Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

K.R. Apt

Introduction to logic programming

Computer Science/Department of Software Technology   Report CS-R8741   September

69F41, 69F32

Introduction to Logic Programming

Krzysztof R. Apt
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands
and
Department of Computer Sciences, University of Texas at Austin, Austin, Texas 78712-1188, USA.

We provide a systematic and self-contained introduction to the theory of logic programming.

Note: This work was performed in part at the Laboratoire d'Informatique, Ecole Normale Supérieure, 45 rue d'Ulm, 75250 Paris, France and at the Centre for Mathematics and Computer Science, Amsterdam, The Netherlands. This paper will appear in : *Handbook of Theoretical Computer Science,* (J. van Leeuwen, Managing Editor), North Holland.

1986 CR Categories: F.4.1, F.3.2.

1. Introduction

1.1. Background
Some formalisms gain a sudden success and it is not always immediately clear why. Consider the case of logic programming. It was introduced in an article of Kowalski [K] in 1974 and for a long time - in the case of computer science - not much happened. But now, 13 years later already the Journal of Logic Programming and Annual Conferences on the subject exist and a few hundred of articles on it have been published.

Its success can be attributed to at least two circumstances. First of all, logic programming is closely related to PROLOG. In fact, logic programming constitutes its theorical framework. And PROLOG gained in turn a success mainly due to its inclusion in the Japanese Fifth Generation Project.
Secondly, in the early eighties a flurry of research on alternative programming styles started and suddenly it turned out that some candidates already existed and even for a considerable time. This led to a renewed interest in logic programming and its extensions.

The power of logic programming stems from two reasons. First, it is an extremely simple formalism. So simple, that some, when confronted with it for the first time, say "Is that all?". Next, it relies on mathematical logic which developed its own methods and techniques and which provides a rigorous mathematical framework. (It should be stated however, that the main basis of logic programming is automatic theorem proving which was developed mainly by computer scientists.)

The aim of this article is to provide a self-contained introduction to the theory of logic programming. In the presentation we try to shed light on the causal dependence between various concepts and notions. Throughout the paper we attempt to adhere to the notation of Lloyd [L], the book which obviously influenced our presentation. This will hopefully further contribute to the standardization of the notation and terminology in the domain.

1.2. Plan of this paper
We now provide a short description of the content the paper. It is hoped that this will facilitate its reading and will allow a better understanding of the structure of its subject.

The aim of Chapter 2 is to introduce in the fastest possible way the notion of *SLD-resolution* central to the subject of logic programming.

In Chapter 3 a semantics is introduced with the purpose of establishing soundness of SLD-

Report CS-R8741
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

resolution and several forms of its completeness. Most of these results are collected in the Success Theorem 3.26.

In Chapter 4 the computability by means of logic programs is investigated. It is among others shown that all recursive functions are computed by logic programs.

SLD-resolution allows us to derive only positive statements. Chapter 5 deals with the other side of the coin - the derivability of the negative statements. After rejecting the *Closed World Assumption* rule as ineffective, the full effort is directed at an analysis of a weaker but effective rule - the *Negation as Failure* rule and its relation to the construction called *completion of a program.* The final outcome is the Finite Failure Theorem 5.30 dual to the Success Theorem.

After this extensive analysis of how to deal with positive and with negative statements, the mixed statements (so called *general goals*) are investigated in Chapter 6. While the resulting form of resolution (called here *SLDNF$^-$-resolution*) is sound, the completeness can be obtained only after imposing a number of restrictions, both on the logic programs and the general goals.

The paper concludes by a short discussion of related topics which are divided into six sections: general programs, alternative approaches, deductive databases, PROLOG, integration of logic and functional programming, and applications in artificial intelligence.

Finally, in the appendix a short history of the subject is traced.

2. Syntax And Proof Theory

2.1. *First order languages*

Logic programs are simply sets of certain formulas of a first order language. So to define them we recall first what a first order language is, a notion essentially due to G. Frege. By necessity our treatment is reduced to a list of definitions. A reader wishing a more motivated introduction should consult one or more standard books on the subject. Personally, we recommend Manin [M] and Shoenfield [S].

A *first order language* consists of an alphabet and all formulas defined over it.

An *alphabet* consists of the following classes of symbols:

a) *variables* denoted by $x,y,z,v,u,...,$
b) *constants* denoted by $a,b,c,d,...,$
c) *function symbols* denoted by $f,g,.,..,$
d) *relation symbols* denoted by $p,q,r,...,$
e) *propositional constants*, which are: **true** and **false**,
f) *connectives*, which are: $\neg$ (negation), $\vee$ (disjunction), $\wedge$ (conjunction), $\rightarrow$ (implication) and $\leftrightarrow$ (equivalence),
g) *quantifiers*, which are: $\exists$ (there exists) and $\forall$ (for all),
h) *parentheses*, which are: (and) and the *comma*, that is: ,.

Thus the sets of connectives, quantifiers and parentheses are fixed. We assume also that the set of variables is infinite and fixed. Those classes of symbols are called *logical symbols*. The other classes of symbols, that is constants, relation symbols (or just *relations*) and function symbols (or just *functions*) may vary and in particular may be empty. They are called *nonlogical symbols*.

Each function and relation symbol has a fixed *arity*, that is the number of arguments. We assume that functions have a positive arity - the rôle of 0-ary functions is played by the constants. In contrast, 0-ary relations are admitted. They are called *propositional symbols*, or simply *propositions*. Note that each alphabet is uniquely determined by its constants, functions and relations.

We now define by induction two classes of strings of symbols over a given alphabet. First we define the class of *terms* as follows:

a) a variable is a term,
b) a constant is a term,
c) if f is an n-ary function and $t_1,...,t_n$ are terms then $f(t_1,...,t_n)$ is a term.

Terms are denoted by s,t,u. Finally, we define the class of *formulas* as follows:

a) if p is an n-ary relation and $t_1,...,t_n$ are terms then $p(t_1,...,t_n)$ is a formula (called an *atomic formula*, or just an *atom*),
b) **true** and **false** are formulas,
c) if F and G are formulas then so are $(\neg F)$, $(F \vee G)$, $(F \wedge G)$, $(F \rightarrow G)$ and $(F \leftrightarrow G)$,
d) if F is a formula and x is a variable then $(\exists x F)$ and $(\forall x F)$ are formulas.

Sometimes we shall write $(G \leftarrow F)$ instead of $(F \rightarrow G)$. Some well known binary functions (like $+$) or relations (like $=$) are usually written in an *infix notation.* Atomic formulas are denoted by A,B and formulas in general by F,G.

Given two strings of symbols e_1 and e_2 from the alphabet we write $e_1 \equiv e_2$ when e_1 and e_2 are identical. Usually these strings will be terms or formulas.

The definition of formulas is rigorous at the expense of excessive use of parentheses. One way to eliminate most of them is by introducing a *binding order* among the connectives and quantifiers. We thus assume that $\neg,\exists$ and $\forall$ bind stronger than $\vee$ which in turn binds stronger than $\wedge$ which binds stronger than $\rightarrow$ and $\leftrightarrow$. Also, we assume that $\vee,\wedge,\rightarrow$ and $\leftrightarrow$ *associate to the right* and omit the outer parentheses. Thus thanks to the binding order we can rewrite the formula

$$(\forall y (\forall x ((p(x) \wedge (\neg r(y))) \rightarrow ((\neg q(x)) \vee (A \vee B)))))$$

as

$$(\forall y (\forall x (p(x) \wedge \neg r(y) \rightarrow \neg q(x) \vee (A \vee B))))$$

which thanks to the convention of the association to the right further simplifies to

$$\forall y \forall x (p(x) \wedge \neg r(y) \rightarrow \ q(x) \vee A \vee B).$$

This completes the definition of a first order language.

2.2. *Logic programs*

To bar an easy access to newcomers every scientific domain has introduced its own terminology and notation. Logic programming is no exception in this matter. Thus an atom or its negation is called a *literal.* A *positive literal* is just an atom while a *negative literal* is the negation of an atom. Note that **true** and **false** are not atoms.

In turn, a formula of the form

$$\forall x_1 \ldots \forall x_s (L_1 \vee \cdots \vee L_m)$$

where $x_1,...,x_s$ are all the variables occurring in the literals $L_1,...,L_m$ is called a *clause.* From now on clauses will be always written in a special form called - yes, you guessed it - a *clausal form.* The above formula in a clausal form is written as

$$A_1,...,A_k \leftarrow B_1,...,B_n$$

where $A_1,...,A_k$ is the list of all positive literals among $L_1,...,L_m$, called *conclusions* and $B_1,...,B_n$ is the list of remaining literals stripped of the negation symbol, called *premises.* Informally, it is to be understood as: (A_1 or ... or A_k) if (B_1 and ... and B_n). Thus for example the formula

$$\forall x \forall y (p(x) \vee \neg A \vee \neg q(y) \vee B)$$

looks in clausal form as

$$p(x), B \leftarrow A, q(y).$$

If a clause has only one conclusion ($k=1$), then it is called a *program clause* or a *definite clause.* Its conclusion is then usually called a *head* and the list of its premises a *body.* When the set of premises of a program clause is empty ($n=0$), then we talk of a *unit clause.* They have the form $A \leftarrow$. When the set of conclusions is empty ($k=0$), then we talk of a *goal* or a *negative clause.* They have the form $\leftarrow B_1,...,B_n$. Finally, when both the set of premises and conclusions is empty then we talk of the *empty clause* and denote it by $\square$. It is interpreted as a contradiction.

To understand this interpretation we are in fact brought to the question of meaning of a formula $L_1 \vee ... \vee L_m$ when $m = 0$, i.e. of the empty disjunction. Now, the empty disjunction is considered as always false because it asks for an existence of a true disjunct when none of them exists. In contrast, the empty conjunction is considered as always true because it asks for truth of all conjuncts, which holds when none of them exists.

Now, we can define a *logic program* (or just a *program*) - it is a finite non-empty set of program clauses.

Logic programs form a subclass of general logic programs. To define the general programs we first introduce the concept of a *general clause*. It is a construct of the form

$$A_1,...,A_k \leftarrow L_1,...,L_n$$

where $A_1,...,A_k$ are positive literals and $L_1,...,L_n$ are (not necessarily positive) literals. When there is only one conclusion ($k=1$), we talk of a *general program clause*, and when the set of conclusions is empty ($k=0$) we talk of a *general goal.*

A general clause $A_1,...,A_k \leftarrow L_1,...,L_n$ represents the formula

$$\forall x_1 ... \forall x_s (A_1 \vee ... \vee A_k \vee \neg L_1 \vee ... \vee \neg L_n).$$

where $x_1,...,x_s$ are all the variables appearing in $A_1,...,A_k, L_1,...,L_n$.

Now, a *general logic program* (or just a *general program*) is a finite non-empty set of general program clauses.

Due to the lack of space we do not discuss in this paper the general programs. However, in Chapter 6 we study general goals. This provides a first step towards an analysis of general programs. Note that **true** and **false** are not used to define (general) programs. These formulas will be however needed later, in Section 5.5. Formulas of the form $\forall x_1 ... \forall x_s F$ ($s \geqslant 0$) where F is quantifier-free are usually called *universal formulas*. Thus each clause is a universal formula.

With each program P we can uniquely associate a first order language L_P whose constants, functions and relations are those occurring in P. All considerations concerning a program P refer to the language L_P. In particular, in statements like "Let P be a program and N a goal" N is always assumed to be a goal from L_P.

There are two ways of interpreting a clause $A \leftarrow B_1,...,B_n$. One is: to solve A solve B_i for $i = 1,...,n$. The other is: A is true if $B_1,...,B_n$ are true. The first interpretation is usually called procedural interpretation whereas the second is called declarative interpretation. It is this first interpretation which distinguishes logic programming from first order logic. We shall discuss this double interpretation in more detail at the end of Chapter 3.

2.3. Substitutions

Consider now a fixed first order language. In logic programming variables are assigned values by means of a special type of substitutions, called "most general unifiers". Formally, a *substitution* is a finite mapping from variables to terms, and is written as

$$\theta = \{x_1/t_1,...,x_n/t_n\}.$$

Informally, it is to be read: the variables $x_1,...,x_n$ become (or are *bound to*) $t_1,...,t_n$, respectively.

The notation implies that the variables $x_1,...,x_n$ are different. We also assume that for $i=1,...,n$ $x_i \neq t_i$. If $t_1,...,t_n$ are different variables then θ is called a *renaming*. A pair x_i/t_i is called a *binding*. If all $t_1,...,t_n$ are variable-free then θ is called *ground.*

Substitutions operate on expressions. By an *expression* we mean a term, a sequence of literals or a clause and denote it by E. For an expression E and a substitution θ, $E\theta$ stands for the result of applying θ to E which is obtained by *simultaneously* replacing each occurrence in E of the variable from the domain of θ by the corresponding term. The resulting expression $E\theta$ is called an *instance* of E. An instance is *variable-free* or *ground* if it contains no variables. If θ is a renaming defined on all variables of an expression E then $E\theta$ is called a *variant* of E.

Given a program P we denote by ground (P) the set of all ground instances of clauses in P. Note that this set can be infinite.

Substitutions can be composed. Given substitutions $\theta = \{x_1/t_1,...,x_n/t_n\}$ and $\eta = \{y_1/s_1,...,y_m/s_m\}$ their *composition* $\theta\eta$ is defined by removing from the set

$$\{x_1/t_1\eta,...,x_n/t_n\eta, y_1/s_1,...,y_m/s_m\}$$

those pairs $x_i/t_i\eta$ for which $x_i \equiv t_i\eta$, as well as those pairs y_i/s_i for which $y_i \in \{x_1,...,x_n\}$.

Thus for example when $\theta = \{x/3, y/x+1\}$ and $\eta = \{x/4\}$ then $\theta\eta = \{x/3, y/4+1\}$. This definition implies the following simple result.

LEMMA 2.1. *For all substitutions θ, η and γ and an expression E*
i) $(E\theta)\eta \equiv E(\theta\eta)$
ii) $(\theta\eta)\gamma = \theta(\eta\gamma)$. □

This lemma shows that when writing a sequence of substitutions, also in the context of an expression, the parentheses can be omitted. By convention substitution binds stronger than any connective or quantifier.

We say that a substitution θ is *more general* than a substitution η if for some substitution γ we have $\eta = \theta\gamma$.

2.4. Unifiers

Finally, we introduce the notion of unification. Consider two atoms A and B. If for a substitution θ we have $A\theta \equiv B\theta$, then θ is called a *unifier* of A and B and we then say that A and B are *unifiable.* A unifier θ of A and B is called a *most general unifier* (or *mgu* in short) if it is more general than any other unifier of A and B. It is an important fact that if two atoms are unifiable then they have a most general unifier. In fact, we have the following theorem due to ROBINSON [Ro].

THEOREM 2.2. (Unification Theorem) *There exists an algorithm (called a* unification algorithm) *which for any two atoms produces their most general unifier if they are unifiable and otherwise reports non-existence of a unifier.*

PROOF. We follow here the presentation of LASSEZ, MAHER and MARRIOTT [LMM]. We present an algorithm based upon Herbrand's original algorithm (HERBRAND [He] p. 148) which deals with solutions of finite sets of term equations. This algorithm is also presented in MARTELLI and MONTANARI [MM].

Two atoms can unify only if they have the same relation symbol. With two atoms $p(s_1,...,s_n)$ and $p(t_1,...,t_n)$ to be unified we associate a set of equations

$$\{s_1 = t_1,...,s_n = t_n\}.$$

A substitution θ such that $s_1\theta \equiv t_1\theta,...,s_n\theta \equiv t_n\theta$ is called a *unifier* of the set of equations $\{s_1 = t_1,...,s_n = t_n\}$. Thus the set of equations $\{s_1 = t_1,...,s_n = t_n\}$ has the same unifiers as the atoms $p(s_1,...,s_n)$ and $p(t_1,...,t_n)$. Two sets of equations are called *equivalent* if they have the same unifiers.

A (possibly empty) set of equations is called *solved* if it is of the form $\{x_1 = u_1,...,x_n = u_n\}$ where x_i's are distinct variables and none of them occurs in a term u_j.

A solved set of equations $\{x_1 = u_1,...,x_n = u_n\}$ determines the substitution $\{x_1/u_1,...,x_n/u_n\}$. This substitution is a unifier of this set of equations and clearly it is its *mgu*, that is it is more general than any other unifier of this set of equations.

Thus to find an *mgu* of two atoms it suffices to transform the associated set of equations into an equivalent one which is solved.

The following algorithm does it if this is possible and otherwise halts with failure.

Unification algorithm

Non-deterministically choose from the set of equations an equation of a form below and perform the associated action.

```
(1) $f(s_1,...,s_n) = f(t_1,...,t_n)$            replace by the equations $s_1 = t_1,...,s_n = t_n$
(2) $f(s_1,...,s_n) = g(t_1,...,t_m)$ where $f \not\equiv g$
                                                 halt with failure
(3) $x = x$                                      delete the equation
(4) $t = x$ where $t$ is not a variable
                                                 replace by the equation $x = t$
(5) $x = t$ where $x \not\equiv t$ and $x$ has another occurrence in the set of equations
                                                 if $x$ appears in $t$ then halt with failure
                                                 otherwise perform the substitution $\{x/t\}$
                                                 in every other equation
```

The algorithm terminates when no step can be performed or when failure arises. To keep the formulation of the algorithm concise we identified here constants with 0-ary functions. Thus step (1) includes the case $c = c$ for every constant c which leads to deletion of such an equation. Also step (2) includes the case of two constants.

First, observe that for each variable x step (5) can be performed at most once, so this step can be performed only a finite number of times. Subsequent applications (if any) of steps (1) and (4) strictly diminish the total number of occurrences of function symbols on the left hand side of the equations. This number is not affected by the application of step (3). Moreover, in the absence of step (1), step (3) can be performed only finitely many times. This implies termination.

Next, observe that applications of steps (1), (3) and (4) replace a set of equations by an equivalent one. The same holds in the case of a successful application of step (5) because for any substitution θ, $x\theta \equiv t\theta$ implies that the substitutions θ and $\{x/t\}\theta$ are identical.

Next, observe that if the algorithm successfully terminates, then by virtue of steps (1), (2) and (4) the left hand sides of the final equations are variables. Moreover, by virtue of step (5) these variables are distinct and none of them occurs on the right hand side of an equation. So if the algorithm successfully terminates it produces a solved set of equations equivalent with the original one.

Finally, observe that if the algorithm halts with failure then the set of equations at the failure step does not have a unifier.

This establishes correctness of the algorithm and concludes the proof of the theorem. □

To illustrate the operation of the above unification algorithm consider the following example.

EXAMPLE 2.3 (LASSEZ, MAHER, MARRIOTT [LMM]). Consider the following set of equations

$$\{f(x) = f(f(z)),\ g(a,y) = g(a,x)\}.$$

Choosing the first equation step (1) applies and produces the new equation set

$$\{x = f(z),\ g(a,y) = g(a,x)\}.$$

Choosing the second equation step (1) applies and yields

$$\{x = f(z),\ a = a,\ y = x\}.$$

Now by applying step (1) again we get

$$\{x = f(z), y = x\}.$$

The only step which can be now applied is step (5). We get

$$\{x = f(z), y = f(z)\}.$$

Now no step can be applied and the algorithm successfully terminates. □

Call a substitution θ *idempotent* if $\theta\theta = \theta$. At the end of Chapter 5 we shall need the following observation.

COROLLARY 2.4. *If two atoms are unifiable then they have an mgu which is idempotent.*

PROOF. The unifier produced by the procedure used in the proof of the above theorem is of the form $\{x_1/u_1,...,x_n/u_n\}$ where none of the variables x_i occurs in a term u_j, so it is idempotent. □

2.5. Computation process - the SLD-resolution

Logic programs compute through a combination of two mechanisms - replacement and unification. This form of computing boils down to a specific form of theorem proving, called a *resolution.* To better understand this computation process let us concentrate first on the issue of a replacement in the absence of variables.

Consider for a moment a logic program P in which all clauses are ground. Let $N = \leftarrow A_1,...,A_n$ $(n \geqslant 1)$ be a ground negative clause and suppose that for some $i, 1 \leqslant i \leqslant n$, $C = A_i \leftarrow B_1,...,B_k$ $(k \geqslant 0)$ is a clause from P. Then

$$N' = \leftarrow A_1,...,A_{i-1},B_1,...,B_k,A_{i+1},...,A_n,$$

is the result of replacing A_i in N by $B_1,...,B_k$ and is called a *resolvent* of N and C. A_i is called the *selected atom* of N.

Iterating this replacement process we obtain a sequence of resolvents which is called a *derivation.* A derivation can be finite or infinite. If its last clause is empty then we speak of a *refutation* of the original negative clause N. We can then say that from the assumption that in presence of the program P the clause $N = \leftarrow A_1,...,A_k$ holds we derived the contradiction, namely the empty clause. This can be viewed as a proof of the negation of N from P.

Assuming for a moment from the reader knowledge of semantics for the first order logic (which is explained in Section 3.1) we note that N stands for $\neg A_1 \vee ... \vee \neg A_k$, so its negation stands for $\neg(\neg A_1 \vee ... \vee \neg A_k)$ which is semantically equivalent to $A_1 \wedge ... \wedge A_k$. Thus a refutation of N can be viewed as a proof of $A_1 \wedge ... \wedge A_k$.

If we reverse the arrows in clauses we can view a program with all clauses ground as a context-free grammar with erasing rules (i.e. rules producing the empty string) and with no start or terminal symbols. Then a refutation of a goal can be viewed as a derivation of the empty string from the word represented by the goal.

An important aspect of logic programs is that they can be used not only to *refute* but also to *compute* - through a repeated use of unification which produces assignments of values to variables. We now explain this process by extending the previous situation to the case of logic programs and negative clauses which can contain variables.

