph are the predicate symbols appearing in P; b) the arcs of the graph are obtained reversing the arrows (implications) of the clauses in P; c) the signs are determined from the polarity of literals in the clauses' body.

EXAMPLE 4.7.
A clause like

$$p(t) \leftarrow \ldots, q(s), \ldots$$

corresponds to a *positive* arc from p to q, while

$$p(t) \leftarrow \ldots, \neg q(s), \ldots$$

to a *negative* one. □

We shall say that p *depends positively* (resp. *negatively*) on q iff in the dependency graph there exists a path from p to q that contain an even (resp. odd) number of negative arcs. To symbolize such positive (resp. negative) dependence we put $p \geqslant_{+1} q$ (resp. $p \geqslant_{-1} q$). We consider *zero* even, thus for all p: $p \geqslant_{+1} p$. Independently from the sign, the relation "to depend on" induces an equivalence relation on the predicates: for all p and q we have $p \approx q$ *iff* $p \geqslant q \wedge q \geqslant p$.

4.3. Hierarchical Programs ([Ca])

The easiest way to ensure consistence to general programs is to forbid the use of recursion. Actually, such restriction does even more: all models are two valued. This is due to the particular structure that the lack of recursion gives to programs and relative completions. No predicate depends on itself, thus the dependency graph is free from cycles and levels can be associated to it (recursion-free programs are usually named *hierarchical*). At the first level are those predicates not depending on any other, at the second those depending only on predicates from the first level, and so on. Correspondly, levels can be introduced in the programs and in their completion.

EXAMPLE 4.8.
A hierarchical program:
first level $r \leftarrow$,
second l. $q \leftarrow r, \neg s$,
third l. $p \leftarrow s$, $p \leftarrow \neg q$.
Its completion:
first l. r , $\neg s$,

second l. $q \leftrightarrow r \wedge \neg s$,
third l. $p \leftrightarrow \neg q \vee s$. □

The models of any hierarchical completed program can be determined ascending those levels. For instance, in the example above r must be true and $\neg r$ false, thus q must be true and p false. As the example suggests, propositional hierarchical programs have just one model for their completion. More in general, there exists a single Herbrand model for the completion of a hierarchical program.

From a computational point of view, a hierarchical program is always terminating because recursion is necessary condition for a program to cycle. But the price to be paid for these nice properties of totality and termination is in terms of expressive power (positive programs are not hierarchical, as well as most significant programs). When local in place of global dependencies are taken into consideration termination is not guaranteed anymore (because the number of levels may be ω or even exceed it), but the expressive power results considerably increased.

In section 4.7 we show how hierarchical programs play a role in the definition of an important class of queries.

4.4. Stratification ([A], [ABW])

Recursion in itself is not necessarily a threat for consistence. Most significant positive programs are recursively defined and still they are consistent under completion. It is recursion through negation which introduces problems (comp. Example 3.14). When avoided, consistence is restablished, without sacrificing too much expressive power. The resulting programs are called *stratified*: since in the dependency graph there is no cycle containing a negative arc, levels can be associated to it considering negative arcs only. A partition into *strata* is thus induced on the programs.

EXAMPLE 4.9.

A stratified program:
first stratum $r \leftarrow$,
second s. $q \leftarrow r, \neg s$,
third s. $p \leftarrow p$, $p \leftarrow \neg q$. □

Every *stratum* is a subprogram whose predicates appearing in negative literals are defined in some strictly lower stratum. The first stratum is thus a positive program (eventually empty). Next strata can be regarded as positive programs which may use negative information concerning some previously defined programs.

Procedurally, stratification is best interpreted as non-recursively nested positive LP together with negation as failure: queries and subqueries may contain negative literals, but SLD-resolution and negation as failure are never mixed, for strata can not be ascended until the selected literal in the (sub-) query has been resolved.

For their structure, stratified programs inherit most of the properties enjoyed by positive programs − consistence under completion among them − coming closer than any other known class to the ideal outlined in the introduction. Next chapters are devoted to exploit such relationship, both at a declarative and at a procedural level. But in the remaining sections of this chapter we proceed with the analysis of the completed program semantics.

4.5. Call-Consistent Programs ([K1])

There are several possibilities other than stratification to structure programs in a consistent way. For instance, the program $p \leftarrow \neg q$, $q \leftarrow \neg p$ is not stratified but still its completion admits some 2-valued models ($\{p\}$ and $\{q\}$). To generalize such example we introduce the class of *call-consistent* programs, which extends stratification allowing recursion through negation when no predicate

depends negatively on itself; that is, when

$$\neg\exists p.\ p \geqslant_{-1} p.$$

This can be stated equivalently as

$$\neg\exists p,q.\ p\approx q,\ p \geqslant_{+1} q,\ p \geqslant_{-1} q,$$

which is easier to compare with stratification ($\neg\exists p,q.\ p\approx q,\ p \geqslant_{-1} q$).

