CHAPTER 10

# Logic Programming

Krzysztof R. APT

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

*Contents*



HANDBOOK OF THEORETICAL COMPUTER SCIENCE
Edited by J. van Leeuwen

## 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 [53] in 1974 and for a long time—in the case of computer science—not much happened. But, sixteen 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 theoretical framework. This close connection led to the adoption of logic programming as the basis for the influential 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 in a large part by computer scientists.)

The aim of this chapter 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 chapter we attempt to adhere to the notation of Lloyd [64], 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 of the chapter. It is hoped that this will facilitate its reading and will allow a better understanding of the structure of its subject.

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

In Section 3 a semantics is introduced with the purpose of establishing soundness of SLD-resolution and several forms of its completeness. Most of these results are collected in the Success Theorem 3.25.

In Section 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. Section 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 effect is directed at an analysis of a stronger 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 5.32 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 Section 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. Finally, in Section 7 we investigate a subclass of general programs, called *stratified programs*, concentrating on their semantics.

The chapter 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 [70, 92].

A *first-order language* consists of an alphabet and all formulas defined over it. An *alphabet* consists of the following classes of symbols:

- *variables* denoted by $x, y, z, v\ u, \ldots,$
- *constants* denoted by $a, b\ c, d, \ldots,$
- *function symbols* denoted by $f, g, ., \ldots,$
- *relation symbols* denoted by $p, q, r, \ldots,$
- *propositional constants*, which are **true** and **false**,
- *connectives*, which are $\neg$ (negation), $\vee$ (disjunction), $\wedge$ (conjunction), $\rightarrow$ (implication) and $\leftrightarrow$ (equivalence),
- *quantifiers*, which are $\exists$ (there exists) and $\forall$ (for all),
- *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 first-order language is thus determined by its 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 variable is a term,
- a constant is a term,
- if $f$ is an $n$-ary function and $t_1, \ldots, t_n$ are terms then $f(t_1, \ldots, t_n)$ is a term.

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

- if $p$ is an $n$-ary relation and $t_1, \ldots, t_n$ are terms then $p(t_1, \ldots, t_n)$ is a formula (called a *atomic formula*, or just an *atom*),
- **true** and **false** are formulas,
- if $F$ and $G$ are formulas then so are $\neg F$, $(F \wedge G)$, $(F \wedge G)$, $(F \rightarrow G)$ and $(F \leftrightarrow G)$,
- 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* i.e. between the arguments. Atomic formulas are denoted by $A$, $B$ and formulas in general by $F$, $G$. If $F$ is a quantifier-free formula with variables $x_1, \ldots, x_n$, we write $\exists F$ for $\exists x_1 \ldots \exists x_n F$ and $\forall F$ for $\forall x_1 \ldots \forall x_n F$. Formulas of the form $\forall F$ are called *universal formulas*. A term or formula with no variables is called *ground*.

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

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

Thus 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 but it borrowed most of its terminology from automatic theorem proving. 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

$$\forall(L_1 \vee \cdots \vee L_m)$$

where $L_1, \ldots, L_m$ are literals, 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

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

where $A_1, \ldots, A_k$ is the list of all positive literals among $L_1, \ldots, L_m$, called *conclusions* and $B_1, \ldots, B_n$ 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 $\leftarrow B_1, \ldots, B_n$. 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 $L_1 \vee \ldots \vee L_m$ 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 nonempty 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

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

where $A_1, \ldots, A_k$ are positive literals and $L_1, \ldots, L_n$ 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 $A_1, \ldots, A_k \leftarrow L_1, \ldots, L_n$ represents the formula

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

Now, a *general logic program* (or just a *general program*) is a finite nonempty set of general program clauses. Note that **true** and **false** are not used to define (general) programs. These formulas will be however needed later, in Subsection 5.5.

With each (general) 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 (general) 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 $A \leftarrow B_1, \ldots, B_n$. One is: to solve $A$ solve $B_1, \ldots, B_n$. The other is: $A$ is true if $B_1, \ldots, B_n$ 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 Section 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

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

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

The notation implies that the variables $x_1, \ldots, x_n$, are different. We also assume that, for $i = 1, \ldots, n$, $x_i \neq t_i$. A pair $x_i/t_i$ is called a *binding*. If all $t_1, \ldots, t_n$ are ground then $\theta$ is called *ground*. If $\theta$ is a 1-1 and onto mapping from its domain to itself, then $\theta$ is called a *renaming*. In other words, $\theta$ is a renaming if it is a permutation of the variables from its domain.

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 $\theta$, $E\theta$ stands for the result of applying $\theta$ to $E$ which is obtained by *simultaneously* replacing each occurrence in $E$ of a variable from the domain of $\theta$ by the corresponding term. The resulting expression $E\theta$ is called an *instance* of $E$. An instance is called *ground* if it contains no variables.

If $\theta$ is a renaming then $E\theta$ is called a *variant* of $E$. Thus, for example, $x < y' + z'$ is a variant of $x < y + z$, since $x < y' + z' \equiv (x < y + z)\{y/y', z/z', y'/y, z'/y\}$, whereas $x < y' + x$ is not.

The following lemma, whose proof we omit, clarifies the concept of a variant and implies that "being a variant of" is a symmetric relation.

**2.1.** LEMMA. *For all expressions $E$ and $F$*

*$E$ is a variant of $F$ iff $E$ is an instance of $F$ and $F$ is an instance 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. Given an atom $A$ we denote by $[A]$ the set of all its ground instances.

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

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

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 $y_i \in \{x_1, \ldots, x_n\}$.

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

**2.2.** LEMMA. *For all substitutions $\theta, \eta$ and $\gamma$ 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 $\theta$ is *more general* than a substitution $\eta$ if for some substitution $\gamma$ 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 $\theta$ we have $A\theta \equiv B\theta$, then $\theta$ is called a *unifier* of $A$ and $B$ and we then say that $A$ and $B$ are *unifiable*. A unifier $\theta$ 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 [84].

**2.3.** THEOREM (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 nonexistence of a unifier.*

PROOF. We follow here the presentation of Lassez, Maher and Marriott [62]. We present an algorithm based upon Herbrand's original algorithm [45, p. 148] which deals with solutions of finite sets of term equations. This algorithm was first presented in [71].

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

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

A substitution $\theta$ such that $s_1\theta \equiv t_1\theta, \ldots, s_n\theta \equiv t_n\theta$ is called a *unifier* of the set of equations $\{s_1 = t_1, \ldots, s_n = t_n\}$. Thus the set of equations $\{s_1 = t_1, \ldots, s_n = t_n\}$ has the same unifiers as the atoms $p(s_1, \ldots, s_n)$ and $p(t_1, \ldots, t_n)$. 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 $\{x_1 = u_1, \ldots, x_n = u_n\}$ where $x_i$'s are distinct variables and none of them occurs in a term $u_j$.

A solved set of equations $\{x_1 = u_1, \ldots, x_n = u_n\}$ determines the substitution $\{x_1/u_1, \ldots, x_n/u_n\}$. 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. Nondeterministically choose from the set of equations an equation of a form below and perform the associated action.

(1) $f(s_1, \ldots, s_n) = f(t_1, \ldots, t_n)$: *replace by the equations* $s_1 = t_1, \ldots, s_n = t_n$;

(2) $f(s_1, \ldots, s_n) = g(t_1, \ldots, t_m)$ 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 $\theta$, $x\theta \equiv t\theta$ implies that the substitutions $\theta$ 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.

**2.4.** EXAMPLE. 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 again 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 now be 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 $\theta$ *idempotent* if $\theta\theta=\theta$. Call a unifier of $\theta$ of two atoms $A$ and $B$ *relevant* if all variables which appear either in the domain of $\theta$ or in the terms from the range of $\theta$ also appear in $A$ or $B$. In Section 2.7 we shall rely on the following observation.

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

PROOF. The unifier produced by the procedure used in the proof of Theorem 2.3 is of the form $\{x_1/u_1,\ldots,x_n/u_n\}$ where none of the variables $x_i$ occurs in a term $u_j$, so it is idempotent. Moreover, in the unification algorithm no variables from outside the unified atoms are introduced. Thus the produced mgu is relevant. □

One can prove that idempotent mgu's are relevant but we shall not need this observation in future.

Given a substitution $\theta$ denote its domain by $dom(\theta)$ and the set of variables which appear in a term from the range of $\theta$ by $r(\theta)$. Given an expression $E$, denote by $var(E)$ the set of variables which appear in it. The following observation will be needed in Subsection 2.7.

**2.6.** LEMMA. *Let $E$ be an expression and $\theta$ an idempotent substitution. Then*

$$var(E\theta)\cap dom(\theta)=\emptyset.$$

PROOF. It is easy to see that for any substitution $\theta$

$$var(E\theta)\cap dom(\theta)\subseteq r(\theta). \tag{2.1}$$

But for an idempotent substitution $\theta$ also

$$dom(\theta)\cap r(\theta)=\emptyset. \tag{2.2}$$

(2.1) and (2.2) imply the claim. □

*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 *SLD-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 $N = \leftarrow A_1, \ldots, A_n$ $(n \geqslant 1)$ be a ground negative clause and suppose that for some $i$, $1 \leqslant i \leqslant n$ and $k \geqslant 0$, $C = A_i \leftarrow B_1, \ldots, B_k$ is a clause from $P$. Then

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

is the result of replacing $A_i$ in $N$ by $B_1, \ldots, B_k$ 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 $N = \leftarrow A_1, \ldots, A_k$ 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 the semantics for first-order logic (which is explained in Subsection 3.1) we note that $N$ stands for $\neg A_1 \vee \cdots \vee \neg A_k$, so its negation stands for $\neg(\neg A_1 \vee \cdots \vee \neg A_k)$ which is semantically equivalent to $A_1 \wedge \cdots \wedge A_k$. Thus a refutation of $N$ can be viewed as a proof of $A_1 \wedge \cdots \wedge A_k$.

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

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

a *resolvent of $N$ and $C$ with the mgu $\theta$.* Thus a resolvent is obtained by performing the following four steps:

(a) select an atom $A_i$,
(b) try to unify $A$ and $A_i$,
(c) if (b) succeeds then perform the replacement of $A_i$ by $B_1, \ldots, B_k$ in $N$,
(d) apply to the resulting clause the mgu $\theta$ 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 $N_0, N_1, \ldots$ of negative clauses where $N = N_0$, together with a sequence $C_0, C_1, \ldots$ of variants of clauses from $P$ and a sequence $\theta_0, \theta_1, \ldots$ of substitutions such that, for all $i = 0, 1, \ldots,$

(i) $N_{i+1}$ is a resolvent of $N_i$ and $C_i$ with the mgu $\theta_i$,

(ii) $C_i$ does not have a variable in common with $N_0, C_0, \ldots, C_{i-1}$.

The clauses $C_0, C_1, \ldots$ 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*. An SLD-derivation is called *failed* if it is finite and it is not a refutation.

A new element in this definition is the use of variants that satisfy (ii) instead of the original clauses. This condition is called *standardization apart*. Its relevance will be extensively discussed in Section 2.7. 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 $N = \leftarrow A_1, \ldots, A_k$ can be viewed as a contradiction. We can then conclude that we proved the negation of $N$. But $N$ stands for $\forall x_1 \ldots \forall x_s(\neg A_1 \vee \cdots \vee \neg A_k)$, where $x_1, \ldots, x_s$ are all variables appearing in $N$, so its negation stands for $\neg \forall x_1 \ldots \forall x_s(\neg A_1 \vee \cdots \vee \neg A_k)$ which is semantically equivalent (see Subsection 3.1) to $\exists x_1 \ldots \exists x_s(A_1 \wedge \cdots \wedge A_k)$. 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 $x_1, \ldots, x_s$. Thus the existence of an SLD-refutation for $P \cup \{N\}$ can be viewed as a proof of the formula $(A_1 \wedge \cdots \wedge A_k)\theta_0 \ldots \theta_m$. We justify this statement in Subsection 3.2.

The restriction of $\theta_0 \ldots \theta_m$ to the variables of $N$ is called a *computed answer substitution* for $P \cup \{N\}$. According to the definition of SLD-derivation, the following two choices are made in each step of constructing a new resolvent:

- choice of the selected atom,
- choice of the input clause whose conclusion unifies with the selected atom.

Now, the first choice is in general dependent on the whole "history" of the derivation up to the current resolvent. Such a history consists of a sequence $N_0, N_1, \ldots, N_{k-1}$ of goals with selected atoms, a goal $N_k$, a sequence $C_0, C_1, \ldots, C_{k-1}$ of input clauses and a sequence $\theta_0, \theta_1, \ldots, \theta_{k-1}$ of substitutions such that, for all $i = 0, \ldots, k-1$, $N_{i+1}$ is a resolvent of $N_i$ and $C_i$ with mgu $\theta_i$ where the selected atom of $N_i$ is used in step (a) above. Let now *HIS* stand for the set of all such histories in which the last goal $N_k$ is nonempty.

By a *selection rule* $R$ we now mean a function which, when applied to an element of *HIS* with the last goal $N_k = \leftarrow A_1, \ldots, A_l$, yields an atom $A_j$, $1 \leqslant j \leqslant l$. Such a general definition allows us to select different atoms in resolvents that occur more than once in the derivation or, in general, in identical resolvents with different histories.

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, for each nonempty goal $M$ of this SLD-derivation with a history $H$, $R(H)$ is the selected atom of $M$.

Now, SLD stands for Selection rule-driven Linear resolution 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 $3 \times 3$ grid filled with eight moveable tiles. Our goal is to rearrange the tiles so that the blank one is in 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, twenty-four pairs are adjacent. (A more succinct representation would be possible if addition and subtraction functions are 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($x$, $y$. $l$)←adjacent($x$, $y$), configuration($y$, $l$) (b)

where $y.l$ 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) $\{l/l_1\}$, $\{x/5, l/y.l_1\}$ |
| ←**adjacent(5, $y$)**, configuration($y$, $l_1$) | (a) $\{x/x_1, y/y_1\}$, $\{x_1/5, y_1/y\}$ |
| ←**adjacent($y$, 5)**, configuration($y$, $l_1$) | adjacent(4, 5)←, $\{y/4\}$ |
| ←**configuration(4, $l_1$)** | (b) $\{x/x_2, y/y_2, l/l_2\}$, $\{x_2/4, l_1/y_2.l_2\}$ |
| ←**adjacent(4, $y_2$)**, configuration($y_2$, $l_2$) | (a) $\{x/x_3, y/y_3\}$, $\{x_3/4, y_3/y_2\}$ |
| ←**adjacent($y_2$, 4)**, configuration($y_2$, $l_2$) | adjacent(1, 4)←, $\{y_2/1\}$ |
| ←**configuration(1, $l_2$)** | configuration(1, nil)←, $\{l_2/\text{nil}\}$ |

□

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 $\{l/y.l_1\}$, $\{y/4\}$, $\{l_1/y_2.l_2\}$, $\{y_2/1\}$, $\{l_2/\text{nil}\}$.

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 $(5, l)$ 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) $\{l/l_1\}, \{x/5, l/y.l_1\}$ |
| $\leftarrow$ **adjacent(5, $y$)**, configuration($y, l_1$) | (a) $\{x/x_1, y/y_1\}, \{x_1/5, y_1/y\}$ |
| $\leftarrow$ **adjacent($y$, 5)**, configuration($y, l_1$) | (a) $\{x/x_2, y/y_2\}, \{x_2/y, y_2/5\}$ |
| $\leftarrow$ **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. *Properties of SLD-derivations*

In the next sections we shall need the following two lemmas concerning SLD-derivations. Both of them rely on the condition of standardizing apart introduced in Subsection 2.5.

**2.7.** LEMMA. *Let $N_0, N_1, \ldots$ be an SLD-derivation with a sequence $C_0, C_1, \ldots$ of input clauses and a sequence $\theta_0, \theta_1, \ldots$ of mgu's. Suppose that all $\theta_i$'s are idempotent and relevant. Then, for all $m \geqslant 0$ and $n > m$,*

(1) $var(N_n) \cap dom(\theta_m) = \emptyset$,

(2) $var(N_n\theta_n) \cap dom(\theta_m) = \emptyset$.

PROOF. (1) We prove by induction on $i$ that, for all $i > 0$,

$$var(N_{m+i}) \cap dom(\theta_m) = \emptyset. \tag{2.3}$$

$N_{m+1}$ is of the form $E\theta_m$, so for $i = 1$ (2.3) is the consequence of Lemma 2.6. Suppose now that (2.3) holds for some $i > 0$. Since each $\theta_j$ is relevant, by the form of $N_{j+1}$, for all $j \geqslant 0$,

$$var(N_{j+1}) \subseteq var(N_j) \cup var(C_j). \tag{2.4}$$

Since $\theta_m$ is relevant,

$$dom(\theta_m) \subseteq var(N_m) \cup var(C_m), \tag{2.5}$$

so using (2.4) $m$ times

$$dom(\theta_m) \subseteq var(N_0) \cup var(C_0) \cup \cdots \cup var(C_m). \tag{2.6}$$

Now

$$
\begin{aligned}
&var(N_{m+i+1}) \cap dom(\theta_m) \\
&\subseteq (var(N_{m+i}) \cap dom(\theta_m)) \cup (var(C_{m+i}) \cap dom(\theta_m)) && \text{(by (2.4) with } j = m+i) \\
&\subseteq var(C_{m+i}) \cap (var(N_0) \cup var(C_0) \cup \cdots \cup var(C_m)) && \text{(by (2.3) and (2.6))} \\
&\subseteq \emptyset && \text{(by standardizing apart).}
\end{aligned}
$$

This proves the induction step and concludes the proof of (1).

(2) It suffices to note that, by assumption on the $\theta_i$'s

$$var(N_n\theta_n) \subseteq var(N_n) \cup var(C_n)$$

and use (1), (2.6) and standardizing apart. □

We now show that up to renaming the computed answer substitution of an SLD-derivation does not depend on the choice of variables in the input clauses. To this end we prove a slightly stronger result first, which uses the notion of a resultant of an SLD-derivation.

Given a goal $N = \leftarrow A_1, \ldots, A_k$ we denote by $N^\sim$ the formula $A_1 \wedge \cdots \wedge A_k$. Then $\square^\sim$ is the empty conjunction which we identify with **true**. Given an SLD-derivation $N_0, N_1, \ldots$ with a sequenceof mgu's $\theta_0, \theta_1, \ldots$ of length $\geqslant i$, by a *resultant (of level i)* we mean the formula

$$N_i^\sim \rightarrow N_0^\sim \theta_0 \ldots \theta_{i-1}.$$

Thus the resultant of level 0 is the formula $N_0^\sim \rightarrow N_0^\sim$.