Let P be a logic program and $N = \leftarrow A_1,...,A_n$ be a negative clause. We first redefine the concept of a *resolvent.* Suppose that $C = A \leftarrow B_1,...,B_k$ is a clause from P. If for some i, $1 \leqslant i \leqslant n$, A_i and A unify with an *mgu* θ, then we call

$$N' = \leftarrow(A_1,...,A_{i-1},B_1,...,B_k,A_{i+1},...,A_n)\theta$$

a *resolvent of N and C*. Thus a resolvent is obtained by performing the following four steps:
a) select an atom A_i,
b) unify (if possible) A and A_i,
c) if b) succeeds then perform the replacement of A_i by $B_1,...,B_k$ in N,
d) apply to the resulting clause the *mgu* θ obtained in b).

As before, iterating this process of computing a resolvent we obtain a sequence of resolvents called a derivation. But now because of the presence of variables we have to be careful.

By an *SLD - derivation* (we explain the abbreviation *SLD* in a moment) of $P \cup \{N\}$ we mean a maximal sequence $N = N_0, N_1, ...$ of negative clauses together with a sequence $C_0, C_1, ...$ of variants of clauses from P and a sequence $\theta_0, \theta_1, ...$ of substitutions such that for all $i = 0,1,...$
a) N_{i+1} is a resolvent of N_i and C_i,
b) θ_i is the *mgu* mentioned in step d) above,
c) C_i does not have a variable in common with N_i.

The clauses $C_0, C_1, ...$ are called the *input clauses* of the derivation. When one of the resolvents N_i is empty then it is the last negative clause of the derivation. Such a derivation is then called an *SLD - refutation*. A finite *SLD*-derivation is called *failed* if it is not an refutation.

A new element in this definition is the use of variants that satisfy c) instead of the original clauses. The idea is that we do not wish to make the result of the derivation dependent on the choice of variable names. Note for example that $p(x)$ and $p(f(y))$ unify by means of the *mgu* binding x to $f(y)$. Thus the goal $\leftarrow p(x)$ can be refuted from the program $p(f(x)) \leftarrow$.

The existence of an *SLD* - refutation of $P \cup \{N\}$ for $N = \leftarrow A_1, ..., A_k$ can be viewed as a contradiction. We can then conclude that we proved the negation of N. But N stands for $\forall x_1 ... \forall x_s (\neg A_1 \vee ... \vee \neg A_k)$, so its negation stands for $\neg \forall x_1 ... \forall x_s (\neg A_1 \vee ... \vee \neg A_k)$ which is semantically equivalent (see Section 3.1) to $\exists x_1 ... \exists x_s (A_1 \wedge ... \wedge A_k)$. Now, an important point is that the sequence of substitutions $\theta_0, \theta_1, \ldots, \theta_m$ performed during the process of the refutation actually provides the bindings for the variables $x_1, ..., x_s$. Thus the existence of an *SLD* - refutation for $P \cup \{N\}$ can be viewed as a proof of the formula $(A_1 \wedge ... \wedge A_k)\theta_0 ... \theta_m$. We justify this statement in Section 3.2.

The restriction of $\theta_0 ... \theta_m$ to the variables of N is called a *computed answer substitution* for $P \cup \{N\}$. The *success set* of a program P is the set of ground atoms A in the language of P for which $P \cup \{\leftarrow A\}$ has an *SLD* - refutation.

According to the definition of *SLD* - derivation the following two choices are made in each step of constructing a new resolvent:
a) choice of the selected atom,
b) choice of the input clause whose conclusion unifies with the selected atom.

By a *selection rule R* we now mean a function which, given $k \geqslant 0$, selects from a sequence $A_1, ..., A_n$ an atom A_j, $1 \leqslant j \leqslant n$. The parameter k is to be interpreted as the depth of the *SLD*-derivation at which the selection takes place. The use of the parameter k allows us to select different atoms in a resolvent that occurs more than once in the derivation.

Given a selection rule R we say that an *SLD* - derivation of $P \cup \{N\}$ is *via R* if all choices of the selected atoms in the derivation are performed according to R. That is, if $\leftarrow A_1, ..., A_n$ is the k-th resolvent in an *SLD*-derivation of $P \cup \{N\}$ then $R(k, A_1, ..., A_n)$ is its selected atom.

Now, *SLD* stands for Linear resolution with Selection rule for Definite clauses.

2.6. An example

To the reader overwhelmed with such a long sequence of definitions we offer an example which hopefully clarifies the introduced concepts. We analyze in it the consequences of the choices in a) and b).

Consider a simplified version of the 8 - puzzle. Assume a 3×3 grid filled with eight moveable tiles. Our goal is to rearrange the tiles so that the blank one is the middle.

We number the fields consecutively as follows:

1	2	3
4	5	6
7	8	9

and represent each legal move as a movement of the "blank" to an adjacent square.

First, we define the relation *adjacent* by providing an exhaustive listing of adjacent squares in ascending order:

adjacent(1,2)←, adjacent(2,3)←,..., adjacent(8,9)←, (*horizontal adjacency*)

adjacent(1,4)←, adjacent(4,7)←,..., adjacent(6,9)← (*vertical adjacency*)

and using a rule

adjacent(x,y)←adjacent(y,x) (*symmetry*) (a)

In total, 24 pairs are adjacent. (A more succinct representation would be possible if addition and subtraction functions were available.)

Then we define an initial configuration by assuming that the blank is initially, say, on square 1. Thus we have

configuration(1,*nil*)←,

where the second argument - here nil - denotes the sequence of squares visited.

Finally, we define a legal move by the rule

configuration(x,$y.\ell$)←adjacent(x,y), configuration(y,ℓ) (b)

where $y.\ell$ is a list with head y and tail ℓ written in the usual infix notation.

As a goal we choose the negative clause

←configuration(5,ℓ)

stating that no sequence of visited squares leads to a situation where square 5 is blank.

The following represents an *SLD* - refutation of the goal of length 7.

← **configuration (5,ℓ)**	(b) $\{\ell/\ell_1\}$, $\{x/5, \ell/y.\ell_1\}$
← **adjacent (5,y)**, configuration (y,ℓ_1)	(a) $\{y/y_1\}$, $\{x/5, y_1/y\}$
← **adjacent (y,5)**, configuration (y,ℓ_1)	adjacent(4,5) ←, $\{y/4\}$
← **configuration (4,ℓ_1)**	(b) $\{y/y_1\}\{\ell/\ell_2\}$, $\{x/4, \ell_1/y_1.\ell_2\}$
← **adjacent (4,y_1)**, configuration (y_1,ℓ_2)	(a) $\{y/y_2\}$, $\{x/4, y_2/y_1\}$
← **adjacent (y_1,4)**, configuration (y_1,ℓ_2)	adjacent (1,4) ←, $\{y_1/1\}$
← **configuration(1,ℓ_2)**	configuration(1,nil) ←, $\{\ell_2/nil\}$

□

Selected atoms are put in bold. We thus always select the leftmost atom. On the right the input clauses and the *mgu*'s are given. Note that at various places variants of the clauses (a) and (b) are used. The sequence of *mgu*'s performed binds the variable ℓ to 4.1.nil through the consecutive substitutions $\{\ell/y.\ell_1\}, \{y/4\}, \{\ell_1/y_1.\ell_2\}, \{y_1/1\}, \{\ell_2/nil\}$.

This provides the sequence of squares leading to the final configuration. Thus the refutation of the initial goal is constructive in the sense that it provides the value of ℓ for which the formula
$\leftarrow$ configuration $(5,\ell)$ does not hold.

Another choice of input clauses can lead to an infinite *SLD* - derivation. For example here is a derivation in which we repeatedly use rule (a):

$\leftarrow$ **configuration (5,ℓ)** (b) $\{\ell/\ell_1\}$, $\{x/5, \ell/y.fsl_1\}$

$\leftarrow$ **adjacent (5,y)**, configuration (y,ℓ_1) (a) $\{y/y_1\}$, $\{x/5, y_1/y\}$

$\leftarrow$ **adjacent (y,5)**, configuration (y,ℓ_1) (a) $\{y/y_1\}$, $\{x/y, y_1/5\}$

$\leftarrow$ **adjacent (5,y)**, configuration (y,ℓ_1)

...

Also, another choice of a selection rule can lead to an infinite *SLD* - derivation. For example, a repeated choice of the rightmost atom and rule (b) leads to an infinite derivation with the goals continuously increasing its length by 1.

2.7. Refutation procedures - SLD-trees

When searching for a refutation of a goal *SLD*-derivations are constructed with the aim of generating the empty clause. The totality of these derivations form a *search space.* One way of organizing this search space is by dividing *SLD*-derivations into categories according to the selection rule used. This brings us to the concept of an *SLD*-tree.

To this purpose we first explain how from sequences (here *SLD*-derivations) a tree can be constructed. Consider a set of possibly infinite sequences *W* such that no element of *W* is an extension of another element of *W*. With such a *W* we can uniquely associate a tree whose nodes are the elements of these sequences, whose branches are all the sequences in *W* and in which different nodes have different prefixes. We call such a tree a *prefix tree constructed from W.*

Let *P* be a program, *N* a goal and *R* a selection rule. Then the *SLD-tree* for $P \cup \{N\}$ via *R* is the prefix tree constructed from all *SLD*-derivations of $P \cup \{N\}$ via *R*. Thus the root node in an *SLD*-tree for $P \cup \{N\}$ is *N* and every node in this tree is a goal whose descendants are all its resolvents with (the variants of) the clauses of *P*, where the selected atom is chosen according to *R*. We call an *SLD*-tree *successful* if it contains the empty clause.

The *SLD*-tree for $P \cup \{N\}$ via a selection rule *R* groups all *SLD*-derivations of $P \cup \{N\}$ via *R*. The *SLD*-trees for $P \cup \{N\}$ can differ in size and form.

EXAMPLE 2.5 (APT and VAN EMDEN [AVE]). Let *P* be the following program:

1. path(x,z)$\leftarrow$ arc(x,y),path(y,z),
2. path$(x,x)\leftarrow$,
3. arc$(b,c)\leftarrow$.

A possible interpretation of *P* is as follows: arc(x,y) holds if there is an arc from x to y and path(x,y) holds if there is a path from x to y.
Figures 1 and 2 show two *SLD*-trees for $P \cup \{\leftarrow$path(x,c)$\}$. The selected atoms are put in bold, used clauses and performed substitutions are indicated. Whenever no confusion could arise we used the original clauses as input clauses.

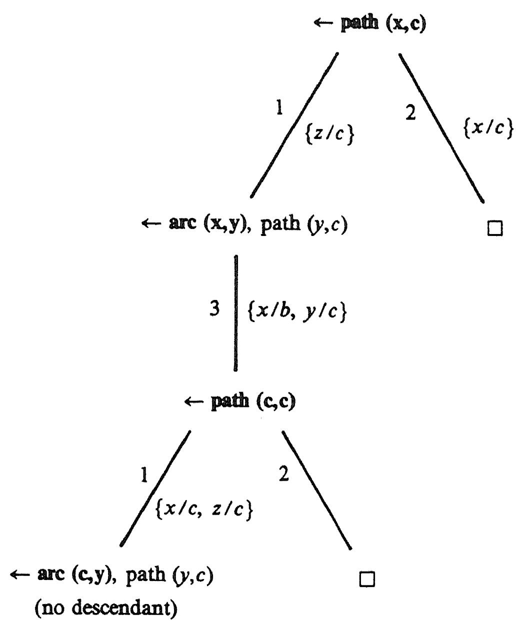

FIGURE 1

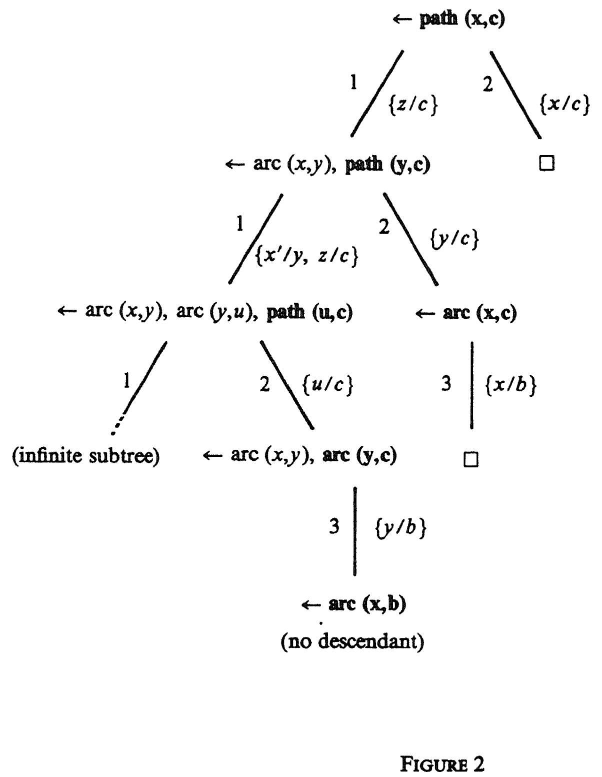

FIGURE 2

Note that the first tree is finite while the second one is infinite. Both trees contain the empty clause.

2.8. Bibliographic remarks

Efficient unification algorithms were proposed by PATERSON and WEGMAN [PW] and MARTELLI and MONTANARI [MM]. See also the survey on unification by SIEKMANN [Si].

SLD-resolution is a special case of *SL*-resolution of KOWALSKI and KUEHNER [KK] and was proposed as a basis for programming in KOWALSKI [K]. The name was first used in APT and VAN EMDEN [AVE] where also the notions of a success set and *SLD*-trees were formally introduced. *SLD*-trees were informally used in CLARK [C] where they were called evaluation trees.

Selection rule was originally required to be function defined on sequences of atoms. Our relaxation follows the suggestion of SHEPHERDSON [She] (see p. 62).

3. SEMANTICS

3.1. Semantics for first order logic

To understand the *meaning* of a logic program, or a first order formula in general, we now provide the definition of semantics due to A. Tarski. Again, our treatment is very brief. More extensive discussion of this fundamental issue can be found e.g. in MANIN [M] or SHOENFIELD [S].

We begin by defining an interpretation. An *interpretation* I for a first order language L consists of:

a) a non-empty set D, called the *domain* or the *universe* of I,

b) an assignment for each constant c in L of an element c_I of the domain,

c) an assignment for each n-ary function f in L of a mapping f_I from D^n to D,

d) an assignment for each n-ary relation r in L of an n-ary predicate r_I on D, i.e. a subset of D^n.

Our aim is now to define when a formula of L is true in an interpretation for L. To this purpose we first relate terms to elements of the domain. We do this by making use of the notion of a state (or a *variable assignment*). A *state* (*over I*) is simply a function assigning to each variable an element of the domain of I.

Given now a state σ we extend its domain to all terms, that is we assign to a term t an element $\sigma(t_I)$ from the domain I proceeding by induction as follows:

a) for a variable v we define $\sigma(v_I)$ as $\sigma(v)$, the result of applying the state σ to v,

b) for a constant c we define $\sigma(c_I)$ as c_I (thus $\sigma(c_I)$ does not depend on σ),

c) if $f(t_1,...,t_n)$ is a term then we define $\sigma(f(t_1,...,t_n)_I)$ as $f_I(\sigma(t_{1I}),...,\sigma(t_{nI}))$, the result of applying the mapping f_I to the sequence of values associated with the terms $t_1,...,t_n$.

Observe that for a ground term $t, \sigma(t_I)$ does not depend on σ.

We can now define a semantics of a formula. Given a formula F we define inductively its truth in a state σ for I, written as $I \models_\sigma F$, as follows:

a) if $p(t_1,...,t_n)$ is an atomic formula then

$$I \models_\sigma p(t_1,...,t_n) \text{ iff } (\sigma(t_{1I}),...,\sigma(t_{nI})) \in p_I,$$

that is, if the sequence of values associated with terms $t_1,...,t_n$ belongs to the predicate p_I,

b) $I \models_\sigma$ **true**, not $I \models_\sigma$ **false**,

c) if F and G are formulas then

$$I \models_\sigma \neg F \text{ iff not } I \models_\sigma F,$$

$$I \models_\sigma F \vee G \text{ iff } I \models_\sigma F \text{ or } I \models_\sigma G,$$

$$I \models_\sigma \forall x F \text{ iff } I \models_{\sigma[x/d]} F \text{ for all } d \in D.$$

Here $\sigma[x/d]$ for a state σ, an element d of the domain of I and a variable x stands for the state which differs from σ only on the variable x to which it assigns the element d.

This allows us already to define truth of clauses. The truth of other formulas is defined by expressing the remaining connectives and the quantifier $\exists$ in terms of $\neg, \vee$ and $\forall$:

$F \wedge G$ as $\neg(\neg F \vee \neg G)$,

$F \rightarrow G$ as $\neg F \vee G$,

$F \leftrightarrow G$ as $(F \rightarrow G) \wedge (G \rightarrow F)$, *(and then using the above two definitions)*

$\exists x F$ as $\neg \forall x \neg F$.

Finally, we say that the formula F *is true in the interpretation* I, and write $I \models F$, when for all states σ, $I \models_\sigma F$. Note that $\square$ as the empty disjunction is false in every interpretation I. Let now S be a set of formulas. We say that an interpretation I is a *model for* S if every formula from S is true in I. When S has a model, we say that it is *satisfiable* or *consistent*. Otherwise, we say that it is *unsatisfiable* or *inconsistent*. When every interpretation is a model for S, we say that S is *valid*. Given another set of formulas S' we say that S *semantically implies* S' or S' *is a semantic consequence of* S, if every model of S is also a model of S'. We write then $S \models S'$. S and S' are *semantically equivalent* if both $S \models S'$ and $S' \models S$ hold.

Several simple facts about semantic equivalence and logical consequence can be proved and will be used in the sequel. Already in Section 2.5 we used the fact that the following formulas are valid:

$$\neg \forall x_1 ... \forall x_s F \leftrightarrow \exists x_1 ... \exists x_s \neg F,$$

$$\neg(A_1 \vee ... \vee A_n) \leftrightarrow \neg A_1 \wedge ... \wedge \neg A_n,$$

$$\neg\neg F \leftrightarrow F.$$

3.2. Soundness of the SLD-resolution

Given a goal $N = \leftarrow A_1,...,A_k$ denote by $N^\sim$ the formula $A_1 \wedge ... \wedge A_k$. Then $\square^\sim$ is the empty conjunction so it is valid. The following lemma is immediate.

LEMMA 3.1. *If N_0 is a resolvent of N and a clause C with an mgu θ then*

$$C \models N_0^\sim \rightarrow N^\sim \theta. \ \square$$

As a consequence we obtain the following theorem due to CLARK [C1] justifying the statement made in Section 2.5.

THEOREM 3.2. (Soundness of SLD - resolution). *Let P be a program and $N = \leftarrow A_1,...,A_k$ a goal. Suppose that there exists an SLD - refutation of $P \cup \{N\}$ with the sequence of substitutions $\theta_0,...,\theta_n$. Then $(A_1 \wedge ... \wedge A_k)\theta_0 ... \theta_n$ is a semantic consequence of P.*

PROOF. Let $N = N_0,...,N_{n+1} = \square$ be the *SLD*-refutation in question and let $C_0,...,C_n$ be its input clauses. Applying Lemma 3.1. $n+1$ times we get

$$P \models \square^\sim \rightarrow (A_1 \wedge \cdots \wedge A_k)\theta_0 ... \theta_n$$

which implies the claim. $\square$

COROLLARY 3.3. *If there exists an SLD - refutation of $P \cup \{N\}$ then $P \cup \{N\}$ is inconsistent.* $\square$

EXAMPLE 3.4. Reconsider now the program P studied in the example in Section 2.6 with the goal $\leftarrow$ configuration $(5,\ell)$. Since we exhibited there an *SLD* - refutation of $P \cup \{\leftarrow$ configuration $(5,\ell)\}$, we conclude by the above corollary that $P \cup \{\leftarrow$ configuration $(5,\ell)\}$ is inconsistent, that is $P \models \exists \ell$ configuration $(5,\ell)$. More specifically, by the Soundness Theorem we have

$P \models$ configuration $(5,\ell)\theta_0 \ldots \theta_7$ where $\theta_0, \ldots, \theta_7$ is the sequence of performed substitutions. As we saw before this sequence binds ℓ to 4.1.nil, so we have $P \models$ configuration (5, 4.1.nil). □

A natural question arises whether a converse of the above Corollary or of the Soundness Theorem can be proved, that is whether certain form of *completeness* of *SLD* - resolution can be shown. To handle this question we introduce a special class of models of logic programs, called Herbrand models.

3.3. Herbrand models

Let L be a first order language whose set of constants is not empty. By the *Herbrand universe* U_L for L we mean the set of all ground terms of L. By the *Herbrand base* B_L for L we mean the set of all ground atoms of L. If L is the first order language associated with a program P (that is L is L_P) then we denote U_L and B_L by U_P and B_P, respectively. Now, by a *Herbrand interpretation* for L we mean an interpretation for L such that

a) its domain is the Herbrand universe U_L,
b) each constant in L is assigned to itself,
c) if f is an n-ary function in L then it is assigned to the mapping from $(U_L)^n$ to U_L defined by assigning the ground term $f(t_1,...,t_n)$ to the sequence $t_1,...,t_n$ of ground terms,
d) if r is an n-ary relation in L then it is assigned to a set of ground atoms of L whose relation symbol is r.

Thus each Herbrand interpretation for L is uniquely determined by a subset I of the Herbrand base B_L which fixes the assignment of predicates to relation symbols of L by assigning the set $\{(t_1,...,t_n):r(t_1,...,t_n)\in I\}$ to the n-ary relation symbol r. In other words, we can identify Herbrand interpretations for L with (possibly empty) subsets of the Herbrand base B_L. This is what we shall do in the sequel.

To avoid some uninteresting complications we assume from now on that whenever a program P has variables then it also has some constants. This guarantees that its Herbrand base and the set ground (P) are not empty. The case of programs containing variables but no constants is hardly of interest and can be handled by considering its derived version obtained by substituting all variables by some fixed constant.