Call-consistent programs inherit the 'layered' structure from the stratified ones, since the equivalence classes form a partial order which induces levels on the program. Ascending these levels, we shall be able to prove that call-consistent programs always admit a total model for their completion. First, notice that, although in a level there can be predicates from more than one class, it is not reductive to consider each level as made up of predicates from a single equivalence class. If a call-consistent program has a single equivalence class a *signing* S can be associated to it, that is, every predicate can be labelled with $+1$ or -1 in the following way:

$$\forall p,q.\ \ p \geqslant_i q \ \Rightarrow\ S(q)=S(p)\cdot i$$

(with i equal to $+1$ or -1). The hypotheses above (call-consistence and single class) ensure that no contradiction between labels may arise; indeed, by contraposition:

$$\exists p.\ S(p)=-S(p) \ \Rightarrow\ \exists p_1,p_2 \text{ s.t. } p_1 \geqslant_i p\ ,\ p_2 \geqslant_j p \text{ and } S(p_1)\cdot i=-S(p_2)\cdot j$$
$$\Rightarrow\ p \geqslant_{-1} p \quad (\text{by } p\approx p_1\approx p_2).$$

In general, there are more signings for a program. We may choose any of them and then associate truth value **t** to the predicates marked by $+1$ and **f** to those marked by -1. The result is a 2-valued model for the completed program.

When a program has more than just one equivalence class, thus more than one level, we can iterate the method above in the following way:

i) extend as far as possible on the current level P_j the model for the underlying levels $P_1,\ldots,P_{j-1}$;
ii) at any place a predicate of P_j is still undefined assign the truth value associated to it by the (arbitrarily chosen, but fixed) signing for P_j.

By all this, we have proved that *any call-consistent program has a consistent completion.*

Call-consistence is a structure with great expressive power, but it does not mark the 'frontier'. A program may have a predicate negatively depending on itself but still be consistent under completion. An instance is:

$$p \leftarrow \neg p,\ q\ ,$$

whose completion

$$p \leftrightarrow \neg p \wedge q\ ,$$
$$\neg q$$

has a 2-valued model (both p and q false). Anyway, notice that such program is logically equivalent to

$$p \vee p \leftarrow q$$

thus it can be *normalized* to

$$p \leftarrow q$$

which is both hierarchical and positive and whose completion has still the same 2-valued model. As far as we know, normalization has not yet been taken into account to enlarge the classes of general programs consistent under completion (although an example similar to the one above is to be found

in VAN GELDER, ROSS AND SCHLIPF [VGRS]). Here is a less trivial example, where normalization does not apply:

EXAMPLE 4.10. ([VGRS])
The program P:

$$p \leftarrow q, \neg r \ ,$$

$$p \leftarrow r, \neg s \ ,$$

$$r \leftarrow q \ ,$$

$$q \leftarrow p$$

is not call-consistent, since $p \approx r$ (via q), $p \geqslant_{+1} r$, $p \geqslant_{-1} r$, but its completion admits two 2-valued models. In one, s is false and the rest true, in the other $-$the intended$-$ everything is false. In section 5.4 we show a fixpoint construction based on an operator extending Φ_P, which is able to produce the desired model in this and similar cases. □

4.6. Local Dependencies ([Ca], [P])
The instantiation of the clauses of a program P with all its Herbrand universe elements is called *ground*(P). As example, take the program P:

$$even(0) \leftarrow \ ,$$

$$even(s(x)) \leftarrow \neg even(x) \ ;$$

then *ground*(P) is the following infinite program:

$$even(0) \leftarrow \ ,$$

$$even(s(0)) \leftarrow \neg even(0) \ ,$$

$$even(s^2(0)) \leftarrow \neg even(s(0)) \ ,$$

$$\ldots \ .$$

Distinct ground atoms can be treated as distinct propositional symbols. The dependencies between such propositions in the '*grounded*' program are those *local dependencies* we mentioned in section 4.2.

Local dependencies define larger classes of programs than global do. For instance, although the program P above is not hierarchical (nor stratified, neither call-consistent), ground(P) has a hierarchical form. (We say then that P is *locally hierarchical*, and in general when the name of a class is preceded by 'locally' it is intended to be determined by grounded programs.)

The enhanced expressiveness provided by 'localization' is to be paid in terms of decidability. Since the number of levels is not anymore finite and may even exceed ω ($q \leftarrow p(x)$, $p(f(x)) \leftarrow p(x)$) it is not trivial to decide whether there is a cycle in the dependency graph (of the grounded program). Indeed, it has been proved that both locally hierarchical and locally stratified programs do not constitute decidable classes. Another non decidable class is that of locally hierarchical programs whose number of levels is not greater than ω (*acyclic programs*). Several nice properties of hierarchical programs $-$like termination on ground queries and single Herbrand model for the completion$-$ are then combined to high expressive power (see BEZEM AND APT [BA] for acyclic programs and BEZEM [B] for decidability and further results on acyclic programs).

