Semantic Approaches to Logic Programming

by

Thanos Tsouanas

A Thesis Presented in Partial Fulfillment of the Requirements for the Degree of Master of Science.

Supervised by

Panos Rondogiannis

Thesis committee

Panos Rondogiannis
Costas D. Koutras
Nikolaos S. Papaspyrou

"MPLA" Graduate Program in Logic, Algorithms and Computation

September 27th, 2010

Dedicated to the rocking memory of my dear friend Iasonas Tzoufras,
a fellow musician and mathematician.

Preface

Prerequisites. The reader is assumed to be familiar with the basic notions of logic, including first-order theories and their semantics (interpretations and models). Also, some facts and tools from set theory are needed (especially the basic facts about ordinals). The appendix sketches over some of them, and gives pointers to the literature for further study.

Familiarity with logic programming is not required, but will certainly make some parts of the text clearer. The reader who is interested in actual implementations and applications of logic programming, is advised to study [SS94].

Notational & typographical conventions. Following common practice, I use the abbreviations: "iff" for "if and only if", "wrt" for "with respect to", and "wff" for "well-formed formula".* A halmos (viz. "▮") is used instead of the more traditional "Q.E.D." to mark the end of a proof; examples die by "◀".

$\mathbb{N}$ and $\mathbb{R}$ are the sets of natural and real numbers respectively. 0 is happy to be a natural number. The powerset of A is identified with the set $\mathbf{2}^A$ and denoted also by $\wp A$, while the set of its *finite* subsets is $\wp_{\text{fin}} A$. $|A|$ is the cardinal number of A. I also use the expression "$S \subseteq_\kappa A$" where κ is a cardinal number, as an abbreviation for "$S \subseteq A$ with $|S| = \kappa$". In a similar fashion, "F is a finite subset of A" shortens to "$F \subseteq_{\text{fin}} A$". When using "$\dot\cup$" in place of "$\cup$", it is implied that the union is disjoint.

$\mathbb{N}$
$\mathbb{R}$
$\mathbf{2}^A$
$\wp A$
$\wp_{\text{fin}} A$
$|A|$
$\subseteq_\kappa$
$\subseteq_{\text{fin}}$
$\dot\cup$

I write "$=_{\text{df}}$" for "is defined as", and "$:=$" (or simply "$=$") for assigning values to metavariables. Ofttimes I contract "$= \ \cdots \ =$" into a single "$\overset{\cdots}{=}$". Speaking of dots, an expression like "$\overset{-i}{\ldots}$" excludes the item indexed by i from

$=_{\text{df}}$
$:=$
$\overset{\cdots}{=}$
$\overset{-i}{\ldots}$

*For some reason, the abbreviation "wrt" tends to cause trouble: using *Google Books*, a search for `"wrt to"` returns approximately 7,760 results! Maybe this is because "with respect" is a 3-syllable phrase, but this is probably a matter for cognitive psychology.

$\therefore$ $\overset{\therefore}{=}$ ↯

the dots, e.g., “$p_1 \vee \overset{-i}{\cdots} \vee p_n$” stands for “$p_1 \vee \cdots \vee p_{i-1} \vee p_{i+1} \vee \cdots \vee p_n$”. When writing mathematical relations I favor “$\therefore$” over “therefore”. In a chain of equations, the symbol “$\overset{\therefore}{=}$” is to be read as “[and] therefore equals”, hinting that this equality holds thanks to something more than just the previous equality. Following not-so-common practice, when reasoning by *reductio ad absurdum*, “↯” signifies a contradiction. I first encountered this in [DP02], and liked it.

$\forall\!\!\forall$ $\exists\!\!\exists$

As this is a mathematical logic text, the symbols “$\forall$” and “$\exists$” are reserved for use in object languages, like the language of first-order logic. Instead, I use the symbols “$\forall\!\!\forall$” and “$\exists\!\!\exists$” in the metalanguage, for “for all” and “there exist(s)” respectively.

$\frown$

Language symbols in program rules are typeset according to their kind: fun is a function symbol, *rel* is a relation symbol, and X is a variable. Constant symbols like a are typeset in the same way as function symbols, since they can be considered to be nullary function symbols themselves. Usually I adopt a more functional notation, dropping parentheses where possible. Syntactic sugar terms (e.g. numerals) always wear a “$\frown$”; e.g., $\widehat{3}$, $\mathsf{sss0}$, $\mathsf{s}^3(\mathsf{0})$, and $\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{0})))$ are all different names for the *same* term. This notation is extended respectfully to sets, e.g., $\widehat{\mathbb{N}}$ is the set of all numerals. Families of models or sets usually appear in “script” typeface: $\mathscr{A}$, $\mathscr{B}$, $\mathscr{C}$, etc.

ε

Regarding variables of the metalanguage—$\mathcal{P}$, $\mathcal{Q}$, and $\mathcal{R}$ usually range over programs; u and v over some (Herbrand) universe of terms; s and t also denote terms; a, b, and c constants; n, m, and k natural numbers (while $\widehat{n}$, $\widehat{m}$, and $\widehat{k}$ the corresponding numeral terms); f, g, and h functions or function symbols if typeset accordingly; similarly, p, q, and r denote relations or relation symbols, and also propositional variables; atomic formulæ usually dwell in $\{A, B, C\}$, wffs in $\{F, G, H\}$, and arbitrary expressions in $\{D, E\}$; x, y, z and w stand for variables of the object language. Lowercase greek letters like ϑ, σ and τ tend to be substitutions, of which the identity is always ε. *None of these conventions is strictly followed: I freely deviate from them as I see fit.*

Acknowledgements

It is hard to acknowledge everyone that deserves to be credited, but I'll give it a try. There's plenty of space to write your name if I forgot to include you while I should.

I should first mention Yiannis Moschovakis whose Set Theory and Recursion Theory courses were so influential and inspiring that turned my interest into mathematical logic in general. Costas Dimitracopoulos who (thankfully) thinks in a logical way, and gets things done instead of just planned and arranged! Panos Rondogiannis, Nikolaos Papaspyrou, Costas Koutras, and Dimitrios Thilikos, for believing in me, for giving unforgettable courses, and for coping with various "peculiarities" of mine, including my night-owl sleeping patterns. Aristides Katavolos, who gives the most vivid and exciting Mathematical Analysis courses and Evangelos Raptis whose Algebra courses were the first I attended. I would also like to thank Mike Pozantzis, my English teacher to whom I owe a lot. The secretariats of MPLA deserve special mention, as they are the most effective, hard-working, efficient, and kind administration I've known: a huge thanks to Anna Vassilaki and Chrisafina Hondrou.

Of course I owe a lot to my family: my mother who's one of the smartest and kindest persons I know, my beloved sister for being supportive since I was born, and my father, who introduced me to computers when I could barely speak.

All my friends that I can rely upon, especially Yiannis "tzi" Tselekounis for his tremendous help with the most unusual and irrational requests that I might have had while working on my thesis; and also Giorgos "zoulou" Kapetanakis, Matoula Petrolia, Harris Hanialidis, for their help and love. Michail "pshlos"

Dimakos for uncountably-many helping hands during my undergraduate studies; things would certainly be much harder without him. Irini Kalliamvakou for being a solid rock by my side, and for so many things that cannot be expressed in any language I know. Panos Tsaknias and Alexandros "enecon" Batsis for many edifying conversations, and also Aristotelis Misios (whose poor formal grammar failed miserably to compete with mine—sorry). Mary Chlouveraki deserves to be singled out, as I feel I owe it to her, my sudden realization of interest in pure mathematics. A brilliant person and dearest friend, probably the only one whose advice I do not automatically ignore—for good reason. I'm happy to thank Ramile Carvalho, who's been a most pleasant and surprising addition to my life during the last month of the compilation of this thesis, and made the boring parts, fun.

I am also indebted to Spyros Liberis who introduced me to the world of Unix over a decade ago, instantly changing my relationship with computers. I have become dependant on all those computer tools that make my life so much easier, and so I feel that I should give credit to the vim text editor, Donald Knuth for TEX, Theo de Raadt for OpenBSD, the entire Haskell community, darcs, xmonad and of course Wikipedia.

I would be unfair not to mention the musicians who kept me great company while studying, researching, and typesetting this thesis, especially Dying Fetus, Rotting Christ, Burzum, Cave, Origin, Crooked Still, Nikos Xylouris, Tori Amos and myself.

It is hard to be productive without good food, and so I'd like to mention Christos Barkas and his «Paradosiako» kebab tavern in Ano Ilissia, «Santé» bougatsæ on Thisseos Ave., «Agapizza» in Hymettos and the «Submarine» crêperie in Goudi. I'm also indebted to the breweries of Leffe, Fraoch, and Alba and of course Marilou who provided fair amounts of bottles.

Finally, I hail Crete—I was on decks of Cretan ferries when I began developing my own ideas on denotational semantics for logic programming.

Thank you!

Short contents

Contents

Chapter 1

Introduction

This chapter constitutes a very sketchy and informal overview of the ideas and concepts that follow.

1.1 Logic programming

Robinson in [Rob65] introduced the *resolution* inference rule, which proved to be specially useful for the birth of logic programming. A few years later, Kowalski and van Emden showed in [vEK76] that logic has a procedural interpretation, which allows us to treat it rigorously as an effective programming language. In this landmark paper, logic programming was born.

A really vague definition of a logic program is the following:

A logic program is a set of rules.

But what is a rule? Depending on what kind of formulæ we allow as rules, we get a different notion of a logic program. In this text the following kinds of *rules* are considered:

program rule

$$
\begin{aligned}
a &\leftarrow b_1, \ldots, b_n && \text{(LP)}\\
a &\leftarrow b_1, \ldots, b_n, {\sim}c_1, \ldots, {\sim}c_m && \text{(LPN)}\\
a_1 \vee \cdots \vee a_k &\leftarrow b_1, \ldots, b_n && \text{(DLP)}\\
a_1 \vee \cdots \vee a_k &\leftarrow b_1, \ldots, b_n, {\sim}c_1, \ldots, {\sim}c_m. && \text{(DLPN)}
\end{aligned}
$$

Given a logic program, we want to be able to ask certain *queries.* The implemented system then gives answers according to the rules of the program, in a way that an affirmative answer is given only if there is enough evidence to support it. If some question can be answered consistently in more than one ways, we always prefer the *least* true one.

▶ *Example 1.1.* Consider the following program

$$\mathcal{P} := \left\{ \begin{array}{r} \mathit{dangerous}(X) \leftarrow \mathit{cat}(X), {\sim}\mathit{sleeping}(X) \\ \mathit{cat}(\mathsf{prokopis}). \\ \mathit{cat}(\mathsf{oliver}). \\ \mathit{sleeping}(\mathsf{oliver}). \end{array} \right\}.$$

Despite the fact that it is consistent to consider both cats to be both dangerous and sleeping, since we have no reason to believe that prokopis is sleeping, we consider him awake, and therefore (by the first rule) dangerous. On the other hand, we know for a fact that oliver is sleeping, so the premise of the first rule fails to hold, and (preferring falsehood again) we do not consider him dangerous. ◀

1.2 Semantics

The idea is that the logic programmer uses a set of rules in order to describe a certain model that she has in mind; we usually refer to it as *the intended model.* Denotational semantics define a precise mathematical object to each program, which we hope to reflect the intended model as well as possible. When giving semantics to a program, it's best to think of a model as a way to consistently assign truth values to possible queries—and this is what we seek to define.