With this restriction another uninteresting complication arises when a program uses only propositional symbols. Then its Herbrand universe is empty. To handle this case one can consider a derived program which additionally contains a ground atom $r(a)$ for a new relation symbol r or simply drop the condition that a domain of an interpretation is non-empty when L is constant-free and function-free.

By a *Herbrand model* for a set S of sentences we mean a Herbrand interpretation which is a model for S. The following simple lemma shows that when studying logic programs it suffices to consider their Herbrand models.

Lemma 3.5. *Let S be a set of universal formulas. If S has a model then it has a Herbrand model.*

Proof. For a model I let $I_H = \{A : A$ is a ground atom and $I \models A\}$ denote the corresponding Herbrand interpretation. A simple induction on the length of the formulas shows that I and I_H satisfy the same quantifier-free variable-free formulas. From this the lemma follows. □

Corollary 3.6. *Let P be a program and N a negative clause. If $P \cup \{N\}$ is consistent then it has a Herbrand model.* □

We conclude this section by introducing two often reoccurring qualifications. A Herbrand model of a set of formulas S is the *least* model of S if it is included in every other Herbrand model of S and it is *minimal* if no proper subset of it is a Herbrand model of S. The least model is minimal but the converse is not always true.

3.4. *The immediate consequence operator*

To study Herbrand models of programs, following VAN EMDEN and KOWALSKI [VEK], we introduce the *immediate consequence operator* T_P mapping Herbrand interpretations to Herbrand interpretations. We put for a program P

$$\begin{aligned} A \in T_P(I) \text{ iff for some atoms } & B_1,\ldots,B_n \\ & A \leftarrow B_1,\ldots,B_n \text{ is in ground } (P) \\ & \text{and } I \models B_1 \wedge \ldots \wedge B_n. \end{aligned}$$

Alternatively, for a ground atom A

$$\begin{aligned} A \in T_P(I) \text{ iff for some substitution } & \theta \\ & \text{and a clause } B \leftarrow B_1,\ldots,B_n \text{ of } P \\ & \text{we have } A \equiv B\theta \text{ and } I \models (B_1 \wedge \ldots \wedge B_n)\theta. \end{aligned}$$

In particular, if $A \leftarrow$ is in P, then every ground instance $A\theta$ of A is in $T_P(I)$ for every I. The following simple observation from [VEK] relates Herbrands models of P with the operator T_P.

PROPOSITION 3.7. *For a program P and a Herbrand interpretation I, I is a model of P iff $T_P(I) \subseteq I$.*

PROOF. First note that I is a model of P iff it is a model of ground (P). Now the latter is true iff for every clause $A \leftarrow B_1,\ldots,B_n$ in ground (P) $I \models B_1 \wedge \ldots \wedge B_n$ implies $I \models A$, i.e. $A \in I$. But this is true iff $T_P(I) \subseteq I$. □

When $T(I) \subseteq I$ holds, I is called a *pre-fixpoint* of T. Thus to study Herbrand models of a program P it suffices to study the pre-fixpoints of its immediate consequence operator T_P. This brings us to a study of operators and their pre-fixpoints in a general setting.

3.5. *Operators and their fixpoints*

Consider now an arbitrary, but fixed, complete lattice (for the definition see e.g. BIRKHOFF [Bi]) with the order relation $\subseteq$, the least upper bound operator $\cup$ and the greatest lower bound operator $\cap$. To keep in mind the subsequent applications to logic programs and their interpretations we denote the least element by $\emptyset$, the largest element by B, and the elements of the lattice by I,J,M. Given a set $A = \{I_n : n = 0,1,\ldots\}$ of elements, we denote $\cup A$ and $\cap A$ by $\bigcup_{n=0}^{\infty} I_n$ and $\bigcap_{n=0}^{\infty} I_n$, respectively. Sometimes we rather write $\bigcup_{n<\omega} I_n$ and $\bigcap_{n<\omega} I_n$.

Consider an operator T on the lattice. T is called *monotonic* if for all I,J $I \subseteq J$ implies $T(I) \subseteq T(J)$. T is called *finitary* if for every infinite sequence

$$I_0 \subseteq I_1 \subseteq \ldots,$$

$$T(\bigcup_{n=0}^{\infty} I_n) \subseteq \bigcup_{n=0}^{\infty} T(I_n)$$

holds. If T is both monotonic and finitary then it is called *continuous*. A more often used, equivalent definition of continuity is: T is continuous iff for every infinite sequence

$$I_0 \subseteq I_1 \subseteq \ldots,$$

$$T(\bigcup_{n=0}^{\infty} I_n) = \bigcup_{n=0}^{\infty} T(I_n)$$

holds.

As already mentioned in the previous section any I such that $T(I) \subseteq I$ is called a *pre-fixpoint of T*. If $T(I) = I$ then I is called a *fixpoint of T* and if $T(I) \supseteq I$ then I is called a *post-fixpoint* of T.

We have the following classical theorem.

THEOREM 3.8. (Fixpoint Theorem) (KNASTER and TARSKI [Ta]). *A monotonic operator T has a least fixpoint lfp(T) which is also its least pre-fixpoint.* □

We now define *powers* of a monotonic operator T. We put

$$T\uparrow 0(I) = I,$$

$$T\uparrow(n+1)(I) = T(T\uparrow n(I)),$$

$$T\uparrow\omega(I) = \bigcup_{n<\omega} T\uparrow n(I)$$

and abbreviate $T\uparrow\alpha(\varnothing)$ to $T\uparrow\alpha$.

Powers of a monotonic operator generalize in a straightforward way to *transfinite powers* $T\uparrow\alpha(I)$ where α is an arbitrary ordinal. We shall not need them in the sequel.

The following well known fact holds.

LEMMA 3.9. *If T is continuous, then $T\uparrow\omega$ is its least pre-fixpoint and its least fixpoint.* □

In the next section we apply these observations to the study of Herbrand models.

In Chapters 4 and 5 we shall also use largest fixpoints and *downward powers* of monotonic operators. We put for a monotonic operator T

$$T\downarrow 0(I) = I,$$

$$T\downarrow(n+1)(I) = T(T\downarrow n(I)),$$

$$T\downarrow\omega(I) = \bigcap_{n<\omega} T\downarrow n(I).$$

Downward powers generalize in a straightforward way to *transfinite downward powers* $T\downarrow\alpha(I)$ where α is an arbitrary ordinal. We abbreviate $T\downarrow\alpha(B)$ to $T\downarrow\alpha$.

Note that

$$T\uparrow n(I) \subseteq T\uparrow(n+1)(I)$$

does not necessarily hold but by monotonicity for all $n \geqslant 0$

$$T\uparrow n \subseteq T\uparrow(n+1)$$

does hold. Analogous statement holds for the downward powers.

The dual theorem to the Fixpoint Theorem 3.8 is

THEOREM 3.10. *A monotonic operator T has a greatest fixpoint gfp(T) which is also its greatest post-fixpoint.* □

A monotonic operator T is called *downward continuous* if for every infinite sequence

$$I_0 \supseteq I_1 \supseteq \ldots,$$

$$T(\bigcap_{n=0}^{\infty} I_n) = \bigcap_{n=0}^{\infty} T(I_n)$$

holds.

We have the following well known lemma.

LEMMA 3.11. *Let T be a monotonic operator. Then for every α we have $T{\downarrow}\alpha \supseteq gfp(T)$. Moreover, for some α, $T{\downarrow}\alpha = gfp(T)$. If T is downward continuous then this ordinal is $\leqslant \omega$.* □

We denote the smallest ordinal α for which $T{\downarrow}\alpha = gfp(T)$ by $\|T{\downarrow}\|$ and call it the *downward closure ordinal* of T or the *closure ordinal* of $T{\downarrow}$.

3.6. Least Herbrand models

Let us first investigate the properties of the immediate consequence operator.

LEMMA 3.12. *Let P be a program. Then*

i) T_P is finitary.

ii) T_P is monotonic.

PROOF.

i) Consider an infinite sequence
$I_0 \subseteq I_1 \subseteq ...$
of Herbrand interpretations and suppose that
$A \in T_P(\bigcup_{n=0}^{\infty} I_n)$.
Then for some atoms $B_1,...,B_k$
$A \leftarrow B_1,...,B_k$ is in ground (P), and moreover $\bigcup_{n=0}^{\infty} I_n \models B_1 \wedge \cdots \wedge B_k$. But the latter implies that for some I_n, namely the one containing all $B_1,...,B_k$,
$I_n \models B_1 \wedge ... \wedge B_k$. So $A \in T_P(I_n)$.

ii) Immediate by definition. □

As an immediate consequence of the above lemma we have:

THEOREM 3.13. (Characterization Theorem) (VAN EMDEN and KOWALSKI [VEK]) *Let P be a program. Then P has a least Herbrand model M_P which satisfies the following properties:*

i) M_P equals the intersection of all Herbrand models of P.

ii) M_P is the least pre-fixpoint of T_P.

iii) M_P is the least fixpoint of T_P.

iv) $M_P = T_P{\uparrow}\omega$.

PROOF. It suffices to apply Theorem 3.8 and Lemma 3.9. □

COROLLARY 3.14. *The success set of a program P is contained in its least Herbrand model.*

PROOF. By Corollary 3.3 and the above theorem. □

3.7. Completeness of the SLD - resolution

We can now return to the problem of completeness.
We first prove the converse of Corollary 3.3 that is the following result due to HILL [H]. The proof is due to APT and VAN EMDEN [AVE].

THEOREM 3.15 (Completeness of *SLD* - resolution) *Let P be a program and N a goal. Suppose $P \cup \{N\}$ is inconsistent. Then there exists an SLD refutation of $P \cup \{N\}$.*

First we need the following lemma.

LEMMA 3.16 (Substitution lemma) *Let P be a program, N a goal and θ a substitution. Suppose that there exists an SLD - refutation of $P \cup \{N\theta\}$. Then there exists an SLD - refutation of $P \cup \{N\}$.*

PROOF. We proceed by induction on the length n of the refutation of $P \cup \{N\theta\}$. We can assume that θ does not act on any of the variables of P. Let $N = \leftarrow A_1,...,A_k$.

If $n=1$ then $k=1$ and $A_1\theta$ unifies with a head of a unit clause of P. So A_1 unifies with the head of the same clause. This settles the claim.

If $n>1$ then consider the first input clause $B_0 \leftarrow B_1,...,B_m$ of the refutation. For an *mgu* η we have $A_i\theta\eta \equiv B_0\eta$ where $A_i\theta$ is the selected atom of $N\theta$. Thus by the assumption on θ $A_i\theta\eta \equiv B_0\theta\eta$, so A_i and B_0 unify. For some *mgu* ξ and a substitution σ we have $\theta\eta = \xi\sigma$.

By the assumption on $P \cup \{N\theta\}$ and θ there exists an *SLD* - refutation of

$P \cup \{\leftarrow (A_1\theta,...,A_{i-1}\theta,B_1\theta,...,B_m\theta,A_{i+1}\theta,...,A_k\theta)\eta\}$

of length $n-1$. By the induction hypothesis there exists an *SLD* - refutation of $P \cup \{\leftarrow (A_1,...,A_{i-1},B_1,...,B_m,A_{i+1},...,A_k)\xi$.

Consider now an *SLD* - derivation of $P \cup \{N\}$ in which the first selected atom is A_i and the first input clause is $B_0 \leftarrow B_1,...,B_m$ with the *mgu* ξ. Its first resolvent is $\leftarrow (A_1,...,A_{i-1},B_1,...,B_m,A_{i+1},...,A_k)\xi$ which by the above settles the claim. □

We now establish the converse of Corollary 3.14.

LEMMA 3.17. *The least Herbrand model of a program P is contained in the success set of P.*

PROOF. We make use of the continuity of the immediate consequence operator T which provides an internal structure to M_P.

Suppose $A \in M_P$. By the Characterization Theorem 3.13 iv) for some $k > 0$, $A \in T_P\uparrow k$. We now prove by induction on k that there exists an *SLD* - refutation of $P \cup \{\leftarrow A\}$. For $k = 1$ the claim is obvious.

If $k > 1$, then for some ground atoms $B_1,...,B_n$ the clause $A \leftarrow B_1,...,B_n$ is in ground (P) and $\{B_1,...,B_n\} \subseteq T_P\uparrow(k-1)$. By the induction hypothesis, for $i = 1,...,n$ there exists an *SLD* - refutation of $P \cup \{\leftarrow B_i\}$. But all B_i are ground so there exists an *SLD* - refutation of $P \cup \{\leftarrow B_1,...,B_n\}$.

Consider now an *SLD* - derivation of $P \cup \{\leftarrow A\}$ with the first input clause being the one of which $A \leftarrow B_1,...,B_n$ is a ground instance. Its first resolvent is a negative clause of which $\leftarrow B_1,...,B_n$ is a ground instance. The claim now follows by Lemma 3.16. □

We are now in position to prove the Completeness Theorem.

PROOF OF THEOREM 3.15. Suppose that $N = \leftarrow A_1,...,A_n$. M_P is not a model of $P \cup \{N\}$ so N is not true in M_P. Thus for some substitution θ $\{A_1\theta,...,A_n\theta\} \subseteq M_P$. By Lemma 3.17, for $i = 1,...,n$ there exists an *SLD* - refutation of $P \cup \{\leftarrow A_i\theta\}$. But all $A_i\theta$ are ground so there exists an *SLD* - refutation of $P \cup \{N\theta\}$ and the claim now follows by Lemma 3.16. □

3.8. Correct answer substitutions

The completeness theorem can be generalized in various ways. We provide here two such generalizations.

First we introduce the following notions. Let P be a program and $N = \leftarrow A_1,...,A_n$ a goal. We say that θ is a *correct answer substitution* for $P \cup \{N\}$ if θ acts only on variables appearing in N and $P \models (A_1 \wedge ... \wedge A_n)\theta$ holds.

Note that if θ is a correct answer substitution for $P \cup \{N\}$ then for all γ, $P \cup \{N\theta\gamma\}$ is inconsistent. Consequently, $P \cup \{N\}$ is inconsistent as it is equivalent to a weaker statement that for some γ $P \cup \{N\gamma\}$ is inconsistent.

The following theorem is a kind of converse of the Soundness Theorem 3.2.

THEOREM 3.18 (CLARK [C1]). *Consider a program P and a goal N. For every correct answer substitution θ for $P \cup \{N\}$ there exists a computed answer substitution for $P \cup \{N\}$ which is more general than θ.*

We present here the proof due to LLOYD [L]. First we need the following strengthening of the Substitution lemma.

LEMMA 3.19 (Lifting lemma). *Let P be a program, N a goal and θ a substitution. Suppose that there exists an SLD - refutation of $P \cup \{N\theta\}$ with the sequence of mgu's $\theta_0, \dots, \theta_n$. Then there exists an SLD - refutation of $P \cup \{N\}$ with the sequence of mgu's $\theta'_0, \dots, \theta'_n$ such that $\theta'_0 ... \theta'_n$ is more general than $\theta\theta_0 ... \theta_n$.*

PROOF. By a straightforward refinement of the proof of the Substitution lemma 3.16. □

LEMMA 3.20. *Let P be a program and N a goal. Suppose that θ is a correct answer substitution for $P \cup \{N\}$. Then the empty substitution is a computed answer substitution for $P \cup \{N\theta\}$.*

PROOF. Let $x_1,...,x_n$ be the variables of $N\theta$. Enrich the language of P by adding new constants $a_1,...,a_n$ and let γ be the substitution $\{x_1/a_1,...,x_n/a_n\}$. $P \cup \{N\theta\gamma\}$ is inconsistent so by the Completeness Theorem 3.15 there exists an *SLD* - refutation of $P \cup \{N\theta\gamma\}$. We can assume that the variables $x_1,...,x_n$ do not appear in the clauses used in this refutation. But $N\theta\gamma$ is ground so the answer substitution computed by this refutation is the empty substitution. By textually replacing in this refutation a_i by x_i, for $i = 1,...,n$, we obtain an *SLD* - refutation of $P \cup \{N\theta\}$ with the empty substitution as the computed answer substitution. □

We are now ready to prove the desired theorem.

PROOF OF THEOREM 3.18. By the above lemma there exists an *SLD* - refutation of $P \cup \{N\theta\}$ with the empty substitution as the computed answer substitution. Let $\theta_0, \dots, \theta_n$ be its sequence of *mgu*'s. By the Lifting lemma 3.19 there exists an *SLD* - refutation of $P \cup \{N\}$ with the sequence of *mgu*'s $\theta'_0, \dots, \theta'_n$ such that $\theta'_0 ... \theta'_n$ is more general than $\theta\theta_0 ... \theta_n$.

Let $\gamma | N$ denote restriction of the substitution γ the variables of N. Then $\theta'_0 ... \theta'_n | N$ is more general than $\theta\theta_0 ... \theta_n | N$. But the former is the computed answer substitution of the *SLD* - refutation of $P \cup \{N\}$ whereas the latter equals $\theta | N$. □

3.9. Strong completeness of the SLD - resolution

Another way to generalize the Completeness Theorem is by taking selection rules into account. We follow here the presentation of APT and VAN EMDEN [AVE].

THEOREM 3.21. (Strong completeness of *SLD*-resolution) (HILL [H]). *Let P be a program and N a goal. Suppose that $P \cup \{N\}$ is inconsistent. Then every SLD-tree with N as root is successful.*

This theorem states that if $P \cup \{N\}$ is inconsistent then there exists an *SLD* - refutation of $P \cup \{N\}$ via every selection rule.

To prove it we first introduce the following notion. Given a program P we call a goal N *k-refutable*, $k \geqslant 1$, if in every *SLD*-tree with N as root there exists the empty clause with a path length from the root of at most k.

Another straightforward refinement of the proof of Substitution lemma yields the following.

LEMMA 3.22. *Let P be a program, N a goal and θ a substitution. Suppose that $N\theta$ is k-refutable. Then N is k-refutable.* □

Next two lemmata generalize corresponding facts about refuted goals.

LEMMA 3.23. *Let P be a program and let $F_1,...,F_n$ be sequences of atoms. Assume that $F_1,...,F_n$ have no variables in common. If each $\leftarrow F_i$ is k_i - refutable for $i = 1,...,n$ then $\leftarrow F_1,...,F_n$ is $k_1+...+k_n$ - refutable.*

PROOF. By straightforward induction on $k_1+...+k_n$. □

LEMMA 3.24. *If A is in the least model of P, then for some k $\leftarrow A$ is k - refutable.*

PROOF. By repeating the argument from the proof of Lemma 3.17 using the above lemma with each F_i being a single ground atom. □

We can now prove the strong completeness of *SLD* - resolution.

PROOF OF THEOREM 3.21. By repeating the argument from the proof of the Completeness Theorem 3.15 using Lemmas 3.24, 3.23 and 3.22. □

Summarizing the results obtained in Sections 3.4, 3.6, 3.7 and the present one we obtain the following characterizations of the success set.

THEOREM 3.25. (Success Theorem) *Consider a program P and a ground atom A. Then the following are equivalent:*
(a) A is in the success set of P.
(b) $A \in T_P\uparrow\omega$.
(c) Every *SLD*-tree with $\leftarrow A$ as root is successful.
(d) $P \models A$.

PROOF. First note that by Corollary 3.6 and the Characterization Theorem 3.13 i)

$$P \models A \text{ iff } A \in M_P.$$

The rest follows by the Characterization Theorem 3.13 iv), Corollary 3.14, Lemma 3.17 and Lemma 3.24. □

The strong completeness theorem shows that when searching for a refutation of a goal any *SLD*-tree is a complete search space. Of course whether a refutation will be actually found in a successful *SLD*-tree depends on the tree search algorithm used.

Note that in fact we proved more.

THEOREM 3.26. *Let P be a program and N a goal. If* $P \cup \{N\}$ *is inconsistent then for some k N is k - refutable.*

PROOF. By inspection of the proof of the Strong Completeness Theorem 3.21. □

This indicates that given a program P when searching for a refutation of a goal N it is enough to explore any *SLD*-tree till a certain depth depending only on N. However, this depth as a function of the goal N is in general not computable. This is an immediate consequence of the results proved in the next chapter.

3.10. Procedural versus declarative interpretation

In the last two chapters we studied two ways of interpretating the logic programs. They are sometimes referred to as a procedural and declarative interpretation.

Procedural interpretation explains *how* the programs compute, i.e. what is the computational mechanism which underlies the program execution. In the framework of programming languages semantics it is sometimes referred to as the operational semantics.

On the other hand, *declarative interpretation* provides the meaning of a program, that is it attempts to answer the question what semantically follows from the program without analyzing the underlying computational mechanism. In such a way declarative interpretation provides a specification for any underlying computational mechanism, i.e. it explains *what* should be computed by the program. In the framework of programming language semantics it corresponds with the denotational semantics.

To summarize the above we can say that procedural interpretation is concerned with the *method* whereas declarative interpretation is concerned with the *meaning*. Any form of a completeness theorem can be viewed as a proof of a match between these two interpretations. In practice of course this match can be destroyed when, as explained at the end of the previous section, the computational mechanism is supplemented by an incomplete (tree) search algorithm.

3.11. Bibliographic remarks

The name *immediate consequence operator* was introduced in CLARK [C1]. GALLIER [G] presents a different proof of the completeness of the *SLD* - resolution based on the use of Gentzen systems. The strongest completeness result is that of CLARK [C1] which combines the claims of Theorems 3.18 and 3.21. LLOYD [L] provides a rigorous proof of this theorem.

4. COMPUTABILITY

4.1. Computability versus definability

Once we defined *how* logic programs compute and analyzed the relation between the proof theoretic and semantic aspects, let us reflect on the question *what* objects logic programs compute. We show here that logic programs are *computationally complete* in the sense that they have the same computational power as recursive functions.