**2.8.** LEMMA (Variant Lemma) (Lloyd and Shepherdson [66]). *Let $N_0, N_1, \ldots$ and $N_0', N_1', \ldots$ be two SLD-derivations of $P \cup \{N\}$ where $N = N_0$ and $N = N_0'$, with the input clauses $C_0, C_1, \ldots$ and $C_0', C_1', \ldots$ respectively. Suppose that each $C_i'$ is a variant of $C_i$ and that in each $N_i'$ atoms in the same positions as in $N_i$ are selected. Also, suppose that all mgu's used in the two SLD-derivations are relevant. Then the resultants of these two SLD-derivations are their respective variants.*

PROOF. We prove the claim by induction on the level $i$ of resultants. For $i = 0$ there is nothing to prove. Assume the claim holds for some $i \geqslant 0$. Let $\theta_0, \theta_1, \ldots$ be the mgu's of the first SLD-derivation and $\theta_0', \theta_1', \ldots$ the mgu's of the second SLD-derivation. By the induction hypothesis

$$Res = N_i^\sim \rightarrow N_0\theta_0 \ldots \theta_{i-1}$$

is a variant of

$$Res' = N_i'^\sim \rightarrow N_0'\theta_0' \ldots \theta_{i-1}'.$$

Thus, for a renaming $\theta$ with $dom(\theta) \subseteq var(Res')$,

$$Res \equiv Res'\theta. \tag{2.7}$$

By assumption $C_i$ is a variant of $C'_i$. Thus for a renaming $\eta$ with $dom(\eta) \subseteq var(C'_i)$

$$C_i \equiv C'_i\eta. \tag{2.8}$$

Given two substitutions $\sigma$ and $\xi$ with disjoint domains, we denote by $\sigma \cup \xi$ their *union* which is defined in the obvious way. Put now $\gamma = (\theta \cup \eta)\theta_i$. We prove the following four facts:

(1) *$\gamma$ is well defined.*
(2) *For some $\sigma$, $\gamma = \theta'_i\sigma$.*
(3) $N_{i+1} \equiv N'_{i+1}\sigma$.
(4) $N_0\theta_0 \ldots \theta_i \equiv N'_0\theta'_0 \ldots \theta'_i\sigma$.

*Re* (1): We only need to show that the domains of $\theta$ and $\eta$ are disjoint. We first show that

$$var(Res') \cap var(C'_i) = \emptyset. \tag{2.9}$$

By the assumption, $\theta'_0, \ldots, \theta'_{i-1}$ are relevant, so by the same argument as the one used in the previous lemma, but now applied to the ranges of $\theta'_j$ instead of their domains, we get, for $j = 0, \ldots, i-1$,

$$r(\theta'_j) \subseteq (N'_0) \cup var(C'_0) \cup \cdots \cup var(C'_{i-1}). \tag{2.10}$$

Also, as in the proof of the previous lemma

$$var(N'_i) \subseteq var(N'_0) \cup var(C'_0) \cup \cdots \cup var(C'_{i-1}). \tag{2.11}$$

Now

$$\begin{aligned} var(Res') &= var(N'_i) \cup var(N'_0\theta'_0 \ldots \theta'_{i-1}) \\ &\subseteq var(N'_i) \cup var(N'_0) \cup r(\theta'_0) \cup \ldots \cup r(\theta'_{i-1}) \\ &\subseteq var(N'_0) \cup var(C'_0) \cup \ldots var(C'_{i-1}) \quad \text{(by (2.10) and (2.11)} \end{aligned}$$

so (2.9) follows from the standardizing apart. Now note that $dom(\theta) \subseteq var(Res')$ and $dom(\eta) \subseteq var(C'_i)$, so by (2.9) the domains of $\theta$ and $\eta$ are indeed disjoint.

*Re* (2): Let $B'$ be an atom from $C'_i$. Then $var(B') \subseteq var(C'_i)$, so by (2.9)

$$var(B') \cap dom(\theta) = \emptyset, \tag{2.12}$$

since $dom(\theta) \subseteq var(Res')$. Similarly, also by (2.9), for an atom $A'$ from $N'_i$,

$$var(A') \cap dom(\eta) = \emptyset. \tag{2.13}$$

Thus by (2.12), for an atom $B'$ from $C'_i$,

$$B'(\theta \cup \eta) \equiv B'\eta \tag{2.14}$$

and by (2.13), for an atom $A'$ from $N'_i$,

$$A'(\theta \cup \eta) \equiv A'\theta. \tag{2.15}$$

Let

$$C_i = B_0 \leftarrow B_1, \ldots, B_k, \qquad N_i = \leftarrow A_1, \ldots, A_m,$$
$$C'_i = B'_0 \leftarrow B'_1, \ldots, B'_k, \qquad N'_i = \leftarrow A'_1, \ldots, A'_m.$$

By (2.7) and (2.14), for $j=0,\ldots,k$,

$$B_j \equiv B'_j(\theta\cup\eta) \tag{2.16}$$

and by (2.8) and (2.15), for $j=1,\ldots,m$,

$$A_j \equiv A'_j(\theta\cup\eta). \tag{2.17}$$

Let now $A'_l$ be the selected atom of $N'_i$. Then $A_l$ is the selected atom of $N_i$ and

$$A_l\theta_i \equiv B_0\theta_i. \tag{2.18}$$

Now

$$\begin{aligned} A'_l\gamma &\equiv A'_l(\theta\cup\eta)\theta_i \\ &\equiv A_l\theta_i && \text{(by (2.17))} \\ &\equiv B_0\theta_i && \text{(by (2.18))} \\ &\equiv B'_0(\theta\cup\eta)\theta_i && \text{(by (2.16))} \\ &\equiv B'_0\gamma, \end{aligned}$$

so $\gamma$ is a unifier of $A'_l$ and $B'_0$. Now, since $\theta_i$ is an mgu of $A'_l$ and $B'_0$, for some $\sigma$, $\gamma=\theta'_i\sigma$.

*Re* (3): We have

$$\begin{aligned} N_{i+1} &\equiv \leftarrow(A_1,\ldots,A_{l-1},B_1,\ldots,B_k,A_{l+1},\ldots,A_m)\theta_i \\ &\equiv \leftarrow(A'_1,\ldots,A'_{l-1},B'_1,\ldots,B'_k,A'_{l+1},\ldots,A'_m)(\theta\cup\eta)\theta_i \\ &\qquad\qquad \text{(by (2.16) and (2.17))} \\ &\equiv \leftarrow(A'_1,\ldots,A'_{l-1},B'_1,\ldots,B'_k,A'_{l+1},\ldots,A'_m)\gamma \\ &\equiv \leftarrow(A'_1,\ldots,A'_{l-1},B'_1,\ldots,B'_k,A'_{l+1},\ldots,A'_m)\theta'_i\sigma \quad \text{(by fact (2))} \\ &\equiv N'_{i+1}\sigma \end{aligned}$$

*Re* (4): We have $dom(\eta)\subseteq var(C'_i)$, so by (2.9)

$$var(N'_0\theta'_0\ldots\theta'_{i-1})\cap dom(\eta)=\emptyset. \tag{2.19}$$

Now

$$\begin{aligned} N_0\theta_0\ldots\theta_i &\equiv N'_0\theta'_0\ldots\theta'_{i-1}\theta\theta_i && \text{(by (2.7))} \\ &\equiv N'_0\theta'_0\ldots\theta'_{i-1}(\theta\cup\eta)\theta_i && \text{(by (2.19))} \\ &\equiv N'_0\theta'_0\ldots\theta'_{i-1}\gamma \\ &\equiv N'_0\theta'_0\ldots\theta'_{i-1}\theta'_i\sigma && \text{(by the form of } \gamma). \end{aligned}$$

Now, putting facts (3) and (4) together we see that the resultant of level $i+1$ of the first SLD-derivation is an instance of the resultant of level $i+1$ of the second SLD-derivation. By symmetry the resultant of level $i+1$ of the second SLD-derivation is an instance of the resultant of level $i+1$ of the first SLD-derivation. By Lemma 2.1 these resultants are the variants of each other. □

**2.9.** COROLLARY (Variant Corollary). *Let $\Phi$ and $\Psi$ be two SLD-derivations of $P\cup\{N\}$ satisfying the conditions of Lemma 2.8. Suppose that $\Phi$ is an SLD-refutation with a*

*computed answer substitution $\theta$. Then $\Psi$ is an SLD-refutation with a computed answer substitution $\eta$ such that $N\theta$ is a variant of $N\eta$.*

PROOF. It suffices to consider resultants of level $k$ of $\Phi$ and $\Psi$, where $k$ is the length of the SLD-refutation $\Psi$, and apply the previous lemma. □

The above corollary shows that the existence of an SLD-refutation does not depend on the choice of variables in the input clauses.

To be able to use the results of this section we shall assume from now on that *all mgu's used in all SLD-derivations are idempotent and relevant.*

### *2.8. 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.

Let $P$ be a program, $N$ a goal and $R$ a selection rule. The SLD-tree for $P \cup \{N\}$ via $R$ groups all SLD-derivations of $P \cup \{N\}$ via $R$. Formally the *SLD-tree for $P \cup \{N\}$ via $R$* is a tree such that

- its branches are SLD-derivations of $P \cup \{N\}$ via $R$,
- every node $N'$ has exactly one descendant for every clause $C$ of $P$ such that the selected atom $A$ of $N'$ unifies with the head of a variant $C'$ of $C$. This descendant is a resolvent of $N'$ and $C'$ with $A$ being the selected atom of $N'$.

We call an SLD-tree *successful* if it contains the empty clause.

The SLD-trees for $P \cup \{N\}$ can differ in size and form.

**2.10.** EXAMPLE. (Apt and van Emden [4]). 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 $P \cup \{\leftarrow \text{path}(x, c)\}$. The selected atoms are put in bold, used clauses and performed substitutions are indicated. The input clauses at the level $i$ are obtained from the original clauses by adding the subscript "$i$" to all variables which were used earlier in the derivation. In this way the standardizing apart condition is satisfied. Note that the first tree is finite while the second one is infinite. Both trees contain the empty clause.

### *2.9. Bibliographic remarks*

The concepts of unification, resolution and standardization apart were introduced in [84]. Efficient unification algorithms were proposed by Paterson and Wegman [79],

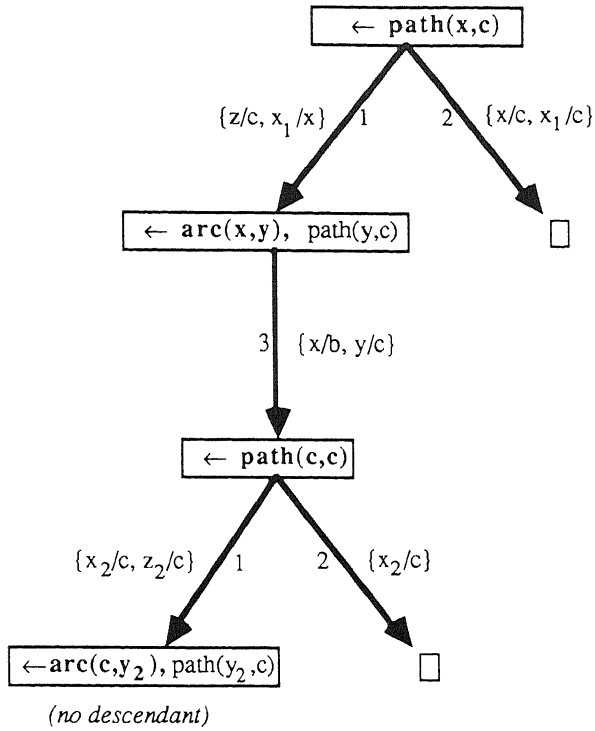


Fig. 1.

and Martelli and Montanari [71]. See also the survey on unification by Siekmann [93].

SLD-resolution is a special case of SL-resolution of Kowalski and Kuehner [57] and was proposed as a basis for programming in Kowalski [53]. The name was first used in [4] where also the notions of a success set and SLD-trees were formally introduced. SLD-trees were informally used in [21] where they were called evaluation trees.

The selection rule was originally required to be a function defined on sequences of atoms. Our formulation follows the suggestion of Shepherdson [88, p. 62]. The proof of Lemma 2.8 differs from the original proof. Corollary 2.9 was independently established in [52].

## 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.

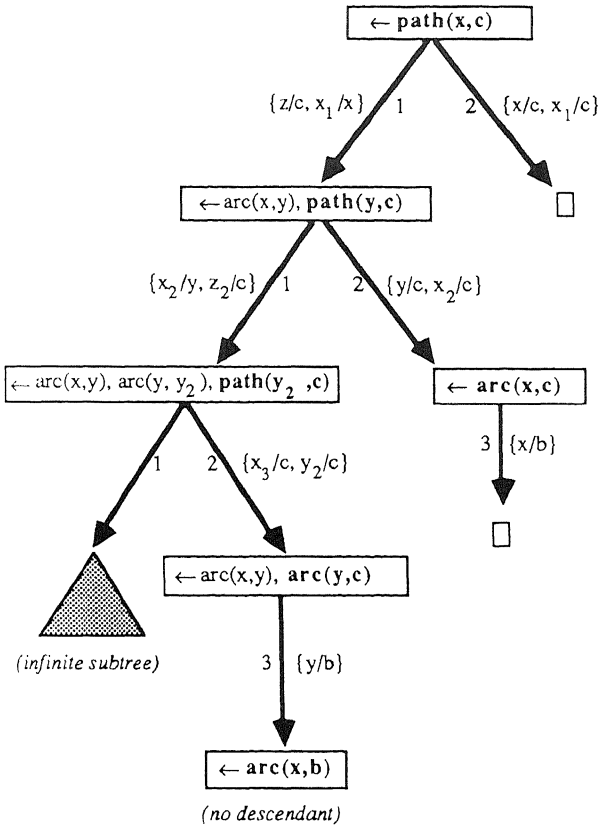


Fig. 2.

in [70], [92]. We begin by defining an interpretation. An *interpretation* $I$ for a first-order language $L$ consists of

- a nonempty set $D$, called the *domain* of $I$,
- an assignment for each constant $c$ in $L$ of an element $c_I$ of $D$,
- an assignment for each $n$-ary function $f$ in $L$ of a mapping $f_I$ from $D^n$ to $D$,
- 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 of an interpretation. 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 from $D$.

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

- for a constant $c$ we define $\sigma(c)$ as $c_I$ (thus $\sigma(c)$ does not depend on $\sigma$),
- if $f(t_1, \ldots, t_n)$ is a term then we define $\sigma(f(t_1, \ldots, t_n))$ as $f_I(\sigma(t_1), \ldots, \sigma(t_n))$, the result

of applying the mapping $f_I$ to the sequence of values associated with the terms $t_1, \ldots, t_n$.

Observe that for a ground term $t$, $\sigma(t)$ has the same value for all $\sigma$.

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

- if $p(t_1, \ldots, t_n)$ is an atomic formula then

$$I \models_\sigma p(t_1, \ldots, t_n) \text{ iff } (\sigma(t_1), \ldots, \sigma(t_n)) \in p_I,$$

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

- $I \models_\sigma$ **true**, not $I \models_\sigma$ **false**,
- if $F$ and $G$ are formulas then

$$\begin{aligned} I &\models_\sigma \neg F \quad \text{iff} \quad \text{not } I \models_\sigma F, \\ I &\models_\sigma F \vee G \quad \text{iff} \quad I \models_\sigma F \text{ or } I \models_\sigma G, \\ I &\models_\sigma \forall x F \quad \text{iff} \quad I \models_{\sigma[x/d]} F \text{ for all } d \in D. \end{aligned}$$

Here $\sigma[x/d]$, for a state $\sigma$, an element $d$ of $D$ and a variable $x$, stands for the state which differs from $\sigma$ 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 $\forall$:

$$\begin{aligned} F \wedge G &\text{ as } \neg(\neg F \vee \neg G), \\ F \rightarrow G &\text{ as } \neg F \vee G, \\ F \leftrightarrow G &\text{ as } (F \rightarrow G) \wedge (G \rightarrow F) \quad \text{(and then using the above two definitions)}, \\ \exists x F &\text{ as } \neg \forall x \neg F. \end{aligned}$$

Finally, we say that the formula $F$ *is true in the interpretation* $I$, and write $I \models F$, when for all states $\sigma$, $I \models_\sigma F$. 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 $S \models S'$ and omit the $\{$ and $\}$ brackets if any of these sets has exactly one element. $S$ and $S'$ are *semantically equivalent* if both $S \models S'$ and $S' \models S$ hold.

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

$$\begin{aligned} &\neg \forall x_1 \ldots \forall x_s F \leftrightarrow \exists x_1 \ldots \exists x_s \neg F, \\ &\neg (A_1 \vee \cdots \vee A_n) \leftrightarrow \neg A_1 \wedge \cdots \wedge \neg A_n, \\ &\neg \neg F \leftrightarrow F. \end{aligned}$$

*3.2. Soundness of the SLD-resolution*

Recall that, for a goal $N = \leftarrow A_1, \ldots, A_k$, $N^\sim$ stands for the formula $A_1 \wedge \cdots \wedge A_k$. Then $\Box^\sim$ is the empty conjunction, so it is valid. The following lemma is immediate.

**3.1.** LEMMA. *If $M$ is a resolvent of $N$ and a clause $C$ with an mgu $\theta$ then $C \models M^\sim \rightarrow N^\sim\theta$.*

As a consequence we obtain the following theorem due to Clark [22] justifying the statement made in Subsection 2.5.

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

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

$$P \models \Box^\sim \rightarrow N^\sim\theta_0 \ldots \theta_n$$

which implies the claim. □

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

Another straightforward consequence of Lemma 3.1, which will not be used in the sequel, is that all resultants of an SLD-refutation of $P \cup \{N\}$ are semantic consequences of $P$.