LP If we restrict our rules to (LP), we speak of *definite* logic programs. Semantically speaking, this is the best scenario. We have the so-called *least Herbrand model*, which provides well-established semantics that everyone finds convenient. But programming in LP is very restrictive, and therefore we have to allow more general rules if we aim to arrive at applications that can actually be applied.

LPN Once we allow negation in the bodies of our rules, we lose certain comforts. Programming in LPN is more convenient, but it's harder to define satisfying semantics. Nevertheless, after a lot of effort, we have arrived at the *well-founded model*, which is widely approved as the “correct” one by the Logic Programming community. Another important class of semantics is based on the *stable model semantics.* But this school of thought assigns zero, one, or many models to each program, and therefore it deviates from common practices.

DLP If instead of negations in bodies, we allow disjunctions in heads, we have what is called “disjunctive logic programming”. Here the task of defining semantics is puzzling, as we no longer have a *least* model. Even the simplest disjunctive program, $\{a \lor b\}$, possesses three models, $\{a\}$, $\{b\}$ and $\{a, b\}$, none of which is least! If we wish to stick with a single model, and also answer affirmatively only when we are certain that something is true, we should somehow capture the idea that $\{a \lor b\}$ is the correct “model”. And this is not a model to begin with!

DLPN This is the best of both worlds for the programmers, which unfortunately means the worst of both worlds in terms of the difficulties one has to face to define semantics. Recently, an *infinite-valued minimal model semantics* was proposed which seems to generalize both the infinite-valued well-founded model approach of LPN and the minimal model semantics of DLP to the general case of DLPN.

Remark 1.1. Consider the following formulæ:

$$p \leftarrow \neg q \qquad\qquad q \leftarrow \neg p.$$

From a classical logic point of view, these sentences are equivalent, and one would expect them to yield equivalent programs. But, using negation-as-failure, this isn't so: in the first program p is true, while q is false, while in the second one the opposite happens.[a]

Abandoning negation-as-failure, we can get rid of negations altogether, by "switching sides", and "changing signs", just like we do with elementary algebraic equations. For example,

$$a \lor b \lor \neg c \leftarrow d \land \neg e \land \neg f \quad \Longleftrightarrow \quad a \lor b \lor e \lor f \leftarrow d \land c.$$

This would reduce the problem of dealing with negations to the problem of dealing with disjunctions, and vice versa. But this is not how we think of rules in logic programming.

[a] In disjunctive databases, the rules have no particular order. These two different "programs" are seen as equivalent to $p \lor q$.

1.3 Using games to define semantics

LP In [DCLN98], a game was defined to provide semantics for LP. This was done in a much satisfactory way: not only did this approach reflect a new and exciting way to deal with logic programs, it was also shown to be equivalent with the least Herbrand model semantics.

LPN Some years later, in [RW05a], another game was defined by Rondogiannis and Wadge, that was able to deal with LPN successfully: it yields the (infinite-valued) well-founded model!

DLP & DLPN *The main contribution of this thesis is the definition of two new games, one for DLP and one for DLPN, in order to complete this picture of the game-theoretic view of logic programming semantics.*

> «Facts are meaningless. You could use facts to prove anything that's even remotely true!»
>
> Homer J. Simpson

Chapter 2

Logic programs

I proceed with a quick pace in this chapter, as it contains well-established facts. More details than you will probably care to know can be found in standard texts like [Llo93, Apt90].

2.1 Introducing programs

Definition 2.1. A *clause* is a formula of the form $\forall(L_1 \vee \cdots \vee L_n)$, where each L_i is a literal. A *Horn clause* is a clause with at most one positive literal. If it has exactly one, we speak of a *definite clause* or *rule* and refer to that unique positive literal as the *head* of the rule; the remaining literals (if any) constitute its *body*. Given a rule r, we write $\mathrm{head}(r)$ for its head and $\mathrm{body}(r)$ for its body. It is common to write the rule

$$A \vee \neg B_1 \vee \cdots \vee \neg B_m \qquad \text{as} \qquad A \leftarrow B_1, \ldots, B_n;$$

in case $n = 0$, we call the rule a *fact*, and write it as either "A.", "$A \leftarrow$.", or "$A \leftarrow$".[a)]

clause
Horn clause
definite clause
LP!rule
head
body
$\mathrm{head}(r)$
$\mathrm{body}(r)$
fact

► *Example 2.1.* Consider the following formulæ:

(i) $p \vee q$
(ii) $p(f(x, w), y) \vee r(y, x) \vee \neg q(x)$
(iii) $p \vee \neg q \vee \neg r$
(iv) $p(x, y) \vee \neg q(x) \vee \neg q(y)$
(v) $p(x, y, z)$
(vi) $\neg p \vee \neg q$
(vii) $\exists x p(x) \vee \neg q$
(viii) $p \vee (q \wedge r)$.

Here (i)–(vi) are clauses, out of which (iii)–(vi) are definite, (iii)–(v) are Horn and (v) is a fact. ◄

Definition 2.2. A set of LP rules is called an *LP program* or a *definite program*.

LP program

Definition 2.3. In a program $\mathcal{P}$, the set of all rules that share a common relation symbol r in their heads constitutes the *definition* of r. It is denoted

definition

[a)] Some authors call facts "unit clauses", but facts will be facts here.

$\mathbf{def}_{\mathcal{P}}(r)$

by $\mathbf{def}_{\mathcal{P}}(r)$.

► *Example 2.2.* The following is a definite program:

$$\left\{ \begin{array}{c} \mathit{odd}(\mathsf{s}(\mathsf{0})) \\ \mathit{odd}(\mathsf{s}(\mathsf{s}(X))) \leftarrow \mathit{odd}(X) \end{array} \right\}.$$

It is meant to describe the odd numbers, and we shall call it $\mathcal{P}_{odd}$. In this case, the whole program is the definition of *odd*. ◄

Since each program has infinitely many logical consequences, we need a way to *query* a program by asking it specific questions. This brings us to the notion of goal.

goal

Definition 2.4. A *definite goal*, is any formula of the form

$$\forall(\neg(A_1 \wedge \cdots \wedge A_n)),$$

which we, following common practice, write as

$$\leftarrow A_1, \ldots, A_n.$$

subgoal
empty goal
⧄

Each A_i is called a *subgoal*. The *empty goal* is denoted by ⧄; it always fails ☹.

To see how such a formula corresponds to a query at a logic programming system, consider the universal closure

$$\forall X_1 \cdots \forall X_k \neg(A_1 \wedge \cdots \wedge A_n),$$

which is equivalent to the formula

$$\neg\exists X_1 \cdots \exists X_k(A_1 \wedge \cdots \wedge A_n).$$

In other words, instead of asking

«For which X_i's is $\{A_1, \ldots, A_n\}$ a consequence of $\mathcal{P}$?»,

we claim that

«There are no X_i's such that $\{A_1, \ldots, A_n\}$ is a consequence of $\mathcal{P}$!»

instead, and the system tries to disprove our claim. It does so *constructively*, returning a suitable substitution if possible. If it fails, it is because our claim was indeed true.

2.2 Herbrand models

ground(r)
Ground($\mathcal{P}$)

Definition 2.5. For any clause r, the set of its ground instances is denoted by ground(r). For any program $\mathcal{P}$, we define Ground($\mathcal{P}$) by

$$\mathrm{Ground}(\mathcal{P}) =_{\mathrm{df}} \bigcup_{r \in \mathcal{P}} \mathrm{ground}(r).$$

Definition 2.6. The *Herbrand universe* $\mathcal{U}_L$ of a first-order language L is the set of all ground terms which can be formed by using the constants and function symbols of L.[b)]

Herbrand universe (HU) $\mathcal{U}_L$

With a slight abuse of notation, when using a program $\mathcal{P}$ in place of a language L, we mean the *language $\mathcal{L}_\mathcal{P}$ associated with* $\mathcal{P}$, i.e., the language that consists of all the symbols that appear in $\mathcal{P}$.

associated language $\mathcal{L}_\mathcal{P}$

▶ *Example 2.3.* The Herbrand universe $\mathcal{U}_\mathcal{P}$ of the program

$$\mathcal{P} := \left\{ \begin{array}{l} p(X) \leftarrow q(\mathsf{f}(X), \mathsf{g}(X)) \\ r(a) \leftarrow \end{array} \right\}$$

looks like

$$\mathcal{U}_\mathcal{P} = \{a, \mathsf{f}(a), \mathsf{g}(a), \mathsf{f}(\mathsf{f}(a)), \mathsf{f}(\mathsf{g}(a)), \mathsf{g}(\mathsf{f}(a)), \mathsf{g}(\mathsf{g}(a)), \mathsf{f}(\mathsf{f}(\mathsf{f}(a))), \dots\}. \qquad ◀$$

Definition 2.7. The *Herbrand base* $\mathcal{HB}_L$ of a language L is the set $\mathcal{HB}_L$ of all ground atoms which can be formed by using relation symbols of L with arguments from $\mathcal{U}_L$.

Herbrand base (HB) $\mathcal{HB}_L$

▶ *Example 2.4.* Consider the program $\mathcal{P}_{odd}$ we met earlier in Example 2.2. Since this program contains just one constant $\mathsf{0}$ and one function symbol s, its Herbrand universe is

$$\mathcal{U}_{\mathcal{P}_{odd}} = \{\mathsf{0}, \mathsf{s}(\mathsf{0}), \mathsf{s}(\mathsf{s}(\mathsf{0})), \mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{0}))), \dots\},$$

and since there is just one relation symbol, *odd*, its Herbrand base is

$$\mathcal{HB}_{\mathcal{P}_{odd}} = \{odd(\mathsf{0}), odd(\mathsf{s}(\mathsf{0})), odd(\mathsf{s}(\mathsf{s}(\mathsf{0}))), \dots\}. \qquad ◀$$

▶ *Example 2.5.* Let $\mathcal{P}$ be the program

$$\begin{array}{l} owns(\mathsf{owner}(\mathsf{hyundai}), \mathsf{hyundai}) \leftarrow \\ \qquad\qquad happy(X) \leftarrow owns(X, \mathsf{hyundai}). \end{array}$$

Then

$$\begin{aligned} \mathcal{U}_\mathcal{P} &= \{\mathsf{hyundai}, \mathsf{owner}(\mathsf{hyundai}), \mathsf{owner}(\mathsf{owner}(\mathsf{hyundai})), \dots\} \\ \mathcal{HB}_\mathcal{P} &= \{owns(u, v) \mid u, v \in \mathcal{U}_\mathcal{P}\} \cup \{happy(w) \mid w \in \mathcal{U}_\mathcal{P}\}. \qquad ◀ \end{aligned}$$

> REMARK 2.1. Unless a program $\mathcal{P}$ contains no function symbols, both $\mathcal{U}_\mathcal{P}$ and $\mathcal{HB}_\mathcal{P}$ will be infinite.

