Foundations of Logic Programming

Vladimir Lifschitz
Department of Computer Sciences
University of Texas
Austin, TX 78712

Abstract

This is a survey of the theory of logic programs with classical negation and negation as failure. The semantics of this class of programs is based on the notion of an answer set. The operation of Prolog is described in terms of proof search in the SLDNF calculus.

1 Introduction

In this survey, we view logic programming as a method for representing declarative knowledge. To put the subject in a proper perspective, we briefly discuss here two other approaches to knowledge representation and compare them with logic programming.

1.1 Relational Databases and First-Order Theories

We would like to represent facts about the parenthood relations among several members of Britain's royal family: the Queen, the Prince and Princess of Wales, and their children. These facts can be represented as a *relational database*, as follows:

Relation *Mother*

PARENT	CHILD
Elizabeth	*Charles*
Diana	*William*
Diana	*Harry*

Relation *Father*

PARENT	CHILD
Charles	*William*
Charles	*Harry*

We will call this database DB_1.

Alternatively, these facts can be encoded as a *first-order theory*. The names of the five individuals involved are the object constants of this theory T_1; *Mother* and

Father are its binary predicate constants. The first group of axioms of T_1 tells us that the domain of reasoning consists of the five distinct objects represented by the object constants:

$$\forall x(x = Elizabeth \vee x = Charles \vee x = Diana \vee x = William \vee x = Harry), \quad (1)$$

$$Elizabeth \neq Charles,\ Elizabeth \neq Diana,\ \ldots,\ William \neq Harry. \quad (2)$$

Axiom (1) is said to express the "domain closure assumption," and axioms (2) express the "unique name assumption." The second group of axioms characterizes the predicates *Mother* and *Father*:

$$\begin{aligned}
\forall xy[Mother(x,y) \equiv\ & (x = Elizabeth \wedge y = Charles)\\
& \vee(x = Diana \wedge y = William)\\
& \vee(x = Diana \wedge y = Harry)],\\
\forall xy[Father(x,y) \equiv\ & (x = Charles \wedge y = William)\\
& \vee(x = Charles \wedge y = Harry)].
\end{aligned} \quad (3)$$

Problem 1.1. Prove that T_1 is consistent.

Problem 1.2. Prove that T_1 is complete.

There is a simple connection between the relational database DB_1 and the first-order theory T_1. Let us agree to identify DB_1 with the set of atomic sentences corresponding to the lines of its tables:

$$\begin{aligned}
& Mother(Elizabeth, Charles),\\
& Mother(Diana, William),\\
& Mother(Diana, Harry),\\
& Father(Charles, William),\\
& Father(Charles, Harry).
\end{aligned} \quad (4)$$

Then any atomic sentence in the language of T_1 other than an equality is provable in T_1 if it belongs to DB_1, and refutable otherwise.

Problem 1.3. Prove this assertion. Would it be true without axioms (1) in the axiom set of T_1? Without axioms (2)?

Theory T_1 can be further extended by axioms defining some new predicates in terms of *Mother* and *Father*, for instance:

$$\forall xy(Parent(x,y) \equiv Mother(x,y) \vee Father(x,y)), \quad (5)$$

$$\forall x(Childless(x) \equiv \neg\exists y Parent(x,y)), \quad (6)$$

$$\forall xy(Grandparent(x,y) \equiv \exists z(Parent(x,z) \wedge Parent(z,y))). \quad (7)$$

We can also add the predicate *Male* and the axioms

$$\begin{aligned}
& \forall xy(Father(x,y) \supset Male(x)),\\
& \forall xy(Mother(x,y) \supset \neg Male(x)).
\end{aligned} \quad (8)$$

Note that the axioms for *Male* are different from the others in that they do not provide an explicit definition of the new predicate; they only give a sufficient condition and a necessary condition. Accordingly, the first-order theory obtained at this stage is not complete.

Problem 1.4. Show that the sentences $Male(William)$ and $Male(Harry)$ are neither provable nor refutable from the axioms introduced above.

1.2 Logic Programs

Now we will turn to our main subject—representing declarative knowledge by logic programs. A logic program consists of *rules*. A rule has two parts: the *head* and the *body*. If the body is nonempty, then we separate it from the head by the symbol $\leftarrow$ ("if"):

$$Head \leftarrow Body.$$

The precise syntax of the head and the body and the semantics of programs will be defined later. Here we only give a few examples.

The definition of *Mother*, in the notation of logic programming, consists of the following rules:

$$\begin{array}{l} Mother(Elizabeth, Charles), \\ Mother(Diana, William), \\ Mother(Diana, Harry), \\ \neg Mother(x,y) \leftarrow not\ Mother(x,y). \end{array} \tag{9}$$

Each of the first three rules has a particularly simple structure: Its body is empty, and its head is an atomic sentence. The head of the last rule is a negated atom, and its body is an expression containing the symbol *not*, called "negation as failure." The rule tells us that, for any individuals x and y under consideration, we can conclude $\neg Mother(x, y)$ if the program gives no evidence that $Mother(x, y)$. This rule expresses the "closed world assumption" for the predicate *Mother*. The availability of this rule allows us, for instance, to conclude $\neg Mother(Elizabeth, Elizabeth)$.

The negation as failure symbol makes logic programs "nonmonotonic." Classical logic is monotonic in the sense that adding an axiom to a first-order theory may only allow us to derive new theorems; no theorems proved earlier will be lost. Logic programs with negation as failure are different: Adding a rule to a logic program may force us to retract some of the conclusions obtained using the more limited set of rules. Consider, for instance, program (9) without the rule $Mother(Diana, Harry)$. Due to the closed world assumption, this smaller program allows us to conclude $\neg Mother(Diana, Harry)$. If we now put the rule $Mother(Diana, Harry)$ back in, this conclusion will have to be retracted. The same phenomenon is observed in relational databases: Adding a line to a database table invalidates a negative conclusion that could be obtained earlier.

The definition of *Father* is similar to (9):

$$\begin{array}{l} Father(Charles, William), \\ Father(Charles, Harry), \\ \neg Father(x,y) \leftarrow not\ Father(x,y). \end{array} \tag{10}$$

Here are the logic programming counterparts of definitions (5)–(7):

$$\begin{aligned} Parent(x,y) &\leftarrow Mother(x,y), \\ Parent(x,y) &\leftarrow Father(x,y), \\ \neg Parent(x,y) &\leftarrow not\ Parent(x,y), \end{aligned} \tag{11}$$

$$\begin{aligned} \neg Childless(x) &\leftarrow Parent(x,y), \\ Childless(x) &\leftarrow not\ \neg Childless(x), \end{aligned} \tag{12}$$

$$\begin{aligned} Grandparent(x,y) &\leftarrow Parent(x,z), Parent(z,y), \\ \neg Grandparent(x,y) &\leftarrow not\ Grandparent(x,y). \end{aligned} \tag{13}$$

The definition of *Childless* is different from the others in that it includes the closed world assumption for the *negation* of the defined predicate: If the program gives no evidence that $\neg Childless(x)$ then we can conclude $Childless(x)$.

The counterpart of (8) in logic programming is

$$\begin{aligned} Male(x) &\leftarrow Father(x,y), \\ \neg Male(x) &\leftarrow Mother(x,y). \end{aligned} \tag{14}$$

Note that there is no closed world assumption here.

Here is one more example of a logic programming definition:

$$\begin{aligned} Ancestor(x,y) &\leftarrow Parent(x,y), \\ Ancestor(x,y) &\leftarrow Ancestor(x,z), Ancestor(z,y), \\ \neg Ancestor(x,y) &\leftarrow not\ Ancestor(x,y). \end{aligned} \tag{15}$$

Ancestor occurs both in the head and in the body of the second rule, so that this definition is "recursive." It has no counterpart in first-order logic. The axiom

$$\forall xy[Ancestor(x,y) \equiv (Parent(x,y) \vee \exists z(Ancestor(x,z) \wedge Ancestor(z,y)))] \tag{16}$$

may seem promising as a translation of (15) into first-order logic; but it would not allow us to prove

$$\neg Ancestor(Elizabeth, Elizabeth) \tag{17}$$

or any other "negative" fact about *Ancestor*.

Problem 1.5. Verify that (17) is not derivable from axioms (1)–(3), (5) and (16).

1.3 Logic Programming Systems

When a body of knowledge is expressed as a logic program, *logic programming systems* can be sometimes used to answer queries on the basis of this knowledge.

Prolog (for PROgramming in LOGic) is the name of a family of logic programming systems. Here is an example of what Prolog can do. We would like to use a Prolog system to answer queries about the relations *Mother*, *Father* and *Parent*. To this end, we save rules (9)–(11) in a file, in the following form:

```
mother(elizabeth,charles).
mother(diana,william).
mother(diana,harry).

father(charles,william).
father(charles,harry).

parent(X,Y) :- mother(X,Y).
parent(X,Y) :- father(X,Y).
```

The syntax of Prolog requires that an identifier be capitalized if it represents a variable, but not otherwise; every rule should be followed by a period; the symbol $\leftarrow$ is represented by the two characters `:-`. More importantly, the closed world assumption rule is dropped from each of the definitions (9)–(11) before the program is presented to Prolog; for Prolog, such a rule is implicit in the definition of every predicate. On the other hand, Prolog does not permit any explicit references to classical negation. For this reason, our definitions of *Childless* and *Male* cannot be given to a Prolog system as directly as the others. But it is not difficult to extend Prolog so that it will know about classical negation. In the discussion of the mathematical principles of Prolog in this survey, a program is allowed to contain both kinds of negation.

When you call a Prolog system, it responds with a login message, for instance:

```
Quintus Prolog Release 3.1.2 (Sun-4, SunOS 4.1)
Copyright (C) 1990, Quintus Corporation.  All rights reserved.
2100 Geng Road, Palo Alto, California U.S.A. (415) 813-3800
```

Then you tell the system which file contains your set of rules, and the system opens and "consults" (or "compiles") it. After that, you can give the system queries, like

```
?- parent(elizabeth,charles).
```

(`?-` is the Prolog prompt) or

```
?- mother(elizabeth,harry).
```

and Prolog will answer—`yes` to the first query, and `no` to the second.

A query may contain variables; this is understood as a request to find a tuple of values of the variables that makes the query true. For example, in response to

```
?- parent(X,harry).
```

Quintus Prolog will say

```
X = diana.
```

You can ask for another solution, and the reply will be

```
X = charles.
```

If you ask for yet another answer, Prolog will reply `no`.

Sometimes Prolog does not produce an answer. If, for instance, the rules

```
ancestor(X,Y) :- parent(X,Y).
```

```
ancestor(X,Y) :- ancestor(X,Z), ancestor(Z,Y).
```

are added to the file, and the query

```
?- ancestor(william,harry).
```

is given to Quintus Prolog, then, instead of saying `no`, it will fail to terminate. Some query evaluation procedures have better termination properties than the one employed in Prolog systems. For instance, *Ancestor* queries can be successfully handled by the procedure called SLG.

1.4 About this Survey

In several ways, this survey of the foundations of logic programming is different from most others available in the literature.

1. Our focus is primarily on semantics, rather than query evaluation—or, as some would say, on the "declarative" semantics of logic programming, rather than "procedural." The mathematical foundations of Prolog are discussed here, but this subject is not given the same prominence as in most other surveys.

2. We treat programs as propositional objects; rules with variables are viewed as "schemata" that represent their ground instances.

In the next two sections, such rules appear in examples only. This is because the difference between a rule with variables and the set of its ground instances is less essential semantically than from the perspective of query evaluation.

3. From the very beginning, we consider programs that may contain classical negation. Negation as failure, conceptually more difficult, is introduced only in Section 3. Historically, however, logic programming with classical negation is a relatively recent invention. Programs without classical negation are called "normal," and their special properties are discussed in Sections 2.7 and 3.8.

4. We accept the semantics of negation as failure given by the concept of an "answer set," and do not talk at all about alternative approaches. The "well-founded model" is defined in Section 3.3 and the "completion" of a program in Section 3.8, because these concepts are closely related to answer sets, but their role as stand-alone theories of negation as failure is not discussed in any detail.

In the next section, we introduce "basic" programs that may include classical negation, but not negation as failure. Programs with negation as failure are investigated in Section 3. All these programs are propositional; programs with variables are the subject of Section 4. In Section 5, we describe two extensions of logic programs with negation as failure: disjunctive programs and default theories.

2 Basic Programs

Basic programs are programs without negation as failure. The expressive possibilities of this subclass are much too limited for meaningful applications to knowledge representation, but the study of its mathematical properties provides a necessary foundation for further discussion.

2.1 Syntax

We begin with a nonempty set $\mathbf{A}$ of symbols, called *atoms*. The choice of $\mathbf{A}$ determines the "language" of the programs under consideration. Atoms will be also called *positive literals*; a *negative literal* is an atom preceded by the classical negation symbol $\neg$. A *literal* is a positive literal or a negative literal. The set of literals will be denoted by $Lit_{\mathbf{A}}$, or simply Lit. For any atom A, the literals A and $\neg A$ are said to be *complementary*. A set of literals is *inconsistent* if it contains a complementary pair, and *consistent* otherwise.

A *basic rule* is an ordered pair $Head \leftarrow Body$, whose first member $Head$ is a literal, and whose second member $Body$ is a finite set of literals. A basic rule with the head L_0 and the body $\{L_1, \ldots, L_k\}$ can be also written as

$$L_0 \leftarrow L_1, \ldots, L_k. \tag{18}$$

If the body is empty then $\leftarrow$ can be dropped.

A *basic program* is a set of basic rules. For instance, if $\mathbf{A}$ is $\{p, q, r, s\}$ then the rules

$$\begin{array}{l} p, \\ \neg q, \\ r \leftarrow p, q, \\ \neg r \leftarrow p, \neg q, \\ s \leftarrow r, \\ s \leftarrow p, s, \\ \neg s \leftarrow p, \neg q, \neg r \end{array} \tag{19}$$

form a basic program. Here, as in many other examples, every atom is used in the program at least once. In such cases, it is not necessary to specify the set of atoms explicitly; we can describe a program simply by listing its rules, and it will be presumed that $\mathbf{A}$ is the set of all symbols used in the rules as atoms.

Note that, according to this definition, the body of a rule is a *set* of literals, rather than a list. There is no such thing as the order of literals, or the number of repetitions of a literal, in the body of a rule. Similarly, a program is a *set* of rules, and not a list.

In applications, $\mathbf{A}$ is usually the set of atomic sentences formed using some supply of object, function and predicate constants. Sets of rules in such a language are often represented by schemata that use metavariables for ground (that is, closed) terms. For instance, "schematic program" (9) stands for a set of 28 rules. Schematic rules are discussed in Section 4. For the time being, we will only observe that, in the presence of function symbols, a schematic rule can represent an infinite set of instances. For this reason, it is important that, in the definition above, a basic program is allowed to consist of infinitely many rules (although the body of each rule is required to be finite).

Here is an example of a program whose language has infinitely many atoms $p_0, p_1, \ldots$:

$$\begin{array}{ll} p_5, & \\ p_{n+1} \leftarrow p_n & \quad (n \geq 0). \end{array} \tag{20}$$

2.2 The Consequence Relation

In the development of classical logic, the "consequences" of a set of sentences Γ are usually defined as the sentences that can be derived from Γ and from some "logical axioms" using some "inference rules." To put it differently, the set of consequences of Γ is the smallest set of sentences that contains Γ and the logical axioms and is closed under the inference rules.

The definition of the consequence operator for basic programs given below is, in one way, simpler. The consequences of a program are literals. What is the counterpart of the logical axioms and inference rules when literals only are involved? We adopt the view that a literal follows from a set of literals "by pure logic" only when it *belongs* to this set, except in the trivial case when the set is inconsistent; in this last case, *every* literal follows from it. This is motivated by a simple fact of classical propositional logic: If Γ is a consistent set of literals and a literal L is a consequence of Γ then $L \in \Gamma$.

Problem 2.1. Prove this fact.

On the other hand, the elements of a program are *rules*, and not merely literals; instead of requiring that the set of consequences of a program include all its rules, we will require that it be "closed" under them.