Assume that the language L has at least one constant, so that the Herbrand universe U_L is not empty. Moreover, assume that L has infinitely many relation symbols in every arity. We say that a program P *computes a predicate* $R \subseteq U_L^n$ *using a relation r* if for all $t_1,...,t_n \in U_L$

$$(t_1,...,t_n) \in R \text{ iff there exists an } \textit{SLD-refutation} \text{ of } P \cup \{\leftarrow r(t_1,...,t_n)\}.$$

A semantic counterpart of this definition is obtained by saying that a program P *defines a predicate* $R \subseteq U_L^n$ *using a relation* r if for all $t_1,...,t_n \in U_L$

$$(t_1,...,t_n) \in R \text{ iff } P \models r(t_1,...,t_n).$$

Both definitions presuppose that $L_P \subseteq L$ and $U_{L_P} = U_L$. We have the following result.

THEOREM 4.1. *Let P be a program, R a predicate and r a relation. Then the following are equivalent:*
(a) P computes R using r.
(b) P defines R using r.
(c) For all $t_1,...,t_n \in U_L$

$$(t_1,...,t_n) \in R \text{ iff } r(t_1,...,t_n) \in M_P.$$

PROOF. By the Success Theorem 3.25 and the Characterization Theorem 3.13. □

Thus the question which predicates are computed by logic programs reduces to the question which predicates are defined over their least Herbrand models.

This question has various answers depending on the form of L. We study here the case when L has finitely many but at least one constant and finitely many but at least one function symbol. Then the Herbrand universe U_L is infinite. The assumption that the set of constants and the set of functions are finite allows us to reverse the question and analyze for a given program P which predicates it computes over its Herbrand universe U_{L_P}. The assumption that in each arity the set of relations is infinite allows us to construct new clauses without syntactic constraints.

4.2. Enumerability of U_L

We call a binary predicate R on U_L an *enumeration of* U_L if R defines the successor function on U_L. In other words, R is an enumeration of U_L if we have $U_L = \{f_R^n(u){:}n<\omega\}$ where u is some fixed ground term and f_R is a one-one function defined by $f_R(x) = y$ iff $(x,y) \in R$.

As a first step towards a characterization of predicates computable by logic programs we prove the following result due to ANDREKA and NEMETI [AN]. Our presentation is based on BLAIR [B2].

THEOREM 4.2. (Enumeration Theorem) *There exists a program* successor *which computes an enumeration of U_L using a binary relation* succ.

PROOF. The construction of the program *successor* is rather tedious. First we define the enumeration *enum* of U_L which will be computed.

We start by defining inductively the notion of height of a ground term. We put

height$(a) = 0$ for each constant a,

height$(f(t_1,...,t_n))$ = max(height(t_1),...,height(t_n))+1.

Next, we define a well-ordering on all ground terms. To this purpose we first order all constants and all function symbols in some way. We extend this ordering inductively to all ground terms of height $\leqslant n$ $(n > 0)$ by putting

$f(s_1,...,s_k) < g(t_1,...,t_m)$ iff

(height$(f(s_1,...,s_k)),f,s_1,...,s_k) \prec$ (height$(g(t_1,...,t_m)),g,t_1,...,t_m)$.

Here $\prec$ is a lexicographic ordering obtained from the ordering of natural numbers, ordering of function symbols and the already defined ordering $<$ on ground terms of height $<n$. This extension is compatible with the fragment of $<$ defined so far. By induction $<$ is defined on all ground terms.

>From the following three observations and the assumption about the number of constants and function symbols it follows that $<$ is a well-ordering of type ω:

a) If height(s) $<$ height(t) then $s < t$.

b) If height($f(s_1,...,s_k)$) $=$ height($g(t_1,...,t_m)$) and f is smaller than g in the chosen ordering then $f(s_1,...,s_k) < g(t_1,...,t_m)$.

c) If height($f(s_1,...,s_i,s_{i+1},...,s_k)$) $=$ height($f(s_1,...,s_i,t_{i+1},...,t_k)$) and $s_{i+1} < t_{i+1}$ then $f(s_1,...,s_i,s_{i+1},...,s_k) < f(s_1,...,s_i,t_{i+1},...,t_k)$.

We now define *enum* to be the graph of the $<$-successor function. Note that

d) If t is the $<$-maximal term of height n then its $<$-successor is the $<$-minimal term of height $n+1$.

e) Otherwise, the $<$-successor of $t = f(t_1,...,t_n)$ is obtained by first locating the rightmost term t_i whose (already defined) $<$-successor t'_i has the height smaller than the height of t. Then $f(t_1,...,t_{i-1},t'_i,a,...,a)$ is the $<$-successor of t, where a is the $<$-least constant.

To compute the relation *enum* we systematically translate its definition into clauses. We proceed by the following steps.

1) For counting purposes we identify a subset N_L of U_L with the set of natural numbers N. Let f_0 be the smallest function in the chosen ordering. We put

$$N_L = \{\hat{n} : n \in N\}$$

where $\hat{0} = a$ and for each n, $\widehat{n+1} = f_0(a,...,a,\hat{n})$.

The following program *Nat* computes N_L using a relation *nat*:

$$nat(a) \leftarrow,$$

$$nat(f_0(a,...,a,x)) \leftarrow nat(x).$$

In turn, the program S_L obtained by adding to *Nat* the clause

$$s_L(x,f_0(a,...,a,x)) \leftarrow nat(x)$$

computes the successor relation on N_L using a relation s_L.

2) Using the programs *Nat* and S_L the definition of the height function can now be translated into a program *height* with a binary relation h such that

$$height \models h(t,k) \text{ iff } t \text{ is a ground term of height } n, \text{ where } k = \hat{n}.$$

3) Note that $\hat{n}$ is the $<$-minimal term of height n. Thus adding a clause $\min(x,x) \leftarrow nat(x)$ we get a program *minimum* such that

$$minimum \models \min(t,k) \text{ iff } t \text{ is the } <-\text{minimal term of height } n, \text{ where } k = \hat{n}.$$

Let now b be the $<$-largest constant and f_1 the largest function in the chosen ordering. Note that the $<$-maximal term of height 0 is b, of height 1 $f_1(b,...,b)$ etc. Thus adding clauses

$$\max(b,a) \leftarrow,$$

$$\max(f_1(x,...,x),y') \leftarrow \max(x,y),s_L(y,y')$$

we get a program *maximum* such that

$$maximum \models \max(t,k) \text{ iff } t \text{ is the } <-\text{maximal term of height } n, \text{ where } k = \hat{n}.$$

4) Using the above auxiliary definitions the program *successor* can now be constructed by translating the statements d) and e) into clauses. The details are straightforward though lengthy and we omit them.

This concludes the proof. □

4.3. Recursive functions

To characterize the predicates computable by logic programs we need to recall the basic concepts of the recursion theory as developed by S.C. Kleene. We follow here SHOENFIELD [S].

For brevity denote the sequence $a_1,...,a_n$ by $\bar{a}$. Let for $i = 1,...,n$ the projection function $\mathbf{P}_i^n$ be defined by

$$\mathbf{P}_i^n(\bar{a}) = a_i.$$

For a given predicate $R \subseteq N^n, K_R$ stands for its characteristic function defined by

$$K_R(\bar{a}) = 1 \text{ iff } \bar{a} \in R$$

$$K_R(\bar{a}) = 0 \text{ iff } \bar{a} \notin R.$$

We define the class of (total) *recursive functions* over N inductively by putting

R1. The functions $\mathbf{P}_i^n$, $+$, $\times$ and $K_<$ are recursive.

R2. If $g,h_1,...,h_k$ are recursive functions and f is defined by

$$f(\bar{a}) = g(h_1(\bar{a}),...,h_k(\bar{a}))$$

then f is recursive.

R3. Let g be a recursive function such that

$$\forall \bar{a} \exists b \; g(\bar{a},b) = 0.$$

Then the function f defined by

$$f(\bar{a}) = \mu b. \; g(\bar{a},b) = 0$$

is recursive. Here $\mu b.\ R$ stands for the least b such that R holds.

A predicate over N is *recursive* if its characteristic function is recursive. A predicate R is *recursively enumerable* if for some recursive predicate S

$$\bar{a} \in R \text{ iff } \exists b (\bar{a},b) \in S.$$

A predicate R is *R.E. complete* if for every recursively enumerable predicate S

$$\bar{a} \in S \textit{ iff } f(\bar{a}) \in R$$

for some recursive function f.

R.E. complete predicates are not recursive. It is a well known fact that there exists a recursively enumerable predicate which is R.E. complete.

In the sequel we shall use various well known simple results from the theory of recursive functions. We also rely on some standard techniques like coding. This allows us to investigate the complexity of subsets of the Herbrand base B_L as its elements can be coded by natural numbers.

We have the following simple result.

THEOREM 4.3. *For every program P, M_P is recursively enumerable.*

PROOF. By the Characterization Theorem 3.13iv) we have $A \in M_P$ iff for some relation p and $t_1,...,t_n \in U_P$, $A = p(t_1,...,t_n)$ and $\exists k\, p(t_1,...,t_n) \in T_P \uparrow k$.

The result now follows by the standard techniques of the recursion theory because the predicate

$$\{(k,A) : A \in T_P \uparrow k\}$$

is, after appropriate coding, recursive. □

4.4. Computability of recursive functions

The Herbrand universe U_L does not coincide with natural numbers but thanks to the Enumeration Theorem 4.2 we can make such an identification. This allows us to transfer the notions of the recursion theory from N to U_L.

We now prove the following theorem.

THEOREM 4.4. (Computability Theorem) (ANDREKA and NEMETI [AN]) *For every recursive function f there is a program P which computes the graph of f using a relation p_f.*

PROOF. We assume that each program here given incorporates the program *successor* which uses different relations than those used here. We proceed by induction on the construction of recursive functions.

ad R1. We can define + in terms of the successor by simply rewriting two well known axioms of Peano arithmetic as clauses:

$$p_+(x,\hat{0},x) \leftarrow,$$

$$p_+(x,y,z) \leftarrow succ(y',y), succ(z',z), p_+(x,y',z').$$

Other functions admit equally straightforward presentations.

ad R2. Suppose by induction that there exist programs $P_0,...,P_k$ computing the graphs of functions $g,h_1,...,h_k$ using the relations $p_g,p_{h_1},...,p_{h_k}$, correspondingly. We can assume that $P_0,...,P_k$ have no relations in common apart of those occurring in *successor.* Then the program $P_0 \cup \cdots \cup P_k$ augmented by the clause

$$p_f(x_1,...,x_t,x_{t+1}) \leftarrow p_{h_1}(x_1,...,x_t,y_1),...,p_{h_k}(x_1,...,x_t,y_k),p_g(y_1,...,y_k,x_{t+1})$$

computes the graph of the function f defined as in R2.

ad R3. Let f and g be recursive functions as given in R3. By induction there exists a program P_g which computes the graph of g using a relation p_g.

The program P_f is obtained by adding to P_g the following clauses with a new relation r:

$$p_f(x_1,...,x_k,x_{k+1}) \leftarrow p_g(x_1,...,x_{k+1},\hat{0}), r(x_1,...,x_{k+1}),$$

$$r(x_1,...,x_k,\hat{0}) \leftarrow,$$

$$r(x_1,...,x_k,y) \leftarrow succ(y',y), r(x_1,...,x_k,y'), p_g(x_1,...,x_k,y',z), p_<(\hat{0},z).$$

The intended meaning of $r(x_1,...,x_{k+1})$ is: $\forall y(y<x_{k+1} \rightarrow g(x_1,...,x_k,y)>0)$. Note that under this interpretation $r(x_1,...,x_k,0)$ holds and $r(x_1,...,x_k,n+1)$ iff $r(x_1,...,x_k,n) \wedge g(x_1,...,x_k,n)>0$ and this is exactly what the last two clauses express. □

COROLLARY 4.5. *A predicate R on U_L is recursively enumerable iff some program P computes it using a relation r.*

PROOF. ⇒. Suppose that for some recursive predicate S

$$\bar{a} \in R \text{ iff } \exists b(\bar{a},b) \in S.$$

Let P_S be the program computing the characteristic function K_S of S using a relation p_S. Then the program P_S augmented by the clause

$$p_R(x_1,...,x_k) \leftarrow p_S(x_1,...,x_k,y,\hat{1})$$

computes the predicate R using relation p_R.

⇐. By Theorem 4.1 and Theorem 4.3. □

This allows us to prove the converse of the Computability Theorem.

COROLLARY 4.6. *Suppose that a program P computes the graph of a total function using some relation. Then this function is recursive.*

PROOF. A total function is recursive iff its graph is recursively enumerable. □

Also, we can obtain the following characterization of the recursion theoretic complexity of M_P.

COROLLARY 4.7. *For some program P, M_P is R.E. complete. A fortiori M_P is not recursive.*

PROOF. Let R be a recursively enumerable, R.E. complete predicate on U_L. By Corollary 4.5 and Theorem 4.1 we have for all $a \in U_L$

$$a \in R \text{ iff } r(a) \in M_P,$$

where P is a program which computes R using a relation r. This shows that M_P is R.E. complete, as well. □

We conclude this section by mentioning the following strengthening of the Computability Theorem 4.4 which we shall use in the next section. Following BLAIR [B2] we call a program P *determinate* if $T_P\uparrow\omega = T_P\downarrow\omega$.

THEOREM 4.8. (BLAIR [B2]). *For every recursive function f there is a determinate program P which computes the graph of f using a relation p_f.* □

The proof is based on a detailed analysis of the programs constructed in the proof of the Computability Theorem 4.4 and we omit it.

4.5. Closure ordinals of $T_P\downarrow$

In this section we study the downward closure ordinals of the operators T_P for programs P.

We noted in Section 3.6 that for a program P the operator T_P is continuous. However, T_P does not need to be downward continuous. To see this consider the following program P:

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

Then for $n \geq 1$ we have $T_P\downarrow n = \{q(a)\} \cup \{p(f^k(a)) : k \geq n\}$, so $T_P\downarrow\omega = \{q(a)\}$. It follows that $T_P\downarrow(\omega+1) = \emptyset$, hence $||T_P\downarrow|| = \omega+1$ and T_P is not downward continuous. Note that by Lemma 3.11 $gfp(T_P) = T_P\downarrow(\omega+1) = \emptyset$.

This asymmetry is one of the most curious phenomena in the theory of logic programming.

To characterize the downward closure ordinals of the operators T_P we first introduce some definitions. We shall consider well-founded orderings on natural numbers. For a well-founded ordering R we write $a <_R b$ instead of $(a,b) \in R$ and denote by $dom(R)$ its domain.

With each well-founded ordering R we can associate in a standard way an ordinal $||R||$ by means of a transfinite induction:

$||a|| = 0$ if a is the $<_R$ −least element of $dom(R)$,

$||a|| = sup(||b||+1 : b <_R a)$ otherwise,

$||R|| = sup(||a|| : a \in dom(R))$.

An ordinal α is called *recursive* if $\alpha = ||R||$ for some well-founded ordering R which is a recursive

predicate. The least non-recursive ordinal is denoted by ω_1^{ck} (ω_1 of Church and Kleene).
The following theorem characterizes the ordinals $\|T_P\downarrow\|$.

THEOREM 4.9. (BLAIR [B1])
i) *For every $\alpha \leqslant \omega_1^{ck}$ there exists a program P such that $\|T_P\downarrow\| = \alpha$.*
ii) *For every program P $\|T_P\downarrow\| \leqslant \omega_1^{ck}$.*

PROOF. i) It is clear how to construct for any natural number $n \geqslant 0$ a program P such that $\|T_P\downarrow\| = n$. Suppose now that $\omega \leqslant \alpha < \omega_1^{ck}$. For some β we have $\alpha = \omega + \beta$.
Assume from now on that L has exactly one, unary, function symbol f and exactly one constant a. Then U_L coincides with natural numbers.
Let R be a recursive well-founded ordering such that $\|R\| = \beta$. Given a relation q we denote by [q] the set of all ground atoms of the form $q(t_1,...,t_n)$.
Let P_1 be the program P from the beginning of this section augmented by the clause

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

Then $T_{P_1}\downarrow\omega = [q]$ and $T_{P_1}\downarrow\alpha = \varnothing$ for $\alpha > \omega$.
By Theorem 4.8 there exists a determinate program P_2 which computes R using some relation r. We can assume that P_1 and P_2 are disjoint. Then for any $\alpha \geqslant \omega$

$$T_{P_2}\downarrow\alpha \cap [r] = R_r,$$

where

$$R_r = \{r(s,t) : (s,t) \in R\}.$$

Let P_3 be the program

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

where q does not appear in P_1 and P_2, and finally let

$$P = P_1 \cup P_2 \cup P_3.$$

Then

$$T_P\downarrow\omega \cap ([q] \cup [r]) = [q] \cup R_r.$$

Thus

$$T_P\downarrow(\omega+1) \cap ([q] \cup [r]) =$$
$$= \{q(s) : s \in dom(R), \|s\| \geqslant 1\} \cup R_r$$

and more generally, for every γ

$$T_P\downarrow(\omega+\gamma) \cap ([q]_P \cup [r]_P) =$$
$$\{q(s) : s \in dom(R), \|s\| \geqslant \gamma\} \cup R_r.$$

Thus for $\gamma < \beta$

$$T_P\downarrow(\omega+\gamma) \neq T_P\downarrow(\omega+\gamma+1).$$

Also

$$T_P\downarrow(\omega+\beta) \cap ([q]_P \cup [r]_P) = R_r,$$

so

$$T_p\downarrow(\omega+\beta) = T_{P_2}\downarrow(\omega+\beta) = T_{P_2}\downarrow\omega$$

and consequently

$$T_P{\downarrow}(\alpha+1) = T_P{\downarrow}\alpha,$$

i.e. $\|T_P{\downarrow}\| = \alpha$.

The proof that for some program P in fact $\|T_P{\downarrow}\| = \omega_1^{ck}$ and the proof of ii) rely on advanced results from the recursion theory and are beyond the scope of this paper. □

4.6. *Bibliographic remarks*

There is a considerable confusion concerning the actual formulation and origin of the results of the first part of this chapter. The statement that logic programming has a full power of recursion theory is usually attributed to TÄRNLUND [T] who showed that Turing machines can be simulated using logic programs. However, in his proof additional function symbols are used and the paper of ANDREKA and NEMETI [AN] actually appeared earlier as a technical report.

A syntactically stronger form of the Computability Theorem 4.4 in case when L has exactly one, unary function symbol and exactly one constant was proved in SEBELIK and STEPANEK [SS]. For such L the Computability Theorem 4.4 is implicitly contained in SMULLYAN [Sm]. Related results were proved in ITAI and MAKOWSKY [IM], KOWALSKI [K3], SHEPHERDSON [She1] and SONENBERG and TOPOR [ST]. The last paper discusses all these results in detail. BÖRGER [Bo] discusses connections between logic programming and computational complexity of various classes of formulas.

That T_P does not need to be downward continuous was originally observed by Andreka and Nemeti, and Clark.

5. NEGATIVE INFORMATION

5.1. *Non-monotonic reasoning*

SLD resolution is an example of a *sound* method of reasoning because only true facts can be deduced using it. More precisely, we call here a reasoning method "$\vdash$" *sound* if for all variable-free formulas ϕ $P\vdash\phi$ implies $P\models\phi$, where $P\vdash\phi$ denotes that ϕ can be proved from a program P. And we call "$\vdash$" *weakly sound* if $P\vdash\phi$ implies consistency of $P\cup\{\phi\}$. Now, putting (see Section 2.5) $P\vdash_{SLD}$ $\exists x_1...\exists x_s$ $(A_1\wedge...\wedge A_k)$ iff there exists an *SLD*-refutation of $P\cup\{\leftarrow A_1,...,A_k\}$, we see that "$\vdash_{SLD}$" is sound by virtue of Soundness Theorem 3.2.

We call a reasoning method "$\vdash$" *effective* if for any program P the set $\{\phi : P\vdash\phi\}$ is recursively enumerable. Now, "$\vdash_{SLD}$" is easily seen to be effective by using the standard techniques of recursion theory. Effectiveness is a desirable property as it amounts to saying that it is decidable whether an object is a proof of a formula. Ineffective reasoning methods cannot be implemented.

SLD-resolution is also an example of a *monotonic* method of reasoning. We call here a reasoning method "$\vdash$" *monotonic* if for any two programs P and P'

$$P\vdash\phi \text{ implies } P\cup P'\vdash\phi.$$

Otherwise, "$\vdash$" is called *non-monotonic*. Clearly, if there exists an *SLD*-refutation of $P\cup\{N\}$ then also there exists an *SLD*-refutation of $P\cup P'\cup\{N\}$.

However, *SLD*-resolution is a very restricted form of reasoning, because only positive facts can be deduced using it. This restriction cannot be overcome if soundness or monotonicity is to be maintained. More precisely, the following simple yet crucial observation holds.

LEMMA 5.1. *Let "$|\sim$" be a reasoning method such that $P\,|\sim\neg A$ for some negative ground literal $\neg A$. Then "$|\sim$" is not sound. Moreover, if "$|\sim$" is weakly sound then it is not monotonic.*

PROOF. Note that the Herbrand base is a model of P but not a model of $\neg A$. Thus "$|\sim$" is not sound. Suppose it is monotonic. Then we get $P \cup \{A\} \mid\sim \neg A$. But $P \cup \{A\} \cup \{\neg A\}$ is inconsistent, so "$\mid\sim$" is not weakly sound. □

However, in some applications it is natural to require that also negative information can be deduced.

EXAMPLE 5.2. Consider $P = \{male(Jerry) \leftarrow, female(Jacky) \leftarrow\}$. Then we naturally expect that $\neg female(Jerry)$ and $\neg male(Jacky)$. □

By Lemma 5.1 any such extension of *SLD*-resolution leads to a non-monotonic reasoning.

5.2. Closed world assumption

One natural possibility is to consider here the following rule (or rather meta-rule):

$$\frac{A \textit{ cannot be proved } P}{\neg A}$$

where A is a ground atom.
This rule is usually called the *closed world assumption* (CWA). It was first considered in REITER [R]. The notion of provability referred in the hypothesis is that in the first order logic. For our purposes it is sufficient to know that it is equivalent here to provability by means of the *SLD*-resolution.