Definition 2.8. Let $\mathcal{P}$ be a definite program. A *Herbrand interpretation* of $\mathcal{P}$ is an interpretation $\mathfrak{I}$ with domain $\mathcal{U}_\mathcal{P}$ such that:

Herbrand interpretation

[b)] One could either restrict this definition to languages that indeed contain at least one constant, or allow the addition of a single constant to form ground terms, in case the language contains none.

(i) $c_{\mathfrak{I}} = c$ for each constant symbol c, and

(ii) $f_{\mathfrak{I}}(t_1, \dots, t_n) = f(t_1, \dots, t_n)$ for each function symbol f.

Herbrand model
$\mathscr{H}_{\mathcal{P}}$

This means that to completely define a Herbrand interpretation all we need to do is to specify how it behaves on the relation symbols. A Herbrand interpretation which satisfies a set $\mathcal{P}$ of (closed) formulæ is called a *Herbrand model* of $\mathcal{P}$. We denote by $\mathscr{H}_{\mathcal{P}}$ the set of all Herbrand models of $\mathcal{P}$.

> 𝄾 Remark 2.2. *We usually identify an interpretation of $\mathcal{P}$ with the subset of the elements of $\mathcal{HB}_{\mathcal{P}}$ that it satisfies.* Thus we write $\mathbf{2}^{\mathcal{HB}_{\mathcal{P}}}$ for the set of Herbrand interpretations of $\mathcal{P}$, which becomes a *complete lattice*, ordered by set inclusion.
>
> 𝄾 Remark 2.3. The Herbrand base of a definite program $\mathcal{P}$ is always a Herbrand model of the program, although not a particularly interesting one.

Theorem 2A (Model intersection property)**.** *Let $\mathscr{M}$ be a non-empty family of Herbrand models of a definite program $\mathcal{P}$. Then $\mathfrak{I} =_{\mathrm{df}} \bigcap \mathscr{M}$ is also a Herbrand model of $\mathcal{P}$.*

Proof. Suppose not. Then there exists in $\mathcal{P}$ a ground instance of a clause

$$A \leftarrow B_1, \dots, B_n$$

which is not true under $\mathfrak{I}$, so that

$$\begin{aligned} &\mathfrak{I} \models \{B_1, \dots, B_n\}, \\ \text{but}\quad &\mathfrak{I} \not\models A. \end{aligned}$$

Therefore, for every $I \in \mathscr{M}$, $I \models \{B_1, \dots, B_n\}$, and there must also be some interpretation $I_0 \in \mathscr{M}$, such that $I_0 \not\models A$. But this means that $A \leftarrow B_1, \dots, B_n$ is not true under I_0, and therefore I_0 is not a model of $\mathcal{P}$, ↯. ▮

> 𝄾 Remark 2.4. Consider the program $\mathcal{P} = \{p(a) \vee q(b)\}$. Both $\{p(a)\}$ and $\{q(b)\}$ are Herbrand models of $\mathcal{P}$, but their intersection $\{p(a)\} \cap \{q(b)\} = \emptyset$ is not! What went wrong? The formula $p(a) \vee q(b)$ is not a definite clause, and consequently $\mathcal{P}$ is not a definite program. We investigate disjunctive clauses in Chapter 4.

least Herbrand model (LHM)
$\mathcal{M}_{\mathcal{P}}$

Theorem 2B (van Emden & Kowalski)**.** *The* $\subseteq$-least Herbrand model $\mathcal{M}_{\mathcal{P}}$ *of a definite program $\mathcal{P}$ satisfies*

$$\mathcal{M}_{\mathcal{P}} = \{A \in \mathcal{HB}_{\mathcal{P}} \mid \mathcal{P} \models A\}.$$

Proof. Both directions are easy. See [vEK76]. ▮

Denotational semantics

We now take $\mathcal{M}_\mathcal{P}$ as the meaning of $\mathcal{P}$; in symbols,

$$[\![\mathcal{P}]\!] =_{\mathrm{df}} \mathcal{M}_\mathcal{P}.$$

2.3 The rôle of fixpoints

It should seem natural by now that it is a good idea to consider the least Herbrand model $\mathcal{M}_\mathcal{P}$ of a definite program $\mathcal{P}$ as the intended one, i.e., as the semantic meaning of $\mathcal{P}$. By definition and Theorem 2A, we know that

$$\mathcal{M}_\mathcal{P} = \bigcap \{ M \subseteq \mathcal{HB}_\mathcal{P} \mid M \text{ is a Herbrand model of } \mathcal{P} \},$$

but this doesn't help much: how can we *construct* $\mathcal{M}_\mathcal{P}$?

We describe a way to successively approximate $\mathcal{M}_\mathcal{P}$, using the so-called "immediate consequence operator" $\mathcal{T}_\mathcal{P}$.

Definition 2.9. Then the *immediate consequence operator* is a unary operator $\mathcal{T}_\mathcal{P}$ on $\mathbf{2}^{\mathcal{HB}_\mathcal{P}}$, defined by

immediate consequence operator

$\mathcal{T}_\mathcal{P}$

$$\mathcal{T}_\mathcal{P}(I) =_{\mathrm{df}} \{ A \mid A \leftarrow B_1, \dots, B_n \in \mathrm{Ground}(\mathcal{P}) \text{ and } I \models \{B_1, \dots, B_n\} \}.$$

Let's see how the immediate consequence operator of a program acts on various inputs.

▶ *Example 2.6.* We examine the $\mathcal{T}_\mathcal{P}$ of the program

$$\mathcal{P} := \left\{ \begin{aligned} p(\mathsf{f}(X)) &\leftarrow p(X) \\ q(a) &\leftarrow p(X) \end{aligned} \right\}.$$

Note that this is a pretty *clueless* program: it contains no facts. This doesn't bother us, however, and so we compute:

clueless program

$$\begin{aligned} \mathcal{U}_\mathcal{P} &= \{a, \mathsf{f}(a), \mathsf{f}(\mathsf{f}(a)), \mathsf{f}(\mathsf{f}(\mathsf{f}(a))), \dots\} \\ \mathcal{HB}_\mathcal{P} &= \{p(a), p(\mathsf{f}(a)), p(\mathsf{f}(\mathsf{f}(a))), \dots\}. \end{aligned}$$

Consider the interpretations

$$\begin{aligned} I_1 &= \mathcal{HB}_\mathcal{P} \\ I_2 &= \{q(a)\} \cup \{p(\mathsf{f}(u) \mid u \in \mathcal{U}_\mathcal{P}\} \\ I_3 &= \emptyset, \end{aligned}$$

and how $\mathcal{T}_\mathcal{P}$ acts on each one of them:

$$\begin{aligned} \mathcal{T}_\mathcal{P}(I_1) &= \{q(a)\} \cup \{p(\mathsf{f}(u) \mid u \in \mathcal{U}_\mathcal{P}\} = I_2 \\ \mathcal{T}_\mathcal{P}(I_2) &= \{q(a)\} \cup \{p(\mathsf{f}(\mathsf{f}(u))) \mid u \in \mathcal{U}_\mathcal{P}\} \\ \mathcal{T}_\mathcal{P}(I_3) &= \emptyset = I_3. \end{aligned}$$

Thus $\mathcal{T}_\mathcal{P}$ maps I_1 to I_2, I_2 to some other (unnamed) interpretation, and I_3 to itself: it is a fixpoint. ◀

> 𝄾 Remark 2.5. It is constructive* to keep this point of view in mind: in a world $\mathbf{2}^{\mathcal{HB}_{\mathcal{P}}}$ of interpretations, $\mathcal{T}_{\mathcal{P}}$ determines a specific way to "move around". Starting from any I, a sequence of iterations is formed:
>
> $$\{I, \mathcal{T}_{\mathcal{P}}(I), \mathcal{T}_{\mathcal{P}}(\mathcal{T}_{\mathcal{P}}(I)), \mathcal{T}_{\mathcal{P}}(\mathcal{T}_{\mathcal{P}}(\mathcal{T}_{\mathcal{P}}(I))), \dots\}$$
>
> Furthermore, with the help of ordinals, this becomes even more useful, as we are about to see.
>
> *Some pun intended.

Let $\mathcal{P}$ be a definite program. Then there is a $\subseteq$-least interpretation I which is a fixpoint of $\mathcal{T}_{\mathcal{P}}$, i.e., $\mathcal{T}_{\mathcal{P}}(I) = I$ and equals the least Herbrand model $\mathcal{M}_{\mathcal{P}}$. Moreover,

$$\mathcal{M}_{\mathcal{P}} = \lim_n \mathcal{T}_{\mathcal{P}}^n(\emptyset).$$

↑-notation

Now is a good time to introduce the convenient "up-arrow" notation:[c)]

$$\begin{aligned} \mathcal{T}_{\mathcal{P}} \uparrow 0 &=_{\mathrm{df}} \emptyset \\ \mathcal{T}_{\mathcal{P}} \uparrow (n+1) &=_{\mathrm{df}} \mathcal{T}_{\mathcal{P}}(\mathcal{T}_{\mathcal{P}} \uparrow n) \\ \mathcal{T}_{\mathcal{P}} \uparrow \omega &=_{\mathrm{df}} \bigcup_{n \in \omega} \mathcal{T}_{\mathcal{P}} \uparrow n. \end{aligned}$$

Theorem 2C. *Let $\mathcal{P}$ be a definite program and $\mathcal{M}_{\mathcal{P}}$ its least Herbrand model.*

(i) *$\mathcal{M}_{\mathcal{P}}$ is the least Herbrand interpretation such that $\mathcal{T}_{\mathcal{P}}(\mathcal{M}_{\mathcal{P}}) = \mathcal{M}_{\mathcal{P}}$, i.e., it is the* least fixpoint *of $\mathcal{T}_{\mathcal{P}}$.*

(ii) *$\mathcal{M}_{\mathcal{P}} = \mathcal{T}_{\mathcal{P}} \uparrow \omega$.*

Proof. [Apt90] or [Llo93]. ▮

▶ *Example 2.7.* Here is how $\mathcal{M}_{P_{\mathsf{odd}}}$ can be constructed using this method:

$$\begin{aligned} &\mathcal{T}_{P_{\mathsf{odd}}} \uparrow 0 = \emptyset \\ &\mathcal{T}_{P_{\mathsf{odd}}} \uparrow 1 = \{odd(\mathsf{s}(0))\} \\ &\mathcal{T}_{P_{\mathsf{odd}}} \uparrow 2 = \{odd(\mathsf{s}(0)), odd(odd(\mathsf{s}(0))), \dots\} \\ &\qquad\qquad \vdots \\ &\mathcal{T}_{P_{\mathsf{odd}}} \uparrow \omega = \{odd(\mathsf{s}^n(0)) \mid n \in \{1,3,5,7,\dots\}\}. \end{aligned}$$ ◀

Theorem 2D (Continuity of $\mathcal{T}_{\mathcal{P}}$)**.** *For any definite program $\mathcal{P}$, the mapping $\mathcal{T}_{\mathcal{P}}$ is continuous.*

Proof. See [Llo93]. ▮

Theorem 2E (Monotonicity of $\mathcal{T}_{\mathcal{P}}$)**.** *The immediate consequence operator enjoys monotonicity in two levels:*

[c)] This is only a special case; for the full apparatus of ↑, see Definition B.4 in p. 41.