**3.4.** EXAMPLE. Reconsider now the program $P$ studied in the example in Subsection 2.6 with the goal $\leftarrow$ configuration(5, $l$). Since we exhibited there an SLD-refutation of $P \cup \{\leftarrow$ configuration(5, $l$)$\}$, we conclude by the above corollary that $P \cup \{\leftarrow$configuration(5, $l$)$\}$ is inconsistent, that is, $P \models \exists l$ configuration(5, $l$). More specifically, by the Soundness Theorem we have $P \models$configuration(5, $l$)$\theta_0 \ldots \theta_7$ 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 the $U_P$ 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 $f(t_1, \ldots, t_n)$ to the sequence $t_1, \ldots, t_n$ of ground terms,
(d) if $r$ is an $n$-ary relation in $L$ then it is assigned to a set of $n$-tuples of ground terms.

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 $\{(t_1, \ldots, t_n)\colon r(t_1, \ldots, t_n) \in I\}$ 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 as 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.

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 simply drop the condition that a domain of an interpretation is nonempty 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 why Herbrand models naturally arise when studying logic programs.

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

PROOF. For an interpretation $I$ let $I_H = \{A : A \text{ is a ground atom and } I \models A\}$ 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 ground formulas. From this the lemma follows. □

**3.6.** COROLLARY. *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 recurring 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 a *minimal model of $S$* if no proper subset of it is a Herbrand model of $S$. The least model is minimal but the converse is not always true (take for example $S = \{A \vee B\}$ with $A, B$ ground atoms).

*3.4. The immediate consequence operator*

To study Herbrand models of programs, following [31], we introduce the *immediate consequence operator* $T_P$ mapping Herbrand interpretations to Herbrand interpretations. For a program $P$ and a Herbrand interpretation $I$, we put

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

Alternatively, for a ground atom $A$,

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

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 [31] relates Herbrand models of $P$ with the operator $T_P$.

**3.7.** LEMMA. *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($P$), $I \models B_1 \wedge \cdots \wedge B_n$ 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. [10]) 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 $\bigcup A$ and $\bigcap A$ 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 $I_0 \subseteq I_1 \subseteq \ldots$,

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

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 if, for every infinite sequence $I_0 \subseteq I_1 \subseteq \ldots$, it holds that

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

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.

**3.8.** THEOREM (Fixpoint Theorem) (Knaster and Tarski [97]). *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

$$T\uparrow 0(I) = I, \qquad T\uparrow(n+1)(I) = T(T\uparrow n(I)), \qquad T\uparrow\omega(I) = \bigcup_{n<\omega} T\uparrow n(I)$$

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

The following well-known fact holds.

**3.9.** LEMMA. *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 Sections 4 and 5 we shall also use largest fixpoints and *downward powers* of monotonic operators. We put for a monotonic operator $T$

$$T\downarrow 0(I) = I, \qquad T\downarrow(n+1)(I) = T(T\downarrow n(I)), \qquad T\downarrow\omega(I) = \bigcup_{n<\omega} T\downarrow n(I).$$

Downward powers generalize in a straightforward way to *transfinite downward powers* $T\downarrow\alpha(I)$ where $\alpha$ 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 the following.

**3.10.** THEOREM. *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

$I_0 \supseteq I_1 \supseteq \cdots$, it holds that

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

We have the following well-known lemma.

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

We denote the smallest ordinal $\alpha$ 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. Note that Herbrand interpretations of $L$ with the usual set-theoretic operations from a complete lattice so when studying this operator we can apply the results of the previous section.

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

(i) *$T_P$ is finitary,*

(ii) *$T_P$ is monotonic.*

PROOF. (i) Consider an infinite sequence $I_0 \subseteq I_1 \subseteq \cdots$ of Herbrand interpretations and suppose that $A \in T_P(\bigcup_{n=0}^{\infty} I_n)$. Then, for some atoms $B_1, \ldots B_k$, the clause $A \leftarrow B_1, \ldots, B_k$ 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 $B_1, \ldots, B_k$, $I_n \models B_1 \wedge \cdots \wedge B_k$. So $A \in T_P(I_n)$.

(ii) Immediate by definition. □

As an immediate consequence of the above lemma we have the following theorem.

**3.13.** THEOREM (Characterization Theorem) (Van Emden and Kowalski [31]). *Let $P$ be a program. Then $P$ has a Herbrand model $M_P$ which satisfies the following properties:*

(i) *$M_P$ is the least Herbrand model 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 Lemma 3.7, Theorem 3.8 and Lemma 3.9. □

By the *success set* of a program $P$ we denote the set of all ground atoms $A$ such that $P \cup \{\leftarrow A\}$ has an SLD-refutation.

**3.14.** COROLLARY. *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 [46]. The proof is due to APT and VAN EMDEN [4].

**3.15.** THEOREM (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.

**3.16.** LEMMA. (Substitution Lemma). *Let $P$ be a program, $N$ a goal and $\theta$ 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 SLD-refutation of $P\cup\{N\theta\}$. By the Variant Corollary 2.9 we can assume that $\theta$ does not act on any of the variables appearing in the input clauses of this refutation. Let $N=\leftarrow A_1,\ldots,A_k$.

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

If $n>1$ then consider the first input clause $B_0\leftarrow B_1,\ldots,B_m$ of the refutation. For an mgu $\eta$ 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 $\theta$, $A_i\theta\eta\equiv B_0\theta\eta$, so $A_i$ and $B_0$ unify. For some mgu $\xi$ and a substitution $\gamma$ we have $\theta\eta=\xi\gamma$.

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

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

of length $n-1$. By the induction hypothesis there exists an SLD-refutation of

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

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

We now establish the converse of Corollary 3.14.

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

Consider now an SLD-derivation of $P\cup\{\leftarrow A\}$ with the first input clause being the one of which $A\leftarrow B_1,\ldots,B_n$ is a ground instance. Its first resolvent is a negative clause of which $\leftarrow B_1,\ldots,B_n$ 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 $N=\leftarrow A_1,\ldots,A_n$. $M_P$ is not a model of $P\cup\{N\}$, so $N$ is not true in $M_P$. Thus, for some substitution $\theta$, $\{A_1\theta,\ldots,A_n\theta\}\subseteq M_P$. By Lemma 3.17, for $i=1,\ldots,n$ 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 notion. Let $P$ be a program and $N=\leftarrow A_1,\ldots,A_n$ a goal. We say that $\theta$ is a *correct answer substitution* for $P\cup\{N\}$ if $\theta$ acts only on variables appearing in $N$ and $P\models(A_1\wedge\cdots\wedge A_n)\theta$ holds.

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

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

**3.18.** THEOREM (Clark [22]). *Consider a program $P$ and a goal $N$. For every correct answer substitution $\theta$ for $P\cup\{N\}$ there exists a computed answer substitution $\sigma$ for $P\cup\{N\}$ such that $N\sigma$ is more general than $N\theta$.*

We present here the proof due to Lloyd [64]. First we need the following strengthening of the Substitution Lemma.

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

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

**3.20.** LEMMA. *Let $P$ be a program and $N$ a goal. Suppose that $\theta$ 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 $x_1, \ldots, x_n$ be the variables of $N\theta$. Enrich the language of $P$ by adding new constants $a_1, \ldots, a_n$ and let $\gamma$ be the substitution $\{x_1/a_1, \ldots, x_n/a_n\}$. $P \cup \{N\theta\gamma\}$ is inconsistent, so by the Completeness Theorem 3.15 there exists an SLD-refutation of $P \cup \{N\theta\gamma\}$. By the Variant Corollary 2.9 we can assume that the variables $x_1, \ldots, x_n$ do not appear in the input 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 $i = 1, \ldots, n$ 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 $\theta_0, \ldots, \theta_n$ be its sequence of mgu's. By the Lifting Lemma 3.19 there exists an SLD-refutation of $P \cup \{N\}$ with a computed answer substitution $\sigma$ and a sequence of mgu's $\theta'_0, \ldots, \theta'_n$ such that $\theta'_0 \ldots \theta'_n$ is more general than $\theta\theta_0 \ldots \theta_n$.

Then $N\theta'_0 \ldots \theta'_n$ is more general than $N\theta\theta_0 \ldots \theta_n$. But the former goal equals $N\sigma$ whereas the latter equals $N\theta$. □

### *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 [4].

**3.21.** THEOREM (strong completeness of SLD-resolution) (Hill [46]). *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*, where $k \geqslant 1$, 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.

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

The next two lemmas generalize corresponding facts about refuted goals.

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

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

**3.24.** LEMMA. *If $A$ is in the least Herbrand 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.

**3.25.** THEOREM (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), $P \models A$ 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 have proved more.

**3.26.** THEOREM. *Let $P$ be a program and $N$ a good. 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 until 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 Section 4.

### *3.10. Procedural versus declarative interpretation*

In the last two sections 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, i.e., 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 subsection, the computational mechanism is supplemented by an incomplete (tree) search algorithm.

### *3.11. Bibliographic remarks*

The name *immediate consequence operator* was introduced in [22]. Gallier [39] presents a different proof of the completeness of the SLD-resolution based on the use of Gentzen systems and indicates how to extend it to obtain a proof of the strong completeness of the SLD-resolution. The strongest completeness result is that of Clark [22], which combines the claims of Theorems 3.18 and 3.21. Lloyd [64] provides a rigorous proof of this theorem.

## 4. Computability

### *4.1. Computability versus definability*

Once we have 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 $t_1, \ldots, t_n \in U_L$,

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

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 $t_1, \ldots, t_n \in U_L$,

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

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

**4.1.** THEOREM. *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 $t_1, \ldots, t_n \in U_L$*

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

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 on $U_L$ 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 Andréka and Németi [1]. Our presentation is based on [12].

**4.2.** THEOREM (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

$$\text{height}(a) = 0 \quad \text{for each constant } a,$$
$$\text{height}(f(t_1, \ldots, t_n)) = \max(\text{height}(t_1), \ldots, \text{height}(t_n)) + 1.$$

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

$$f(s_1, \ldots, s_k) < g(t_1, \ldots, t_m)$$
$$\text{iff } (\text{height}(f(s_1, \ldots, s_k)), f, s_1, \ldots s_k) \prec (\text{height}(g(t_1, \ldots t_m)), g, t_1, \ldots, t_m).$$

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 $\omega$:

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

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

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

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

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 $n$, $\widehat{n+1} = f_0(a, \ldots, a, \hat{n})$.

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

$$nat(a) \leftarrow,$$

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

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

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

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

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

(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

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

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 $f_1(b, \ldots, b)$, etc.

Thus adding clauses

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

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

we get a program *maximum* such that

$$\textit{maximum} \models \max(t, k)$$
iff $t$ is the $<$-maximal term of height $n$, 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. □

### 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 [92].

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

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

For a given predicate $R \subseteq N^n$, $\boldsymbol{K}_R$ stands for its characteristic function defined by

$$\boldsymbol{K}_R(\bar{a}) = \begin{cases} 0 \text{ iff } \bar{a} \in R, \\ 1 \text{ iff } \bar{a} \notin R. \end{cases}$$

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

(R1) the functions $\boldsymbol{P}_i^n$, $+$, $\times$ and $\boldsymbol{K}_<$ are recursive;

(R2) if $g, h_1, \ldots, h_k$ are recursive functions and $f$ is defined by

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

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, where $\mu b. \, R$ 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* (r.e.) 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 $S$ there is

some recursive function $f$ such that

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

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.

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

PROOF. By the Characterization Theorem 3.13 (iv) we have $A \in M_P$ iff, for some $k > 0$, relation $p$ and $t_1, \ldots, t_n \in U_P$, $A = p(t_1, \ldots, t_n)$ and $p(t_1, \ldots, t_n) \in T_P \uparrow k$. 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.

**4.4** THEOREM (Computability Theorem) (Andréka and Németi [1]). *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 given here incorporates the program *successor* which uses different relations than those used here. We proceed by induction on the construction of recursive functions.

*Re* (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.

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

$$p_f(x_1, \ldots, x_l, x_{l+1}) \leftarrow p_{h_1}(x_1, \ldots, x_l, y_1), \ldots, p_{h_k}(x_1, \ldots, x_l, y_k), p_g(y_1, \ldots, y_k, x_{l+1})$$

computes the graph of the function $f$ defined as in (R2).

*Re* (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 $r$:

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

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

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

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

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

PROOF. ($\Rightarrow$) Suppose that for some recursive predicate $S$, $\bar{a} \in R$ 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

$$p_R(x_1, \ldots, x_k) \leftarrow p_S(x_1, \ldots, x_k, y, \hat{0})$$

computes the predicate $R$ using relation $p_R$.

($\Rightarrow$) By Theorems 4.1 and 4.3. □

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

**4.6.** COROLLARY. *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 $M_P$.

**4.7.** COROLLARY. *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 $a$, $a \in R$ 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 subsection. Following [12] we call a program $P$ *determinate* if $T_P \uparrow \omega = T_P \downarrow \omega$.

**4.8.** THEOREM (Blair [12]). *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 subsection we study the downward closure ordinals of the operators $T_P$ for programs $P$.

We noted in subsection 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 $n \geqslant 1$ we have $T_P\downarrow n = \{q(a)\} \cup (p(f^k(a)) : k \geqslant n\}$, 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 (partial) 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:

$$\| a \| = \begin{cases} 0 & \text{if } a \text{ is a } <_R\text{-minimal element of } dom(R), \\ sup(\| b \| + 1 : b <_R a) & \text{otherwise,} \end{cases}$$

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

An ordinal $\alpha$ is called *recursive* if $\alpha = \| R \|$ for some well-founded ordering $R$ which is a recursive predicate. The least nonrecursive ordinal is denoted by $\omega_1^{ck}$ ($\omega_1$ of Church and Kleene).

The following theorem characterizes the ordinals $\| T_P\downarrow \|$.

**4.9.** THEOREM (Blair[11]). (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 $\beta$ 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 the set of 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 $q(t_1, \ldots, t_n)$.

Let $P_1$ be the program $P$ from the beginning of this subsection augmented by the clause

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

Then $T_{P_1}\downarrow\omega = [q]$ and $T_{P_1}\downarrow\alpha = 0$ 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 $\alpha \geqslant \omega$,

$$T_{P_2}\downarrow\alpha \cap [r] = R_r, \quad \text{where } R_r = \{r(s,t): (s,t) \in R\}.$$

Let $P_3$ be the program

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

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 $\gamma$,

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

Thus, for $\gamma < \beta$,

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

Also

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

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 recursion theory and are beyond the scope of this paper. □

### *4.6. Bibliographic remarks*

There is considerable confusion concerning the actual formulation and origin of the results of the first part of this section. The statement that logic programming has a full power of recursion theory is usually attributed to Tärnlund [98] who showed that Turing machines can be simulated using logic programs. However, in his proof additional function symbols are used and the paper of Andréka and Németi [1] actually appeared earlier as a technical report.

A syntactically stronger form of the Computability Theorem 4.4 in the case when $L$ has exactly one, unary function symbol and exactly one constant was proved in [86]. For such $L$ the Computability Theorem 4.4 is implicitly contained in [94]. Related

results were proved in [47, 56, 89, 95]. The last paper discusses all these results in detail. Börger [14] discusses connections between logic programming and computational complexity of various classes of formulas. Fitting [37] studies in detail computability by means of logic programs on domains other than the Herbrand base, in particular integers, words and trees.

That $T_P$ does not need to be downward continuous was originally observed by Andréka and Németi, and Clark. The class of determinate programs is extensively studied in [5], where they are called functional programs.

## 5. Negative information

### 5.1. *Nonmonotonic 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 $\varphi$, $P \vdash \varphi$ implies $P \models \varphi$, where $P \vdash \varphi$ denotes that $\varphi$ can be proved from a program $P$. And we call "$\vdash$" *weakly sound* if $P \vdash \varphi$ implies consistency of $P \cup \{\varphi\}$. Now, putting (see subsection 2.5) $P \vdash_{\mathrm{SLD}} \exists x_1 \ldots \exists x_s (A_1 \wedge \cdots \wedge A_k)$ iff there exists an SLD-refutation of $P \cup \{\leftarrow A_1, \ldots, A_k\}$, we see that "$\vdash_{\mathrm{SLD}}$" is sound by virtue of the Soundness Theorem 3.2.

We call a reasoning method "$\vdash$" *effective* if for any program $P$ the set $\{\varphi: P \vdash \varphi\}$ is recursively enumerable. Now, "$\vdash_{\mathrm{SLD}}$" 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 $P'$,

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

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

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

**5.2.** EXAMPLE Consider

$$P = \{element(fire)\leftarrow,\ element(air)\leftarrow,\ element(water)\leftarrow,\ element(earth)\leftarrow,\ stuff(mud)\leftarrow\}.$$

Then we naturally expect that $\neg element(mud)$, $\neg stuff(fire)$ and similarly with other elements.

By Lemma 5.1 any such extension of SLD-resolution leads to a nonmonotonic reasoning.

### 5.2. *Closed world assumption*

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

$$\frac{A \text{ cannot be proved from } P}{\neg A}$$

where $A$ is a ground atom. This rule is usually called the *closed world assumption* (CWA). It was first considered in [83]. The notion of provability referred to in the hypothesis is that in 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

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

We have the following lemma.

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

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 3.17. □

As an immediate consequence we get this theorem.

**5.4.** THEOREM (Reiter [83]). *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.

**5.5.** THEOREM. *Assume that $L$ is as in Section 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 well-known theorem, $B_P - M_P$, 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 some 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. Thus all branches of a finitely failed SLD-tree are failed SLD-derivations. 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:

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

introduced in [21] and called the *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 the 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 Theorem 5.4. Thus by Lemma 5.1 negation as failure is a nonmonotonic 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 the negation as failure rule we can trivially deduce $\neg element(mud)$ and $\neg stuff(fire)$ from the program $P$ given in Example 5.2.

### 5.4. *Characterizations of finite failure*

We now provide two characterizations of finite failure, due to Apt and van Emden [4], and Lassez and Maher [61]. We follow here the presentation of [64].

First we introduce the concept of a fair SLD-derivation due to Lassez and Maher [61]. 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 instance of the original version.) For example, the second derivation given in Subsection 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. A selection rule $R$ is *fair* if all SLD-derivations via $R$ are fair. Thus an SLD-tree is fair if it is via a fair selection rule.

**5.6.** 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.*

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

**5.7.** LEMMA. *Consider a program $P$, a negative clause $N$ and a substitution $\theta$. 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 $k$. □

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

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)⇒(b): We prove a stronger claim, namely the following lemma.