This discussion suggests the following definitions. Let X be a set of literals. We say that X is *logically closed* if it is consistent or equals Lit. We say that X is *closed under* a basic program Π if, for every rule $Head \leftarrow Body$ in Π, $Head \in X$ whenever $Body \subseteq X$. By $\mathrm{Cn}(\Pi)$ we denote the smallest set of literals which is both logically closed and closed under Π.

Problem 2.2. Prove that such a set always exists.

The elements of $\mathrm{Cn}(\Pi)$ are called the *consequences* of Π.

A basic program Π is *consistent* if $\mathrm{Cn}(\Pi)$ is consistent, and *inconsistent* otherwise. The following fact easily follows from the definition of $\mathrm{Cn}(\Pi)$:

Proposition 2.1. *For any basic program* Π,

- *if* Π *is consistent then* $\mathrm{Cn}(\Pi)$ *is the smallest set of literals closed under* Π*;*
- *if* Π *is inconsistent then* $\mathrm{Cn}(\Pi) = Lit$.

Let us find, for instance, the consequences of program (19). The program includes two rules whose bodies are empty; it is clear that their heads

$$p, \neg q \tag{21}$$

belong to every set of literals closed under (19). Furthermore, the program includes a rule with the body $p, \neg q$; consequently, the head

$$\neg r \tag{22}$$

of this rule belongs to every such set also. The next step is to observe that there is one more rule in the program whose body consists of the literals that are all among (21) and (22); its head

$$\neg s \tag{23}$$

also belongs to every set closed under the rules of (19). The set of literals generated by now, (21)–(23), is both logically closed and closed under (19). It follows that this is the set of all consequences of (19).

Problem 2.3. Find the set of consequences of program (20).

Note that replacing the rule

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

in (19) by a "contrapositive" rule

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

would change the set of consequences of the program: (23) would not be among the consequences anymore. Unlike conditionals in classical logic, a basic rule is, generally, not equivalent to its contrapositive.

Proposition 2.2. *For any basic programs* Π_1 *and* Π_2, *if* $\Pi_1 \subseteq \Pi_2$ *then* $\mathrm{Cn}(\Pi_1) \subseteq \mathrm{Cn}(\Pi_2)$.

The monotonicity of Cn is related to the fact that basic programs do not use negation as failure. The consequence operator has also the following *compactness property*:

Proposition 2.3. *Every consequence of a basic program* Π *is a consequence of a finite subset of* Π.

Problem 2.4. Verify this assertion for program (20).

Let Π be a basic program, and X a set of literals. We say that X is *supported* by Π if, for each literal $L \in X$, there exists a rule $\mathit{Head} \leftarrow \mathit{Body}$ in Π such that $\mathit{Head} = L$ and $\mathit{Body} \subseteq X$. A rule satisfying these conditions "supports" the presence of L in X; it provides a "reason" for including L in X.

Proposition 2.4. *For any consistent basic program* Π, $\mathrm{Cn}(\Pi)$ *is supported by* Π.

Problem 2.5. Verify this assertion for programs (19) and (20).

Problem 2.6. Show that without the consistency assumption this assertion would be incorrect.

A *head literal* of a basic program Π is the head of a rule of Π.

Corollary. *If a basic program* Π *is consistent then every consequence of* Π *is a head literal of* Π.

There is a simple syntactic sufficient condition for consistency. A basic program is *head-consistent* if the set of its head literals is consistent.

Proposition 2.5. *Every head-consistent basic program is consistent.*

For instance, (20) is head-consistent. Program (19) is consistent but not head-consistent.

2.3 Bottom-Up Evaluation

Computing the set of consequences of program (19) in Section 2.2 is an example of the process of "bottom-up evaluation" that can be applied to any basic program. In order to describe this process, we define, for any basic program Π, the function T_Π from sets of literals to sets of literals, as follows: $T_\Pi X$ is

$$\{Head \, : \, Head \leftarrow Body \in \Pi, \, Body \subseteq X\}$$

if X is consistent, and *Lit* otherwise. Thus $T_\Pi X$ is the set of literals that can be derived from X "in one step" using either a rule of Π or "pure logic."

It is clear that T_Π is monotone. The discussion of this function below uses the terminology and results of the theory of monotone functions reviewed in Appendix.

There is a simple relationship between the pre-fixpoints of T_Π and the two closure properties defined in Section 2.2:

Proposition 2.6. *For any basic program* Π *and any set of literals* X, X *is a pre-fixpoint of* T_Π *iff* X *is both logically closed and closed under* Π.

Using Proposition A.1, we conclude:

Corollary. $\mathrm{Cn}(\Pi)$ *is the least fixpoint of* T_Π.

According to Proposition A.2, the union of the sets obtained by iterating T_Π on the empty set is a subset of the least fixpoint of T_Π. For this particular function, the union happens to be *equal* to this fixpoint:

Proposition 2.7. *For any basic program* Π,

$$\mathrm{Cn}(\Pi) = \bigcup_{n \geq 0} T_\Pi^n \emptyset.$$

For example, if Π is (19) then

$$\begin{aligned}
&T_\Pi^0 \emptyset = \emptyset, \\
&T_\Pi^1 \emptyset = \{p, \neg q\}, \\
&T_\Pi^2 \emptyset = \{p, \neg q, \neg r\}, \\
&T_\Pi^3 \emptyset = \{p, \neg q, \neg r, \neg s\}.
\end{aligned}$$

The last set is a fixpoint of T_Π, so that, for every $n > 3$, $T^n_\Pi\emptyset = T^3_\Pi\emptyset$. According to Proposition 2.7,

$$\mathrm{Cn}(\Pi) = \{p, \neg q, \neg r, \neg s\}.$$

If Π is infinite then it is possible that none of the sets $T^n_\Pi\emptyset$ is a fixpoint of Π, so that every next term of the sequence adds new consequences to the set accumulated earlier.

Problem 2.7. Find the sets $T^n_\Pi\emptyset$ for program (20).

The process of bottom-up evaluation is different from the process of computation used in Prolog—the latter is "goal-directed." The operation of Prolog for basic programs is discussed in Section 2.6.

2.4 Splitting

Computing the consequences of a program can be sometimes simplified by "splitting" it into parts.

We say that a set U of literals *splits* a basic program Π if, for every rule $Head \leftarrow Body$ in Π, $Body \subseteq U$ whenever $Head \in U$. If U splits Π then the set of rules in Π whose heads belong to U will be called the *base* of Π (relative to U), and denoted by $b_U(\Pi)$.

For example, the set $U = \{p, q, \neg q, r\}$ splits program (19), and the base consists of the first three rules of the program:

$$\begin{array}{l} p, \\ \neg q, \\ r \leftarrow p, q. \end{array} \tag{24}$$

Problem 2.8. A literal *occurs* in a basic program Π if it is the head of a rule of Π or belongs to the body of a rule of Π. A splitting set U for a basic program Π is *trivial* if U contains no literals occurring in Π or contains all of them. Give an example of a basic program that consists of 100 rules and has no nontrivial splitting sets.

If U splits Π then, according to Proposition 2.8 stated below, the consequences of Π can be computed in two steps. First, we find the set C of the consequences of the base $b_U(\Pi)$. Second, C is used to eliminate the elements of U from the remaining rules of the program: If $L \in C$ then L is "trivial" and can be deleted from the bodies of the remaining rules; if $L \in U \setminus C$ then every rule with L in the body is "useless" and can be deleted as a whole. After that, we find the consequences of the resulting program and append them to C.

In the example, $C = \{p, \neg q\}$ and $U \setminus C = \{q, r\}$. The complement of the base is

$$\begin{array}{l} \neg r \leftarrow p, \neg q, \\ s \leftarrow r, \\ s \leftarrow p, s, \\ \neg s \leftarrow p, \neg q, \neg r. \end{array}$$

We drop the rule with r in the body and delete p and $\neg q$ from the bodies of the remaining rules. The result is

$$\begin{aligned}
&\neg r,\\
&s \leftarrow s,\\
&\neg s \leftarrow \neg r.
\end{aligned}$$

The set of consequences of this program is $\{\neg r, \neg s\}$. The union $C \cup \{\neg r, \neg s\}$ is the set of consequences of (19).

In general, the elimination process can be described using the following notation. For any basic program Π, any set U of literals and any subset C of U, $e_U(\Pi, C)$ stands for the basic program obtained from Π by

- deleting each rule $Head \leftarrow Body$ such that $Body \cap (U \setminus C) \neq \emptyset$, and
- replacing each remaining rule $Head \leftarrow Body$ by $Head \leftarrow (Body \setminus U)$.

Here is a theorem expressing the soundness of the splitting method for computing $\mathrm{Cn}(\Pi)$:

Proposition 2.8. *Let U be a set of literals that splits a basic program Π, and let $C = \mathrm{Cn}(b_U(\Pi))$. If the set $C \cup \mathrm{Cn}(e_U(\Pi \setminus b_U(\Pi), C))$ is consistent then it equals $\mathrm{Cn}(\Pi)$; otherwise, Π is inconsistent.*

Problem 2.9. Verify the assertion of Proposition 2.8 for the case when U is trivial (as defined in Problem 2.8).

Problem 2.10. Verify the assertion of Proposition 2.8 for program (19) and $U = \{q\}$.

Problem 2.11. Show that the first assertion of Proposition 2.8 would be incorrect without the consistency assumption.

2.5 The SLD Calculus

Consider the program

$$\begin{aligned}
&p \leftarrow q,\\
&q,\\
&r \leftarrow s,\\
&r \leftarrow p, q.
\end{aligned} \tag{25}$$

We would like to find out whether r is a consequence of this program. The program has two rules with the head r. The first of them shows that r is a consequence of the program if s is. There is no evidence, however, that s is a consequence, because the program does not contain rules with s in the head. Let us try the second rule with the head r. It shows that r is a consequence if both p and q are consequences. The only rule with the head p shows that p is a consequence if q is. It remains to determine whether q is a consequence of the program. The answer to this question is yes, because q is the head of a rule with the empty body. We conclude that r is a consequence of (25).

The reasoning that has led us to this conclusion can be symbolically represented as follows:

$$\dfrac{\dfrac{\dfrac{\models \emptyset}{\models \{q\}}}{\models \{p, q\}}}{\models \{r\}} \tag{26}$$

The sign $\models$ here expresses "success."

Let us determine now whether r is a consequence of the program obtained from (25) by adding s to the body of the second rule:

$$\begin{array}{l} p \leftarrow q, \\ q \leftarrow s, \\ r \leftarrow s, \\ r \leftarrow p, q. \end{array} \tag{27}$$

As before, the attempt to establish that r is a consequence by using the first rule with the head r fails, and the second rule leads us to the question whether q is a consequence. The program contains one rule with the head q, and its body is s. Consequently, we need to determine whether s is a consequence. Since s is not the head of any rule, it appears that the answer to this question must be no, so that r is not a consequence of (27).

The reasoning that has led us to this conclusion can be symbolically represented by a tree:

$$\dfrac{\dfrac{}{\Dashv \{s\}} \qquad \dfrac{\dfrac{\dfrac{}{\Dashv \{s\}}}{\Dashv \{q\}}}{\Dashv \{p, q\}}}{\Dashv \{r\}} \tag{28}$$

The sign $\Dashv$ expresses "failure."

Figures (26) and (28) are examples of derivations in the "SLD calculus" that we will now introduce.[1]

A *goal* is a finite set of literals. In the *SLD calculus* for a basic program Π, the derivable objects are expressions of the forms $\models G$ and $\Dashv G$, where G is a goal. The

[1] The letters S, L and D are the initial letters of the words "selection," "linear" and "definite"—the names of three ideas that were historically associated with goal-directed search in logic programming. The notion of a definite clause and its relation to logic programs are discussed in Section 2.7 below. The other two words come from the theory of resolution in automated reasoning.

only axiom is

$$\models \emptyset.$$

In the inference rules, the following notation is used: For any literal L, $Bodies(L)$ is the set of the bodies of all rules in Π with the head L. There are two inference rules, one for proving success and one for proving failure:

$$(S)\ \frac{\models G \cup B}{\models G \cup \{L\}} \qquad \text{if } B \in Bodies(L)$$

$$(F)\ \frac{\dashv G \cup B \qquad \text{for all } B \in Bodies(L)}{\dashv G \cup \{L\}}$$

Note that the number of premises of (F) equals the cardinality of $Bodies(L)$. In particular, it can be zero or infinite.[2]

Figure (26) is a derivation in the SLD calculus for program (25). Every horizontal bar in this figure represents an application of (S). For instance, the transition from $\models \{p, q\}$ to $\models \{r\}$ at the end of the derivation is an application of (S) with $G = \emptyset$, $L = r$ and $B = \{p, q\}$. Here $B \in Bodies(L)$ because

$$Bodies(r) = \{\{s\}, \{p, q\}\}.$$

Figure (28) is a derivation in the SLD calculus for (27), in which every bar is an application of (F). The bars on top of the two branches of (28) represent applications of (F) with zero premises.

If $\models G$ is derivable in the SLD calculus for Π then we say that G *succeeds* relative to Π. If $\dashv G$ is derivable then we say that G *fails* relative to Π. For instance, $\{r\}$ succeeds relative to (25) and fails relative to (27).

Problem 2.12. Show that the goal $\{q\}$ fails relative to the program that consists of the rules

$$\begin{array}{l} q \leftarrow p_n, \\ p_{n+1} \leftarrow p_n \end{array} \tag{29}$$

$(n \geq 0)$.

Proposition 2.10. *For any basic program Π, no goal both succeeds and fails relative to Π.*

Problem 2.13. Show that the goal $\{p\}$ neither succeeds nor fails relative to the program

$$p \leftarrow p. \tag{30}$$

The following proposition expresses the soundness of the SLD calculus.

[2] For a calculus with infinitary rules, a derivation is defined as a (possibly infinite) tree without infinite branches, whose nodes are derivable objects such that every node either (a) is a leaf and an axiom, or (b) can be obtained by applying an inference rule to its successor nodes.

Proposition 2.11. *For any basic program* Π *and any literal* L,

- *if* $\{L\}$ *succeeds relative to* Π *then* L *is a consequence of* Π,
- *if* Π *is consistent and* $\{L\}$ *fails relative to* Π *then* L *is not a consequence of* Π.

This proposition is applicable to programs (25) and (27), because they are head-consistent and consequently consistent (Proposition 2.5).

Note that the SLD calculus may be unsound for failure if the program is inconsistent. For instance, if Π is inconsistent and L is not a head literal of Π then L is a consequence of Π that fails: $\dashv \{L\}$ can be derived by one application of the failure rule to the empty set of premises.

If the program is consistent then the success rule of the SLD calculus is complete:

Proposition 2.12. *For any consistent basic program* Π *and any consequence* L *of* Π, $\{L\}$ *succeeds relative to* Π.

The failure rule is generally incomplete even for consistent programs. For instance, $\{p\}$ does not fail with respect to (30). We will now define a syntactic property of basic programs that guarantees the completeness of the SLD calculus for failure.

A *level mapping* is a function from literals to ordinals. A basic program Π is said to be *hierarchical* if there exists a level mapping λ such that, for every rule $Head \leftarrow Body$ in Π,

$$\lambda(Head) > \max_{L \in Body} \lambda(L). \tag{31}$$

For instance, programs (25) and (27) are hierarchical: for each of them, we can take

$$\lambda(s) = 0,\ \lambda(q) = 1,\ \lambda(p) = 2,\ \lambda(r) = 3.$$

Program (20) is hierarchical also: take $\lambda(p_n) = n$. To show that (29) is hierarchical, define

$$\lambda(p_n) = n,\ \lambda(q) = \omega.$$

Program (30) is not hierarchical because, for this program, condition (31) turns into $\lambda(p) > \lambda(p)$. The programs

$$\begin{array}{l} p \leftarrow q, \\ q \leftarrow p \end{array}$$

and

$$p_n \leftarrow p_{n+1} \qquad (n \geq 0)$$

are not hierarchical either.