(i) *Let $\mathcal{P}$ be a definite program and $I, J \in \mathbf{2}^{\mathcal{HB}_{\mathcal{P}}}$. Then*

$$I \subseteq J \implies \mathcal{T}_{\mathcal{P}}(I) \subseteq \mathcal{T}_{\mathcal{P}}(J).$$

(ii) *Let $\mathcal{P}_1$ and $\mathcal{P}_2$ be definite programs and I a Herbrand interpretation of $\mathcal{P}_1$. Then*

$$P_1 \subseteq \mathcal{P}_2 \implies \mathcal{T}_{P_1}(I) \subseteq \mathcal{T}_{P_2}(I).$$

Proof. [Llo93] again. ▮

Theorem 2F. *Let $\mathcal{P}$ be a definite program and I a Herbrand interpretation. Then*

$$I \models \mathcal{P} \iff \mathcal{T}_{\mathcal{P}}(I) \subseteq I.$$

Proof. See [Apt90]. ▮

2.4 Examples

In this section we examine and give semantics to various definite programs.

▶ *Example 2.8.* We begin with three very simple ground programs:

$$\mathcal{P} := \left\{ p \leftarrow q \right\}, \qquad Q := \left\{ \begin{matrix} p \leftarrow q \\ p \leftarrow \end{matrix} \right\}, \qquad \text{and} \qquad R := \left\{ \begin{matrix} p \leftarrow q \\ q \leftarrow \end{matrix} \right\}.$$

Since there are neither constant nor function symbols, looking for the Herbrand universe is meaningless here. All three programs share the same Herbrand base

$$\mathcal{HB} = \{p, q\}.$$

Notice that $\mathcal{HB}$ indeed models each one of them. Their least Herbrand models are given by

$$\mathcal{M}_{\mathcal{P}} = \emptyset, \qquad \mathcal{M}_{\mathcal{Q}} = \{p\}, \qquad \text{and} \qquad \mathcal{M}_{\mathcal{R}} = \{q, p\}.$$

respectively. It is instructive to verify this using their immediate consequence operators:

$$\begin{array}{lrr} \mathcal{T}_{\mathcal{P}}(\emptyset) = \emptyset & \mathcal{T}_{\mathcal{Q}}(\emptyset) = \{p\} & \mathcal{T}_{\mathcal{R}}(\emptyset) = \{q\} \\ & \mathcal{T}_{\mathcal{Q}}(\{p\}) = \{p\} & \mathcal{T}_{\mathcal{R}}(\{q\}) = \{q, p\} \\ & & \mathcal{T}_{\mathcal{R}}(\{q, p\}) = \{q, p\}. \end{array}$$

Notice that once we have reached a fixpoint, there is no need to iterate any further: this least fixpoint *is* the least Herbrand model. ◀

▶ *Example 2.9.* Let $\mathcal{P}$ be the program

$$\mathcal{P} := \left\{ \begin{array}{l} \quad p(\mathsf{f}(X)) \leftarrow q(X, \mathsf{g}(X)) \\ q(a, \mathsf{g}(b)) \leftarrow \\ q(b, \mathsf{g}(b)) \leftarrow \end{array} \right\}.$$

We compute

$$\begin{aligned} \mathcal{U}_{\mathcal{P}} &= \{a, b, \mathsf{f}(a), \mathsf{f}(b), \mathsf{g}(a), \mathsf{g}(b), \mathsf{f}(\mathsf{f}(a)), \mathsf{f}(\mathsf{f}(b)), \mathsf{f}(\mathsf{g}(a)), \mathsf{f}(\mathsf{g}(b)), \mathsf{g}(\mathsf{f}(a)), \dots\} \\ &= \bigcup_n U_n, \quad \text{where } \left\{ \begin{array}{r} U_0 = \{a, b\} \\ U_{n+1} = \{\mathsf{f}(u) \mid u \in U_n\} \cup \{\mathsf{g}(u) \mid u \in U_n\} \end{array} \right. \\ \mathcal{HB}_{\mathcal{P}} &= \{p(u) \mid u \in \mathcal{U}_{\mathcal{P}}\} \cup \{q(u, v) \mid u, v \in \mathcal{U}_{\mathcal{P}}\}. \end{aligned}$$

Iterating $\mathcal{T}_{\mathcal{P}}$ a few times fixes a point:

$$\begin{aligned} &\mathcal{T}_{\mathcal{P}} \uparrow 0 = \emptyset \\ &\mathcal{T}_{\mathcal{P}} \uparrow 1 = \{q(a, \mathsf{g}(b)), q(b, \mathsf{g}(b))\} \\ &\mathcal{T}_{\mathcal{P}} \uparrow 2 = \{q(a, \mathsf{g}(b)), q(b, \mathsf{g}(b)), p(\mathsf{f}(b))\} \\ &\mathcal{T}_{\mathcal{P}} \uparrow 3 = \{q(a, \mathsf{g}(b)), q(b, \mathsf{g}(b)), p(\mathsf{f}(b))\} \\ &\therefore\ \mathcal{M}_{\mathcal{P}} = \{q(a, \mathsf{g}(b)), q(b, \mathsf{g}(b)), p(\mathsf{f}(b))\}. \end{aligned}$$ ◀

So far, least Herbrand models have all been finite; but not anymore—in the following example, a program consisting of just two rules and yet manages to have an infinite least Herbrand model.

▶ *Example 2.10.* Let $\mathcal{P}_{plus}$ be the following program:

$$\begin{aligned} plus(\mathsf{s}(X), Y, \mathsf{s}(Z)) &\leftarrow plus(X, Y, Z) \\ plus(\mathsf{0}, X, X) &\leftarrow . \end{aligned}$$

Logic programmers will at once recognize that this program recursively defines addition, and will hope that our denotational semantics assigns the function of addition as the meaning of *plus*. No experience with logic programming is needed to verify this; one can readily compute the Herbrand universe and base:

$$\begin{aligned} \mathcal{U}_{\mathcal{P}_{plus}} &= \{\mathsf{0}, \mathsf{s0}, \mathsf{ss0}, \mathsf{sss0}, \dots\} = \{\widehat{0}, \widehat{1}, \widehat{2}, \widehat{3}, \dots\} = \widehat{\mathbb{N}} \\ \mathcal{HB}_{\mathcal{P}_{plus}} &= \{p(u, v, w) \mid u, v, w \in \mathcal{U}_{\mathcal{P}}\}. \end{aligned}$$

We are now ready to start iterating $\mathcal{T}_{\mathcal{P}_{plus}}$:

$$\begin{aligned} &\mathcal{T}_{\mathcal{P}_{plus}} \uparrow 0 = \emptyset \\ &\mathcal{T}_{\mathcal{P}_{plus}} \uparrow 1 = \{plus(\mathsf{0}, u, u) \mid u \in \mathcal{U}_{\mathcal{P}}\} \\ &\mathcal{T}_{\mathcal{P}_{plus}} \uparrow 2 = \{plus(\mathsf{0}, u, u) \mid u \in \mathcal{U}_{\mathcal{P}}\} \cup \{plus(\mathsf{s0}, u, \mathsf{s}u) \mid u \in \mathcal{U}_{\mathcal{P}}\}. \end{aligned}$$

It is more insightful to rewrite this last equation as

$$\begin{aligned}\mathcal{T}_{\mathcal{P}_{plus}} \uparrow 2 &= \{plus(\widehat{0}, \widehat{m}, \widehat{m}) \mid m \in \mathbb{N}\} \cup \{plus(\widehat{1}, \widehat{m}, \widehat{m+1}) \mid m \in \mathbb{N}\} \\ &= \{plus(\widehat{n}, \widehat{m}, \widehat{n+m}) \mid n \in \{0,1\},\ m \in \mathbb{N}\}.\end{aligned}$$

Similarly,

$$\begin{aligned}\mathcal{T}_{\mathcal{P}_{plus}} \uparrow 3 &\overset{\cdots}{=} \{plus(\widehat{n}, \widehat{m}, \widehat{n+m}) \mid n \in \{0,1,2\},\ m \in \mathbb{N}\} \\ &\vdots \\ \mathcal{T}_{\mathcal{P}_{plus}} \uparrow \omega &= \{plus(\widehat{n}, \widehat{m}, \widehat{n+m}) \mid n, m \in \mathbb{N}\} \\ \mathcal{T}_{\mathcal{P}_{plus}} \uparrow (\omega + 1) &= \{plus(\widehat{n}, \widehat{m}, \widehat{n+m}) \mid n, m \in \mathbb{N}\},\end{aligned}$$

and thus we have reached a fixpoint in no more than ω steps. We therefore set

$$\begin{aligned}\mathcal{M}_{\mathcal{P}_{plus}} &= \{plus(\widehat{n}, \widehat{m}, \widehat{n+m}) \mid n, m \in \mathbb{N}\} \\ &= \{plus(\widehat{n}, \widehat{m}, \widehat{k}) \mid k = n + m,\ n, m, k \in \mathbb{N}\},\end{aligned}$$

which is indeed the intended model of $\mathcal{P}_{plus}$. ◀

2.5 Infinite programs

There are two obvious ways that we can generalize the notion of a definite program with respect to cardinality:

- Drop the restriction that a definite program is a *finite* set of definite rules. Such a program is called an *infinite program*.

infinite program

- Generalize the notion of a rule to include infinitely long formulæ.

