Deductive Computing over Knowledge Bases: Prolog and Datalog

Luis M. AUGUSTO[*]

Editor-in-Chief

Journal of Knowledge Structures & Systems

July 2024

Vol.: 5 Issue: 1 Pages: 1-62

Abstract

Knowledge representation (KR) is actually more than representation: It involves also inference, namely inference of "new" knowledge, i.e. new facts. Logic programming is a suitable KR medium, but more often than not discussions on this programming paradigm focus on aspects other than KR. In this paper, I elaborate on the general theory of logic programming and give the essentials of two of its main implementations, to wit, Prolog and Datalog, from the viewpoint of deductive computing over knowledge bases, which includes deductive programming.

Key words: Logic Programming; Knowledge Representation (KR); Deductive Computation; Deductive Programming; Prolog; Datalog

1 Introduction

Logic programming (LP) is a programming paradigm based on the notion of a program as defining a set of logical consequences, so that a computation is actually a deduction of consequences of the program.[1] This is called *deductive computation* in general; more narrowly, one speaks of *deductive programming.* In effect, a *logic program* is a set of formulae that are rules or axioms. In particular, these rules and axioms are so of relations between objects, reason why it should be obvious that logic programming languages rely heavily on first- or even second-order logic.[2]

A. Newell influentially defined the relationship between *knowledge* and *representation* by means of the equation

[*]✉ luis.ml.augusto@gmail.com

[1]See Augusto (2020b) for a monograph on logical consequence.

[2]I restrict my discussion to first-order predicate logic. I refer the reader to Augusto (2019) for a textbook on classical first-order logic (CFOL).

$$\textit{Representation} = \textit{Knowledge} + \textit{Access}$$

in which knowledge is defined as the abstract set of all the possible expressions – including new ones – that can be derived from a symbolic structure (such as a knowledge base). The term *derivation* is to be understood here in its logical sense, and this – entailment or proof – is what Newell meant by *access.* Without this *logical level* one could not attain the *knowledge level* of a system, that which makes of a system, either organic (say, an ant) or mechanical (e.g., a robot), an *intelligent system.*[3] Although A. Newell segregated the two levels clearly, apparently favoring other kinds of programming language over a logical language as a KR medium at the knowledge level, LP has been proved to be highly adequate for KR.[4] In this introduction to LP, I elaborate on its general theory and its adequacy to provide us with a suitable KR medium, namely by means of *deductive knowledge bases*, i.e. knowledge bases (abbr.: KBs) over which deduction is carried out. I shall here consider a KB as a major component of a *knowledge system* taken as a triple

$$\mathbb{K} = (\mathscr{KS}, \mathscr{KP}, \mathscr{KA})$$

where $\mathscr{KS}$ is a collection of *knowledge structures*, $\mathscr{KP}$ is a collection of *knowledge processes*, and $\mathscr{KA}$ is a collection of *knowledge agents.* (Figure 1 shows a standard knowledge system.) I shall consider that the main knowledge structure is a KB, a symbolic structure constituted by facts and rules expressed in the standard language of first-order predicate logic or a fragment thereof; the core knowledge process is deduction (or inference); the main task of the knowledge agent (KA) is that of querying the (updated) KB, denoted by $\Xi^{(i)}$.

I shall further consider that there are formal ways to (i) distinguish a fact (i.e. a knowledge unit) from a datum and (ii) convert data into facts and the other way round (see Augusto, 2020c). Hence, I shall see collections of data – programs or databases – as (potentially) KBs, especially the elements thereof that are seen as facts, and I speak of programs and databases mostly for consistency with the standard literature.

Created in the early 1970s, the fact that this programming paradigm not only has resisted the test of time but has now a large plethora of recent applications (see, e.g., Kifer & Liu, 2018) justifies yet another review on it.[5] I first approach LP at the metalanguage level, and then discuss it at the object-language level. This latter discussion focuses on more strictly logical aspects of LP. As for the implementations, I give the essentials of the main LP language, to wit, Prolog, as well as of Datalog. The objective is to enable the reader to understand the main theoretical aspects of this programming paradigm that, despite being spoken of as *the declarative paradigm*, is often given in the literature by the equation

$$\textit{Algorithm} = \textit{Logic} + \textit{Control}.$$

[3]Newell (1980, 1982, 1990) comprise to a great extent the central aspects of A. Newell's perspective. See Augusto (2021) for an analysis of this perspective and its lasting influence in KR and knowledge-based AI. ("KR" is the common acronym for "knowledge representation." Perhaps needless to say – but one never knows – "AI" is so for "Artificial Intelligence.")

[4]See on this subject, for example, Brewka & Dix (2005).

[5]The aim of this paper is not "teaching" how to program with Prolog and Datalog. This said, references that can be used for this aim are given below.

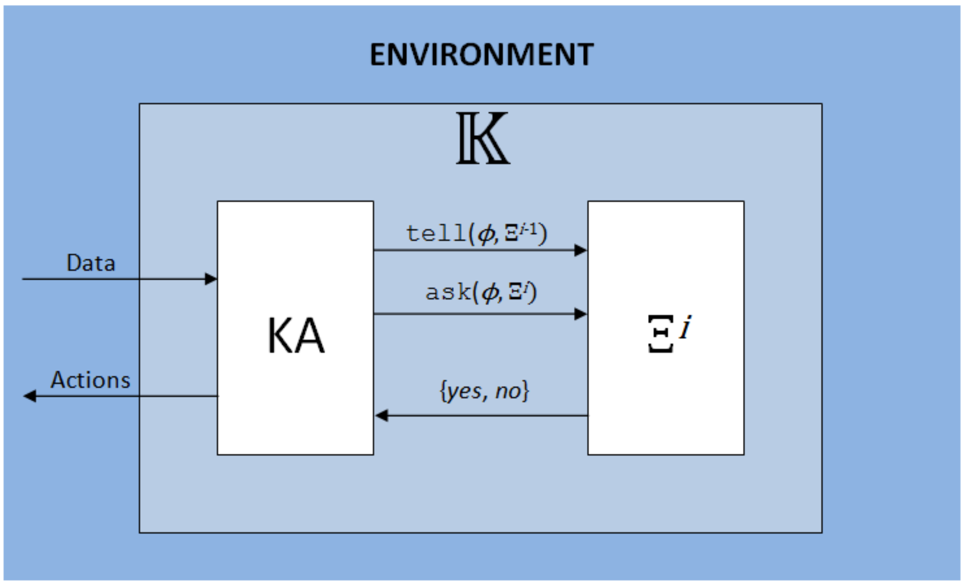


Figure 1: A standard knowledge system. (Source: Augusto, 2020c.)

2 Logic programming as deductive programming

2.1 Query systems and programming systems

The notion of deductive programming finds theoretical support on a few rather intuitive analogies, or equivalences, between the logical and the computing domains: Besides the already mentioned notions of a *program* as a *set of formulae* and a *computation* as a *proof*, further equivalences can be established between *inputs* and *queries*, as well as between *outputs* and *replies* to queries, where both queries and replies are formulae.

These intuitive analogies or equivalences are also clearly formalizable. I begin by defining a proof system.

Definition 1. A *proof system* is a triple $\mathcal{P} = (\Theta, P, \vdash)$ where Θ is a set of formulae, $P = \{\blacksquare_1, ..., \blacksquare_k\}$ is a set of proofs, and $\vdash \subseteq 2^{\Theta} \times P \times \Theta$ is a ternary relation of *syntactical consequence* satisfying the following conditions for any sets $X, Y \subseteq \Theta$ of formulae and for arbitrary formulae $\phi, \chi \in \Theta$:

(R1) If $\phi \in X$, then $X \vdash_{\blacksquare} \phi$
(R2) If $X \vdash_{\blacksquare} \phi$ and $X \subseteq Y$, then $Y \vdash_{\blacksquare} \phi$
(R3) If $X \vdash_{\blacksquare} \phi$ and $Y \vdash_{\blacksquare} \chi$ for every $\chi \in X$, then $Y \vdash_{\blacksquare} \phi$

For some proof $\blacksquare \in P$, the relation $\Theta \vdash_{\blacksquare} \phi$ means that $\blacksquare$ is a proof of ϕ from premises in Θ. $\mathcal{P}$ is a *monotonic* proof system if it is verified that, for $F \subseteq \Theta$ and $\blacksquare \in P$,

$$\text{if } F \vdash_{\blacksquare} \phi \text{ and } F \subseteq G, \text{ then } G \vdash_{\blacksquare} \phi.$$

Proofs and the syntactical consequence are well known concepts of formal logic (see Augusto, 2019; 2020b) and one entails the other, so that we can write simply $X \vdash \phi$ to denote that there is a proof that ϕ follows from X. I now formalize the less known logical objects *queries* and *replies*, as well as the relation between both.

Definition 2. Let the symbol ¿ denote a *query*. A *query system* is a triple $\mathcal{Q} = (\Theta_{¿}, Q, \gtrdot)$, where $\Theta_{¿}$ is a set of formulae that are replies to queries, Q is a set of queries, and $\gtrdot \subseteq Q \times \Theta_{¿}$ is a binary relation.

Definition 3. A *proof system for a query system* $\mathcal{Q} = (\Theta_{¿}, Q, \gtrdot)$ is a triple $\mathcal{P}_{\mathcal{Q}} = (\Theta, P, \vdash)$ where $\Theta \supseteq \Theta_{¿}$ such that $\phi \in \Theta_{¿}$ is a *provably correct* reply to a query $¿ \in Q$ if, for some $F \subset 2^{\Theta}$, we have

$$¿ \gtrdot F \vdash_{\blacksquare} \phi$$

formalizing the fact that, given the premises (of Θ) in F, we may have a proof $\blacksquare \in P$ that ϕ *as a reply* (denoted by $\gtrdot$) to the query ¿. This can be specified according to the condition

$$¿ \gtrdot F \vdash_{\blacksquare} \phi \quad \text{iff} \quad ¿ \gtrdot \phi \text{ and } F \vdash_{\blacksquare} \phi.$$

I now introduce the fundamental notion of a programming system.

Definition 4. A *programming system* is a 6-tuple

$$\mathfrak{P} = (\underline{\Pi}, I, O, \Xi, \rhd, \vdash)$$

where $\underline{\Pi} = \{\Pi_1, ..., \Pi_n\}$ is a set of programs, $I = \{\iota_1, ..., \iota_n\}$ is a set of inputs, $O = \{o_1, ..., o_n\}$ is a set of outputs, $\Xi = \{\xi_1, ..., \xi_n\}$ is a set of computations, $\vdash$ is a ternary relation on $\underline{\Pi} \times \Xi \times O$ such that for each $\Pi \in \underline{\Pi}$ there is a $\xi \in \Xi$ and an $o \in O$ with

$$(\mathfrak{P}_{\vdash}) \qquad \Pi \vdash_{\xi} o$$

and $\rhd$ is a relation on $I \times \mathfrak{P}_{\vdash}$ such that for each $\iota \in I$ we have

$$\iota \rhd \Pi \vdash_{\xi} o$$

denoting that given input ι one possible computation ξ carried out by program Π yields output o. If $\iota \rhd \Pi \vdash o$ iff there is a computation $\xi \in \Xi$ such that $\iota \rhd \Pi \vdash_{\xi} o$, then this defines the *computed-output* relation on $I \times \underline{\Pi} \times O$.

The above definition shows clearly that deductive programming is a specific instance of the more general concept of deductive computation.

Definition 5. A programming system $\mathfrak{P} = (\underline{\Pi}, I, O, \Xi, \rhd, \vdash)$ is said to be *deterministic* iff, for each input $\iota \in I$ and program $\Pi \in \underline{\Pi}$, there is a *unique* computation $\xi \in \Xi$ such that $\iota \rhd \Pi \vdash_{\xi} (o)$, and it is said to be *determinate* iff for each input $\iota \in I$ and each program $\Pi \in \underline{\Pi}$ there is *at most one* output $o \in O$ such that $\iota \rhd \Pi \vdash_{(\xi)} o$. These are the properties of *determinism* and *determinacy*, respectively, of a programming system.

We have the following result:[6]

[6]Proofs are left as an exercise for the reader, whenever we do not give them.

Proposition 6. *For a query system $\mathcal{Q} = (\Theta_{¿}, Q, \gtrdot)$ with a monotonic proof system $\mathcal{P}_{\mathcal{Q}} = (\Theta, P, \vdash)$, $\Theta \supseteq \Theta_{¿}$, and a programming system $\mathfrak{P} = (\underline{\Pi}, I, O, \Xi, \rhd, \vdash)$ the following equivalence holds:*

$$\left(\overset{LP}{\underset{\vdash}{\Longleftrightarrow}}\right) \qquad ¿ \gtrdot F \vdash_{\blacksquare} \phi \Longleftrightarrow \iota \rhd \Pi \vdash_{\xi} o$$

This equivalence expresses the correspondence between the objects of logic and the objects of computation that account for the notion of deductive computation, and, in this particular case, deductive programming, or LP. This equivalence is particularly well established by the fact that I identify the symbol for syntactical *consequence* with the symbol for *computational yield*, an identity that I formalize more clearly now, namely from the semantical viewpoint.

As seen above, defined by means of a proof system the relation $\gtrdot$ merely determines the *formal acceptability* of a reply to a query. In order to determine the *semantical correctness* of a reply to a query we need to specify a semantics that is adequate for $\mathcal{Q}$.

Definition 7. A *semantical system* for a query system $\mathcal{Q} = (\Theta_{¿}, Q, \gtrdot)$ is a triple $\mathfrak{S} = (\Theta, M, \models)$ where $\Theta \supseteq \Theta_{¿}$ and M is a set of models such that $\phi \in \Theta_{¿}$ is a *semantically correct* reply to a query $¿$, denoted by $¿ \gtrdot K \models_{\mathcal{M}} \phi$ for $K \subseteq \Theta$ and a model $\mathcal{M} \in M$, according to the condition

$$¿ \gtrdot K \models_{\mathcal{M}} \phi \quad \text{iff} \quad ¿ \gtrdot \phi \text{ and } K \models_{\mathcal{M}} \phi.$$

If in the above definition we let K denote "explicit knowledge," we begin to see that a program of LP is in fact a KB, and the role of the *user* of the program (the KA) is merely that of asking questions concerning the KB.

Proposition 8. *For a query system $\mathcal{Q} = (\Theta_{¿}, Q, \gtrdot)$ with a semantical system $\mathfrak{S} = (\Theta, M, \models)$, $\Theta \supseteq \Theta_{¿}$, and a programming system $\mathfrak{P} = (\underline{\Pi}, I, O, \Xi, \rhd, \vdash)$ the following equivalence holds:*

$$\left(\overset{LP}{\underset{\models}{\Longleftrightarrow}}\right) \qquad ¿ \gtrdot F \models_{\mathcal{M}} \phi \Longleftrightarrow \iota \rhd \Pi \vdash_{\xi} o$$

The adequateness of a query system $\mathcal{Q} = (\Theta_{¿}, Q, \gtrdot)$ is then established if, if $¿ \gtrdot K \vdash_{\blacksquare} \phi$ then $¿ \gtrdot K \models_{\mathcal{M}} \phi$ (soundness), and if $¿ \gtrdot K \models_{\mathcal{M}} \phi$ then $¿ \gtrdot K \vdash_{\blacksquare} \phi$ (completeness). Note, however, that by *adequateness* of a query system $\mathcal{Q} = (\Theta_{¿}, Q, \gtrdot)$ it is now meant that a formula $\phi \in \Theta_{¿}$ is both formally acceptable and semantically correct as a reply to a query $¿ \in Q$. Generalizing the adequateness of $\mathcal{Q} = (\Theta_{¿}, Q, \gtrdot)$ to LP is then a straightforward matter.

However, we may start with a sound and complete proof system but end up with an incomplete programming system if we do not pay attention to the proof strategies to be of use in the latter. Typically, these proof strategies are actually restricted forms of proofs (e.g., cut elimination), and thus this is a problem that is rather easy to avoid. I concentrate on semantical aspects, as these are rather more complex and can lead to serious specification errors.

Definition 9. Let $\mathcal{Q} = (\Theta_{¿}, Q, \gtrdot)$ be a query system and $\mathfrak{S} = (\Theta, M, \models)$ be a semantical system with $\Theta_{¿} \subseteq \Theta$. We say that a formula ϕ is a *consequentially strongest correct reply* to the query $¿$ iff for $K \subseteq \Theta$

1. $¿ \gtrdot K \models \phi$;
2. for all $\psi \in \Theta_{¿}$, whenever $¿ \gtrdot K \models \psi$, then $\phi \models \psi$.

The above gives us the semantical equivalence of all consequentially strongest correct replies to a query $¿$. In LP we are particularly interested in questions of the form

$$¿x_1, ..., x_k$$

where $x_1, ..., x_k$ is a list of non-repeating variables.

Definition 10. Let now

$$Q = \{(¿x_1, ..., x_k : \phi) \mid \phi \in \Theta_{¿}\} .$$

We define the relation $\gtrdot \subseteq Q \times \Theta_{¿}$ by

$$(¿x_1, ..., x_k : \phi) \gtrdot \psi$$

iff

$$\psi = \phi \left[x_1/t_1, ..., x_k/t_k\right]$$

for some terms $t_1, ..., t_k \in K \subseteq \Theta_{¿}$.

In a query as specified above, we are interested in knowing for what terms $t_1, ..., t_k$ does $\phi \left[x_1/t_1, ..., x_k/t_k\right]$ hold.

Proposition 11. *The set*

$$\{(t_1, ..., t_k) \mid ¿ \gtrdot K \models \phi \left[x_1/t_1, ..., x_k/t_k\right]\}$$

is semi-computable.

Proposition 12. *Let ψ' be an instance of ψ. Then, $\psi \models \psi'$, and it follows immediately that if $¿ \gtrdot K \models \psi$, then $¿ \gtrdot K \models \psi'$. If ψ' is a variable renaming of ψ, then $\psi' \equiv \psi$.*

Proof. Trivial. □

Definition 13. Let $K \subseteq \Theta$ be a set of formulae and let $q = (¿x_1, .., x_k : \phi) \in Q$ be a query. We say that a reply ψ is a *most general reply* to q iff

1. $q \gtrdot K \models \psi$;
2. for all $\psi' \in \Theta_{¿}$, if ψ is an instance of ψ' and $q \gtrdot K \models \psi'$, then ψ' is a variable renaming of ψ.

Definition 14. A set $R \subseteq \Theta_{¿} \subseteq \Theta$ is a *most general set of correct replies* to q for K iff

1. each $\phi \in R$ is a most general reply to q for K;

2. for all $\psi \in \Theta$, if $q \gtrdot K \models \psi$, then ψ is an instance of some $\phi \in R$;

3. for all $\phi_1, \phi_2 \in R$, if ϕ_2 is an instance of ϕ_1, then $\phi_2 = \phi_1$.

Definition 15. Let now

$$Q_\wedge = \left\{ \left(\overset{¿}{\wedge} x_1, ..., x_k : \phi \right) \mid \phi \in \Theta_¿ \right\}$$

and define the relation $\gtrdot \subseteq Q_\wedge \times \Theta_¿$

$$\left(\overset{¿}{\wedge} x_1, ..., x_k : \phi \right) \gtrdot \psi$$

iff

$$\psi = \phi \left[x_1/t_1^1, ..., x_k/t_k^1 \right] \wedge ... \wedge \phi \left[x_1/t_1^n, ..., x_k/t_k^n \right]$$

for some terms $t_1^1, ..., t_k^n \in K \subseteq \Theta_¿$. Then, the triple $\mathcal{Q}_\wedge = \left(\Theta_\wedge^¿, Q_\wedge, \gtrdot \right)$ is a query system for *conjunctive replies* to queries of the form "for what terms $t_1, ..., t_k$ does $\phi \left[x_1/t_1, ..., x_k/t_k \right]$ hold?"

In $\mathcal{Q}_\wedge$, replies to $\overset{¿}{\wedge} x_1, ..., x_k : \phi$ are conjunctions of replies to $¿x_1, ..., x_k : \phi$. The former may have a consequentially strongest reply even if the latter does not. In fact, if the most general set of replies to $¿x_1, ..., x_k : \phi$ is finite, their conjunction is a consequentially strongest reply to $\overset{¿}{\wedge} x_1, ..., x_k : \phi$.

2.2 LP programs and their meaning

2.2.1 The language of LP

In the above discussion, I elaborated on the semantical systems that allow us to *specify* an LP language and on the proof systems that allow us to *implement* an LP language. Importantly, there is no question of precedence of one over the other, implementation or specification, in the design of an LP language.

I now move from the metalanguage of LP to the object-language level. LP as a *programming language* can be considered as a synonym for *pure Prolog*, a proper subset of *real Prolog*, which I approach in Section 3 below. Unless otherwise stated, the points that I next discuss, as well as the examples given, are so with pure Prolog in mind. However, the distinction between pure and real Prolog, though an important one, is of no import in this Section, and the examples below can be implemented in any Prolog environment. (I suggest the latest version of the LP implementation SWI-Prolog.)[7]

It will be easy to see how the above metalanguage definitions apply to the object-language constructs of LP. The definitions of expressions, substitution, and unification for CFOL hold generally, with the following specifications:[8]

[7]Freely available at www.swi-prolog.org.

[8]A complete specification is given for real Prolog in Section 3 below. For the standard CFOL definitions, see Augusto (2019).

Definition 16. An *LP formula* is a logical expression of the form

$$(\phi_{LP}) \qquad \forall x_1 ... \forall x_l \left(A_1, ..., A_m \leftarrow B_1, ..., B_n\right)$$

where $l, m, n \geq 0$, $A_1, ..., A_m, B_1, ..., B_n$ are atoms, $A_1, ..., A_m = A_1 \vee ... \vee A_m$, and $B_1, ..., B_n = B_1 \wedge ... \wedge B_n$. An LP formula is *closed* (*ground*) in the same circumstances as a CFOL formula is closed (ground, respectively).

More specifically, an LP formula is always a clause.[9]

Definition 17. An *LP clause* is an LP formula of the form

$$(\mathcal{C}_{LP}) \qquad \forall x_1 ... \forall x_l \left(A \leftarrow B_1, ..., B_n\right).$$

An LP clause is typically simplified as

$$(\mathcal{C}_{LP}) \qquad A \leftarrow B_1, ..., B_n.$$

In $\mathcal{C}_{LP}$, A is called the *head*, and $B_1, ..., B_n$ is the *body.* This is in fact a *Horn clause*, as inverting the symbol $\leftarrow$ we have the formula $B_1 \wedge ... \wedge B_n \rightarrow A$; applying now $p \rightarrow q := \neg p \vee q$ and the De Morgan law for $\wedge$, we have

$$B_1 \wedge ... \wedge B_n \rightarrow A \equiv \neg\left(B_1 \wedge ... \wedge B_n\right) \vee A \equiv \neg B_1 \vee ... \vee \neg B_n \vee A.$$

Obviously, $\mathcal{C}_{LP}$ is a *definite clause.*

Intuitively, $\mathcal{C}_{LP}$ can be interpreted as, for $0 \leq i \leq n$, if every B_i is true, then A is true; or, what is the same, A can be proved by proving all the B_i. In other words, we say that A is implied by $\bigwedge_{i=1}^{n} B_i$ (the *declarative* reading), or that in order to answer query A we have to answer the query $\bigwedge_{i=1}^{n} B_i$ (the *procedural* reading).[10]

Definition 18. The basic constructs of LP are *terms* and *statements.*

1. An LP *term* can be simple or compound. A *simple term* is a *variable* or a *constant.* A *compound term* comprises a *functor* and a sequence of one or more terms called *arguments.* A compound term of arity n has the form $p\left(t_1, ..., t_n\right)$, where p is the *name* of the functor and $t_1, ..., t_n$ are the arguments of p. A functor p with arity n is denoted by *p/n.* A functor can be a *relation symbol* or a *function symbol.* The name of a functor is an *atom.* A constant just is a functor of arity 0, so that it is also an atom.[11]

[9]Note the following basic definitions: We define a *literal*, denoted by L, to be an atom (e.g., P) or the negation of an atom (e.g., $\neg P$). We say that the literals P and $\neg P$ are *complementary.* A *clause* $\mathcal{C}$ is a finite disjunction of literals, i.e. $\mathcal{C} = L_1 \vee ... \vee L_n = \|L_1, ..., L_n\|$. $\mathcal{C}$ is a *Horn clause* if it contains at most one positive literal. A Horn clause with exactly one positive literal is a *definite clause.* $\mathcal{C}$ is a *dual-Horn clause* if it has at most one negative literal. The *empty clause* $\|\ \|$, denoted by $\square$, is a clause that contains no literals. A clause is called *ground* if no individual variables occur in it.