4.7. Strictness ([A], [K1])
Our interest for programs consistent under completion is motivated by our purpose of finding conditions for equivalence to hold between 2-valued and 3-valued logical consequences of completion.

However, although necessary, consistence does not suffice, because it does not ensure that queries which are 2-valued consequences be 3-valued consequences as well. Program and query of Example 3.13 illustrate the problem: with respect to the 3-valued model of comp(P) in which only $r(a)$ is true and the rest is unknown the query is not true anymore. The program is positive, thus consistent, but the query depends both positively (via $r(x)$) and negatively (via $\neg q(x)$) on the predicate q in the program – dependencies are crucial once more. In general, given a query ϕ for a program P and a predicate q in P, we put

$$\phi \geqslant_{+1} q,\ \phi \geqslant_{-1} q$$

if there are two predicates p_1 and p_2 (eventually identical) in ϕ such that

$$p_1 \geqslant_{+1} q,\ p_2 \geqslant_{-1} q$$

When

$$\neg\exists q.\ \ \phi \geqslant_{+1} q,\ \phi \geqslant_{-1} q$$

we say that ϕ is *strict* with respect to P. Strictness is the equivalent for queries of call-consistence for programs: it ensures a signing for the subprogram defined by the dependencies of ϕ. Consequently, *if a program P is call-consistent and a query ϕ is strict with respect to it, then*

$$comp(P) \models_2 \forall\phi \ \ \textit{iff} \ \ comp(P) \models_3 \forall\phi. \tag{1}$$

The proof is analogous to that for consistence shown in section 4.5. Call-consistence is used to complete the partial model which arises from the signing induced by the query.

Actually, we do not need strictness to hold for all predicates. It suffices that those predicates violating strictness define a hierarchical subprogram. Then all models of such subprogram are 2-valued and (1) still holds.

5. Models II

5.1. Minimal Model Semantics for Stratified Programs ([A], [ABW])

In this chapter we show how most of the declarative features of positive programs are to be found in stratified programs, too. Every consideration is to be extended to locally stratified programs as well, but taking care of the consequences that a non finite amount of levels may arise. When not otherwise specified, we assume an arbitrary but fixed program P stratified by $P_1 \cup ... \cup P_n$. We denote $P_1 \cup ... \cup P_i$ $(i \leqslant n)$ by P_i.

Cardinal to the declarative semantics of positive programs is the iteration of the immediate consequence operator up to the least Herbrand model of the program. This is also a fixpoint of the operator and thus a model for the completed program. Somehow, the same holds for stratified programs, but iterating the operator we have to take into account partition into strata and non-monotonicity. Let us consider the stratified program $P = P_1 \cup P_2$, with P_1:

$$r \leftarrow ,$$

$$q \leftarrow r ,$$

and P_2:

$$p \leftarrow \neg q ,$$

$$p \leftarrow p .$$

While its intended model is $\{r,q\}$, iterating T_P from the empty set we obtain a fixpoint ($\{r,q,p\}$) other than the intended one and anyway not a minimal model. To obtain $\{r,q\}$ we have rather to split T_P

in T_{P_1} and T_{P_2}, calculate the least fixpoint of T_{P_1} ($\{r,q\}$) and then iterate T_{P_2} on it in a 'cumulative' manner (in order to not loose previously derived information, since each T_{P_i} refers only to the stratum P_i).

The *cumulative powers* of an operator T are defined as:

$$T\Uparrow 0(I)=I$$

$$T\Uparrow(k+1)(I)=T(T\Uparrow k(I))\cup T\Uparrow k(I)$$

$$T\Uparrow\omega(I)=\bigcup_{k<\omega} T\Uparrow k(I).$$

If T is monotonic then $T\uparrow k \subseteq T\uparrow(k+1)$ and the definition is equivalent to that of powers. It is for non-monotonic operators that such definition is relevant. In the example above, $T_{P_2}\Uparrow 1(\{r,q\})=\{r,q\}$ while $T_{P_2}\uparrow 1(\{r,q\})=\varnothing$ and $T_{P_2}\uparrow 2(\{r,q\})=\{p\}$.

Making use of cumulative powers we can give a general construction for the *intended* model of a stratified program:

$$M_1=T_{P_1}\Uparrow\omega(\varnothing)$$

$$M_2=T_{P_2}\Uparrow\omega(M_1)$$

$$\ldots$$

$$M_n=T_{P_n}\Uparrow\omega(M_{n-1}).$$

$M_P=M_n$ is fixpoint for T_P (thus model for comp(P)) and minimal model for the program. But the analogy with positive programs goes further: M_P is also characterizable as intersection of a certain class of models for the program (comp. principle (b) of section 1.2), namely for

$$M(P_1)=\cap\{M: T_{P_1}(M)=M\}$$

$$M(P_2)=\cap\{M: T_{P_2}(M)=M,\ M\cap B_{P_1}=M(P_1)\}$$

$$\ldots$$

$$M(P_n)=\cap\{M: T_{P_n}(M)=M,\ M\cap B_{P_{n-1}}=M(P_{n-1})\}.$$

we have

$$M_P=M(P_n).$$

Here a major point. We have a class of *general* programs whose entire set of models is too heterogeneous to represent the intended meaning, but which has a homogeneous subset of models whose intersection is 'representative', constructive and natural.