Problem 2.14. Determine whether program (19) is hierarchical.

Here is a completeness theorem for the failure rule:

Proposition 2.13. *For any hierarchical basic program* Π *and any literal* L *that is not a consequence of* Π, $\{L\}$ *fails with relative to* Π.

2.6 Propositional Prolog

The work of Prolog, for a program without negation and variables, can be viewed as an attempt to establish that a goal G succeeds or that it fails (to "evaluate" G) by constructing a derivation of one of the expressions $\models G$, $=\!\!| \, G$ in the SLD calculus using "backward chaining." This kind of search was used in the examples at the beginning of Section 2.5.

Here is a general description of the operation of Prolog. The task is to evaluate a goal G relative to a finite basic program Π. If G is empty then $\models G$ is an axiom. Otherwise, in order to find a rule application that leads to one of the expressions $\models G$, $=\!\!| \, G$, the goal G is represented in the form $G' \cup \{L\}$; L is called the "selected subgoal" of G. Prolog attempts to evaluate, one by one, the goals $G' \cup B$ for all $B \in \mathit{Bodies}(L)$. If at least one of these goals succeeds then $\models G$ can be derived by (S). If they all fail then $=\!\!| \, G$ can be derived by (F).

At the beginning of this process, the given program is represented as the list of its rules, with the body of every rule represented as the list of its elements. Similarly, the goal G is represented as a list. The order in which the elements of all these lists are initially arranged can affect the search process. Specifically, it determines how subgoals are selected (Prolog tries the leftmost element of G first) and in what order the elements of $\mathit{Bodies}(L)$ are considered when the bottom-up application of (S) or (F) is attempted. Ordering atoms and rules in a logic program "in the right way" is an important part of the art of Prolog programming.

Consider, for instance, the operation of Prolog on program (25) and the goal $\{r\}$. The initial stage of the process can be symbolically represented by the expression

$$\perp\!\!\!\perp \{r\}.$$

Here $\perp\!\!\!\perp$ is a label that will be replaced by either $\models$ or $=\!\!|$ at the end of the computation. This expression is the first in the chain of "partial derivations" that describes the operation of Prolog on the given goal. The first element of $\mathit{Bodies}(r)$ is $\{s\}$, so that the next partial derivation is

$$\frac{\perp\!\!\!\perp \{s\}}{\perp\!\!\!\perp \{r\}}$$

Then we observe that $\mathit{Bodies}(s)$ is empty, so that $=\!\!| \, \{s\}$ can be derived by one application of (F) to the empty set of premises. This leads us to the partial derivation in which the label $\perp\!\!\!\perp$ in front of $\{s\}$ is replaced by the failure symbol:

$$\frac{(F)\ \dfrac{}{=\!\!| \ \{s\}}}{\perp\!\!\!\perp \{r\}} \tag{32}$$

It remains to evaluate the second element of $\mathit{Bodies}(r)$, that is, $\{p, q\}$. First we form the partial derivation

$$\dfrac{(F)\,\dfrac{}{\mathrel{=\!\!|}\ \{s\}} \qquad \perp\!\!\!\perp\ \{p,q\}}{\perp\!\!\!\perp\ \{r\}}$$

As soon as we know whether $\perp\!\!\!\perp$ in front of $\{p,q\}$ turns into $\models$ or $\mathrel{=\!\!|}$, the evaluation of $\{r\}$ will be completed. Specifically, if a derivation of $\mathrel{=\!\!|}\{p,q\}$ is found then it will be appended to (32) to form a derivation of $\mathrel{=\!\!|}\{r\}$:

$$(F)\,\dfrac{(F)\,\dfrac{}{\mathrel{=\!\!|}\ \{s\}} \qquad (F)\,\dfrac{\cdots}{\mathrel{=\!\!|}\ \{p,q\}}}{\mathrel{=\!\!|}\ \{r\}}$$

If, on the other hand, a derivation of $\models\ \{p,q\}$ is found then we will discard the derivation of $\mathrel{=\!\!|}\{s\}$ in (32) and obtain a derivation of $\models\{r\}$:

$$(S)\,\dfrac{(S)\,\dfrac{\cdots}{\models\ \{p,q\}}}{\models\ \{r\}}$$

In the process of evaluation of $\{p,q\}$, Prolog designates p as the selected subgoal, because it is the leftmost element of the list p,q. The next partial derivation is

$$\dfrac{(F)\,\dfrac{}{\mathrel{=\!\!|}\ \{s\}} \qquad \dfrac{\perp\!\!\!\perp\ \{q,q\}}{\perp\!\!\!\perp\ \{p,q\}}}{\perp\!\!\!\perp\ \{r\}} \tag{33}$$

In order to remain faithful to the actual operation of Prolog, it is important at this stage not to remove repetitions in the expression $\{q,q\}$. This expression stands for a singleton set that is represented as a list of length 2.

On the next step, the first member of the list q,q is selected, so that $L=q$ and $G'=G=\{q\}$. The next partial derivation is

$$\dfrac{(F)\,\dfrac{}{\mathrel{=\!\!|}\ \{s\}} \qquad \dfrac{\dfrac{\perp\!\!\!\perp\ \{q\}}{\perp\!\!\!\perp\ \{q,q\}}}{\perp\!\!\!\perp\ \{p,q\}}}{\perp\!\!\!\perp\ \{r\}}$$

After that, we form the partial derivation

$$\frac{(F)\,\dfrac{}{\mathrel{=\!\!|}\ \{s\}} \qquad \dfrac{\dfrac{\dfrac{\perp\!\!\!\perp\ \emptyset}{\perp\!\!\!\perp\ \{q\}}}{\perp\!\!\!\perp\ \{q,q\}}}{\perp\!\!\!\perp\ \{p,q\}}}{\perp\!\!\!\perp\ \{r\}} \tag{34}$$

Here we recognize that $\models \emptyset$ is an axiom. All occurrences of the label $\perp\!\!\!\perp$ in the right branch of (34) are replaced by $\models$, and the left branch is discarded. The final product of this search process is the derivation

$$(S)\,\dfrac{(S)\,\dfrac{(S)\,\dfrac{(S)\,\dfrac{\models \emptyset}{\models \{q\}}}{\models \{q\}}}{\models \{p,q\}}}{\models \{r\}} \tag{35}$$

Note that this derivation includes a redundant step that is absent from (26). The difference is due to the fact that Prolog does not check the list representations of goals for repetitions.

If we start with program (27) instead of (25) then the first several partial derivations constructed in the process of evaluating $\{r\}$ are going to be the same, up to (33). After that, the partial derivation

$$\frac{(F)\,\dfrac{}{\mathrel{=\!\!|}\ \{s\}} \qquad \dfrac{\dfrac{\perp\!\!\!\perp\ \{s,q\}}{\perp\!\!\!\perp\ \{q,q\}}}{\perp\!\!\!\perp\ \{p,q\}}}{\perp\!\!\!\perp\ \{r\}}$$

will be formed. Then s will be selected, and $\mathrel{=\!\!|}\{s,q\}$ will be derived from the empty set of premises. Having replaced each $\perp\!\!\!\perp$ by $\mathrel{=\!\!|}$ in the right branch of this partial derivation, we will get the derivation

$$(F)\,\dfrac{(F)\,\dfrac{}{\dashv \{s\}} \qquad (F)\,\dfrac{(F)\,\dfrac{(F)\,\dfrac{}{\dashv \{s,q\}}}{\dashv \{q\}}}{\dashv \{p,q\}}}{\dashv \{r\}}$$

The goal $\{r\}$ has failed.

Problem 2.15. Describe the work of Prolog on the program

$$\begin{array}{l} p \leftarrow q, r \\ p \leftarrow r, s \\ r \leftarrow u, v \\ s \\ u \end{array}$$

and the goal $\{p\}$. How would the process be affected by appending v to the program as an additional rule?

Problem 2.16. Describe the work of Prolog on the program

$$\begin{array}{l} p \leftarrow p, \\ p \end{array}$$

and the goal $\{p\}$.

The last example demonstrates the incompleteness of the Prolog search strategy: Prolog never finds the one-step derivation of $\models \{p\}$ in the SLD calculus for this program. This is similar to the case of nontermination of Prolog mentioned at the end of Section 1.3.

2.7 Normal Programs

A basic rule or program that does not contain the negation symbol $\neg$ is said to be *normal*. Many programs that we have used as examples are normal. Normal basic programs are head-consistent. Consequently, any normal basic program is consistent (Proposition 2.5), and any consequence of such a program is an atom (Corollary to Proposition 2.4).

Arbitrary basic programs can be reduced to normal basic programs in the following way. For every atom $A \in \mathbf{A}$, select a new symbol A', and let $\mathbf{A}'$ be the set of these new symbols. For any $L \in Lit_{\mathbf{A}}$, let $\text{Norm}(L)$ be the symbol from $\mathbf{A} \cup \mathbf{A}'$ defined as follows:

$$\text{Norm}(A) = A, \ \text{Norm}(\neg A) = A' \qquad (A \in \mathbf{A}).$$

The map Norm is extended to sets of literals, basic rules and basic programs in a natural way:

$$\begin{array}{l}\mathrm{Norm}(X) = \{\mathrm{Norm}(L) \,:\, L \in X\},\\ \mathrm{Norm}(\mathit{Head} \leftarrow \mathit{Body}) = \mathrm{Norm}(\mathit{Head}) \leftarrow \mathrm{Norm}(\mathit{Body}),\\ \mathrm{Norm}(\Pi) = \{\mathrm{Norm}(R) \,:\, R \in \Pi\}.\end{array}$$

Thus, to transform Π into $\mathrm{Norm}(\Pi)$, we simply replace every negative literal $\neg A$ in Π by A'. It is clear that Norm is a one-to-one map of the set of basic programs onto the set of normal basic programs over the extended set of atoms $\mathbf{A} \cup \mathbf{A}'$.

If, for example, Π is (19), then $\mathrm{Norm}(\Pi)$ is

$$\begin{array}{l} p,\\ q',\\ r \leftarrow p, q,\\ r' \leftarrow p, q',\\ s \leftarrow r,\\ s \leftarrow p, s,\\ s' \leftarrow p, q', r'. \end{array} \tag{36}$$

By *Contr* we denote the set of *contradiction rules*

$$\begin{array}{l} A \leftarrow B, B',\\ A' \leftarrow B, B' \end{array}$$

for all pairs of distinct atoms $A, B \in \mathbf{A}$.

Proposition 2.14. *For any basic program* Π,

$$\mathrm{Norm}(\mathrm{Cn}(\Pi)) = \mathrm{Cn}(\mathrm{Norm}(\Pi) \cup \mathit{Contr}).$$

Moreover, if Π *is consistent then*

$$\mathrm{Norm}(\mathrm{Cn}(\Pi)) = \mathrm{Cn}(\mathrm{Norm}(\Pi)).$$

Problem 2.17. Show that the second assertion would not be correct without the consistency assumption.

Proposition 2.14 shows that Norm is a one-to-one correspondence between the consequences of Π and the consequences of the normal basic program $\mathrm{Norm}(\Pi) \cup \mathit{Contr}$; if Π is consistent then the contradiction rules can be dropped.

A normal basic program can be encoded by a propositional formula. The function φ from normal basic rules to propositional formulas is defined by

$$\varphi(\mathit{Head} \leftarrow \mathit{Body}) \;=\; \bigwedge_{A \in \mathit{Body}} A \;\supset\; \mathit{Head}.$$

For any normal basic program Π, $\varphi\Pi$ stands for the set of formulas φR for all rules $R \in \Pi$. For example, if Π is (36) then $\varphi\Pi$ consists of the formulas

$$\begin{array}{l} p, \\ q', \\ p \wedge q \supset r, \\ p \wedge q' \supset r', \\ r \supset s, \\ p \wedge s \supset s, \\ p \wedge q' \wedge r' \supset s'. \end{array}$$

The following proposition describes the relationship between the set $\varphi\Pi$ and the concept of closure under Π defined in Section 2.2. Recall that, in propositional logic, an *interpretation* is a function from atoms to truth values, and a *model* of a set of formulas is an interpretation that satisfies all formulas in the set. We will identify an interpretation with the set of atoms to which it assigns the value *true*.

Proposition 2.15. *For any normal basic program* Π, *an interpretation* I *is a model of* $\varphi\Pi$ *iff* I *is closed under* Π.

Corollary 1. *For any normal basic program* Π, $\mathrm{Cn}(\Pi)$ *is the least model of* $\varphi\Pi$.

Corollary 2. *For any normal basic program* Π, $\mathrm{Cn}(\Pi)$ *is the set of atoms entailed by* $\varphi\Pi$.

A *clause* is a disjunction of literals. A clause is *definite* if exactly one of its literals is positive. It is clear that, for any normal basic rule R, the formula φR is equivalent to a definite clause, and, conversely, every (propositional) definite clause is equivalent to a formula of this form. Corollaries 1 and 2 show that recognizing the consequences of a normal basic program amounts to recognizing the atoms that belong to the least model of a set of definite clauses, or, equivalently, the atoms that are entailed by such a set.

3 Negation as Failure

Now we turn to the study of logic programs with negation as failure. The examples discussed in Section 1.2 are (schematic representations of) programs of this kind.

3.1 Answer Sets

A *rule element* is a literal possibly preceded by the negation as failure symbol *not*. A *rule* is an ordered pair $Head \leftarrow Body$, whose first member $Head$ is a literal, and whose second member $Body$ is a finite set of rule elements. For any set X of literals, we will denote the set $\{not\ L : L \in X\}$ by $not(X)$. Then any rule can be represented as

$Head \leftarrow Pos \cup not(Neg)$, for some finite sets of literals Pos, Neg. The rule with the head L_0 and the body $\{L_1, \dots, L_m, not\ L_{m+1}, \dots, not\ L_n\}$ will be also written as

$$L_0 \leftarrow L_1, \dots, L_m, not\ L_{m+1}, \dots, not\ L_n. \tag{37}$$

A *program* is a set of rules. For instance,

$$\begin{array}{l} p, \\ q \leftarrow p, not\ r, \\ q \leftarrow r, not\ p, \\ r \leftarrow p, not\ s \end{array} \tag{38}$$

is a program. This program does not contain the classical negation symbol $\neg$; the syntax of rules allows us to insert this symbol in front of any of the atoms p, q, r, s anywhere in the program.

We would like to generalize the definition of $\mathrm{Cn}(\Pi)$ from Section 2.2 to arbitrary programs.

Intuitively, the presence of a rule element $not\ L$ in the body of a rule limits the applicability of the rule to the case when the program as a whole provides no possibilities for deriving L. For instance, rules (38) differ from the basic rules

$$\begin{array}{l} p, \\ q \leftarrow p, \\ q \leftarrow r, \\ r \leftarrow p \end{array} \tag{39}$$

in that

- the second rule of (38) allows us to derive q from p only if r cannot be derived,
- the third rule of (38) allows us to derive q from r only if p cannot be derived,
- the last rule of (38) allows us to derive r from p only if s cannot be derived.

This informal description of how the symbol *not* "blocks" the application of program rules is circular, because it characterizes the process of applying rules in terms of what can be derived using these rules. Nevertheless, for any set X of literals, that description makes it possible to "test" the claim that rules (38) allow us to derive the elements of X and nothing else.

Take, for instance, X to be $\{p, r\}$. If p and r are indeed derivable, and the other literals are not, then the second rule of (38) is "blocked" in view of the presence of $not\ r$ in its body, and the third rule is "blocked" by the presence of $not\ p$; the other two rules are not "blocked." Then the effect of rules (38) is the same as the effect of

$$\begin{array}{l} p, \\ r \leftarrow p \end{array} \tag{40}$$

—the subset of (39) obtained by deleting its second and third rules. This is a basic program. The set of its consequences is $\{p, r\}$, which is exactly the set X that we initially assumed to be the set of derivable literals. This fact confirms that $\{p, r\}$ was a "good guess."