[10]See Apt (1996) for an elaboration on these two readings or interpretations.

[11]Note that variables, too, can be atoms in the language Prolog that I shall use for real Prolog (see below).

2. LP *statements* can be *facts*, *goals*, *rules*, and *queries*. Let $\mathcal{C}_{LP}$ be given; then, the unit clause A is a fact, the B_i are goals, and $\mathcal{C}_{LP}$, for $n \geq 0$, is a rule. Thus, a fact is the special case of a rule when $n = 0$. When $n = 1$, we have an *iterative clause.* The empty clause $\square$ is considered a goal. A *query* is a clause with a question mark.

It should be obvious that *relation symbol* just is another name for *predicate (symbol)*, and I shall favor the latter over the former for consistency reasons. Contrarily to the standard language of CFOL, the same functor name can be used for functions or predicates of different arity, a feature that is responsible for what is spoken of as *ambivalent syntax.* This is a useful feature when there is a natural relation between predicate and function symbols.

Example 19. `X`, `Y`, `John`, `john`, `sara`, `father`, `father(X,Y)`, and `male(john)` are terms of LP.[12]

- `X`, `Y` and `John` are individual variables, `john` and `sara` are atoms (constants, or names of individuals), and `father(X,Y)`, as well as `male(john)`, are functors whose arity is denoted by `father/2` and `male/1`, respectively. It is easy to see that individual variables are written with initial uppercase letters and atoms are written with initial lowercase letters; variables can also start with an underscore "_".[13]

- `father(X,Y)` is a non-ground predicate and `male(john)` is a ground predicate.

- `father(X,sara).` and `male(john).` (note the end marks) are facts built from the predicates `father(X,sara)` and `male(john)`.

- `father(john,sara)?` is a query asking whether the relation "X is the father of Y" holds between John and Sara, i.e. whether John is the father of Sara.

- `daughter(X,Y)` $\leftarrow$ `father(Y,X),female(X).` is a rule for the relation *daughter-of.* In this rule, `father(Y,X)` and `female(X)` are the goals, but the head, `daughter(X,Y)`, can also be a goal.

- `father(john,father(rita))` is a legal atom of LP.

[12]This example illustrates clearly that it is very useful to use `this font` when writing LP terms and statements: It helps to distinguish the natural language English from the formal language(s) of LP, a distinction that is crucial given the "denotational" character of the latter. I shall carry this practice over to both Prolog and Datalog. Precisely because of these, I also adopt the common practice – actually required by most software – of always ending a fact or a rule with an end mark. This said, I shall often relax these practices, writing simply, say, $p(X,Y) \leftarrow q(Y,X), r(X)$ instead of `p(X,Y)` $\leftarrow$ `q(Y,X),r(X).`, namely when LP languages are considered more immediately as logical languages. Further variations are $p(X,Y) \leftarrow q(Y,X) \wedge r(X)$ and `p(X,Y) :-q(Y,X),r(X).`.

[13]Variables with "_" are called *anonymous variables* and each such occurrence in a clause or query denotes a different variable.

2.2.2 Logic programs: Goals and meanings

Definition 20. A *logic program* in its simplest form is a finite set of facts. More typically, a logic program is a finite set of rules formulated as definite clauses, reason why we call this a *definite program.*

I shall often write *Prolog program* as a synonym for *logic*, or *LP*, *program.* More properly, though, an LP program is a finite *sequence* of rules. In effect, conjunction, as well as disjunction, is not a commutative operation, with $\phi_1 \overset{\wedge}{\vee} \phi_2 \neq \phi_2 \overset{\wedge}{\vee} \phi_1$ in terms of processing for ϕ a rule or a literal. These aspects, and their import to deductive programming, will become clearer below.

Example 21. The following facts constitute the program *Fatherhood*:

```
father(john,sara).       male(john).
father(john,peter).      male(rick).
father(john,rick).       male(peter).
father(rick,carl).       male(carl).
father(harry,louis).     male(harry).
father(harry,mary).      male(louis).
father(harry,jane).      female(sara).
                         female(mary).
                         female(jane).
```

Example 22. The following program, called *Addition*, has only two rules:

```
plus(0,X,X).
plus(s(X),Y,s(Z)) ← plus(X,Y,Z).
```

The basic operation on an LP program is *unification.* It is essentially the same as for CFOL and I provide here only a few specifications.

Definition 23. An LP *substitution* is a set of pairs of the form $X_i = t_i$, $0 \leq i \leq n$, where X_i is a variable and t_i is a term, $X_i \neq X_j$ for every $i \neq j$, and X_i does not occur in t_j for any i and j. Let σ be a substitution and A a term; then the result of applying substitution σ to term A, denoted by $A\sigma$, is the term obtained by substituting t for every occurrence of X in A for every pair $(X = t) \in \sigma$.

Definition 24. We say that B is an *instance* of A if there is a substitution σ such that $A\sigma = B$. C is a *common instance* of A and B if it is an instance of A and an instance of B, i.e. if there are substitutions σ_1, σ_2 such that $A\sigma_1 \equiv B\sigma_2$.

Example 25. Consider the program *Fatherhood.* Let there be given the substitution $\sigma = \{\texttt{X} = \texttt{john}, \texttt{Y} = \texttt{sara}\}$. The result of applying σ to the term `father(X,Y)`, denoted by $(\texttt{father}(\texttt{X},\texttt{Y}))\,\sigma$, is the term `father(john,sara)`. The goal `father(john,sara)` is an instance of the goal `father(X,Y)` (under substitution σ). Equally,

$$(\texttt{daughter}(\texttt{Y},\texttt{X}) \leftarrow \texttt{father}(\texttt{X},\texttt{Y}),\texttt{female}(\texttt{Y}).)\,\sigma$$

gives us

$$\texttt{daughter}\,(\texttt{sara}, \texttt{john}) \leftarrow \texttt{father}\,(\texttt{john}, \texttt{sara})\,, \texttt{female}\,(\texttt{sara})\,.$$

This unification for the goal $? - \texttt{daughter}\,(\texttt{sara}, \texttt{john})\,.$ can be represented in a tree as shown in Figure 2.

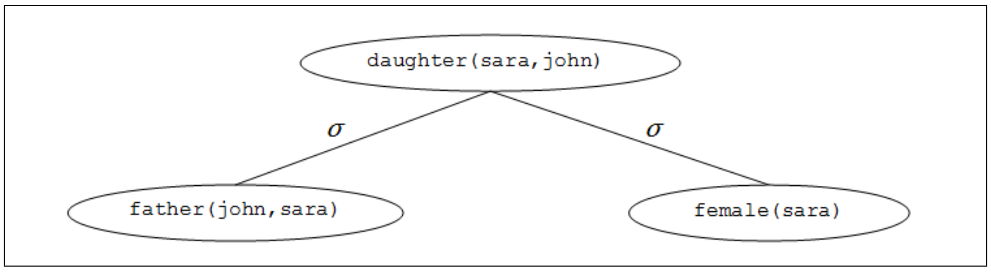


Figure 2: Unification via a substitution σ.

Example 26. Consider now the program *Addition*. Let there be given the substitutions $\sigma_1 = \{\texttt{X} = \texttt{1}\}$ and $\sigma_2 = \{\texttt{Y} = \texttt{1}\}$. Then the goals $\texttt{plus}\,(\texttt{0}, \texttt{1}, \texttt{X})$ and $\texttt{plus}\,(\texttt{0}, \texttt{Y}, \texttt{Y})$ have the common instance $\texttt{plus}\,(\texttt{0}, \texttt{1}, \texttt{1})$ by applying to them the substitutions σ_1 and σ_2, respectively.

Common instances are important because in fact we can reply to a query by finding a common instance of both a query and a fact.

Variables in facts are implicitly universally quantified, whereas variables in queries are implicitly existentially quantified.

Definition 27. A fact $p\,(t_1, ..., t_n)$ reads "for all $X_1, ..., X_n$, where the X_i are variables in the fact, $p\,(t_1, ..., t_n)$ holds or is true," i.e.

$$\forall X_1, ..., \forall X_n\,(p\,(t_1, ..., t_n))\,.$$

This definition holds for rules, too: The variables occurring in the head are universally quantified and their scope is the whole rule. However, variables occurring in the body of a rule but not in its head are considered to be existentially quantified.

Example 28. $\texttt{grandfather}\,(\texttt{X}, \texttt{Y}) \leftarrow \texttt{father}\,(\texttt{X}, \texttt{Z})\,, \texttt{father}\,(\texttt{Z}, \texttt{Y})\,.$ is read "for all X and Y, X is the grandfather of Y if there exists a Z such that X is the father of Z and Z is the father of Y."

Definition 29. A query $p\,(t_1, ..., t_n)$? reads "are there variables $X_1, ..., X_n$ such that $p\,(t_1, ..., t_n)$ holds or is true?", i.e.

$$\exists X_1, ..., \exists X_n\,(p\,(t_1, ..., t_n))?$$

For convenience, the universal quantifiers are omitted in facts and the existential quantifiers are so in queries; both quantifiers are omitted in rules.

Definition 30. For any substitution σ,

1. from a universal fact P deduce an instance $P\sigma$ of it. We call this *instantiation* and denote it by $\vdash_{inst}$.
2. an existential query P is a logical consequence of an instance $P\sigma$ of it. We call this *generalization* and denote it by $\vdash_{gen}$.

By combining 1 and 2 above, we have a reply to a query by means of a common instance, i.e. we have

$$P. \vdash_{inst} P\sigma \vdash_{gen} (P?)$$

Definition 31. A *solution* to a query is a fact that is a (common) instance of the query.

Example 32. The goal $\mathtt{father(john, sara)}$ implies that there exists an X such that $\mathtt{father(X, sara)}$ is *true*, in this case $\mathtt{X} = \mathtt{john}$. Then, $\mathtt{father(john, sara)}$ is a solution to the query $\mathtt{father(X, sara)}$?, and the solution is represented by the substitution $\mathtt{X} = \mathtt{john}$.

An existential query may have several solutions.

Example 33. Consider the program *Addition.* $\mathtt{plus\,(X, Y, 4)}$?, the query asking for numbers X and Y that add up to four, has as possible solutions $\{\mathtt{X} = \mathtt{0}, \mathtt{Y} = \mathtt{4}\}$ and $\{\mathtt{X} = \mathtt{3}, \mathtt{Y} = \mathtt{1}\}$.[14]

Definition 34. A query that is a goal is actually a special case of a *conjunctive query*, i.e. a conjunction of goals of the form

$$(q_1, ..., q_n?) \equiv (q_1 \wedge ... \wedge q_n?)$$

We denote a conjunctive query by $Q_\wedge$.

Definition 29 above determines that the scope of the existential quantifier in a conjunctive query is the whole conjunction, so that a query $p\,(X)\,, q\,(X)$? actually asks whether there is an X such that both $p\,(X)$ and $q\,(X)$ hold.

Definition 35. In a conjunctive query of the form $p\,(X)\,, q\,(X)$? we say that X is a *shared variable.*

Definition 36. A conjunctive query $Q_\wedge = q_1, ..., q_n$? is a logical consequence of a program $\mathit{\Pi}$, denoted by

$$\mathit{\Pi} \vdash q_1, ..., q_n?$$

if for every goal $q_i \in Q_\wedge$, $0 < i \leq n$, we have

$$\mathit{\Pi} \vdash q_i$$

where shared variables in the q_i are instantiated to the same values.

[14]Note that in fact we have the query $\mathtt{plus\,(X, Y, s\,(s\,(s\,(s\,(0)))))}$? and the possible solutions $\{\mathtt{X} = \mathtt{0}, \mathtt{Y} = \mathtt{s\,(s\,(s\,(s\,(0))))}\}$ and $\{\mathtt{X} = \mathtt{s\,(s\,(s\,(0)))}, \mathtt{Y} = \mathtt{s\,(0)}\}$, as the natural numbers are defined by means of the function $s\,(0)$ such that $s\,(0) = 1$, $s\,(s\,(0)) = 2$, etc.

In order to allow a logic program to compute deductively we need to define one further rule:

Definition 37. *Universal modus ponens (UMP)* – From the rule $r = A \leftarrow B_1, ..., B_n$ and the facts $B_1^{'}, ..., B_n^{'}$ we can deduce A' if $A^{'} \leftarrow B_1^{'}, ..., B_n^{'}$ is an instance of r.

For convenience, let us denote a quantified goal by G.

Definition 38. An existentially quantified goal G is a logical consequence of a program Π, denoted by $\Pi \vdash G$, if there is a clause in Π with a ground instance $A \leftarrow B_1, ..., B_n$, $n \geq 0$, such that $\Pi \vdash B_i$ for all $0 < i \leq n$ and A is an instance of G. In other words, G is a logical consequence of Π iff G can be deduced from Π by a finite number of applications of UMP.

Definition 39. An *abstract interpreter* Ψ for a logic program is an algorithm that takes as input a program Π and a goal G, answering *true* (or *yes*) if G is a logical consequence of Π and *false* (or *no*) otherwise. In the first case we say that Ψ performs a *true-computation*, and in the second case we say that Ψ performs a *false-computation.*

Definition 40. The *meaning of a program Π*, denoted by $M(\Pi)$, is the set of unit ground goals $G = \{B_1, ..., B_n\}$ such that for all $0 < i \leq n$ we have

$$\Pi \vdash B_i.$$

Let now the *intended meaning of a program Π* be denoted by $IM(\Pi)$. A program Π is said to be *correct* with respect to some $IM(\Pi)$ iff $M(\Pi) \subseteq IM(\Pi)$, and it is said to be *complete* with respect to some $IM(\Pi)$ iff $IM(\Pi) \subseteq M(\Pi)$. A program Π is *adequate*, i.e. both correct and complete, with respect to some intended meaning M iff $IM(\Pi) = M(\Pi)$.

Informally, a program Π is correct iff it does not "say" unintended "things," and it is complete if every "thing" that is intended can be "said." The meaning of a basic program built up solely of ground facts is the program itself. Put differently, the program "means" just what it "says." The meaning of a regular logic program (i.e. a logic program comprising rules) contains explicitly whatever the program states implicitly.

Example 41. The meaning of the program *Fatherhood* (Example 21), $M(Fatherhood)$, just is the program itself. If we add to this program the rule

$$\texttt{parent(X,Y)} \leftarrow \texttt{father(X,Y)}.$$

then $M(Fatherhood)$ additionally contains all goals of the form `parent(X,Y)` for every pair (X, Y) such that `father(X,Y)`. is in the program.

This said, it should be obvious that the intended meaning of a program Π – a set $IM(\Pi)$ of unit ground goals – is intuitively given by the choice of names in the program. This allows a semantics of quasi-truth values in the following way:

Definition 42. Given a program Π, we say that a ground goal G is *true* with respect to $IM(\Pi)$ if $G \in IM(\Pi)$; otherwise, we say that G is *false.*

Algorithm 1 Ground reduction

Input: (Π, G)
Output: True or False

```
Initialize the resolvent to $G$

while resolvent $\neq \Box$ do
    choose a goal $A$ from the resolvent
    choose a ground instance of a clause $A' \leftarrow B_1, ..., B_n \in \Pi$ s.t. $A = A'$
      if no such goal and clause exist, leave the while loop
    replace $A$ by $B_1, ..., B_n$ in the resolvent
if resolvent $= \Box$, then output true, else output false
```

2.3 Resolution and LP computations

2.3.1 Reductions and reduction proof trees

I now expand on the abstract interpreter Ψ of Definition 39, namely as a search algorithm of LP when computing a goal.

Definition 43. We call *resolvent* the current (usually conjunctive) stage of an LP computation. The *empty resolvent* (or *empty clause*), denoted by $\Box$, is the clause with empty head and empty body. The sequence of resolvents produced during a computation is called the *trace* of the interpreter.

Definition 44. Given an LP program Π and a goal G, the replacement of G by the body of an instance of a clause $\mathcal{C} \in \Pi$ whose head is identical to G is called a *reduction*. A reduction is *ground* if both the goal G and the instance of the clause $\mathcal{C}$ are ground. The goal replaced in a reduction is said to be *reduced* and we say that the new goals are *derived*.

The algorithm for this procedure is given as Algorithm 1. If goal G is not deducible from program Π, then Ψ may fail to terminate. Note that each iteration of the "while loop" is a single application of UMP, i.e. a reduction. It is easy to see that reduction is the basic computational step in LP. The selection of the goal to be reduced and the order of the reductions thereof is arbitrary, as all the goals in a given resolvent must be reduced. The selection of a clause and a suitable instance thereof is non-deterministic but critical.

Recall from the above discussion that given a query system $\mathcal{Q} = (\Theta_{¿}, Q, \gtrdot)$ and a proof system for it $\mathcal{P}_{\mathcal{Q}} = (\Theta \supseteq \Theta_{¿}, P, \gtrdot)$, for some query ¿ and a set of formulae $F \subseteq 2^{\Theta}$ we have a provably correct reply $\phi \in \Theta_{¿}$ to ¿, i.e. $¿ \gtrdot F \vdash_{\blacksquare} \phi$, iff we have

$$¿ \gtrdot \phi \text{ and } F \vdash_{\blacksquare} \phi.$$

This, by Proposition 6, is equivalent to $\iota \rhd \Pi \vdash_{\xi} o$ in terms of LP. This entails that given input ι, a computation ξ producing output o from a program Π actually is a proof that the query follows from the program. Such a proof is implicitly represented in the trace of a query, but we can represent it explicitly in the form of a tree.

Definition 45. A *(reduction) proof tree* is a (directed) tree whose nodes represent the goals that are reduced during a computation, there being a (directed) edge from a node to each node that corresponds to a derived goal of a reduced goal. In a proof tree, the number of nodes corresponds to the number of reduction steps in a computation. The root of a proof tree for a simple query is the query itself. The proof tree for a conjunctive query is the collection of all the proof trees for its individual goals.[15]

Example 46. Figure 2 above shows in fact the (reduction) proof tree for the goal $? -$ `daughter`$($`sara`$,$ `john`$)$. given the program *Fatherhood* augmented as in Example 41.

2.3.2 LI resolution and SLD resolution

However, knowledge of the resolution calculus and a little thought will reveal that reduction, a generalization of UMP, can be re-expressed in terms of this calculus.[16] Indeed, it is easy to see that we can apply resolution to find a contradiction from the combination of a goal clause, with solely negative literals, and a fact (in a rule), a positive literal. It is important to remark that a goal clause $\mathcal{G}$ is not a *program clause*, which can be only either a rule or a fact. This should be born in mind when considering $\Pi \cup \{\mathcal{G}\}$, i.e. when we add a goal clause $\mathcal{G}$ to an LP program Π. The goal clause $\mathcal{G} = \|\neg q_1, ..., \neg q_n\|$ is added to the program, in order to test if $\mathcal{Q}_\wedge = q_1 \wedge ... \wedge q_n$ follows from it, and this is the case iff $\Pi \cup \{\mathcal{G}\}$ is unsatisfiable. This is so iff we can deduce the *empty goal clause* $\square$ from $\Pi \cup \{\mathcal{G}\}$ by an application of resolution. More specifically, we refer here to *linear input resolution* (abbr.: LI resolution), as this has been proven complete for Horn clauses. LI resolution, in turn, is a refinement of linear resolution.

Definition 47. Given a set of clauses C, we say that a clause $\mathcal{C}$ is a *linear resolution* deduction from C, and write $C \vdash_{lres} \mathcal{C}$, if there is a sequence of pairs $(\mathcal{C}_0, \mathcal{D}_0), ..., (\mathcal{C}_n, \mathcal{D}_n)$ such that $\mathcal{C} = \mathcal{C}_{n+1}$ and (i) $\mathcal{C}_0$, called the *starting clause*, and the $\mathcal{D}_i$ are elements of C or some $\mathcal{C}_j, j < i$, (ii) each $\mathcal{C}_{i+1}, i \leq n$, is a resolvent of $\mathcal{C}_i$ and $\mathcal{D}_i$. The elements of C are called the *input clauses*, the $\mathcal{C}_i$ are the *center clauses* and the $\mathcal{D}_i$ the *side clauses*. If $\mathcal{C} = \square$, we say that there is a *linear-resolution refutation* of C, and write $C \vdash_{lres} \square$.

Example 48. Figure 3 shows the linear resolution refutation of the obviously unsatisfiable set of clauses $C = \{\|p, q\|, \|p, \neg q\|, \|\neg p, q\|, \|\neg p, \neg q\|\}$.

[15] Basically, a (reduction) proof tree shows the instantiation of goals up to the queried goal. See Fig. 2.

[16] This is a complex calculus and here I give only some of its core aspects. Just as a reminder, recall that the first-order resolution principle states that given two clauses $\mathcal{C}_1 = \mathcal{C}_1' \vee L$ and $\mathcal{C}_2 = \mathcal{C}_2' \vee \neg L$ of first-order predicate logic we have the inference rule

$$\frac{\mathcal{C}_1' \vee L \quad \mathcal{C}_2' \vee \neg L}{(\mathcal{C}_1' \vee \mathcal{C}_2')\,\sigma}$$

if there is a substitution σ such that σ unifies the pair of complementary literals L and $\neg L$. A *resolution deduction* of $\mathcal{C}$ from a set of clauses C, denoted by $C \vdash_{res} \mathcal{C}$, is a finite sequence $\mathcal{C}_1, \mathcal{C}_2, ..., \mathcal{C}_k$ of clauses such that each $\mathcal{C}_i$ is either a clause in C or a resolvent of clauses preceding $\mathcal{C}_i$, and $\mathcal{C}_k = \mathcal{C}$. We call the deduction of the empty set $\square$ from C a *refutation*, or *proof* of C. (The reader is referred to Augusto [2019; 2022] for an extensive discussion of the resolution calculus and to Leitsch [1997] for a monograph thereon.)

$$\begin{tikzcd}
\|p,q\| \arrow[d, dash] & \|p,\neg q\| \arrow[dl, dash] \\
\|p\| \arrow[d, dash] & \|\neg p,q\| \arrow[dl, dash] \\
\|q\| \arrow[d, dash] & \|\neg p,\neg q\| \arrow[dl, dash] \\
\|\neg p\| \arrow[d, dash] & \|p\| \arrow[dl, dash] \\
\square &
\end{tikzcd}$$

Figure 3: A linear-resolution refutation.

Definition 49. Linear resolution is called *linear input resolution* (abbr.: *LI resolution*) if, given a sequence of pairs $(\mathcal{C}_0, \mathcal{D}_0), ..., (\mathcal{C}_n, \mathcal{D}_n)$, then all the $\mathcal{D}_i$ are variants of clauses in C.

Example 50. Consider the program *Addition*. Figure 4 shows an LI-resolution proof that $0+2=2$, i.e. `plus (0, s (s (0)) , s (s (0)))`. Notice that each resolvent is a reduced goal. Figure 5 is the corresponding resolution-proof tree, in which for convenience I label the tree only with the numbers in the proof of Figure 4.