5.2. Perfect Models ([A], [P], [P1])

If, how it seems, the two semantical constructions above were really depending on the program's strata, our claim that 'M_P' is the intended model would not hold, since, in general, stratified programs admit more than a single stratification. To show that this is not the case we introduce the notion of perfect model.

The dependency graph of a stratified program induces a well founded ordering on its Herbrand base, independently from the stratification under consideration. Indeed, for any two ground atoms A and B in B_P we can put $A>B$ whenever in the dependency graph there is a path from the predicate symbol in A to that in B which contains a negative arc.

In turn, *any* well founded ordering on the Herbrand base introduces a *preference* among Herbrand interpretations. We can indeed compare two interpretations M and N for a program, preferring M to N if for each atom of M which does not belong to N there exists a lower (w.r.t. the ordering) atom

which is in N but not in M. For instance, if we consider the stratified program $p \leftarrow \neg q$ together with the 'canonical' ordering of its atoms, we have that of the two minimal models $\{p\}$ and $\{q\}$ we prefer $\{p\}$, since $p > q$. Notice that such notion of preference is by no way quantitative: for the program $p_1 \leftarrow \neg q, \ldots, p_n \leftarrow \neg q$ we still prefer $\{p_1,\ldots,p_n\}$ to $\{q\}$.

We can now define a model *perfect* if there is no other model preferable to it. Perfect models are thus a class of minimal models. With respect to the canonical ordering, *stratified programs admit an unique perfect Herbrand model.* Eventually, this model, no matter which stratification is considered, is *equivalent to 'M_P'*. Therefore, all stratifications for a program are semantically equivalent.

What above holds not only for Herbrand pre-interpretations but also for any other CET's model. Thus a stratified program has a unique perfect model based on a CET's model J and this is equivalent to M^J_P (to be obtained both by iteration and intersection). $PERF(P)$, the collection of perfect models of P+CET, plays then for stratified programs the same role that $MIN(P)$ plays for positive programs. In the next chapter we present an interpreter complete with respect to it and in TURI [T] we show how s-interpretations can be extended to stratified programs so that the resulting perfect supra-model mirror the least s-model of positive programs.

The import of perfect models goes far beyond stratification and LP. Last years have seen an increasing interest from AI researchers in logic as foundation of commonsense reasoning mechanization. Efforts have concentrated on representing knowledge merely by logical axioms, independently from specific applications and relying only on some forms of logical deduction as inference mechanism, so to allow a declarative reading of knowledge. This in contrast with AI common practice of writing down procedures (or algorithms) to solve specific problems and inferring from their behaviour – that is, procedurally – the knowledge they implicitely represent. Roughly speaking, under a procedural approach AI is a craft, while a declarative approach would transform it into science.

Put in these terms, logic is just a mean to achieve (the goal of) a declarative theory of commonsense reasoning. Originally, such research programme was based on logic alone, but many difficulties have arisen which indicate that declarative (i.e. mathematical and independent from the implementation level) notions other than logical are required. Of these problems an example is the isolation of an intended subclass of models from an inconsistent collection of minimal models for a knowledge base. (this is the case of any circumscriptive theory, eventually applied to LP.) Then, there is no 'purely' logical criterion to perform such selection. But perfect models, with their declarative notion of preference, offer a neat solution to the problem, as well as a methodological directive for a more general approach to the matter.

5.3. Prioritized Circumscription ([L2], [P2])

From what above it should be clear *why* and *how* we may modify circumscription to rend it more adequate to model non-monotonic reasoning. Already in section 3.3 we mentioned the problem which minimizing each predicate in a program independently from the others gives. It is more convenient to circumscribe just a subset, say $\{q_1,\ldots,q_h\}$, of the whole set of predicates at the time, leaving another subset, say $\{r_1,\ldots,r_k\}$, free to vary and ignoring the rest. This process amounts to the so-called *pointwise circumscription* of P:

$$circ(P;\{\boldsymbol{q}\};\{\boldsymbol{r}\}) \equiv P(\boldsymbol{q},\boldsymbol{r}) \wedge \forall \boldsymbol{q}',\boldsymbol{r}' \, [P(\boldsymbol{q}',\boldsymbol{r}') \rightarrow \boldsymbol{q} \leqslant \boldsymbol{q}']$$

(where bold letters are the usual abbreviation of tuples).