Generally, there can be several "good guesses" about the result of application of a given set of rules. Consider, for instance, the program

$$\begin{array}{l} p \leftarrow \mathit{not}\ q, \\ q \leftarrow \mathit{not}\ p, \\ r \leftarrow p, \\ r \leftarrow q. \end{array} \tag{41}$$

There are two reasonable conjectures about what can be derived using these rules. One is that we can derive p and r, but not q. If so, then (41) has the same meaning as the basic program

$$\begin{array}{l} p, \\ r \leftarrow p, \\ r \leftarrow q. \end{array} \tag{42}$$

The set of consequences of this program is, indeed, $\{p, r\}$. The other possibility is that q and r can be derived, but not p. In this case, (41) has the same meaning as the basic program

$$\begin{array}{l} q, \\ r \leftarrow p, \\ r \leftarrow q, \end{array} \tag{43}$$

whose consequences are, indeed, q and r.

This example leads us to the view that negation as failure can make the rules of a program "nondeterministic." There can be several "correct" ways to organize the process of deriving literals using the rules of a program that contains negation as failure. Each of them produces a different set of literals; these sets will be called the "answer sets" for the program. A consequence of a program is a literal that is guaranteed to be produced no matter which derivation process is selected—a literal that belongs to all answer sets. For instance, the only answer set for (38) is $\{p, r\}$, so that the consequences of this program are p and r; the answer sets for (41) are $\{p, r\}$ and $\{q, r\}$, so that its only consequence is r.

In order to give the definition of an answer set, we need a general description of the process of reducing an arbitrary program to a basic program that was used above to obtain (40) from (38), and (42), (43) from (41).

Let Π be a program, and X a set of literals. The *reduct of* Π *relative to* X is the basic program obtained from Π by

- deleting each rule $\mathit{Head} \leftarrow \mathit{Pos} \cup \mathit{not}(\mathit{Neg})$ such that $\mathit{Neg} \cap X \neq \emptyset$, and
- replacing each remaining rule $\mathit{Head} \leftarrow \mathit{Pos} \cup \mathit{not}(\mathit{Neg})$ by $\mathit{Head} \leftarrow \mathit{Pos}$.

This program will be denoted by Π^X. We say that X is an *answer set* for Π if $\mathrm{Cn}(\Pi^X) = X$.

It is clear that every answer set is logically closed. We have seen that a program can have one or several answer sets. Some programs have no answer sets, for instance

$$p \leftarrow \mathit{not}\ p. \tag{44}$$

A *consequence* of a program is a literal that belongs to all its answer sets. Alternatively, the consequences of a program can be characterized as the literals that belong to all its *consistent* answer sets. It is clear that the set of consequences is logically closed.

If a program Π is basic then its reduct relative to any set of literals is Π. It follows that the only answer set for a basic program Π is $\mathrm{Cn}(\Pi)$, so that the new definition of a consequence, applied to a basic program, is equivalent to the one given in Section 2.2.

For the set of consequences of a program Π, we will use the same notation $\mathrm{Cn}(\Pi)$ as in the basic case.

On programs with negation as failure, the consequence operator is not monotone. For instance, the set of consequences of $\{p \leftarrow \mathit{not}\ q\}$ is $\{p\}$; if we add q to this program as another rule, the set of consequences will be $\{q\}$. In this sense, logic programming with negation as failure is a "nonmonotonic formalism."

Problem 3.1. Find all answer sets for the program

$$\begin{array}{l} p \leftarrow \mathit{not}\ q, \\ q \leftarrow \mathit{not}\ p, \\ r \leftarrow \mathit{not}\ r, \\ r \leftarrow p. \end{array} \tag{45}$$

Problem 3.2. Find all answer sets for the program

$$p_{n+1} \leftarrow \mathit{not}\ p_n \qquad (n \geq 0). \tag{46}$$

Problem 3.3. Find all answer sets for the program

$$p_n \leftarrow \mathit{not}\ p_{n+1} \qquad (n \geq 0). \tag{47}$$

Proposition 3.1. *If X and Y are answer sets for a program Π and $X \subseteq Y$ then $X = Y$.*

Corollary. *Every program Π satisfies exactly one of the following conditions:*

- *Π has no answer sets,*
- *the only answer set for Π is Lit,*
- *Π has an answer set, and all its answer sets are consistent.*

The consistency of a program is defined as it was defined for basic programs in Section 2.2: A program is *consistent* if the set of its consequences is consistent, and *inconsistent* otherwise. In the first two cases listed in the statement of the corollary, Π is inconsistent and $\mathrm{Cn}(\Pi) = \mathit{Lit}$. In the third case, Π is consistent.

The definition of closure under a program given in Section 2.2 is extended to arbitrary programs as follows. A set X of literals is *closed under* a program Π if, for every rule $\mathit{Head} \leftarrow \mathit{Pos} \cup \mathit{not}(\mathit{Neg})$ in Π, $\mathit{Head} \in X$ whenever $\mathit{Pos} \subseteq X$ and $\mathit{Neg} \cap X = \emptyset$.

Proposition 3.2. *Every answer set for a program Π is closed under Π.*

The set of consequences of Π, however, is not necessarily closed under Π. This can be illustrated by program (41).

We say that a set X of literals is *supported* by Π if, for each literal $L \in X$, there exists a rule $\mathit{Head} \leftarrow \mathit{Pos} \cup \mathit{not}(\mathit{Neg})$ in Π such that

$$\mathit{Head} = L,\ \ \mathit{Pos} \subseteq X,\ \ \mathit{Neg} \cap X = \emptyset.$$

Proposition 2.4 can be generalized to arbitrary programs in the following way:

Proposition 3.3. *Any consistent answer set for a program Π is supported by Π.*

As in the case of basic programs, a *head literal* of a program Π is the head of a rule of Π.

Corollary 1. *Any element of any consistent answer set for a program Π is a head literal of Π.*

Corollary 2. *If a program Π is consistent then every consequence of Π is a head literal of Π.*

As in the case of basic programs, a program Π is *head-consistent* if the set of its head literals is consistent. Proposition 2.5 can be generalized to arbitrary programs as follows:

Proposition 3.4. *If a program Π is head-consistent then every answer set for Π is consistent.*

This proposition tells us that a head-consistent program cannot belong to the second of the three groups listed in the corollary to Proposition 3.1. It is possible, however, that a head-consistent program belongs to the first group, so that such a program can be inconsistent. For instance, (44) is a head-consistent program without answer sets. An additional condition needed to guarantee the existence of an answer set, called "order-consistency," is discussed in Section 3.5.

3.2 Tight Programs

According to Propositions 3.2 and 3.3, every consistent answer set is closed and supported. Proposition 3.5 below shows that for a large class of programs the converse is also true, so that the two properties, closure and supportedness, completely characterize the class of consistent answer sets for programs in this class.

A program Π is *tight* if there exists a level mapping λ such that, for every rule $Head \leftarrow Pos \cup not(Neg)$ in Π,

$$\lambda(Head) > \max_{L \in Pos} \lambda(L).$$

Note that this condition does not impose any restriction on *Neg*, that is, on the rule elements that include negation as failure. If $Pos = \emptyset$ in every rule of the program then the program is trivially tight. A basic program is tight iff it is hierarchical.

Problem 3.4. Determine which of the programs (38), (41), (44)–(47) are tight.

Proposition 3.5. *For any tight program Π and consistent set X of literals, X is an answer set for Π iff X is both closed under Π and supported by Π.*

Problem 3.5. Show that this assertion would be incorrect without the assumption that Π is tight.

Any program can be "tightened" at the price of introducing infinitely many new atoms. For every A in the set $\mathbf{A}$ of atoms and for every integer $n \neq 0$, let A^n be a new symbol. For every $n > 0$, define

$$(\neg A)^n = A^{-n}$$

for any atom A, and

$$X^n = \{L^n \, : \, L \in X\}$$

for any set X of literals. Intuitively, A^n says that A can be "established in n steps" using the rules of the program; A^{-n} says that A can be "refuted in n steps."

Let $\mathbf{A}^\infty$ be the extended set of atoms obtained from $\mathbf{A}$ by adding the new symbols A^n $(n \neq 0)$. The *tightening* of Π is the program with the set of atoms $\mathbf{A}^\infty$ that consists of

- the rules

  $$Head^{n+1} \leftarrow Pos^n \cup not(Neg)$$

  for every rule $Head \leftarrow Pos \cup not(Neg)$ in Π and every $n > 0$, and

- the rules

  $$\begin{array}{l} A \leftarrow A^n, \\ \neg A \leftarrow A^{-n} \end{array}$$

  for every $A \in \mathbf{A}$ and every $n > 0$.

For example, the tightening of

$$\begin{array}{l} p \leftarrow p, not\ q, \\ \neg p \leftarrow not\ p \end{array}$$

consists of the rules

$$
\begin{aligned}
&p^{n+1} \leftarrow p^n, \textit{not}\ q,\\
&p^{-n-1} \leftarrow \textit{not}\ p,\\
&p \leftarrow p^n,\\
&\neg p \leftarrow p^{-n},\\
&q \leftarrow q^n,\\
&\neg q \leftarrow q^{-n}
\end{aligned}
$$

for all $n > 0$.

The tightening of any program is tight: define

$$\lambda(A^n) = |n|,\ \lambda(A) = \omega.$$

The tightening of a program is its "conservative extension":

Proposition 3.6. *A subset of $Lit_{\mathbf{A}}$ is an answer set for a program Π iff it can be represented in the form $X \cap Lit_{\mathbf{A}}$, where X is an answer set for the tightening of Π.*

Corollary. *If Π' is the tightening of a program Π then*

$$\mathrm{Cn}(\Pi) = \mathrm{Cn}(\Pi') \cap Lit_{\mathbf{A}}.$$

3.3 Well-Founded Consequences

In Section 2.3 we saw that the set of consequences of a basic program Π is the least fixpoint of a certain monotone function, called T_Π, and that this set can be approximated from below by iterating that function on the empty set. A similar construction can be defined for programs with negation as failure. However, the monotone function involved in it is defined in a more complicated way than T_Π. Also, the least fixpoint of this function is sometimes a proper subset of the set of consequences, so that some consequences may be impossible to reach by iterating it—even for a finite program. Nevertheless, this function and its fixpoints are important, both theoretically and computationally.

For any program Π, the function γ_Π from sets of literals to sets of literals is defined by the equation

$$\gamma_\Pi X = \mathrm{Cn}(\Pi^X).$$

It is clear that the answer sets for Π can be characterized as the fixpoints of γ_Π.

Proposition 3.7. *For any program Π, γ_Π is anti-monotone.*

As discussed in Appendix, it follows that γ_Π^2 is monotone, and its least and greatest fixpoints limit the fixpoints of γ_Π from below and from above. The literals that belong to the least fixpoint of γ_Π^2 are said to be *well-founded* relative to Π. The literals that do not belong to the greatest fixpoint of γ_Π^2 are *unfounded* relative to Π. Thus any program partitions the set of literals into three groups: the well-founded literals, the unfounded literals, and the rest.

The second part of Proposition A.3, in application to this case, can be stated as follows:

Proposition 3.8. *Any answer set for a program* Π

- *includes all literals that are well-founded relative to* Π, *and*
- *includes no literals unfounded relative to* Π.

Corollary. *For any program* Π *and any literal* L,

- *if* L *is well-founded relative to* Π *then* L *is a consequence of* Π,
- *if* Π *is consistent and* L *is unfounded relative to* Π *then* L *is not a consequence of* Π.

The Corollary to Proposition A.3, in application to this case, can be stated as follows:

Proposition 3.9. *If every literal is either well-founded or unfounded relative to* Π *then the set of well-founded literals is the only answer set for* Π.

If Π is finite then the literals that are well-founded or unfounded relative to Π can be found by iterating γ_Π on the empty set or on the set of all literals, as discussed at the end of Appendix. For instance, in case of program (41),

$$\begin{array}{ll} \gamma^0_\Pi\emptyset & = \emptyset, \\ \gamma^1_\Pi\emptyset & = \{p, q, r\}, \\ \gamma^2_\Pi\emptyset & = \emptyset, \end{array}$$

so that $\emptyset$ is the least fixpoint of γ^2_Π, and $\{p, q, r\}$ is its greatest fixpoint. We see that the set of well-founded literals is empty, so that p is a consequence of the program that is not well-founded. The only unfounded literals are the negative literals $\neg p, \neg q, \neg r$.

Problem 3.6. Find the well-founded and unfounded literals for programs (38) and (45)–(47).

3.4 Splitting a Program with Negation as Failure

The splitting process described in Section 2.4 for basic programs can be extended to arbitrary programs as follows. We say that a set U of literals *splits* a program Π if, for every rule $\mathit{Head} \leftarrow \mathit{Pos} \cup \mathit{not}(\mathit{Neg})$ in Π, $\mathit{Pos} \cup \mathit{Neg} \subseteq U$ whenever $\mathit{Head} \in U$. If U splits Π then the set of rules in Π whose heads belong to U is the *base* of Π (relative to U), denoted by $b_U(\Pi)$.

For instance, program (38) is split by the sets $\{p\}$ and $\{p, s\}$ (and by several others). The base of (38) relative to each of these two sets consists of its first rule, p.

For any program Π, any set U of literals and any subset C of U, $e_U(\Pi, C)$ stands for the program obtained from Π by

- deleting each rule $\mathit{Head} \leftarrow \mathit{Pos} \cup \mathit{not}(\mathit{Neg})$ such that $\mathit{Pos} \cap (U \setminus C) \neq \emptyset$ or $\mathit{Neg} \cap C \neq \emptyset$,

- replacing each remaining rule $Head \leftarrow Pos \cup not(Neg)$ by

$$Head \leftarrow (Pos \setminus U) \cup not(Neg \setminus U).$$

Proposition 3.10. *Let U be a set of literals that splits a program Π. A consistent set of literals is an answer set for Π iff it can be represented in the form $C_1 \cup C_2$, where C_1 is an answer set for $b_U(\Pi)$ and C_2 is an answer set for $e_U(\Pi \setminus b_U(\Pi), C_1)$.*

This theorem suggests the following approach to computing the consistent answer sets for a program Π that is split by some set U. First find all answer sets for the base $b_U(\Pi)$. For each of these sets C_1, compute the program $e_U(\Pi \setminus b_U(\Pi), C_1)$, and find all its answer sets. For each of these sets C_2, form the union $C_1 \cup C_2$. The consistent unions found in this way are the consistent answer sets for Π. The intersection of all these consistent unions is $\mathrm{Cn}(\Pi)$.

Take program (38) as an example. Let $U = \{p, s\}$. Then $b_U(\Pi)$ is the basic program whose only rule is p, and the only answer set C_1 for this program is $\{p\}$. Furthermore, $\Pi \setminus b_U(\Pi)$ is

$$\begin{array}{l} q \leftarrow p, not\ r, \\ q \leftarrow r, not\ p, \\ r \leftarrow p, not\ s, \end{array}$$

and $e_U(\Pi \setminus b_U(\Pi), C_1)$ is

$$\begin{array}{l} q \leftarrow not\ r, \\ r. \end{array}$$

The only answer set C_2 for this program is $\{r\}$. We conclude that the union of C_1 and C_2, that is, $\{p, r\}$, is the only consistent answer set for (38). (By the corollary to Proposition 3.1, it follows that (38) has no other answer sets.)

As another application, consider rules (9)–(15) from the royal family example (Section 1.2). Recall that metavariables x, y, z in the schematic rules stand for the constants *Elizabeth*,...,*Harry*. This program Π is split by the set U consisting of all positive *Mother*, *Father*, *Parent*, *Grandparent* and *Ancestor* literals, all negative *Childless* literals, and all *Male* literals. The base $b_U(\Pi)$ is a consistent basic program; let C_1 be the set of its consequences. The remaining rules $\Pi \setminus b_U(\Pi)$ are the closed world assumptions, and all rules in $e_U(\Pi \setminus b_U(\Pi), C_1)$ are quite simple: Their bodies are empty, and their heads are complementary to the *Mother*, *Father*, *Parent*, *Childless*, *Grandparent* and *Ancestor* literals in $U \setminus C_1$. This program is identical to its only answer set C_2. According to Proposition 3.10, $C_1 \cup C_2$ is the only answer set for (9)–(15).