Occasionally I may call infinite programs *tall*, and programs with infinitely long rules *wide*, even though this is not standard terminology.

tall program

wide program

▶ *Example 2.11.* The following two programs are infinite, definite programs:

$$\mathcal{P}_1 \coloneqq \left\{\begin{aligned} p_0 &\leftarrow \\ p_1 &\leftarrow p_0 \\ p_2 &\leftarrow p_1 \\ p_3 &\leftarrow p_2 \\ &\vdots \end{aligned}\right\}, \qquad \mathcal{P}_2 \coloneqq \left\{\begin{aligned} nat(\mathsf{0}) &\leftarrow \\ nat(\mathsf{s}(\mathsf{0})) &\leftarrow nat(\mathsf{0}) \\ nat(\mathsf{s}^2(\mathsf{0})) &\leftarrow nat(\mathsf{s}(\mathsf{0})) \\ nat(\mathsf{s}^3(\mathsf{0})) &\leftarrow nat(\mathsf{s}^2(\mathsf{0})) \\ &\vdots \end{aligned}\right\}.$$

Note that the underlying language of the first program includes an infinite number of relational symbols, while the second one is finite. ◀

▶ *Example 2.12.* Let $\mathcal{P}_3$ be the following program:

$$\mathcal{P}_3 \coloneqq \{ q \leftarrow p_0, p_1, p_2, p_3, \dots \} .$$

It is a finite program that consists of a single infinite rule. It is immediately clear that $\mathcal{HB}_{\mathcal{P}_3} = \{q, p_0, p_1, p_2, \dots\}$ and $\mathcal{M}_{\mathcal{P}_3} = \emptyset$. ◀

Usually we tend to ignore variables, functions and constants of finite programs and choose to work with infinite versions that more or less capture the same rules. This is done for our convenience in working formally with semantics.

We may as well choose to drop both restrictions and consider programs that are both tall and wide. But hold your excitement—such totally unrestricted and untamed programs are beyond the scope of this text. Still, how could we construct their intended model? Once some obvious concept generalizations are agreed upon, we can work in similar ways, as the next example illustrates.

▶ *Example 2.13.* Let's put the tall program $\mathcal{P}_1$ and the wide program $\mathcal{P}_3$ of the previous examples together:

$$\mathcal{P} \coloneqq \mathcal{P}_1 \cup \mathcal{P}_3 = \left\{ \begin{array}{l} \quad q \leftarrow p_0, p_1, p_2, p_3, \dots \\ p_0 \leftarrow \\ p_1 \leftarrow p_0 \\ p_2 \leftarrow p_1 \\ p_3 \leftarrow p_2 \\ \quad \vdots \end{array} \right\} .$$

We can even use the (generalized) $\mathcal{T}_\mathcal{P}$ to yield the intended model:

$$\begin{aligned} \mathcal{T}_\mathcal{P} \uparrow 0 &= \emptyset \\ \mathcal{T}_\mathcal{P} \uparrow 1 &= \{p_0\} \\ \mathcal{T}_\mathcal{P} \uparrow 2 &= \{p_0, p_1\} \\ &\vdots \\ \mathcal{T}_\mathcal{P} \uparrow \omega &= \{p_0, p_1, p_2, \dots\} \\ \mathcal{T}_\mathcal{P} \uparrow (\omega + 1) &= \{q, p_0, p_1, p_2, \dots\} \\ \mathcal{T}_\mathcal{P} \uparrow (\omega + 2) &= \{q, p_0, p_1, p_2, \dots\} \\ &\doteq \mathcal{M}_\mathcal{P} . \end{aligned}$$

Notice, however, that it takes $\omega + 1$ steps to reach this fixpoint. ◀

2.6 Historical remarks

Why does every definite program have a unique minimal Herbrand model? It was proved in van Emden and Kowalski's landmark paper [vEK76]. They showed that the least fixed point of $\mathcal{T}_{\mathcal{P}}$ coincides with the model-theoretic meaning of definite programs. Apt and van Emden in [AvE82] showed a similar result, relating the greatest fixed point of $\mathcal{T}_{\mathcal{P}}$ to the subset of the Herbrand base whose members finitely fail (i.e., they have a finite SLD-tree without any refutations). The name "immediate consequence operator" was coined by Clark, in [Cla79].

Summary

In this chapter we dealt with *definite programs* and their denotational semantics using *Herbrand models*, and by singling out the $\subseteq$-least one as the meaning of a program. We saw that by using $\mathcal{T}_{\mathcal{P}}$ we are able to construct, for any such program $\mathcal{P}$, its least Herbrand model $\mathcal{M}_{\mathcal{P}}$ as the *least fixed point* of $\mathcal{T}_{\mathcal{P}}$.

It is important to note that by restricting our programming language to definite clauses, we enjoy *monotonicity*: if we are able to infer p from a program $\mathcal{P}$, then we will still be able to infer it no matter what extra rules may be added to $\mathcal{P}$. Also, the exact same restriction guarantees that each program will definitely have a model, and specifically a least Herbrand model.

Next we consider *negation*, the issues that are brought along with it, as well as various alternative solutions for dealing with them.

«I've done everything the Bible
says—even the stuff that contradicts the
other stuff!»

NED FLANDERS

Chapter 3

Logic programs with negation

So far so good. As we have seen, for definite programs everything is well-established and quite unquestionable. Once we allow negations in bodies, *things become spooky*. We lose monotonicity, in the sense that the addition of a rule may forbid us to infer a formula that we were able to infer before this addition.

In this chapter we investigate various different approaches to giving semantics to logic programs with negation. There is no such thing as a best approach, although there certainly is a great amount of bias in favor of the *well-founded semantics*. Its main rival, the *stable models semantics*, is also satisfactory, though not as close to the logic programming paradigm; fruitful as it was, it also gave rise to answer set programming, which lies beyond the scope of this text.

3.1 Introducing negation

Consider the program $\mathcal{P} \coloneqq \{p \leftarrow \neg q\}$. It has three models, viz. $\{p\}$, $\{q\}$ and $\{p, q\}$, none of which is *least* (however, the first two are *minimal*). It is therefore immediately clear that we cannot hope for the methods that we investigated in the previous chapter to automagically work in programs with negation. For even such a simple program as this, $\mathcal{T}_{\mathcal{P}}$ fails to be monotone:

$$\emptyset \subseteq \{q\}, \quad \text{but} \quad \{p\} = \mathcal{T}_{\mathcal{P}}(\emptyset) \nsubseteq \mathcal{T}_{\mathcal{P}}(\{q\}) = \emptyset.$$

A non-monotone $\mathcal{T} : D \to D$ can't be continuous and therefore we can no longer apply the Knaster–Tarski theorem—so long, fixpoints![a)]

The road so far...

Up to now, Herbrand interpretations mapped elements of the Herbrand base to the truth-value set $\{\mathbf{T}, \mathbf{F}\}$, and so it made sense to identify them with the sets

[a)] This example program $\mathcal{P}$ happens to behave well: $\mathcal{T}_{\mathcal{P}}$ fixes a point in just 1 step. Think of the program $\{p \leftarrow \neg p\}$ if you are not yet convinced.

valuations

of the ground atoms that they satisfy. It is time to remember that Herbrand interpretations really are functions, which we call *valuations*. Their codomain is essentially the truth-value space of the semantics.

It is beneficial to impose some kind of ordering on this truth-value set and on the set of valuations, transforming them into richer algebraic structures. As we have seen on LP programs, the set of Herbrand interpretations was ordered under set-inclusion to become a complete lattice, which granted us machinery such as the Knaster–Tarski fixpoint theorem, enabling us to construct using $\mathcal{T}_{\mathcal{P}}$ the very meaning of $\mathcal{P}$ as $[\![\mathcal{P}]\!] \coloneqq \operatorname{lfp} \mathcal{T}_{\mathcal{P}}$.

3.2 Assumptions related with negation

It becomes apparent that there are various different notions of negation that are worthy of our concern. Negation from the classical logic point-of-view (viz. "$\neg$"), is well-known. The negation that appears on programs as "$\sim$", can be interpreted in many different ways. For instance, $\sim A$ may mean that we are not able to infer A. If our inference algorithm is complete (as is the case for definite programs with SLD-Resolution) these two nots will coincide—but this is not always the case.

negation as failure
closed world assumption

In this text, we focus on *negation as failure* (NAF), which states that one can derive $\sim p$ if deriving p fails. A closely related notion, is the *closed world assumption* (CWA), which one would also call "negation as *infinite* failure".

> REMARK 3.1. Even in simple cases, the set of consequences wrt to CWA may be inconsistent;* e.g.,
>
> $$\{p \vee q\} \mathrel{|\!\!\!\underset{\mathsf{CWA}}{=\!=}} \underbrace{\{p \vee q, \neg p, \neg q\}}_{\text{inconsistent}}.$$
>
> *7,761!

One of the main limitations of negation as failure is that we cannot make a query like «Is there an X such that $\neg p(X)$?». This shortcoming is attacked by *constructive negation*, about which the interested reader should consult [Dra95].

Closed world assumption is due to Reiter (see [Rei78]). Clark suggested the "negation as finite failure" rule in [Cla78] and proved its soundness. A few years later, in [JlLL83], it was also shown to be complete.

3.3 LPN programs

LPN rule

Definition 3.1. An *LPN rule* is a formula of the form

$$A \leftarrow L_1, \ldots, L_n,$$

where A is atomic and each L_i is a literal; or equivalently, of the form

$$A \leftarrow B_1, \ldots, B_n, \sim C_1, \ldots, \sim C_m, \tag{LPN}$$

LPN program

where A and all the B_i's and the C_j's are atomic. An *LPN program* is a finite set of LPN rules.[b)]

$\mathrm{head}(r)$ $\mathrm{body}(r)$ $\mathbf{B}^+(r)$ $\mathbf{B}^-(r)$ $\mathbf{B}(r)$ $\mathrm{Neg}(\mathcal{P})$

Definition 3.2. Let r be a rule of the form (LPN). Then we define

$$\begin{aligned}
\mathrm{head}(r) &=_{\mathrm{df}} A && \text{(the head atom)}\\
\mathrm{body}(r) &=_{\mathrm{df}} \{B_1,\dots,B_n,{\sim}C_1,\dots,{\sim}C_m\} && \text{(the body literals)}\\
\mathbf{B}^+(r) &=_{\mathrm{df}} \{B_1,\dots,B_n\} && \text{(the positive body atoms)}\\
\mathbf{B}^-(r) &=_{\mathrm{df}} \{C_1,\dots,C_m\} && \text{(the negative body atoms)}\\
\mathbf{B}(r) &=_{\mathrm{df}} \mathbf{B}^+(r)\cup\mathbf{B}^-(r) && \text{(the body atoms)}\\
\mathrm{Neg}(\mathcal{P}) &=_{\mathrm{df}} \bigcup_{r\in\mathcal{P}} \mathbf{B}^-(r) && \text{(the negative atoms).}
\end{aligned}$$

3.4 Restricting programs

We start by the obvious generalization of $\mathcal{T}_{\mathcal{P}}$ to LPN programs. The concept is so similar that we continue to use the same symbol.

immediate consequence operator

Definition 3.3. The *immediate consequence operator* of a LPN program $\mathcal{P}$ is the operator $\mathcal{T}_{\mathcal{P}}$ defined by

$$\mathcal{T}_{\mathcal{P}}(I) =_{\mathrm{df}} \{A \mid (\exists\phi\in\mathcal{P})[A=\mathrm{head}(\phi)\ \&\ I\models\mathrm{body}(\phi)]\}.$$

In [ABW88], Apt, Blair, and Walker introduced *stratified programs*. Working independently, Van Gelder defined the same class of programs in 1989.

positively refers to $\xrightarrow{+}$ negatively refers to $\xrightarrow{-}$

Definition 3.4. Let $\mathcal{P}$ be an LPN program. Then A *positively refers to* B if $A\xrightarrow{+}B$, and A *negatively refers to* B if $A\xrightarrow{-}B$, where

$$\begin{aligned}
A\xrightarrow{+}B &\overset{\mathrm{df}}{\iff} (\exists\phi\in\mathrm{Ground}(\mathcal{P}))[A=\mathrm{head}(\phi)\ \&\ B\in\mathrm{body}(\phi)]\\
A\xrightarrow{-}B &\overset{\mathrm{df}}{\iff} (\exists\phi\in\mathrm{Ground}(\mathcal{P}))[A=\mathrm{head}(\phi)\ \&\ {\sim}B\in\mathrm{body}(\phi)].
\end{aligned}$$

We also define $\longrightarrow$ as their union $\xrightarrow{+}\cup\xrightarrow{-}$, i.e.,

$$A\longrightarrow B \overset{\mathrm{df}}{\iff} A\xrightarrow{+}B \text{ or } A\xrightarrow{-}B,$$

refers to $\longrightarrow^*$ depends on

and say that A *refers to* B in that case. If the transitive closure $A\longrightarrow^* B$ holds, we say that A *depends on* B.