In general, pointwise circumscription does not solve the problem because we lack of a principle to make proper partitions. But for stratified programs we have a valid and effective criterion at the hand. If we put $\boldsymbol{p}_i$ for the predicates in a stratum P_i, we may define *prioritized circumscription* of a program P the following formula:

$$circ(P;\{\boldsymbol{p}_1\}>\ldots>\{\boldsymbol{p}_n\}) \equiv circ(P;\{\boldsymbol{p}_1\};\{\boldsymbol{p}_2,\ldots,\boldsymbol{p}_n\}) \wedge$$

$$\wedge\ circ(P;\{p_2\};\{p_3,\ldots,p_n\})\ \wedge$$

$$\ldots$$

$$\wedge\ circ(P;\{p_n\};\varnothing).$$

It should not surprise that *the only Herbrand model of* $circ(P;\{p_1\}>\ldots>\{p_n\})$ *turns out to be* M_P *—the unique perfect Herbrand model of* P— and that thus any stratification can be indifferently used to circumscribe.

As for circumscription also for the closed world assumption there is a natural way to express it for stratified programs. It suffices to 'close' each stratum P_i of the program making use of the *supported* Herbrand model for the strata P_i, that is, the model which we characterized both as $M(P_i)$ and as M_i (see section 5.1). The resulting closure is called *iterated closed world assumption* (*icwa*) and it is obviously equivalent to prioritized circumscription plus CET+DCA (comp. Theorem 3.10). Clearly, *icwa* is the closure in between *cwa* and *gcwa* announced in section 3.2.

5.4. Well-Founded Models ([VGRS])

It is very tempting to close here this chapter, when stratification and perfect models provide general programs with most of the completeness enjoyed by positive programs. However, our semantical investigation has not yet come to an end.

Perfect models are based on 2-valued logic. In order to provide with at least a partial model semantics those programs otherwise rejected as intractable, it is worth to extend perfect models taking 3-valued logic into account. Of course, to be significant, the resulting semantics should be more powerful than Φ_P's least fixpoint's. That is, when there is an intended model which is total, this should be acknowledged. Example 4.10 shows that this is not the case with Φ_P's least fixpoint semantics. There, the program was

$$p \leftarrow q, \neg r\ ,$$

$$p \leftarrow r, \neg s\ ,$$

$$r \leftarrow q\ ,$$

$$q \leftarrow p\ ,$$

with intended —as well as total— model the one whose atoms are all false. Instead, iterating Φ_P bottom-up we do not come further than the partial model in which only s is false and the rest is unknown. Here, Φ_P's symmetric handling of positive and negative atoms is felt has a problem. No choice between declaring p, q and r (together) either true or false is taken.

Modifying Φ_P so to perform the opportune choice in cases like the one above, we shall obtain a fixpoint semantics *a*) extending perfect models', *b*) able to give a partial model when a total does not exist, and *c*) not depending on priorities —at least not explicitely.

Notice that, while we set about intending to extend perfect models, we are now going to extend the 3-valued operator Φ_P, expecting the former to follow from the latter.

Let us return to our example: why is it more natural to put p, q and r false rather than true? Well, simply because of our negation as failure approach, which gives to *false* priority over *true*. It might be objected that negation as failure deals with single atoms, while here we have to infer an entire set at once: since p, q and r are defined by *mutual recursion*, there is no way to separate them. (Notice that only the following fragment is relevant here:

$$p \leftarrow r\ ,$$

$$r \leftarrow q\ ,$$

$$q \leftarrow p\ ;$$

ie negative literals have no influence on this problem.) The point is that nothing prevents us from generalizing negation as failure to sets of atoms. To this purpose we introduce the notion of *unfounded set* (VAN GELDER, ROSS AND SCHLIPF [VGRS]), based on the considerations above. A set of ground atoms A is *unfounded* wrt a partial interpretation I iff

$$\forall\, p \in A \;\Rightarrow\; \forall\, p \leftarrow F \text{ in ground(P)}$$

$$\text{either } I \models \neg F \tag{1}$$

$$\text{or } \exists\; q \text{ positively occurring in } F \text{ s.t. } q \in A \tag{2}$$

Clearly, (1) amounts to classical negation as failure, while (2) deals with mutual recursion. $U_P(I)$, the union of P's unfounded sets wrt I, is the transformation which allowes us to modify $\Phi_P(I)$. The result is the so-called *well-founded* operator W_P ([VGRS]), defined as follows (recall the notation introduced in section 3.4 for partial interpretations):

$$\begin{aligned} W_P(I) = W_P(I_T,\, I_F) &\equiv (T_P(I),\, U_P(I)) \\ &= (I_T',\, U_P(I)) \\ &= (I_T',\, I_F' \cup L_P(I)). \end{aligned}$$

With $L_P(I)$ we intend to characterize the set of atoms not entailed by I (thus 'loose' from I) —what in $W_P(I)$ truly differs from $\Phi_P(I)$. To this purpose we need a 3-steps transformation.

First of all, we put

$$P' \equiv D_1(ground(P),\, I),$$

with D_1 being the transformation which *deletes* all the rules with body false (in I). Notice that if an atom p belongs to I_F' then no clause with head P is in P'. Therefore, passing from P to P' we rule out part (1) of the unfounded set definition.