The following problem shows how this example can be generalized.

Problem 3.7. The *closed world assumption rule* for a literal L is the rule

$$L \leftarrow not\ \overline{L},$$

where $\overline{L}$ stands for the literal complementary to L. Let Π be a program, let C be a consistent set of literals that do not occur in Π, and let Π' be the program obtained from Π by adding the closed world assumption rules for all literals in C. (a) Show that, if X is a consistent answer set for Π, then

$$X \cup \{L \in C : \overline{L} \notin X\}$$

is a consistent answer set for Π. (b) Show that every consistent answer set for Π' can be represented in this form for some consistent answer set X for Π.

3.5 Stratified and Order-Consistent Programs

Both examples used above to illustrate the process of splitting belong to an important class of programs, called "stratified." This property guarantees that the program is split by a nonempty set U such that the base of the program relative to U does not contain negation as failure. Moreover, the result of eliminating the elements of U from the rest of the program is again a stratified program. Consequently, by repeating the splitting process several times, we can reduce any finite stratified program to a series of basic programs.

A program Π is *stratified* if there exists a level mapping λ such that, for every rule $Head \leftarrow Pos \cup not(Neg)$ in Π,

$$\begin{array}{l} \lambda(Head) \geq \max_{L \in Pos} \lambda(L), \\ \lambda(Head) > \max_{L \in Neg} \lambda(L). \end{array} \tag{48}$$

It is clear that any basic program is stratified (take λ identically equal to 1). For program (38), we can take

$$\lambda(L) = \begin{cases} 2, & \text{if } L = q, \\ 1, & \text{if } L = r, \\ 0, & \text{otherwise.} \end{cases}$$

Programs (41) and (44) are not stratified.

Any stratified program is split by the set of atoms on which λ is minimal, and the base of the program relative to this set is a basic program.

The following theorem gives another important property of stratified programs:

Proposition 3.11. *A stratified program has at most one answer set.*

"Order-consistency" is a condition more general than stratification, which, in combination with head-consistency (Section 3.1), implies the existence of an answer set. Its definition uses the following notation. For any program Π and any literal L, Π_L^+ and Π_L^- are the smallest sets of literals such that $L \in \Pi_L^+$ and, for every rule $Head \leftarrow Pos \cup not(Neg)$ in Π,

- if $Head \in \Pi_L^+$ then $Pos \subseteq \Pi_L^+$ and $Neg \subseteq \Pi_L^-$,
- if $Head \in \Pi_L^-$ then $Pos \subseteq \Pi_L^-$ and $Neg \subseteq \Pi_L^+$.

We say that a program Π is *order-consistent* if there exists a level mapping λ such that $\lambda(L_2) < \lambda(L_1)$ whenever $L_2 \in \Pi^+_{L_1} \cap \Pi^-_{L_1}$.

For example, a program is order-consistent if, for every literal L,

$$\Pi^+_L \cap \Pi^-_L = \emptyset.$$

(Programs with this property are called *strict*.) To see why all stratified programs are order-consistent, note that for any level mapping λ satisfying (48) and for any literals L_1 and L_2,

- $\lambda(L_2) \leq \lambda(L_1)$ whenever $L_2 \in \Pi^+_{L_1}$,
- $\lambda(L_2) < \lambda(L_1)$ whenever $L_2 \in \Pi^-_{L_1}$.

A finite program Π is order-consistent iff, for every atom A, $A \notin \Pi^-_A$.

Problem 3.8. For each of the programs (41) and (44) determine whether it is (a) strict, (b) order-consistent.

Proposition 3.12. *If a program is head-consistent and order-consistent then it is consistent.*

From the last two propositions we see that a program which is both head-consistent and stratified has a unique answer set.

3.6 The SLDNF Calculus

The definition of the *SLDNF calculus* associated with a program Π generalizes the definition of the SLD calculus introduced in Section 2.5 to programs with negation as failure. Its derivable objects are again expressions of the forms $\models G$ and $=\!\mid G$, except that a *goal* G is now defined as a finite set of rule elements. The axiom is the same as before,

$$\models \emptyset.$$

There are four inference rules:

$$(SP)\ \frac{\models G \cup B}{\models G \cup \{L\}} \qquad \text{if } B \in \mathit{Bodies}(L)$$

$$(FP)\ \frac{=\!\mid G \cup B \qquad \text{for all } B \in \mathit{Bodies}(L)}{=\!\mid G \cup \{L\}}$$

$$(SN)\ \frac{\models G \qquad =\!\mid \{L\}}{\models G \cup \{not\ L\}}$$

$$(FN)\ \frac{\models \{L\}}{=\!\mid G \cup \{not\ L\}}$$

As in Section 2.5, *Bodies*(L) stands here for the set of the bodies of all rules in Π whose head is L. The success and failure rules for positive subgoals, (SP) and (FP), look the same as rules (S) and (F) of the SLD calculus. The rules for negative subgoals, (SN) and (FN), allow us to derive success from failure and failure from success. As a result, failure expressions can occur now in derivations of success expressions, and the other way around.

For instance, here is a derivation in the SLDNF calculus for program (38):

$$
(FP)\,\dfrac{
 (FN)\,\dfrac{
  (SP)\,\dfrac{
   (SP)\,\dfrac{
    (SN)\,\dfrac{\models \emptyset \qquad (FP)\,\dfrac{}{\dashv \{s\}}}{\models \{not\ s\}}
   }{\models \{p, not\ s\}}
  }{\models \{r\}}
 }{\dashv \{p, not\ r\}}
 \qquad
 (FN)\,\dfrac{
  (SP)\,\dfrac{\models \emptyset}{\models \{p\}}
 }{\dashv \{r, not\ p\}}
}{\dashv \{q\}}
$$

If $\models G$ is derivable in the SLDNF calculus for Π then we say that G *succeeds* relative to Π. If $\dashv G$ is derivable then we say that G *fails* relative to Π. For instance, the derivation above demonstrates that $\{q\}$ fails with respect to program (38).

Propositions 2.10 and 2.11 can be extended to programs with negation as failure in the following way:

Proposition 3.13. *For any program Π, no goal both succeeds and fails relative to Π.*

Proposition 3.14. *For any program Π and any literal L,*

- *if $\{L\}$ succeeds relative to Π then L is a well-founded consequence of Π,*
- *if Π is consistent and $\{L\}$ fails relative to Π then L is unfounded relative to Π.*

The last proposition is the soundness theorem for the SLDNF calculus. Its second half, in combination with Proposition 3.9 and its corollary, shows that if $\{L\}$ fails with respect to a consistent program Π then L does not belong to any answer set for Π, nor is a consequence of Π.

Proposition 3.14 shows also that if L is neither well-founded nor unfounded then $\{L\}$ neither succeeds nor fails. For instance, the goals $\{p\}$, $\{q\}$, $\{r\}$ neither succeed nor fail relative to program (41).

Problem 3.9. Determine for which values of n the goal $\{p_n\}$ succeeds relative to program (46), and for which values of n it fails.

The concept of a hierarchical program (Section 2.5), used in the statement of the completeness theorem, is extended to programs with negation as failure in the following way. We say that a program Π is *hierarchical* if there exists a level mapping λ such that, for every rule $Head \leftarrow Pos \cup not(Neg)$ in Π,

$$\lambda(Head) > \max_{L \in Pos \cup Neg} \lambda(L).$$

For example, program (38) is hierarchical; programs (41) and (44) are not. It is clear that any hierarchical program is stratified, and consequently has at most one answer set.

Proposition 3.15. *Let Π be a hierarchical program. For any literal L,*

- *if Π is consistent and L is a consequence of Π then $\{L\}$ succeeds relative to Π,*
- *if L is not a consequence of Π then $\{L\}$ fails relative to Π.*

3.7 Prolog with Negation as Failure

In order to describe the work of Prolog in the presence of negation as failure, we need to note the following fact:

Proposition 3.16. *For any goals G_1 and G_2, if G_1 fails then $G_1 \cup G_2$ fails also.*

This proposition shows that adding the "thinning" rule

$$(T)\ \frac{\dashv G_1}{\dashv G_1 \cup G_2}$$

to the calculus described above would not change the set of derivable expressions.

The work of propositional Prolog can be described as proof search in the calculus consisting of the rules (SP), (FP), (SN), (FN) and (T). Search proceeds as described in Section 2.6, except that now the selected subgoal can include the negation as failure symbol. If G, the goal to be evaluated, is represented as $G' \cup \{L\}$ then the rule applied in the derivation last will be either (SP) or (FP). If G is represented as $G' \cup \{not\ L\}$ then the rule applied in the derivation last will be (SN), (FN) or (T); Prolog determines which one by attempting to evaluate $\{L\}$. If this goal succeeds then $\dashv G$ can be derived by (FN). If it fails then Prolog evaluates G'. If G' succeeds then $\models G$ will follow by (SN). If G' fails then $\dashv G$ will follow by (T).

Consider, for instance, the operation of Prolog on the program that consists of one rule

$$p \leftarrow not\ q$$

and the goal $\{p\}$. The search process first produces the partial derivation

$$\perp\!\!\perp\ \{p\}$$

then

$$\frac{\perp\!\!\!\perp \ \{not\ q\}}{\perp\!\!\!\perp \ \{p\}}$$

and then

$$\dfrac{\dfrac{\perp\!\!\!\perp \ \{q\}}{\perp\!\!\!\perp \ \{not\ q\}}}{\{p\}}$$

Since $Bodies(q)$ is empty, we next arrive at

$$\dfrac{\dfrac{(FP)\ \dfrac{}{=\!\!\!| \ \{q\}}}{\perp\!\!\!\perp \ \{not\ q\}}}{\perp\!\!\!\perp \ \{p\}}$$

and then at

$$\dfrac{\dfrac{\perp\!\!\!\perp \ \emptyset \qquad (FP)\ \dfrac{}{=\!\!\!| \ \{q\}}}{\perp\!\!\!\perp \ \{not\ q\}}}{\perp\!\!\!\perp \ \{p\}}$$

The next step leads to the derivation

$$(SP)\ \dfrac{(SN)\ \dfrac{\models\ \emptyset \qquad (FP)\ \dfrac{}{=\!\!\!| \ \{q\}}}{\models\ \{not\ q\}}}{\models\ \{p\}}$$

Consider now the program

$$\begin{array}{l} p \leftarrow not\ q, \\ q \end{array}$$

and the same goal $\{p\}$. The first three partial derivations will be the same as before, and then we will arrive at the derivation

$$(FP)\,\dfrac{(FN)\,\dfrac{(SP)\,\dfrac{\models\ \emptyset}{\models\ \{q\}}}{\dashv\ \{not\ q\}}}{\dashv\ \{p\}}$$

Finally, consider the program

$$\begin{array}{l} p \leftarrow not\ r, not\ q, \\ q \end{array}$$

and the same goal $\{p\}$. The search process leads first to the partial derivation

$$\dfrac{\perp\!\!\!\perp\ \{not\ r, not\ q\}}{\perp\!\!\!\perp\ \{p\}}$$

then to

$$\dfrac{\dfrac{\perp\!\!\!\perp\ \{r\}}{\perp\!\!\!\perp\ \{not\ r, not\ q\}}}{\perp\!\!\!\perp\ \{p\}}$$

and then to

$$\dfrac{\dfrac{(FP)\,\dfrac{}{\dashv\ \{r\}}}{\perp\!\!\!\perp\ \{not\ r, not\ q\}}}{\perp\!\!\!\perp\ \{p\}}$$

Next Prolog evaluates the goal $\{not\ q\}$. We obtain first

$$\dfrac{\dfrac{\perp\!\!\!\perp\ \{not\ q\} \qquad (FP)\,\dfrac{}{\dashv\ \{r\}}}{\perp\!\!\!\perp\ \{not\ r, not\ q\}}}{\perp\!\!\!\perp\ \{p\}}$$

then

$$\dfrac{\dfrac{\dfrac{\perp\!\!\!\perp \{q\}}{\perp\!\!\!\perp \{not\ q\}} \quad (FP)\,\dfrac{}{\Dashv \{r\}}}{\perp\!\!\!\perp \{not\ r, not\ q\}}}{\perp\!\!\!\perp \{p\}}$$

and then

$$\dfrac{\dfrac{\dfrac{\dfrac{\perp\!\!\!\perp \emptyset}{\perp\!\!\!\perp \{q\}}}{\perp\!\!\!\perp \{not\ q\}} \quad (FP)\,\dfrac{}{\Dashv \{r\}}}{\perp\!\!\!\perp \{not\ r, not\ q\}}}{\perp\!\!\!\perp \{p\}}$$

The final result is the derivation

$$(FP)\,\dfrac{(T)\,\dfrac{(FN)\,\dfrac{(SP)\,\dfrac{\models \emptyset}{\models \{q\}}}{\Dashv \{not\ q\}}}{\Dashv \{not\ r, not\ q\}}}{\Dashv \{p\}}$$

Problem 3.10. Describe the work of Prolog on the program

$$\begin{aligned} p &\leftarrow not\ q, \\ q &\leftarrow not\ r, s \end{aligned}$$

and the goal $\{p\}$. How would the process be affected by inserting *not* in front of s in the last rule?

3.8 Normal Programs with Negation as Failure

A rule element, rule or program is *normal* if it does not contain the classical negation symbol $\neg$. By Proposition 3.4, every answer set for a normal program is consistent. By Corollary 1 to Proposition 3.3, it follows that every answer set for a normal program is a set of atoms. Program (44) is an example of a normal program without answer sets.

In Section 2.7, we defined an encoding Norm of basic programs by normal basic programs over a larger set of atoms. In order to extend this encoding to arbitrary rules and programs, we define

$$\mathrm{Norm}(Head \leftarrow Pos \cup not(Neg))$$

to be

$$\mathrm{Norm}(Head) \leftarrow \mathrm{Norm}(Pos) \cup not(\mathrm{Norm}(Neg)).$$

Here are two generalizations of Proposition 2.14 to arbitrary programs:

Proposition 3.17. *For any program Π and any set X of literals, the following conditions are equivalent:*

(i) X *is an answer set for* Π,

(ii) $\mathrm{Norm}(X)$ *is an answer set for* $\mathrm{Norm}(\Pi) \cup Contr$.

Moreover, if X is consistent then these conditions are equivalent to

(iii) $\mathrm{Norm}(X)$ *is an answer set for* $\mathrm{Norm}(\Pi)$.

Corollary. *For any program Π,*

$$\mathrm{Norm}(\mathrm{Cn}(\Pi)) = \mathrm{Cn}(\mathrm{Norm}(\Pi) \cup Contr).$$

Adding *Contr* in the right-hand side of this equality may be needed even if Π is consistent. This can be seen from the following counterexample:

$$\begin{array}{l} p \leftarrow not\ q, \\ q \leftarrow not\ p, \\ \neg p. \end{array}$$

In Section 2.7, we defined a function φ that encodes normal basic rules by propositional formulas. The extension of this function to normal rules with negation as failure, defined below, replaces negation as failure by classical negation. For any set B of normal rule elements, by $B^{not}_{\neg}$ we denote the conjunction of the literals obtained from the elements of B by substituting $\neg$ for each *not*. Then, for any normal rule $Head \leftarrow Body$, we define:

$$\varphi(Head \leftarrow Body) = Body^{not}_{\neg} \supset Head.$$

For instance,

$$\varphi(p \leftarrow q, not\ r)$$

is

$$q \wedge \neg r \supset p.$$

For any normal program Π, $\varphi\Pi$ stands for the set of formulas φR for all rules $R \in \Pi$.

The following theorem is a counterpart of Corollary 1 to Proposition 2.15.

Proposition 3.18. *For any normal program* Π,

- *any answer set for* Π *is a model of* $\varphi\Pi$,
- *no proper subset of an answer set for* Π *is a model of* $\varphi\Pi$.