1.	$\leftarrow$ `plus (X, Y, s (s (0)))` .	Goal
2.	`plus (s (X1) , Y1, s (Z1))` $\leftarrow$ `plus (X1, Y1, Z1)` .	Variable renaming
3.	$\leftarrow$ `plus (X1, Y1, s (0))` .	Resolution (1, 2)
		$\sigma = \{\mathtt{X} = \mathtt{s\,(X1)}, \mathtt{Y} = \mathtt{Y1}, \mathtt{Z1} = \mathtt{s\,(0)}\}$
4.	`plus (s (X2) , Y2, s (Z2))` $\leftarrow$ `plus (X2, Y2, Z2)` .	Variable renaming (2)
5.	$\leftarrow$ `plus (X2, Y2, 0)` .	Resolution (3, 4)
		$\tau = \{\mathtt{X1} = \mathtt{s\,(X2)}, \mathtt{Y1} = \mathtt{Y2}, \mathtt{Z2} = \mathtt{0}\}$
6.	`plus (0, X3, X3)` .	Variable renaming
7.	$\square$	Resolution (5, 6)
		$\lambda = \{\mathtt{X2} = \mathtt{0}, \mathtt{X3} = \mathtt{0}, \mathtt{Y2} = \mathtt{0}\}$

Figure 4: An LI-resolution proof on an LP program.

Lemma 51. *Let Π be an LP program and $\mathcal{G} = \|\neg q_1, ..., \neg q_n\|$ a goal clause. Then, all the q_i are consequences of Π iff $\Pi \cup \{\mathcal{G}\}$ is unsatisfiable.*

Theorem 52. *(Refutation completeness of linear resolution for Horn clauses) If C is an unsatisfiable set of Horn clauses, then there is a linear resolution proof that is a refutation of C, i.e. $C \vdash_{lres} \square$.*

Proof. (Idea) Assume that C is finite and proceed by induction on the elements of C. $\square$

$$\begin{tikzcd}
1. \arrow[d, dash] & 2. \arrow[dl, dash] \\
3. \arrow[d, dash] & 4. \arrow[dl, dash] \\
5. \arrow[d, dash] & 6. \arrow[dl, dash] \\
\square &
\end{tikzcd}$$

Figure 5: An LI-resolution proof tree.

Although LI resolution is the general resolution rule for LP, when given as input the query $q_1, ..., q_n$? an LP interpreter actually searches for an *SLD-resolution proof of* $\square$, in turn a case of LD resolution.

Definition 53. Let $\mathcal{C} = \|\neg L_1, ..., \neg L_n\|$, $\mathcal{D} = \|M, \neg M_1, ..., \neg M_m\|$ be ordered clauses. The following rule of inference, where we have $\sigma = mgu\,(M, L_i)$ and the resolvent is an ordered (or definite) clause, is called *linear definite resolution* (abbr.: *LD resolution*):

$$\frac{\neg L_1 \vee ... \vee \neg L_n \qquad M \vee \neg M_1 \vee ... \vee \neg M_m}{(\neg L_1 \vee ... \vee \neg L_{i-1} \vee \neg M_1 \vee ... \vee \neg M_m \vee \neg L_{i+1} \vee ... \vee \neg L_n)\,\sigma}$$

The following inference rule, where $\sigma = mgu\,(M, L_i)$ and $\mathbf{r}$ is a selection rule or function, is called *selective linear definite resolution* (abbr.: *SLD resolution*):

$$\frac{\mathbf{r}\,(\neg L_1 \vee ... \vee \neg L_n) \qquad M \vee \neg M_1 \vee ... \vee \neg M_m}{(\neg L_1 \vee ... \vee \neg L_{i-1} \vee \neg M_1 \vee ... \vee \neg M_m \vee \neg L_{i+1} \vee ... \vee \neg L_n)\,\sigma}$$

I now define these two resolution refinements with respect to LP.

Definition 54. Let $\Pi \cup \{\mathcal{G}\}$ be given as a set of ordered clauses. Then, an LD-resolution refutation of $\Pi \cup \{\mathcal{G}\}$, denoted by $\Pi \cup \{\mathcal{G}\} \vdash_{ldres} \square$, is a sequence

$$(\mathcal{G}_0, \mathcal{C}_0), ..., (\mathcal{G}_n, \mathcal{C}_n)$$

where the $\mathcal{G}_i, \mathcal{C}_i$, $0 \leq i \leq n$, are ordered clauses, such that $\mathcal{G}_0 = \mathcal{G} = \|\neg p_1, ..., \neg p_n\|$ and $\mathcal{G}_{n+1} = \square$. More specifically, we have

$$\mathcal{G}_i = \left\|\neg p_{i,1}, ..., \neg p_{i,n(i)}\right\|, |\mathcal{G}_i| = n\,(i)$$

are the goal clauses, and for the $\mathcal{C}_i \in \Pi$ we have

$$\mathcal{C}_i = \left\|q, \neg q_{i,1}, ..., \neg q_{i,m(i)}\right\|, |\mathcal{C}_i| = m\,(i) + 1 \text{ or } 1 \text{ if } \mathcal{C}_i = \|q\|.$$

Then, for each $i < n$ there is a resolution rule

$$\frac{\mathcal{G}_i \qquad \mathcal{C}_i}{\mathcal{G}_{i+1}}$$

where $\mathcal{G}_{i+1} = \left\|\neg p_{i,1}, ..., \neg p_{i,k-1}, \neg q_{i,1}, ..., \neg q_{i,m(i)}, \neg p_{i,k+1}, ..., \neg p_{i,n(i)}\right\|$, an ordered clause with $|\mathcal{G}_{i+1}| = (n\,(i) - 1) + m\,(i)$, is the resolvent.

Lemma 55. *For an LP program $\mathit{\Pi}$ and a goal clause $\mathcal{G} = \|\neg q_1, ..., \neg q_n\|$, if $\mathit{\Pi} \cup \{\mathcal{G}\}$ is an unsatisfiable set of ordered clauses, then there is an LD-resolution refutation of $\mathit{\Pi} \cup \{\mathcal{G}\}$ beginning with $\mathcal{G}$.*

We obtain SLD resolution by introducing a *selection rule* r to choose the literal $p_i \in \mathcal{G}_i$ to be resolved upon with LD resolution, so that $r(\mathcal{G}_i)$ is the literal resolved upon in the $(i+1)$-th step of the proof.

Definition 56. The LD-resolution rule

$$\frac{r(\mathcal{G}_i) \qquad \mathcal{C}_i}{\mathcal{G}_{i+1}}$$

where r is a selection rule, is called an *SLD-resolution rule.*

In LP, the selection rule simply chooses the leftmost literal in the goal clause to be resolved upon.

Theorem 57. *(Completeness of SLD resolution for LP) Given an LP program $\mathit{\Pi}$ and a goal clause $\mathcal{G} = \|\neg q_1, ..., \neg q_n\|$, if $\mathit{\Pi} \cup \{\mathcal{G}\}$ is an unsatisfiable set of ordered clauses, then there is a selection rule r such that there is an SLD-resolution refutation of $\mathit{\Pi} \cup \{\mathcal{G}\}$ via r beginning with $r(\mathcal{G})$.*

Proof. (Sketch). Lemma 55 assures us that there is an LD-resolution refutation of $(\mathit{\Pi} \cup \{\mathcal{G}\}) \notin SAT$ beginning with $\mathcal{G}$. We only need to prove that there is an SLD-resolution refutation of $(\mathit{\Pi} \cup \{\mathcal{G}\}) \notin SAT$ via $r(\mathcal{G})$. The proof is by induction on the length of $\mathcal{G}$. If $|\mathcal{G}| = 1$, then $\mathcal{G}_0$ is a unit clause and $r(\mathcal{G}_0)$ is irrelevant. We now let $(\mathcal{G}_0, \mathcal{C}_0), ..., (\mathcal{G}_n, \mathcal{C}_n)$ be an LD-resolution refutation of $(\mathit{\Pi} \cup \{\mathcal{G}_0\}) \notin SAT$ and we suppose that the selection rule r chooses the literal $\neg p_{0,k} \in \mathcal{G}_0$. Because $(\mathit{\Pi} \cup \{\mathcal{G}_0\}) \notin SAT$, we must have $\mathcal{G}_{n+1} = \square$, and hence there must be some $j < n$ at which we resolve on $\neg p_{0,k}$. If $j = 0$, we are done. If $j \geq 1$, then there must be some $\mathcal{C}$ that is a resolvent of $\mathcal{G}_0$ and $\mathcal{C}_j$. Then, there must be an LD-resolution refutation of length $n-1$ of $\mathit{\Pi} \cup \{\mathcal{C}\}$ beginning with $\mathcal{C}$. By induction, this refutation can be replaced by an SLD-resolution refutation via r. We add this refutation onto the single step resolution of $\mathcal{G}_0$ and $\mathcal{C}_j$ obtaining the SLD-resolution of $(\mathit{\Pi} \cup \{\mathcal{G}\}) \notin SAT$ via $r(\mathcal{G})$ beginning with $\mathcal{G} = \mathcal{G}_0$. $\square$

I now elaborate on how SLD resolution corresponds to the search process in LP when a query is entered as input. An LP-proof tree corresponds to the search process known as *depth-first search with backtracking*: By "depth-first search" it is meant that, given a finitely branching tree, all the descendants of a node are checked before their siblings on the right of the tree and no edge is traversed more than once; if a *fail* leaf is encountered, then the search "backtracks" to the immediate ancestor of this leaf and the depth-search process is resumed. If a success leaf (denoted by $\square$) is found, the search stops until we prompt the search to proceed further by means of an *expand* "command" that makes the search retake.[17] The search is considered successful if at least one $\square$-resolvent is found on the tree; otherwise, the search fails and "false" is the output to the query.

[17]In the SWI-Prolog interpreter, we simply enter ";".

Put briefly, given an LP program Π and a goal clause $\mathcal{G}$, every branch of a complete LP-proof tree is either a successful SLD-resolution proof or a failed SLD-resolution proof. For this reason, we refer to this tree as an *SLD-resolution tree.*

Example 58. Let there be given the following LP program Π_1:

1. $\mathtt{p(X,Y)} \leftarrow \mathtt{q(X,Z), r(Z,Y)}$.
2. $\mathtt{p(X,X)} \leftarrow \mathtt{s(X)}$.
3. $\mathtt{q(X,b)}$.
4. $\mathtt{q(b,a)}$.
5. $\mathtt{q(X,a)}$.
6. $\mathtt{r(b,a)}$.
7. $\mathtt{s(X)} \leftarrow \mathtt{t(X,a)}$.
8. $\mathtt{s(X)} \leftarrow \mathtt{t(X,b)}$.
9. $\mathtt{s(X)} \leftarrow \mathtt{t(X,X)}$.
10. $\mathtt{t(a,b)}$.
11. $\mathtt{t(b,a)}$.

Our query is $? - \mathtt{p(X,X)}$, i.e. $\|\neg\mathtt{p(X,X)}\|$, where I use the symbol $\neg$ for convenience (see next Section). The depth-first algorithm starts by checking premise 1 and then moves to premise 2. Beginning with premise 1, we have it that there is a successful SLD-resolution proof when we apply SLD resolution to the premises 1, 3, and 6, in this exact order, with, after renaming of variables, substitutions $\sigma = \{\mathtt{X_1} \to \mathtt{X}, \mathtt{Y} \to \mathtt{X}\}$, $\theta = \{\mathtt{X_2} \to \mathtt{X}, \mathtt{Z} \to \mathtt{b}\}$, and $\lambda = \{\mathtt{X} \to \mathtt{a}\}$ (cf. Figure 6). Applying SLD resolution to the sequences 1 and 4 or 1 and 5 will not produce successful proofs. The search starting by checking premise 2 produces two successful proofs and a failure. Figure 7 shows the complete SLD-proof tree for $\Pi_1 \cup \{\|\neg\mathtt{p(X,X)}\|\}$ with the further substitutions $\omega = \{\mathtt{X} \to \mathtt{b}, \mathtt{Z} \to \mathtt{a}\}$, $\varsigma = \{\mathtt{Z} \to \mathtt{a}\}$, and $\mu = \{\mathtt{X} \to \mathtt{b}\}$; renaming of variables was omitted and ε denotes the empty substitution.

$$\begin{array}{rcl}
\|\neg\mathtt{p(X,X)}\| & & 1.\ \|\mathtt{p(X_1,Y)}, \neg\mathtt{q(X_1,Z)}, \neg\mathtt{r(Z,Y)}\| \\
| & /\sigma & \\
\|\neg\mathtt{q(X,Z)}, \neg\mathtt{r(Z,X)}\| & & 3.\ \|\mathtt{q(X_2,b)}\| \\
| & /\theta & \\
\|\neg\mathtt{r(b,X)}\| & & 6.\ \|\mathtt{r(b,a)}\| \\
| & /\lambda & \\
\square & &
\end{array}$$

Figure 6: An SLD-resolution proof.

Example 59. Figure 8 shows how SWI-Prolog answers the query $? - \mathtt{p(X,X)}$. when given program Π_1 and how it answers the request to produce a trace of some instantiations. In the first case, given the input (query) $? - \mathtt{p(X,X)}$., SWI-Prolog gives the first answer, to wit, $\mathtt{X} = \mathtt{a}$. Asked to provide more answers by means of the prompt ";", SWI-Prolog gives the replies $\mathtt{X} = \mathtt{b}$ and $\mathtt{X} = \mathtt{a}$, finally replying that there are no more instantiations (denoted by `false`). Asked to output traces of $? - \mathtt{p(X,X)}$., $? - \mathtt{p(a,X)}$., and $? - \mathtt{p(b,X)}$., SWI-Prolog does so, in each case adding that $\mathtt{X} = \mathtt{a}$. Compare these with the SLD-proof tree in Figure 7. Figure 9 shows both the case of a successful instantiation $? - \mathtt{p(b,b)}$. and a failed instantiation $? - \mathtt{p(c,d)}$.. In these last traces, "redo" indicates backtracking.

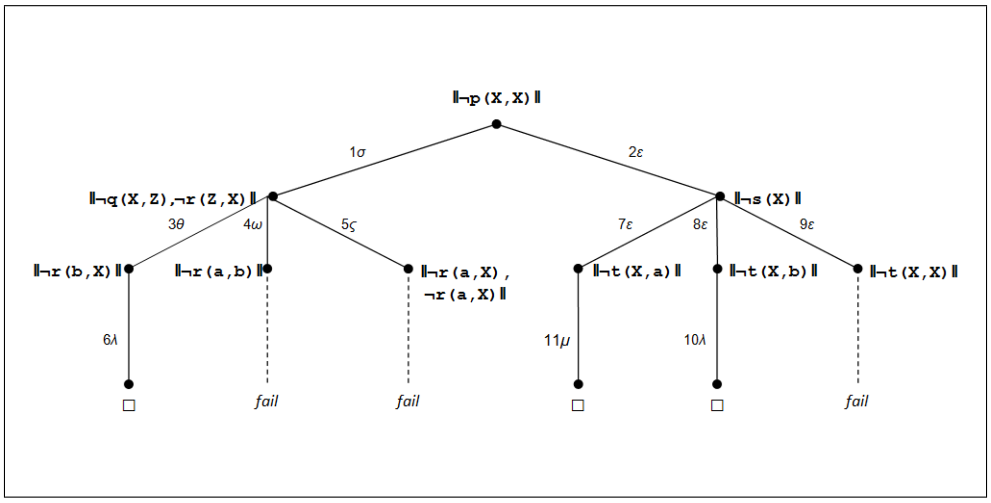


Figure 7: A complete SLD-proof tree for a Prolog program.

2.4 Negation in LP

2.4.1 Negation and completeness I: Negation by failure

I begin this Section by retaking the LP program Π_1 of Example 58. Consulting this program, SWI-Prolog will reply `true` to the queries $? - \mathtt{p}(\mathtt{a},\mathtt{a}).$, $? - \mathtt{p}(\mathtt{b},\mathtt{a}).$, and $? - \mathtt{p}(\mathtt{b},\mathtt{b}).$; but the queries $? - \mathtt{p}(\mathtt{a},\mathtt{b}).$ and $? - \mathtt{p}(\mathtt{c},\mathtt{c}).$, for example, are given the reply `false`. In CFOL, this would mean that, given some interpretation $\mathcal{I}$, there is a valuation $val_{\mathcal{I}}$ such that $val_{\mathcal{I}}(p(a,a)) = \mathtt{t}$ iff $val_{\mathcal{I}}(\neg p(a,a)) = \mathtt{f}$, and $val_{\mathcal{I}}(p(c,c)) = \mathtt{f}$ iff $val_{\mathcal{I}}(\neg p(c,c)) = \mathtt{t}$. In this resides the bivalent character of CFOL. However, in LP this simply means that

$$\Pi_1, ? - \mathtt{p}(\mathtt{a},\mathtt{a}). \vdash_{sldres} \square \quad \Rightarrow \quad \mathtt{p}(\mathtt{a},\mathtt{a}) = \mathtt{true}$$

indicating *success*, and

$$\Pi_1, ? - \mathtt{p}(\mathtt{c},\mathtt{c}). \nvdash_{sldres} \square \quad \Rightarrow \mathtt{p}(\mathtt{c},\mathtt{c}) = \mathtt{false}$$

indicating *failure*. In other words, given an LP program Π and a goal $\mathcal{G} = \|\neg q\|$, SWI-Prolog replies to a query with `true` if $(\Pi \cup \{\mathcal{G}\}) \notin SAT$ and with `false` if $(\Pi \cup \{\mathcal{G}\}) \in SAT$. This latter case just means that SWI-Prolog failed to prove goal $\mathcal{G}$ by means of SLD resolution; it means neither that $val(\mathcal{G}) = \mathtt{f}$ nor that $val(\neg\mathcal{G}) = \mathtt{t}$.

However, LP is deductive computation *also* because it is truth-preserving computation, and above I elaborated at length on the completeness of LP, so that one can say that there is a truth-based semantics for LP. Indeed, this is the case; what there is *not* is a semantics of falsity, namely with respect to the classical negation denoted by $\neg$. In LP, falsity with respect to a query q is the case when, as seen, q cannot be deduced from the program, or q does not match any of the data in the program. This is well expressed in terms of the meaning of a program Π, denoted by $M(\Pi)$, in Definition 42. But this does not mean that we have $\neg q$, instead. It just means that

```
?- p(X,X).
X = a ;
X = b ;
X = a ;
false.

?- trace.
true.

[trace] ?- p(X,X).
    Call:  (8) p(_2768, _2768) ?  creep
    Call:  (9) q(_2768, _2986) ?  creep
    Exit:  (9) q(_2768, b) ?  creep
    Call:  (9) r(b, _2768) ?  creep
    Exit:  (9) r(b, a) ?  creep
    Exit:  (8) p(a, a) ?  creep
X = a .

[trace] ?- p(a,X).
    Call:  (8) p(a, _2770) ?  creep
    Call:  (9) q(a, _2986) ?  creep
    Exit:  (9) q(a, b) ?  creep
    Call:  (9) r(b, _2770) ?  creep
    Exit:  (9) r(b, a) ?  creep
    Exit:  (8) p(a, a) ?  creep
X = a .

[trace] ?- p(b,X).
    Call:  (8) p(b, _2770) ?  creep
    Call:  (9) q(b, _2986) ?  creep
    Exit:  (9) q(b, b) ?  creep
    Call:  (9) r(b, _2770) ?  creep
    Exit:  (9) r(b, a) ?  creep
    Exit:  (8) p(b, a) ?  creep
X = a .
```

Figure 8: SWI-Prolog answering a query and outputting traces for some “true” instantiations.

```
[trace] ?- p(b,b).
    Call:  (8) p(b, b) ?  creep
    Call:  (9) q(b, _2950) ?  creep
    Exit:  (9) q(b, b) ?  creep
    Call:  (9) r(b, b) ?  creep
    Fail:  (9) r(b, b) ?  creep
    Redo:  (9) q(b, _2950) ?  creep
    Exit:  (9) q(b, a) ?  creep
    Call:  (9) r(a, b) ?  creep
    Fail:  (9) r(a, b) ?  creep
    Redo:  (9) q(b, _2950) ?  creep
    Exit:  (9) q(b, a) ?  creep
    Call:  (9) r(a, b) ?  creep
    Fail:  (9) r(a, b) ?  creep
    Redo:  (8) p(b, b) ?  creep
    Call:  (9) s(b) ?  creep
    Call:  (10) t(b, a) ?  creep
    Exit:  (10) t(b, a) ?  creep
    Exit:  (9) s(b) ?  creep
    Exit:  (8) p(b, b) ?  creep
true.

[trace] ?- p(c,d).
    Call:  (8) p(c, d) ?  creep
    Call:  (9) q(c, _2950) ?  creep
    Exit:  (9) q(c, b) ?  creep
    Call:  (9) r(b, d) ?  creep
    Fail:  (9) r(b, d) ?  creep
    Redo:  (9) q(c, _2950) ?  creep
    Exit:  (9) q(c, a) ?  creep
    Call:  (9) r(a, d) ?  creep
    Fail:  (9) r(a, d) ?  creep
    Redo:  (8) p(c, d) ?  creep
    Fail:  (8) p(c, d) ?  creep
false.
```

Figure 9: SWI-Prolog traces of a “true” and a “false” instantiation.

we failed to prove that q is deducible from the program or matches some data in it. This should clarify why above I spoke of a "semantics of quasi-truth values" for LP.

This is perfectly evident if we bear in mind that SLD resolution is a convenient proof calculus to "express" the proof mechanism proper of LP, which is *reduction*, in turn based on UMP. In this proof mechanism, what is sought is a repeated replacement of a goal A by the head of a rule $A' = A$ until no more replacements can take place: If we obtain the empty resolvent, then the reduction is considered successful, and A is deducible from the database via UMP; otherwise, this algorithm fails. In a sense, then, and a rather implicit one, we can say that if an LP interpreter fails to prove that q, then $\neg q$ must be the case.

This *implicitness* of negation can be stated as follows:

Definition 60. Given an LP program Π and a query that is a ground atom q, we have either $?\square$ or $?\boxtimes$:

$$\begin{array}{lll} (?\square) & \Pi, \neg q \vdash_{sldres} \square & \Rightarrow \quad q \\ (?\boxtimes) & \Pi, \neg q \nvdash_{sldres} \square & \Rightarrow \quad \neg q \end{array}$$

Then, $?\boxtimes$ can be interpreted in terms of $\neg$ as "$\neg q$ succeeds if q finitely fails" and $?\square$ as "$\neg q$ fails if q finitely succeeds", where by "finitely succeeds" ("finitely fails") it is meant that there is an SLD-resolution tree $\mathcal{T}_{\Pi,\neg q}$ with at least one $\square$-leaf (with no $\square$-leaf, respectively). We accordingly call this interpretation of the symbol "$\neg$" *negation by failure* (NBF).

Definition 61. Let the ground atom A be a goal and let $?A$ denote a query. Then, the following rule of inference is called negation by failure:

$$(\text{NF}) \qquad \frac{?\,(\neg A) \quad ?A \text{ fails}}{\square}$$

Rather than a rule of inference, NF is a *meta-rule*. There are two cases in which NBF may be an interpretation for $\neg$, depending on the way we define *completeness* with respect to a program. Firstly, the program, or database, is *complete* in the sense that it contains all the information that is "true" about some domain. We can assume that what is "true" is also known to be "true," and what is not known to be "true" is "false." We call this *closed-world assumption* (CWA).

Example 62. CWA is actually a very frequent kind of reasoning. Suppose you want a direct flight from Berlin, Germany to Sydney, Australia. You consult the flights available in a specific airline. If there is no direct flight Berlin-Sydney listed, you conclude that there is no such flight in this airline, or that the contrary statement is false. In either case, you can be considered to employ NBF.