Next, we can put

$$P'' \equiv D_2(P'),$$

where D_2 *deletes* all negative literals (they do not play any role in part (2) of the unfounded set definition).

Finally, $L_P(I)$ is the set of atoms **not** entailed by P'' using I. Since P'' is positive we can consider I_T alone and thus put

$$L_P(I) \equiv B_{P''} - T_{P''}\Uparrow\omega(I_T).$$

Cumulative rather than simple powers are necessary, as the following program shows: For P''

$$p \leftarrow q, r\ ,$$

$$r \leftarrow\ ,$$

we have

$$T_{P''}\Uparrow\omega(\{q\}) = \{p, q, r\},$$

$$T_{P''}\uparrow\omega(\{q\}) = \{r\}.$$

(*Remark* The operator L_P does not appear in [VGRS]: it is our own reformulation of their operator U_P.)

The least fixpoint of W_P (which by monotony is to be obtained iterating bottom-up, but —by analogy with Φ_P— is not recursively enumerable, ie more than ω steps are in general necessary) *is a fixpoint for* Φ_P as well, thus a (partial) model for the program and its completion. When such model is total we call it well-founded and we say that the program has a *well-founded semantics.*

Stratified programs (as well as locally stratified) have a well-founded semantics. In fact, the well-founded model is then equivalent to the perfect (Herbrand) one. (For the proof −by induction− see [VGRS].) It is in this sense that we regard the well-founded semantics as extending perfect models'.

6. Interpreters II

6.1. SLS-Resolution ([P])

Let us first summarize the results obtained as far in the declarative modelling of SLDNF-resolution. To this purpose is convenient to classify all the possible SLDNF-computations in the following way:

i) terminating with success;
ii) terminating with failure;
iii) floundering;
iv) infinite.

Of these classes: the first two have a natural declarative modelling; the third requires either a restriction on programs and queries (allowedness) or an extension of the interpreter (constructive negation); the forth calls for either a restriction on the programs (acyclic programs) or 3-valued modelling.

Up to chapter 4 we have been guided through our semantical investigation by the procedural meaning which SLDNF-resolution gives to programs. But in chapter 5 we have taken a different attitude: our declarative modelling of stratified programs has been based upon an *intended* meaning, whose source, rather than SLDNF-resolution, is the reduction of stratified programs to a hierarchy of positive programs. Little wonder then that the resulting semantics is not complete wrt SLDNF-resolution.

Example 6.1.

In the perfect model for the program

$$p \leftarrow p$$

p is false, while SLDNF-resolution fails to give an answer to the query $\leftarrow p$. □

Clearly, any interpreter which has to match the perfect model semantics has to differ from SLDNF-resolution in the infinite computations handling. That is, of the four classes of SLDNF-computations above the last one (infinite) should be reduced to the second one (termination with failure). But this is the interpreter we have announced in section 3.2 −based on negation as failure, rather than *finite* failure− and we have already shown that it cannot be effective. However, we have also advocated a theoretical significance for it and here we add another argumentation to support this position. Let us quote Przymusinski [P]:

> We can consider SLDNF-resolution as a special effective variant of SLS-resolution, providing one of many possible effective approximation to SLS-resolution. In this role, SLS-resolution can be used as a performance yardstick to evaluate to what extent any actual implementation measures up to this 'ideal' and when and how it differs from it. The fact that SLS-resolution is not, in general, effective should not diminish its theoretical value, as the fact that Riemann integral $\int e^{-x^2} dx$ does not have a finite representation −and therefore can only be approximated− does not diminish its theoretical importance.

SLS-Resolution is Przymusinski's interpreter for stratified programs (the name stems from SLD-resolution and 'S' stands for 'Stratification'). We do not give here its formal definition, but the intuition is as follows: Each literal in a query for a stratified program provides an entry to a single stratum of the program and every sub-query from that literal will refer either to the same stratum (through a positive literal) or to lower ones. As consequence, infinite loops concern exclusively positive literals

and only from a single stratum. This allowes to infer that when for a positive literal, say α, the interpreter both fails to terminate with success and does not flounder there is no rule in the program to deduce α bottom-up either $-$ie $M_P \models \neg\alpha$. Therefore, α can be declared false and used recursively in higher strata (what would not happen with SLDNF-resolution). Here is a simple example:

EXAMPLE 6.2.
For P:

$$p \leftarrow \neg q \ ,$$

$$q \leftarrow q \ ,$$

SLS-resolution succeeds on the query

$$\leftarrow p \ ,$$

since $q \leftarrow q$ $-$the only definition of q in P$-$ is an infinite loop. □

For positive programs and positive queries SLS-resolution is equivalent to SLD-resolution. The same universal query problem (comp. section 3.1) has thus to be overcome. For this reason we express the completeness result for SLS-resolution in terms of $PERF(P)$ $-$the collection of perfect models for P+CET$-$ rather than perfect Herbrand model only.