**5.9.** LEMMA. *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 $B \leftarrow B_1,\ldots,B_n$ in $P$, $A \equiv B\theta$ and $\{B_1\theta,\ldots,B_n\theta\} \subseteq T_P{\downarrow}(k-1)$ for some substitution $\theta$. Thus, for some mgu $\gamma$, $A\gamma \equiv B\gamma$ and $\theta = \gamma\sigma$ for some $\sigma$. Hence $\leftarrow(B_1,\ldots,B_n)\gamma$ is the root of a finitely failed SLD-tree of depth $\leqslant k-1$. By Lemma 5.7 so is $\leftarrow(B_1,\ldots,B_n)\theta$. Now using Lemma 5.8 with each $F_i$ being a single ground atom we get that, for some $i$, $1 \leqslant i \leqslant n$, the goal $\leftarrow B_i\theta$ 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 Theorem 5.6(b) implies (c) we need the following lemma.

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

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

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

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

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

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

so

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

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

PROOF OF THEOREM 5.6 (*continued*). (b)⇒(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 $A\in T_P\downarrow\omega\subseteq T_P\downarrow\omega$. Thus it is a finitely failed SLD-tree.

(c)⇒(a): Obvious, as for every goal $N$ there is a fair SLD-tree with $N$ as root. □

Equivalence between (a) and (b) is due to Apt and van Emden [4], and between (a) and (c) due to Lassez and Maher [61]. 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 negative information from a logic program is that of using the concept of a completion of a program due to Clark [21].

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

*Step* 1: *Remove terms.* Transform each clause $p(t_1,\ldots, t_n)\leftarrow B_1,\ldots, B_m$ of $P$ into

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

*Step* 2: *Introduce existential quantifiers.* Transform each formula $p(x_1,\ldots, x_n)\leftarrow F$ obtained in the previous step into

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

where $y_1,\ldots, y_d$ are the variables of the original clause.

*Step* 3: *Group similar formulas.* Let

$$p(x_1,\ldots,x_n)\leftarrow F_1,$$
$$\ldots$$
$$p(x_1,\ldots,x_n)\leftarrow F_k$$

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

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

If $F_1\vee\cdots\vee F_k$ 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

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

*Step* 5: *Introduce universal quantifiers.* Replace each formula $p(x_1,\ldots,x_n)\leftarrow F$ by

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

*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 "$\rightarrow$".

**5.11.** EXAMPLE. (i) Reconsider the program $P$ from Example 5.2. Then

$$IFF(P)=\{\forall x(element(x)\leftrightarrow x=fire\ \vee\ x=air\ \vee\ x=water\ \vee\ x=earth),$$
$$\forall x(stuff(x)\leftrightarrow x=mud)\}.$$

Note that both $IFF(P)\models\neg stuff(fire)$ and $IFF(P)\models\neg element(mud)$ provided we interpret "$=$" as identity.

(ii) Consider the program

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

Then

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

It is easy to see 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 negative information can be inferred using the $IFF$-definition

provided we interpret the relation symbol "=" properly. The problem of the proper interpretation of "=" is more subtle than it appears. As a first step 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 $\sigma$ over $I$

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

However, this does not yet solve the problem because, even though *mud* and *earth* 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) $f(x_1,\ldots,x_n)=f(y_1,\ldots,y_n)\rightarrow x_1=y_1\wedge\cdots\wedge x_n=y_n$ for each $n$-ary function $f$,

(2) $f(x_1,\ldots,x_n)\neq g(y_1,\ldots,y_m)$ for each $n$-ary function $f$ and $m$-ary function $g$ such that $f\neq g$,

(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 2.3, we identify constants with 0-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.

The resulting interpretation of "=" turns out to be sufficient for our purposes. Observe the striking similarity between the free equality axioms and steps (1), (2) and (5) of the unification algorithm used in the proof of the Unification Theorem 2.3. We shall exploit it in Subsection 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 attention here to Herbrand models. This is the content of a proposition we prove at the end of this subsection.

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 [48], 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 Subsection 3.1 with the exception that the clause 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 $r(a_1,\ldots,a_n)$ where $r$ is an $n$-ary relation of $L$ and $a_1,\ldots,a_n$ 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 $A = p(t_1,\ldots, t_n)$ be an atom and let $\sigma$ be a state over $I$. Then we denote by $A\sigma$ the generalized atom $p(\sigma(t_1),\ldots, \sigma(t_n))$.

Let now $J$ be a pre-interpretation and let $I$ be an interpretation based on $J$. For a program $P$ and a generalized atom $D$, we put

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

Thus $T_P^J$ maps interpretations based on $J$ to interpretations based on $J$. The operator $T_P^J$ enjoys several properties similar to those of $T_P$. We list them in the following lemma omitting the proofs analogous to those of Lemma 3.7 and Lemma 3.12.

**5.12.** LEMMA. *Let $P$ be a program and $J$ a pre-interpretation. Then*

(i) *$T_P^J$ is finitary.*

(ii) *$T_P^J$ is monotonic.*

(iii) *For an interpretation $I$ based on $J$, $I$ 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.

**5.13.** LEMMA. *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. □

**5.14.** COROLLARY. *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.** THEOREM. *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.

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

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

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

$p(x_1, \ldots, x_n)\sigma \in T_P^J(I)$

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

iff $I \models_\sigma \exists y_1 \ldots \exists y_d((x_1 = t_1) \wedge \cdots \wedge (x_n = t_n) \wedge B_1 \wedge \cdots \wedge B_m)$ for some clause $p(t_1, \ldots, t_n) \leftarrow B_1, \ldots, B_m$ of $P$ with $y_1, \ldots, y_d$ being all its variables

iff $I \models_\sigma F$. □

PROOF OF THEOREM 5.15. We have

$I$ is a model of $ONLY\text{-}IF(P)$

iff for every formula $\forall x_1 \ldots \forall x_n(p(x_1, \ldots, x_n) \rightarrow F)$ in $ONLY\text{-}IF(P)$ and every state $\sigma$ over $I$ $p(x_1, \ldots, x_n)\sigma \in I$ implies $I \models_\sigma F$

iff (by Lemma 5.16) for every relation $p$ of $P$ and state $\sigma$ over $I$ $p(x_1, \ldots, x_n)\sigma \in I$ implies $p(x_1, \ldots, x_n)\sigma \in T_P^J(I)$

iff $T_P^J(I) \supseteq I$. □

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

**5.17.** THEOREM. *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 $P$.

**5.18.** THEOREM. (Apt and Van Emden [4]). *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.

**5.19.** THEOREM. *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 beginning of Subsection 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 Subsection 5.10 we provide a characterization of the finite failure which provides a more direct proof of the above theorem.

### 5.7. *Soundness of the negation as failure rule*

Recall that completion of a program was introduced in order to infer negative information from a program. We now relate it to the previously studied way of deducing negative information—that by means of the negation as failure rule. To this purpose we first investigate models of the free equality axioms (Subsection 5.5). Assume a program $P$ and denote these axioms by Eq. As Eq does not refer to relations, it makes sense to say that a pre-interpretation $J$ is a model of Eq. Similarly, it is meaningful to talk about states over a pre-interpretation. For each ground term $t$ denote its value in the domain of $J$ by $t_J$. We write $J\models_\sigma s=t$ when $\sigma(s)$ equals $\sigma(t)$.

**5.20.** LEMMA. *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 all its 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.

**5.21.** LEMMA (Clark [21]). (a) *If the set $\{s_1=t_1,\ldots,s_n=t_n\}$ has a unifier then for some of its mgu $\{x_1/u_1,\ldots,x_k/u_k\}$*

$$\text{Eq}\models s_1=t_1\wedge\cdots\wedge s_n=t_n\rightarrow x_1=u_1\wedge\cdots\wedge x_k=u_k.$$

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

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

PROOF. Modify the Unification Algorithm given in the proof of the Unification

Theorem 2.3 as follows. First display each set $\{s_1=t_1,\ldots,s_n=t_n\}$ of equations as a formula $s_1=t_1\wedge\cdots\wedge s_n=t_n$. 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 $\psi$ is obtained from $\varphi$ by applying one of the steps of the algorithm then $\mathrm{Eq}\models\varphi\to\psi$. Indeed, for any $x$ and $t$, $\varphi_0\wedge x=t\wedge\varphi_1\to(\varphi_0\wedge\varphi_1)\{x/t\}$ is a valid formula. Other cases are immediate.

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

Given a pre-interpretation $J$ and a state $\sigma$ over $J$, call a substitution $\theta$ *invariant over a state* $\sigma$ if, for all $x$, $\sigma(x)=\sigma(x\theta)$.

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

$$J\models_\sigma s_1=t_1\wedge\cdots\wedge s_n=t_n,$$

*then, for some mgu $\theta$ of $\{s_1=t_1,\ldots,s_n=t_n\}$ invariant over $\sigma$,*

$$J\models(s_1=t_1\wedge\cdots\wedge s_n=t_n)\theta.$$

Call now an interpretation $I$ based on $J$ *good* if, for all conjunctions of atoms $F$, $I\models_\sigma F$ for some state $\sigma$ implies $I\models F\theta$ for some substitution $\theta$. Obviously not all interpretations are good. But those of interest to us are. First we need the following two lemmas.

**5.23.** LEMMA. *Let $J$ be a pre-interpretation which is a model of* Eq. *Let $I$ be based on $J$. Suppose that $I$ is good. Then $T_P^J(I)$ is good, as well.*

PROOF. Consider a sequence $A_1,\ldots,A_k$ of atoms. The operator $T_P^J$ does not depend on the choice of the names of variables in $P$. Thus we can assume that each of the variables of $P$ appears in at most one clause of $P$ and none of them appears in $A_1,\ldots,A_k$. Suppose now that $T_P^J(I)\models_\sigma A_1\wedge\cdots\wedge A_k$ for some state $\sigma$. 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 $P$ and a state $\tau_i$ such that $I\models_{\tau_i}B_1^i\wedge\cdots\wedge B_{m_i}^i$ and $A_i\sigma=B_i\tau_i$. Define now a state $\tau$ by

$$\tau(x)=\begin{cases}\tau_i(x) & \text{if } x \text{ appears in } B_i\leftarrow B_1^i,\ldots,B_{m_i}^i,\\ \sigma(x) & \text{otherwise.}\end{cases}$$

Then

$$I\models_\tau \bigwedge_{\substack{i=1,\ldots,k\\ j=1,\ldots,m_i}} B_j^i \tag{5.1}$$

and, for each $i=1,\ldots,k$,

$$A_i\tau=B_i\tau. \tag{5.2}$$

By Corollary 5.22 and (5.2) there exists a substitution $\theta$ invariant over $\tau$ such that, for each $i=1,\ldots,k$,

$$A_i\theta\equiv B_i\theta. \tag{5.3}$$

By the definition of invariance and (5.1)

$$I \models_\tau \bigwedge_{\substack{i=1,\ldots,k \\ j=1,\ldots,m_i}} B_j^i \theta$$

But $I$ is good, so for some substitution $\gamma$

$$I \models_\tau \bigwedge_{\substack{i=1,\ldots,k \\ j=1,\ldots,m_i}} B_j^i \theta\gamma.$$

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

Thus by the definition of $T_P^J$, for each $i=1,\ldots,k$, $B_i\theta\gamma \in T_P^J(I)$, i.e. by (5.3) $T_P^J(I) \models (A_1 \wedge \cdots \wedge A_k)\theta\gamma$. This concludes the proof. □

**5.24.** LEMMA. *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 $\sigma$ over $I$, $A \equiv B\sigma$ and $I \models_\sigma A_1 \wedge \cdots \wedge A_n$ where $B \leftarrow A_1,\ldots,A_n$ is a clause from $P$. Thus $\sigma$ when restricted to the variables of $B$ is a ground substitution, say $\eta$. We thus have $I \models_\sigma (A_1 \wedge \cdots \wedge A_n)\eta$. But $I$ is good, so for some substitution $\theta$, $I \models (A_1 \wedge \cdots \wedge A_n)\eta\theta$. Thus $B_p \cap I \models (A_1 \wedge \cdots \wedge A_n)\eta\theta$. 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 of $T_p^J$ can already be inferred by means of $T_p$, provided $I$ is good.

This brings us to the following important consequences of Lemmas 5.23 and 5.24 which will be also used in Section 6.

**5.25.** COROLLARY. *Let $J$ be a pre-interpretation which is a model of* Eq.

(i) *For every $n \geqslant 0$, $T_P^J \downarrow n$ is good.*

(ii) *For every $n \geqslant 0$, $B_p \cap T_P^J \downarrow n = T_p \downarrow n$.*

(iii) $B_p \cap T_p^T \downarrow \omega = T_p \downarrow \omega$.

PROOF. We have $T_P^J \downarrow 0 = B_J$. But $B_P \subseteq B_J$, so for all conjunctions of atoms $F$ and all substitutions $\theta$, $B_J \models F\theta$. Thus $T_P^J \downarrow 0$ is good and, by induction using Lemma 5.23, for every $n \geqslant 0$, $T_P^J \downarrow n$ is good.

(ii) We proceed 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

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

This implies the claim for $n+1$.

(iii) Immediate, by (ii). □

Finally, we prove the following lemma which will also be needed in Section 6.

**5.26.** LEMMA. *Let $P$ be a program and $I$ a model of comp(P). Then $B_P \cap I \subseteq T_P \downarrow \omega$.*

PROOF. $I$ is based on some pre-interpretation $J$. $I$ is a model of $IFF(P)$, so, by Theorem 5.17, $T_P^J(I) = I$. Thus, by Lemma 3.11, $I \subseteq T_P^J \downarrow \omega$. $J$ is a model of Eq, so by Corollary 5.25(iii) and the above inclusion, the claim follows. □

We can now relate the completion of a program and negation as failure rule.

**5.27.** THEOREM (soundness of the negation as failure rule) (Clark [21]). *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 a model of $comp(P)$ and suppose that $A$ is in the finite failure set of $P$. Then, by Theorem 5.6, $A \notin T_P \downarrow \omega$, so by Lemma 5.26 $A \notin B_P \cap I$, i.e. $I \models \neg A$. □

### 5.8. *Completeness of the negation as failure rule*

We now prove the converse of the above theorem. We follow here essentially the presentation of Lloyd [64] based on a proof due to Wolfram, Maher and Lassez [102]. We first show how to construct models of the free equality axioms.

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

$$\theta, \eta \in \mathscr{C} \Rightarrow \text{there exists a } \gamma \in \mathscr{C} \text{ such that } \theta \leqslant \gamma \text{ and } \eta \leqslant \gamma.$$

Here $\theta \leqslant \gamma$ means that $\theta$ is more general than $\gamma$. Suppose now that $\mathscr{C}$ is a set of substitutions. For two terms $s, t$, put

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

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

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

Let $[s]$ stand for the equivalence class of term $s$ w.r.t. $\sim_{\mathscr{C}}$. 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 directedness of $\mathscr{C}$, for some $\theta \in \mathscr{C}$,

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

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 $\sim_{\mathscr{C}}$ is indeed a congruence. This means that $\sim_{\mathscr{C}}$ induces a pre-interpretation of $L$. That this interpretation is indeed a model of Eq is easy to see, as nonunifiable terms have necessarily different equivalence classes w.r.t. $\sim_{\mathscr{C}}$. □

The essence of the proof of the completeness theorem lies in the following lemma.

**5.29.** LEMMA. *Consider a program $P$ and a goal $N$. Suppose there is a nonfailed fair SLD-derivation with $N$ as the initial goal. Then $comp(P) \cup \{\neg N\}$ is consistent.*

PROOF. Let $\Phi = N_0, N_1, \ldots$ with $N_0 = N$ and with the sequence of substitutions $\theta_0, \theta_1, \ldots$ be the *SLD*-derivation in question and let $N = \leftarrow A_1, \ldots, A_s$. Then $\neg N \equiv \exists(A_1 \wedge \cdots \wedge A_s)$. We use this derivation to construct a model of $comp(P) \cup \{\exists(A_1 \wedge \cdots \wedge A_s)\}$. Let $\mathscr{C} = \{\theta_0 \ldots \theta_i \colon i \geqslant 0\}$. Note that $\mathscr{C}$ is directed. By the last lemma the pre-interpretation $J$ induced by $\sim_{\mathscr{C}}$ is a model of Eq. Let $[s]$ denote the equivalence class under $\sim_{\mathscr{C}}$ of a term $s$.

We now construct an interpretation $I$ based on $J$ by putting

$$I = \{p([t_1], \ldots, [t_n]) \colon p(t_1, \ldots, t_n) \text{ appears in a goal from } \Phi\}.$$

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

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

In subsection 2.7 we assumed that each mgu $\theta_l$ is idempotent and relevant. Thus by Lemma 2.7 for any $l, m$ such that $m > l$, $\theta_l$ does not act on the variables from $N_m$ or $N_m\theta_m$. Fix $k$, $1 \leqslant k \leqslant n$. Thus, since $t_k$ appears in $N_i$,

$$t_k\theta_l \equiv t_k \quad \text{for } l < i, \tag{5.4}$$

and, since $t_k\theta_i \ldots \theta_j$ appears in $N_j\theta_j$

$$t_k\theta_i \ldots \theta_j\theta_l \equiv t_k\theta_i \ldots \theta_j \quad \text{for } l < j. \tag{5.5}$$

Thus

$$\begin{aligned} t_k\theta_i \ldots \theta_j\theta_0 \ldots \theta_{j-1}\theta_j &\equiv t_k\theta_i \ldots \theta_j\theta_j && \text{(by (5.5) applied } j \text{ times)} \\ &\equiv t_k\theta_i \ldots \theta_j && \text{(by idempotence of } \theta_j) \\ &\equiv t_k\theta_0 \ldots \theta_j. && \text{(by (5.4) applied } i \text{ times)} \end{aligned} \tag{5.6}$$

Hence for all $k$, $1 \leqslant k \leqslant n$

$$\begin{aligned} [t_k] &= [t_k\theta_i \ldots \theta_j] && \text{(by (5.6))} \\ &= [s_k\theta_j]. && \text{(by definition of } s_k) \end{aligned}$$

But by the definition of $I$ we have $p([s_1\theta_j], \ldots, [s_n\theta_j]) \in T_P^J(I)$, so $p([t_1], \ldots, [t_n]) \in T_P^J(I)$, 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 $\exists(A_1 \wedge \cdots \wedge A_s)$, and a fortiori so is its extension. □

We are now in position to prove the desired theorem. It is formulated in a slightly more general form which will be needed in Section 6.