In other words, an answer set for a normal program Π is a "minimal model" of $\varphi\Pi$. The converse is not necessarily true, even for hierarchical programs. For instance, if Π is

$$\begin{array}{l} p \leftarrow not\ q, \\ q \leftarrow not\ r \end{array} \tag{49}$$

then the minimal models of $\varphi\Pi$ are $\{q\}$ and $\{p, r\}$; the latter is not an answer set for Π.

The *completion* of a finite normal program Π, $\varphi_{comp}\Pi$, is the set of formulas

$$H \equiv \bigvee_{B \in Bodies(H)} B_{\neg}^{not}$$

for all atoms H. For example, the completion of (49) is

$$\begin{array}{l} p \equiv \neg q, \\ q \equiv \neg r, \\ r \equiv false. \end{array}$$

It is clear that all formulas in $\varphi\Pi$ are propositional consequences of $\varphi_{comp}\Pi$.

The completion of a program is closely related to the two properties of programs discussed in Sections 3.1 and 3.2—closure and supportedness:

Proposition 3.19. *For any finite normal program* Π, *an interpretation* I *is a model of* $\varphi_{comp}\Pi$ *iff* I *is both closed under* Π *and supported by* Π.

Using Proposition 3.5, we conclude that answer sets for a finite tight normal program can be characterized in terms of propositional logic:

Corollary. *For any finite tight normal program* Π, *an interpretation* I *is an answer set for* Π *iff* I *satisfies* $\varphi_{comp}\Pi$.

Problem 3.11. Use this corollary to find the answer sets for programs (38), (41) and (45).

4 Schematic Programs

To turn logic programming into a usable representational and computational tool, we need to add variables to the language of programs introduced above. In this section, we define the syntax and semantics of programs with variables, show how they can be used for representing defaults, and then describe the operation of Prolog in the presence of variables.

4.1 Syntax and Semantics

Consider a first-order language **L** without equality that has at least one object constant and at least one predicate constant. Literals of this language will be called *schematic literals*. The definitions of *a schematic rule element*, *a schematic rule* and *a schematic program* are parallel to the definitions of a rule element, a rule and a program given in Section 3.1, with schematic literals used instead of literals. For instance, (9) is a schematic program in the language with the object constants *Elizabeth*,...,*Harry*, no function constants, and the binary predicate constant *Mother*.

For any schematic rule R, $Ground(R)$ stands for the set of all ground instances of R. For any schematic normal program Π,

$$Ground(\Pi) = \bigcup_{R \in \Pi} Ground(R).$$

It is clear that $Ground(R)$ and $Ground(\Pi)$ are programs in the sense of Section 3.1, if the set of atoms **A** is taken to be the set of all ground atoms of **L**.

Consider, for instance, the language with the object constant 0, the unary function constant s, and the unary predicate constant p. If Π consists of one schematic rule

$$p(s(x)) \leftarrow not\ p(x)$$

then $Ground(\Pi)$ is program (46), assuming that p_n is identified with $p(s^n(0))$.

Problem 4.1. Find a program with the same set of atoms as (46) that cannot be represented in the form $Ground(\Pi)$ for a finite schematic program Π.

An *answer set* for a schematic program Π is an answer set for $Ground(\Pi)$. Consequences, consistency, well-founded and unfounded literals are defined for schematic programs in a similar way. Note that the consequences of a schematic program are *ground* literals of **L**.

4.2 Representing Defaults

One important use of negation as failure in schematic programs is for representing "defaults." Consider, for instance, the rule

$$Married(x, y) \leftarrow Father(x, z), Mother(y, z), \tag{50}$$

asserting that two persons with a common child are a married couple. One of the consequences of this rule, in combination with (9) and (10), is

$$Married(Charles, Diana). \tag{51}$$

Problem 4.2. Verify this assertion.

In real life, of course, the general assertion expressed by (50) is known to admit exceptions, so that it would be better to treat (51) as a "default conclusion" in the absence of evidence to the contrary. The weaker assertion that two persons with a common child are *normally* married to each other can be expressed by the schematic rule

$$Married(x, y) \leftarrow Father(x, z), Mother(y, z), not\ Ab(x, y). \tag{52}$$

Here Ab is an auxiliary "abnormality" predicate. The program consisting of rules (9), (10) and (52) leads to conclusion (51) also.

Problem 4.3. Verify this assertion.

The difference between "rigid" rule (50) and "default" rule (52) is that, with the latter used instead of the former, conclusion (51) becomes defeasible. The rule

$$Ab(Charles, Diana) \tag{53}$$

expresses that Charles and Diana are a possible exception to the default about couples with common children. By adding it to the program, we will "nonmonotonically" make (51) undecidable.

Problem 4.4. Verify that the program consisting of rules (9), (10), (52) and (53), indeed, has neither (51) nor the negation of (51) among its consequences.

Moreover, if we wish to assert that Charles is not married to Diana, this can be expressed by

$$\neg Married(Charles, Diana). \tag{54}$$

In combination with (9), (10) and (50), this rule would lead to an inconsistency; the schematic program with rules (9), (10) and (52)–(54) is consistent.

Problem 4.5. Verify these assertions.

A schematic rule can be used to express that all objects in a certain class are exceptions to a default. For instance, we can express that adults are normally employed, but high school dropouts are possible exceptions, by writing

$$\begin{array}{l} Employed(x) \leftarrow Adult(x), not\ Ab(x), \\ Ab(x) \leftarrow Dropout(x). \end{array}$$

Problem 4.6. Blocks $B_1, \ldots, B_{10}$ are normally located on the table. However, B_1 is not on the table, and we are not sure about B_2. Express these assertions as a schematic program in the language with variables for blocks, the object constants $B_1, \ldots, B_{10}$ and the unary predicate constants $OnTable$ and Ab. Find the consequences of this program.

Problem 4.7. Birds normally fly. However, penguins do not fly. Opus is a penguin, and Tweety is not. Express these assertions as a schematic program in the language with variables for birds, the object constants $Opus$ and $Tweety$ and the unary predicate constants $Flies$, $Penguin$ and Ab. Find the consequences of this program.

If several defaults are involved then several abnormality predicates have to be used, one per default.

Problem 4.8. Quakers are normally pacifists, and Republicans are normally not pacifists. Alice and Bob are Quakers, and Carol is not. Bob and Carol are Republicans, and Alice is not. Express these assertions as a schematic program in the language with variables for people, the object constants *Alice*, *Bob* and *Carol*, and the unary predicate constants *Quaker*, *Republican*, *Pacifist*, Ab_1 and Ab_2. Find the consequences of this program.

4.3 The SLDNF Calculus for Schematic Programs

A *schematic goal* is a finite set of schematic literals. In the SLDNF calculus for a schematic program, derivable objects are expressions of the forms $\models G : \delta$ and $\dashv G$, where G is a schematic goal and δ is a substitution[3] whose support is a subset of the set of variables occurring in G. If $\models G : \delta$ is derivable then we will say that G *succeeds relative to* Π *with computed answer substitution (c.a.s.)* δ; if $\dashv G$ is derivable then we will say that G *fails relative to* Π. The soundness theorem (Corollary 2 to Proposition 4.1 below) shows that if $\{L\}$ succeeds with c.a.s. δ then every ground instance of $L\delta$ is a consequence of the program; if $\{L\}$ fails then no ground instance of L is a consequence.

The following schematic program will be used here for illustration:

$$\begin{array}{l} p(a,b), \\ q(b), \\ r(y) \leftarrow p(x,y), not\ q(x). \end{array} \tag{55}$$

We will see that, for this program, $\{r(x)\}$ succeeds with c.a.s. $\{x/b\}$. This fact shows that $r(b)$ is a consequence of the program.

In propositional case (Section 3.6), the SLDNF calculus is based on the idea that a rule of the program is "applicable" to a literal L when the head of the rule is L. For schematic programs, the definition of applicability is different. About a schematic rule we say that it is *applicable* to a schematic literal L if the head H of the rule has a common instance with L, that is, if there exist substitutions σ_1 and σ_2 such that $H\sigma_1 = L\sigma_2$. For example, the last rule of (55) is applicable to $r(x)$.

The "application" of a schematic rule to a schematic literal L is achieved by unifying L with the head of the rule. Prior to the unification, the variables in the rule are renamed so that they do not appear in the goal under consideration. For instance, before the last rule of (55) is applied to $r(x)$, it can be replaced by

$$r(y) \leftarrow p(x_1,y), not\ q(x_1). \tag{56}$$

We assume that a specific variable renaming procedure *vr* is chosen: For every schematic rule R and every finite set V of variables, an invertible[4] substitution

[3]For the definitions related to the notion of a substitution, the reader is referred to the textbook by Fitting [1990], Sections 5.2 and 7.2.

[4]A substitution θ is *invertible* if there is a substitution θ^{-1} such that both $\theta\theta^{-1}$ and $\theta^{-1}\theta$ equal the identity substitution ϵ.

$vr(R,V)$ is selected such that the rule $R \cdot vr(R,V)$ does not contain variables from V. For instance, if R is the last rule of (55) then one possible choice for $vr(R,\{x\})$ is $\{x/x_1, x_1/x\}$; the result of applying this substitution to R is (56).

By $mgu(L_1,L_2)$ we denote a most general unifier of schematic literals L_1 and L_2. For any goal G, $vars(G)$ stands for the set of variables occurring in G. For any substitution δ and any set V of variables, $\delta \mid V$ stands for the restriction of δ to V:

$$x(\delta \mid V) = \begin{cases} x\delta, & \text{if } x \in V, \\ x, & \text{otherwise.} \end{cases}$$

The *SLDNF calculus* for a schematic program Π consists of the axiom

$$\models \emptyset : \epsilon$$

and the following inference rules:

$$(SP)\ \frac{\models (G \cup B\theta)\sigma \ : \ \delta}{\models G \cup \{L\} \ : \ \sigma\delta \mid vars(G \cup \{L\})}$$

if $H \leftarrow B$ is a rule of Π applicable to L,
where $\theta = vr(H \leftarrow B, vars(G \cup \{L\}))$ and $\sigma = mgu(L, H\theta)$

$$(FP)\ \frac{=\!\mid (G \cup B\theta)\sigma \qquad \text{for all rules } H \leftarrow B \text{ of } \Pi \text{ applicable to } L}{=\!\mid G \cup \{L\}}$$

where $\theta = vr(H \leftarrow B, vars(G \cup \{L\}))$ and $\sigma = mgu(L, H\theta)$

$$(SN)\ \frac{\models G \ : \ \delta \qquad =\!\mid \{L\}}{\models G \cup \{not\ L\} \ : \ \delta} \qquad \text{if } L \text{ is ground}$$

$$(FN)\ \frac{\models \{L\} \ : \ \epsilon}{=\!\mid G \cup \{not\ L\}} \qquad \text{if } L \text{ is ground}$$

The following comments on the inference rule (SP) may be helpful.

1. The variable renaming θ is selected in such a way that the schematic rule obtained from $H \leftarrow B$ by the renaming, that is,

$$H\theta \leftarrow B\theta, \tag{57}$$

does not have common variables with the goal $G \cup \{L\}$.

2. The substitution σ unifies L with the head $H\theta$ of (57). These two literals are indeed unifiable because they have a common instance ($H \leftarrow B$ is applicable to L, and θ is invertible) and have no common variables.

3. The premise of (SP) can be thought of as a symbolic representation of the set of ground instances of

$$(G \cup B\theta)\sigma\delta;$$

this goal can be written as

$$G\sigma\delta \cup B\theta\sigma\delta.$$

Similarly, the conclusion of (SP) represents the set of ground instances of

$$(G \cup \{L\})\sigma\delta;$$

by the choice of σ, this goal can be written as

$$G\sigma\delta \cup \{H\theta\sigma\delta\}.$$

We see that the bottom-up application of (SP) amounts to replacing the head of the instance

$$H\theta\sigma\delta \leftarrow B\theta\sigma\delta$$

of (57) with its body.

4. In the conclusion of (SP), the substitution $\sigma\delta$ is restricted to the variables occurring in the goal $G \cup \{L\}$. Without this, the conclusion might be not in the class of derivable expressions of this calculus.

Note that in (SN) and (FN) the literal L is required to be ground. It is clear that any goal of the form $\{not\ L\}$, where L is nonground, neither succeeds nor fails.

Here is a derivation in the SLDNF calculus for (55):

$$
(SP)\;\dfrac{(SP)\;\dfrac{(SN)\;\dfrac{\models \emptyset \;:\; \epsilon \qquad (FP)\;\dfrac{}{\dashv \{q(a)\}}}{\models \{not\ q(a)\} \;:\; \epsilon}}{\models \{p(x_1,x), not\ q(x_1)\} \;:\; \{x/b, x_1/a\}}}{\models \{r(x)\} \;:\; \{x/b\}}
$$

In the first application of (SP),

$$
\begin{gathered}
H = p(a,b),\ B = \emptyset,\ G = \{not\ q(x_1)\},\ L = p(x_1,x),\\
\theta = \epsilon,\ \sigma = mgu(p(x_1,x), p(a,b)) = \{x/b, x_1/a\},\ \delta = \epsilon,\\
\sigma\delta \mid vars(G \cup \{L\}) = \{x/b, x_1/a\} \mid \{x, x_1\} = \{x/b, x_1/a\}.
\end{gathered}
$$

In the second application,

$$
\begin{gathered}
H = r(y),\ B = \{p(x,y), not\ q(x)\},\ G = \emptyset, L = r(x),\\
\theta = \{x/x_1, x_1/x\},\ \sigma = mgu(r(x), r(y)) = \{y/x\},\ \delta = \{x/b, x_1/a\},\\
\sigma\delta \mid vars(G \cup \{L\}) = \{x/b, y/b, x_1/a\} \mid \{x\} = \{x/b\}.
\end{gathered}
$$

Problem 4.9. For the schematic program whose only rule is $p(x,x)$, find a substitution δ such that $\{p(x,y)\}$ succeeds with c.a.s. δ.

The following theorem relates the SLDNF calculus for a schematic program Π to the SLDNF calculus for $Ground(\Pi)$.

Proposition 4.1. *For any schematic program Π, schematic goal G and substitution δ,*

- *if G succeeds relative to Π with c.a.s. δ then every ground instance of $G\delta$ succeeds relative to $Ground(\Pi)$,*
- *if G fails relative to Π then every ground instance of G fails relative to $Ground(\Pi)$.*

For instance, the derivation above shows that $\{r(x)\}$ succeeds relative to program (55) with c.a.s. $\{x/b\}$; consequently, $\{r(b)\}$ succeeds relative to the corresponding ground program.

Problem 4.10. Find a derivation of $\models \{r(b)\}$ in the SLDNF calculus for this ground program.

Using Propositions 3.13 and 3.14, we conclude:

Corollary 1. *For any schematic program Π and schematic goal G, if G succeeds relative to Π with some c.a.s. then G does not fail relative to Π.*

Corollary 2. *For any schematic program Π, schematic literal L, and substitution δ,*

- *if $\{L\}$ succeeds relative to Π with c.a.s. δ then every ground instance of $L\delta$ is a well-founded consequence of Π,*
- *if Π is consistent and $\{L\}$ fails relative to Π then every ground instance of L is unfounded relative to Π.*

The last corollary expresses the soundness of the SLDNF calculus for consistent schematic programs.

4.4 Prolog

Proposition 3.16 and the thinning rule (Section 3.6) are extended to schematic programs in a straightforward way.

The process of computing answer substitutions in Prolog can be described as follows. Prolog attempts to "evaluate" a schematic goal G, that is, to generate a series of computed answer substitutions for G or to establish that G fails. This is done by backward chaining, and Prolog performs the evaluation correctly as long as it does not "flounder" (see below).[5]

If the given schematic goal G is empty then it succeeds with the c.a.s. ϵ, and the process terminates. Otherwise, the first step is to select a singleton subgoal in G. If the selected subgoal does not contain negation as failure, so that G is represented as $G' \cup \{L\}$, then Prolog takes, one by one, the schematic rules $H \leftarrow B$ applicable to L, computes for each of them $\theta = vr(H \leftarrow B, vars(G' \cup \{L\}))$ and $\sigma = mgu(L, H\theta)$, and starts evaluating $(G' \cup B\theta)\sigma$. Whenever an expression of the form $\models (G' \cup B\theta)\sigma : \delta$ is

[5]Floundering is one of two reasons why a Prolog system can evaluate a schematic goal incorrectly. The second reason is that, for the sake of efficiency, most Prolog implementations omit the occurs check from the unification algorithm.

derived, a new c.a.s. for G is produced by one application of (SP). If $\dashv (G' \cup B\theta)\sigma$ has been derived for each schematic rule applicable to L then the failure of G is concluded by one application of (FP).