Given now a program P consider the set

$$CWA(P) = \{\neg A : A \textit{ is a ground atom for which there does not exist an } SLD-\textit{refutation of } P \cup \{\leftarrow A\}\}.$$

We have

LEMMA 5.3. $\neg A \in CWA(P)$ *iff* $A \in B_P \setminus M_P$.

PROOF. We have $\neg A \in CWA(P)$ iff A is not in the success set of P. The claim now follows by Corollary 3.14 and Lemma 3.16. □

As an immediate consequence we get

THEOREM 5.4. (REITER [R]). *For any program P, $P \cup CWA(P)$ is consistent.* □

Thus closed world assumption viewed as a reasoning method is weakly sound. Unfortunately, it is not an effective reasoning method. Namely, we have the following theorem.

THEOREM 5.5. *Assume that L is as in Chapter 4. Then for some program P the set $CWA(P)$ is not recursively enumerable.*

PROOF. By Corollary 4.7 there exists a program P such that M_P is a recursively enumerable but not recursive subset of U_L. Then by a well known theorem $B_P \setminus M_P$, the complement of M_P, is not recursively enumerable.

This concludes the proof in view of Lemma 5.3. □

5.3. *Negation as failure rule*

A way out of this dilemma is to adopt same more restrictive forms of unprovability. A natural possibility is to consider $\neg A$ proved when an attempt to prove A using *SLD*-resolution fails finitely. This leads to the following definitions.

An *SLD*-tree is *finitely failed* if it is finite and contains no empty clause. Given a program P its *finite failure set* is the set of all ground atoms A such that there exists a finitely failed *SLD*-tree with $\leftarrow A$ as root.

We now replace *CWA* by the following rule:

$$\frac{A \textit{ is in the finite failure set of } P}{\neg A}$$

introduced in CLARK [C] and called *negation as failure* rule. (A more appropriate name would be: negation as a *finite* failure rule.)

First of all it is useful to note that negation as failure rule viewed as a reasoning method is weakly sound. Indeed, if A is in the finite failure set of P then by the strong completeness of *SLD*-resolution (Theorem 3.21) $\neg A$ is in $CWA(P)$, so it suffices to apply Lemma 5.3.

Thus by Theorem 5.4 negation as failure is a non-monotonic form of reasoning. It is also an effective form of reasoning because it is decidable whether a finite tree is a finitely failed *SLD*-tree.

Finally, observe that using negation as failure rule we can trivially deduce $\neg$*female*(*Jerry*) and $\neg$*male*(*Jacky*) from the program P given in Example 5.2.

5.4. *Characterizations of finite failure*

We now provide two characterizations of the finite failure, due to APT and VAN EMDEN [AVE] and LASSEZ and MAHER [LM]. We follow here the presentation of LLOYD [L].

First we introduce the concept of a fair *SLD*-derivation due to LASSEZ and MAHER [LM]. An *SLD*-derivation is called *fair* if it is either finite or every atom appearing in it is eventually selected. (An atom at the moment of selection will be actually an instantiation of the original version.) For example, second derivation given in Section 2.6 is not fair as the atom *configuration* (y, l_1) is never selected in it. An *SLD*-tree is *fair* if each of its branches is a fair *SLD*-derivation.

THEOREM 5.6. *Consider a program P and a ground atom A. Then the following are equivalent:*
(a) A is in the finite failure set of P.
(b) $A \notin T_P{\downarrow}\omega$.
(c) Every fair SLD-tree with $\leftarrow A$ as root is finitely failed.

To prove that (a) implies (b) we need two simple lemmata which are counterparts of Lemmas 3.22 and 3.23.

LEMMA 5.7. *Consider a program P, a negative clause N and a substitution θ. If $P \cup \{N\}$ has a finitely failed SLD-tree of depth $\leqslant k$ then so has $P \cup \{N\theta\}$.*

PROOF. By a straightforward induction on k. □

LEMMA 5.8. *Consider a program P and sequences of atoms $F_1,...,F_n$. Assume that $F_1,...,F_n$ have no variables in common. If $P \cup \{\leftarrow F_1,...,F_n\}$ has a finitely failed SLD-tree of depth $\leqslant k$ then so has $P \cup \{\leftarrow F_i\}$ for some $i \in \{1,...,n\}$.*

PROOF. By a simple induction on k using an analogous argument as that in the proof of Lemma 3.23. □

PROOF OF THEOREM 5.6
(a) $\Rightarrow$ (b).
We prove a stronger claim namely:

LEMMA 5.9. *Suppose $P \cup \{\leftarrow A\}$ has a finitely failed SLD-tree of depth $\leqslant k$. Then $A \notin T_P{\downarrow}k$.*

PROOF. We proceed by induction on k.
The claim clearly holds when $k=1$. Assume it holds for $k-1$ and suppose by contradiction that $A \in T_P{\downarrow}k$. Then for some clause $B \leftarrow B_1,...,B_n$ in P $A \equiv B\theta$ and $\{B_1\theta,...,B_n\theta\} \subseteq T_P{\downarrow}(k-1)$ for some substitution θ. Thus for some *mgu* γ $A\gamma \equiv B\gamma$ and $\theta = \gamma\sigma$ for some σ.
Hence $\leftarrow(B_1,...,B_n)\gamma$ is the root of a finitely failed *SLD*-tree of depth $\leqslant k-1$. By Lemma 5.7 so is $\leftarrow(B_1,...,B_n)\theta$. Now using Lemma 5.8 with each F_i being a single ground atom we get that $\leftarrow B_i\theta$ is also the root of a finitely failed *SLD*-tree of depth $\leqslant k-1$. By the induction hypothesis $B_i\theta \notin T_P{\downarrow}(k-1)$ which gives the contradiction. □

To prove that (b) implies (c) we need the following lemma.

LEMMA 5.10. *Consider a program P and a goal $\leftarrow A_1,...,A_m$. Suppose there is an infinite fair SLD-derivation $\leftarrow A_1,...,A_m = N_0, N_1,...$ with the sequence of substitutions $\theta_0,\theta_1,....$ Then for every $k \geqslant 0$ there exists $n \geqslant 0$ such that*

$$\bigcup_{i=1}^{m} [A_i\theta_0...\theta_n] \subseteq T_P{\downarrow}k.$$

PROOF. We proceed by induction on k. The claim is clearly true $k=0$. Suppose it holds for $k-1$. Fix $i \in \{1,...,m\}$. By fairness for some $p \geqslant 0$ $\leftarrow A_i\theta_0...\theta_{p-1}$ is selected in the goal N_p. By the induction hypothesis for some $s \geqslant 0$

$$\bigcup_{j=1}^{q} [B_j\theta_p...\theta_{p+s}] \subseteq T_P{\downarrow}(k-1)$$

holds where N_{p+1} is $\leftarrow B_1,...,B_q$. But

$$[A_i\theta_0...\theta_{p+s}] \subseteq T_P(\bigcup_{j=1}^{q} [B_j\theta_p...\theta_{p+s}])$$

so

$$[A_i\theta_0...\theta_{p+s}] \subseteq T_P{\downarrow}k$$

by the monotonicity of T_P.
Thus for each $i \in \{1,...,m\}$ there exists $n_i \geqslant 0$ such that $[A_i\theta_0...\theta_{n_i}] \subseteq T_P{\downarrow}k$. Put now $n = max(n_1,...,n_m)$ □

PROOF OF THEOREM 5.6 CONTINUED
(b)$\Rightarrow$(c).
Suppose that $A \notin T_P{\downarrow}\omega$. Consider a fair *SLD*-tree with $\leftarrow A$ as root. By Lemma 5.10 all of its branches are finite. But this tree does not contain the empty clause. Otherwise by the Success Theorem 3.25 we would have $A \in T_P{\uparrow}\omega \subseteq T_P{\downarrow}\omega$. Thus it is a finitely failed *SLD*-tree.
(c)$\Rightarrow$(a).
Obvious. □

Equivalence between (a) and (b) is due to APT and VAN EMDEN [AVE] and between (a) and (c) due to LASSEZ and MAHER [LM]. The first equivalence can be seen as a theorem dual to the equivalence between (a) and (b) in the Success Theorem 3.25. The second equivalence can be seen as a counterpart of the equivalence between (a) and (c) in the Success Theorem 3.25 where duality is achieved by restricting the attention to fair *SLD*-trees.

5.5. Completion of a program

Another way of inferring a negative information from a logic program is that using the concept of a completion of a program due to CLARK [C].

A program can be seen as a collection of statements of the form "if ... then ---". This does not allow us to conclude negative facts because only positive conclusions are admitted. But treating the clauses as statements of the form "... iff ---" we obtain a stronger interpretation which allows us to draw negative conclusions. In doing so we should exercise some care. For example we wish to interpret the program $\{A\leftarrow B,\ A\leftarrow C\}$ as $A\leftrightarrow B\vee C$ and not as $(A\leftrightarrow B)\wedge(A\leftrightarrow C)$.

First, assume that "$=$" is a new binary relation symbol not appearing in P. We write $s\neq t$ as an abbreviation for $\neg(s=t)$. We perform successively the following steps.

Step 1. Remove terms.

Transform each clause $p(t_1,...,t_n)\leftarrow B_1,...,B_m$ of P into

$$p(x_1,...,x_n)\leftarrow(x_1=t_1)\wedge...\wedge(x_n=t_n)\wedge B_1\wedge...\wedge B_m$$

where $x_1,...,x_n$ are new variables.

Step 2. Introduce existential quantifiers.

Let $y_1,...,y_d$ be the variables of the original clause. Transform each formula $p(x_1,...,x_n)\leftarrow F$ into

$$p(x_1,...,x_n)\leftarrow\exists y_1...\exists y_d F.$$

Step 3. Group similar formulas.

Let

$$p(x_1,...,x_n)\leftarrow F_1,...$$

$$p(x_1,...,x_n)\leftarrow F_k$$

be all formulas obtained in the previous step with a relation p on the left hand side. Replace them by one formula

$$p(x_1,...,x_n)\leftarrow F_1\vee...\vee F_k.$$

If $F_1\vee...\vee F_k$ is empty, replace it by **true**.

Step 4. Handle "undefined" relation symbols.

For each n-ary relation symbol q not appearing in a head of a clause in P add a formula

$$q(x_1,...,x_n)\leftarrow \textbf{false}.$$

Step 5. Introduce universal quantifiers.

Replace each formula $p(x_1,...,x_n)\leftarrow F$ by

$$\forall x_1...\forall x_n(p(x_1,...,x_n)\leftarrow F)$$

Step 6. Introduce equivalence.

In each formula replace "$\leftarrow$" by "$\leftrightarrow$".

We call the intermediate form of P obtained after step 5 the *IF-definition associated with P* and denote it by $IF(P)$. We call the final form the *IFF-definition associated with P* and denote it by $IFF(P)$. By $ONLY\text{-}IF(P)$ we denote the set of formulas obtained from $IF(P)$ by replacing everywhere "$\leftarrow$" by "$\rightarrow$".

EXAMPLE 5.11.

i) Reconsider the program

$$P = \{male(Jerry) \leftarrow, female(Jacky) \leftarrow\}.$$

Then

$$IFF(P) = \{\forall x(male(x) \leftrightarrow x = Jerry),$$
$$\forall x(female(x) \leftrightarrow x = Jacky)\}.$$

Note that both $IFF(P) \models \neg male(Jacky)$ and $IFF(P) \models \neg female(Jerry)$ provided we interpret "=" as identity.

ii) Consider the program

$$P = \{link(a,b) \leftarrow, link(b,c) \leftarrow,$$
$$connected(x,y) \leftarrow link(x,y),$$
$$connected(x,y) \leftarrow link(x,z), connected(z,y)\}.$$

Then

$$IFF(P) = \{\forall x \forall y(link(x,y) \leftrightarrow (x=a \wedge y=b)$$
$$\vee (x=b \wedge y=c),$$
$$\forall x \forall y(connected(x,y) \leftrightarrow link(x,y)$$
$$\vee \exists z(link(x,z) \wedge connected(z,y)))\}.$$

Note that both $IFF(P) \models connected(a,c)$ and $IFF(P) \models \neg connected(a,a)$ provided we interpret "=" as identity. □

We thus see that a negative information can be inferred using the *IFF*-definition provided we interpret the relation symbol "=" properly. To this purpose we extend the interpretation of a first order language so that "=" is interpreted as identity.

Let I be an interpretation of the first order language associated with P. We put for any two terms t_1 and t_2 and a state σ

$$I \models_\sigma t_1 = t_2 \text{ iff } \sigma(t_{1I}) \text{ and } \sigma(t_{2I}) \text{ are the same elements of the domain of } I.$$

This does not yet solve the problem because even though Jerry and Jacky or a and b are different constants, they still can become equal under some interpretation. To exclude such situations we add to the *IFF*-definitions the following *free equality axioms* which enforce proper interpretation of "=".

(1) $f(x_1,...,x_n) = f(y_1,...,y_n) \rightarrow x_1 = y_1 \wedge ... \wedge x_n = y_n$
for each function f,

(2) $f(x_1,...,x_n) \neq g(y_1,...,y_m)$
for all pairs of functions f,g such that $f \not\equiv g$,

(3) $x \neq t$
for each variable x and term t such that $x \not\equiv t$ and x occurs in t.

Here, similarly as in the proof of the Unification Theorem 2.2 we identify constants with *O*-ary functions. Thus (1) includes $c = c$ for every constant c as a special case and (2) includes $c \neq d$ for all pairs of distinct constants as a special case.

Observe the striking similarity between the strong equality axioms and steps 1, 2 and 5 of the unification algorithm used in the proof of the Unification Theorem 2.2. We shall exploit it in Section 5.7.

Given now a program P we denote by $comp(P)$ the set of formulas $IFF(P)$ augmented by the free equality axioms. $comp(P)$ is called the *completion* of P.

5.6. Models of completions

In order to assess the proof theoretic power of completions we study their models first. However, in contrast to the case of models of logic programs it is not sufficient to restrict here attention to Herbrand models. This is the content of a proposition we prove at the end of this section.

Therefore we shall consider here arbitrary models but we shall study them by means of a natural generalization of the immediate consequence operator T_P. First, following JAFFAR, LASSEZ and LLOYD [JLL], we introduce the concept of a *pre-interpretation* for a first order language L. Its definition is identical to that of an interpretation given in Section 3.1 with the exception that clause d) explaining the meaning of relations is dropped. We then say that an interpretation I *is based on* J if I is obtained from J by assigning to each n-ary relation r of L an n-ary predicate r_I on the domain of J, that is by fixing the meaning of the relations of L. Thus each interpretation based on J can be uniquely identified with a set of *generalized atoms*, i.e. objects of the form $r(a_1,...,a_n)$ where r is an n-ary relation of L and $a_1,...,a_n$ are elements of the domain of J. That is what we shall do in the sequel.

We now generalize the operator T_P so that it acts on interpretations based on a given pre-interpretation. To this purpose we first introduce the following useful notation.

Fix an interpretation I. Let $A = p(t_1,...,t_n)$ be an atom and let σ be a state over I. Then we denote by $A\sigma$ the generalized atom $p(\sigma(t_{1I}),...,\sigma(t_{nI}))$.

Let now J be a pre-interpretation and let I be an interpretation based on J. We put for a program P and a generalized atom D

$$\begin{aligned} D \in T_P^J(I) \text{ iff } & \text{ for some state } \sigma \text{ over } I \\ & \text{and a clause } B \leftarrow B_1,...,B_n \text{ of } P \\ & \text{we have } D = B\sigma \text{ and } I \models_\sigma B_1 \wedge ... \wedge B_n. \end{aligned}$$

Thus T_P^J maps interpretations based on J to interpretations based on J. Similarly as in the case of T_P, the operator T_P^J is continuous. The following is an obvious generalization of Proposition 3.7.

PROPOSITION 5.12. *For a program P and a pre-interpretation J, an interpretation I based on J is a model of P iff $T_P^J(I) \subseteq I$.* □

We now wish to prove a similar characterization for models of completions. To this purpose we first note the following.

LEMMA 5.13. *For a program P, P and $IF(P)$ are semantically equivalent.*

PROOF. In steps 1,2,3,5 each formula is replaced by a semantically equivalent one. In turn, in step 4 valid formulas are introduced. □

COROLLARY 5.14. *For a program P and a pre-interpretation J, an interpretation I based on J is a model of $IF(P)$ iff $T_P^J(I) \subseteq I$.* □

We also have the following.

THEOREM 5.15. *For a program P and a pre-interpretation J, an interpretation I based on J is a model of $ONLY\text{-}IF(P)$ iff $T_P^J(I) \supseteq I$.*

To prove it we first need the following lemma.

LEMMA 5.16. *Let I be an interpretation based on a pre-interpretation J and P a program. Let $\forall x_1...\forall x_n(p(x_1,...,x_n) \rightarrow F)$ be a formula in $ONLY\text{-}IF(P)$. Then for every state σ over I*

$$p(x_1,...,x_n)\sigma \in T_P^J(I) \text{ iff } I \models_\sigma F.$$

PROOF. If p does not appear in a head of a clause in P then both sides of the claimed equivalence are necessarily false. Otherwise
$p(x_1,...,x_n)\sigma \in T_P^J$ (I)

iff for some state τ over I and some clause $p(t_1,...,t_n) \leftarrow B_1,...,B_m$ of P
$I \models_\tau B_1 \wedge ... \wedge B_m$ and $\sigma(x_i) = \tau(t_i)$ for $i = 1,...,n$
iff $I \models_\sigma E$. □

PROOF OF THEOREM 5.15. We have
J is a model of $ONLY\text{-}IF(P)$

iff for every formula
$\forall x_1 ... \forall x_n (p(x_1,...,x_n) \rightarrow F)$ in $ONLY\text{-}IF(P)$
and every state σ over I
$p(x_1,...,x_n)\sigma \in I$ implies $I \models_\sigma F$
iff (by Lemma 5.16)
for every relation p of P and state σ over I
$p(x_1,...,x_n)\sigma \in I$ implies $p(x_1,...,x_n)\sigma \in T_P^J(I)$
iff $T_P^J(I) \supseteq I$. □

Combining Corollary 5.14 and Theorem 5.15 we get the following characterization of the models of $IFF(P)$:

THEOREM 5.17. *Let P be a program and J a pre-interpretation. Then an interpretation I based on J is a model of IFF(P) iff* $T_P^J(I) = I$.

PROOF. $IFF(P)$ is semantically equivalent to the set $IF(P) \cup ONLY\text{-}IF(P)$ of formulas. □

Restricting attention to Herbrand interpretations we can now draw some consequences about the completion of P.

THEOREM 5.18 (APT and VAN EMDEN [AVE]). *Let P be a program.*
i) *A Herbrand interpretation I is a model of comp(P) iff* $T_P(I) = I$.
ii) *comp(P) has a Herbrand model.*
iii) *For any ground atom A, comp(P)* $\cup$ *{A} has a Herbrand model iff* $A \in gfp(T_P)$.

PROOF.
i) Every Herbrand interpretation is a model of the free equality axioms.
ii) By i) and the Characterization Theorem 3.13.
iii) By i), Lemma 3.12 ii) and Theorem 3.10. □

Moreover, we have the following observation which brings us to the end of this section.

PROPOSITION 5.19. *There is a program P and a ground atom A such that comp(P)* $\cup$ *{A} has a model but it has no Herbrand model.*

PROOF. Take the program P considered at the end Section 4.5. As $gfp(T_P) = \emptyset$, by Theorem 5.18 iii) $comp(P) \cup \{q(a)\}$ has no Herbrand model.
However, $comp(P) \cup \{q(a)\}$ is consistent. Indeed, take as a domain of the interpretation a disjoint union $\mathbb{Z} \cup \mathbb{N}$ of the set of integers and the set of natural numbers. Interpret the constant a as zero in the set $\mathbb{N}$ and f as a successor function, both on the set $\mathbb{Z}$ and the set $\mathbb{N}$. Finally, interpret p as true for all elements of $\mathbb{Z}$ and q true only for the zero of $\mathbb{N}$. The resulting interpretation is a model of $comp(P) \cup \{q(a)\}$. □

In the next section we provide a characterization of a finite failure which provides a more direct proof of the above proposition.

5.7. Soundness of the negation as failure rule

Recall that completion of program was introduced in order to infer negative information from a program. We now relate it to the previously studied way of deducing a negative information - that by means of the negation as failure rule. To this purpose we first investigate models of the free equality axioms. Assume a program P and denote these axioms by Eq. As Eq do not refer to relations, it makes sense to say that a pre-interpretation J is a model of Eq. For each ground term t denote its value in the domain of J by t_J. We write $J \models_\sigma s = t$ when $\sigma(s_J)$ equals $\sigma(t_J)$.

LEMMA 5.20. *Let J be a pre-interpretation which is a model of Eq. Then the domain of J contains an isomorphic copy of U_P.*

PROOF. It suffices to show that for all ground terms s,t $s_J = t_J$ implies $s \equiv t$. We proceed by induction on the structure of ground terms.

If $s_J = t_J$ then by axioms 1 and 2 s and t are either the same constants or are respectively of the form $f(s_1, \ldots, s_n)$ and $f(t_1, \ldots, t_n)$. The claim now follows by axiom 1 and the induction hypothesis. □

In the sequel we shall identify this isomorphic copy with U_P. Given a pre-interpretation J let B_J stand for the set of its all generalized atoms. If J is a model of Eq then by the above lemma B_J contains an isomorphic copy of the Herbrand base B_P. We identify this copy with B_P.

The following lemma clarifies the relation between the unification and free equality axioms.