**5.30.** THEOREM (completeness of the negation as failure rule) (Jaffar, Lassez and Lloyd [48]). *Let $P$ be a program. If, for a goal $N$, $comp(P) \models N$, then $P \cup \{N\}$ has a finitely failed SLD-tree.*

PROOF. Assume there is a nonfailed fair SLD-derivation with $N$ as the initial goal. By the last lemma, $comp(P) \cup \{\neg N\}$ is consistent. Thus by contraposition, $comp(P) \models N$ implies that every fair SLD-tree with $N$ as root is finitely failed. Thus $P \cup \{N\}$ has a finitely failed SLD-tree. □

It is perhaps useful to indicate here that, using Lemma 5.29, an alternative proof of the implication (b)⇒(c) in Theorem 5.6 can be given without the use of Lemma 5.10. Indeed, assume there is a nonfailed fair SLD-derivation with $\leftarrow A$ as the initial goal. Then by Lemma 5.29 $comp(P) \cup \{A\}$ has a model. By Lemma 5.26 this implies that $A \in T_P \downarrow \omega$. Thus, by contraposition, $A \notin T_P \downarrow \omega$ implies that every fair SLD-tree with $\leftarrow A$ as root is finitely failed.

### 5.9. *Equality axioms versus identity*

Clark's [21] original definition of free equality additionally included the following usual equality axioms:

(1) $x = x$,

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

(3) $x_1 = y_1 \wedge \cdots \wedge x_n = y_n \rightarrow (p(x_1, \ldots, x_n) \rightarrow p(y_1, \ldots, y_n))$ 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., [72, p. 80]) shows.

**5.31.** THEOREM. *Let $S$ be a set of formulas in a first-order language $L$ including $=$. Then for every formula $\varphi$*

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

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

PROOF. (⇒): 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 to (i.e., satisfies the same formulas of) 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.

(⇐): When $=$ is interpreted as identity, all axioms of EQ became valid. □

### 5.10. *Summary*

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

**5.32.** THEOREM (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) $comp(P) \models \neg A$.

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 3.25, 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 3.25, $A$ 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 $gfp(T_P) \neq T_P \downarrow \omega$.

For any such program $P$ and a ground atom $A \in T_P \downarrow \omega - gfp(T_P)$ by the above theorem and Theorem 5.18(iii), $comp(P) \cup \{A\}$ has a model but it has no Herbrand model. This yields a more direct proof of Theorem 5.19.

Clause (d) of the Finite Failure Theorem suggests another possibility of inferring negation. Consider the following rule implicitly studied in [4].

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

Call this rule the *Herbrand rule*. Then the results of this section can be summarized by Fig. 3 from [64, p. 86] assessing the content of Lemma 5.3, Theorem 5.18(iii) and Theorem 5.6.

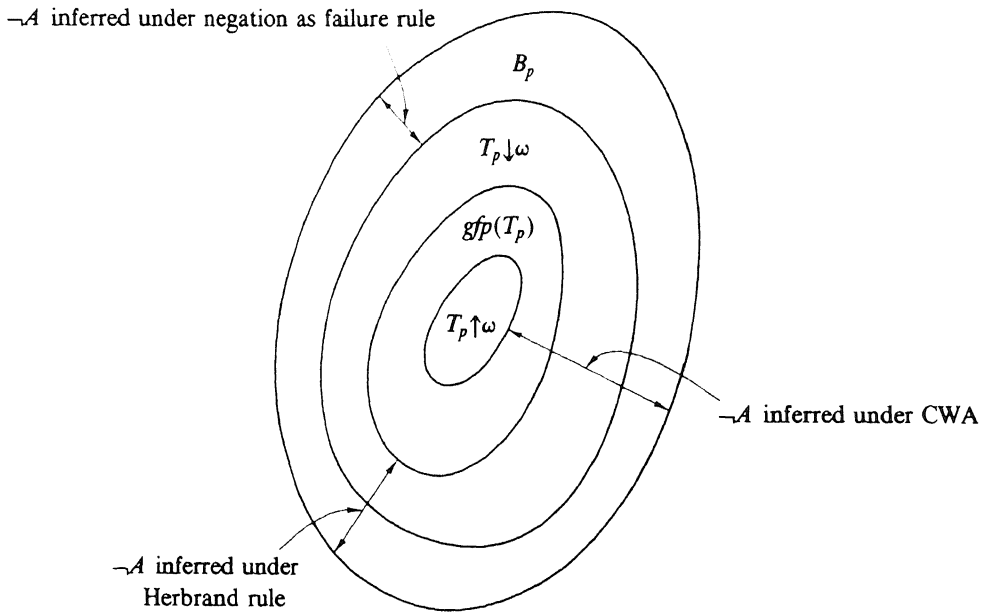


Fig. 3.

### 5.11. *Bibliographic remarks*

Theorem 5.17 is a straightforward generalization due to Jaffar, Lassez and Lloyd [48] of a special case (Theorem 5.18(a)) proved in [4]. The notion of a finite failure set was introduced in [4].

Lemma 5.20 appears as an exercise in [64, p. 88]. Proofs of Lemma 5.21 and Theorem 5.26 seem to be new. Lemma 5.21 was generalized by Kunen [59] who proved that that Eq is a complete axiomatization for the fragment $L(=)$ of $L$ containing $=$ as the only relation symbol.

Jaffar and Stuckey [49] proved that every program is semantically equivalent to a program $P$ for which $gfp(T_P) = T_P{\downarrow}\omega$. Maher [69] provided a partial characterization of programs $P$ for which $gfp(T_P) = T_P{\downarrow}\omega$.

## 6. General goals

### 6.1. *SLDNF$^-$-resolution*

When trying to extend the results of Sections 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 section. Strictly speaking this rule was defined only for ground atoms, but it can be extended in an obvious way to the nonground case.

This leads us to an extension of the SLD-resolution called SLDNF$^-$-resolution (SLD-resolution with negation as failure rule) introduced in [21]. We added the superscript "$-$" to indicate that it is used here only with nongeneral programs. Formally, we first introduce the notion of a resolvent of a general goal. Let $P$ be a program and $G = \leftarrow L_1, \ldots, L_n$ a general goal. We distinguish two cases. Fix $i$, $1 \leqslant i \leqslant n$.

(a) *Literal $L_i$ is positive.* Suppose that $C = A \leftarrow B_1, \ldots, B_k$ is a clause from $P$. If $L_i$ and $A$ unify with an mgu $\theta$ then

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

is a *resolvent of $G$ and $C$ with the mgu $\theta$.*

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

$$\leftarrow L_1, \ldots, L_{i-1}, L_{i+1}, \ldots, L_n$$

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 $G_0, G_1, \ldots$ of general goals where $G_0 = G$, together with

a sequence $C_0, C_1, \ldots$ of variants of clauses from $P$ and a sequence $\theta_0, \theta_1, \ldots$ of substitution such that, for all $i = 0, 1, \ldots$,

- 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 $\theta_i$;
- if the selected literal in $G_i$ is negative then $G_{i+1}$ is a resolvent of $G_i$, $C_i$ is arbitrary and $\theta_i$ is the empty substitution;
- $C_i$ does not have a variable in common with $G_0, C_0, \ldots, C_{i-1}$.

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 [21] 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 [21].

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

To prove it, we need the following mild generalization of Theorem 5.27, essentially due to Clark [21].

**6.2.** LEMMA. *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 an $n_0 \geqslant 1$ such that, for every ground substitution $\theta$, $P \cup \{A\theta\}$ has a finitely failed SLD-tree of depth $\leqslant n_0$. By Lemma 5.9, for every ground substitution $\theta$, $A\theta \notin T_P \downarrow n_0$. Suppose now that for some interpretation $I$ based on a pre-interpretation $J$, $I \models comp(P)$, and moreover, for some state $\sigma$, $I \models_\sigma A$. 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 Corollary 5.25(i), $T_P^J \downarrow n_0$ is good, so for some ground substitution $\theta$, $T_P^J \downarrow n_0 \models A\theta$. Now by Corollary 5.25(ii), $A\theta \in T_P^J \downarrow n_0$. This contradicts the former conclusion. □

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

PROOF OF THEOREM 6.1. Let $A_1, \ldots, A_l$ be the sequence of positive literals of $G$ and $\neg B_1, \ldots, \neg B_m$ 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 $P \cup \{\leftarrow A_1, \ldots, A_l\}$ 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,

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

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

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

Also, by Lemma 6.2, for $i=1, \ldots, m$,

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

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

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

which concludes the proof. □

**6.3.** COROLLARY. *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.

**6.4.** EXAMPLE. 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 nonground negative literals. Notice for instance that in the above example $P \cup \{\leftarrow p(b)\}$ 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 $G_0, \ldots, G_k$ with $G_0 = G$, where $G_k$ contains only nonground 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, does help. Unfortunately, such hopes are vain.

**6.5.** EXAMPLE. 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 $G^+$ (respectively $G^-$) stands for the set of relations occurring in positive (respectively 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 [19], and by Apt (unpublished).

**6.6.** THEOREM (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. [92]).

**6.7.** THEOREM (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.

**6.8.** LEMMA. *Let $P$ be a program. There exists a model $N_P$ of $comp(P)$ such that $B_P \cap N_P = T_P{\downarrow}\omega$.*

PROOF. Let $\{A_1, \ldots, A_n\}$ be a finite set of $T_P{\downarrow}\omega$. By Theorem 5.6, for $i = 1, \ldots, n$, $A_i$ is not in the failure set of $P$. Thus, by Lemma 5.8, $P \cup \{\leftarrow A_1, \ldots, A_n\}$ does not have a finitely failed SLD-tree. Now, by the completeness of the negation as failure rule (Theorem 5.30), there is a model of $comp(P) \cup \{A_1, \ldots, A_n\}$. Thus, by the Compactness Theorem 6.7, $comp(P) \cup T_P{\downarrow}\omega$ has a model, say $N_P$. We have $B_P \cap N_P \supseteq T_P{\downarrow}\omega$. Moreover, we have by virtue of Lemma 5.26, $B_P \cap N_P \subseteq T_P{\downarrow}\omega$. □

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

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

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.

**6.9.** LEMMA. *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. Let $B$ be the largest element of the original lattice $L$. The set $\{J: I \subseteq J \subseteq B\}$ with the operations $\subseteq$, $\cup$ and $\cap$ from $L$ forms a complete lattice with least element $I$. By assumption on $T$ and $I$, $T$ is an operator on this lattice and the claim follows by Lemma 3.9. □

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.

**6.10.** LEMMA. *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 $n \geqslant 1$,*

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

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

**6.11.** LEMMA. *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, by Lemma 5.12, $T^J_{P_1 \cup P_2}$ 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 Theorem 5.17, $T^J_{P_1 \cup P_2}\uparrow\omega(I)$ is a model of *comp*($P_1 \cup P_2$).

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

$$T^J_{P_1 \cup P_2}\uparrow n(I) = T^J_{P_2}\uparrow n(I) \quad \text{for } n \geqslant 0.$$

Hence

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

We can now prove the desired result.

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

$$P = P_0 \cup P^+ \cup P^-.$$

Note that $P^+ \cup P^- < P_0$. Consider now the interpretation $M_{P^+} \cup_{P^-}$. 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 $\sigma$,

$$M \models_\sigma A_1 \wedge \cdots \wedge A_l \wedge \neg B_1 \wedge \cdots \wedge \neg B_m$$

where $A_l, \ldots, A_l$ is the sequence of positive literals of $G$ and $\neg B_1, \ldots, \neg B_m$ is the sequence of negative literals of $G$. If $l = 0$ 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 $M_{P^+} \models_\sigma A_1 \wedge \cdots \wedge A_l$ and $N_{P^-} \models_\sigma \neg B_1 \wedge \cdots \wedge \neg B_m$. Thus $\sigma$, when restricted to the variables of $A_1 \wedge \cdots \wedge A_l$, is a ground substitution, say $\theta$. By Corollary 3.6 and the Characterization Theorem 3.13(i), $\theta$ is a correct answer substitution for $P^+ \cup \{\leftarrow A_1, \ldots, A_l\}$. Applying now Theorem 3.18 we obtain a computed answer substitution $\gamma$ for $P^+ \cup \{\leftarrow A_1, \ldots, A_l\}$ such that $(\leftarrow A_1, \ldots, A_l)\gamma$ is more general than $(\leftarrow A_1, \ldots, A_l)\theta$. But $(\leftarrow A_1, \ldots, A_l)\theta$ is ground, so in fact $\gamma$ is more general than $\theta$.

Fix some $i$, $1 \leqslant i \leqslant m$. By the assumption, $P \cup \{G\}$ does not flounder. Thus if $l = 0$ then $B_i$ is ground, so $B_i\sigma$ is a ground atom. If $l > 0$ then $B_i\gamma$ is ground, so $B_i\theta$ is ground and consequently $B_i\sigma$ is a ground atom, as well as $B_i\gamma \equiv B_i\sigma$. But $N_P \models_\sigma \neg B_1 \wedge \cdots \wedge \neg B_m$, so $B_i\sigma \in B_P - N_P$. By Lemma 6.8 we now have $B_i\sigma \notin T_P\downarrow\omega$. 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 Subsections 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.

**6.12.** THEOREM (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 $P \cup \{\leftarrow A\}$ 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

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

But by Corollary 4.7 for some program $P$, (the complement of) $M_P$ is not recursive. 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 [68]. 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 the 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.

**6.13.** THEOREM (Lloyd and Topor [68]). *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.

**6.14.** COROLLARY. *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 weakened a bit by requiring condition (b) to hold only for clauses whose heads contain a relation appearing in $DEP(G^+)$. 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 [21]. Example 6.5 seems to be new. The name *floundering* was introduced in [87] but the concept first appeared in [21]. Lemma 6.8 was indepen-

dently proved in [90]. Theorem 6.12 was independently, but somewhat earlier, proved in Börger [13].

The notion of *strictness* was first introduced in [3] for the case of general programs. The definition adopted here is inspired by Cavedon and Lloyd [19] 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 [68] for general programs. Similar, but less general notions were considered in [21, 87, 3].

## 7. Stratified programs

### 7.1. *Preliminaries*

General programs are difficult to analyze because of their irregular behaviour. In this section we study a subclass of general programs obtained by imposing on them some natural syntactic restrictions. Programs from this subclass enjoy several natural properties.

First, we generalize in an obvious way some of the concepts to the case of general programs. To start with, given a general program $P$ we introduce its *immediate consequence operator* $T_P$ by putting for a Herbrand interpretation $I$

$A \in T_P(I)$ iff for some literals $L_1, \ldots, L_n$
$A \leftarrow L_1, \ldots, L_n$ is in ground($P$)
and $I \models L_1 \wedge \cdots \wedge L_n$.

Next, given a general program $P$, we define its *completion* by using the same definition as the one given in Subsection 5.5 but now applied to general clauses instead of clauses. As before, *comp*($P$) stands for the completion of $P$.

Some of the results relating models of $P$ and *comp*($P$) to the operator $T_P$ remain valid and will be used in the sequel. We have the following lemma.

**7.1.** LEMMA. *Let $P$ be a general program and $I$ a Herbrand interpretation.*
(i) *$I$ is a model of $P$ iff $T_P(I) \subseteq I$.*
(ii) *$I$ is a model of comp($P$) iff $T_P(I) = I$.*
(iii) *$T_P$ is finitary.*

PROOF. (i) Analogous to the proof of Lemma 3.7.
(ii) Analogous to the proof of Theorem 5.18(i)—all corresponding lemmas remain valid.
(iii) Analogous to the proof of Lemma 3.12(ii). □

Lemma 7.1(iii) remains valid when $T_P$ is considered as an operator on a larger lattice formed by all subsets of $B_{P'}$, where $P \subseteq P'$, as then, for any $J \subseteq B_{P'}$, $T_P(J) = T_P(J \cap B_P)$. We shall use this observation in Subsection 7.4.

It is worthwhile to note that several other results do not generalize to the case of

general programs. For example, for the general program $P = \{A \leftarrow \neg B\}$, the associated operator $T_P$ is no longer monotonic, as $T_P(\emptyset) = \{A\}$ and $T_P(\{B\}) = \emptyset$. Thus Lemma 3.12(ii) does not generalize.

The same general program has two minimal models—$\{A\}$ and $\{B\}$ but none of them is the smallest. Thus Theorem 3.13 does not generalize. In turn, completion of the general program $A \leftarrow \neg A$ is inconsistent, so Theorem 5.18(ii) does not generalize either.

We thus see that it is not clear what intended meaning should be associated with a general program. None of the previously available possibilities—the one, semantic, based on $M_P$ and another, proof-theoretic, based on $comp(P)$, can be considered.

### 7.2. *Stratification*

To resolve these difficulties we introduce appropriate syntactic restrictions. Intuitively, we simply disallow a recursion "through negation". To express this idea more precisely we use the notion of a dependency graph introduced in Subsection 6.4. Given a general program $P$, consider its dependency graph $D_P$. We call an arc $(r, q)$ from $D_P$ *positive* (respectively *negative*) if there is a general clause in $P$ such that $r$ appears in its head and $q$ appears in a positive (respectively negative) literal of its body. Thus an arc may be both positive and negative.

Following [3, 99] we call a general program *stratified* if its dependency graph does not contain a cycle with a negative arc. An alternative definition of stratified programs is the following: Given a general program $P$ and a relation $r$, by a *definition of* $r$ (within $P$) we mean the set of all general clauses of $P$ in whose heads $r$ appears. We call a partition $P = P_1 \cup \cdots \cup P_n$ a *stratification* of $P$ if the following two conditions hold for $i = 1, \ldots, n$:

(i) if a relation appears in a positive literal of a general clause from $P_i$, then its definition is contained within $\bigcup_{j \leq i} P_j$;

(ii) if a relation appears in a negative literal of a general clause from $P_i$, then its definition is contained within $\bigcup_{j<i} P_j$.

We allow $P_1$ to be empty. A head of a general clause is viewed here as one of its positive literals. We call each $P_i$ *a stratum.*

Now, both definitions are equivalent as the following lemma shows.

**7.2.** LEMMA (Apt, Blair and Walker [3]). *A general program $P$ is stratified iff there exists a stratification of $P$.*

PROOF. If a general program admits some stratifaction then the definition of each relation symbol is contained in some stratum. Assign to each relation the index of the stratum within which it is defined. Then if $(p, q)$ is a positive arc in the dependency graph of $P$, then the index assigned to $q$ is smaller or equal than that assigned to $p$, and if $(p, q)$ is a negative arc, then the index assigned to $q$ is strictly smaller than that assigned to $p$. Thus there are no cycles in the dependency graph through a negative edge.