CWA was originally proposed by R. Reiter (1978), and the CWA interpretation of $\neg$ in LP was defended in Shepherdson (1984). This interpretation is highly relevant, as it makes of LP a kind of *non-monotonic* deduction. In fact, CWA is, too, an inference meta-rule.

Definition 63. Let Π be a definite LP program and let the ground atom A be a query. The following meta-rule of inference is called CWA:

$$(\text{CWA}) \qquad \frac{\Pi \nvdash_{sldres} A}{\neg A}$$

2.4.2 Negation and completeness II: The Clark completion

Secondly, a program, or database, is said to be *complete* in the sense that the statements of the definite-clause program or database are assumed to *axiomatize completely* all the possible reasons that make atomic ground formulae true. This assumption, known as *complete-database assumption* (CDB) (Clark, 1978), is concretized in the replacement of all $\leftarrow$ by $\leftrightarrow$.

Example 64. Let there be given the following LP program Π_2:

```
1.  p(X) ← q(X).
2.  p(X) ← r(X).
3.  p(X) ← t(X).
4.  q(a).
5.  q(c).
6.  r(b).
```

Then, we can simply replace statements 1-3 by

$$7.\ \mathtt{p}\,(\mathtt{X}) \leftrightarrow (\mathtt{q}\,(\mathtt{X}) \vee \mathtt{r}\,(\mathtt{X}) \vee \mathtt{t}\,(\mathtt{X}))\,.$$

called the *Clark formula for the predicate* $\mathtt{p}\,(\mathtt{X})$. The remaining Clark formulae are

$$8.\ \mathtt{q}\,(\mathtt{X}) \leftrightarrow (\mathtt{X} = \mathtt{a} \vee \mathtt{X} = \mathtt{c})\,.$$

and

$$9.\ \mathtt{r}\,(\mathtt{X}) \leftrightarrow \mathtt{X} = \mathtt{b}.$$

However, the Clark formula for the predicate $\mathtt{t}\,(\mathtt{X})$ is

$$10.\ \neg\mathtt{t}\,(\mathtt{X})\,.$$

expressing the fact that there is no X such that X is instantiated by a ground term in $\mathtt{t}\,(\mathtt{X})$. In effect, recall from above that a fact in LP is implicitly universally quantified, so that we have

$$\forall \mathtt{X}\,(\neg\mathtt{t}\,(\mathtt{X})).$$

As in Example 62, I employ here NBF. In order to have a Clark-complete database Δ corresponding to program Π_2, we have

$$\Delta = \{7, 8, 9, 10, 11, 12, 13\}$$

where 7-10 are as above but universally quantified, and the remaining statements are

$$11.\ \neg\,(\mathtt{a} = \mathtt{b})\,.$$

$$12.\ \neg\,(\mathtt{a} = \mathtt{c})\,.$$

$$13.\ \neg\,(\mathtt{b} = \mathtt{c})\,.$$

Definition 65. Given a definite-clause theory or database Θ, the *Clark completion of* Θ, denoted by $Compl\,(\Theta)$, is the theory consisting of (i) all the Clark formulae for every predicate $P \in \Theta$, and (ii) statements of the form $\neg\,(t_1 = t_2)$ for every non-unifiable pair of terms $(t_1, t_2) \in T\,(\Theta)$.

Algorithm 2 Clark completion

Input: A definite-clause theory or database Θ
Output: $Compl\,(\Theta) = \Delta$

1. Rewrite every individual definite clause

$$(\mathcal{C}_{LP}) \quad \forall \vec{X}\left(p\left(\vec{t}\right) \leftarrow B_1 \wedge ... \wedge B_n\right)$$

as

$$(CC_{LP}) \quad \forall \vec{Y}\left[\exists \vec{X}\left(B_1 \wedge ... \wedge B_n \wedge \left(\vec{Y} = \vec{t}\right)\right) \rightarrow p\left(\vec{Y}\right)\right]$$

where $\vec{Y}$ is a sequence of new variables.

   (a) If $n = 0$, then we have

$$(CC_{LP}) \qquad \forall \vec{Y}\left[\exists \vec{X}\left(\vec{Y} = \vec{t}\right) \rightarrow p\left(\vec{Y}\right)\right].$$

   (b) When $\mathcal{C}_{LP}$ is a ground formula, then we have

$$(CC_{LP}) \quad \forall \vec{Y}\left[\left(B_1 \wedge ... \wedge B_n \wedge \left(\vec{Y} = \vec{t}\right)\right) \rightarrow p\left(\vec{Y}\right)\right].$$

2. Let us simplify CC_{LP} as

$$\forall \vec{Y}\left(E \rightarrow p\left(\vec{Y}\right)\right).$$

Suppose that there are *k* such clauses, i.e. there are $E_1, ..., E_k$. Then, we have the single formula

$$\forall \vec{Y}\left[p\left(\vec{Y}\right) \leftrightarrow \left(E_1 \vee ... \vee E_k\right)\right].$$

   (a) If $k = 0$, then we have the Clark formula

$$\forall \vec{Y}\left(\neg p\left(\vec{Y}\right)\right).$$

3. For every non-unifiable pair of terms $(t_1, t_2) \in \Theta$, construct the statement

$$\neg\left(t_1 = t_2\right).$$

Given a definite-clause theory or database Θ, Algorithm 2 provides an efficient procedure for constructing the Clark completion $Compl(\Theta) = \Delta$.

We have the following important results:

Proposition 66. *For a theory Θ to be Clark-complete, all functions $f, g \in Fun(\Theta)$ must obey the following three non-logical axioms:*

$$(\mathcal{E}f1) \qquad \forall f \forall g [\neg (f = g) \Rightarrow \neg (f(x_1, ..., x_n) = g(y_1, ..., y_n))]$$

$$(\mathcal{E}f2)\ \forall fxy [(f(x_1, ..., x_n) = f(y_1, ..., y_n)) \Rightarrow (x_1 = y_1) \wedge ... \wedge (x_n = y_n)]$$

$$(\mathcal{E}f3) \qquad \forall f \forall x \neg (f(x) = x)$$

Theorem 67. *Let Π be a definite program and q a ground atom. Then,*

$$\Pi \models q \quad \Rightarrow \quad Compl(\Pi) \models q.$$

2.4.3 General programs and stratification

The above notwithstanding, for most practical applications we actually have to extend $\mathcal{C}_{LP}$ to the form

$$\left(\mathcal{C}_{LP}^{-}\right) \qquad A \leftarrow B_1, ..., B_n$$

where the B_i are either positive literals or negative literals.

Definition 68. $\mathcal{C}_{LP}^{-}$ is called a *general clause.* An LP program in which negation $\neg$ is allowed to occur in the body of a rule, i.e. an LP program with $\mathcal{C}_{LP}^{-}$, is called a *general program.*

I remark that, just as in the case of definite programs, it is not possible for a negative literal to be a logical consequence of a general program. I introduce now an LP notion, to wit, *stratification*, which will also be relevant for Section 4:

Definition 69. We say that a general LP program Π is *stratified* if the set *Pred*(Π) of the predicates in Π can be partitioned into $Pred_0(\Pi), ..., Pred_n(\Pi)$ such that if $A \leftarrow B_1, ..., B_m$ is a rule of Π and $A \in Pred_k(\Pi)$, $0 \leq k \leq n$, then:

1. If there occurs no negation in B_i for $1 \leq i \leq m$, then
$$B_i \in \bigcup_{j=0}^{k} Pred_j(\Pi).$$
2. If, however, $B_i = \neg C_i$, then
$$C_i \in \bigcup_{j=0}^{k-1} Pred_j(\Pi).$$

It can be shown that if Π is stratified, then $Compl(\Pi)$ is consistent. A simple way to check whether an LP program is stratifiable is by means of a dependency graph for Π, a directed graph $\vec{\mathfrak{G}}_{\Pi}$ in which the arcs are of the form $p \longrightarrow q$ whenever there is a rule $\mathbf{r} \in \Pi$ with $q \in Head_{\mathbf{r}}$ and $p \in Body_{\mathbf{r}}$. Because a general LP program can be seen as a generalization of a Datalog program with negation, I leave the discussion on stratification with relation to dependency graphs for Section 4 below; the application of this discussion to general LP programs is straightforward.

3 Declarative + procedural interpretation: Prolog

Prolog, abbreviating the French expression *programmation en logique*, is the main language (family) of LP. Although not the most commonly used programming language for commercial or industrial applications, Prolog has the advantage of being a Turing-complete language, which partly explains its considerable success within the (European) AI community since its original development in the early 1970s by A. Colmerauer and P. Roussel. In effect, any computable function – and we know by the Church-Turing Thesis that this is any function that can be computed by a Turing machine – can be represented by means of Prolog; in other words, Prolog can simulate any Turing machine.

As elsewhere in this paper, I concentrate on what in Prolog relates more directly to logic – or impacts on its deductive properties. Because this is basically the material above on LP, or *pure* Prolog, I now have only a few remarks on *real* Prolog. Readers seeking a more hands-on approach comprising aspects not discussed here but fundamental to Prolog (e.g., lists) can benefit from Sterling & Shapiro (1994).

3.1 Prolog and Prolog

By "real Prolog" I intend to capture pure Prolog extended with the predicates `not/1`, !/0, and `fail/0`. The distinction between *real* and *pure* Prolog is actually an important one, as only the latter is Turing-complete; the addition of *cut*, denoted by the symbol "!", divests Prolog of this desirable property.[18] But most of all, real Prolog, though of a more procedural type than the more declarative pure Prolog, shows how deductive programming can be carried out by procedures, too.

In what follows, when I write simply "Prolog" I mean "real Prolog," unless otherwise stated.[19]

Definition 70. A *Prolog rule* has the form $\mathcal{C}_{LP}$ and is written

$$\mathtt{A} : -\mathtt{B}_1, ..., \mathtt{B}_n.$$

for $n \geq 1$; if $n = 0$, then we have a *Prolog fact* and we write simply `A.`. Similarly, *Prolog goals* and *queries* are as in LP, and the same holds for *Prolog terms.*

Definition 71. A *Prolog program* Π is a sequence of Prolog facts and rules.

Definitions 70 and 71 show how similar Prolog and LP are. This similarity notwithstanding, Prolog is more procedural than LP, which is more declarative, and this is expressed in a few features present solely in Prolog. In turn, these features are associated to a specific language that we can call Prolog.[20] I define some of these features:

[18] I remark that this is not an established distinction in the field, with the label "pure Prolog" capturing many different versions of Prolog or LP.

[19] Additionally, I shall only write "Prolog rule," "Prolog fact," etc. if I need to disambiguate; otherwise, I write simply "rule," "fact," etc.

[20] In this paper, I use this font for the names of languages. In the case of the systems of LP discussed in this article, the names of the *languages* and of the corresponding *programming systems* are identical (e.g, we have Prolog as the language employed in Prolog). Consistency in using this font would require a fastidious distinction between the languages and the programs, and then between these and the corresponding programming systems – a distinction that actually goes against the common practice in the field of programming. Thus, after the introduction of each of the two LP languages Prolog and Datalog I relax the use of this font.

Definition 72. A *meta-variable* is a variable that can occur as an atom.

Definition 73. A *built-in predicate* is a predicate that is defined by internal rules of Prolog, i.e. it need not be defined explicitly when writing a program.

Built-in predicates are particularly relevant for (autonomous) management of the data in a program, but some were conceived with interaction with the user in mind. Most built-in predicates have a procedural interpretation, but this can have an impact on the declarative interpretation of a program. This is the case for the predicates `!/0` and `fail/0`. I discuss them below, given their importance, but anticipate that they cannot occur in the head of a rule, i.e. they are always goals. Another built-in predicate of particular interest from the viewpoint of the declarative interpretation is `not/1`, and I discuss it at length below.

Example 74. Further built-in predicates relevant for deductive programming (a sample) are:

- The order predicates $<, >, <=, >=$ corresponding to the more common symbols $<, >, \leq, \geq$.
- The arithmetical predicates $==$ and $= \setminus =$ for arithmetical equality and difference, and $+$, $*$, etc.
- The infix predicate $=$ for unification or matching.
- The predicate `read(X)` allows the user to unify the variable X with a specific constant, thus allowing the user to manipulate domains and instantiations at will.
- The infix predicate `is` expresses equality in Prolog; in `X is Y`, X must be instantiated to numbers or other arithmetical expressions (e.g., `X is 5 * 2`).
- The predicate `true/0` that, like `!/0` and `fail/0`, is a goal that always succeeds, can be used to force the attempt to satisfy subsequent subgoals regardless of the failure of an earlier goal.
- The predicate `call (X)` allows us to instantiate a variable X to a term that can be interpreted as a goal.

I can now define the language Prolog in the Backus-Naur notation as follows:

Definition 75. Let there be given the set

$$O_{Prolog} = \left\{ \begin{array}{c} :- \\ , \end{array} \right\} \equiv \left\{ \begin{array}{c} \leftarrow \\ \wedge \end{array} \right\}$$

of operators and the punctuation marks "," (between arguments), ".", and left and right parentheses. If we define inductively terms, atoms, and statements over a signature $\Upsilon = (Pred, Fun, ar)$ with $ar \geq 0$ denoted by $\cdot/\mathtt{n}$ for $t_1, ..., t_n$ terms, as

Terms	t ::=	*Variables*	X	::=	*e.g.*, `X` $\mid$ `John` $\mid$ `_9`
		Constants	c	::=	*e.g.*, `a` $\mid$ `john` $\mid$ 9
		Functors	$p; f$	::=	`p/n`; `f/n`
Atoms			$A\ (B, ...)$	::=	$p \mid f \mid X$
Statements		*Facts*	A	::=	$\mathtt{A.}$
		Rules	r	::=	$\mathtt{A} : -\mathtt{B}_1, ..., \mathtt{B}_n.$
		Goals	$G\ (A, ...)$	::=	$\mathtt{B}_1, ..., \mathtt{B}_n$

then we have the language (basic FO) Prolog and its syntax.[21]

We can extend O_{Prolog} with the operator ";" denoting disjunction. I have already used it in queries to prompt all the possible replies, but it can also be employed in the body of rules. However, it is interpreted as different clauses. Say you have a rule $\mathtt{A} : -\mathtt{B}_1; \mathtt{B}_2$. Then, the interpreter will consider this as two rules, to wit $\mathtt{A} : -\mathtt{B}_1$. and $\mathtt{A} : -\mathtt{B}_2$.. Example 84 below shows the utility of this operator in the body of rules.

Strictly, Prolog statements are solely facts and rules; goals are parts of rules, and only as such can they be considered statements. Queries are goals with a question mark, i.e. "statements" of the form $? - \mathtt{q}$ where $\mathtt{q} = G$; in other words, queries are not statements of the programs. However, we can extend O_{Prolog} with the operator ?. As a matter of fact, full Prolog contains more operators, some of which are actually built-in predicates.

The above allows for a redefinition of a Prolog program:

Definition 76. A Prolog program is a sequence of facts and rules in which there can occur (i) meta-variables and (ii) built-in predicates.

This definition shows that, though an FO logical language, Prolog differs from the standard language of FO logic in important ways. The occurrence of meta-variables and of functions as atoms, as well as of predicates as arguments (see below), accounts for what I above referred to as the ambivalent syntax of LP. It should be noted, however, that this ambivalent syntax does not entail that Prolog is an orderless language. I omitted quantification in Definition 75, because this is a rather implicit business in LP, as seen above. In effect, we can use Prolog at a purely propositional level, but then Prolog is a rather uninteresting programming system; and we can use it at second or higher orders, but this requires further specifications that I do not discuss here.[22]

3.2 Logic + control: `!` and `fail`

3.2.1 Adding control to logic

In the Introduction, I touched upon the aspect of computational logic summarized as the equation *Algorithm = Logic + Control*. As seen above, SLD resolution is sound

[21]In Prolog, a predicate $p(t_1, ..., t_n)$ with n terms, denoted by `p/n`, may itself be a term in the negative literal $not(p(t_1, ..., t_n))$. This is a feature that contributes to the ambivalent syntax of Prolog, a feature already mentioned above for LP and which I further discuss below.

[22]See, for instance, Sterling & Shapiro (1994), Chapter 16, for 2nd-order Prolog programming.

and complete for LP; but this does not mean that it is efficient. In fact, it is not, as it entails the construction and complete traversal of often complex SLD-resolution trees. This is precisely the point where we add the factor "Control" to "Logic," making real Prolog more efficient than pure Prolog as far as computation is concerned. The cost of controlling deductive computation may be high (see below), but the trade-off may be favorable in the end, at least in some cases.

Control is added to logic in Prolog by means of special atoms that have a procedural impact on the deductive computation without for that changing the meaning of a program. In other words, the programmer controls how the deductive computation takes place. This is done by means of predicates that, inside the body of a rule, have an interpretation of the kind "If X, then do Y," rather than the more logical interpretation "If X, then Y follows." We speak here of the predicates !/0, read "cut," and `fail/0`. As 0-ary predicates, they are interpreted as operators rather than predicates.

3.2.2 The operator !

I begin with the operator !, called *cut operator*: Given an SLD-resolution tree, its use is intended to cut off failing branches and to prune succeeding branches.

Definition 77. Given an LP program $\Pi = \{\mathcal{C}_1, ..., \mathcal{C}_n\}$, where $\mathcal{C}$ abbreviates $\mathcal{C}_{LP}$, let $\mathcal{C}_i \subseteq \Pi$ be the clause

$$A \leftarrow B_1, ..., B_j, !, B_{j+2}, ..., B_n$$

and let G be a goal. Then, if G unifies with $A \in \mathcal{C}_i$, and $B_1, ..., B_j \in \mathcal{C}$ succeed, $! \in \mathcal{C}$ has the following effects:

1. The program is so to say committed to $\mathcal{C}_i$ to reduce G, no alternative to $A \in \mathcal{C}_k$, $k > i$, being considered.
2. In case B_i, $i > j + 1$, fails, then backtracking goes no further back than !, the $B_1, ..., B_j$ being pruned from the search tree.
3. If the backtracking search goes as far back as to !, then ! fails, and the search goes back to the last $\mathcal{C}_j$ prior to the choice of $\mathcal{C}_i$.

Although real Prolog is often seen as a good example of a programming system that allies declarative and procedural paradigms, this alliance is not without issues. As a matter of fact, this alliance entails the loss of Turing-completeness, a very desirable property of pure Prolog.

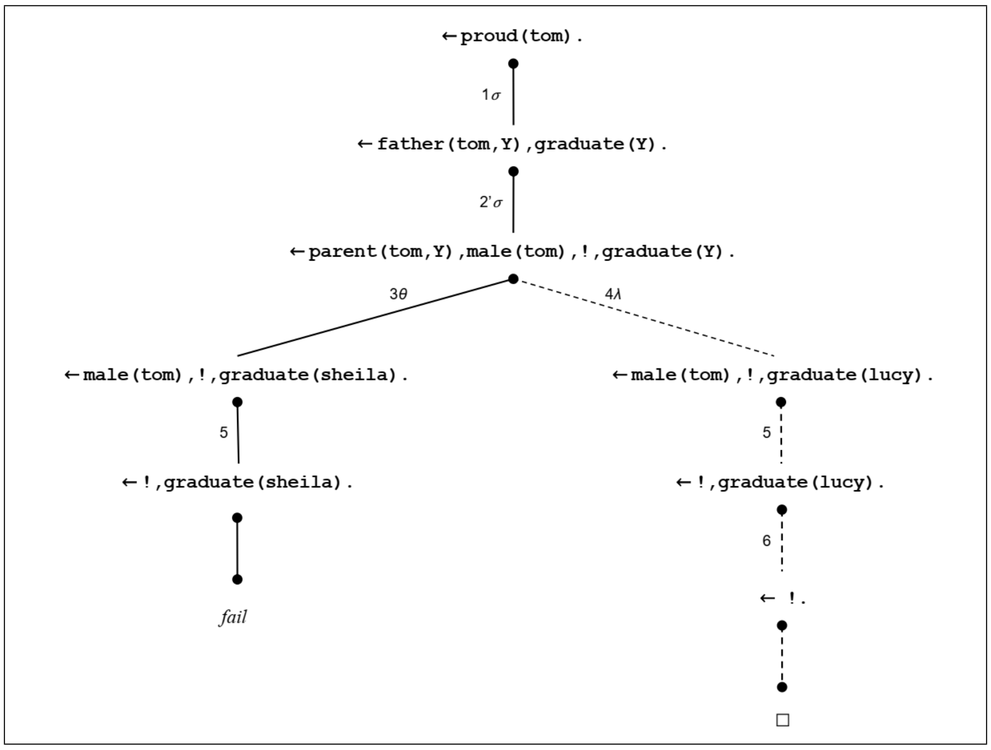
Example 78. Fathers of graduate children are typically proud. This can be deduced from the following program called *Proud_fathers*:

```
1. proud(X) :−father(X,Y),graduate(Y).
2. father(X,Y) :−parent(X,Y),male(X).
3. parent(tom,sheila).
4. parent(tom,lucy).
5. male(tom).
6. graduate(lucy).
```

Given the query ? − `proud(tom)`. as input, SWI-Prolog will answer "true." Edit the program by adding ! at the end of rule 2, so that you have

2′. `father(X,Y) :−parent(X,Y),male(X),!.`

Given the same input, SWI-Prolog now replies "false." This is so because Sheila, who is not graduate (by NBF), is the first of Tom's children in the program sequence, and after ! there are no more attempts to find any other children. Figure 10 shows – with a broken line – the pruned successful branch of the SLD-proof tree with the substitution $\lambda = \{\texttt{Y} = \texttt{lucy}\}$ and the first, failed, branch with substitution $\theta = \{\texttt{Y} = \texttt{sheila}\}$ ($\sigma = \{\texttt{X} = \texttt{tom}\}$). If we now interchange the positions of facts 3 and 4, so that Lucy appears as the first child of Tom, the same query will be answered "true," even with 2′; additionally, the failure branch will be pruned from the tree, thus making the computation more efficient.



←proud(tom).
1σ
←father(tom,Y),graduate(Y).
2'σ
←parent(tom,Y),male(tom),!,graduate(Y).
3θ
4λ
←male(tom),!,graduate(sheila).
←male(tom),!,graduate(lucy).
5
5
←!,graduate(sheila).
←!,graduate(lucy).
6
fail
← !.
□

Figure 10: An SLD-proof tree for a Prolog program with !.

Example 78 illustrates clearly the fact that ! is a problematic operator in deductive programming. Indeed, the addition of ! to the program *Proud_fathers* does impact significantly on the meaning of the program. Let us denote this program by Π and the query ? − `proud(tom)`. by G. Then we have

$$\Pi \vdash G$$

accordingly to its meaning, to wit, $M(\Pi) = G = \{\texttt{proud(tom)}\}$. Let us now consider the same program but with ! added, i.e. $\Pi \cup \{!\}$. Then we have

$$\Pi \cup \{!\} \nvdash G$$

as well as[23]

$$\Pi \cup \{!\} \vdash \neg G.$$

Clearly, $M(\Pi \cup \{!\}) \neq G$. This shows how a procedural interpretation of a program may in fact impact on the declarative meaning thereof. The fact that interchanging facts 3 and 4 corrects the problem can be used to "blame" the programmer for the undesirable results above, which is partly true, but it should also highlight the problems that may arise with the addition of ! to a program, particularly so in the case of complex programs. When the operator ! does change the meaning of a program, we call it a *red cut* – the opposite being a *green cut.*

Theorem 79. *Prolog with the operator ! is not Turing-complete.*

Proof. (Idea) In pure Prolog we are assured that, given a program Π and a query q, Prolog either proves or refutes $\Pi \vdash q$. In effect, Prolog carries out the unification algorithm and is guaranteed to stop when all the variables have been instantiated or when $\Pi \vdash q$ has been disproved. The operator ! "cuts off" branches of the search tree, possibly precisely those that contain the proof of $\Pi \vdash q$. ☐

3.2.3 The `fail` operator

Yet another way to control deductive computation in Prolog is by means of the predicate `fail/0`. We already know *fail* from the context of SLD-resolution proofs for given goals: Given an LP program Π, if a goal G succeeds, then the goal $\neg G$ fails at all the possible attempts to satisfy it by backtracking, and we have

$$\Pi \vdash G.$$

Otherwise, i.e. if a goal G fails (for the reasons above), $\neg G$ succeeds and we have

$$\Pi \nvdash G.$$

In the latter case, to the query `? − G.` SWI-Prolog replies "false." The predicate `fail/0`, which just like ! is rather an operator, is a means to control this deductive effect: The programmer elects the goals that are to fail. But contrarily to !, `fail` has the procedural effect that no alternative solution whatsoever is considered to satisfy a goal followed by it: The goal immediately fails.