LEMMA 5.21. (CLARK [C])

(a) *If the set $\{s_1 = t_1, \ldots, s_n = t_n\}$ has a unifier then for some of its mgu $\{x_1/u_1, \ldots, x_k/u_k\}$*

$$Eq \models s_1 = t_1 \wedge \ldots \wedge s_n = t_n \rightarrow x_1 = u_1 \wedge \ldots \wedge x_k = u_k.$$

(b) *If the set $\{s_1 = t_1, \ldots, s_n = t_n\}$ has no unifier then*

$$Eq \models s_1 = t_1 \wedge \ldots \wedge s_n = t_n \rightarrow \textbf{false}.$$

PROOF. Modify the unification algorithm given in the proof of the Unification Theorem 2.2 as follows. First display each set $\{s_1 = t_1, \ldots, s_n = t_n\}$ of equations as a formula $s_1 = t_1 \wedge \ldots \wedge s_n = t_n$. Then interpret the replacement and deletion steps as operations on these formulas. Interpret the halt with failure action as a replacement of the formula by **false**.

Observe that if ψ is obtained from ϕ by applying one of the steps of the algorithm then Eq $\models \phi \rightarrow \psi$. Indeed, if x does not appear in t then Eq $\models \phi_0 \wedge x = t \wedge \phi_1 \rightarrow (\phi_0 \wedge \phi_1)[x/t]$. Other cases are immediate.

The lemma now follows from the correctness of the unification algorithm. □

Call a substitution θ *invariant over a state* σ if for all x, $\sigma(x) = \sigma(x\theta)$.

COROLLARY 5.22. *Let J be a pre-interpretation which is a model of Eq. If for some state σ*

$$J \models_\sigma s_1 = t_1 \wedge \ldots \wedge s_n = t_n$$

then for some mgu θ of $\{s_1 = t_1, \ldots, s_n = t_n\}$ invariant over σ

$$J \models (s_1 = t_1 \wedge \ldots \wedge s_n = t_n)\theta. \quad □$$

Call now an interpretation I based J *good* if for all sequences of atoms F and all states σ, $I \models_\sigma F$ implies $I \models F\theta$ for some substitution θ.

Obviously not all interpretations are good. But those of interest to us are.

LEMMA 5.23. *Let J be a pre-interpretation which is a model of Eq. Then for every $n \geqslant 0$ $T_P^J \downarrow n$ is good.*

PROOF. We proceed by induction on n.
For $n = 0$ we have $T_P^J \downarrow n = B_J$. But $B_P \subseteq B_J$ so for all sequences of atoms F and all substitutions σ $B_J \models F\delta$ holds.

Assume now the claim holds for some $n \geqslant 0$. Consider a sequence $A_1, \ldots, A_k$ of atoms. Suppose that $T_P^J \downarrow (n+1) \models_\sigma A_1 \wedge \ldots \wedge A_k$ for some state σ. By the definition of T_P^J, for each $i = 1, \ldots, k$ there exists a clause $B_i \leftarrow B_1^i, \ldots, B_{m_i}^i$ in P such that $T_P^J \downarrow n \models_\sigma B_1^i \wedge \ldots \wedge B_{m_i}^i$ and $A_i\sigma = B_i\sigma$. Thus

$$T_P^J \downarrow n \models_\sigma \bigwedge_{\substack{i=1,\ldots,k \\ j=1,\ldots,m_i}} B_j^i.$$

By Corollary 5.22 there exists a substitution θ invariant over σ such that $A_i\theta \equiv B_i\theta$.
By the definition of invariance

$$T_P^J \downarrow n \models_\sigma \bigwedge_{\substack{i=1,\ldots,k \\ j=1,\ldots,m_i}} B_j^i\theta.$$

By the induction hypothesis for some substitution γ

$$T_P^J \downarrow n \models \bigwedge_{\substack{i=1,\ldots,k \\ j=1,\ldots,m_i}} B_j^i\theta\gamma.$$

We can assume that γ is such that each $B_j^i\theta\gamma$ ground.

Thus for $i = 1, \ldots, k$ $B_i\theta\gamma \in T_P^J \downarrow (n+1)$, i.e. for $i = 1, \ldots, k$ $T_P^J \downarrow (n+1) \models (A_1 \wedge \ldots \wedge A_k)\theta\gamma$. This proves the claim for $n+1$ and concludes the proof. □

LEMMA 5.24. *Let J be a pre-interpretation which is a model of Eq. Let I be based on J. Suppose that I is good. Then*

$$B_P \cap T_P^J(I) = T_P(B_P \cap I).$$

PROOF. Suppose $A \in B_P \cap T_P^J(I)$. Then for some state σ over I $A \equiv B\sigma$ and $I \models_\sigma A_1 \wedge \ldots \wedge A_n$ where $B \leftarrow A_1, \ldots, A_n$ is a clause from P. Thus σ when restricted to the variables of B is a ground substitution, say η. We thus have $I \models_\sigma (A_1 \wedge \ldots \wedge A_n)\eta$. But I is good so for some substitution θ $I \models (A_1 \wedge \ldots \wedge A_n)\eta\theta$. Thus $B_P \cap I \models (A_1 \wedge \ldots \wedge A_n)\eta\theta$. Moreover $A \equiv B\eta\theta$, so $A \in T_P(B_P \cap I)$.

If now $A \in T_P(B_P \cap I)$ then a fortiori $A \in B_P \cap T_P^J(B_P \cap I)$, so by the monotonicity of T_P^J we have $A \in B_P \cap T_P^J(I)$. □

This lemma states that all ground atoms inferred from I by means od T_P^J can already be inferred by means of T_P, provided I is good.

This brings us to the following important consequence of Lemmata 5.23 and 5.24.

COROLLARY 5.25. *Let J be a pre-interpretation which is a model of Eq. Then for every $n \geqslant 0$*

$$B_P \cap T_P^J \downarrow n = T_P \downarrow n.$$

Consequently

$$B_P \cap T_P^J \downarrow \omega = T_P \downarrow \omega.$$

PROOF. We prove the first claim by induction on n. For $n = 0$ it is a consequence of the fact that $B_P \subseteq B_J$.

Suppose this claim holds for some $n \geqslant 0$. Then

$$\begin{aligned} B_p \cap T_p^J{\downarrow}(n+1) &= B_p \cap T_p^J(T_p^J{\downarrow}n) \\ \text{(by Lemmata 5.23 and 5.24)} &= T_p(B_p \cap T_p^J{\downarrow}n) \\ \text{(by induction hypothesis)} &= T_p(T_p{\downarrow}n) \\ &= T_p{\downarrow}(n+1). \end{aligned}$$

This implies the claim and concludes the proof. □

Finally, we can relate completion of a program and negation as failure.

THEOREM 5.26. (Soundness of the negation as failure rule) (CLARK [C]) *Let P be a program. If A is in the finite failure set of P then comp $(P) \models \neg A$.*

PROOF. Let I be an interpretation based on a pre-interpretation J and which is a model of $IFF(P)$. Then by Theorem 5.17 $T_p^J(I) = I$. Thus by Lemma 3.11

$$I \subseteq T_p^J{\downarrow}\omega.$$

Suppose now moreover that J is a model of Eq, i.e. that I is a model of $comp(P)$. Then by Corollary 5.25

$$B_P \cap I \subseteq T_p{\downarrow}\omega.$$

Suppose now that A is in the finite failure set of P. Then by Theorem 5.6 $A \notin T_p{\downarrow}\omega$, so by the above $A \notin B_p \cap I$, i.e. $I \models \neg A$. □

5.8. Completeness of the negation as failure rule

We now prove the converse of the above theorem. To this purpose we first show how to construct models of the free equality axioms.

Let $\mathcal{C}$ be a set of substitutions. We call $\mathcal{C}$ *downward directed* if

$$\theta,\eta \in \mathcal{C} \Rightarrow \text{there exists } \gamma \in \mathcal{C} \text{ such that } \gamma < \theta \text{ and } \gamma < \eta.$$

Here $\gamma < \theta$ means that θ is more general then γ.

Suppose now that $\mathcal{C}$ is a set of substitutions. Put for two terms s,t

$$s \sim_{\mathcal{C}} t \text{ iff for some } \theta \in \mathcal{C}\ s\theta \equiv t\theta.$$

LEMMA 5.27. *Suppose that $\mathcal{C}$ is a downward directed set of substitutions. Then $\sim_{\mathcal{C}}$ is an equivalence relation which is a congruence w.r.t. all function symbols. Moreover, the pre-interpretation induced by $\sim_{\mathcal{C}}$ is a model of Eq.*

PROOF. The relation $\sim_{\mathcal{C}}$ is always reflexive and symmetric. By downward directedness of $\mathcal{C}$ it is also transitive.

Let $[s]$ stand for the equivalence class of term s w.r.t. $\sim_{\mathcal{C}}$. Let f be an n-ary function symbol. If $[s_1] = [t_1], \ldots, [s_n] = [t_n]$ for some terms $s_1,t_1,\ldots,s_n,t_n$, then by downward directedness of $\mathcal{C}$ for some $\theta \in \mathcal{C}$

$$s_1\theta \equiv t_1\theta, \ldots, s_n\theta \equiv t_n\theta.$$

Hence $f(s_1,\ldots,s_n)\theta \equiv f(t_1,\ldots,t_n)\theta$, i.e. $[f(s_1,\ldots,s_n)] = [f(t_1,\ldots,t_n)]$.

Thus the equivalence relation induced by $\sim_{\mathcal{C}}$ is indeed a congruence. This means that $\sim_{\mathcal{C}}$ induces a pre-interpretation of L. That this interpretation is indeed a model of Eq is easy to see as non-unifiable terms have necessarily different equivalent classes w.r.t. $\sim_{\mathcal{C}}$. □

We are now in position to prove the desired theorem. It is formulated in a slightly more general form. This theorem is due to JAFFAR, LASSEZ AND LLOYD [JLL]. We follow here essentially presentation of LLOYD [L] based on a simpler proof due to WOLFRAM, MAHER AND LASSEZ [WML].

THEOREM 5.28. (Completeness of the negation as failure rule) *Let P be a program. If for a goal G $comp(P) \models G$, then $P \cup \{G\}$ has a finitely failed SLD-tree.*

PROOF. Let $G = \leftarrow A_1,...,A_s$. Suppose that $P \cup \{G\}$ does not have a finitely failed SLD-tree. Then by Theorem 5.6 there is a non-failed fair SLD-derivation $\xi = \leftarrow A = N_0, N_1,...$ with the sequence of substitutions $\theta_0, \theta_1,....$ We use this derivation to construct a model of $comp(P) \cup \{\exists x_1...\exists x_t(A_1 \wedge ... \wedge A_s)\}$ where $x_1,...,x_t$ are all variables appearing in G.

Let $\mathcal{C} = \{\theta_0...\theta_i : i \geqslant 0\}$. Note that $\mathcal{C}$ is downward directed. By the last lemma the pre-interpretation induced by $\sim_{\mathcal{C}}$ is a model of Eq. Let $[s]$ denote the equivalence class under $\sim_{\mathcal{C}}$ of a term s.

We now construct an interpretation I based on J by putting

$$I = \{p([t_1],...,[t_n]) : p(t_1,...,t_n) \text{ appears in } \xi\}$$

We first show that $I \subseteq T_p^J(I)$, i.e. that I is a model of $ONLY-IF(P)$.

Suppose that $p(t_1,...,t_n)$ appears in a goal N_i of ξ. Since ξ is non-failed and fair, there exists $j \geqslant i$ such that $p(s_1,...,s_n) \equiv p(t_1,...,t_n)\theta_i...\theta_{i+j-1}$ is the selected atom in N_j.

By Corollary 2.4 each θ_k can be chosen idempotent. Moreover, these substitutions commute, so each element of $\mathcal{C}$ is idempotent, as well. Thus for each $k = 1,...,n$

$$\begin{aligned}[t_k] &= [t_k\theta_0...\theta_{i+j-1}] \\ &= [t_k\theta_i...\theta_{i+j-1}] \\ &= [s_k] \\ &= [s_k\theta_{i+j}].\end{aligned}$$

But by the definition of I we have $p([s_1\theta_{i+j}],...,[s_n\theta_{i+j}]) \in T_p^J(I)$, so $p([t_1],...,[t_n]) \in T_p^J(I)$, as desired.

Now by Theorem 3.10 and Theorem 5.17 I can be extended to a model of $comp(P)$. By the construction I is a model of $\exists x_1...\exists x_t(A_1 \wedge ... \wedge A_s)$, and a fortiori so is its extension. □

5.9. *Equality axioms versus identity*

CLARK'S [C] original definition of free equality additionally included the following usual equality axioms:

(1) $x = x$,

(2) $x_1 = y_1 \wedge ... \wedge x_n = y_n \rightarrow f(x_1,...,x_n) = f(y_1,...,y_n)$
for each function symbol f,

(3) $x_1 = y_1 \wedge ... \wedge x_n = y_n \rightarrow (p(x_1,...,x_n) \rightarrow p(y_1,...,y_n))$
for each relation symbol p including $=$.

Denote these axioms by EQ. We did not use EQ at the expense of interpreting equality as identity. Fortunately, both approaches are equivalent as the following well known theorem (see e.g. MENDELSON [Me] p. 80) shows.

THEOREM 5.29. *Let S be a set of formulas in a first order language L including $=$. Then for every formula ϕ*

$$S \models \phi \text{ iff } S \cup EQ \models_+ \phi,$$

where $\models_+$ stands for validity w.r.t. interpretations of L which interpret $=$ in an arbitrary fashion.

PROOF. $\Rightarrow$. An interpretation of $=$ in a model of EQ is an equivalence relation which is a congruence w.r.t. all function and relation symbols.

This implies that every model of EQ is equivalent (i.e. satisfies the same formulas) to a model in which equality is interpreted as identity. This model has as the domain the equivalence classes of the interpretation of $=$ with the function and relation symbols interpreted in it in a natural way. The proof of the equivalence proceeds by straightforward induction on the structure of the formulas.
$\Leftarrow$. When $=$ is interpreted as indentity, all axioms of EQ became valid. $\square$

5.10. Summary

Summarizing the results obtained in Sections 5.4, 5.7 and 5.8 we obtain the following characterizations of the finite failure.

THEOREM 5.30. (Finite Failure Theorem) *Consider a program P and a ground atom A. Then the following are equivalent:*
(a) A is in the finite failure set of P.
(b) $A \notin T_P{\downarrow}\omega$.
(c) Every fair SLD-tree with $\leftarrow A$ as root is finitely failed.
(d) $comp(P) \models \neg A$. $\square$

These results show that the negation as failure rule is a proof theoretic concept with very natural mathematical properties. Comparing the above theorem with the Success Theorem 3.26 we see a natural duality between the notions of success and finite failure.

However, this duality is not complete. By the Characterization Theorem 3.13 and the success Theorem 3.26 A is in the success set of P iff $A \in lfp(T_P)$. On the other hand the "dual" statement: A is in the finite failure of P iff $A \notin gfp(T_P)$ does not hold because as noted in Section 4.5 for certain programs P we have $gfp(T_P) \neq T_P{\downarrow}\omega$.

Clause (*d*) of the Finite Failure Theorem suggests another possibility of inferring negation. Consider the following rule implicitly studied in APT and VAN EMDEN [AVE].

$$\frac{A \text{ is false in all Herbrand models of } comp\ (P)}{\neg A}$$

Call this rule *Herbrand rule*. Then the results of this chapter can be summarized by the following figure from LLOYD [L] (p. 86) assessing the content of Lemma 5.3, Theorem 5.18 iii) and Theorem 5.6.

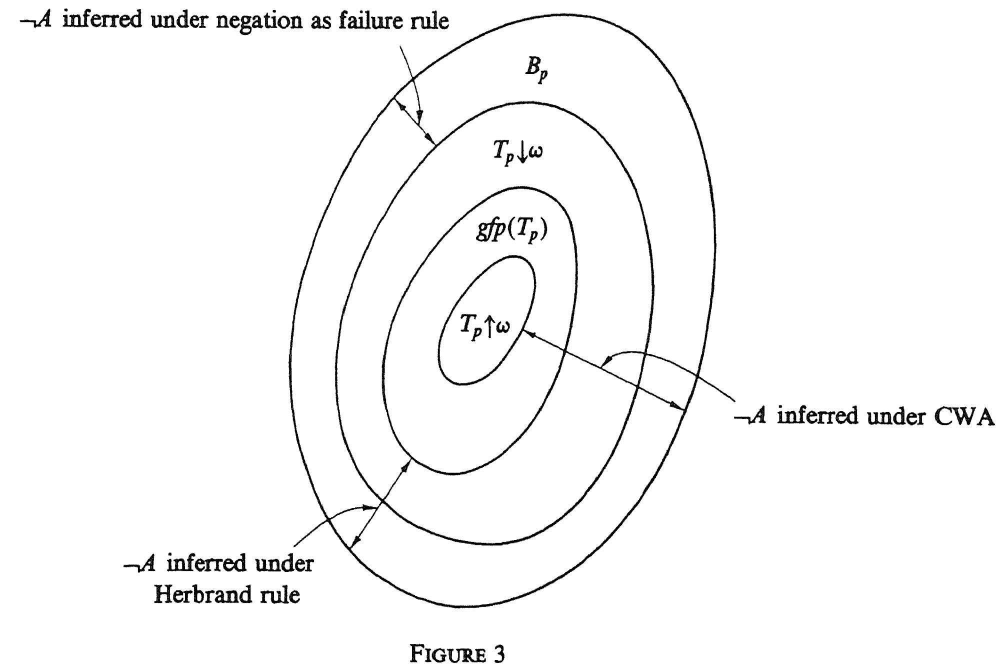
$\neg A$ inferred under negation as failure rule
B_P
$T_P{\downarrow}\omega$
$gfp(T_P)$
$T_P{\uparrow}\omega$
$\neg A$ inferred under CWA
$\neg A$ inferred under Herbrand rule

FIGURE 3

5.11. Bibliographic remarks

Theorem 5.17 is a straightforward generalization due to JAFFAR, LASSEZ and LLOYD [JLL] of a special case (Theorem 5.18 a)) proved in APT and VAN EMDEN [AVE].

Lemma 5.20 appears as an exercise in LLOYD [L] (p. 88). Proofs of Lemma 5.21 and Theorem 5.26 seem to be new. Lemma 5.21 was generalized by KUNEN [Ku1] who proved that that Eq is a complete axiomatization for the fragment $L(=)$ of L containing $=$ as the only relation symbol.

6. GENERAL GOALS

6.1. SLDNF$^-$-resolution

When trying to extend the results of chapters 3 and 5 to general programs we encounter several difficulties. In this paper we examine only a very mild extension of the previous framework, namely the use of logic programs together with *general* goals. This provides some insight into the nature of the new problems.

We have to explain first how general goals are to be refuted. For this purpose we need only to clarify how negative literals are to be resolved. It is natural to use for this purpose the negation as failure rule studied in the previous chapter. Strictly speaking this rule was defined only for ground atoms, but it can be extended in an obvious way to the non-ground case.

This leads us to an extension of the SLD-resolution called SLDNF$^-$-resolution (SLD-resolution with Negation as Failure rule) introduced in Clark [C]. We added the superscript "-" to indicate that it is used here only with non-general programs.

Formally, we first introduce the notion of a resolvent of a general goal. Let P be a program and $G = \leftarrow L_1,...,L_n$ a general goal. We distinguish two cases.

a) Literal L_i $(1 \leqslant i \leqslant n)$ is positive. Suppose that $C = A \leftarrow B_1,...,B_k$ is a clause from P. If L_1 and A unify with an mgu θ then

$$\leftarrow (L_1,...,L_{i-1},B_1,...,B_k,L_{i+1},...,L_n)\theta$$

is a *resolvent of G and C*.

b) Literal L_i $(1 \leqslant i \leqslant n)$ is negative, say $\neg A_i$. Suppose that $P \cup \{\leftarrow A\}$ has a finitely failed SLD-tree. Then

$$\leftarrow L_i,...,L_{i-1},L_{i+1},...,L_n$$

is a *resolvent* of G.

L_i is called the *selected literal of G*.

Now, given a program P and a general goal G, by an SLDNF$^-$*-derivation* of $P \cup \{G\}$ we mean a maximal sequence $G = G_0, G_1,...$ of general goals together with a sequence $C_0, C_1,...$ of variants of clauses from P and a sequence $\theta_0, \theta_1,...$ of substitution such that for all $i = 0,1,...$

a) If the selected literal in G_i is positive then G_{i+1} is a resolvent of G_i and C_i with the mgu θ_i,

b) if the selected literal in G_i is negative then G_{i+1} is a resolvent of G_i, C_i is arbitrary and θ_i is the empty substitution,

c) C_i does not have a variable in common with G_i.

Note that if the selected negative literal $\neg A$ in a general goal G is such that $P \cup \{\leftarrow A\}$ has no finitely failed SLD-tree, then G has no successor in the SLDNF$^-$-derivation. Also note that a successful resolving of a negative literal introduces no variable bindings.

The notions of SLD-refutation, computed answer substitution, selection rule and SLD-trees generalize in an obvious way to the case of SLDNF$^-$-resolution. In particular we can talk of successful and failed SLDNF$^-$-trees.

6.2. Soundness of the SLDNF$^-$-resolution

In any soundness or completeness theorem we need to compare the existence of SLDNF$^-$-refutations with some statements referring to semantics of the program under consideration. However, a direct use of the programs is not sufficient here because of the negative literals. For example $P \cup \{\leftarrow \neg A\}$ is always consistent. What we need here is an extension of P which implies some negative information. An obvious candidate is the completion of P, *comp*(P), which was actually introduced by CLARK [C] to serve as a meaning of general programs when studying SLDNF-resolution.

After these preparations we can formulate the appropriate soundness theorem, essentially due to CLARK [C].

THEOREM 6.1. (Soundness of SLDNF$^-$-resolution) *Let P be a program and $G = \leftarrow L_1,...,L_k$ a general goal. Suppose that there exists an SLDNF$^-$-refutation of $P \cup \{G\}$ with the sequence of substitutions $\theta_0,...,\theta_n$. Then $(L_1 \wedge ... \wedge L_k)\theta_0...\theta_n$ is a semantic consequence of comp(P).*

To prove it we need the following mild generalization of Theorem 5.26, essentially due to CLARK [C].