For the converse, decompose the dependency graph of $P$ into *strongly connected* components each of maximum cardinality, (i.e., such that any two nodes in a component are connected by a cycle). Then the relation "there is an edge from component $G$ to component $H$" is *well-founded*, since it is finite and contains no cycles. Thus for

some $n$ the numbers $1,\dots,n$ can be assigned to the components so that if there is an edge from $G$ to $H$, then the number assigned to $H$ is smaller than that assigned to $G$. Now, let $P_i$ be the subset of the general program $P$ consisting of the definitions of all relations which lie within a component with the number $i$.

We claim that $P = P_1 \cup \cdots \cup P_n$ is a stratification of $P$. Indeed, if $q$ is defined within some $P_i$ and refers to $r$, then $r$ lies in the same component or in a component with a smaller number. In other words, the definition of $r$ is contained in $P_j$ for some $j \leqslant i$. And if this reference is negative, then $r$ lies in a component with a smaller number because, by assumption, there is no cycle through a negative edge. Thus the definition of $r$ is then contained in $P_j$ for some $j < i$. □

This lemma allows us to use both definitions of a stratified general program interchangably.

**7.3.** EXAMPLE. (i) Consider the general program

$$P = \{p \leftarrow,\ q \leftarrow p,\ r,\ r \leftarrow \neg q\}.$$

Then $P$ is not stratified because the dependency graph of $P$ contains a cycle $(q, r)$, $(r, q)$ with a negative edge.

(ii) Consider the general program $P = \{p \leftarrow,\ q \leftarrow p,\ r \leftarrow \neg q\}$. Then $P$ is stratified by $\{p \leftarrow\} \cup \{q \leftarrow p\} \cup \{r \leftarrow \neg q\}$. Also $\{p \leftarrow,\ q \leftarrow p\} \cup \{r \leftarrow \neg q\}$ is a stratification of $P$.

Thus a general program can be stratified in more than one way.

Of course, it also makes sense to talk about stratification of programs (i.e., general programs "without negation"). By definition, every program is stratified but not every partition of it is a stratification. The following simple lemma relates the notion of stratification to the notation introduced in Subsection 6.4.

**7.4.** LEMMA. *A partition $P = P_1 \cup \cdots \cup P_n$ of a program $P$ is its stratification iff for every $i = 1,\dots,n$ we have $(\bigcup_{j<i} P_j) < P_i$.*

As a first step towards a better understanding of stratified (general) programs, we study in more detail their semantics. In view of Lemma 7.1, to study Herbrand models of a general program $P$ and its completion, it suffices to consider the pre-fixpoints and fixpoints of its immediate consequence operator $T_P$. However, as just observed, the associated immediate consequence operator $T_P$ does not need to be monotonic. This brings us to the study of nonmonotonic operators and their pre-fixpoints and fixpoints in an abstract setting. We follow here the presentation of [3].

### *7.3. Nonmonotonic operators and their fixpoints*

Consider an arbitrary, but fixed, complete lattice and assume the notation used in Subsection 3.5. All operators are considered on this fixed lattice. First we define *cumulative powers* of an operator $T$. We put

$$T \Uparrow 0(I) = I, \qquad T \Uparrow (n+1)(I) = T(T \Uparrow n(I)) \cup T \Uparrow n(I),$$

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

Cumulative powers easily relate to the usual powers as clearly for all $\alpha \leqslant \omega$ and $I$

$$T \Uparrow \alpha(I) = (T \cup Id) \uparrow \alpha(I)$$

where $Id$ is the identity operator, $\cup$ stands for a union of two operators and the powers defined in Subsection 3.5 are now adopted for arbitrary operators.

We have the following lemma.

**7.5.** LEMMA. *If $T$ is finitary then, for all $I$, $T \Uparrow \omega(I)$ is a pre-fixpoint of $T$, i.e.*

$$T(T \Uparrow \omega(I)) \subseteq T \Uparrow \omega(I).$$

PROOF. Since $T$ is finitary,

$$T(T \Uparrow \omega(I)) \subseteq \bigcup_{n=0}^{\infty} T(T \Uparrow n(I)) \subseteq \bigcup_{n=0}^{\infty} T \Uparrow (n+1)(I) \subseteq T \Uparrow \omega(I). \qquad \square$$

We say that an operator $T$ is *growing* if, for all $I, J, M$,

$$I \subseteq J \subseteq M \subseteq T \Uparrow \omega(I) \text{ implies } T(J) \subseteq T(M).$$

Thus growing is a restricted from of monotonicity.

The following lemma holds.

**7.6.** LEMMA. *If $T$ is growing then, for all $I$, $T \Uparrow \omega(I) \subseteq I \cup T(T \Uparrow \omega(I))$.*

PROOF. An easy proof by induction shows that, for all $i \geqslant 0$,

$$T \Uparrow i(I) \subseteq I \cup \bigcup_{n=0}^{\infty} T(T \Uparrow n(I)). \tag{7.1}$$

We now have

$$\begin{aligned} T \Uparrow \omega(I) &= \bigcup_{n=0}^{\infty} T \Uparrow n(I) \\ &\subseteq I \cup \bigcup_{n=0}^{\infty} T(T \Uparrow n(I)) \quad \text{(by (7.1))} \\ &\subseteq I \cup T(T \Uparrow \omega(I)). \quad \text{(by assumption).} \qquad \square \end{aligned}$$

The following corollary generalizes Lemma 6.9 and shows interest in studying finitary and growing operators.

**7.7.** COROLLARY. *Let $T$ be finitary and growing. Suppose that $I \subseteq T(I)$. Then $T \Uparrow \omega(I)$ is a fixpoint of $T$.*

PROOF. Since $T$ is growing, $I \subseteq T(I) \subseteq T(T \Uparrow \omega(I))$, so $I \cup T(T \Uparrow \omega(I)) = T(T \Uparrow \omega(I))$ and the claim follows by Lemmas 7.5 and 7.6. $\square$

Next, we study families of operators. Let $T_1, \ldots, T_n$ be operators. We put

$$N_0 = I, \qquad N_1 = T_1 \Uparrow \omega(N_0), \ \ldots, \ N_n = T_n \Uparrow \omega(N_{n-1}).$$

Clearly, $N_0 \subseteq N_1 \subseteq \cdots \subseteq N_n$. Of course, all $N_i$'s depend on $I$ and from the context it will be always clear from which one.

Let $T$ stand for the *union* of the operators $T_1, \ldots, T_n$, i.e. for the operator defined by

$$T(X) = \bigcup_{i=1}^{n} T_i(X).$$

We wish to determine under which conditions $N_n$ is a fixpoint of $T$. To this purpose we introduce the following concept. We call a sequence of operators $T_1, \ldots, T_n$ *local* if, for all $I$, $J$,

$$I \subseteq J \subseteq N_n \text{ implies } T_i(J) = T_i(J \cap N_i) \quad \text{for } i = 1, \ldots, n.$$

Informally, locality means that each $T_i$ is determined by its values on the subsets of $N_i$.

The following two lemmas show interest in studying local sequences of operators.

**7.8.** LEMMA. *Suppose that the sequence $T_1, \ldots, T_n$ is local and that all $T_i$'s are finitary. Then $T(N_n) \subseteq N_n$.*

PROOF. We have

$$\begin{aligned} T(N_n) &= \bigcup_{i=1}^{n} T_i(N_n) \\ &= \bigcup_{i=1}^{n} T_i(N_i) \quad \text{(by locality)} \\ &\subseteq \bigcup_{i=1}^{n} N_i \quad \text{(by Lemma 7.5)} \\ &= N_n. \quad \square \end{aligned}$$

**7.9.** LEMMA. *Suppose that the sequence $T_1, \ldots, T_n$ is local and that all $T_i$'s are growing. Then $N_n \subseteq I \cup T(N_n)$.*

PROOF. We proceed by induction on $n$. If $n = 1$, the lemma reduces to Lemma 7.6.

Assume the lemma holds for $n-1$. Then, again by Lemma 7.6,

$$\begin{aligned} N_n &\subseteq N_{n-1} \cup T_n(N_n) \\ &\subseteq I \cup \bigcup_{i=1}^{n-1} T_i(N_{n-1}) \cup T_n(N_n) \quad \text{(by induction hypothesis)} \\ &= I \cup \bigcup_{i=1}^{n-1} T_i(N_n) \cup T(N_n) \quad \text{(by locality)} \\ &= I \cup T(N_n). \quad \square \end{aligned}$$

**7.10.** COROLLARY. *Suppose that the sequence $T_1, \dots, T_n$ is local and that all $T_i$'s are finitary and growing. Then $N_n = I \cup T(N_n)$.*

Thus for a local sequence $T_1, \dots, T_n$ of finitary and growing operators, $N_n$ is a fixpoint of $T$ when $I = \emptyset$.

We now prove that under some assumptions $N_n$ is a minimal pre-fixpoint of $T$ containing $I$.

**7.11.** LEMMA. *Suppose that the sequence $T_1, \dots, T_n$ is local and that all $T_i$'s are growing. Suppose $I \subseteq J \subseteq N_n$ and $T(J) \subseteq J$. Then $J = N_n$.*

PROOF. We prove by induction on $j = 0, \dots, n$ that

$$N_j \subseteq J. \tag{7.2}$$

For $j = 0$ it is part of the assumptions. Assume the claim holds for some $j < n$. We now prove by induction on $k$ that

$$T_{j+1} \Uparrow k(N_j) \subseteq J. \tag{7.3}$$

For $k = 0$ this is just (7.2). So assume (7.3) holds for some $k \geqslant 0$. We then have

$$\begin{aligned} T_{j+1} \Uparrow (k+1)(N_j) &\subseteq T_{j+1}(T_{j+1} \Uparrow k(N_j)) \cup J \\ &\subseteq T_{j+1}(J \cap N_{j+1}) \cup J && \text{(by (7.3) and since } T_{j+1} \text{ is growing)} \\ &= T_{j+1}(J) \cup J && \text{(by locality)} \\ &\subseteq J. && \text{(by the assumptions)} \end{aligned}$$

Thus, by induction, for all $k \geqslant 0$ (7.3) holds, so $N_{j+1} \subseteq J$. This proves (7.2) for all $j = 0, \dots, n$ and concludes the proof. □

Finally, we provide an alternative characterization of $N_i$. To make it more readable we now assume that $I = \emptyset$. Then, by definition, $N_0 = \emptyset$.

Let now $\boldsymbol{T}_i$ denote the union of $T_1, \dots, T_i$, i.e. $\boldsymbol{T}_i(X) = T_1(X) \cup \cdots \cup T_i(X)$.

**7.12.** LEMMA. *Suppose that the sequence $T_1, \dots, T_n$ is local and that all $T_i$'s are finitary and growing. Let*

$$\begin{aligned} \boldsymbol{K}_1 &= \{J : \boldsymbol{T}_1(J) = J,\ T_1(J \cap N_1) \subseteq T_1(J)\}, \\ \boldsymbol{K}_2 &= \{J : \boldsymbol{T}_2(J) = J,\ T_2(J \cap N_2) \subseteq T_2(J),\ N_1 \subseteq J\}, \\ &\dots \\ \boldsymbol{K}_n &= \{J : \boldsymbol{T}_n(J) = J,\ T_n(J \cap N_n) \subseteq T_n(J),\ N_{n-1} \subseteq J\}. \end{aligned}$$

*Then for $i = 1, \dots, n$, $\bigcap \boldsymbol{K}_i = N_i$.*

Note that each $\boldsymbol{K}_i$ is the collection of all fixpoints of $\boldsymbol{T}_i$ which include $N_{i-1}$, where additionally the condition $T_i(J \cap N_i) \subseteq T_i(J)$ is required.

PROOF. Fix some $i$, $1 \leqslant i \leqslant n$. By Corollary 7.10 used for $I = \emptyset$ and $n = i$ and the fact that $N_{i-1} \subseteq N_i$, we conclude that $N_i$ belongs to $\boldsymbol{K}_i$. Thus $\bigcap \boldsymbol{K}_i \subseteq N_i$.

To prove the converse take $J \in \boldsymbol{K}_i$. We prove by induction on $k$ that for $k \geqslant 0$

$$T_i \Uparrow k(N_{i-1}) \subseteq J. \tag{7.4}$$

For $k = 0$ it holds by the definition of $\boldsymbol{K}_i$. Assume this claim holds for some $k \geqslant 0$. Then

$$T_i \Uparrow k(N_{i-1}) \subseteq N_i, \tag{7.5}$$

so by (7.4) and (7.5) and the fact that $T_i$ is growing

$$\begin{aligned} T_i(T_i \Uparrow k(N_{i-1})) &\subseteq T_i(J \cap N_i) \\ &\subseteq T_i(J) \qquad \text{(by definition of } \boldsymbol{K}_i) \\ &\subseteq \boldsymbol{T}_i(J) \\ &\subseteq J. \qquad \text{(by definition of } \boldsymbol{K}_i). \end{aligned}$$

Thus the claim holds for $k+1$. This implies $N_i \subseteq J$, so $N_i \subseteq \bigcap \boldsymbol{K}_i$. □

### 7.4. *Semantics of stratified programs*

We now apply the results of the previous subsection to provide a semantics for stratified programs. Throughout this section we consider a general program $P$ stratified by $P = P_1 \cup \cdots \cup P_n$. We now define a sequence of Herbrand interpretations by putting

$$M_1 = T_{P_1} \Uparrow \omega(\emptyset), \qquad M_2 = T_{P_2} \Uparrow \omega(M_1), \ \ldots, \ M_n = T_{P_n} \Uparrow \omega(M_{n-1}).$$

Let $M_P = M_n$. Note that $M_P$ depends on the stratification and that for programs $P$, $M_P$ has already a different meaning. We shall show in the next subsection that these apparent ambiguities in fact do not exist—$M_P$ does not depend on the stratification of $P$ and consequently, by virtue of the Characterization Theorem 3.13, it coincides for programs $P$ with the previous meaning.

We first prove that $M_P$ is a model of $P$. To this purpose we need the following lemmas.

**7.13.** LEMMA. *Consider a stratum $P_i$ $(1 \leqslant i \leqslant n)$. $T_{P_i}$ considered as an operator on the complete lattice $\{I : I \subseteq B_P\}$ is growing.*

PROOF. Suppose that for some $I \subseteq B_P$, $I \subseteq J \subseteq M \subseteq T_{P_i} \Uparrow \omega(I)$ and let $A \in T_{P_i}(J)$. For some general clause $A \leftarrow L_1, \ldots, L_n$ from ground$(P_i)$ we have $J \models L_1 \wedge \cdots \wedge L_n$. If $L_i$ is positive then also $M \models L_i$. If $L_i$ is negative, say $\neg p(t_1, \ldots, t_k)$, then neither $p(t_1, \ldots, t_k) \in I$ nor $p$ appears in a head of a general clause from $P_i$ because $P_i$ is a stratum. However, for any Herbrand interpretation $N \subseteq B_P$ and a ground atom $r(s_1, \ldots, s_m)$, if $r(s_1, \ldots, s_m) \in T_{P_i} \Uparrow \omega(N)$ then $r(s_1, \ldots, s_m) \in N$ or $r$ appears in a head of general clause from $P_i$. Thus $p(t_1, \ldots, t_k) \notin T_{P_i} \Uparrow \omega(I)$, so $M \models L_i$, as well. This implies that $A \in T_{P_i}(M)$. □

**7.14.** LEMMA. *Consider the strata $P_1, \ldots, P_n$. The sequence of operators $T_{P_1}, \ldots, T_{P_n}$ considered on the complete lattice $\{I : I \subseteq B_P\}$ is local.*

PROOF. Choose some $I \subseteq B_P$ and consider the sequence $N_1, \ldots, N_n$ of subsets of $B_P$ defined in the previous subsection. Fix some $i$, $1 \leqslant i \leqslant n$. Suppose that $p(t_1, \ldots, t_k) \in N_n - N_i$. Then $p$ appears in a head of a general clause from $\bigcup_{j=i+1}^{n} P_j$, so by the definition of stratification $p$ does not appear in a general clause from $P_i$. Thus $p(t_1, \ldots, t_k) \notin B_{P_i}$. Hence $N_n \cap B_{P_i} \subseteq N_i$ and consequently,

$$N_n \cap B_{P_i} = N_i \cap B_{P_i}, \tag{7.6}$$

since $N_i \subseteq N_n$. Suppose now that $I \subseteq J \subseteq N_n$. We have

$$\begin{aligned} J \cap B_{P_i} &= J \cap N_n \cap B_{P_i} \\ &= J \cap N_i \cap B_{P_i} \quad \text{(by (7.6))}. \end{aligned} \tag{7.7}$$

Thus

$$\begin{aligned} T_{P_i}(J) &= T_{P_i}(J \cap B_{P_i}) && \text{(by definition of } T_{P_i}) \\ &= T_{P_i}(J \cap N_i \cap B_{P_i}) && \text{(by (7.7))} \\ &= T_{P_i}(J \cap N_i). && \text{(by definition of } T_{P_i}). \end{aligned}$$ □

We can now conclude by the following theorem

**7.15.** THEOREM (Characterization Theorem) (Apt, Blair and Walker [3]). *Let $P$ be a general program stratified by $P = P_1 \cup \cdots \cup P_n$. Then*

(i) *$M_P$ is a Herbrand model of $P$.*
(ii) *$M_P$ is a minimal Herbrand model of $P$.*
(iii) *$M_P$ is a Herbrand model of comp($P$).*

PROOF. (i) By Lemmas 7.1(i), (iii), 7.13 and 7.14 and Corollary 7.10.
(ii) By Lemmas 7.1(i) and 7.11.
(iii) By Lemmas 7.1(ii), (iii), 7.13 and 7.14 and Corollary 7.10. □.

Finally, we provide an alternative characterization of $M_P$. To prove the desired theorem we first introduce a notation and prove a lemma.

Given a general program $P$, let

$$Neg_P = \{A: \text{for some } B \leftarrow L_1, \ldots, L_n \in \text{ground}(P) \text{ and } i,\ 1 \leqslant i \leqslant n,\ L_i = \neg A\}.$$

Thus $Neg_P$ stands for the set of ground instances of atoms whose negation occurs in a hypothesis of general clause from $P$.