Definition 80. The 0-ary predicate `fail` is a built-in predicate with the empty definition whose procedural interpretation is as follows: Given this predicate in the body of a rule, the head of the rule fails.

Example 81. Let the Prolog program with the single rule `crazy(daffy) : −fail.` be given. Then, given the query `? − crazy(daffy)`, SWI-Prolog will reply "false."

[23]That is to say, if given the goal `? − not(G).` as input, SWI-Prolog, for instance, will reply "true." See next Section for the predicate `not/1`.

According to Definition 75, both ! and `fail` are goals; as built-in undefined goals, they have the property that they always succeed, so they should be cautiously employed when writing a Prolog program. Their combination, in particular, is a powerful means to control deductive computation (see next Section).

3.3 Negation in Prolog

3.3.1 The predicate `not`

As seen, the operator ! may – and often does – lead to incorrect programs, and `fail` requires cautious employment as well. An alternative to each of these, but actually employing them in combination, is *negation*. In effect, a form of NBF can be implemented in Prolog by means of ! and `fail` in this exact order. We express it by means of the predicate `not/1` for convenience, but if using SWI-Prolog, the unary predicate \+ must be used instead.

Definition 82. The Prolog predicate `not/1` is defined as:

```
not(X) :- X, !, fail.
not(X).
```

We call this combination of ! and `fail` *cut-failure negation*.

This is a built-in predicate, so the programmer does not need to write this definition as part of a program.

Example 83. Given the program above *Proud_fathers* SWI-Prolog gives the following replies to the indicated queries:

```
?- graduate(sheila).
false

?- not(graduate(sheila)).
true

?- not(graduate(lucy)).
false
```

For the first query SWI-Prolog simply applies NBF: The goal `graduate(sheila)` does not match any data in the program. As for the second and third queries, SWI-Prolog applies the definition of `not/1`. In these two queries, the goal G is $\leftarrow$ `not(graduate(Y)).` and the subgoal G' is $\leftarrow$ `graduate(Y)..` If G' succeeds, then the operator ! cuts off the possibility that G may be the case and G fails. In this case (the third query above), SWI-Prolog replies "false." If, however, G' finitely fails, the search algorithm tries G by means of backtracking and G succeeds. In this case (the second query above), the answer is "true."

It should be noted that cut-failure negation is indeed a form of NBF: Given the program *Proud_fathers*, the query `?- not(graduate(a)).` will be answered "true" for any ground term a such that $a \neq lucy$. This leaves us with an infinite domain in which the predicate `not(graduate(X))` $\in Proud\_fathers$ is "true."

Example 84. As said above, for practical reasons the predicate `not/1` is often essential in a program. For instance, suppose an avian center requires a database from which it can be easily known which bird(s) might have escaped the closed precincts by flying. This center has all sorts of birds, a few of which do not normally fly, but most of which do fly. By adding a rule to the database like

```
fly(X) : −bird(X), not(abnormal(X)).
```

we can eliminate all non-flying birds, i.e. the "abnormal" birds, such as penguins, ostriches, etc. This possibly still leaves a lot of flying birds, but we can further reduce the possibilities by employing another negated atom, say, `not(quarantined(X))`. The program *Avian_center* can be sketched as follows:[24]

```
fly(X):-bird(X),\+(abnormal(X)),\+(quarantined(X)).
bird(X):-canary(X);nightingale(X);penguin(X);ostrich(X);
    crow(X);emu(X);woodpecker(X);turkey(X);duck(X);hen(X).
abnormal(X):-penguin(X);ostrich(X);emu(X);turkey(X);hen(X).
penguin(toto).
ostrich(sheila).
emu(tom).
turkey(sam).
turkey(sandra).
hen(lolita).
canary(roberto).
nightingale(sarita).
crow(bob).
woodpecker(lola).
duck(cassandra).
duck(samantha).
quarantined(roberto).
quarantined(bob).
```

The query ? − `fly(X)`. and the operator ";" will output all the reduced possibilities:

```
X = sarita;
X = lola;
X = cassandra;
X = samantha;
false
```

More negated goals will help us to obtain a further reduced sample of the birds that could have escaped by flying, from which sample it might be much easier to determine the fleeing bird(s).

[24]In, for example, the predicate `penguin(toto)`, *toto* is just a constant; it can stand equally for a single penguin (say, in a shelter for birds) or for all the penguins (in a large avian center, for instance). Note also the useful operator ";" in the body of the second and third rules of this program.

3.3.2 Restricted Prolog programs and queries

In Definition 82, it should be noted that X is a meta-variable, a syntactical feature that is absent in CFOL, as already commented upon. This entails yet another feature that violates CFOL syntax: Given the n-ary function symbol $\mathtt{f}\left(\mathtt{t_1}, ..., \mathtt{t_n}\right)$, then the query $? - \mathtt{not}\left(\mathtt{f}\left(\mathtt{t_1}, ..., \mathtt{t_n}\right)\right).$ in Definition 82 requires a computation in which $\mathtt{f/n}$ is both a function and a predicate symbol. The same is true of a rule such as

$$\mathtt{p}\left(\mathtt{t_1}, ..., \mathtt{t_n}\right) : -\mathtt{not}\left(\mathtt{p}\left(\mathtt{t_1}, ..., \mathtt{t_n}\right)\right).$$

in which $\mathtt{p/n}$ is a predicate symbol in the head and a function symbol in the body of the rule. This requires that we accept that, given a signature for a Prolog program Π, we may have

$$Pred\left(\Pi\right) \cap Fun\left(\Pi\right) \neq \emptyset.$$

In other words, we take ambivalent syntax to be a feature of Prolog.

One of the main goals when writing a Prolog program should be to make the deductive computation on it run error-free. Useful as negated atoms may be in Prolog, negation is unfortunately a source of errors. Let us say that a program is *safe* if no errors are generated, namely with respect to the SLD-resolution process.

Definition 85. We say of an atom A that it is *unsafe* if

1. A is a meta-variable.
2. $A = \mathtt{not}\left(\mathtt{X}\right)$, for X a variable.
3. $A = \mathtt{not}\left(\mathtt{not}\left(\mathtt{t}\right)\right)$, for a term t.

Definition 86. A Prolog query is *meta-safe* if none of its atoms is unsafe.

If a meta-variable is selected in the SLD-resolution process, then errors are bound to occur. Hence, we have a problem, as Definition 82 does indeed contain a meta-variable.[25] The possible solution runs as follows, from definitions to theorem:

Definition 87. A Prolog program Π containing the two rules of Definition 82 as a prefix and in which for every LP atom $\neg A$ we write $\mathtt{not}\left(\mathtt{A}\right)$ instead is called a *restricted Prolog program* if, when omitting the mentioned prefix and writing $\neg A$ instead of $\mathtt{not}\left(\mathtt{A}\right)$, we still have a syntactically correct general LP program. In the same way, we say that a Prolog query in which we write $\mathtt{not}\left(\mathtt{A}\right)$ instead of $\neg A$ is a *restricted Prolog query* if it is a syntactically correct LP query when writing $\neg A$ instead of $\mathtt{not}\left(\mathtt{A}\right)$.

Lemma 88. *Let q be a meta-safe Prolog query and Π a restricted Prolog program. Then all resolvents of q are meta-safe.*

Theorem 89. *The computation of a restricted Prolog query and a restricted Prolog program generates no errors.*

[25]It should be noted that even if we choose to define the first rule without a meta-variable, namely as

$$\mathtt{not}\left(\mathtt{p}\left(\mathtt{t_1}, ..., \mathtt{t_n}\right)\right) : -\mathtt{p}\left(\mathtt{t_1}, ..., \mathtt{t_n}\right), !, \mathtt{fail}.$$

we have the ambivalent-syntax problem.

4 Purely declarative interpretation: Datalog

Henceforth I shall be focusing on databases, namely on *relational databases*. More precisely, I shall concentrate on these as *deductive databases*. However, as stated in the Introduction I approach databases as KBs, i.e. as collections of data that are facts (cf. Augusto, 2020c). This is not (explicitly) the standard approach in the literature and I give the following suggestions for the readers interested in standard discussions: For the large topic of databases, the reader may benefit from Date (2004); for databases and LP I refer the reader to Ceri et al. (1989, 1990), Gallaire et al. (1984), Minker (1997) – which also provides historical aspects on the topic of logic and databases –, and Abiteboul et al. (1995), a comprehensive textbook. Finally, Greco & Molinaro (2016) provides an extensive elaboration on Datalog.

In terms of deductive programming, Datalog has a purely declarative interpretation. This means that as far as the equation *Algorithm* = *Logic* + *Control* is concerned, we have now the identity *Algorithm* = *Logic*. This identity is supported by an FO language – a subset of Prolog – that, besides being capable of representing knowledge in terms of relations, also allows deduction to be carried out over sets of formulae expressing relations. This means that Datalog both is a *relational language* over which we can build *relational databases* and allows deduction over those databases. Although my interest falls mainly on Datalog as a deductive database, these relational notions need to be briefly elaborated on, as in fact Datalog is the addition to relational databases of logical features pertaining to CFOL. In other words, Datalog is the formalization of relational databases by means of a subset of the standard CFOL, a formalization that entails the deductive properties inherent in this subset when considered in the framework of a calculus under (mostly) Herbrand semantics. Anticipating the due formal definitions, this subset is function-free – which in principle guarantees decidability under Herbrand semantics – and includes solely the connectives for conjunction and (inverse) implication, so that, just as in the case of Prolog, we shall be employing SLD resolution over sets of Horn clauses.

The contents of Section 2 hold *mutatis mutandis*, as I include deductive databases in deductive programming, for which the notions *query* and *query system* are central. Exceptions and specifications will be clearly stated. In particular, I go on denoting arbitrary variables by the uppercase letters $X, Y, ...$ and arbitrary predicate symbols by lowercase letters (e.g., p).

4.1 Relational languages and databases

Definition 90. Let L be a formal language over a signature $\Upsilon = (Pred, \emptyset, Cons)$.[26] Then, if the assertions of L are of the form $R(t_1, ..., t_n)$, where R is an *n-ary relation symbol* and the t_i are variables and/or constants, L is called a *relational language*.

Although, as it will be seen, we may consider a relational language as a function-free fragment of the language of CFOL, in the more narrowly defined context of relational databases the following specifications need to be made:

Definition 91. Let a domain $\mathscr{D}_i$, for $1 \leq i \leq n$, be a set of *values*.

[26]Or $\Upsilon = (Pred, \emptyset, ar \geq 0)$.

1. We say that the (commutative) Cartesian product $\mathscr{D}_1 \times ... \times \mathscr{D}_n$ of $i = 1, ..., n$ domains is a *(finite) relation* $R \subseteq \mathscr{D}_1 \times ... \times \mathscr{D}_n$ of arity $i = 1, ..., n$.

2. The values of domain $\mathscr{D}_i$ are called *attributes*. Let $A_1, ..., A_n$ be attributes. Then, a relation R with n attributes defines a (non-ordered) n-tuple $(A_1, ..., A_n)$ such that $R(A_1, ..., A_n)$ is called a *relation scheme* and
$$R = \left\{ \left(c_1^1, ..., c_n^1\right), ..., \left(c_1^k, ..., c_n^k\right) \right\}$$
where the c_i^j, $1 \leq j \leq k$, denote specific elements (items or individuals), is said to be an *instance* (or *extension*) thereof.

3. A finite set of instances of relations $R_l \subseteq \mathscr{D}_1^l \times ... \times \mathscr{D}_n^l$, $l = 1, ..., m$, is a *(finite) relational database*.

We can envisage a relation R as a table of which the attributes A_i, $i = 1, ..., n$, are the columns and the instances $u_j = \left(c_1^j, ..., c_n^j\right)$, $1 \leq j \leq k$, are the rows.

Example 92. Let us revisit the avian center of Example 84 above. A relational database for this center can be constructed by means of a relation $R(A_1, A_2)$ where $R = BIRD$, $A_1 = SPECIES$, and $A_2 = NAME$. Attributes A_1, A_2 are associated with the corresponding domains $\mathscr{D}_1, \mathscr{D}_2$ of, respectively, *bird species* and *proper nouns*. The table for this relation is shown in Figure 11.

BIRD	
SPECIES	**NAME**
Penguin	Toto
Ostrich	Sheila
Emu	Tom
Turkey	Sam
Turkey	Sandra
Hen	Lolita
Canary	Roberto
Nightingale	Sarita
Crow	Bob
Woodpecker	Lola
Duck	Cassandra
Duck	Samantha

Figure 11: Table for $BIRD\,(SPECIES, NAME)$.

Then, the relational database *Avian_center* is constituted by the finite instances of the relation scheme $BIRD\,(SPECIES, NAME)$. (Note how for the values *Turkey* and *Duck* there are two rows on the table for this relation scheme.) The extension of this relation scheme is the set:

$$BIRD = \left\{ \begin{array}{c} (penguin, toto), (ostrich, sheila), (emu, tom), \\ (turkey, sam), (turkey, sandra), (hen, lolita), \\ (canary, roberto), (nightingale, sarita), \\ (crow, bob), (woodpecker, lola), \\ (duck, cassandra), (duck, samantha) \end{array} \right\}$$

More frequently than not, a relational database has more than a single relation scheme. As a matter of fact, a relational database can be further specified as a set of ground assertions over a relational language L *together* with the following axioms:

Definition 93. If $c_1, ..., c_p$ are all the constant symbols of a relational language L and $R\left(c_1^1, ..., c_n^1\right), ..., R\left(c_1^m, ..., c_n^m\right)$ denote all the facts under R for each relational symbol $R \in \Sigma_{\mathsf{L}}$, then the following, together with the axioms for equality, are the *axioms of a relational database over* L:[27]

$$\text{(UN)} \qquad (c_1 \neq c_2), ..., (c_1 \neq c_p), (c_2 \neq c_3), ..., (c_{p-1} \neq c_p)$$

$$\text{(DC)} \qquad \forall X\left[(X = c_1) \vee ... \vee (X = c_p)\right]$$

$$\text{(CO)} \qquad \forall X_1 ... \forall X_n [R(X_1, ..., X_n) \rightarrow$$
$$\left(\left(X_1 = c_1^1\right) \wedge ... \wedge \left(X_n = c_n^1\right)\right) \vee ... \vee \left(\left(X_1 = c_1^m\right) \wedge ... \wedge \left(X_n = c_n^m\right)\right)]$$

I denote the set of all the axioms above, often called the *particularization axioms*, by AX_Δ, where Δ denotes an arbitrary relational database.[28] This axiomatization, firstly formulated in Reiter (1984), formalizes three assumptions of relational databases: The *unique-name assumption* (axiom UN), which states that every distinct individual in the database has a different name or, which is the same, individuals with different names are distinct; the *domain-closure assumption* (axiom DC), according to which there are no other individuals than those in the database; the *completion assumption* (axiom CO), which states that the only tuples that a relation R can have are those specified in the relational database. Importantly, the completion assumption "translates" the *closed-world assumption* (CWA) already known from the discussion above on Prolog: That there are no other instances of some relation R than those implied by the database entails that $\neg R(c_1, ..., c_n)$ is assumed to be true if the tuple $(c_1, ..., c_n)$ does not constitute an instance of R in the database.

Definition 94. Given a finite set of ground assertions R, a relational database Δ is defined as:

$$\Delta = R \cup AX_\Delta$$

[27]The axioms for equality are as follows:

$$\begin{array}{ll}
(\mathcal{E}1) & \forall x\,(x = x) \\
(\mathcal{E}2) & \forall x \forall y\,((x = y) \rightarrow (y = x)) \\
(\mathcal{E}3) & \forall x \forall y \forall z\,[((x = y) \wedge (y = z)) \rightarrow (x = z)] \\
(\mathcal{E}4) & \forall f \forall x \forall y\,[(x = y) \rightarrow (f(x) = f(y))] \\
(\mathcal{E}5) & \forall P \forall x \forall y\,[(x = y) \rightarrow (P(x) = P(y))]
\end{array}$$

$\mathcal{E}4$ is irrelevant for Datalog. To simplify, consider $\mathcal{E}5$ as formalized in FOL:

$$\forall x \forall y\,[(P(x) \wedge (x = y)) \rightarrow P(y)].$$

[28]Not to be confused with a general database $\varDelta$, as used above in the context of Prolog programs.

Example 95. Given the relational database *Avian_center*, by axiom UN we can assume that Sam and Sandra are two distinct turkeys, i.e. $sam \neq sandra$; by DC, we know that these are the only two turkeys in the database, as for all the birds X in the database and the value *Turkey* we have $(X = sam) \vee (X = sandra)$; and by CO we can safely assume that Sheila is not a turkey, as $(turkey, sheila) \notin BIRD$, and hence we can conclude $\neg BIRD\,(turkey, sheila)$.

Definition 96. Let there be given a relational database Δ. Then:

1. The set of ground assertions of the form $R\,(c_1, ..., c_n)$, is called the *extensional database* (EDB).
2. The set of axioms of the form $R_1\,(\vec{x}_1) \leftarrow R_2\,(\vec{x}_2)\,, ..., R_m\,(\vec{x}_m)$, where the $\vec{x}_i$ are tuples of appropriate arities, is called the *intensional database* (IDB).

4.2 Deductive databases and Datalog

Given a relational language, for some relation $R \subseteq \mathscr{D}_1 \times ... \times \mathscr{D}_n$, the corresponding attributes $A_1, ..., A_n$ can be expressed as variables $X_1, ..., X_n \in Vi$ (as constants $c_1, ..., c_n \in Cons$), so that we actually have $p\,(X_1, ..., X_n)$ (respectively, $p\,(c_1, ..., c_n)$), where p is a *predicate symbol* of arity n. In other words, we can express n-ary relations of a relational database as atoms of the form $p\,(t_1, ..., t_n)$ where $t_i \in (Vi \cup Cons)$. The language Datalog allows us to do this.[29]

Definition 97. Let there be given the set

$$O_{Dlog} = \left\{ \begin{array}{c} :- \\ , \end{array} \right\}$$

of operators and the punctuation marks "," (between arguments), ".", and left and right parentheses. If we define inductively terms, functors (predicate, or relation, symbols of arity n), atoms, and statements over a signature $\Upsilon = (Pred, \emptyset, Cons)$ as

$$\begin{array}{lll|lcll}
Terms & t & ::= & Variables & X & ::= & e.g., \mathtt{X}\,|\,\mathtt{John} \\
 & & & Constants & c & ::= & e.g., \mathtt{a}\,|\,\mathtt{john}\,|\,\mathtt{8} \\
 & & & & & & \\
Functors & & & & p & ::= & \mathtt{p}\,(\mathtt{t_1}, ..., \mathtt{t_n}) \\
 & & & & & & \\
Atoms & & & & A\ (B, ...) & ::= & p \\
 & & & & & & \\
 & & & Facts & A & ::= & \mathtt{A.} \\
Statements & & & Rules & r & ::= & \mathtt{A} :- \mathtt{B_1}, ..., \mathtt{B_n}. \\
 & & & Goals & G\ (A, ...) & ::= & \mathtt{B_1}, ..., \mathtt{B_n}
\end{array}$$

then we have the language Datalog and its syntax if the following *safety conditions* are satisfied:

1. Each fact A is a ground atom.

[29] Cf. footnote 20 above for the use of this font for language names.

2. Given a rule $\mathbf{r} = \mathtt{A} : -\mathtt{B}_1, ..., \mathtt{B}_n.$, if a variable X occurs in A, the *head* of the rule (i.e. $\{A\} \subseteq Head_{\mathbf{r}}$), then it must occur at least in one of the (non-arithmetical) $B_i \in Body_{\mathbf{r}}$ constituting the *body* of $\mathbf{r}$ (for $\{B_1, ..., B_n\} \subseteq Body_{\mathbf{r}}$).

From the above definitions, it is obvious that Datalog is a *function-free* subset of the standard language of CFOL. Let us denote this language by L1 and its function-free fragment by $\mathsf{L1}_{\mathit{ff}}$.

Proposition 98. Datalog $\subseteq \mathsf{L1}_{\mathit{ff}}$ *is a relational language.*

Proof. Trivial. □

Trivial as the proof above might be, some remarks need to be made. Let $\mathscr{RTL}$ be the class of relational languages; then, Datalog is a relational language in the sense that we have Datalog $\supseteq \mathscr{RTL}$. This containment relation formalizes the fact that Datalog is more expressive than a relational language. This is particularly so with respect to recursion: While this property is expressible in Datalog, it is not a feature of the class $\mathscr{RTL}$. But the impact of applying Datalog on a relational database has further interesting and important consequences. I next elaborate on this topic.

The main objective of "logicizing" relational databases, namely by means of Datalog, is that of allowing deduction to be carried out over them. We call these "logicized" relational databases *deductive databases* (abbrev.: DDBs). In effect, given a query q and a relational database Δ, we may wish to know whether $q \in \Delta$. This is clearly a decision problem, and, as is well known, we can formulate it as a logical problem: Given the pair (Δ, q), we wish to find out a "Yes/No" answer to the question

$$\Delta \overset{?}{\vdash} q.$$

This logical formulation, in turn, requires that our relational database be capable of having deduction carried out over it, so that $q = R\left(c_1^i, ..., c_n^i\right)$ may be a *new* relation instance – a *view*, in DDB jargon – deducible from Δ, but not an explicit assertion in Δ. Furthermore, we aim at adequate deduction over our database, so that we want the proof system $\mathcal{P}_\Delta$ for Δ to correspond to some semantics $\mathfrak{S}_\Delta$ in the sense that we have $\Delta \vdash q$ iff we have $\Delta \models q$.

Interestingly enough, all we have to do is to see Δ as an FO logical theory. In effect, just compare Definition 94 above with the following definition, armed with the knowledge that $AX_\Delta \subseteq (EDB \cup IDB)$.

Definition 99. A *theory* Θ is a deductively closed (sub)set of formulae of some logical language L, i.e. $\Theta = F_{\mathsf{L}}^{(\prime)} \cup AX \cup RI$.

1. Let Θ be a theory. Θ^{*} is said to be an *extension* of Θ if every theorem of Θ is a theorem of Θ^{*}, i.e. if $\Theta \subset \Theta^{*}$, and Θ' is said to be a *subtheory* of Θ if $\Theta' \subset \Theta$.

2. A theory Θ is said to be *consistent* if it is *not* the case that we have *both* $\Theta \vdash \phi$ *and* $\Theta \vdash \neg\phi$ for some formula ϕ. Otherwise, Θ is *inconsistent.*

3. A theory Θ is *complete* if it is the case that *either* $\Theta \vdash \phi$ *or* $\Theta \vdash \neg\phi$ for some formula ϕ. Otherwise, Θ is *incomplete.*

We shall consider this FO formalization of Δ to be a deductive database.