► *Example 3.1.* Here are some LPN programs:

$$\mathcal{P} \coloneqq \left\{\begin{array}{c} p(X)\leftarrow{\sim}q(X)\\ q(Y)\leftarrow{\sim}r(Y),q(\mathsf{s}(Y))\\ r(b)\leftarrow \end{array}\right\},\quad \mathcal{Q}\coloneqq\left\{\begin{array}{c} even(\mathsf{0})\leftarrow\\ even(\mathsf{s}(X))\leftarrow{\sim}even(X)\end{array}\right\}.$$

[b)] LPN programs are usually called *general*, and sometimes even *normal*, though the latter is also used for "disjunctionless" in the literature.

Some example dependencies, then, are:

$$\begin{aligned}
p(b) &\xrightarrow{-} q(b) \\
q(b) &\xrightarrow{-} r(b) \\
q(\mathsf{s}b) &\xrightarrow{+} q(\mathsf{ss}b) & \mathit{even}(\widehat{n+1}) &\xrightarrow{-} \mathit{even}(\widehat{n}) & &(n \in \mathbb{N}) \\
q(\mathsf{ss}b) &\to q(\mathsf{sss}b) & \mathit{even}(\widehat{n}) &\to^{*} \mathit{even}(\widehat{k}) & &(k, n \in \mathbb{N},\, k \leq n). \\
p(b) &\to^{*} r(b) \\
p(b) &\to^{*} q(\mathsf{s}b)
\end{aligned}$$

◀

$\mathcal{D}_\mathcal{P}$
dependency graph

Definition 3.5. The digraph $\mathcal{D}_\mathcal{P} =_{\mathrm{df}} \langle \mathcal{HB}_\mathcal{P}; \to \rangle$, is called the *dependency graph* of $\mathcal{P}$.

stratified program

Definition 3.6. An LPN program $\mathcal{P}$ is *stratified* if $\mathcal{D}_\mathcal{P}$ does not contain a cycle with a negative arc.

stratification

Definition 3.7. We call a partition $\mathcal{P} = P_1 \uplus \cdots \uplus P_n$ a *stratification* of $\mathcal{P}$, if for every $i \in \{1, \ldots, n\}$,

(i) $r \in \mathrm{Literals}(P_i) \implies \mathbf{def}_\mathcal{P}(r) \subseteq \bigcup_{j \leq i} P_j$,

(ii) $\sim r \in \mathrm{Literals}(\mathrm{Ground}(P_i)) \implies \mathbf{def}_\mathcal{P}(r) \subseteq \bigcup_{j < i} P_j$.

▶ *Example 3.2.* Let $\mathcal{P}$ be the LPN program

$$\mathcal{P} := \left\{ \begin{array}{l} p(X) \leftarrow \sim q(X) \\ q(Y) \leftarrow \sim r(Y), q(\mathsf{s}(Y)) \end{array} \right\}.$$

The following partition of the relations of $\mathcal{P}$ is a stratification

$$\begin{aligned}
P_0 &= \{r\} \\
P_1 &= \{q\} \\
P_2 &= \{p\}.
\end{aligned}$$

◀

Once we have restricted ourselves to only consider stratified programs, we can give semantics to them using *standard models*. Similar but more general restrictions have been investigated, and fancy names have been given to the corresponding models that provide the semantics.

Przymusinski and Przymusinska defined the *perfect models* for *locally stratified programs* in [PP90]. They can be further extended to

- stable model semantics,
- weakly perfect model semantics for *weakly stratified programs*,
- well-founded semantics.

Bidoit and Froidevaux in [BF91]: *General logical databases and programs: Default logic semantics and stratification*, define the notion of *effective stratification*, extending even more the weakly stratified semantics. This is close to the well-founded semantics.

3.5 Stable model semantics

Proposed by Gelfond and Lifschitz in [GL88], stable model semantics coincide with the perfect ones when restricted to locally stratified programs. The models are 2-valued, but each program might have none, one, or many stable models.

3.6 Well-founded semantics

This is the widely accepted approach to semantics for LPN programs. The well-founded model coincides with the standard model of stratified programs, and with the perfect model of locally stratified ones. It is three-valued, and thus it applies to arbitrary LPN programs.

Przymusinski introduced *stationary models* by generalizing two-valued stable models to three-valued ones. He singled out a minimal model, which is now called the *well-founded model*, for which various alternative characterizations and constructions are known, also thanks to Van Gelder, Ross and Schlipf in [VGRS91].

stationary model

well-founded model

> Remark 3.2. If a ground atom is true in the well-founded model of $\mathcal{P}$, then it is true under every stable model of P. One may be tempted to think that the converse holds as well, so here is a counterexample that proves otherwise. Let
>
> $$\mathcal{P} := \left\{\begin{array}{l} p \leftarrow {\sim}q \\ q \leftarrow {\sim}p \\ r \leftarrow p \\ r \leftarrow q \end{array}\right\}.$$
>
> This program has two stable models: $\{r, p\}$ and $\{r, q\}$. Even though r belongs to both of them, its value is "unknown" in the well-founded model.
>
> Remark 3.3. If $\mathcal{P}$ happens to have a *two*-valued, well-founded model, then $\mathcal{P}$ will possess a *unique* stable model, and those two models coincide.

3.7 Infinite-valued minimum model semantics

The well-founded semantics are not purely model theoretic. To ameliorate this situation, Rondogiannis and Wadge suggested a shift in the underlying logic. In [RW05b] introduced a new, infinite-valued logic and built on it purely model-theoretic semantics which refined the well-founded semantics.

3.8 Program completions

Clark's program completion, defined in [Cla78], essentially transforms a program by considering ← to mean ↔, and grouping all rules with a common head in a single rule, where the corresponding bodies are joined disjunctively. The program is also extended to include certain *free equality axioms* to enable the correct reasoning to take place. This yields a two-valued model semantics.

With negation, Clark's model can behave rather irrationally. For example, the consistent LPN program $\{p \leftarrow \sim p\}$, has an inconsistent completion, and so everything may be derived from it.

Fitting and Kunen extended Clark's two-valued completion to three valued models, amending this problematic behaviour. Fitting called this approach the Kripke–Kleene semantics. In [Kun89], it is shown that 2- and 3-valued models coincide for programs that are *strict*.

Comparing this 3-valued approach to the well-founded one, the essential difference boils down to the underlying logic. While the well-founded model considers $\mathbf{F} < \mathbf{0} < \mathbf{T}$, here Kleene's strong three valued logic is used instead. This makes $\perp$ the preferred truth value. For instance, consider the following simple LP program:

$$\left\{\, p \leftarrow p \,\right\}.$$

According to the well-founded semantics, p is $\mathbf{F}$, while in the three-valued completion models, p is $\perp$.

3.9 Four-valued semantics

Fitting used Belnap's logic and machinery from bilattices to provide 4-valued, stable model semantics to LPN programs. The fourth value represents an "overdefined" element for which we have reasons to consider it both true and false. This, this truth value can be seen as a representation of inconsistencies. The motivation behind this approach to semantics is that even though our program may contain contradictory facts regarding a subject, it may still be well-behaved in other parts. For more information regarding this approach, see [Fit90, Fit99].

Summary

We have seen various approaches to deal with LPN programs. If we wish to remain in 2-valued classical logic, we must either sacrifice the uniqueness of semantic models or narrow-down the notion of programs that we give semantics to. If we wish to deal with all programs in a unified way, a third value is needed. The well-founded model, especially as constructed by the infinite-valued model semantics, is the standard semantics that we will keep in mind for LPN programs. Suggested reading: [Fit99], [NM00], [RW05b].

«In re mathematica ars proponendi pluris
facienda est quam solvendi.»
Georg Cantor

Chapter 4

Disjunctive logic programs

If we allow disjunctions in a program, we no longer have a *least* Herbrand model. There may be more than one *minimal* models, and one approach to denotational semantics is exactly this: it is proved that each program $\mathcal{P}$ has a *nonempty set* of minimal models, and this very set may be dubbed "the meaning of $\mathcal{P}$".

Another approach involves extending the concept of a Herbrand base, so that we are able to assign a unique, least *model state* as the semantics of a DLP program.

The definitions and results that follow are mainly due to Lobo, Minker and Rajasekar, and can be collectively found in [LMR92].

4.1 Introducing disjunction

As we certainly know by now, a definite clause is a disjunction of literals, in which exactly one is positive. When seen from a logic programming aspect, we speak of a definite (LP) rule, and think of it as "$a \leftarrow b_1, \dots, b_n$". Since there is only one positive literal, there is only one way to bring a definite clause into that form. On the other hand, an LPN rule essentially is a disjunction of literals in which at least one must be positive. But this is not enough to uniquely determine a corresponding form like (LPN): a positive literal must be designated as the head. It is essentially this last restriction that we drop when dealing with disjunctive programs.