**7.16.** LEMMA. *Let $P$ be a general program and $I$,$J$ Herbrand interpretations. Suppose that $I \subseteq J$ and $I \cap Neg_P = J \cap Neg_P$. Then $T_P(I) \subseteq T_P(J)$.*

PROOF. Suppose that $A \in T_P(I)$. For some general clause $A \leftarrow L_1, \ldots, L_n$ from ground($P$), we have $I \models L_1 \wedge \cdots \wedge L_n$. If $L_i$ is positive then, by assumption, also $J \models L_i$. If $L_i$ is negative, say $\neg B$, then $B \notin I$, so $B \notin I \cap Neg_P$ and by assumption $B \notin J \cap Neg_P$. But, by definition, $B \in Neg_P$, so $B \notin J$, i.e. $J \models L_i$. This implies that $A \in T_P(J)$. □

Assume now a given stratification $P_1 \cup \cdots \cup P_n$ of $P$. To shorten the notation let from now on $\boldsymbol{P}_i$ stand for $P_1 \cup \cdots \cup P_i$. Then $P = \boldsymbol{P}_n$. Let $M$ range over the subsets of $B_P$. Put

$$M(\boldsymbol{P}_1) = \bigcap \{M : T_{P_1}(M) = M\},$$
$$M(\boldsymbol{P}_2) = \bigcap \{M : T_{P_2}(M) = M,\ M \cap B_{P_1} = M(\boldsymbol{P}_1)\},$$
$$\ldots$$
$$M(\boldsymbol{P}_n) = \bigcap \{M : T_{P_n}(M) = M,\ M \cap B_{P_{n-1}} = M(\boldsymbol{P}_{n-1})\}.$$

Note that by Theorem 7.1(ii) each $M(\boldsymbol{P}_i)$ is the intersection of all Herbrand models of $comp(\boldsymbol{P}_i)$ which on the previous Herbrand base $B_{P_{i-1}}$ agree with the previous model $M(\boldsymbol{P}_{i-1})$. In the definition of $M(\boldsymbol{P}_i)$ each $T_{P_i}$ is considered as an operator on the complete lattice $B_P$. We now prove the following theorem.

**7.17.** THEOREM (Apt, Blair and Walker [3]). $M_P = M(P)$.

PROOF. We prove by induction that, for $i = 1, \ldots, n$, $M_i = M(\boldsymbol{P}_i)$. This implies the claim since $M_n = M_P$ and $M(\boldsymbol{P}_n) = M(P)$. For $i = 1$ it is a consequence of the Characterization Theorem 3.13 and the fact that $T_{P_1}(M) \subseteq B_{P_1}$.

Suppose the claim holds for some $i$, $1 \leqslant i < n$. Note that by the Characterization Theorem 7.15(iii) and Lemma 7.1(ii), $T_{P_{i+1}}(M_{i+1}) = M_{i+1}$. Also $M_{i+1} \cap B_{P_i} = M(\boldsymbol{P}_i)$ by the induction hypothesis and the definition of stratification. Thus $M_{i+1}$ is an element of the collection whose intersection is $M(\boldsymbol{P}_{i+1})$. This proves that $M(\boldsymbol{P}_{i+1}) \subseteq M_{i+1}$.

To establish the converse inclusion, take $M$ from the collection whose intersection is $M(\boldsymbol{P}_{i+1})$. Thus

$$T_{P_{i+1}}(M) = M \tag{7.8}$$

and

$$M(\boldsymbol{P}_i) = M \cap B_{P_i}. \tag{7.9}$$

Equation (7.9) implies by the induction hypothesis $M_i = M \cap B_{P_i}$ so

$$M_i \subseteq M. \tag{7.10}$$

Moreover, by the definition of stratification, $M \cap Neg_{P_{i+1}} \subseteq B_{P_i}$, so

$$M \cap Neg_{P_{i+1}} = Neg_{P_{i+1}} \cap B_{P_i}. \tag{7.11}$$

Now

$$\begin{aligned} M \cap M_i \cap Neg_{P_{i+1}} &= M_i \cap Neg_{P_{i+1}} && \text{(by (7.10))} \\ &= M(\boldsymbol{P}_i) \cap Neg_{P_{i+1}} && \text{(by the induction hypothesis)} \\ &= M \cap B_{P_i} \cap Neg_{P_{i+1}} && \text{(by (7.9))} \\ &= M \cap Neg_{P_{i+1}} && \text{(by (7.11)).} \end{aligned}$$

Thus by Lemma 7.16

$$T_{P_{i+1}}(M \cap M_i) \subseteq T_{P_{i+1}}(M). \tag{7.12}$$

We can now apply Lemma 7.12 with $N_j = M_j$ and $T_j = T_{P_j}$. By (7.8), (7.12) and (7.10)

$M \in \boldsymbol{K}_{i+1}$ so $\bigcap \boldsymbol{K}_{i+1} \subseteq M$ and, by Lemma 7.12, $M_{i+1} \subseteq M$. By the choice of $M$, $M_{i+1} \subseteq M(\boldsymbol{P}_{i+1})$. This concludes the proof of the induction step. □

### 7.5. *Perfect model semantics*

We now prove that the Herbrand model $M_P$ does not depend on the stratification of $P$. We follow here the approach of Przymusiński [81]. It is conceptually advantageous to carry out these considerations in a more abstract setting.

Consider a given general program $P$. Let $<$ be a well-founded ordering on the Herbrand base $B_P$ of $P$. If $A < B$ then we say that $A$ has a *higher priority than* $B$.

Let $M, N \subseteq B_P$. We call a Herbrand interpretation $N$ *preferable to* $M$, and write $N < M$, if $N \neq M$ and for every $B \in N - M$ there exists an $A \in M - N$ such that $A < B$. We write $N \leqslant M$ if $N = M$ or $N < M$. We call a Herbrand model of $P$ *perfect* if there are no Herbrand models of $P$ preferable to it. Thus a perfect model of $P$ is a $<$-minimal Herbrand model of $P$.

The intuition behind these definitions is the following. $N$ is preferable to $M$ if it is obtained from $M$ by possible adding/removing some atoms and an addition of an atom to $N$ is always compensated by the simultaneous removal from $M$ of an atom of higher priority. This reflects the fact that we are determined to minimize higher-priority atoms even at the cost of adding atoms of lower priority. A model is then perfect if this form of minimization of higher-priority atoms is achieved in it.

The following lemma clarifies the status of perfect models.

**7.18.** LEMMA. *Let $P$ be a general program and let $<$ be a well-founded ordering on $B_P$.*

(i) *Every perfect model of $P$ is minimal.*

(ii) *For no two Herbrand interpretations $M$, $N$ of $P$, both $M < N$ and $N < M$.*

PROOF. (i) Immediate, since $N \subseteq M$ implies $N < M$.

(ii) Suppose by contradiction that for some Herbrand interpretations $M$, $N$ of $P$ both $M < N$ and $N < M$. Then none of them is a subset of the other. Thus $N - M$ is nonempty. Let $A_0 \in N - M$. $N$ is preferable to $M$, so for some $A_1 \in M - N$, $A_1 < A_0$. But $M$ is preferable to $N$ so, for some $A_2 \in N - M$, $A_2 < A_1$. Continuing in this way we obtain an infinite $<$-descending sequence of ground atoms which contradicts the assumption that $<$ is a well-founded ordering on $B_P$. □

One can also prove that the relation "$N$ is preferable to $M$" is a partial order but we shall not need this in the sequel.

Subsequent considerations are carried out for a fixed stratified general program $P$ and a well-founded ordering $<$ on $B_P$ defined by first putting, for two relation symbols $p$, $q$,

$$p < q \text{ iff there exists a path from } q \text{ to } p \text{ in } D_P \text{ with a negative arc,}$$

and then putting, for two atoms $A$, $B \in B_P$,

$$A < B \text{ iff } p < q \text{ where } p \text{ appears in } A \text{ and } q \text{ appears in } B.$$

By the definition of a stratified program, $<$ is a well-founded ordering on $B_P$. Note that the orientation of $<$ is different than the one suggested by $D_P$. If $p<q$ then $p$ is defined in a strictly lower stratum than $q$ and all ground atoms containing $p$ are of higher priority than those containing $q$. Fix from now on a stratification $P_1 \cup \cdots \cup P_n$ of $P$. Note that $M_P \cap B_{P_i} = M_i$. For a Herbrand interpretation $N$ of $L_{P_i}$, denote $N \cap B_{P_i}$ by $N_i$. Note that $N_1 \subseteq N_2 \subseteq \cdots \subseteq N_n$.

**7.19.** LEMMA. *Let $N$ be a Herbrand model of $P$. Then for all $i=1,\ldots,n$ we have $M_i \leqslant N_i$.*

PROOF. We proceed by induction on $i$. Note that $N_i \models P_i$. As $P_1$ is a program, by the Characterization Theorem 3.13, $M_1$ is its smallest model. Thus $M_1 \subseteq N_1$, and a fortiori $M_1 \leqslant N_1$.

Suppose the claim holds for some $i \geqslant 1$. Call an element $B \in M_{i+1}$ *regular* if $B \notin N_{i+1}$ implies that, for some $A \in N_{i+1} - M_{i+1}$, $A < B$. To prove that $M_{i+1} \leqslant N_{i+1}$ we need to show that all elements of $M_{i+1}$ are regular.

We have $M_{i+1} = \bigcup_{k=0}^{\infty} T_{P_{i+1}} \Uparrow k(M_i)$. We now prove by induction on $k$ that all elements of $T_{P_{i+1}} \Uparrow k(M_i)$ are regular. To take care of the case $k=0$, consider some $B \in M_i - N_{i+1}$. Then $B \notin N_i$, so, by the induction hypothesis, for some $A \in N_i - M_i$, $A < B$. Moreover, $N_i \subseteq B_{P_i}$, so $A \in B_{P_i}$. But $M_{i+1} \cap B_{P_i} = M_i$, so $A \notin M_{i+1}$. Thus $A \in N_{i+1} - M_{i+1}$ and consequently $B$ is regular.

To take care of the induction step, fix $k \geqslant 0$ and denote $T_{P_{i+1}} \Uparrow k(M_i)$ by $M$. Assume that all elements of $M$ are regular and consider some $B \in T_{P_{i+1}}(M) - M$. For some general clause $B \leftarrow L_1, \ldots, L_s$ in ground$(P_{i+1})$, $M \models L_1 \wedge \cdots \wedge L_s$. Let $A_1, \ldots, A_l$ be the positive literals among $L_1, \ldots, L_s$ and let $\neg B_1, \ldots, \neg B_m$ be the negative literals among $L_1, \ldots, L_s$. We have $A_1, \ldots, A_l \in M$ and $B_1, \ldots, B_m \notin M$. Suppose now $B \notin N_{i+1}$. $N_{i+1}$ is a model of $P_{i+1}$, so either some $A_j \notin N_{i+1}$ or some $B_j \in N_{i+1}$. If some $A_j \notin N_{i+1}$ then $A_j \in M - N_{i+1}$. As $A_j$ is regular, for some $A \in N_{i+1} - M_{i+1}$, $A < A_j$. By the definition of $<$, also $A < B$. If some $B_j \in N_{i+1}$ then $B_j \in N_{i+1} - M$, so $B_j \in N_{i+1} - M_i$. Moreover, by the definition of stratification $B_j \in B_{P_i}$. But $M_{i+1} \cap B_{P_i} = M_i$, so $B_j \notin M_{i+1}$. Thus $B_j \in N_{i+1} - M_{i+1}$. Moreover, by the definition of $<$ we have $B_j < B$.

We thus showed that $B$ is regular. By induction on $k$ we now proved that $M_{i+1} \leqslant N_{i+1}$. Thus by induction on $i$, we proved the lemma. □

**7.20.** LEMMA. *Let $I$, $J$ be Herbrand interpretations for $L_P$. If for all $i=1,\ldots,n$ we have $I_i \leqslant J_i$, then $I \leqslant J$.*

PROOF. Let $B \in I - J$. For some $i$, $1 \leqslant i \leqslant n$, we have $B \in I_i - J$. So $B \in B_{P_i}$. But $J_i = J \cap B_{P_i}$, so $B \notin J_i$. Since $I_i \leqslant J_i$, for some $A \in J_i - I_i$, $A < B$. So $A \in B_{P_i}$. But $I_i = I \cap B_{P_i}$, so $A \notin I$. □

This brings us to the main result of this subsection.

**7.21.** THEOREM (Przymusiński [81]). (i) *For every Herbrand model $N$ of $P$, $M_P \leqslant N$.*
(ii) *$M_P$ is the unique perfect model of $P$.*

PROOF. (i) By Lemmata 7.19 and 7.20.

(ii) By (i) and Lemma 7.18(ii), $M_P$ is a perfect model of $P$. By (i) it is also unique. □

Note that (ii) in view of lemma 7.18(i) provides an alternative proof of Theorem 7.15(ii).

**7.22.** COROLLARY (Apt, Blair and Walker [3]). *$M_P$ does not depend on the stratification of $P$.*

PROOF. The proof of Theorem 7.12(ii) does not depend on the stratification of $M_P$. □

Theorems 7.15 and 7.16 show that $M_P$ is a natural model of a stratified program $P$. However, the most convincing evidence that $M_P$ is indeed natural, is supplied by Theorem 7.22. The notion of a perfect model turns out to be the key concept in assessing the character of $M_P$.

### *7.6. Bibliographic remarks*

Stratified programs form a simple generalization of a class of database queries introduced in [20]. Similar concepts were also introduced in [7] and, in the context of deductive databases, in [76].

The proofs of Theorems 7.17 and 7.21 and of Corollary 7.22 differ from the original ones. The notion of a stratified program was further generalized by Przymusiński [81] to a *locally stratified program*. Lifschitz [63] provides a characterization of the model $M_P$ of a stratified program $P$ using the prioritized circumscription. Other connections between stratification, the model $M_P$ and nonmonotonic reasoning are surveyed in [82]. Apt and Blair [2] analyze the recursion-theoretic complexity of the model $M_P$.

## 8. 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.

### *8.1. General programs*

SLD-resolution and the negation as failure rule was combined by Clark [21] into a more powerful computation mechanism called *SLDNF-resolution* allowing us to refute general goals from general programs. The reader is referred to [65] for a detailed account of SLDNF-resolution.

Shepherdson [90] discusses and compares various approaches to the proof theory and semantics of general programs. The strongest completeness results dealing with the SLDNF-resolution were proved in [19, 60].

### 8.2. *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 [36] proposed on alternative computation mechanism based on a tableau method. Gallier and Raatz [40] introduced a computation mechanism in the form of an interpreter using graph reduction. Brough and Walker [17] studied interpreters with various stopping criteria for function-free programs. Apt, Blair and Walker [3] introduced an interpreter with a loop-checking mechanism and with an ineffective means of handling negative literals. Przymusiński [81] 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 [76] and [100] respectively.

*Semantics*

Mycroft [75] suggested to use 3-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 [35, 58, 60].

To describe the meaning of general programs Minker [73] proposed the use of minimal models (leading to the *generalized closed world assumption* GCWA), Bidoit and Hull [9] proposed the use of *positivistic models* and Przymusiński [81] introduced the concept of a *perfect model.*

### 8.3. *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 Ramakrishnan [6]), reduction of recursive queries to nonrecursive ones (see e.g. [78]), comparison of expressive power between various query languages (see e.g. [20, 91]),

and handling of negative information both in terms of intended semantics (see e.g. [73, 3, 99, 63, 77, 81]) and in terms of query processing, handling of updates and integrity constraint checking (see e.g. [44, 29, 67]).

Earlier research in this area is surveyed in [38] while more recent research is discussed in [51, Section 4] and [74].

### *8.4. PROLOG*

PROLOG stands for *pro*gramming in *logic*. It is a programming language conceived and implemented in the beginning of 1970s by Colmerauer et al. [26]. 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. [49a]), by means of denotational semantics (see e.g. [34]) or by relating both approaches (see e.g. [28]).

More practical considerations, apart of a study of implementations of PROLOG (see e.g. [18]), led to an investigation of efficient backtracking mechanisms (see e.g. [27]) and of various additions, like metafacilities (see e.g. [13, 96]), modules (see e.g. [41]), control mechanisms (see e.g. [76]) and parallelism (see e.g. Concurrent Prolog of Shapiro [87] and PARLOG of Clark and Gregory [24]).

Good books on PROLOG programming have been written by Bratko [16], and Sterling and Shapiro [96].

### *8.5. 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 [85].

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 Subsection 5.9 written in a clausal form.

Recent proposals in this area are collected in de Groot and Lindstrom [43] which is a standard reference in this domain. See also [8, 41, 32].

### *8.6. 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 certain investigations of more theoretical nature.

Use of logic programming as a formalism for knowledge representation and reasoning was advocated by Kowalski [54]. Analysis and implementation of more powerful logics and various forms of reasoning in the framework of logic programming was undertaken by Fariñas del Cerro [33] for modal logic, by Van Emden [30] for quantitative reasoning and by Poole [80] 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 the Journal of Logic Programming [50]) and with the use of logic programming and PROLOG for the construction of expert system shells (see e.g. [16, 101].)

## 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 by no means objective (as none is).

1972: A. Colmerauer and R. Kowalski collaborated to develop from resolution theorem proving a programming language.

1973: Colmerauer et al. [26] implemented PROLOG.

1974: Kowalski [53] proposed logic (programming) as a programming language and introduced what is now called SLD-resolution.

1976: Van Emden and Kowalski [31] studied the semantics of logic programs and introduced the ubiquitous immediate consequence operator $T_P$.

1978: Reiter [83] proposed in the context of deductive databases the *Closed World Assumption* rule as a means of deducing negative information.

1978: Clark [21] 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 [54] analyzed logic programming as a formalism for knowledge representation and problem solving.

1979: Kowalski [55] investigated logic programming as a formalism for a systematic development of algorithms.

1981: Clark and Gregory [23] proposed a parallel version of logic programming which influenced subsequent language proposals in this area.
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 [4] 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 [25], 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: Jaffar, Lassez and Lloyd [48] proved completeness of the negation as failure rule with respect to the completion of a program.
1984: Lloyd [64] 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 [43] 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 [3] and Van Gelder [99] 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.
1985–1989: M. Fitting and K. Kunen developed in [35, 36, 58, 60] a theory of logic programming based on 3-valued logic.