Definition 100. A *deductive database* (DDB), denoted by Δ, is defined as:

$$\Delta = EDB \cup IDB$$

Proposition 101. *A deductive database* Δ *is a logical theory. In particular, a deductive database* Δ *is a logical theory over* $\mathsf{L1}_{ff}$*.*

Proof. Trivial. □

For reasons to do with implementation, we shall consider that every formula in Δ satisfies the particularization axioms, but shall not consider explicitly $AX_\Delta \subseteq \Delta$. One of the reasons for this omission is that we do not wish to employ $\mathsf{L1}^=$, the language $\mathsf{L1}$ augmented with the symbol for identity "=", or any of its reducts containing this symbol; in particular, AX_Δ would contribute to combinatorial complexity leading to inefficient implementations. This restriction can be satisfied by considering a DDB as:[30]

$$\Delta = EDB \cup IDB \cup \{\mathrm{NF}\}$$

It is evident that we can construct such a DDB by means of the logical-relational language Datalog.[31] As a matter of fact, we can speak of a *Datalog database* as a synonym for DDB.

Definition 102. Let there be given a Datalog database Δ. Then, a *Datalog program* $\mathit{\Pi}_\Delta \subseteq \Delta$ is defined as:

$$\mathit{\Pi}_\Delta = IDB \setminus EDB$$

In other words, a Datalog program is a finite set of Datalog rules. This may appear at first a very narrow definition, accepted on the basis that the EDB be physically stored in a relational database, a feature that is explained by the need to store a large number of assertions. Indeed, it may appear that an *assertion* in a relational language just is a *fact* in LP, in which a fact just is a special rule, namely a rule without a body. But this is actually not supported by the semantics for Datalog. Thus, we have both pragmatic and formal reasons to define a Datalog program as above.

I next elaborate on this, for which end we shall require a more precise definition of a Datalog program:

Definition 103. Let there be given a Datalog DB Δ. Then, the finite set

$$\mathit{\Pi}_\Delta = \underbrace{Datalog\,rules}_{IDB \subseteq \Delta} \setminus \underbrace{Assertions}_{EDB \subseteq \Delta}$$

is a Datalog program if

[30] I shall have more to say about negation in Datalog below.

[31] A DDB also typically comprises a set of *integrity constraints*, FO formulae expressing facts such as "an individual cannot be both mother and father of a child." However, these constraints are not essential in DDBs. If a constraint is indeed essential, then it can be formulated as a rule in the DDB.

1. the head predicate of every rule in Π_Δ does not occur in *EDB*, or, in other words, is an *intensional predicate* (or *relation*), and

2. predicates in *EDB* occur only in the bodies of rules in Π_Δ, being thus called *extensional predicates* (or *relations*).

Example 104. Let us retake the relational database of Examples 92 and 95, based on the Prolog program *Avian_center* (cf. Example 84). We extend this database with the supplementary relation schemas $ABNORMAL\,(c)$, $QUARANTINED\,(c)$, and $EATS\,(c_1, c_2)$. The corresponding attributes and domains should be evident from the values given. The complete relational database *Avian_Center_DB* is given in Figure 12.

$$BIRD = \left\{ \begin{array}{c} (penguin, toto)\,, (ostrich, sheila)\,, (emu, tom)\,, \\ ... \\ (duck, cassandra)\,, (duck, samantha) \end{array} \right\}$$

$$ABNORMAL = \{(penguin)\,, (ostrich)\,, (emu)\,, (turkey)\,, (hen)\}$$

$$QUARANTINED = \{(roberto)\,, (bob)\}$$

$$EATS = \left\{ \begin{array}{c} (penguin, fish)\,, (ostrich, all)\,, (emu, all)\,, \\ (turkey, seeds)\,, (hen, all)\,, (canary, seeds)\,, \\ (nightingale, seeds)\,, (crow, all)\,, \\ (woodpecker, bugs)\,, (duck, all) \end{array} \right\}$$

Figure 12: The relational database *Avian_Center_DB*.

The complete EDB *Avian_Center_EDB* is shown in Figure 13. We can now build a Datalog DB *Avian_Center_DDB* constituted by the EDB *Avian_Center_EDB* and the following Datalog rules, which, in turn, constitute the program *Avian_Sick_Prog*:

$$\left\{ \begin{array}{c} bird\,(penguin, toto)\,, bird\,(ostrich, sheila)\,, bird\,(emu, tom)\,, \\ bird\,(turkey, sam)\,, bird\,(turkey, sandra)\,, bird\,(hen, lolita)\,, \\ bird\,(canary, roberto)\,, bird\,(nightingale, sarita)\,, \\ bird\,(crow, bob)\,, bird\,(woodpecker, lola)\,, \\ bird\,(duck, cassandra)\,, bird\,(duck, samantha)\,, \\ abnormal\,(penguin)\,, abnormal\,(ostrich)\,, \\ abnormal\,(emu)\,, abnormal\,(turkey)\,, abnormal\,(hen)\,, \\ quarantined\,(roberto)\,, quarantined\,(bob)\,, \\ eats\,(penguin, fish)\,, eats\,(ostrich, all)\,, eats\,(emu, all)\,, \\ eats\,(turkey, seeds)\,, eats\,(hen, all)\,, eats\,(canary, seeds)\,, \\ eats\,(nightingale, seeds)\,, eats\,(crow, all)\,, \\ eats\,(woodpecker, bugs)\,, eats\,(duck, all) \end{array} \right\}$$

Figure 13: The EDB *Avian_Center_EDB*.

```
r1 :  sick (Y) : −quarantined (Y) .
r2 :  on_diet (Y, Z) : −bird (X, Y) , sick(Y), eats (X, Z) .
```

Here, the predicate $\texttt{on\_diet}\,(\texttt{Y},\texttt{Z})$ denotes the fact that the sick bird Y is on an extra portion of Z, the food adequate for its species, for quick convalescence. After querying the DB with respect to the currently sick birds (query: $? - \texttt{sick}\,(\texttt{Y})\,.$), staff responsible for quarantined birds can check on their dietary needs by means of the query $? - \texttt{on\_diet}\,(\texttt{Y},\texttt{Z})\,.$ For instance, knowing that Bob is quarantined, one can query the DB with the query $? - \texttt{on\_diet}\,(\texttt{bob},\texttt{Z})\,.$. (Actually, one can simply query the DB by means of $? - \texttt{on\_diet}\,(\texttt{Y},\texttt{Z})\,.$, getting at once the information on the quarantined birds and their corresponding diets.)

This example illustrates the following facts: Firstly, it shows why the separation between the relational DB and the program proper is of advantage. Indeed, the DB requires regular, perhaps daily, updating, as the number of birds in the center changes frequently and quarantines need regular updates, too. An industrial-scale relational DB may be so complex as to actually require some sort of automatic updating. On the contrary, the rules of the program do not change. Secondly, a large institution typically has different departments. In this particular example, the reader can easily imagine a department in the avian center for the quarantined birds. Staff working in this department might very likely be the only ones who need to use this program, but the DB must be available to the whole center. This accounts for the head predicates in the program rules not occurring in the DB, as well as for the fact that the DB, in turn, has predicates that do not occur in the program. For instance, in the program of Example 104, the predicate name *ABNORMAL* does not occur.

The practical account for Definitions 102-3 being given, I focus now on the formal account. This requires yet a further specification for a Datalog program:

Definition 105. Given a Datalog program $\Pi_\triangle$, let us call the finite set of all extensional predicate names or symbols the *extensional schema*, denoted by $EDB\,(\Pi_\triangle)$, and the finite set of all intensional predicate names or symbols the *intensional schema*, denoted by $IDB\,(\Pi_\triangle)$. We set

$$EDB\,(\Pi_\triangle) \cap IDB\,(\Pi_\triangle) = \emptyset.$$

Then, the *schema* of a Datalog program is defined as the set

$$Sch\,(\Pi_\triangle) = [EDB\,(\Pi_\triangle) \subseteq Pred\,(EDB)] \cup IDB\,(\Pi_\triangle)\,.$$

Intuitively, $Sch\,(\Pi_\triangle)$ gives us the *structure* of the corresponding Datalog program.

Example 106. I abbreviate the name of the program in Example 104 as A. With respect to this program, we have the sets

$$EDB\,(A) = \{\texttt{bird}, \texttt{quarantined}, \texttt{eats}\}$$

and

$$IDB\,(A) = \{\texttt{sick}, \texttt{on\_diet}\}\,.$$

The schema for this program is

$$Sch\,(A) = \{\texttt{bird}, \texttt{quarantined}, \texttt{eats}, \texttt{sick}, \texttt{on\_diet}\}\,.$$

Note that the predicate $\texttt{abnormal} \in Pred\,(EDB)$ does not belong to the schema of this program, as we have $\texttt{abnormal} \notin EDB\,(A)$. Additionally, the intensional predicate name `sick` $\in IDB\,(A)$ occurs also in the body of rule r_2, which illustrates the fact that the body of a Datalog program rule can have both extensional and intensional predicate names or symbols. It is this fact that accounts for recursion as a property of Datalog programs: In effect, recursion is the case when some intensional predicate occurs both in the head and in the body of rules.

4.3 Semantics for Datalog DDBs

I am now ready to introduce semantics for a Datalog DB, namely for a Datalog program. There are at least two semantics for a Datalog program, to wit, Herbrand semantics and fixed-point semantics. Importantly, these two semantics are equivalent. Although with different weights, I discuss here both semantics. I first give some general remarks on semantics for Datalog DBs.[32]

Definition 107. Given a Datalog DB Δ, a semantics $\mathfrak{S}_\Delta$ for Δ is the set of models based on the function

$$f_\Delta : EDB\,(\Pi_\Delta) \longrightarrow IDB\,(\Pi_\Delta)$$

mapping *EDB facts* to *IDB facts*.

This definition, which introduces the notion of an IDB fact, entails a semantics $\mathfrak{S}_\Delta$ for a DDB Δ such that $\mathfrak{S}_\Delta$ is the set of models based on a mapping from inputs over the *EDB* to outputs over the *IDB*. Hence, a semantics $\mathfrak{S}_\Delta$ for Δ just is the semantics for the program Π_Δ associated with it. This motivates the following definition that gives a new meaning to the concept *query*:

Definition 108. Given a Datalog DB Δ, a program Π_Δ is a *query* against $E_\Delta \subseteq EDB$, where E_Δ is the set of Datalog formulae expressing the known facts of Δ that can be queried via Π_Δ.[33]

In other words, a Datalog program Π_Δ provides a means of querying a subset of an associated relational DB Δ. It should not be hard to see that both the EDB facts in E_Δ and the IDB facts proper are constituted by the ground instances of the predicates over $Sch\,(\Pi_\Delta)$, denoted by $Sch\,(\Pi_\Delta)_g$, which, in turn, constitute the set of ground instances of the goals of a Datalog program. This just is the Herbrand base obtained from the Herbrand universe of Δ. I can now further specify a semantics $\mathfrak{S}_{\Pi_\Delta}$ for a Datalog program as follows.

4.3.1 Herbrand semantics

Definition 109. Given the Herbrand universe H_Δ for a Datalog DB Δ with a program Π_Δ, we denote by $H\,(\Delta)$ the Herbrand base of Δ. Let us denote by $EDB\,(\Pi_\Delta)_g$

[32] The reader is assumed to be familiar with Herbrand semantics. Augusto (2019; 2020a) contain extensive sections on this semantics.

[33] I emphasize the fact that E_Δ is not necessarily identical to *EDB*. Precisely because of this inequality, and because the *IDB* of a Datalog DB may also contain further rules, namely integrity constraints and particularization axioms, $EDB\,(\Pi_\Delta)$ and $IDB\,(\Pi_\Delta)$ are relevant notions. In particular, if a predicate symbol $p \in [EDB\,(\Pi_\Delta) \subset Sch\,(\Pi_\Delta)]$, then p occurs in a formula of E_Δ. As for the set $IDB\,(\Pi_\Delta)$, we may assume that $IDB\,(\Pi_\Delta) = Pred\,(IDB\,(\Delta))$. If $IDB\,(\Delta)$ contains more rules than those whose predicate names are to be found in $IDB\,(\Pi_\Delta)$, then we may specify the facts formed by means of $IDB\,(\Pi_\Delta) \subsetneq Pred\,(IDB\,(\Delta))$ as *IDB facts proper*. In effect, the *IDB* of any DDB is the output of querying the associated *EDB* by means of a Datalog program.

the ground atoms of $H(\Delta)$ corresponding to the predicates in $EDB(\mathit{\Pi}_\Delta)$, and by $IDB(\mathit{\Pi}_\Delta)_g$ the ground atoms of $H(\Delta)$ corresponding to the predicates in $IDB(\mathit{\Pi}_\Delta)$, so that we have:

$$Sch(\mathit{\Pi}_\Delta)_g = EDB(\mathit{\Pi}_\Delta)_g \cup IDB(\mathit{\Pi}_\Delta)_g$$

$Sch(\mathit{\Pi}_\Delta)_g$ is called an *instance* of Δ *over* $Sch(\mathit{\Pi}_\Delta)$.[34] Then, a semantics $\mathfrak{S}_{\mathit{\Pi}_\Delta}$ for a Datalog program $\mathit{\Pi}_\Delta$

1. is a function
$$g_\Delta : EDB(\mathit{\Pi}_\Delta) \longrightarrow IDB(\mathit{\Pi}_\Delta)$$
such that, given an extensional schema $EDB(\mathit{\Pi}_\Delta) \supseteq E_\Delta$, we have[35]
$$g_\Delta(EDB(\mathit{\Pi}_\Delta)) = Cn(\mathit{\Pi}_\Delta \cup E_\Delta) = g_\Delta(E_\Delta)$$
where $p(c_1^i, ..., c_n^i) \in Cn(\mathit{\Pi}_\Delta \cup E_\Delta)$ if either
   (a) $p \in EDB(\mathit{\Pi}_\Delta)$ and $p(c_1^i, ..., c_n^i) \in (E_\Delta)_g$, or
   (b) there is some rule $\mathbf{r}$ in $\mathit{\Pi}_\Delta$ such that $p(X_1, .., X_n) = Head_\mathbf{r}$ and $Body_\mathbf{r} \subseteq Sch(\mathit{\Pi}_\Delta)_g$, and
2. given some goal $G = ? - \mathtt{p}(\mathtt{X_1}, ..., \mathtt{X_n})$., we have
$$g_\Delta(G) = \bigcup \left\{ (c_1^i, ..., c_n^i) \mid (\mathit{\Pi}_\Delta \cup E_\Delta) \models p(c_1^i, ..., c_n^i) \right\}$$
for $i = 1, 2, ..., k$ and where $p \in IDB(\mathit{\Pi}_\Delta)$ and $p(c_1^i, ..., c_n^i) \in g_\Delta(EDB(\mathit{\Pi}_\Delta))$ is a ground instance of G.

Example 110. Let us retake the Datalog DB *Avian_Center_DDB* and the respective Datalog program *Avian_Sick_Prog*, that I shall abbreviate as A. As seen in Example 106, the schema of this program is

$$Sch(A) = \{\mathtt{bird}, \mathtt{quarantined}, \mathtt{eats}, \mathtt{sick}, \mathtt{on\_diet}\}.$$

An instance over this schema, denoted by $Sch(A)_g$, is shown in Figure 14. Then, given goals $G_1 = ? - \mathtt{sick(Y)}$. and $G_2 = ? - \mathtt{on\_diet}(\mathtt{Y}, \mathtt{Z})$., we have:

$$g_\Delta(G_1) = \{(\mathtt{roberto})\} \cup \{(\mathtt{bob})\}$$

and

$$g_\Delta(G_2) = \{(\mathtt{roberto}, \mathtt{seeds})\} \cup \{(\mathtt{bob}, \mathtt{all})\}$$

Note how $\mathtt{sick}(\mathtt{Y}) = Head_{r_1}$ and $\mathtt{on\_diet}(\mathtt{Y}, \mathtt{Z}) = Head_{r_2}$, as well as that $Body_{r_1} \subset Sch(A)_g$ and $Body_{r_2} \subset Sch(A)_g$, thus satisfying condition 1 in Definition 109. Furthermore, taking into consideration the associated $IDB(A)$, the domain $\mathscr{D}_A$ is reduced to

$$\mathscr{D}_A^{'} = \{canary, crow, roberto, bob, seeds, all\}.$$

[34] Equivalently, given a domain $\mathscr{D}_\Delta \supseteq \mathscr{D}_1, ..., \mathscr{D}_n$ for a DDB Δ, a *database instance* $Sch(\mathit{\Pi}_\Delta)_g$ is a finite Herbrand interpretation over $\mathscr{D}_\Delta$. Note that whereas $Sch(\mathit{\Pi}_\Delta)$ specifies the structure of the deductive database Δ, $Sch(\mathit{\Pi}_\Delta)_g$ specifies its *content.*

[35] Cn denotes the bivalent logical consequence operation. See Augusto (2020c).

```
bird(penguin, toto).
bird(ostrich, sheila).
bird(emu, tom).
bird(turkey, sam).
bird(turkey, sandra).
bird(hen, lolita).
bird(canary, roberto).
bird(nightingale, sarita).
bird(crow, bob).
bird(woodpecker, lola).
bird(duck, cassandra).
bird(duck, samantha).

quarantined(roberto).
quarantined(bob).

eats(penguin, fish).
eats(ostrich, all).
eats(emu, all).
eats(turkey, seeds).
eats(hen, all).
eats(canary, seeds).
eats(nightingale, seeds).
eats(crow, all).
eats(woodpecker, bugs).
eats(duck, all).

sick(roberto).
sick(bob).

on_diet(roberto, seeds).
on_diet(bob, all).
```

Figure 14: An instance of the Datalog database *Avian_Center_DDB* with respect to the program *Avian_Sick_Prog*.

This gives us the reduced instance of Figure 15, which in fact corresponds to a minimal Herbrand model for this program. In effect, we have, for A abbreviating *Avian_Sick_Prog* and AC doing so for the associated EDB *Avian_Center_EDB*, and for the goals above G_1 and G_2,

$$A \cup AC \models \begin{cases} \texttt{sick}\,(\texttt{roberto}) \\ \\ \texttt{sick}\,(\texttt{bob}) \end{cases}$$

and

$$A \cup AC \models \begin{cases} \texttt{on\_diet}\,(\texttt{roberto}, \texttt{seeds}) \\ \\ \texttt{on\_diet}\,(\texttt{bob}, \texttt{all}) \end{cases}.$$

```
bird(canary, roberto).
bird(crow, bob).

quarantined(roberto).
quarantined(bob).

eats(canary, seeds).
eats(crow, all).

sick(roberto).
sick(bob).

on_diet(roberto, seeds).
on_diet(bob, all).
```

Figure 15: $Cn(\mathit{Avian\_Sick\_Prog} \cup E_{\mathit{Avian\_Center\_DDB}})$.

In particular, we have

$$(E_{AC})_g = \left\{ \begin{array}{c} \mathit{bird}\,(\mathit{canary}, \mathit{roberto})\,, \\ \mathit{bird}\,(\mathit{crow}, \mathit{bob})\,, \\ \mathit{quarantined}\,(\mathit{roberto})\,, \\ \mathit{quarantined}\,(\mathit{bob})\,, \\ \mathit{eats}\,(\mathit{canary}, \mathit{seeds})\,, \\ \mathit{eats}\,(\mathit{crow}, \mathit{all}) \end{array} \right\}$$

and

$$g_{\Delta}\,(E_{AC}) = \left\{ \begin{array}{c} \mathit{bird}\,(\mathit{canary}, \mathit{roberto})\,, \\ \mathit{bird}\,(\mathit{crow}, \mathit{bob})\,, \\ \mathit{quarantined}\,(\mathit{roberto})\,, \\ \mathit{quarantined}\,(\mathit{bob})\,, \\ \mathit{eats}\,(\mathit{canary}, \mathit{seeds})\,, \\ \mathit{eats}\,(\mathit{crow}, \mathit{all})\,, \\ \mathit{sick}\,(\mathit{roberto})\,, \\ \mathit{sick}\,(\mathit{bob})\,, \\ \mathit{on\_diet}\,(\mathit{roberto}, \mathit{seeds})\,, \\ \mathit{on\_diet}\,(\mathit{bob}, \mathit{all}) \end{array} \right\}.$$

In fact, $g_\Delta\left(E_{AC}\right)$ can be further reduced to a subset $g_\Delta\left(E^{'}_{AC}\right)$ if we consider a sub-domain; for instance, let us focus on Bob, so that we have

$$\mathscr{D}^{''}_{A} = \{crow, bob, all\}$$

and

$$g_\Delta\left(E^{'}_{AC}\right) = \left\{\begin{array}{c} bird\left(crow, bob\right), \\ quarantined\left(bob\right), \\ eats\left(crow, all\right), \\ sick\left(bob\right), \\ on\_diet\left(bob, all\right) \end{array}\right\}.$$

Consider now Definition 109: In it, $\mathit{\Pi}_\Delta \cup E_\Delta \models p\left(c^i_1, ..., c^i_n\right)$ can be abbreviated as $\mathit{\Pi}_\Delta \models p\left(c^i_1, ..., c^i_n\right)$ if we conceive $\mathit{\Pi}_\Delta = E_\Delta \cup IDB_\Delta$. This we do if we conceive a Datalog program as a definite program. Then, we can further specify

$$\mathit{\Pi}_\Delta \models_{Sch(\mathit{\Pi}_\Delta)_g} p\left(c^i_1, ..., c^i_n\right)$$

so that we actually have

$$A \cup AC \models_{Sch(A)_g} \begin{cases} \texttt{sick}\left(\texttt{roberto}\right) \\ \\ \texttt{sick}\left(\texttt{bob}\right) \end{cases}$$

and

$$A \cup AC \models_{Sch(A)_g} \begin{cases} \texttt{on\_diet}\left(\texttt{roberto}, \texttt{seeds}\right) \\ \\ \texttt{on\_diet}\left(\texttt{bob}, \texttt{all}\right) \end{cases}.$$

In effect, logical consequence with respect to a Datalog program $\mathit{\Pi}_\Delta$ is defined here in terms of the Herbrand interpretations therefor. Because there are no function symbols in Datalog, the Herbrand universe $H_{\mathit{\Pi}_\Delta}$ is finite. Also, because the domains are specified in the associated relational DB, the constants in $H\mathcal{I}_{\mathit{\Pi}_\Delta}$ are all the constants occurring in $EDB\left(\mathit{\Pi}_\Delta\right)_g$. Finally, as there are no (explicitly) negative literals in our Datalog DDB, there is actually only one H-interpretation $H\mathcal{I}_{\mathit{\Pi}_\Delta}$ for a Datalog program if all the predicate symbols of $Sch\left(\mathit{\Pi}_\Delta\right)$ are included, i.e. if $H\mathcal{I}_{\mathit{\Pi}_\Delta} = \bigcup_i H\mathcal{I}^i_{\mathit{\Pi}_\Delta}$. Then, we have the following result:

Proposition 111. *Given a Datalog program $\mathit{\Pi}_\Delta$, there is a least H-model*

$$\underline{H}\mathcal{M}_{\mathit{\Pi}_\Delta} =$$
$$\left\{p\left(c^i_1, ..., c^i_n\right) \mid p\left(c^i_1, ..., c^i_n\right) \in Sch\left(\mathit{\Pi}_\Delta\right)_g \text{ and } \left(\mathit{\Pi}_\Delta \cup E_\Delta\right) \models p\left(c^i_1, ..., c^i_n\right)\right\}$$

such that

$$\underline{H}\mathcal{M}_{\mathit{\Pi}_\Delta} = g_\Delta\left(E_\Delta\right).$$

Proof. Left as an exercise. (Hint: Note how, in Example 110, $g_\Delta\left(E^{'}_{AC}\right)$ is a minimal H-model, but *not* the least H-model for AC; $g_\Delta\left(E_{AC}\right)$ is.) □

In other words – and informally –, $g_\Delta\left(E_\Delta\right)$ provides *all* the information, and *only* the information, expressed by a Datalog program $\mathit{\Pi}_\Delta$. In terms of Herbrand semantics, this just is the least H-model $\underline{H}\mathcal{M}_{\mathit{\Pi}_\Delta}$.