THEOREM 6.3. *For P stratified program and ϕ non-floundered query:*

$PERF(P) \models \forall\phi\theta$ *iff* SLS-resolution succeeds on ϕ with answer more general than θ

$PERF(P) \models \neg\exists\phi$ *iff* SLS-resolution fails on ϕ. □

(In contrast with *success*, where a single SLS-derivation suffices, we intend SLS-resolution to *fail* on a query only if all possible derivations do not succeed nor flounder.)

7. CONCLUSION

Reducing this survey to its essence we may describe it as follows.

Positive LP is the combination of Horn clause logic with SLD-resolution. From the former it inherits a declarative semantics, from the latter a procedural one. The equivalence between these two semantics (*completeness*) is probably the most significative property of (Positive) LP.

A full exploitation of positive programs suggests inferring negative information from them. There are two ways to do this. One amounts to a declarative reading of the queries, the other to a procedural one. Roughly speaking, answering a query declaratively amounts to checking whether the atoms in the query are members of the least Herbrand model of the program ($\leftarrow q$ is true for P if q belongs to M_P). For non-ground atoms instantiation is required, but the principle is the same. But then, as membership is used to determine truth, *non*-membership can be used to determine falsity. In short, *negation as failure.* The alternative is *negation as finite failure,* the form of negation arising from a procedural reading of the query: finite (ie observable) failure of SLD-resolution to prove a query true determines its falsity.

The problem is that the two approaches above are not equivalent. This means that, in order to re-establish the original (and precious) completeness of Positive LP, it is necessary $-$either to design an interpreter after the declarative requirements $-$or to find a declarative characterization of SLDNF-resolution (ie SLD-resolution extended by the negation as finite failure rule). Two opposite paths, indeed.

A second theme, strictly depending on the previous, and of even greater relevance in this survey, is the use in LP of programs with negated atoms appearing in the body of the clauses (*general* programs). Indeed, once negation can appear in the queries for positive programs it is natural to allow it to appear also in the subqueries, that is, in the body of program clauses. Here the problem is that no semantics for positive programs is trivially applicable to general ones, since recursion may bring about writing ill-founded programs, for which no reference to positive programs helps. Again, two solutions have been pursued. One is to impose a structure on the programs so that the semantics for positive programs can be applied. The other is to apply one of the semantics for positive programs to as much of the general program as possible, leaving the rest simply undetermined (*partial models semantics*).

The combination of all these ingredients results in two lines of research, marked by the 'declarative/procedural' contraposition. The first runs as follows:

i) positive programs: least Herbrand model, cwa, circumscription;
ii) stratified programs: perfect Herbrand model, iterated cwa, prioritized circumscription;
iii) SLS-resolution.

This is clearly declaratively-driven: first models for positive programs, then structuring general programs to preserve such models, and finally the interpreter. Designing the interpreter after purely declarative requirements leads to loss of effectiveness, so that the interpreter has only a theoretical significance. On the other hand, the declarative modelling is free and can result in a theory of wider application than merely LP. Concepts like the closed world assumption and circumscription are borrowed from AI and perfect models are even an original contribution to the logical foundations of AI. The idealization required for stating completeness is not experienced as a problem, there.

The second line of research runs as follows:

1) SLDNF-resolution;
2) positive programs: completion;
3) general programs:
   3a) 3-valued completion: allowedness;
   3b) 2-valued completion: call-consistence + strictness + allowedness.

Here we have a procedurally-driven approach. The interpreter −SLDNF-resolution− is at the top of the 'chain' influencing all the declarative modelling. The result is a closure (*completion*) which is an adaptation of circumscription to LP and, consequently, a notion of no relevance outside LP. However, effectiveness is achieved −even in the declarative semantics (with $\Phi^{*}_{P}\uparrow\omega$)− and this is of greater importance for LP, when we regard it as a programming language, rather than as a formalism for knowledge representation.

To complete the picture we still have to mention a few other points. The well-founded semantics (section 5.4), for instance, which is a promising development of the 'chain' i-ii-iii, combining both elements of what we described as 'second theme': structuring of the programs and partial models semantics. It also provides a very interesting generalization of negation as failure.

Another point is the s-semantics (section 3.1): −how can it be extended from positive to general programs modelling? In TURI [T] we give an answer to this question, basing our work on Chan's constructive negation ([Ch], [P3]), which at the moment is one of the most promising development areas for LP. To this we address the reader eager to contribute to the development of the field.

ACKNOWLEDGEMENTS

Together with TURI [T] this survey is meant to form my Master thesis. It has been written at the *Centre for Mathematics and Computer Science* (*CWI*) of Amsterdam, where, very generously, I have been supplied with more than the necessary support for what has become an unforgettable experience to me. For this I wish to thank Professor Krzysztof R. Apt, who was so kind to provide me with his valuable guidance during my stay at *CWI*. I would also like to thank Professor Roberto Barbuti (University of Pisa) for inviting me at the *Advanced School on Foundations of Logic Programming* (Alghero, Italy, Sept 88), my first contact with this research world.