4.2 DLP programs

Definition 4.1. A *DLP rule* is a rule of the form

DLP rule

$$a_1 \vee \cdots \vee a_k \leftarrow b_1, \dots, b_n. \tag{DLP}$$

It is probably already apparent by now, that if we hope to pinpoint a unique "model" as the meaning of a disjunctive program $\mathcal{P}$, we should choose its elements from a set that's more general than $\mathcal{HB}_{\mathcal{P}}$. For this reason, we introduce the following set.

disjunctive Herbrand base
$\mathcal{DHB}_{\mathcal{P}}$

Definition 4.2. Let $\mathcal{P}$ be a DLP program. Its *disjunctive Herbrand base* is the set $\mathcal{DHB}_{\mathcal{P}}$ of all positive disjunctive ground clauses which can be formed by using *distinct* ground atoms from $\mathcal{HB}_{\mathcal{P}}$, such that *no two logically equivalent clauses belong to it.*

> ☡ Remark 4.1. Is the set $\mathcal{DHB}_{\mathcal{P}}$ well-defined? No, it isn't. In fact, the last restriction in the above definition does not uniquely determine which of the logically equivalent clauses belongs to it. We can either impose an ordering on the underlying language and always pick the least, or allow ourselves the luxury of abusing the notation "$C \in \mathcal{DHB}_{\mathcal{P}}$" to stand for the claim "either C or some syntactic variant of C belongs to $\mathcal{DHB}_{\mathcal{P}}$".

expansion
$\exp(S)$

Definition 4.3. Let $\mathcal{P}$ be a DLP program and S a set of positive ground clauses in P. The *expansion* of S is the set $\exp(S)$ defined by

$$\exp(S) = \{C \in \mathcal{DHB}_{\mathcal{P}} \mid C \in S \text{ or } \exists C' \in S \text{ such that } C' \text{ is a subclause of } C\}.$$

Herbrand state
expanded Herbrand state

Definition 4.4. A set $S \subseteq \mathcal{DHB}_{\mathcal{P}}$ is called a *Herbrand state* for $\mathcal{P}$. If additionally $S = \exp(S)$, we call it an *expanded Herbrand state*.

model-state

Definition 4.5. A *model-state* for a set S of closed formulæ S of $\mathcal{L}_{\mathcal{P}}$ is an expanded state St such that

(i) every Herbrand model of St is a Herbrand model of S; and

(ii) every minimal Herbrand model of S is contained in *some* minimal Herbrand model of St.

Minimal models semantics

Theorem 4A. *Let $\mathcal{P}$ be a DLP program. A positive ground clause C is a logical consequence of $\mathcal{P}$ iff C is true in* every minimal *Herbrand model of $\mathcal{P}$.*

Proof. See [LMR92]. ∎

► *Example 4.1.* Let $\mathcal{P}$ be the disjunctive program

$$\mathcal{P} := \left\{ \begin{array}{r} a \vee b \leftarrow \\ c \leftarrow \end{array} \right\}.$$

Then $\mathcal{M}_{\mathcal{P}}^{min} = \{\{a, c\}, \{b, c\}\}$, and the clause $a \vee b$ is a consequence of $\mathcal{P}$, since it is true wrt both minimal models of $\mathcal{P}$. On the other hand, b isn't. ◄

Model state semantics

The idea here is that a model-state for a DLP program $\mathcal{P}$ must already contain every minimal model of $\mathcal{P}$. By considering model-states instead of sets of minimal models, we regain an intersection property, perhaps at the cost of elegance.

Theorem 4B (Model-state intersection property)**.** *Let $\mathcal{P}$ be a DLP program and $\{W_i\}_{i\in\mathbb{N}}$ a non-empty family of model-states of $\mathcal{P}$. Then $\bigcap_{i\in\mathbb{N}} W_i$ is also model-state of $\mathcal{P}$.*

Proof. See [LMR92]. ▮

Observing that $\mathcal{DHB}_\mathcal{P}$ is always a model-state of $\mathcal{P}$, we arrive at the following pleasant conclusion:

Corollary 4C. *Every DLP program $\mathcal{P}$ has a* unique*, least model-state $\mathcal{M}^\ell_\mathcal{P}$.*

Theorem 4D. *Let $\mathcal{P}$ be a DLP program. Then,*

$$\mathcal{M}^\ell_\mathcal{P} = \{C \in \mathcal{DHB}_\mathcal{P} \mid P \models C\}.$$

Proof. Also in [LMR92]. ▮

The two different points of view are hereby connected: *a clause is in the least model-state iff it is true in every minimal model.* Formally:

Theorem 4E. *Let $\mathcal{P}$ be a DLP program, and $C \in \mathcal{DHB}_\mathcal{P}$. Then,*

$$C \in \mathcal{M}^\ell_\mathcal{P} \iff (\forall M \in \mathfrak{M}^{min}_\mathcal{P})\, M \models C.$$

Proof. Immediate from Theorem 4D and Theorem 4A. ▮

4.3 Answer set programming

Bonatti, Calimeri, Leone and Ricca in [BCLR10] present denotational semantics for disjunctive logic programming under the stable model semantics. But this deviates enough from the practices of logic programming to be considered as a different programming paradigm!

4.4 DLPN programs—the best (worst) of two worlds

Definition 4.6. A *DLPN rule* is a rule of the form DLPN rule

$$a_1 \vee \cdots \vee a_k \leftarrow b_1, \ldots, b_n, {\sim}c_1, \ldots, {\sim}c_m. \qquad \text{(DLPN)}$$

Sets of infinite-valued, minimal models

L_∞^{min}

Cabalar, Pearce, Rondogiannis and Wadge in [CPRW07] present a model-theoretic semantics for disjunctive logic programs with negation, they call L_∞^{min}. They show that every finite, propositional DLPN program has a nonempty, finite set of minimal infinite-valued models. Furthermore, a bound in this set's size is calculated, which leads to a brute-force algorithm that computes all those infinite-valued minimal models in finite time.

This approach combines the infinite-valued semantics that we've dealt with for LPN and the minimal model semantics for DLP. In fact, it generalizes both methods, in the sense that if we restrict it to either LPN or DLP, we obtain the desired semantics.

Summary

Disjunctions allow uncertainty to be naturally expressed in logic programs, and is therefore a useful language feature to have available. Of course, this complicates the semantics involved, which once we also allow negations in the programs seem abysmal.

Luckily, for the case of DLP we have the minimal model semantics, while for DLPN (and therefore for every logic programming language considered in this text!) the infinite-valued minimal model semantics.

«I'm tired of hearing about money, money, money, money, money. I just want to play the game, drink Pepsi, wear Reebok.»

SHAQUILLE O'NEAL

Chapter 5

Game semantics

The idea of using game-theoretic concepts for the study of Logic and its semantics, goes back to Hintikka, Lorenzen and Lorenz. Games have also been used extensively to provide semantics for (functional) programming languages and the reader is referred to [Abr97] for more information regarding this subject.

It therefore seems natural to investigate what kind of games we can define and play for logic programming.

5.1 Playing with definite programs

Our story begins with van Emden in [vE86], who uses games to give semantics to a fuzzy version of logic programming, using fuzzy logic and sets.

It was Di Cosmo, Loddo and Nicolet who studied definite programs extensively, in their [DCLN98]. Their approach is outlined in this section.

The idea behind game semantics is that once we are given a program $\mathcal{P}$ and a ground atom p, a game will tell us p's truth value. Each of the four games presented here has its own set of rules, and its own game-theoretic way to answer queries. However, they all share the following:

- Two players: Player I (who plays first) and Player II.
- The game has *perfect information*: at any moment, both players know all moves that have taken place.

perfect information

- One of them is the *Believer*, the other one is the *Doubter*.
- A player who can't make a legal move, loses.
- Doubter has the "benefit of the doubt", i.e., a Doubter who can always doubt, wins.

Rules of the LP game

LP1. Player I is the Believer, and begins the game by playing the goal clause "$\leftarrow p$".

LP2. If the previous move was a clause, the next move must be one of the conjuncts in the clause's body.

LP3. If the previous move was an atom, the next move must be a rule with that atom as head.

Defining semantics from the LP game

Defining semantics out of this game is done in a quite natural way.

- The goal $\leftarrow p$ succeeds, if Player I (the Believer) has a winning strategy.
- Otherwise, it fails.

Since we are using a two-valued classical logic, these possibilities correspond to the two truth values of our underlying logic.

5.2 Playing with negations

In [RW05a], Rondogiannis and Wadge define a new game that is capable of dealing with LPN programs. Its semantics are equivalent to the well-founded model. The key idea behind this game is that one can believe $\sim p$ by doubting p. This way, the rôles of the players switch.

Rules of the LPN game

For a program $\mathcal{P}$ and a goal clause $G = \leftarrow p$, the rules of the game are the following:

LPN1. Player I starts as the Believer, by playing "$\leftarrow p$".

LPN2. If the previous move was a clause, the next move is one of the literals in the body of that clause.

LPN3. If the previous move was a positive literal p, the next move is a rule with head p.

LPN4. If the previous move was a negative literal $\sim p$, the next move has to be p (*rôle-switch* move).

Defining semantics from the LPN game

As we have seen in the LP game, the Doubter has the "benefit of the doubt"; therefore, an infinite LP game will always have a winner. On the other hand, an infinite LPN game might involve infinitely many rôle changes between the players; and this is exactly what the **0** truth value represents. The truth assignment is the following:

- p is $\mathbf{T}$, if Player I (the *initial* Believer) has a winning strategy.
- p is $\mathbf{F}$, if Player II (the *initial* Doubter) has a winning strategy.

- p is $\mathbf{0}$, if each player has a strategy, which, when played against any strategy of the opponent, it can *at least ensure a tie*.

It was proved in [GRW08] that the semantics provided by the LPN game coincide with the infinite-valued model semantics of §3.7.[a]

5.3 Playing with disjunctions

To deal with disjunctions, one should adjust the rules of the game accordingly. Here we allow a disjunction of atomic formulæ to be a single goal.

Rules of the DLP game

DLP1: Player I is the Believer, and starts the game with the goal clause

$$\leftarrow p_1 \vee \cdots \vee p_n.$$

DLP2: If the previous move was a clause, the next move must be one of the conjuncts of this clause's body.

DLP3: If the previous move was a disjunction, the next move must be a set of rules, whose union of heads is a subset of the disjuncts just played (*combo* move).

This set of rules, is played as a *single rule*, with the disjunction union of heads as a head, and with the disjunction of the bodies, reformatted into CNF.

To clarify with an example, suppose that the previous move was $a \vee b \vee c$. Since $\{a, b\} \subseteq \{a, b, c\}$, the current move can be

$$\left\{\begin{matrix} a \leftarrow p, r \\ b \leftarrow q \end{matrix}\right\},$$

which becomes the single rule $a \vee b \leftarrow (p, r) \vee q$. Reformatting its body into CNF, the move is finally played as:

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

> ☡ Remark 5.1. Note that **DLP2** forces the player to choose one of the *conjuncts* of the previously played rule's body. There is nothing bogus if this body is a disjunction, as in this case this whole disjunction can be seen as a single conjunct of a (rather trivial) conjunction. For example, if the last move was $p \leftarrow q \vee r$, the following move must necessarily be $q \vee r$.

[a] Actually, a slightly different mechanism is used in their game, that also involves pay-off, and where the score determines the exact "subscript" ordinal of the exact value.

▶ *Example 5.1.* Given the program

$$\mathcal{P} := \left\{ \begin{array}{r} p \leftarrow a \\ p \leftarrow b \\ p \leftarrow c \\ a \vee c \leftarrow t \\ t \leftarrow . \end{array} \right\}$$

and the goal clause $\leftarrow p$, the following is an example game between the two players:

Player I	Player II
$\leftarrow p$	p
$p \leftarrow a \vee c$	$a \vee c$
$a \vee c \leftarrow t$	t
$t \leftarrow .$	Loser!