4.3.2 Fixed-point semantics

We obtain an equivalent result in the following way: If working with fixed-point semantics for a DDB, the immediate consequence operator takes over the role of Cn and $\models$. We denote this operator by $\mathbf{T}$, and define it as follows:

Definition 112. Let $I \subseteq Sch\left(\mathit{\Pi}_{\Delta}\right)_{g}$ be a ground instance of a Datalog program $\mathit{\Pi}_{\Delta}$. Then, the *immediate consequence operator* is the mapping

$$\mathbf{T}_{\mathit{\Pi}} : 2^{I} \longrightarrow 2^{I}$$

where $\mathbf{T}_{\mathit{\Pi}}$ is a simplified notation for $\mathbf{T}_{\mathit{\Pi}_{\Delta}}$. For a Datalog program $\mathit{\Pi}_{\Delta}$ and $I \subseteq Sch\left(\mathit{\Pi}_{\Delta}\right)_{g}$, we define

$$\mathbf{T}_{\mathit{\Pi}}\left(I\right) = \left\{\sigma A \middle| \left(A \leftarrow B_{1}, ..., B_{n}\right) \in \mathit{\Pi}_{\Delta} \text{ and } \sigma\left(B_{1}\right), ..., \sigma\left(B_{n}\right) \in I\right\}$$

for some ground substitution σ.[36]

1. We set
$$\mathbf{T}_{\mathit{\Pi}}^{0}\left(I\right) := I$$
after which we iterate
$$\begin{aligned}\mathbf{T}_{\mathit{\Pi}}^{1}\left(I\right) &:= \mathbf{T}_{\mathit{\Pi}}\left(I\right)\\ \mathbf{T}_{\mathit{\Pi}}^{2}\left(I\right) &:= \mathbf{T}_{\mathit{\Pi}}\left(\mathbf{T}_{\mathit{\Pi}}^{1}\left(I\right)\right)\\ &\vdots\\ \mathbf{T}_{\mathit{\Pi}}^{n+1}\left(I\right) &:= \mathbf{T}_{\mathit{\Pi}}\left(\mathbf{T}_{\mathit{\Pi}}^{n}\left(I\right)\right)\end{aligned}$$
until no more ground atoms can be output, i.e. until
$$\mathbf{T}_{\mathit{\Pi}}\left(\mathbf{T}_{\mathit{\Pi}}^{n}\left(I\right)\right) = \mathbf{T}_{\mathit{\Pi}}^{n}\left(I\right)$$
and $\mathbf{T}_{\mathit{\Pi}}^{n}\left(I\right)$ is a *fixed point* of $\mathit{\Pi}_{\Delta}$.

2. Let now
$$\begin{aligned}\mathbf{T}_{\mathit{\Pi}} \uparrow^{0}\left(\emptyset\right) &:= \emptyset\\ \mathbf{T}_{\mathit{\Pi}} \uparrow^{1}\left(\emptyset\right) &:= \mathbf{T}_{\mathit{\Pi}}\left(\mathbf{T}_{\mathit{\Pi}} \uparrow^{0}\left(\emptyset\right)\right) = \mathbf{T}_{\mathit{\Pi}}^{0}\left(\emptyset\right)\\ \mathbf{T}_{\mathit{\Pi}} \uparrow^{2}\left(\emptyset\right) &:= \mathbf{T}_{\mathit{\Pi}}\left(\mathbf{T}_{\mathit{\Pi}} \uparrow^{1}\left(\emptyset\right)\right) = \mathbf{T}_{\mathit{\Pi}}^{1}\left(\emptyset\right)\\ &\vdots\\ \mathbf{T}_{\mathit{\Pi}} \uparrow^{n+1}\left(\emptyset\right) &:= \mathbf{T}_{\mathit{\Pi}}\left(\mathbf{T}_{\mathit{\Pi}} \uparrow^{n}\left(\emptyset\right)\right) = \mathbf{T}_{\mathit{\Pi}}^{n}\left(\emptyset\right)\end{aligned}$$
such that
$$\mathbf{T}_{\mathit{\Pi}} \uparrow^{\omega}\left(\emptyset\right) := \bigcup_{n\in\mathbb{N}}^{\infty} \mathbf{T}_{\mathit{\Pi}} \uparrow^{n}\left(\emptyset\right) = \lim_{n\rightarrow\infty} \mathbf{T}_{\mathit{\Pi}} \uparrow^{n}\left(\emptyset\right)$$
so that for finite n we have $\mathbf{T}_{\mathit{\Pi}} \uparrow^{\omega}\left(\emptyset\right) = \mathbf{T}_{\mathit{\Pi}} \uparrow^{n}\left(\emptyset\right)$ and we say that $\mathbf{T}_{\mathit{\Pi}} \uparrow^{\omega}\left(\emptyset\right)$ is the *least fixed point* of $\mathit{\Pi}_{\Delta}$, denoted by $lfp\left(\mathit{\Pi}_{\mathit{\Pi}}\right)$.

[36]More correctly, a *matching* σ (see the next Section).

Example 113. Consider the following Datalog program as constituted by the following set of logical formulae:

$$\mathit{\Pi}_{\Delta} = \left\{ \begin{array}{c} p\left(a\right) \\ q\left(X\right) \leftarrow p\left(X\right) \\ r\left(X\right) \leftarrow q\left(X\right) \\ s\left(X\right) \leftarrow r\left(X\right), q\left(X\right) \end{array} \right\}$$

We start with $\mathbf{T}_{\mathit{\Pi}} \uparrow^{0} (\emptyset) = \emptyset$, and we then iterate until a least fixed-point is found:

$$\mathbf{T}_{\mathit{\Pi}} \uparrow^{1} (\emptyset) = \mathbf{T}_{\mathit{\Pi}} (\emptyset) = \{p\left(a\right)\}$$

$$\mathbf{T}_{\mathit{\Pi}} \uparrow^{2} (\emptyset) = \mathbf{T}_{\mathit{\Pi}} (\{p\left(a\right)\}) = \{p\left(a\right), q\left(a\right)\}$$

$$\mathbf{T}_{\mathit{\Pi}} \uparrow^{3} (\emptyset) = \mathbf{T}_{\mathit{\Pi}} (\{p\left(a\right), q\left(a\right)\}) = \{p\left(a\right), q\left(a\right), r\left(a\right)\}$$

$$\mathbf{T}_{\mathit{\Pi}} \uparrow^{4} (\emptyset) = \mathbf{T}_{\mathit{\Pi}} (\{p\left(a\right), q\left(a\right), r\left(a\right)\}) = \{p\left(a\right), q\left(a\right), r\left(a\right), s\left(a\right)\}$$

$$\mathbf{T}_{\mathit{\Pi}} \uparrow^{5} (\emptyset) = \mathbf{T}_{\mathit{\Pi}} (\{p\left(a\right), q\left(a\right), r\left(a\right), s\left(a\right)\}) = \{p\left(a\right), q\left(a\right), r\left(a\right), s\left(a\right)\}$$

and

$$\mathbf{T}_{\mathit{\Pi}} \uparrow^{4} (\emptyset) = lfp\left(\mathit{\Pi}_{\Delta}\right).$$

$\mathbf{T}_{\mathit{\Pi}} \uparrow^{4} (\emptyset)$ is the least fixed-point of $\mathit{\Pi}_{\Delta}$.

The equivalence of Herbrand semantics and fixed-point semantics with respect to a Datalog program is expressed in the following theorem:

Theorem 114. *(van Emden & Kowalski, 1976) Let $\mathit{\Pi}_{\Delta}$ be a set of definite clauses. Then:*

$$\underline{H}\mathcal{M}_{\mathit{\Pi}_{\Delta}} = lfp\left(\mathit{\Pi}_{\Delta}\right) = \mathbf{T}_{\mathit{\Pi}} \uparrow^{\omega} (\emptyset)$$

4.4 A proof system for Datalog definite programs: SLD resolution

It should be obvious that each set in $g_{\Delta}(G)$ for some goal G corresponds to a ground instantiation $G\sigma$ where σ is a substitution $\{X \mapsto c\}$.

Example 115. In Example 110, $g_{\Delta}(G_1)$ is obtained from the ground instantiations

$$\texttt{sick}\left(\texttt{roberto}\right) : -\texttt{quarantined}\left(\texttt{roberto}\right).$$

given the substitution $\sigma_1 = \{\texttt{Y} \mapsto \texttt{roberto}\}$, and

$$\texttt{sick}\left(\texttt{bob}\right) : -\texttt{quarantined}\left(\texttt{bob}\right).$$

given the substitution $\theta_1 = \{\texttt{Y} \mapsto \texttt{bob}\}$. As for $g_{\Delta}(G_2)$, we have the ground instantiations

$$\texttt{on\_diet}\left(\texttt{roberto}, \texttt{seeds}\right) : -\texttt{bird}\left(\texttt{canary}, \texttt{roberto}\right), \\ \texttt{sick(roberto)}, \texttt{eats}\left(\texttt{canary}, \texttt{seeds}\right).$$

given the substitution $\sigma_2 = \{\mathtt{Y} \mapsto \mathtt{roberto}, \mathtt{Z} \mapsto \mathtt{seeds}, \mathtt{X} \mapsto \mathtt{canary}\}$, and

$$\mathtt{on\_diet}\,(\mathtt{bob}, \mathtt{all}) : -\mathtt{bird}\,(\mathtt{crow}, \mathtt{bob})\,,$$

$$\mathtt{sick(bob)}, \mathtt{eats}\,(\mathtt{crow}, \mathtt{all})\,.$$

given the substitution $\theta_2 = \{\mathtt{Y} \mapsto \mathtt{bob}, \mathtt{Z} \mapsto \mathtt{all}, \mathtt{X} \mapsto \mathtt{crow}\}$.

The process of ground-instantiating a goal in a Datalog DB is by means of unification. However, we can make this process more precise with respect to a Datalog DB $\Delta = E_\Delta \cup \Pi_\Delta$ in the following way:

Definition 116. Let $E_\Delta = \{A_1, ..., A_n\}$, where each A_i is a ground assertion or ground fact. Then, given some rule $(A \leftarrow B_1, ..., B_k) \in \Pi_\Delta$, we say that B_j *matches* some A_i if there is a substitution σ such that $B_j\sigma = A_i$ for $0 < j \leq k, 0 < i \leq n$.

Definition 117. Given a Datalog DB $\Delta = E_\Delta \cup \Pi_\Delta$, we can *infer* a (new) fact A from $E_\Delta = \{A_1, ..., A_n\}$ and a rule $(A \leftarrow B_1, ..., B_k) \in \Pi_\Delta$ if there is a substitution σ such that $B_j\sigma = A_i$ is a matching for $0 < j \leq k, 0 < i \leq n$.

1. We call this inference rule *universal modus ponens* (UMP) and denote this inference by $\Delta \vdash_{UMP} A$.

2. The process of obtaining the inference $\Delta \vdash A$ by a finite number of applications of UMP is called *reduction*. We specify this inference by writing $\Delta \vdash_{red} A$.

Compare Definition 117 with Definitions 37-8 and 44 for Prolog: UMP and reduction are essentially the same rule and process as defined for (pure) Prolog, the single difference being in the fact that we now speak of matching as a special form of unification, namely a form thereof that does not involve function symbols and their terms. Basically, we say that given some substitution σ some IDB fact matches an EDB fact. This conceptual equivalence holds for the notion of a proof tree (cf. Def. 45), which, given the notion of matching, is more interesting in the context of Datalog, as every leaf thereof is an EDB fact. An example of a Datalog proof tree is given in Figure 16.

Example 118. Figure 16 shows the proof tree for the IDB fact $\mathtt{on\_diet}\,(\mathtt{bob}, \mathtt{all})$. obtained from the program *Avian_Sick_Prog* as applied over the EDB *Avian_Center_EDB*. The bold-line ellipses are facts in the EDB *Avian_Center_EDB* (cf. Fig. 13). $\sigma = \{\mathtt{Y} \mapsto \mathtt{bob}\}$ and $\theta = \{\mathtt{X} \mapsto \mathtt{crow}, \mathtt{Z} \mapsto \mathtt{all}\}$ are the substitutions employed. Note in this proof tree that the goal – the IDB fact – is the root and each EDB fact is a leaf.

Proposition 119. *The pair* $(\Delta, \vdash_{red}) = \mathcal{P}_\Delta$ *constitutes a proof system for a Datalog DB* Δ.

Proof. Let Δ be a set of clauses and let A be a ground fact. Then, either $A \in \Delta$, in which case we have $\Delta \vdash A$ by the very definition of the logical consequence relation (namely by R1; cf. Def. 1), or A can be inferred by repeated applications of UMP to a rule $r \in \Delta$ and a set of ground facts $\{B_1, ..., B_k\} \subset \Delta$, so that we have $\Delta \vdash_{red} A$. □

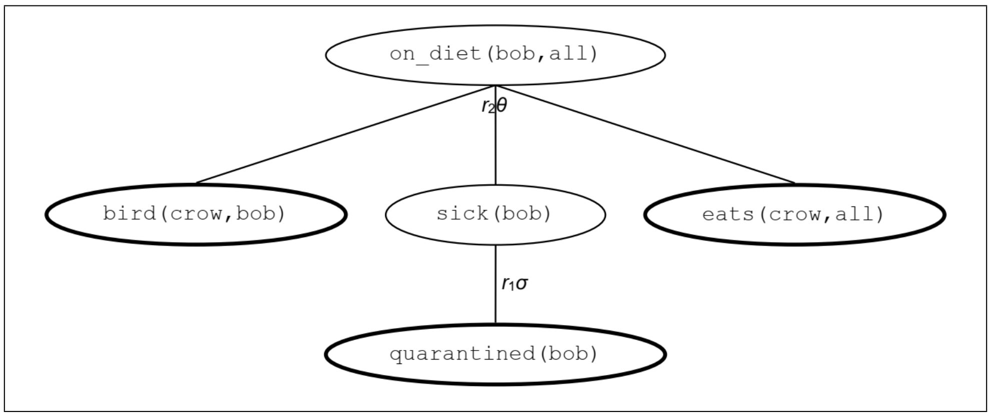


Figure 16: A Datalog proof tree.

Just as in the case of Prolog, the LP notion of reduction corresponds to a proof by *resolution.* In effect, note that in Proposition 119 there is no mention to a distinction between the ground assertions in EDB and the rules in IDB: We now see Δ as a single set of clauses whose elements are both unit clauses (facts) and definite clauses (rules), it being the case that these clauses respect the safety conditions of Definition 97. In other words, Δ is a set of Datalog *formulae.*

Recall that

$$\mathsf{Datalog} \subseteq \mathsf{L1}_{\mathit{ff}} \subset \mathsf{Prolog}$$

which means that Datalog can be seen as a function-free "dialect" of Prolog. Thus, besides the fact that all the facts in Δ are ground literals, a Datalog database Δ distinguishes itself from a Prolog database $\mathit{\Delta}$ in that no predicate has functions as arguments. Hence, just as in Prolog, in Datalog we have a database Δ as a basis for a *definite program* Π.

Example 120. Figure 17 shows the Datalog definite program *Avian_Center_Quarantine*.

With respect to the program of Example 120, it must be remarked that the order of the facts and rules is wholly irrelevant, as Datalog, contrarily to Prolog, satisfies the property of commutativity with respect to both $\wedge$ and $\vee$. This means that the Prolog operator ! for *cut* is not a Datalog operator, the same holding for the Prolog operator `fail`.

The importance of having a Datalog definite program is that, just as in the case of Prolog, SLD-resolution provides a complete proof system for it.[37]

Example 121. Figure 18 shows the SLD-resolution proof for $? -$ `on_diet(bob, all).` given $\sigma = \{\mathtt{Y} \mapsto \mathtt{bob}, \mathtt{Z} \mapsto \mathtt{all}\}$, $\theta = \{\mathtt{X} \mapsto \mathtt{crow}\}$, and $\lambda = \{\mathtt{Y}_1 \mapsto \mathtt{bob}\}$.

Because conjunction and disjunction are commutative operators in Datalog, we are assured that, if there is a resolution proof of a query, then there is an SLD-resolution

[37]Note, however, that for Datalog a breadth-first search is more adequate than the depth-first search characteristic of Prolog. Indeed, neither the order of the rules nor that of the goals affect the evaluation of a Datalog program.

```
bird (penguin, toto) .
bird (ostrich, sheila) .
bird (emu, tom) .
bird (turkey, sam) .
bird (turkey, sandra) .
bird (hen, lolita) .
bird (canary, roberto) .
bird (nightingale, sarita) .
bird (crow, bob) .
bird (woodpecker, lola) .
bird (duck, cassandra) .
bird (duck, samantha) .
abnormal (penguin) .
abnormal (ostrich) .
abnormal (emu) .
abnormal (turkey) .
abnormal (hen) .
quarantined (roberto) .
quarantined (bob) .
eats (penguin, fish) .
eats (ostrich, all) .
eats (emu, all) .
eats (turkey, seeds) .
eats (hen, all) .
eats (canary, seeds) .
eats (nightingale, seeds) .
eats (crow, all) .
eats (woodpecker, bugs) .
eats (duck, all) .
sick (Y) : −quarantined (Y) .
on_diet (Y, Z) : −bird (X, Y) , sick (Y) , eats (X, Z) .
```

Figure 17: Datalog definite program *Avian_center_Quarantine*.

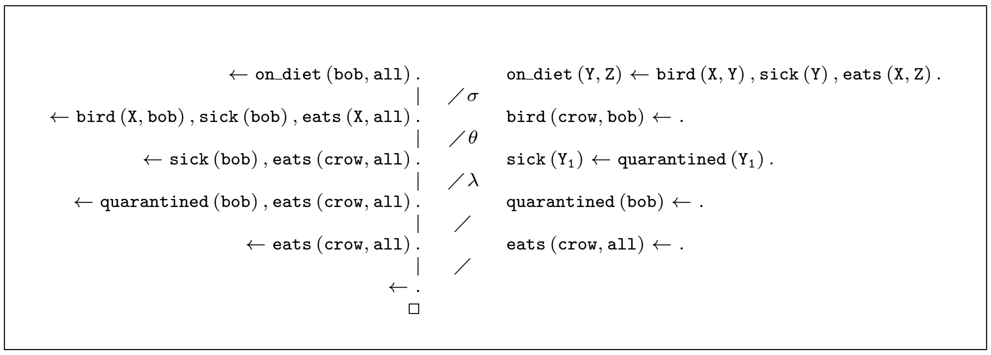

← on_diet (bob, all) .
on_diet (Y, Z) ← bird (X, Y) , sick (Y) , eats (X, Z) .
σ
← bird (X, bob) , sick (bob) , eats (X, all) .
bird (crow, bob) ← .
θ
← sick (bob) , eats (crow, all) .
sick (Y₁) ← quarantined (Y₁) .
λ
← quarantined (bob) , eats (crow, all) .
quarantined (bob) ← .
← eats (crow, all) .
eats (crow, all) ← .
← .
□

Figure 18: An SLD-resolution proof of a Datalog query.

proof of it. (I leave the proofs of soundness and completeness as exercises.) This means that a Datalog DDB can be seen as a theory, and we can check for goals by means of proving theorems (for instance, by using Prover9-Mace4).[38] In the proof above, we simply moved the goal `quarantined (bob)` to the leftmost position in the goal clause, but we could actually have placed the goal `sick (Y)` at the rightmost position in the body of rule r_2. However, contrarily to Prolog, to which resolution is so to say inherent, for Datalog resolution as a proof calculus is a matter of adopting a *top-down* vs. a *bottom-up evaluation technique*, where by "evaluation" I mean the process (the algorithm) and/or its implementation (e.g., a proof tree) of finding an answer to a query given a DDB.

Example 122. Figure 16 shows the top-down evaluation of the query $?-$ `on_diet (Y, Z)` . for the matching $\theta = \{\texttt{Y} = \texttt{bob}, \texttt{Z} = \texttt{all}\}$. We say that the query $?-$ `on_diet (Y, Z)` . (given θ) *evaluates to* the reply `on_diet (bob, all)` .. In turn, in this evaluation we say that $r_1\sigma$ was evaluated before $r_2\theta$, so that the term "evaluation" applies also to every step of the algorithm at hand. Actually, given a rule $A \leftarrow B_1, ..., B_n$, we speak also of the evaluation of each of $A, B_1, ..., B_n$. Thus, in the case at hand, the sub-goal $?-$ `sick (Y)` . in the body of r_2 was evaluated before the other sub-goals in this rule.

4.5 Datalog with negation: Datalog$^\neg$

Just as in the case of Prolog, one may simply accept CWA as a meta-rule (cf. Def. 63), and take NF to be an implicit rule of a Datalog DDB; just as in Prolog, this entails non-monotonicity with respect to a Datalog DDB. However, there are circumstances that call for *explicit* negation in a Datalog DDB. For instance, given a program for some electronic device (e.g., a printer), a rule such as

$$ready\,(X) \leftarrow device\,(X)\,, \neg busy\,(X)$$

may be required for the correct functioning of the device. Yet another illustration: In the program for a game, the following rule expresses the fact that one wins when one forces the opponent to a situation in which they have no chance to move:

[38] This is only possible for small EDBs. Note that a Datalog DDB may be implemented as a Prolog program – again, solely for small EDBs – only if one is aware of the non-commutativity of conjunction and disjunction in the latter.

$$win\,(X) \leftarrow move\,(X, Y)\,, \neg win\,(Y)$$

From the viewpoint of programming, we say that negation increases the *expressiveness* of the language. This is especially relevant when computing over finite domains, which is typically the case of a Datalog DDB. Datalog$^{\neg}$ is a more expressive extension of Datalog; however, as we shall see, this increased expressiveness comes at the cost of syntactic restrictions.

Definition 123. A Datalog rule $\mathbf{r}$ of the form

$$A \leftarrow B_1, ..., B_n$$

where each $B_i \in Body_{\mathbf{r}}$ is a positive or a negative literal, i.e. $B_i = p\,(c_1, ..., c_k)$ or $B_i = \neg p\,(c_1, ..., c_k)$, respectively, constitutes an extension of Datalog called *Datalog with negation.* I abbreviate it as, or denote it by, Datalog$^{\neg}$.

Just as in the case of Datalog, safety conditions apply:

Definition 124. Let $\mathbf{r}$ be a rule in a Datalog$^{\neg}$ program, denoted by $\Pi_{\Delta}^{\neg}$. Then, $\mathbf{r}$ is said to be a *safe rule* if

1. negation does not occur in the head of $\mathbf{r}$, and

2. every variable occurring in a negative literal must also occur in a positive literal.

As seen above, one of the advantages of Datalog DDBs over relational DBs is the fact that the former allow for recursion; in the presence of negation, however, we might have *recursion through negation*, i.e. predicates being defined recursively in terms of their own negation. An example of this is the rule $p \leftarrow \neg p$, which can be read as "p is provable if *not-p* is provable (or p is not provable)," which is clearly a contradiction.[39]

It so happens that neither Herbrand semantics nor fixed-point semantics are adequate for Datalog$^{\neg}$. On the other hand, several other semantics can take over in Datalog$^{\neg}$, with greater or lesser success. I chose to discuss here the so-called *stratified semantics*, firstly elaborated on in van Gelder (1986) for general logic programs and in Apt et al. (1988). In effect, even if not all meaningful Datalog$^{\neg}$ programs are stratified, the handling of negation in this semantics is highly adequate in my view, namely from the viewpoint of deduction. Moreover, it is an extension of fixed-point semantics, already studied above. It is useful to consider stratified semantics as a natural extension of semi-positive Datalog$^{\neg}$:

Definition 125. We say that a Datalog$^{\neg}$ program $\Pi_{\Delta}^{\neg}$ is *semi-positive* if, whenever $\neg p\,(\vec{x}) \in Body_{\mathbf{r}}$ for a rule $\mathbf{r} \in \Pi_{\Delta}^{\neg}$, then $p \in EDB\,(\Pi_{\Delta}^{\neg})$.