Several friends at *CWI* helped and encouraged me, but I owe Jan, Dr Jan Rutten, an amazingly unconditional support when I was – a stranger.

To Hilde, I dedicate this work.

REFERENCES

[A] K.R. APT, *Introduction to Logic Programming,* Technical Report No. CS-R8826, Centre for Mathematics and Computer Science, Amsterdam, 1988, to appear as a chapter in Handbook of Theoretical Computer Science, J. van Leeuwen managing ed., North-Holland.

[ABW] K.R. APT, H.A. BLAIR and A. WALKER, *Towards a Theory of Declarative Knowledge*, in: Foundations of Deductive Databases and Logic Programming (Minker, J., ed.), Morgan Kaufmann, Los Altos, 1988, 89-148.

[B] M. BEZEM, *Characterizing Termination of Logic Programs with Level Mapping,* in: Proc. North American Conference on Logic Programming (Lusk, E.L. and R.A. Overbeek, eds.), The MIT Press, Cambridge, Mass., 1989, 69-80.

[BA] M. BEZEM and K.R. APT, *Acyclic Programs,* Draft paper.

[Ca] L. CAVEDON, *On the Completeness of SLDNF-Resolution,* Technical Report No. 88/17, Department of Computer Science, University of Melbourne, 1988.

[Ch] D. CHAN, *Constructive Negation Based on the Completed Database,* in: Proc. 5th International Conference and Symposium on Logic Programming, (Kowalski, R.A. and K. Bowen, eds), The MIT Press, Cambridge, Mass., 1988, 111-125.

[C] K.L. CLARK, *Negation as Failure*, in: Logic and Data Bases, (Gallaire, H. and J. Minker, eds), Plenum Press, New York, 1978, 293-322.

[FLMP] M. FALASCHI, G. LEVI, M. MARTELLI and C. PALAMIDESSI, *Declarative Modeling of the Operational Behaviour of Logic Languages,* Theoretical Computer Science, vol. 70, 1989.

[GMN] H. GALLAIRE, J. MINKER and J.M. NICOLAS, *Logic and Databases: a Deductive Approach*, ACM Computing Surveys, vol. 16, No. 2, 1984, 153-186.

[K] K. KUNEN, *Negation in Logic Programming,* Journal of Logic Programming, vol. 4, No. 4, 1987, 289-308.

[K1] K. KUNEN, *Signed Data Dependencies in Logic Programs,* Technical Report No. 719, Department of Computer Science University of Wisconsin, 1987, to appear in Journal of LP.

[L] V. LIFSCHITZ, *Computing Circumscription,* in: Proc. IJCAI-85, Los Angeles 1985, 121-127.

[L1] V. LIFSCHITZ, *Closed World Data Bases and Circumscription,* Artificial Intelligence, vol. 27, 1985, 229-235.

[L2] V. LIFSCHITZ, *On the Declarative Semantics of Logic Programs with Negation,* in: Foundations of Deductive Databases and Logic Programming (Minker, J. ed.), Morgan Kaufmann, Los Altos, 1988, 177-192.

[Ld] J.W. LLOYD, *Foundations of Logic Programming*, Springer Verlag, 1984.

[Ld1] J.W. LLOYD, *Foundations of Logic Programming*, Second Edition, Springer Verlag, 1987.

[P] T. PRZYMUSINSKI, *On the Declarative and Procedural Semantics of Logic Programs,* Journal of Automated Reasoning, vol. 5, No. 2, 1989.

[P1] T. PRZYMUSINSKI, *On the Declarative Semantics of Stratified Deductive Databases and Logic*

Programs, in: Foundation of Deductive Databases and Logic Programming, (Minker, J., ed.), Morgan Kaufmann, Los Altos, 1988, 193-216.

[P2] T. PRZYMUSINSKI, *Non-Monotonic Reasoning vs Logic Programming: A New Perspective,* to appear in Handbook on the Formal Foundations of AI, Y. Wilks and D. Partridge, eds.

[P3] T. PRZYMUSINSKI, *On Constructive Negation in Logic Programming,* Draft paper.

[R] R. REITER, *Circumscription implies Predicate Completion (sometimes),* in: Proc. AAAI-82, 1982, 418-420.

[S] J.C. SHEPHERDSON, *Negation in Logic Programming,* in: Foundations of Deductive Databases and Logic Programming, (Minker, J. ed.), Morgan Kaufmann, Los Altos, 1988, 19-88.

[T] D. TURI, *Extending S-Interpretations to Logic Programs with Negation,* Technical Report, Centre for Mathematics and Computer Science, Amsterdam, 1989.

[VGRS] A. VAN GELDER, K.A. ROSS and J.S. SCHLIPF, *The Well-Founded Semantics for General Logic Programs,* to appear in Journal of ACM.

Table of Contents