LEMMA 6.2. *Consider a program P and an atom A. Suppose there is a finitely failed SLD-tree with $\leftarrow A$ as root. Then comp$(P) \models \neg A$.*

PROOF. By Lemma 5.7 there exists $n_0 \geqslant 1$ such that for every ground substitution θ, $P \cup \{A\theta\}$ has a finitely failed SLD-tree of depth $\leqslant n_0$. By Lemma 5.9 for every ground substitution θ, $A\theta \notin T_p \downarrow n_0$.

Suppose now that for some interpretation I based on a pre-interpretation J, $I \models comp(P)$, and moreover for some state σ $I \models_\sigma A$. By Theorem 5.17 $T_p^J(I) = I$. Thus by Lemma 3.11 $I \subseteq T_p^J \downarrow n_0$. So we have $T_p^J \downarrow n_0 \models_\sigma A$. But by Lemma 5.23 $T_p^J \downarrow n_0$ is good, so for some ground substitution θ $T_p^J \downarrow n_0 \models A\theta$.

Now by Corollary 5.25 $A\theta \in T_p \downarrow n_0$. This contradicts the former conclusion. □

We can now prove soundness of SLDNF$^-$-resolution.

PROOF OF THEOREM 6.1. Let $A_1,...,A_l$ be the sequence of positive literals of G and $\neg B_1,...,\neg B_m$ the sequence of negative literals of G.

If $l = 0$ or $m = 0$ we disregard the corresponding step in the considerations below.

With each SLDNF$^-$-refutation of $P \cup \{G\}$ we can associate an SLD-refutation of $P \cup \{\leftarrow A_1,...,A_l\}$ obtained by deleting all resolvents arising from the selection of negative literals and by deleting all negative literals in the remaining resolvents. By the soundness of SLD-resolution (Theorem 3.2) and the fact that empty substitutions are used when resolving negative literals

$$P \models (A_1 \wedge ... \wedge A_l)\theta_0...\theta_n \ .$$

But *comp*$(P) \models IF(P)$ so by Lemma 5.13

$$comp(P) \models (A_1 \wedge ... \wedge A_l)\theta_0...\theta_n \ .$$

Also, by Lemma 6.2, for $i = 1,...,m$

$$comp(P) \models \neg B_i\theta_0...\theta_{p-1} \ ,$$

where $\neg B_i\theta_0...\theta_{p-1}$ is the selected literal of G_p $(0 \leqslant p \leqslant n)$.

Thus

$$comp(P) \models (\neg B_1 \wedge ... \wedge \neg B_m)\theta_0...\theta_n$$

which concludes the proof. □

COROLLARY 6.3. *If there exists an SLDNF$^-$-refutation of $P \cup \{G\}$ then comp$(P) \cup \{G\}$ is inconsistent.*
□

6.3. Floundering

We now consider the problem of completeness of the SLDNF$^-$-resolution. Unfortunately even the weakest form of completeness does not hold as the following example shows.

EXAMPLE 6.4. Consider the following program P:

$$p(a) \leftarrow p(a)\ ,$$
$$r(b) \leftarrow .$$

Then in every model I of the free equality axioms

$$I \models (\forall x(p(x) \leftrightarrow x=a \wedge p(a))) \rightarrow \neg p(b)\ ,$$

so by the definition of completion $comp(P) \models \neg p(b)$, that is $comp(P) \cup \{\leftarrow \neg p(x)\}$ is inconsistent. However, $P \cup \{\leftarrow p(x)\}$ has no finitely failed SLD-tree,, so there is no SLDNF$^-$-refutation of $P \cup \{\leftarrow \neg p(x)\}$. □

A natural way out of this dilemma is to impose on SLDNF$^-$-resolution some restrictions. Clearly, the problem is caused here by the use of non-ground negative literals. Notice for instance that in the above example $P \cup \{\leftarrow \neg p(b)\}$ has a finitely failed SLD-tree so there exists an SLDNF$^-$-refutation of $P \cup \{\leftarrow \neg p(b)\}$.

We thus introduce the following restriction. We say that a selection rule is *safe* if it only selects negative literals which are ground. >From now on we shall use *only* safe selection rules. But a safe selection rule is not defined on some sequences of literals. This means that certain general goals have no resolvents under a safe selection rule.

We say that an SLDNF$^-$-derivation of $P \cup \{G\}$ via a safe selection rule *flounders* if it is of the form $G=G_0,...,G_k$ where G_k contains only non-ground negative literals. $P \cup \{G\}$ *flounders* if some SLDNF$^-$-derivation of $P \cup \{G\}$ (via a safe selection rule) flounders.

Obviously, restriction to safe selection rules does not restore completeness of SLDNF$^-$-resolution - a smaller number of selection rules cannot help. But one would hope that a restriction to programs P and general goals G such that $P \cup \{G\}$ does not flounder but does help. Unfortunately such hopes are vain.

EXAMPLE 6.5. Consider the following program P:

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

and the general goal $G = \leftarrow r(x),\ \neg q(x)$.

We now claim that

i) $P \cup G$ does not flounder,

ii) there is no SLDNF$^-$-refutation of $P \cup G$,

iii) $comp(P) \cup \{G\}$ is inconsistent.

Both i) and ii) are easy to check. To prove iii) take an interpretation I based on a pre-interpretation J such that $I \models comp(P)$. By Theorem 5.17 $T_P^J(I)=I$. Thus by the form of P the following three facts hold:

a) $r(a) \in I$,

b) $q(a) \in I \rightarrow r(b) \in I$,

c) $q(b) \notin I$.

This means that either $I \models r(a) \wedge \neg q(a)$ or $I \models r(b) \wedge \neg q(b)$ holds, i.e. $I \models \exists x(r(x) \wedge \neg q(x))$ so G is not true in I. □

6.4. Restricted completeness of the SLDNF$^-$*-resolution*

Thus to obtain completeness of SLDNF$^-$-resolution further restrictions are necessary. To this purpose we first introduce the following notions.

Given a program P we define its *dependency graph* D_P by putting for two relations r,q

$$(r,q) \in D_P \text{ iff there is a clause in } P \text{ using } r \text{ in its head and } q \text{ in its body.}$$

We then say that r *refers to* q; *depends on* is the reflexive transitive closure of the relation refers to. Thus a relation does not need to refer to itself but by reflexivity every relation depends on itself.

Now, given a program P and a general goal G, we say that $P \cup \{G\}$ is *strict* if the relations occurring in positive literals of G depend on different relations than those on which relations occurring in negative literals of G depend. Note that this implies that no relation occurs both in a positive and negative literal of G.

More precisely, given a program P and a set of relations R first put

$$DEP(R) = \{q : \text{some } p \text{ in } R \text{ depends on } q\}.$$

Then $P \cup \{G\}$ is *strict* if

$$DEP(G^+) \cap DEP(G^-) = \emptyset,$$

where G^+ (resp. G^-) stands for the set of relations occurring in positive (resp. negative) literals of G.

Note that for the program P and the general goal G studied in Example 6.5 $P \cup \{G\}$ is not strict.

We now prove the following result established independently by CAVEDON and LLOYD [CL] and K.R. Apt (unpublished).

THEOREM 6.6. (Restricted completeness of SLDNF$^-$-resolution) *Let P be a program and G a general goal such that $P \cup \{G\}$ is strict and $P \cup \{G\}$ does not flounder. Suppose $comp(P) \cup \{G\}$ is inconsistent. Then there exists an* SLDNF$^-$*-refutation of $P \cup \{G\}$.*

In the proof we shall use the following well known theorem from mathematical logic due to K. Gödel (see e.g. SHOENFIELD [S]).

THEOREM 6.7. (Compactness Theorem) *A set of formulas has a model iff every finite subset of it has a model.* □

Using the Compactness Theorem we obtain the following lemma which will be needed in the sequel.

LEMMA 6.8. *Let P be a program. There exists a model N_P of $comp(P)$ such that*

$$N_P \cap B_P = T_P{\downarrow}\omega.$$

PROOF. Let $\{A_1,...,A_n\}$ be a finite subset of $T_P{\downarrow}\omega$. By Theorem 5.6, for $i = 1,...,n$, A_i is not in the failure set of P. Thus by Lemma 5.8 $P \cup \{\leftarrow A_1,...,A_n\}$ does not have a finitely failed *SLD*-tree. Now, by the completeness of the negation as failure rule (Theorem 5.28) there is a model of $comp(P) \cup \{A_1,...,A_n\}$. Thus by the Compactness Theorem 6.7 $comp(P) \cup T_P{\downarrow}\omega$ has a model, say N_P. We have

$$N_P \cap B_P \supseteq T_P{\downarrow}\omega.$$

Moreover, as already observed in the proof of soundness of the negation as failure rule (Theorem 5.26), we have by virtue of Lemma 3.11 and Corollary 5.25

$$N_P \cap B_P \subseteq T_P{\downarrow}\omega.$$

This concludes the proof. □

The model of *comp*(P) constructed in this lemma is in a sense "big". Note that by Theorem 5.6 we have

$$N_P \models \neg A \textit{ iff } A \text{ is in the failure set of } P.$$

Thus in a sense N_P is "dual" to M_P which is a "small" model of *comp*(P) and for which by the Characterization Theorem 3.13 and the Success Theorem 3.25

$$M_P \models A \text{ iff } A \text{ is in the success set of } P.$$

In the proof of Theorem 6.6 we shall use both types of models. But first we need the following simple modification of Lemma 3.9.

LEMMA 6.9. *Let T be a continuous operator on a complete lattice. Suppose that $I \subseteq T(I)$. Then $T\uparrow\omega(I)$ is a fixpoint of T.*

PROOF. We have

$$\begin{array}{rcl} T\uparrow\omega(I) & = & I \cup \bigcup_{n=1}^{\infty} T\uparrow n(I) \\ \text{(by assumption)} & = & \bigcup_{n=1}^{\infty} T\uparrow n(I) \\ & = & \bigcup_{n=0}^{\infty} T(T\uparrow n(I)) \\ \text{(by continuity)} & = & T(T\uparrow\omega(I)). \ \square \end{array}$$

Before we apply this lemma we introduce the following notation. Given two programs P_1 and P_2, we write $P_1 < P_2$ to denote the fact that relations appearing in the heads of clauses from P_2 do not appear in P_1.

Informally, when $P_1 < P_2$ then P_1 does not depend on P_2. More formally, we have the following lemma.

LEMMA 6.10. *Let P_1 and P_2 be two programs such that $P_1 < P_2$. Then for any interpretation I based on a pre-interpretation J and $n \geqslant 1$*

$$T^J_{P_1}(T^J_{P_2}\uparrow n(I)) = T^J_{P_1}(\varnothing).$$

PROOF. All elements of $T^J_{P_2}\uparrow n(I)$ are of the form $r(t_1,\dots,t_m)\sigma$ where r appears in a head of a clause from P_2. $\square$

LEMMA 6.11. *Let P_1 and P_2 be two programs such that $P_1 < P_2$. Suppose that I is a model of comp(P_1) based on a pre-interpretation J. Then $T^J_{P_2}\uparrow\omega(I)$ is a model of comp$(P_1 \cup P_2)$.*

PROOF. By Theorem 5.17 we have $I = T^J_{P_1}(I) \subseteq T^J_{P_1 \cup P_2}(I)$. Moreover, $T^J_{P_1 \cup P_2}$ is continuous. By Lemma 6.9 $T^J_{P_1 \cup P_2}\uparrow\omega(I)$ is a fixpoint of $T^J_{P_1 \cup P_2}$ so by Proposition 5.12 $T^J_{P_1 \cup P_2}\uparrow\omega(I)$ is a model of *comp*$(P_1 \cup P_2)$.

On the other hand, using Lemma 6.10 and the fact that $T^J_{P_1}(\varnothing) \subseteq I$, we get by an induction on n

$$T^J_{P_1 \cup P_2}\uparrow n(I) = T^J_{P_2}\uparrow n(I)$$

for every $n \geqslant 0$.

Hence

$$T^J_{P_1 \cup P_2}\uparrow\omega(I) = T^J_{P_2}\uparrow\omega(I)$$

which concludes the proof. $\square$

We can now prove the desired result.

PROOF OF THEOREM 6.6. Let P^+ (resp. P^-) be the set of clauses of P whose heads contain a relation belonging to $DEP(G^+)$ (resp. $DEP(G^-)$). By the assumption of strictness, P^+ and P^- are disjoint. For some set P_0 of clauses

$$P = P_0 \dot{\cup} P^+ \dot{\cup} P^-.$$

Note that $P^+ \cup P^- < P_0$. Consider now the interpretation $M_{P^+} \cup N_{P^-}$. Note that M_{P^+} and N_{P^-} are disjoint because no relation occurs both in P^+ and P^-. Thus $M_{P^+} \cup N_{P^-}$ is a model of $comp(P^+) \cup comp(P^-)$, i.e. a model of $comp(P^+ \cup P^-)$. This model is based on some pre-interpretation J. By Lemma 6.11 $M = T^J_{P_0}\uparrow\omega(M_{P^+} \cup N_{P^-})$ is a model of $comp(P)$.

By the assumption $comp(P) \cup \{G\}$ is inconsistent, so for some state σ

$$J \models_\sigma A_1 \wedge ... \wedge A_\ell \wedge \neg B_1 \wedge ... \wedge \neg B_m$$

where $A_1,...,A_\ell$ is the sequence of positive literals of G and $\neg B_1,...,\neg B_m$ is the sequence of negative literals of G.

If $\ell = 0$ or $m = 0$ we disregard the corresponding step in the considerations below.

By the definition of P^+ and P^- and the form of M we have $M_{P^+} \models_\sigma A_1 \wedge ... \wedge A_\ell$ and $N_{P^-} \models_\sigma \neg B_1 \wedge ... \wedge \neg B_m$. Thus σ, when restricted to the variables of $A_1 \wedge ... \wedge A_\ell$, is a ground substitution, say θ.

By Corollary 3.6 and the Characterization Theorem 3.13i) θ is a correct answer substitution for $P^+ \cup \{\leftarrow A_1,...,A_\ell\}$.

Applying now Theorem 3.18 we obtain a computed answer substitution γ for $P^+ \cup \{\leftarrow A_1,...,A_\ell\}$ which is more general than θ.

Fix some i, $1 \leqslant i \leqslant m$. By the assumption $P \cup \{G\}$ does not flounder. Thus if $\ell = 0$ then B_i is ground, so $B_i\sigma$ is a ground atom. If $\ell > 0$ then $B_i\gamma$ is ground, so $B_i\theta$ is ground and consequently $B_i\sigma$ is a ground atom, as well. But $N_{P^-} \models_\sigma \neg B_1 \wedge ... \wedge \neg B_m$, so $B_i\sigma \in B_{P^-} \setminus N_{P^-}$.

By Lemma 6.8 we now have $B_i\sigma \notin T_{P^-}\uparrow\omega$. By Theorem 5.6 $B_i\sigma$ is in the finite failure set of P^-. By the form of P^-, $B_i\sigma$ is in the finite failure set of P.

We thus showed that there exists an SLDNF$^-$-refutation of $P \cup \{G\}$. □

This theorem can be generalized in the same ways as the completeness theorem of *SLD*-resolution (Theorem 3.15) was. The proofs of these generalizations are straightforward modifications of the above proof and use the generalizations of Theorem 3.15 presented in Sections 3.8 and 3.9.

6.5. Allowedness

Unfortunately restriction to programs P and general goals G such that $P \cup \{G\}$ does not flounder is not satisfactory as the following theorem shows.

THEOREM 6.12. (Undecidability of non-floundering) *For some program P it is undecidable whether for a general goal G $P \cup \{G\}$ does not flounder.*

PROOF. This is a simple consequence of the computability results established in Section 4.4.

Let P be a program and $q(x)$ an atom such that the variable x does not appear in P. Note that for any ground atom A there exists an *SLD*-refutation of $P \cup \{A\}$ iff $P \cup \{\leftarrow A, \neg q(x)\}$ flounders. Indeed, in the SLDNF$^-$-derivations no new negative literals are introduced.

By Corollary 3.14 and Lemma 3.17 we thus have

$$A \in M_P \textit{ iff } P \cup \{\leftarrow A, \neg q(x)\} \text{ flounders.}$$

But by Corollary 4.7 for some program P M_P is not recursively enumerable. Consequently it is not decidable whether for such a program P, $P \cup \{\leftarrow A, \neg q(x)\}$ does not flounder. □

A way to solve this problem is by imposing on $P \cup \{G\}$ some syntactic restrictions which imply that $P \cup \{G\}$ does not flounder.

To this purpose we introduce the following notion due to LLOYD and TOPOR [LT]. Given a program P and a general goal G, we call $P \cup \{G\}$ *allowed* if the following two conditions are satisfied:

a) every variable of G appears in a positive literal of G,
b) every variable of a clause in P appears in a body of this clause.

Note that a) implies that all negative literals of G are ground if G has no positive literals and b) implies that every unit clause in P is ground.

Allowedness is the notion we are looking for as the following theorem shows.

THEOREM 6.13. (LLOYD and TOPOR [LT]) *Consider a program P and a general goal G such that* $P \cup \{G\}$ *is allowed. Then*

i) $P \cup \{G\}$ does not flounder,
ii) every computed answer substitution for $P \cup \{G\}$ is ground.

PROOF. i) Condition b) ensures that every general goal appearing in an SLDNF$^-$-derivation satisfies condition a). Thus $P \cup \{G\}$ does not flounder.
ii) By the fact that every unit clause in P is ground. □

Property ii) shows the price we have to pay for ensuring property i).

Combining Theorems 6.6 and 6.13 we obtain the following conclusion.

COROLLARY 6.14. *Let P be a program and G a general goal such that* $P \cup \{G\}$ *is strict and allowed. Suppose* $comp(P) \cup \{G\}$ *is inconsistent. Then there exists an* SLDNF*-refutation of* $P \cup \{G\}$. □

Finally, observe that the definition of allowedness can be weaken a bit by requiring condition b) to hold only for clauses whose heads contain a relation appearing in a positive literal of G. Indeed, Theorem 6.13 then still holds by virtue of the same argument.

6.6. Bibliographic remarks

Usually, the case of programs and *general* goals is not considered separately. Consequently soundness of the SLDNF$^-$-resolution (Theorem 6.1) is not spelled out separately. The proof of Lemma 6.2 seems to be new. The problem noted in Example 6.4 was first identified in CLARK [C]. Example 6.5 seems to be new. The name *floundering* was introduced in SHEPHERDSON [Sh] but the concept first appeared in CLARK [C]. Theorem 6.12 belongs to the folklore.

The notion of strictness was first introduced in APT, BLAIR and WALKER [ABW] for the case of general programs. The definition adopted here is inspired by CAVEDON and LLOYD [CL] where a much stronger version of Theorem 6.6 dealing with general programs is proved. The definition of allowedness is a special case of the one introduced in LLOYD and TOPOR [LT] for general programs. Similar, but less general notions were considered in CLARK [C], SHEPHERDSON [Sh] and APT, BLAIR and WALKER [ABW].

7. Related topics

Our presentation of logic programming is obviously incomplete. In this section we briefly discuss the subjects we omitted and provide a number of pointers to the literature.

7.1 General programs

SLD-resolution and the negation as failure rule was combined by Clark[C] into a more powerful computation mechanism called *SLDNF-resolution* allowing us to refute general goals from general programs. The reader is referred to Lloyd[L1] for a detailed account of *SLDNF* - resolution.

General programs are difficult to study because of their irregular behaviour. For example, they do not need to have the least Herbrand model and the completion of a general program can be inconsistent.

Recent work in this area concentrated on a subclass of general programs called *stratified programs* which exhibit a more regular behaviour. Stratified programs are general programs in which recursion "through" negation is disallowed. They were introduced in Apt, Blair and Walker[ABW] and Van Gelder[VG]. They form a simple generalization of a class of database queries introduced in Chandra and Harel[CH]. Further work on this subject was carried out by Lifschitz[Li], Przymusinski[P] and others.

Shepherdson[She2] discusses and compares various approaches to the proof theory and semantics of general programs.

7.2. Alternative approaches

The approach to logic programming we discussed in this paper is undoubtedly the most widely accepted. However, various alternatives exist and it is worthwhile to point them out.

Proof theory. Fitting [F] proposed an alternative computation mechanism based on a tableau method. Gallier and Raatz [GR] introduced a computation mechanism in the form of an interpreter using graph reduction. Brough and Walker [BW] studied interpreters with various stopping criteria for function-free programs. Apt, Blair and Walker [ABW] introduced an interpreter with a loop checking mechanism and with an ineffective means of handling negative literals. Przymusinski [P] generalized this interpreter to an *SLS-resolution* (Linear resolution with Selection rule for Stratified programs) in which negative literals are resolved in an ineffective way.

Variants of *SLD* - resolution, called *HLSD* - resolution and *SLD-AL* - resolution were introduced and studied in Naish [N] and Vieille [V], respectively.

Semantics. Mycroft [My] suggested to use three valued logic (corresponding to the possibilities: provable, refuted and undecidable) to capture the meaning of logic programs. This approach was subsequently studied in detail in Fitting [F1] and Kunen [Ku].

To describe the meaning of general programs Minker [Mi] proposed the use of minimal models (leading to the *generalized closed world assumption* GCWA), Bidoit and Hull [BH] proposed the use of *positivistic models* and Przymusinski [P] introduced the concept of a *perfect model.*

7.3. Deductive databases

Deductive databases form an extension of relational databases in which some of the relations are implicitly defined. They can be viewed as logic programs where the *explicitly* defined relations are those defined only by means of unit clauses, whereas the *implicitly* defined relations are those defined by means of non-unit clauses, as well. Moreover, so-called *particularization axioms* are needed to define the intended domain. Additionally, integrity constraints are used to impose a desired meaning on the relations used.

The main difference between deductive databases and logic programming lies in their emphasis on different problems. In deductive databases one studies such issues like query processing (i.e. computation of *all* answers to a given goal), integrity constraint checking, handling of updates (i.e. additions and deletions of ground unit clauses) and processing of negative information.