This means that, given some relation $R\,(t_1, ..., t_n)$ in a Datalog DDB, $\neg R\,(t_1, ..., t_n)$ is *true* iff $t_i \in \mathscr{D}_i$ for all $i = 1, ..., n$ and $\{t_i\}_{i>0}^{n} \notin R$, so that we have

$$\left(\Pi_{\Delta}^{\neg} \cup EDB\right) \nvDash R\,(t_1, ..., t_n)\,.$$

[39]Note how this collides with the classical equivalence $p \leftarrow \neg p \equiv \neg\neg p \vee p \equiv p$.

But if $\{t_i\}_{i>0}^{n} \notin R$, then by transitive closure there is some relation R' such that $\{t_i\}_{i>0}^{n} \in R'$. This, in turn, means that for any negated relation $\neg R(t_1, ..., t_n)$ we can compute its *complement* $\overline{R}(t_1, ..., t_n)$ such that

$$(\mathit{\Pi}_\Delta^\neg \cup EDB) \models \overline{R}(t_1, ..., t_n)$$

and thus obtain $\overline{EDB}$. In other words, given some predicate $p' \in EDB(\mathit{\Pi}_\Delta^\neg)$, we can replace all occurrences of $\neg p'$ by the *new* predicate $\overline{p'} \in EDB(\mathit{\Pi}_\Delta^\neg)$, thus eliminating negation and obtaining a program that is basically a (positive) Datalog program $\mathit{\Pi}_\Delta$.

Example 126. Given $EDB = \{R(a,b), R(b,c), R(a,a), R(a,c)\}$, we consider the semi-positive Datalog$^\neg$ IDB

$$IDB = \left\{ \begin{array}{c} r'(x,y) \leftarrow v(x), \neg r'(x,y) \\ t(x,y) \leftarrow r'(x,y) \\ t(x,y) \leftarrow t(x,z), r'(z,y) \end{array} \right\}.$$

We replace $\neg r'$ by $\overline{r'}$, obtaining the "positive" IDB

$$IDB = \left\{ \begin{array}{c} r'(x,y) \leftarrow v(x), \overline{r'}(x,y) \\ t(x,y) \leftarrow r'(x,y) \\ t(x,y) \leftarrow t(x,z), r'(z,y) \end{array} \right\}.$$

The obtained program can be run as a (positive) Datalog program and we can compute

$$\overline{EDB} = \{R'(b,a), R'(b,b), R'(c,c), R'(c,b), R'(c,a)\}.$$

Let now the notation $J|EDB(\mathit{\Pi}_\Delta^\neg)$ denote the *restriction* of instance J to $EDB(\mathit{\Pi}_\Delta^\neg)$. Then we have the following result:

Theorem 127. *Let $\mathit{\Pi}_\Delta^\neg$ be a semi-positive Datalog$^\neg$ program. Then, for every instance I over $EDB(\mathit{\Pi}_\Delta^\neg)$, there exists a least H-model / a least fixed-point satisfying $J|EDB(\mathit{\Pi}_\Delta^\neg)$ such that*

$$\underline{H}\mathcal{M}_{\mathit{\Pi}_\Delta^\neg} = lfp(\mathit{\Pi}_\Delta^\neg) = \lim_{i\to\infty} \left\{ \mathbf{T}^i_{\mathit{\Pi}_\Delta^\neg}(I) \right\}_{i>0}.$$

Let now there be given a Datalog$^\neg$ program $\mathit{\Pi}_\Delta^\neg$ such that there is some rule $\mathbf{r} \in \mathit{\Pi}_\Delta^\neg$ and $\neg R(\vec{t}) \in Body_{\mathbf{r}}$, where $\vec{t}$ abbreviates the sequence $\{t_i\}_{i=1}^{n}$ for some finite n. Intuitively, we are interested in knowing the value of $R(\vec{t})$ in order to evaluate $Body_{\mathbf{r}}$: If $R(\vec{t})$ is *false*, then $\neg R(\vec{t})$ is *true*; otherwise, if $R(\vec{t})$ is *true*, then $\neg R(\vec{t})$ is *false* and $\mathbf{r}$ is not applicable. Thus, the first evaluation falls on $R(\vec{t})$. If rule $\mathbf{r}$ is applicable, we can then move to the stratification of $\mathit{\Pi}_\Delta^\neg$, for whose definition a few previous notions are required.

Definition 128. Let $\vec{\mathfrak{G}} = \left(V, E, \vec{f}\right)$ be a directed graph such that, given a Datalog$^\neg$ program $\mathit{\Pi}_\Delta^\neg$, we have

$$V = Sch(\mathit{\Pi}_\Delta^\neg)$$

and for some rules $\mathbf{r}_i, \mathbf{r}_j \in (\mathit{\Pi}_\Delta^\neg \cup E_\Delta)$,

$$E = \left\{(R,S) \,|\, Head_{\mathbf{r}_i} = R \text{ and } S \in Body_{\mathbf{r}_j}\right\}$$

where for any pair of relations (R,S) there is at most one arc $R \longrightarrow S$. We call this the *dependency graph* of $\mathit{\Pi}_\Delta^\neg$ and denote it by $\vec{\mathfrak{G}}_{\mathit{\Pi}_\Delta^\neg}$ (abbreviated: $\vec{\mathfrak{G}}_{\mathit{\Pi}}$).

1. An arc (R, S) such that $Head_{\mathbf{r}_i} = R$ and $\neg S \in Body_{\mathbf{r}_j}$ is said to be a *negative* arc; otherwise it is called a *positive* arc.

Given a dependency graph $\vec{\mathfrak{G}}_{\Pi}$, for the sake of graphical convenience for each negative arc we draw an arc of the form $R \overset{\neg}{\longrightarrow} S$ in $\vec{\mathfrak{G}}_{\Pi}$.

Example 129. Let there be given the following Datalog$^{\neg}$ program $\Pi^{\neg}_{\Delta_1}$ (abbr.: $\Pi^{\neg}_1$):

$$\Pi^{\neg}_1 = \left\{ \begin{array}{c} p(X,Y) \leftarrow q(X,Y) \\ p(X,Y) \leftarrow q(X,Z), p(Z,Y) \\ r(X,Y) \leftarrow s(X,Y), \neg p(X,Y) \\ t(X,Y) \leftarrow r(X,Y) \\ t(X,Y) \leftarrow r(X,Z), t(Z,Y) \end{array} \right\}$$

Figure 19 shows the dependency graph of $\Pi^{\neg}_1$.

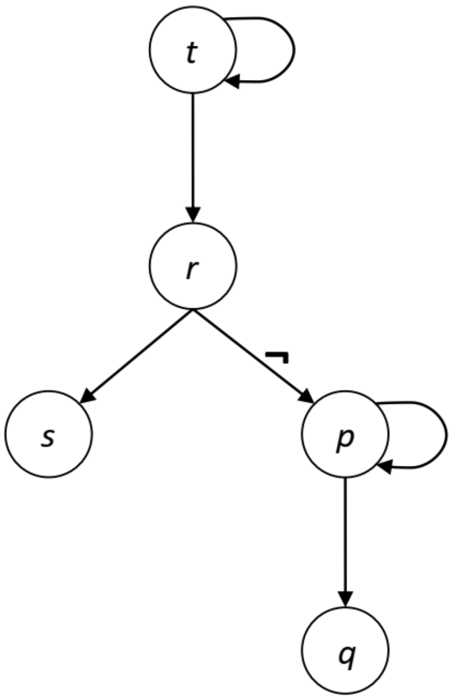


Figure 19: Dependency graph $\vec{\mathfrak{G}}_{\Pi^{\neg}_1}$ of the Datalog$^{\neg}$ program $\Pi^{\neg}_1$.

Proposition 130. *Given some dependency graph $\vec{\mathfrak{G}}_{\Pi}$, if*

$$((R = R_1) \longrightarrow R_2 \longrightarrow ... \longrightarrow R_{k-1} \longrightarrow (R_k = S)) \in \vec{\mathfrak{G}}_{\Pi}$$

such that some $R_i \longrightarrow R_{i+1}$ for $1 \leq i \leq k-1$ is a negative arc, then S must be evaluated prior to R.

Proposition 130 is called the *stratification principle.* In order to elaborate on this principle a few notions are required.

Definition 131. Consider the set $Sch\left(\Pi_{\Delta}^{\neg}\right)$ of predicate symbols of a Datalog$^{\neg}$ program $\Pi_{\Delta}^{\neg}$ and the function

$$\ell : Sch\left(\Pi_{\Delta}^{\neg}\right) \longrightarrow \mathbb{N}$$

such that $\ell\left(p\right)=0$ if p is a leaf in $\vec{\mathfrak{G}}_{\Pi}$ and $\ell\left(p\right)=i$ for $1\leq i\leq n$ if there is a path ε with $1,2,...,n$ arcs in $\vec{\mathfrak{G}}_{\Pi}$ originating in p and ending in some leaf q. We call $\ell\left(p\right)$ the *length* of predicate p and the corresponding set

$$\mathscr{R}\left(Sch\left(\Pi_{\Delta}^{\neg}\right)\right)_{i}=\left\{p\,|\,p\in Sch\left(\Pi_{\Delta}^{\neg}\right)\text{ and }\ell\left(p\right)=i\in\left[0,n\right]\right\}$$

is called the i-th *rank* of $Sch\left(\Pi_{\Delta}^{\neg}\right)$.

We abbreviate $\mathscr{R}\left(Sch\left(\Pi_{\Delta}^{\neg}\right)\right)_{i}$ simply as $\mathscr{R}_i$. It should be evident that rank $\mathscr{R}_0$ contains the predicates in $E_{\Delta}\subseteq EDB\left(\Pi_{\Delta}^{\neg}\right)$.

Example 132. The ranking of the predicates of $Sch\left(\Pi_{1}^{\neg}\right)$ (cf. Example 129) is as follows:

$$\begin{gathered}\mathscr{R}_0=\{q,s\}\\ \mathscr{R}_1=\{p\}\\ \mathscr{R}_2=\{r\}\\ \mathscr{R}_3=\{t\}\end{gathered}$$

Definition 133. A Datalog$^{\neg}$ program $\Pi_{\Delta}^{\neg}$ is said to be *stratified* if there is a partition

$$\mathscr{P}_{Sch\left(\Pi_{\Delta}^{\neg}\right)}=\bigcup_{i\geq 0}^{n}\Sigma_i$$

of Σ_i *strata* (singular: *stratum*) such that for two relations $R\in\mathscr{R}_i, S\in\mathscr{R}_j$ such that there is an arc $e=\left(R\longrightarrow S\right)\in\vec{\mathfrak{G}}_{\Pi}$ (i) if e is a positive arc, then $i\geq j$ and for every rule $\mathbf{r}$ such that $Head_{\mathbf{r}}\supseteq S$ we have $\mathbf{r}\in\bigcup_{i\geq j}\Sigma_i$; (ii) if e is a negative arc, then $i>j$ and for every rule $\mathbf{r}$ such that $Head_{\mathbf{r}}\supseteq S$ we have $\mathbf{r}\in\bigcup_{i>j}\Sigma_i$.

From this Definition, it is evident that in fact we have

$$\mathscr{P}_{Sch\left(\Pi_{\Delta}^{\neg}\right)}=\bigcup_{i\geq 1}^{n}\Sigma_i=\bigcup\left\{\left(\mathbf{r},p\right)|Head_{\mathbf{r}}\supseteq p\text{ and }p\in\mathscr{R}_i\right\}$$

such that each stratum Σ_i corresponds to a rank $\mathscr{R}_i$.

Example 134. The computation of $\mathscr{P}_{Sch\left(\Pi_{1}^{\neg}\right)}$ yields

$$\Sigma_0=\emptyset$$

$$\Sigma_1=\left\{p\left(X,Y\right)\leftarrow q\left(X,Y\right)\right\}\cup\left\{p\left(X,Y\right)\leftarrow q\left(X,Z\right),p\left(Z,Y\right)\right\}$$

$$\Sigma_2=\left\{r\left(X,Y\right)\leftarrow s\left(X,Y\right),\neg p\left(X,Y\right)\right\}$$

and

$$\Sigma_3=\left\{t\left(X,Y\right)\leftarrow r\left(X,Y\right)\right\}\cup\left\{t\left(X,Y\right)\leftarrow r\left(X,Z\right),t\left(Z,Y\right)\right\}.$$

Intuitively, we have a finite sequence of subprograms $\Sigma_1, \Sigma_2, ..., \Sigma_n$ where each Σ_i for $1 \leq i \leq n$ defines at least one EDB relation. The evaluation order for the relations falls on the ranks in the following way:

Definition 135. *Evaluation order:* Let there be given the relations in $\mathscr{R}_0$ and $I = (E_\Delta)_g \subseteq EDB\left(\Pi_\Delta^\neg\right)_g$.

1. Evaluate the relations in $\mathscr{R}_1$: All relations $R \in Head_\mathbf{r}$ for $\mathbf{r} \in \Sigma_1$ are defined.[40] This evaluation yields
$$J_1 \subseteq Sch\left(\Sigma_1\right)_g.$$
2. Evaluate the relations in $\mathscr{R}_2$ considering the relations in $EDB\left(\Pi_\Delta^\neg\right)$ and $\mathscr{R}_1$ as $EDB\left(\Sigma_2\right)$, where $\neg R\left(\vec{t}\right)$ is true if $R\left(\vec{t}\right)$ is false in $I \cup J_1$: All relations $R \in Head_\mathbf{r}$ for $\mathbf{r} \in \Sigma_2$ are defined. This evaluation yields
$$J_2 \subseteq Sch\left(\Sigma_2\right)_g.$$
3. Repeat for $\mathscr{R}_i$, $i = 3, ..., n$ considering the relations in $EDB\left(\Pi_\Delta^\neg\right)$ and $\mathscr{R}_1, \mathscr{R}_2, ..., \mathscr{R}_{i-1}$ as $EDB\left(\Sigma_i\right)$, where $\neg R\left(\vec{t}\right)$ is true if $R\left(\vec{t}\right)$ is false in $I \cup J_1 \cup J_2 \cup ... \cup J_{i-1}$: All relations $R \in Head_\mathbf{r}$ for $\mathbf{r} \in \Sigma_i$ are defined. This evaluation yields:
$$\begin{gathered} J_3 \subseteq Sch\left(\Sigma_3\right)_g \\ \vdots \\ J_n \subseteq Sch\left(\Sigma_n\right)_g \end{gathered}$$
4. $$\Pi_\mathscr{P}\left(I\right) = I \cup J_1 \cup J_2 \cup ... \cup J_n$$
where $\Pi_\mathscr{P}\left(I\right)$ denotes the evaluation of $\Pi_\Delta^\neg$ on I with respect to $\mathscr{P}$.

Example 136. With respect to $\Pi_1^\neg$, let us consider $EDB\left(\Pi_1^\neg\right) = \{q, s\}$ such that $I = \{q\left(a, b\right), s\left(a, b\right)\}$. Then, the evaluations of $\mathscr{R}_i$ for $i = 1, 2, 3$ give:

$$J_1 = \{p\left(a, b\right)\}$$

$$J_2 = \{\neg r\left(a, b\right)\}$$

$$J_3 = \{\neg t\left(a, b\right)\}$$

$$\Pi_\mathscr{P}\left(I\right) = \{q\left(a, b\right), s\left(a, b\right), p\left(a, b\right), \neg r\left(a, b\right), \neg t\left(a, b\right)\}$$

Note that $\neg r\left(a, b\right)$ and $\neg t\left(a, b\right)$ are here to be interpreted as "$r\left(a, b\right)$ is false in J_2" and "$t\left(a, b\right)$ is false in J_3", as facts, whether EDB or IDB ones, are never "negative." To play it one the safe side, one may in fact consider that $J_2 = \emptyset$ and $J_3 = \emptyset$, so that

$$\Pi_\mathscr{P}\left(I\right) = \{q\left(a, b\right), s\left(a, b\right), p\left(a, b\right)\}.$$

From Definition 133, it is also obvious that no negative relation occurs in the body of a rule in Σ_0, which can actually be empty.

[40]Note that Σ_1 does not have any negated relations for the reason that EDB facts cannot be negated relations.

Theorem 137. *A Datalog*$^{\neg}$ *program* $\Pi^{\neg}_{\Delta}$ *is stratified iff in its dependency graph there are no cycles containing a negative arc.*

Proof. (Sketch) ($\Rightarrow$) Suppose that we have some program $\Pi^{\neg}_{\Delta}$ that is stratifiable. Let $(R_1 \longrightarrow ... \longrightarrow R_k \longrightarrow R_1) \in \vec{\mathfrak{G}}_{\Pi^{\neg}_{\Delta}}$ where some $R_i \longrightarrow R_j$ for $1 \leq i \leq k, 1 \leq j \leq k$ is a negative arc. Let this arc be $R_k \rightarrow R_1$. Then, $R_1 > R_1$ by Definition 133.ii, but this is a contradiction. ($\Leftarrow$) Left as an exercise. $\square$

Example 138. Let us add the rule $p(X,Y) \leftarrow t(X,Y)$ to program $\Pi^{\neg}_{1}$ of Example 129. The new program $\Pi^{\neg}_{1'}$ is not stratifiable. In effect, the dependency graph of this new program has a cycle containing a negative arc (see Fig. 20).

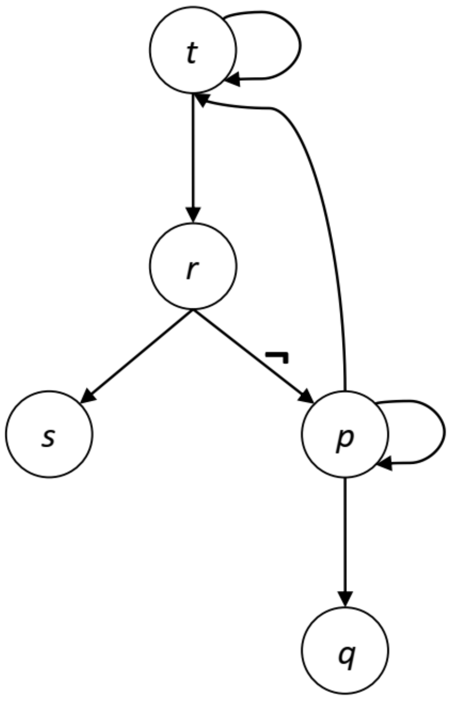


Figure 20: Dependency graph of a non-stratifiable program.

I conclude the study of Datalog$^{\neg}$ by giving two important statements.

Proposition 139. *For any given program* $\Pi^{\neg}_{\Delta}$ *that is stratifiable* $\Pi_{\mathscr{P}}(I)$ *is well defined.*

Theorem 140. *For any given program* $\Pi^{\neg}_{\Delta}$ *that is stratifiable* $\Pi_{\mathscr{P}}(I)$ *is a minimal model* K *of* $\Pi^{\neg}_{\Delta}$ *such that* $K|EDB(\Pi^{\neg}_{\Delta}) = I$.

Acknowledgments

This article is essentially a reprint of contents first published in Augusto (2022), with just a few additions (e.g., the short discussion of A. Newell's concept of knowledge level in the

Introduction). My thanks to College Publications, London, for allowing authors to keep the copyright of their books and to reprint parts of them.

References

Abiteboul, S., Hull, R., & Vianu, V. (1995). *Foundations of databases.* Reading, MA, etc.: Addison-Wesley.

Apt, K. R. (1996). *From logic programming to Prolog.* Upper Saddle River, NJ: Prentice Hall.

Apt, K. R., Blair, H. A., & Walker, A. (1988). Towards a theory of declarative knowledge. In J. Minker (ed.), *Foundations of deductive databases and logic programming* (pp. 89-148). Los Altos, CA: Morgan Kaufmann.

Augusto, L. M. (2019). *Formal logic: Classical problems and proofs.* London: College Publications.

Augusto, L. M. (2020a). *Many-valued logics: A mathematical and computational introduction.* 2nd ed. London: College Publications.

Augusto, L. M. (2020b). *Logical consequences. Theory and applications: An introduction.* 2nd ed. London: College Publications.

Augusto, L. M. (2020c). Toward a general theory of knowledge. *Journal of Knowledge Structures & Systems, 1*(1), 63-97.

Augusto, L. M. (2021). From symbols to knowledge systems: A. Newell and H. A. Simon's contribution to symbolic AI. *Journal of Knowledge Structures & Systems*, *2*(1), 29-62.

Augusto, L. M. (2022). *Computational logic. Vol. 1: Classical deductive computing with classical logic.* 3rd ed. London: College Publications.

Brewka, G. & Dix, J. (2005). Knowledge representation with logic programs. In D. M. Gabbay & F. Guenthner (eds.), *Handbook of philosophical logic. Vol. 12.* 2nd ed. (pp. 1-85). Dordrecht: Springer.

Ceri, S., Gottlob, G., & Tanca, L. (1989). What you always wanted to know about Datalog (and never dared to ask). *IEEE Transactions on Knowledge and Data Engineering*, *1*(1), 146-166.

Ceri, S., Gottlob, G., & Tanca, L. (1990). *Logic programming and databases.* Berlin & Heidelberg: Springer.

Clark, K. L. (1978). Negation as failure. In H. Gallaire & J. Minker (eds.), *Logic and data bases* (pp. 293-322). New York: Plenum.

Date, C. J. (2004). *Introduction to database systems.* 8th ed. Reading, MA: Addison-Wesley.

Gallaire, H., Minker, J., & Nicolas, J.-M. (1984). Logic and databases: A deductive approach. *Computing Surveys*, *16*(2), 153-185.

Greco, S. & Molinaro, C. (2016). *Datalog and logic databases.* Morgan & Claypool.

Kifer, M. & Liu, Y. A. (2018). *Declarative logic programming: Theory, systems, and applications.* ACM and Morgan & Claypool Publishers.

Leitsch, A. (1997). *The resolution calculus.* Berlin, etc.: Springer.

Minker, J. (1997). Logic and databases: Past, present, and future. *AI Magazine*, *18*(3), 21-47.

Newell, A. (1980). Physical symbol systems. *Cognitive Science*, *4*, 135-183.

Newell, A. (1982). The knowledge level. *Artificial Intelligence*, *18*(1), 87-127.

Newell, A. (1990). *Unified theories of cognition.* Cambridge, MA & London, UK: Harvard University Press.

Reiter, R. (1978). On closed world data bases. In H. Gallaire & J. Minker (eds.), *Logic and data bases* (pp. 55-76). New York: Plenum.

Reiter, R. (1984). Towards a logical reconstruction of relational database theory. In M. L. Brodie, J. Mylopolous, & J. W. Schmidt (eds.), *On conceptual modeling. Perspectives from artificial intelligence, databases, and programming languages* (pp. 191-238). New York: Springer.

Shepherdson, J. C. (1984). Negation as failure: A comparison of Clark's completed data base and Reiter's closed world assumption. *Journal of Logic Programming*, *1*(1), 1-48.

Sterling, L. & Shapiro, E. (1994). *The art of Prolog.* Cambridge, MA & London, England: The MIT Press.

van Emden, M. H. & Kowalski, R. A. (1976). The semantics of predicate logic as a programming language. *Journal of the Association for Computing Machinery*, *23*(4), 733-742.

van Gelder, A. (1986). Negation as failure using tight derivations for general logic programs. In *Proceedings of the Third IEEE Symposium on Logic Programming* (pp. 137-146).

Cite this article as:

Augusto, L. M. (2024). Deductive computing over knowledge bases: Prolog and Datalog. *Journal of Knowledge Structures & Systems*, *5*(1), 1-62.

EDITORIAL INFORMATION

Editor-in-chief: Luis M. Augusto
Reprint [Source: Augusto (2022), Chapter 9]