If the selected subgoal contains negation as failure, so that G is represented as $G' \cup \{not\ L\}$, then what happens next depends on whether or not L is ground. If it is then Prolog starts evaluating the goal $\{L\}$. If it succeeds then, since L is ground, the only possible c.a.s. for this goal is ϵ, and $\dashv G$ can be derived by (FN). If it fails then Prolog starts evaluating G'. Whenever an expression of the form $\models G' : \delta$ is derived, a new c.a.s. for G is produced by one application of (SN). If $\dashv G'$ is derived then the failure of G is concluded by one application of (T). Finally, if L is nonground then Prolog "flounders." It is not capable of continuing the evaluation process correctly.

As an example, consider the Prolog evaluation of the goal $\{r(x)\}$ for program (55). The process begins with the partial derivation

$$\perp\!\!\!\perp\ \{r(x)\}\ :\ \delta_1$$

where the symbol δ_1 represents an unknown answer substitution; it will be computed later if $\perp\!\!\!\perp$ turns out to represent success. On the first step, $L = r(x)$ and $G' = \emptyset$. The only rule of (55) applicable to L is the last one:

$$H = r(y),\ B = \{p(x,y), not\ q(x)\}.$$

We compute

$$\begin{gathered}\theta = \{x/x_1, x_1/x\},\ \sigma = \{y/x\},\\ (G' \cup B\theta)\sigma = \{p(x_1,x), not\ q(x_1)\},\end{gathered}$$

and arrive at the partial derivation

$$\frac{\perp\!\!\!\perp\ \{p(x_1,x), not\ q(x_1)\}\ :\ \delta_2}{\perp\!\!\!\perp\ \{r(x)\} : \delta_1}$$

with the following equation relating the unknown substitutions δ_1 and δ_2:

$$\delta_1 = \{y/x\}\delta_2 \mid \{x\}. \tag{58}$$

Now $L = p(x_1, x)$ and $G' = \{not\ q(x_1)\}$. The only rule of (55) applicable to L is the first one:

$$H = p(a,b),\ B = \emptyset.$$

We compute

$$\begin{gathered}\theta = \epsilon,\ \sigma = \{x/b, x_1/a\},\\ (G' \cup B\theta)\sigma = \{not\ q(a)\},\end{gathered}$$

and arrive at the partial derivation

$$\cfrac{\cfrac{\perp\!\!\!\perp\ \{not\ q(a)\}\ :\ \delta_3}{\perp\!\!\!\perp\ \{p(x_1,x), not\ q(x_1)\}\ :\ \delta_2}}{\perp\!\!\!\perp\ \{r(x)\}\ :\ \delta_1}$$

with the additional equation

$$\delta_2 = \{x/b, x_1/a\}\delta_3 \mid \{x, x_1\}.$$

After that, we generate the figure

$$\dfrac{\dfrac{\dfrac{\perp\!\!\!\perp \ \{q(a)\} \ : \ \epsilon}{\perp\!\!\!\perp \ \{not\ q(a)\} \ : \ \delta_3}}{\perp\!\!\!\perp \ \{p(x_1, x), not\ q(x_1)\} \ : \ \delta_2}}{\perp\!\!\!\perp \ \{r(x)\} \ : \ \delta_1}$$

Since no rule in (55) is applicable to $q(a)$, the goal $\{q(a)\}$ fails:

$$\dfrac{\dfrac{\dfrac{(FP)\ \dfrac{}{\Dashv \ \{q(a)\}}}{\perp\!\!\!\perp \ \{not\ q(a)\} \ : \ \delta_3}}{\perp\!\!\!\perp \ \{p(x_1, x), not\ q(x_1)\} \ : \ \delta_2}}{\perp\!\!\!\perp \ \{r(x)\} \ : \ \delta_1}$$

The next step takes us to the figure

$$\dfrac{\dfrac{\dfrac{\perp\!\!\!\perp \ \emptyset \ : \ \delta_4 \qquad (FP)\ \dfrac{}{\Dashv \ \{q(a)\}}}{\perp\!\!\!\perp \ \{not\ q(a)\} \ : \ \delta_3}}{\perp\!\!\!\perp \ \{p(x_1, x), not\ q(x_1)\} \ : \ \delta_2}}{\perp\!\!\!\perp \ \{r(x)\} \ : \ \delta_1}$$

with the equation

$$\delta_3 = \delta_4.$$

Finally we observe that $\emptyset$ succeeds with c.a.s. ϵ and arrive at the derivation

$$(SP)\ \dfrac{(SP)\ \dfrac{(SN)\ \dfrac{\models \ \emptyset \ : \ \delta_4 \qquad (FP)\ \dfrac{}{\Dashv \ \{q(a)\}}}{\models \ \{not\ q(a)\} \ : \ \delta_3}}{\models \ \{p(x_1, x), not\ q(x_1)\} \ : \ \delta_2}}{\models \ \{r(x)\} \ : \ \delta_1}$$

and the equation

$$\delta_4 = \epsilon.$$

The values of $\delta_1, \ldots, \delta_4$ can be determined from the equations given above. Having done this, we will see that this is the same derivation as the one given in Section 4.3.

In this example, Prolog has computed only one answer substitution for the given goal. To see how it can produce several answer substitutions, consider the following enhancement of (55):

$$\begin{array}{l} p(a,b), \\ p(x,c) \leftarrow p(x,b), \\ q(b), \\ r(y) \leftarrow p(x,y), not\ q(x). \end{array}$$

The process will proceed as described above, but it will not stop here, because the new program has a second rule applicable to $p(x_1, x)$. We go back to the step after equation (58) and compute:

$$\begin{gathered} L = p(x_1, x),\ G' = \{not\ q(x_1)\}, \\ H = p(x,c),\ B = \{p(x,b)\}, \\ \theta = \{x/x_2, x_2/x\},\ \sigma = \{x/c, x_2/x_1\}, \\ (G' \cup B\theta)\sigma = \{p(x_1,b), not\ q(x_1)\}. \end{gathered}$$

This leads to the partial derivation

$$\frac{\dfrac{\perp\!\!\!\perp\ \{p(x_1,b), not\ q(x_1)\}\ :\ \delta_3'}{\perp\!\!\!\perp\ \{p(x_1,x), not\ q(x_1)\}\ :\ \delta_2}}{\perp\!\!\!\perp\ \{r(x)\}\ :\ \delta_1}$$

with the additional equation

$$\delta_2 = \{x/c, x_2/x_1\}\delta_3' \mid \{x, x_1\}.$$

This is a path to a new chain of partial derivations and to a second computed answer substitution.

Problem 4.11. Find these partial derivations and the substitution.

Problem 4.12. Describe the work of Prolog on the program obtained from (55) by dropping the second rule, and the same goal as above.

5 Disjunctive Programs and Default Theories

In this section, the class of programs with negation as failure is extended in two different directions. In a "disjunctive program," the head of a rule, like the body, is allowed to be a finite set of rule elements, rather than a single literal. In a "default theory," arbitrary propositional formulas can be used in place of literals.

5.1 Disjunctive Programs

A *disjunctive rule* is an ordered pair $Head \leftarrow Body$, where $Head$ and $Body$ are finite sets of rule elements. A rule $L \leftarrow Body$ in the sense of Section 3.1 will be identified with the disjunctive rule $\{L\} \leftarrow Body$. A disjunctive rule can be represented in the form

$$HPos \cup not(HNeg) \leftarrow BPos \cup not(BNeg) \tag{59}$$

for some finite sets of literals $HPos$, $HNeg$, $BPos$, $BNeg$. The rule with the head $\{L_1, \dots, L_k, not\ L_{k+1}, \dots, not\ L_l\}$ and the body $\{L_{l+1}, \dots, L_m, not\ L_{m+1}, \dots, not\ L_n\}$ will be also written as

$$L_1 \mid \cdots \mid L_k \mid not\ L_{k+1} \mid \cdots \mid not\ L_l \leftarrow L_{l+1}, \dots, L_m, not\ L_{m+1}, \dots, not\ L_n \tag{60}$$

($\mid$ reads "or").

A *disjunctive program* is a set of disjunctive rules. For instance, here is a disjunctive program without negation as failure:

$$\begin{array}{l} p \mid q \leftarrow, \\ \neg r \leftarrow p. \end{array} \tag{61}$$

In the next example, the head of one of the disjunctive rules is empty; such rules are called *constraints:*

$$\begin{array}{l} p \leftarrow not\ q, \\ q \leftarrow not\ p, \\ \leftarrow p. \end{array} \tag{62}$$

A disjunctive rule may contain the negation as failure symbol in the head, as, for instance, the second rule of the program

$$\begin{array}{l} q \leftarrow p, \\ p \mid not\ p \leftarrow . \end{array} \tag{63}$$

The notion of an answer set is defined for disjunctive programs in two steps. First we give the definition for disjunctive programs without negation as failure, such as (61). Then it is extended to the general case by means of a process similar to the one used in Section 3.1.

Let Π be a disjunctive program without negation as failure. About a set X of literals we say that it is *closed under* Π if, for every disjunctive rule $Head \leftarrow Body$ in Π, $Head \cap X \neq \emptyset$ whenever $Body \subseteq X$. We say that X is an *answer set* for Π if it is a minimal (relative to set inclusion) set of literals that is both closed under Π and logically closed.

For instance, the answer sets for program (61) are $\{p, \neg r\}$ and $\{q\}$.

The *reduct* of a disjunctive program Π relative to a set of literals X is the disjunctive program without negation as failure obtained from Π by

- deleting each disjunctive rule (59) such that $HNeg \not\subseteq X$ or $BNeg \cap X \neq \emptyset$, and

- replacing each remaining disjunctive rule (59) by $HPos \leftarrow BPos$.

This disjunctive program will be denoted by Π^X. We say that X is an *answer set* for Π if X is an answer set for Π^X. A *consequence* of a disjunctive program is a literal that belongs to all its answer sets.

When applied to a program in the sense of Section 3.1, these definitions are clearly equivalent to the ones given there.

Problem 5.1. Find the answer sets for the disjunctive program

$$\begin{aligned}
&p \mid q \leftarrow, \\
&\neg p \leftarrow not\ p, \\
&\neg q \leftarrow not\ q, \\
&r \leftarrow p, \\
&r \leftarrow q.
\end{aligned}$$

It is easy to check that $\{q\}$ is the only answer set for program (62). The effect of adding the constraint $\leftarrow p$ to the first two rules of (62) is to eliminate the answer set that includes p. This is an instance of the general fact stated below as Proposition 5.1.

About a set X of literals we say that it *violates* a constraint

$$\leftarrow Pos \cup not(Neg)$$

if $Pos \subseteq X$ and $X \cap Neg = \emptyset$. Otherwise, X *satisfies* the constraint.

Proposition 5.1. *Let Π be a disjunctive program, and C a set of constraints. A set of literals is an answer set for $\Pi \cup C$ iff it is an answer set for Π and satisfies all constraints in C.*

Program (63) has two answer sets: $\emptyset$ and $\{p, q\}$. This example shows that the assertion of Proposition 3.1 does not generally hold for disjunctive programs with the negation as failure symbol in the head.

5.2 Default Logic

A *default* is an expression of the form

$$\frac{F \quad not\ G_1 \quad \cdots \quad not\ G_n}{H} \tag{64}$$

where $F, G_1, \ldots, G_n, H$ are propositional formulas, $n \geq 0$. We will drop F if it equals *true*. A default of the form

$$\frac{}{H}$$

will be identified with the formula H.

A *default theory* is a set of defaults. For instance, here is a default theory:

$$\left\{ p \vee q, \frac{not\ p}{\neg p} \right\}. \tag{65}$$

A rule

$$L_0 \leftarrow L_1, \ldots, L_m, \mathit{not}\ L_{m+1}, \ldots, \mathit{not}\ L_n$$

will be identified with the default

$$\frac{L_1 \wedge \cdots \wedge L_m \quad \mathit{not}\ L_{m+1} \quad \cdots \quad \mathit{not}\ L_n}{L_0}.$$

According to this convention, programs in the sense of Section 3 are default theories of a special syntactic form.

Our syntax of defaults stresses their similarity to rules in logic programming and is somewhat different from the standard one. Traditionally, defaults are defined as expressions of the form

$$\frac{F \ :\ \mathsf{M}\ G_1 \quad \cdots \quad \mathsf{M}\ G_n}{H},$$

where M expresses "consistency," or simply

$$\frac{F \ :\ G_1 \quad \cdots \quad G_n}{H}.$$

In our notation, this expression corresponds to

$$\frac{F \quad \mathit{not}\ \neg G_1 \quad \cdots \quad \mathit{not}\ \neg G_n}{H}.$$

Let T be a default theory. The function Γ_T from sets of formulas to sets of formulas is defined as follows: For any set of formulas X, $\Gamma_T X$ is the smallest set of formulas Y such that

- for every default (64) in T, if $F \in Y$ and $G_1, \ldots, G_n \notin X$ then $H \in Y$,
- Y is logically closed (in the sense of propositional logic).

A set of X of formulas is an *extension* for T if $\Gamma_T X = X$. A *consequence* of a default theory is a formula that belongs to all its extensions.

If, for example, T is (65) then, for any set of formulas X,

$$\Gamma_T X = \begin{cases} \mathit{Closure}(\{p \vee q\}), & \text{if } p \in X, \\ \mathit{Closure}(\{p \vee q, \neg p\}), & \text{otherwise,} \end{cases}$$

where *Closure* denotes the deductive closure in the sense of propositional logic. It is clear that the only extension for (65) is $\mathit{Closure}(\{p \vee q, \neg p\})$, that is, $\mathit{Closure}(\{\neg p, q\})$. The consequences of this theory are the consequences of $\neg p$ and q in the sense of propositional logic.

Problem 5.2. Find the extensions for the default theory

$$\left\{ p \vee q, \frac{\mathit{not}\ p}{\neg p}, \frac{\mathit{not}\ q}{\neg q} \right\}.$$

Default logic is an extension of propositional logic: If T is a set of formulas then the only extension for T is $\mathit{Closure}(T)$. If, on the other hand, T is a program then there is a simple one-to-one correspondence between its answer sets and its extensions in the sense of default logic:

Proposition 5.2. *For any program* Π,

- *if* X *is an answer set for* Π *then* $Closure(X)$ *is an extension for* Π,
- *every extension for* Π *has the form* $Closure(X)$ *for exactly one answer set* X *for* Π.

6 Bibliographical and Historical Remarks

The use of logic-based languages for representing declarative knowledge was proposed by McCarthy [1959] a few years before the creation of first relational databases. Connecting the two approaches using the domain closure assumption and the unique name assumption is the idea of Reiter [1984].

Credit for founding the field of logic programming is usually given to Kowalski and Colmerauer, whose early work on this subject was done in the mid-seventies. Kowalski, in the words of Minker [1988], "was the visionary who kept doggedly speaking about logic programming until the field became recognized as important." The first Prolog interpreter was designed by Colmerauer in 1973; in 1977, David L. D. Warren developed a more efficient implementation of Prolog and made it available to others. The second query evaluation method mentioned here, SLG, was designed and implemented by Chen and David S. Warren [Chen *et al.*, 1995].

The idea of negation as failure was introduced by Clark [1978], and the closed world assumption by Reiter [1978]. Classical negation was incorporated into the syntax of logic programming rules by Gelfond and Lifschitz [1990]; a similar idea was independently developed by Pearce and Wagner [1990].

Head-consistency was introduced in [Turner, 1994]. The properties of T_Π and φ are due to van Emden and Kowalski [1976].