Recent research concentrates on efficient implementation of recursive queries, i.e. queries about recursively defined relations (see e.g. the survey of BANCILHON and RAMAKRISHNA [BR]), reduction of recursive queries to non-recursive ones (see e.g. NAUGHTON and SAGIV [NS]), comparison of expressive power between various query languages (see e.g. CHANDRA and HAREL [CH], SHMUELI [Shm]), and handling of negative information both in terms of intended semantics (see e.g. MINKER [Mi], APT, BLAIR and WALKER [ABW], VAN GELDER [VG], LIFSCHITZ [Li], NAQVI [Na], PRZYMUSINSKI [P]) and in terms of query processing, handling of updates and integrity constraint checking (see e.g. HENSCHEN and PARK [HP], DECKER [D], LLOYD, SONENBERG and TOPOR [LST]).

Earlier research in this area is surveyed in GALLAIRE, MINKER and NICOLAS [GMN] while more recent research is discussed in MINKER [Mi1].

7.4. PROLOG

PROLOG stands for *pro*gramming in *log*ic. It is a programming language conceived and implemented in the beginning of 1970's by COLMERAUER et al. [CKRP]. In its pure form it can be viewed as logic programming with the "left-first" selection rule and with the depth-first strategy for searching the empty node in an SLD-tree. Negation is implemented by means of the negation as failure rule. For efficiency reasons an important test (the check in step 5 of the unification algorithm whether x appears in t - so-called *occur check*) is usually deleted from the unification algorithm and a special control facility (called *cut*) to prune the search tree is introduced. These changes make PROLOG different from logic programming and make it difficult to apply to its study the theoretical results concerning logic programming.

Theoretical study of PROLOG concentrated on efforts to provide a rigorous semantics of it in terms of interpreters explaining the process of SLD-tree traversal (see e.g. JONES and MYCROFT [JM]), by means of denotational semantics (see e.g. FITTING [F1]) or by relating both approaches (see e.g. DEBRAY and MISHRA [DM]).

More practical considerations, apart of a study of implementations of PROLOG (see e.g. CAMPBELL [Ca]), led to an investigation of efficient backtracking mechanisms (see e.g. COX and PIETRZYKOWSKI [CP]) and of various additions, like meta-facilities (see e.g. BOWEN and KOWALSKI [BK], STERLING and SHAPIRO [SSh]), modules (see e.g. GOGUEN and MESEGUER [GM]), control mechanisms (see e.g. NAISH [N]) and parallelism (see e.g. Concurrent Prolog of SHAPIRO [Sha] and PARLOG of CLARK and GREGORY [CG]).

Good books on PROLOG programming are BRATKO [B] and STERLING and SHAPIRO [SSh].

7.5. Integration of logic and functional programming

Logic or PROLOG programs use relational notation. This makes it awkward to define functions explicitly which have to be rewritten and used as relations.

Functional programming is based on the use of functions as primitive objects and shares with logic programming several aspects like the use of recursion as the main control structure and reliance on mathematical logic (especially lambda calculus).

Several attempts to combine advantages of both formalisms in one framework originated with the LOGLISP language of ROBINSON and SIEBERT [RS].

Direct definition of functions by means of equations leads to the problem how in the framework of logic programming equality is to be handled. Solutions to this problem involve the use of *extended unification*, where identity is replaced by equality derivable from axioms defining functions, the use of term rewriting techniques in the form of a *narrowing procedure* and the use of some subset of the standard equality axioms EQ defined in Section 5.9 written in a clausal form.

Recent proposals in this area are collected in DE GROOT and LINDSTROM [dGL] which is a standard reference in this domain. See also BELLIA and LEVI [BL], GALLIER and RAATZ [GR], and VAN EMDEN and YUKAWA [VEY].

7.6. Applications in artificial intelligence

Strictly speaking, logic programming is just a restricted form of automatic theorem proving. Various proposals of extending it to more powerful fragments of certain logics can be seen as attempts to increase its expressive and manipulative power while preserving efficiency. In particular a substantial effort has been made to adapt it to the needs of artificial intelligence. While research in this area is of a much more practical character, we can still single out out certain investigations of more theoretical nature.

Use of logic programming as a formalism for knowledge representation and reasoning was advocated in KOWALSKI [K1]. Analysis and implementation of more powerful logics and various forms of reasoning in the framework of logic programming was undertaken by FARIÑAS DEL CERRO [Fa] for modal logic, by SHAPIRO [Sha] and VAN EMDEN [vE] for quantitative reasoning and by POOLE [Po] for hypothetical reasoning.

More practical work in this area deals with natural language processing, the original application domain of PROLOG (see e.g. the special issue of Journal of Logic Programming [JLP]) and with the use of logic programming and PROLOG for the construction of expert system shells (see e.g. BRATKO [B] and WALKER [W].)

APPENDIX

Short history of the subject

The following is a list of papers and events which have shaped our views of this subject. Obviously, this account of the history of the subject is by no means objective (as none is).

1973: COLMERAUER et al. [CKRP] implemented PROLOG.

1974: KOWALSKI [K] proposed logic (programming) as a programming language and introduced what is now called *SLD*-resolution.

1976: VAN EMDEN and KOWALSKI [VEK] studied the semantics of logic programs and introduced the ubiquitous immediate consequence operator T_P.

1978: REITER [R] proposed in the context of deductive databases the *Closed World Assumption* rule as a means of deducing negative information.

1978: CLARK [C] introduced the *Negation as Failure* rule as an effective means of deducing negative information for logic programs and proposed the *completion of a program*, *comp*(P), as a description of its meaning.

1979: Kowalski [K1] analyzed logic programming as a formalism for knowledge representation and problem solving.

1979: Kowalski [K2] investigated logic programming as a formalism for a systematic development of algorithms.

1982: Logic programming was chosen as the basis for a new programming language in the Japanese Fifth Generation computer system project.

1982: Apt and Van Emden [AVE] characterized the *SLD*-resolution, Negation as Failure rule and completion of a program by means of the operator T_P and its fixpoints.

1983: In the book [CT] edited by K.L. Clark and S.-A. Tärnlund, a number of articles were collected that indicated a wide scope of applications of logic programming and revealed its manipulative and expressive power.

1983: Shapiro [Sh] implemented CONCURRENT PROLOG, a natural extension of PROLOG, allowing us to write concurrent programs.

1983: Jaffar, Lassez and Lloyd [JLL] proved completeness of the Negation as Failure rule with respect to the completion of a program.

1984: Lloyd [L] gathered in his book several results on logic programming in a single, uniform framework.

1984: A.J. Robinson founded the *Journal of Logic Programming*.

1986 In the book [dGL] edited by D. De Groot and G. Lindstrom, several approaches aiming at an integration of logic and functional programming were presented.

1986: Apt, Blair and Walker [ABW] and Van Gelder [VG] identified *stratified programs* as a natural subclass of general logic programs and proposed *stratification* as a means of handling negative information.

1986: J. Minker organized the Workshop on Foundations of Deductive Databases and Logic Programming which brought together researchers working in both areas.

Note
When this paper neared its completion we learned of the second edition of Lloyd [L], Lloyd [L1]. In Lloyd [L1] a program is called a definite program and in turn a general program is called a normal program. Similar terminology is used there for goals and general goals.

Acknowledgements
We would like to thank Marc Bezem and Stephane Grumbach for detailed comments on this paper. Our task was significantly simplified thanks to John Lloyd who collected in Lloyd [L] most of the results presented here in a single framework. Figure 3 was reproduced with his permission. Also, we profited from discussions on selected fragments of this paper with Howard Blair, Lawrence Cavedon, Maarten van Emden, Joxan Jaffar, Jan Willem Klop, Jean-Louis Lassez, John Lloyd and Rodney Topor. We would like to thank Eline Meys and Ria Riechelmann-Huis for typing the continuously growing and changing manuscript.

References

[AN]H. Andreka and I. Nemeti, *The Generalized Completeness of Horn Predicate Logic as a Programming Language*, Acta Cybernetica 4, 1978, 3-10.
[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, 1987.

[AVE]K.R.Apt and M.H. van Emden, *Contributions to the Theory of Logic Programming*, J. ACM, vol. 29, No. 3, 1982, 841-862.

[BR]F. Bancilhon and R. Ramakrishnan, *An Amateur's Introduction to Recursive Query Processing Strategies*, in: Proc. ACM Int. Conf. on Management of Data, Washington, D.C., 1986, 16-52.

[BL] M. Bellia and G. Levi, *The Relation between Logic and Functional Languages, a Survey*, Journal of Logic Programming, vol. 3, 1986, 217-236.

[BH]N. Bidoit and R. Hull, *Positivism Versus Minimalism in Deductive Databases*, in: Proc. of the 5th ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, Cambridge, Mass., 1986, 123-132.

[Bi] G. Birkhoff, *Lattice Theory,* American Mathematical Society Colloquium Publications, vol. 25, 1973.

[B1] H.A. Blair, *The Recursion-Theoretic Complexity of Predicate Logic as a Programming Language*, Information and Control, vol. 54, No. 1-2, 1982, 25-47.

[B2] H.A. Blair, *Decidability in the Herbrand Base*, manuscript (presented at the Workshop on Foundations of Deductive Databases and Logic Programming, Washington, D.C., August 1986).

[BK]K.A. Bowen and R.A. Kowalski, *Amalgamating Language and Metalanguage in Logic Programming*, in: Logic Programming (Clark, K.L. and S.-A. Tärnlund, eds.), Academic Press, New York, 1982.

[Bo] E. Börger, *Logic as Machine: Complexity Relations between Programs and Formulae*, to appear in: Current Trends in Theoretical Computer Science (E. Börger, ed.), Computer Science Press, 1987.

[B] I. Bratko, *PROLOG Programming for Artificial Intelligence*, Addison Wesley, 1986.

[BW]D. Brough and A. Walker, *Some Practical Properties of Logic Programming Interpreters,* in: Proc. of the Japan FGCS84 Conference, 1984, 149-156.

[Ca] J.A. Campbell, (ed.), *Implementations of Prolog*, Ellis Horwood, 1984.

[CL] L. Cavedon, and J. Lloyd, *A Completeness Theorem for SLDNF-Resolution,* Technical Report 87/9, Department of Computer Science, University of Melbourne, 1987.

[CH]A.K. Chandra and D. Harel, *Horn Clause Queries and Generalizations, Journal of Logic Programming, vol. 2, No. 1, 1985, 1-15.*

[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.

[C1] K.L. Clark, *Predicate Logic as a Computational Formalism*, Research Report DOC 79/59, Department of Computing, Imperial College, 1979.

[CG]K.L. Clark and S. Gregory, *PARLOG: A Parallel Logic Programming Language*, ACM Trans. on Prog. Lang. and Systems, vol. 8, No. 1, 1986, 1-49.

[CT] K.L. Clark and S.-A. Tärnlund (eds.), *Logic Programming*, Academic Press, New York, 1982.

[CKRP]A. Colmerauer, H. Kanoui, P. Roussel and R. Pasero, *Un Système de Communication Homme-Machine en Francais*, Groupe de Recherche en Intelligence Artificielle, Université d'Aix-Marseille, 1973.

[CP] P.T. Cox and T. Pietrzykowski, *Deduction Plans: a Basis for Intelligent Backtracking*, in: IEEE PAMI, vol. 3, 1981, 52-65.

[DM]S.K. Debray and P. Mishra, *Denotational and Operational Semantics for PROLOG,* in: Formal Description of Programming Concepts - III, (M. Wirsing, ed.), North Holland, 1987, 245-269.

[D] H. Decker, *Integrity Enforcement in Deductive Databases*, in: Proc. 1st Int. Conf. on Expert Database Systems, Charleston, S.C., 1986.

[VEK]M.H. van Emden and R.A. Kowalski, *The Semantics of Predicate Logic as a Programming Language*, J. ACM, vol. 23, No. 4, 1976, 733-742.

[VEY]M. van Emden and Yukawa, *Logic Programming with Equations*, Technical Report 86-05, Department of Computer Science, University of Waterloo, 1986. Journal of Logic Programming, to appear.

[Fa] L. Fariñas del Cerro, *MOLOG: A System that extends PROLOG with Modal Logic*, New Generation Computing, vol. 4, No. 1, 1986, 35-50.

[F] M. Fitting, *Logic Programming Based on Logic,* Manuscript, Dept. of Math. and Computer Science, Lehman College, Bronx, NY, 1985.

[F1] M. Fitting, *A Deterministic PROLOG Fixpoint Semantics*, Journal of Logic Programming, vol. 2, No. 2, 1985, 111-118.

[F2] M. Fitting, *A Kripke-Kleene Semantics for Logic Programs*, Journal of Logic Programming, vol. 2, No. 4, 1985, 295-312.

[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.

[G] J. Gallier, *Logic for Computer Science*, Harper & Row, Inc., 1986.

[GR]J. Gallier and S. Raatz, *A Graph-based Interpreter for General Horn Clauses*, Journal of Logic Programming, vol. 4, No. 2, 1987, 119-156.

[GM]J.A. Goguen and J. Meseguer, *Equality, Types, Modules and (Why Not?) Generics for Logic Programming*, Journal of Logic Programming, vol. 1, No. 2, 1984, 179-210.

[dGL]D. de Groot and G. Lindstrom (eds), *Logic Programming, Functions, Relations and Equations*, Prentice-Hall, Englewood Cliffs, N.J., 1986.

[HP]L. Henschen and H.S. Park, *Compiling the GCWA in Indefinite Deductive Databases,* in: Foundations of Deductive Databases and Logic Programming, (Minker, J., ed.), Morgan Kaufmann, Los Altos, 1987.

[He] J. Herbrand, *Logical Writings*, (W.D. Goldfarb, ed.), D. Reiter Publishing Company, Dordrecht, 1971.

[H] R. Hill, *LUSH-Resolution and its Completeness,* DCL Memo 78, Department of Artificial Intelligence, University of Edinburgh, 1974.

[IM]A. Itai and J.A. Makowsky, *Unification as a Complexity Measure for Logic Programming,* Journal of Logic Programming, vol. 4, No. 2, 1987, 105-118.

[JLL]J. Jaffar, J.-L. Lassez and J.W. Lloyd, *Completeness of the Negation as Failure Rule,* in: Proc. IJCAI-83, Karlsruhe, 1983, 500-506.

[K] R.A. Kowalski, *Predicate Logic as a Programming Language,* in: Proc. IFIP '74, Stockholm, North Holland, 1974, 569-574.

[K1] R.A. Kowalski, *Logic for Problem Solving*, North Holland, New York, 1979.

[K2] R.A. Kowalski, *Algorithm = Logic + Control,* C. ACM, vol. 22, No. 7, 1979, 424-435.

[K3] R.A. Kowalski, *The Relation Between Logic Programming and Logic Specification,* in: Mathematical Logic and Programming Languages, (Hoare, C.A.R. and J.C. Shepherdson, eds), Prentice-Hall, Englewood Cliffs, N.J., 1985, 11-27.

[KK]R.A. Kowalski and D. Kuehner, *Linear Resolution with Selection Function,* Artificial Intelligence 2, 1971, 227-260.

[Ku] K. Kunen, *Negation in Logic Programming,* Journal of Logic Programming, to appear.

[Ku1]K. Kunen, *Answer Sets and Negation as Failure,* in: Proc. of the 4th International Conference on Logic Programming, The MIT Press, Cambridge, Mass., 1987, 219-228.

[LM]J.-L. Lassez and M.J. Maher, *Closures and Fairness in the Semantics of Programming Logic,* Theoretical Computer Science, vol. 29, 1984, 167-184.

[LMM]J.L. Lassez, M.J. Maher and K. Marriott, *Unification Revisited,* in: Foundations of Deductive Databases and Logic Programming, (Minker, J., ed), Morgan Kaufmann, Los Altos, 1987.

[Li] V. Lifschitz [1986], *On the Declarative Semantics of Logic Programs with Negation,* in: Foundations of Deductive Databases and Logic Programming (Minker, J. ed.), Morgan Kaufmann, Los Altos, 1987.

[L] J.W. Lloyd, *Foundations of Logic Programming*, Springer-Verlag, Berlin, 1984.

[L1] J.W. Lloyd, *Foundations of Logic Programming*, Second Edition, Springer-Verlag, Berlin, 1987.

[LST]J.W. Lloyd, E.A. Sonenberg and R.W. Topor, *Integrity Constraint Checking in Stratified*

Databases, Technical Report 86/5, Department of Computer Science, University of Melbourne, 1986. Journal Logic Programming, to appear.

[LT] J.W. Lloyd and R. Topor, *A Basis for Deductive Databases II,* Journal of Logic Programming, vol. 3, No. 1, 1986, 55-67.

[M] Y.I. Manin, *A Course in Mathematical Logic,* Springer-Verlag, New York, 1977.

[MM] A. Martelli and U. Montanari, *An Efficient Unification Algorithm,* ACM Trans. on Prog. Lang. and Systems, vol. 4, No. 2, 1982, 258-282.

[Me] E. Mendelson, *Introduction to Mathematical Logic,* 2nd Edition, Van Nostrand, Princeton, N.J. 1979.

[Mi] J. Minker [1982], *On Indefinite Databases and the Closed World Assumption,* in: Proc. of the 6th Conference on Automated Deduction, (D.W. Loveland, ed.) Lecture Notes in Computer Science 138, Springer-Verlag, Berlin, 1982, 292-307.

[Mi1] J. Minker, *Perspectives in Deductive Databases,* Technical Report 87-7, Department of Computer Science, University of Maryland, 1987. Journal of Logic Programming, to appear.

[My] A. Mycroft, *Logic Programs and Many-valued Logic,* in: Proc. of Symposium on Theoretical Aspects of Computer Science, Lecture Notes in Computer Science 166, Springer-Verlag, Berlin, 1984, 274-286.

[N] L. Naish, *Negation and Control in PROLOG,* Lecture Notes in Computer Science 238, Springer-Verlag, Berlin, 1986.

[Na] S.A. Naqvi [1986], *A Logic for Negation in Database Systems,* in: Foundations of Deductive Databases and Logic Programming, (Minker, J., ed.), Morgan Kaufmann, Los Altos, 1987.

[NS] J.F. Naughton and Y Sagiv, *A Decidable Class of Bounded Recursions,* in: Proc. of the 6th ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, San Diego, 1987, 227-237.

[PW] M.S. Paterson and M.N. Wegman, *Linear Unification,* J. Computer and System Sciences, vol. 16, No. 2, 1978, 158-167.

[Po] D.L. Poole, *Default Reasoning and Diagnosis as Theory Formation,* Technical Report 86-08, Department of Computer Science, University of Waterloo, 1986.

[R] R. Reiter, *On Closed World Data Bases,* in: Logic and Data Bases, (Gallaire, H. and J. Minker, eds), Plenum Press, New York, 1978, 55-76.

[Ro] J.A. Robinson, *A Machine-oriented Logic Based on the Resolution Principle,* J. ACM, vol. 12, No. 1, 1965, 23-41.

[RS] J.A. Robinson and E.E. Siebert, *LOGLISP: Motivation, Design and Implementation,* in: Logic Programming, (Clark, K.L. and S.-A. Tärnlund, eds), Academic Press, New York, 1982, 299-313.

[SS] J. Sebelik and P. Stepanek, *Horn Clause Programs for Recursive Functions,* in: Logic Programming, (Clark, K.L. and S.Å. Tärnlund, eds), Academic Press, New York, 1982, 324-340.

[Sha] E.Y. Shapiro, *A Subset of Concurrent PROLOG and its Interpreter,* Technical Report TR-003, ICOT, Tokyo, 1983.

[She] J.C. Shepherdson, *Negation as Failure: A Comparison of Clark's Completed Data Base and Reiter's Closed World Assumption,* Journal of Logic Programming, vol. 1, No. 1, 1984, 51-79.

[She1] J.C. Shepherdson, *Undecidability of Horn Clause Logic and Pure Prolog,* Unpublished manuscript, 1985.

[She2] J.C. Shepherdson, *Negation in Logic Programming,* in: Foundations of Deductive Databases and Logic Programming, (Minker, J. ed.), Morgan Kaufmann, Los Altos, 1987.

[Shm] O. Shmueli, *Decidability and Expressiveness Aspects of Logic Queries,* in: Proc. of the 6th ACM-SIGACT-SIGMOD Symposium on Principles of Database Systems, San Diego, 1987, 237-249.

[S] J. Shoenfield, *Mathematical Logic,* Addison-Wesley, Reading, Mass., 1967.

[Si] J.H. Siekmann, *Universal Unification,* in: Proc. of the 7th Conference on Automated Deduction, Lecture Notes in Computer Science 170, Springer-Verlag, Berlin, 1984, 1-42.

[Sm] R.M. Smullyan, *Theory of Formal Systems,* Annals of Mathematical Studies, vol. 47, Princeton University Press, 1961.

[ST] E.A. Sonenberg and R. Topor, *Logic Programs and Computable Functions,* Technical Report 87/5, Department of Computer Science, University of Melbourne, 1987.

[SSh]L. Sterling and E.Y. Shapiro, *The Art of Prolog,* The MIT Press, Cambridge, Mass. 1986.

[Ta] A. Tarski, *A Lattice-theoretical Fixpoint Theorem and its Applications,* Pacific J. Math. vol. 5, 1955, 285-309.

[T] S.-A. Tärnlund, *Horn Clause Computability,* BIT, vol. 17, No. 2, 215-226.

[VG]A. van Gelder, *Negation as Failure using Tight Derivations for General Logic Programs,* in: Proc. 3rd IEEE Symp. on Logic Programming, Salt Lake City, 1986, 127-138.

[V] L. Vieille, *A Database-Complete Proof Procedure Based on SLD-Resolution,* in: Proc. of the Fourth International Conference on Logic Programming, 1987, The MIT Press, 74-103.

[W] A. Walker, *Syllog: an Approach to PROLOG for Non-programmers,* in: Logic Programming and its Applications, (Van Caneghem, M. and D.H.D. Warren, eds), Ablex, 1986, 32-49.