## Note

In this chapter we use the terminology of Lloyd in [64] which differs from that of Lloyd in [65]. In [65] 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.

## Acknowledgment

We would like to thank Marc Bezem, Roland Bol, Stephane Grumbach and Jan Willem Klop for detailed comments on the first version of this paper. Also, we profited from discussions with Howard Blair, Lawrence Cavedon, Maarten van Emden, Jean Gallier, Joxan Jaffar, Jean-Louis Lassez, John Lloyd, Michael Maher, Katuscia Palamidessi, Teodor Przymusiński, John Shepherdson, Wayne Snyder and Rodney Topor who commented on the subject of this paper in four languages. Our task was significantly simplified thanks to John Lloyd who collected in [64] most of the results presented here in a single framework. Figure 3 was reproduced with his permission. We would like to thank Eline Meys and Ria Riechelmann-Huis for typing the continuously growing and changing manuscript.

## References


[1] ANDRÉKA, H. and I. NÉMETI, The generalized completeness of Horn predicate logic as a programming language, *Acta Cybernet.* **4** (1978) 3–10.

[2] APT, K.R. and H.A. BLAIR, Arithmetic classification of perfect models of stratified programs, in: *Proc. 5th Internat. Conf. on Logic Programming* (MIT Press, Cambridge, MA, 1988) 765–779.

[3] APT, K.R., H.A. BLAIR and A. WALKER, Towards a theory of declarative knowledge, in: J. Minker, ed., *Foundations of Deductive Databases and Logic Programming* (Morgan Kaufmann, Los Altos, CA, 1988).

[4] APT, K.R. and M.H. VAN EMDEN, Contributions to the theory of logic programming, *J. ACM* **29** (3) (1982) 841–862.

[5] AQUILANO, C., R. BARBUTI, P. BOCCHETTI and M. MARTELLI, Negation as failure: Completeness of the query evaluation process for Horn clause programs with recursive definitions, *J. Automat. Reason.* **2** (1986) 155–170.

[6] BANCILHON, F. and R. RAMAKRISHNAN, An amateur's introduction to recursive query processing strategies, in: *Proc. ACM Internat. Conf. on Management of Data* (1986) 16–52.

[7] BARBUTI, R. and M. MARTELLI, Completeness of the SLDNF-resolution for a class of logic programs, in: *Proc. 3rd Internat. Conf. on Logic Programming*, Lecture Notes in Computer Science, Vol. 225 (Springer, Berlin, 1986) 600–614.

[8] BELLIA, M. and G. LEVI, The relation between logic and functional languages, a survey, *J. Logic Programming* **3** (1986) 217–236.

[9] BIDOIT, N. and R. HULL, Positivism versus minimalism in deductive databases, in: *Proc. 5th ACM SIGACT-SIGMOD Symp. on Principles of Database Systems* (1986) 123–132.

[10] BIRKHOFF, G., *Lattice Theory*, American Mathematical Society Colloquium Publications, Vol. 25 (1973).

[11] BLAIR, H.A., The recursion-theoretic complexity of predicate logic as a programming language, *Inform. and Control* **54** (1–2) (1982) 25–47.

[12] BLAIR, H.A., Decidability in the Herbrand base, Manuscript, presented at the Workshop on Foundations of Deductive Databases and Logic Programming, Washington, DC, 1986.

[13] BÖRGER, E., Unsolvable decision problems for PROLOG programs, in: E. Börger, ed., *Computation Theory and Logic*, Lecture Notes in Computer Science, Vol. 270 (Springer, Berlin, 1987) 37–48.

[14] BÖRGER, E., Logic as machine: complexity relations between programs and formulae, in: E. Börger, ed., *Trends in Theoretical Computer Science* (Computer Science Press, Rockville, MD, 1988).

[15] BOWEN, K.A. and R.A. KOWALSKI, Amalgamating language and metalanguage in logic programming, in: K.L. Clark and S.-A. Tärnlung, eds., *Logic Programming* (Academic Press, New York, 1982).

[16] BRATKO, I., *PROLOG Programming for Artificial Intelligence* (Addison Wesley, Reading, MA, 1986).

[17] BROUGH, D. and A. WALKER, Some practical properties of logic programming interpreters, in: *Proc. Japan FGCS84 Conf.* (1984) 149–156.

[18] CAMPBELL, J.A., ed., *Implementations of PROLOG* (Ellis Horwood, Chichester, UK, 1984).

[19] CAVEDON, L. and J. LLOYD, A completeness theorem for SLDNF-resolution, *J. Logic Programming* **7**(4) (1989) 177–193.

[20] CHANDRA, A.K. and D. HAREL, Horn clause queries and generalizations, *J. Logic Programming* **2** (1) (1985) 1–15.

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

[22] CLARK, K.L., Predicate logic as a computational formalism, Research Report DOC 79/59, Dept. of Computing, Imperial College, London 1979.

[23] CLARK, K.L. and S. GREGORY, A relational language for parallel programming, in: *Proc. ACM Conf. on Functional Programming Languages and Computer Architecture* (1981) 171–178.

[24] CLARK, K.L. and S. GREGORY, PARLOG: A parallel logic programming language, *ACM Trans. on Programming Languages and Systems* **8** (1) (1986) 1–49.

[25] CLARK, K.L. and S.-A. TÄRNLUND. eds., *Logic Programming* (Academic Press, New York, 1982).

[26] COLMERAUER, A., H. KANOUI, P. ROUSSEL and R. PASERO, Un système de communication

homme-machine en Francais, Tech. Report. Groupe de Recherche en Intelligence Artificielle, Univ. d'Aix-Marseille, 1973.

[27] COX, P.T. and T. PIETRZYKOWSKI, Deduction plans: a basis for intelligent backtracking, in: *IEEE PAMI* **3** (1981) 52–65.

[28] DEBRAY, S.K. and P. MISHRA, Denotational and operational semantics for PROLOG, *J. Logic Programming* **5** (1) (1988) 61–91.

[29] DECKER, H., Integrity enforcement in deductive databases, in: *Proc. 1st Internat. Conf. on Expert Database Systems* (1986).

[30] EMDEN, M.H. VAN, Quantitative deduction and its fixpoint theory, *J. Logic Programming* **3** (1) (1986) 37–53.

[31] EMDEN, M.H. VAN and R.A. KOWALSKI, The semantics of predicate logic as a programming language, *J. ACM* **23** (4) (1976) 733–742.

[32] EMDEN, M.H. VAN and K. YUKAWA, Logic programming with equations, *J. Logic Programming* **4** (4) (1987) 265–288.

[33] FARIÑAS, L., DEL CERRO, MOLOG: A system that extends PROLOG with modal logic, *New Generation Comput.* **4** (1) (1986) 35–50.

[34] FITTING, M., A deterministic PROLOG fixpoint semantics, *J. Logic Programming* **2** (2) (1985) 111–118.

[35] FITTING, M., A Kripke–Kleene semantics for logic programs, *J. Logic Programming* **2** (4) (1985) 295–312.

[36] FITTING, M., Partial models and logic programming, *Theoret. Comput. Sci.* **48** (1986) 229–255.

[37] FITTING, M., *Computability Theory, Semantics, and Logic Programming* (Oxford Univ. Press, New York, 1987).

[38] GALLAIRE, H., J. MINKER and J.M. NICOLAS, Logic and databases: a deductive approach, *ACM Comput. Surveys* **16** (2) (1984) 153–186.

[39] GALLIER, J., *Logic for Computer Science* (Harper & Row, New York, 1986).

[40] GALLIER, and S. RAATZ, A graph-based interpreter for general Horn clauses, *J. Logic Programming* **4** (2) (1987) 119–156.

[41] GALLIER, J. and S. RAATZ, Extending SLD-resolution to equational Horn clauses using *E*-unification, *J. Logic Programming* **6** (1) (1988) 3–44.

[42] GOGUEN, J.A. and J. MESEGUER, Equality, types, modules and (why not?) generics for logic programming, *J. Logic Programming* **1** (2) (1984) 179–210.

[43] GROOT, D. DE and G. LINDSTROM, eds., *Logic Programming, Functions, Relations and Equations* (Prentice-Hall, Englewood Cliffs, NJ, 1986).

[44] HENSCHEN, L. and H.S. PARK, Compiling the GCWA in indefinite deductive databases, in: J. Minker, ed., *Foundations of Deductive Databases and Logic Programming* (Morgan Kaufmann, Los Altos, CA, 1988).

[45] HERBRAND, J. in: W.D. Goldfarb, ed., *Logical Writings* (Reidel, Dordrecht, 1971).

[46] HILL, R., LUSH-resolution and its completeness, DCL Memo 78, Dept. of Artificial Intelligence, Univ. of Edinburgh, 1974.

[47] ITAI, and J.A. MAKOWSKY, Unification as a complexity measure for logic programming, *J. Logic Programming* **4** (2) (1987) 105–118.

[48] JAFFAR, J., J.-L. LASSEZ and J.W. LLOYD, Completeness of the negation as failure rule, in: *Proc. IJCAI'83* (1983) 500–506.

[49] JAFFAR, J. and P.J. STUCKEY, Canonical logic programs, *J. Logic Programming* **3** (2) (1986) 143–155.

[49a] JONES, N.D. and A. MYCROFT, Stepwise development of operational and denotational semantics for PROLOG, in: *Proc. Internat. Symp. on Logic Programming* (1984) 289–298.

[50] Journal of Logic Programming **4** (1986) *Special Issue on Natural Language and Logic Programming* (MCCORD, M.C., V. DAHL and H. ABRAMSON, guest editors).

[51] KANELLAKIS, P., Elements of relational database theory, in: J. van Leewen, ed., *Handbook of Theoretical Computer Science, Vol. B* (North-Holland, Amsterdam, 1990).

[52] KLOP, J.W. and J.J. CH. MEYER, Toegepaste logica: resolutie logica en epistemische logica, Course Notes, Free University Amsterdam, 1987 in Dutch.

[53] KOWALSKI, R.A., Predicate logic as a programming language, in: *Proc. IFIP'74* (North-Holland, Amsterdam, 1974) 569–574.

[54] KOWALSKI, R.A., *Logic for Problem Solving* (North-Holland, New York, 1979).
[55] KOWALSKI, R.A., Algorithm = logic + control, *Comm. ACM* **22** (7) (1979) 424–435.
[56] KOWALSKI, R.A., The relation between logic programming and logic specification, in: C.A.R. Hoare and J.C. Shepherdson, eds., *Mathematical Logic and Programming Languages* (Prentice-Hall, Englewood Cliffs, NJ, 1985) 11–27.
[57] KOWALSKI, R.A. and D. KUEHNER, Linear resolution with selection function, *Artificial Intelligence* **2** (1971) 227–260.
[58] KUNEN, K., Negation in logic programming, *J. Logic Programming* **4** (4) (1987) 289–308.
[59] KUNEN, K., Answer sets and negation as failure, in: *Proc. 4th Internat. Conf. on Logic Programming* (MIT Press, Cambridge, MA, 1987) 219–228.
[60] KUNEN, K., Signed data dependencies in logic programs, *J. Logic Programming* **7** (4) (1989) 231–245.
[61] LASSEZ, J.-L. and M.J. MAHER, Closures and fairness in the semantics of programming logic, *Theoret. Comput. Sci* **29** (1984) 167–184.
[62] LASSEZ, J.L., M.J. MAHER and K. MARRIOTT, Unification revisited, in: J. Minker, ed., *Foundations of Deductive Databases and Logic Programming* (Morga Kaufmann, Los Altos, CA, 1988).
[63] LIFSCHITZ, V., On the declarative semantics of logic programs with negation, in: J. Minker, ed., *Foundations of Deductive Databases and Logic Programming* (Morgan Kaufmann, Los Altos, CA, 1988).
[64] LLOYD, J.W., *Foundations of Logic Programming* (Springer, Berlin, 1984).
[65] LLOYD, J.W., *Foundations of Logic Programming* (Springer, Berlin, 2nd ed., 1987).
[66] LLOYD, J.W. and J.C. SHEPHERDSON, Partial evaluation in logic programming, Tech. Report CS-87-09, Dept. of Computer Science, Univ. of Bristol, 1987.
[67] LLOYD, J.W., E.A. SONENBERG and R.W. TOPOR, Integrity constraint checking in stratified databases, *J. Logic Programming* **4** (4) (1987) 331–345.
[68] LLOYD, J.W. and R. TOPOR, A basis for deductive databases II, *J. Logic Programming* **3** (1) (1986) 55–67.
[69] MAHER, M., Equivalences of logic programs, in: J. Minker, ed., *Foundations of Deductive Databases and Logic Programming* (Morgan Kaufmann, Los Altos, CA, 1988).
[70] MANIN, Y.I., *A Course in Mathematical Logic* (Springer, New York, 1977).
[71] MARTELLI, A. and U. MONTANARI, An efficient unification algorithm, *ACM Trans. on Programming Languages and Systems* **4** (2) (1982) 258–282.
[72] MENDELSON, E., *Introduction to Mathematical Logic* (Van Nostrand, Princeton, NJ, 2nd ed., 1979).
[73] MINKER, J., On indefinite databases and the closed world assumption, in: D.W. Loveland, ed., *Proc. 6th Conf. on Automated Deduction* Lecture Notes in Computer Science, Vol. 138 (Springer, Berlin, 1982) 292–307.
[74] MINKER, J., Perspectives in deductive databases, *J. Logic Programming* **5** (1) (1988) 33–60.
[75] MYCROFT, A., Logic programs and many-valued logic, in: *Proc. of Symp. on Theoretical Aspects of Computer Science*, Lecture Notes in Computer Science, Vol. 166 (Springer, Berlin, 1984) 274–286.
[76] NAISH, L., *Negation and Control in PROLOG*, Lecture Notes in Computer Science, Vol. 238 (Springer, Berlin, 1986).
[77] NAQVI, S.A., A logic for negation in database systems, in: J. Minker, ed., *Foundations of Deductive Databases and Logic Programming* (Morgan Kaufmann, Los Altos, CA, 1988).
[78] NAUGHTON, J.F. and Y. SAGIV, A decidable class of bounded recursions, in: *Proc. 6th ACM SIGACT-SIGMOD Symp. on Principles of Database Systems* (1987) 227–237.
[79] PATERSON, M.S. and M.N. WEGMAN, Linear unification, *J. Comput. System Sci.* **16** (2) (1978) 158–167.
[80] POOLE, D.L., Default reasoning and diagnosis on theory formation, Tech. Report 86–08, Dept. of Computer Science, Univ. of Waterloo, Waterloo, 1986.
[81] PRZYMUSIŃSKI, T., On the semantics of stratified databases, in: J. Minker, ed., *Foundations of Deductive Databases and Logic Programming* (Morgan Kaufmann, Los Altos, CA, 1988).
[82] PRZYMUSIŃSKI, T., Non-monotonic reasoning vs logic programming: a new perspective, in: Y. Wilks and D. Partridge, eds. *Handbook on the Formal Foundations of A.I.* (Cambridge University Press, Cambridge, in press).
[83] REITER, R., On closed world data bases, in: H. Gallaire and J. Minker, eds., *Logic and Data Bases* (Plenum Press, New York, 1978) 55–76.
[84] ROBINSON, J.A., A machine-oriented logic based on the resolution principle, *J. ACM* **12** (1) (1965) 23–41.

[85] ROBINSON, J.A. and E.E. SIEBERT, LOGLISP: motivation, design and implementation, in: K.L. Clark and S.-A. Tärnlund, eds., *Logic Programming* (Academic Press, New York, 1982) 299–313.

[86] SEBELIK, J. and P. STEPANEK, Horn clause programs for recursive functions, in: K.L. Clark and S.-A. Tärnlund, eds., *Logic Programming* (Academic Press, New York, 1982) 324–340.

[87] SHAPIRO, E.Y., A subset of concurrent PROLOG and its interpreter, Tech. Report TR-003, ICOT, Tokyo, 1983.

[88] SHEPHERDSON, J.C., Negation as failure: a comparison of Clark's completed data base and Reiter's closed world assumption, *J. Logic Programming* **1** (1) (1984) 51–79.

[89] SHEPHERDSON, J.C., Undecidability of Horn clause logic and pure PROLOG, Unpublished manuscript, 1985.

[90] SHEPHERDSON, J.C., Negation in logic programming, in J. Minker, ed. *Foundations of Deductive Databases and Logic Programming* (Morgan Kaufmann, Los Altos, CA, 1988).

[91] SHMUELI, O., Decidability and expressiveness aspects of logic queries, in: *Proc. 6th ACM SIGACT-SIGMOD Symp. on Principles of Database Systems* (1987) 237–249.

[92] SHOENFIELD, J., *Mathematical Logic* (Addison-Wesley, Reading, MA, 1967).

[93] SIEKMANN, J.H., Unification theory, *J. Symbolic Comput.* **7** (1988) 207–274.

[94] SMULLYAN, R.M., *Theory of Formal Systems*, Annals of Mathematical Studies, Vol. 47 (Princeton Univ. Press, Princeton, NJ, 1961).

[95] SONENBERG, E.A. and R. TOPOR, Logic programs and computable functions, Tech. Report 87/5, Dept. of Computer Science, Univ. of Melbourne, 1987.

[96] STERLING, L. and E.Y. SHAPIRO, *The Art of PROLOG* (MIT Press, Cambridge, MA, 1986).

[97] TARSKI, A., A lattice-theoretical fixpoint theorem and its applications, *Pacific J. Math.* **5** (1955) 285–309.

[98] TÄRNLUND, S.-A., Horn clause computability, *BIT* **17** (2) 215–226.

[99] VAN GELDER, A., Negation as failure using tight derivations for general logic programs, in: J. Minker, ed., *Foundations of Deductive Databases and Logic Programming* (Morgan Kaufman, Los Altos, CA, 1988).

[100] VIEILLE, L., A database-complete proof procedure based on SLD-resolution, in: *Proc. 4th Internat. Conf. on Logic Programming* (1987) 74–103.

[101] WALKER, A., Syllog: an approach to PROLOG for non-programmers, in: M. van Caneghem and P.H.D. Warren, eds., *Logic Programming and its Applications* (Ablex, Norwood, NJ, 1986) 32–49.

[102] WOLFRAM, D., M. MAHER and J.L. LASSEZ, A unified treatment of resolution strategies for logic programs, in: *Proc. 2nd Internat. Conf. on Logic Programming* (1984) 263–276.