Definitions equivalent to the definition of an answer set for normal programs were proposed independently by Bidoit and Froidevaux [1987] and Gelfond [1987]. The definition used in this survey was given by Gelfond and Lifschitz, first for normal programs [1988] (where answer sets were called "stable models") and then for programs with both kinds of negation [1990].

Clark [1978] defined completion—not only for propositional programs, as in this survey, but also for programs with variables—and proposed to view any (normal) program as shorthand for the corresponding first-order theory, the "completed database." Historically, this was the first declarative semantics for programs with negation as failure. Supported sets were defined in [Apt *et al.*, 1988]. Well-founded and unfounded literals were introduced (for normal programs) in [Van Gelder *et al.*, 1990] and [Przymusinski, 1991]. This concept is often used as a semantical foundation for logic programming. According to the "well-founded semantics," only well-founded literals are counted as consequences of the program. This approach and its extensions are applied to knowledge representation in [Pereira *et al.*, 1993].

The discussion of splitting in Sections 2.4 and 3.4 is based on [Lifschitz and Turner, 1994]. The elimination step performed as part of splitting is an example of "partial deduction" [Komorowski, 1990].

Hierarchical programs were defined in [Cavedon, 1989], on the basis of a special case defined in [Clark, 1978]. Another special case of hierarchical programs was studied and applied to reasoning about action in [Apt and Bezem, 1990]. Tight programs were introduced by Fages [1994], under the name "positive-order-consistent." The notion of a stratified program was developed in a series of papers by Chandra and Harel [1985], Apt, Blair and Walker [1988], Van Gelder [1988] and Przymusinski [1988]. Strict programs were defined in [Apt *et al.*, 1988], and order-consistent programs in [Sato, 1990]. Propositions 3.5 and 3.12 are from [Fages, 1994]. The idea of tightening and Proposition 3.6 belong to Wallace [1993].

The reduction of arbitrary programs to normal programs was proposed in [Gelfond and Lifschitz, 1990]. The counterexample following Corollary to Proposition 3.17 is due to Hudson Turner (personal communication) who uncovered an error in the printed version of this survey. Proposition 3.18 is from [Gelfond and Lifschitz, 1988].

The abnormality predicate was first used by McCarthy [1986] in the context of his theory of circumscription.

Our treatment of Prolog is based on the ideas of [Mints, 1986] (translated into English as [Mints, 1990]) and [Kunen, 1989]. SLDNF calculi are further generalized in [Lifschitz, 1995] and [Lifschitz *et al.*, 1995].

The notion of an answer set for disjunctive programs without negation as failure in the heads of rules was defined in [Gelfond and Lifschitz, 1991]. This last limitation was removed by Lifschitz and Woo [1992]. Inoue and Sakama [1994] related negation as failure in heads to the important area of "abductive logic programming" [Kakas *et al.*, 1992].

Default logic was invented by Reiter [1980]. His definition of an extension was, historically, a source of the idea of an answer set; the work by Bidoit and Froidevaux mentioned above was based on the reduction of logic programs to default theories. In a similar way, Gelfond's paper related logic programs to another nonmonotonic formalism whose semantics is defined by a fixpoint construction—to the system of autoepistemic logic introduced by Moore [1985].

The monographs [Lloyd, 1987] and [Lobo *et al.*, 1992] use the approaches to the semantics of logic programming different from ours. A survey of early work on negation as failure can be found in [Shepherdson, 1988].

The 1994 special issue of the *Journal of Logic Programming* and its continuation, celebrating the tenth anniversary of the journal, contain, among others, three surveys closely related to this one—by Apt and Bol [1994], by Baral and Gelfond [1994], and by Ramakrishnan and Ullman [1995]. They provide somewhat different perspectives and contain extensive bibliographies.

Acknowledgements

Drafts of this survey were used in the courses taught at the University of Texas and at the Fifth European Summer School in Language, Logic and Information, and I am grateful to my students for their criticisms and suggestions. I would also like to thank Michael Gelfond, Norman McCain and Hudson Turner for their useful comments. This work was partially supported by National Science Foundation under grant IRI-9306751.

Appendix. Monotone Functions

Consider an arbitrary set Ω and a function T from subsets of Ω to subsets of Ω. A subset X of Ω is a *pre-fixpoint* of T if $TX \subseteq X$; X is a *post-fixpoint* of T if $X \subseteq TX$. Thus X is a fixpoint of T ($TX = X$) iff it is both a pre-fixpoint and post-fixpoint of T.

We say that T is *monotone* if, for any subsets X, Y of Ω, $TX \subseteq TY$ whenever $X \subseteq Y$. The following fact is known as the *Knaster-Tarski theorem* [Tarski, 1955]:

Proposition A.1. *Every monotone function has*

- *a least fixpoint, that is also its least pre-fixpoint, and*
- *a greatest fixpoint, that is also its greatest post-fixpoint.*

These fixpoints can be approached (but not necessarily reached) by iterating T:

Proposition A.2. *For every monotone function T,*

- *the sequence $(T^n\emptyset)_{n\geq 0}$ is increasing (that is, every element of this sequence is a subset of the next one), and its union a subset of the least fixpoint of T,*
- *the sequence $(T^n\Omega)_{n\geq 0}$ is decreasing (that is, every element of this sequence is a superset of the next one), and its intersection is a superset of the greatest fixpoint of T.*

Problem A.1. Let Ω be the set of natural numbers. (a) Give an example of a monotone function T such that all sets $T^n\emptyset$ $(n \geq 0)$ are different from each other, and the union of these sets is a fixpoint of T. (b) Give an example of a monotone function T such that all sets $T^n\Omega$ $(n \geq 0)$ are different from each other, and the intersection of these sets is a fixpoint of T.

Problem A.2. Let Ω be the set of natural numbers. (a) Give an example of a monotone function T such that the union of the sets $T^n\emptyset$ $(n \geq 0)$ is not a fixpoint of T. (b) Give an example of a monotone function T such that the intersection of the sets $T^n\Omega$ $(n \geq 0)$ is not a fixpoint of T.

If Ω is finite then the least and the greatest fixpoints of a monotone function T can be computed by iterating T a finite number of times: For all sufficiently large n, $T^n\emptyset$ equals the least fixpoint of T, and $T^n\Omega$ equals the greatest fixpoint of T.

A function T from subsets of Ω to subsets of Ω is *anti-monotone* if, for any subsets X, Y of Ω, $TY \subseteq TX$ whenever $X \subseteq Y$. It is clear that, for any anti-monotone function T, T^2 is monotone.

Proposition A.3. *Let T be an anti-monotone function, and let X_0 and X_1 be the least and the greatest fixpoints of T^2. Then*

- $TX_0 = X_1$, $TX_1 = X_0$,
- *for any fixpoint X of T, $X_0 \subseteq X \subseteq X_1$.*

Corollary. *For any anti-monotone function T, if X is the only fixpoint of T^2 then X is the only fixpoint of T.*

If Ω is finite then, for any anti-monotone function T, the least and the greatest fixpoints of T^2 can be computed by iterating T a finite number of times, as follows. Compute the sets $T^n\emptyset$ $(n = 0, 1, \ldots)$ until a value of n is found for which $T^n\emptyset = T^{n+2}\emptyset$. Then one of the sets $T^n\emptyset$, $T^{n+1}\emptyset$ is the least fixpoint of T^2, and the other is the greatest fixpoint of T^2, depending on whether n is even or odd. Alternatively, these fixpoints can be found by iterating T on Ω.

References

[Apt and Bezem, 1990] Krzysztof Apt and Marc Bezem. Acyclic programs. In David Warren and Peter Szeredi, editors, *Logic Programming: Proc. Seventh Int'l Conf.*, pages 617–633, 1990.

[Apt and Bol, 1994] Krzysztof Apt and Ronald Bol. Logic programming and negation: a survey. *Journal of Logic Programming*, 19,20:9–71, 1994.

[Apt *et al.*, 1988] Krzysztof Apt, Howard Blair, and Adrian Walker. Towards a theory of declarative knowledge. In Jack Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 89–148. Morgan Kaufmann, San Mateo, CA, 1988.

[Baral and Gelfond, 1994] Chitta Baral and Michael Gelfond. Logic programming and knowledge representation. *Journal of Logic Programming*, 19,20:73–148, 1994.

[Bidoit and Froidevaux, 1987] Nicole Bidoit and Christine Froidevaux. Minimalism subsumes default logic and circumscription. In *Proc. LICS-87*, pages 89–97, 1987.

[Cavedon, 1989] Lawrence Cavedon. Continuity, consistency and completeness properties for logic programs. In Giorgio Levi and Maurizio Martelli, editors, *Logic Programming: Proc. Sixth Int'l Conf.*, pages 89–97, 1989.

[Chandra and Harel, 1985] Ashok Chandra and David Harel. Horn clause queries and generalizations. *Journal of Logic Programming*, 2(1):1–5, 1985.

[Chen *et al.*, 1995] Weidong Chen, Terrance Swift, and David Warren. Efficient top-down computation of queries under the well-founded semantics. *Journal of Logic Programming*, 24:161–199, 1995.

[Clark, 1978] Keith Clark. Negation as failure. In Herve Gallaire and Jack Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, New York, 1978.

[Emden and Kowalski, 1976] Maarten van Emden and Robert Kowalski. The semantics of predicate logic as a programming language. *Journal of the ACM*, 23(4):733–742, 1976.

[Fages, 1994] François Fages. Consistency of Clark's completion and existence of stable models. *Journal of Methods of Logic in Computer Science*, 1:51–60, 1994.

[Fitting, 1990] Melvin Fitting. *First-Order Logic and Automated Theorem Proving.* Springer-Verlag, 1990.

[Gelfond and Lifschitz, 1988] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In Robert Kowalski and Kenneth Bowen, editors, *Logic Programming: Proc. Fifth Int'l Conf. and Symp.*, pages 1070–1080, 1988.

[Gelfond and Lifschitz, 1990] Michael Gelfond and Vladimir Lifschitz. Logic programs with classical negation. In David Warren and Peter Szeredi, editors, *Logic Programming: Proc. Seventh Int'l Conf.*, pages 579–597, 1990.

[Gelfond and Lifschitz, 1991] Michael Gelfond and Vladimir Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9:365–385, 1991.

[Gelfond, 1987] Michael Gelfond. On stratified autoepistemic theories. In *Proc. AAAI-87*, pages 207–211, 1987.

[Inoue and Sakama, 1994] Katsumi Inoue and Chiaki Sakama. On positive occurrences of negation as failure. In *Proc. Fourth Int'l Conf. on Principles of Knowledge Representation and Reasoning*, pages 293–304, 1994.

[Kakas *et al.*, 1992] Antonis Kakas, Robert Kowalski, and F. Toni. Abductive logic programming. *Journal of Logic and Computation*, 2(6):719–770, 1992.

[Komorowski, 1990] Jan Komorowski. Towards a programming methodology founded on partial deduction. In *Proc. ECAI-90*, 1990.

[Kunen, 1989] Kenneth Kunen. Signed data dependencies in logic programs. *Journal of Logic Programming*, 7(3):231–245, 1989.

[Lifschitz and Turner, 1994] Vladimir Lifschitz and Hudson Turner. Splitting a logic program. In Pascal Van Hentenryck, editor, *Proc. Eleventh Int'l Conf. on Logic Programming*, pages 23–37, 1994.

[Lifschitz and Woo, 1992] Vladimir Lifschitz and Thomas Woo. Answer sets in general nonmonotonic reasoning (preliminary report). In Bernhard Nebel, Charles Rich, and William Swartout, editors, *Proc. Third Int'l Conf. on Principles of Knowledge Representation and Reasoning*, pages 603–614, 1992.

[Lifschitz *et al.*, 1995] Vladimir Lifschitz, Norman McCain, Teodor Przymusinski, and Robert Stärk. Loop checking and the well-founded semantics. In *Logic Programming and Non-monotonic Reasoning: Proceedings of the Third International Conference*, pages 127–142, 1995.

[Lifschitz, 1995] Vladimir Lifschitz. SLDNF, constructive negation and grounding. In *Proc. ICLP-95*, pages 581–595, 1995.

[Lloyd, 1987] John Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 1987. Second, extended edition.

[Lobo *et al.*, 1992] Jorge Lobo, Jack Minker, and Arcot Rajasekar. *Foundations of Disjunctive Logic Programming.* MIT Press, 1992.

[McCarthy, 1959] John McCarthy. Programs with common sense. In *Proc. Teddington Conference on the Mechanization of Thought Processes*, pages 75–91, London, 1959. Her Majesty's Stationery Office. Reproduced in [McCarthy, 1990].

[McCarthy, 1986] John McCarthy. Applications of circumscription to formalizing common sense knowledge. *Artificial Intelligence*, 26(3):89–116, 1986. Reproduced in [McCarthy, 1990].

[McCarthy, 1990] John McCarthy. *Formalizing Common Sense: Papers by John McCarthy.* Ablex, Norwood, NJ, 1990.

[Minker, 1988] Jack Minker. Perspectives in deductive databases. *Journal of Logic Programming*, 5:33–60, 1988.

[Mints, 1986] Grigori Mints. A complete calculus for pure Prolog. *Proc. Academy of Sciences of Estonian SSR*, 35(4):367–380, 1986. In Russian.

[Mints, 1990] Grigori Mints. Several formal systems of logic programming. *Computers and Artificial Intelligence*, 9(1):19–41, 1990.

[Moore, 1985] Robert Moore. Semantical considerations on nonmonotonic logic. *Artificial Intelligence*, 25(1):75–94, 1985.

[Pearce and Wagner, 1990] David Pearce and Gerd Wagner. Reasoning with negative information I: Strong negation in logic programs. *Acta Philosophica Fennica*, 49, 1990.

[Pereira *et al.*, 1993] Luis Pereira, Joaquim Aparicio, and Jose Alferes. Nonmonotonic reasoning with logic programming. *Journal of Logic Programming*, 17:227–263, 1993.

[Przymusinski, 1988] Teodor Przymusinski. On the declarative semantics of deductive databases and logic programs. In Jack Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 193–216. Morgan Kaufmann, San Mateo, CA, 1988.

[Przymusinski, 1991] Teodor Przymusinski. Stable semantics for disjunctive programs. *New Generation Computing*, 9:401–424, 1991.

[Ramakrishnan and Ullman, 1995] Raghu Ramakrishnan and Jeffrey Ullman. A survey of deductive database systems. *Journal of Logic Programming*, 23:125–150, 1995.

[Reiter, 1978] Raymond Reiter. On closed world data bases. In Herve Gallaire and Jack Minker, editors, *Logic and Data Bases*, pages 119–140. Plenum Press, New York, 1978.

[Reiter, 1980] Raymond Reiter. A logic for default reasoning. *Artificial Intelligence*, 13:81–132, 1980.

[Reiter, 1984] Raymond Reiter. Towards a logical reconstruction of relational database theory. In M.L. Brodie, J. Mylopoulos, and J.W. Schmidt, editors, *On Conceptual Modelling: Perspectives from Artificial Intelligence, Databases and Programming Languages*, pages 191–233. Springer-Verlag, 1984.

[Sato, 1990] Taisuke Sato. Completed logic programs and their consistency. *Journal of Logic Programming*, 9:33–44, 1990.

[Shepherdson, 1988] John Shepherdson. Negation in logic programming. In Jack Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 19–88. Morgan Kaufmann, San Mateo, CA, 1988.

[Tarski, 1955] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.

[Turner, 1994] Hudson Turner. Signed logic programs. In *Proc. ILPS-94*, pages 61–75, 1994.

[Van Gelder *et al.*, 1990] Allen Van Gelder, Kenneth Ross, and John Schlipf. The well-founded semantics for general logic programs. *Journal of ACM*, pages 620–650, 1990.

[Van Gelder, 1988] Allen Van Gelder. Negation as failure using tight derivations for general logic programs. In Jack Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 149–176. Morgan Kaufmann, San Mateo, CA, 1988.

[Wallace, 1993] Mark Wallace. Tight, consistent and computable completions for unrestricted logic programs. *Journal of Logic Programming*, 15:243–273, 1993.