Note the second move of Player I, which is a combo move which resulted from the two rules "$p \leftarrow a$" and "$p \leftarrow c$".

A bad Player I could have played the combo "$p \leftarrow a \vee b$" (using which rules?) instead. Player II would then respond with "$a \vee b$", a move that would grand him victory, since there are no rules whose heads form a subset of $\{a, b\}$. ◀

Defining semantics out of the DLP game

Having defined a way to obtain semantics out of the LP game, it is straightforward to extend it to DLP games: we define exactly the same semantics!

5.4 Playing with negations and disjunctions

Rules of the DLPN game

DLPN1: Player I starts as the Believer, by playing "$\leftarrow p$".

DLPN2: If the previous move was a clause, the next move must be one of the conjuncts of this clause's body.

DLPN3: If the previous move was a disjunction

$$p_1 \vee \cdots \vee p_n \vee {\sim}q_1 \vee \cdots \vee {\sim}q_m,$$

the next move may be one of the following:

(i) a set of rules with heads in $\{p_1, \ldots, p_n\}$ (*combo*),

(ii) one of the literals q_i (*rôle-switch*).

As in the case of the DLP game, the combo move is played as a single rule, whose body is transformed into CNF by considering $\sim$ to be $\neg$.

Defining semantics out of the DLPN game

This time we use the same way to define semantics as we did for the LPN game. Again, the model we end up with is a three-valued one, as was the case with the well-founded model.

5.5 Future work

In lack of something as evident as the least Herbrand model for LP, and the well-founded model for LPN, it is hard to establish the correctness of the semantics provided by the DLP and the DLPN games. Nevertheless, there are a couple of conjectures that I currently investigate:

Conjecture 1 (DLP)**.** *Given a DLP program $\mathcal{P}$, if Player I has a winning strategy in the DLP game with the goal "$\leftarrow q_1 \vee \cdots \vee q_n$" then p is true in every minimal model of $\mathcal{P}'$, where*

$$\mathcal{P}' := \mathcal{P} \cup \{p \leftarrow q_1 \vee \cdots \vee q_n\},$$

and p is some fresh relation symbol, not occuring in $\mathcal{P}$.

Conversely, if there exists a minimal model of $\mathcal{P}'$ in which p is false, there is no winning strategy for Player I with that goal.

Conjecture 2 (DLPN)**.** *Given a DLPN program $\mathcal{P}$, if Player I has a winning strategy in the DLP game for the goal "$\leftarrow q_1 \vee \cdots \vee q_n \vee {\sim}r_1 \cdots \vee {\sim}r_m$" then p is true according to the infinite-valued minimal model semantics, where*

$$\mathcal{P}' := \mathcal{P} \cup \{p \leftarrow q_1 \vee \cdots \vee q_n \vee {\sim}r_1 \cdots \vee {\sim}r_m\},$$

and p is some fresh relation symbol, not occuring in $\mathcal{P}$.

Conversely, if there exists a minimal model of $\mathcal{P}'$ in which p is false, there is no winning strategy for Player I with that goal.

Summary

In this chapter, four games were investigated. Regarding the first two (LP and LPN), it was proved that they correspond to the semantics that we hoped: the least Herbrand model and the well-founded model respectively.

On the other hand, there is nothing as well-established as those semantics for the cases of DLP and DLPN. Instead, I have tried to define them in such a way so that they fit in harmony with the existing games and complete the picture, and therefore convince the reader of their "correctness".

Finally, two conjectures currently under investigation by the author were stated to conclude this chapter.

-

Appendices

> «. . . en les appliquant aux questions les plus importantes de la vie, qui ne sont en effet, pour la plupart, que des problèmes de probabilité.»
>
> Pierre Simon de Laplace

Appendix A

Further considerations

In this appendix, several different approaches to denotational semantics of logic programming are outlined along with some brief, relevant ideas. I had been working on this material until the point I met the LP and LPN games and focused on defining new ones that would be able to deal with disjunctive programs. However, it seemed unfair to leave these parts completely out of my thesis, so I include some of my notes, mostly as a reminder that I owe them further investigation.

A.1 Probabilistic semantics

Some probability theory comes into play here: the truth-value space is the unit interval $[0, 1]$, where each value represents the probability that a formula is true.

Extensions of the programming language

We allow numeric values in $[0, 1]$ as constants, and so a rule like

$$a \leftarrow 0.5$$

is now valid. The intended meaning of this rule is to claim that formula a is true with a 0.5 probability.

Customizing the semantics

The end-user is given the flexibility to customize the semantics to her taste, by providing answers to two questions.

Naïveness. The first deals with lack of information: it is the truth value of an atom in case we have no information about it. Setting this to 0 *should* correspond to negation as failure, while setting it at 1 represents the naïve stance where we believe anything unless we have some evidence against it. Note that 1/2 seems to be a fair value for this.

Acceptance. The second question represents the truth threshold, and is used when we collapse the various truth values to either 2- or 3-valued models. All we need is to divide the unit interval $[0, 1]$ into three, consecutive, disjoint subintervals. The first one will represent falsehood, the last one truth, and the middle one will stand for "unknown". If the middle one is non-empty we obtain a 3-valued model, otherwise a 2-valued one. There seems to be no benefit in allowing either the first or the last interval to be empty.

> Remark A.1. Note that since we have allowed constants as part of our syntax, the user can further fine-tune the naïveness by assigning truth values to specific atoms, like so:
>
> $$a \leftarrow 0.5.$$
>
> This in effect gives a the value of $1/2$, so that, for example, even if we have chosen to use negation as failure in general (by setting naïveness to 0), we can make an exception for the case of a, where we interpret lack of information as $1/2$ instead of 0.

Probably dealing with negation and disjunction

Influenced by the way negation works in the infinite-valued semantics, we arrive at the following definition.

probabilistic complement
$\overline{x}$
doubted complement
$\widetilde{x}$

Definition A.1. Let $x \in [0, 1]$. Then we define its *probabilistic complement* $\overline{x}$ and its *doubted complement* $\widetilde{x}$ by

$$\begin{aligned} \overline{x} &=_{\mathrm{df}} 1 - x \\ \widetilde{x} &=_{\mathrm{df}} \frac{(1-x) + 1/2}{2}. \end{aligned}$$

Here is an example of how to use the probabilistic semantics on a program which we have met in [RW05b].

▶ *Example A.1.* Consider the program

$$\mathcal{P} := \left\{ \begin{array}{l} p \leftarrow \sim q \\ q \leftarrow \sim r \\ s \leftarrow p \\ s \leftarrow \sim s \\ r \leftarrow \mathit{false} \end{array} \right\}.$$

Our first task is to unite common-headed rules:

$$\mathcal{P}' := \left\{ \begin{array}{l} p \leftarrow \sim q \\ q \leftarrow \sim r \\ s \leftarrow p \vee \sim s \\ r \leftarrow \mathit{false} \end{array} \right\}.$$

We are now ready to calculate the semantics. We set naïveness to 1/2, so that

$$\mathbf{P}_0(p) = \mathbf{P}_0(q) = \mathbf{P}_0(s) = \mathbf{P}_0(r) = 1/2.$$

We now compute the values of $\mathbf{P}_1$ and $\mathbf{P}_2$:

$$\begin{array}{l|l}
\mathbf{P}_1(p) = \mathbf{P}_0({\sim}q) = \widetilde{\mathbf{P}_0(q)} = \widetilde{1/2} = 1/2 & \mathbf{P}_2(p) = \mathbf{P}_1({\sim}q) = \widetilde{\mathbf{P}_1(q)} = \widetilde{1/2} = 1/2 \\
\mathbf{P}_1(q) = \mathbf{P}_0({\sim}r) = \widetilde{\mathbf{P}_0(r)} = \widetilde{1/2} = 1/2 & \mathbf{P}_2(q) = \mathbf{P}_1({\sim}r) = \widetilde{\mathbf{P}_1(r)} = \widetilde{0} = 3/4 \\
\mathbf{P}_1(s) = \overline{\overline{\mathbf{P}_0(p)} \cdot \overline{\mathbf{P}_0({\sim}s)}} & \mathbf{P}_2(s) = \overline{\overline{\mathbf{P}_1(p)} \cdot \overline{\mathbf{P}_1({\sim}s)}} \\
\qquad = 1 - \overline{\mathbf{P}_0(p)} \cdot \overline{\mathbf{P}_0({\sim}s)} & \qquad = 1 - \overline{\mathbf{P}_1(p)} \cdot \overline{\mathbf{P}_1({\sim}s)} \\
\qquad = 1 - (1 - \mathbf{P}_0(p)) \cdot (1 - \mathbf{P}_0({\sim}s)) & \qquad = 1 - (1 - \mathbf{P}_1(p)) \cdot (1 - \mathbf{P}_1({\sim}s)) \\
\qquad = 1 - (1 - \mathbf{P}_0(p)) \cdot \left(1 - \widetilde{\mathbf{P}_0(s)}\right) & \qquad = 1 - (1 - \mathbf{P}_1(p)) \cdot \left(1 - \widetilde{\mathbf{P}_1(s)}\right) \\
\qquad = 1 - (1 - 1/2) \cdot \left(1 - \widetilde{1/2}\right) & \qquad = 1 - (1 - 1/2) \cdot \left(1 - \widetilde{3/4}\right) \\
\qquad = 1 - (1 - 1/2) \cdot (1 - 1/2) & \qquad = 1 - (1 - 1/2) \cdot (1 - 3/8) \\
\qquad \overset{\cdots}{=} 3/4 & \qquad \overset{\cdots}{=} 11/16 \\
\mathbf{P}_1(r) = 0, & \mathbf{P}_2(r) = 0.
\end{array}$$

Proceeding in this way, we arrive at the table

	$\mathbf{P}_0(\cdot)$	$\mathbf{P}_1(\cdot)$	$\mathbf{P}_2(\cdot)$	$\mathbf{P}_3(\cdot)$	$\mathbf{P}_4(\cdot)$	$\mathbf{P}_5(\cdot)$	$\mathbf{P}_6(\cdot)$	$\cdots$
p	0.5	0.5	0.75	0.75	0.75	0.75	0.75	$\cdots$
q	0.5	0.5	0.5	0.375	0.375	0.375	0.375	$\cdots$
s	0.5	0.75	0.6875	0.70313	0.62402	0.64874	0.64102	$\cdots$
r	0.5	0	0	0	0	0	0	$\cdots$

which we use to assign truth values to the relational symbols p, q, s, and r. ◀

Future work

Continuing the above example, the following things are desired and seem easy to prove for an arbitrary logic program.

The sequence $(\mathbf{P}_n(\mathsf{s}))_{n\in\omega}$ tends to a limit ξ_{s} as $n \to \infty$, which is the value assigned to s. Thanks to Banach's fixpoint theorem (see p. 42), all the limits whose existence is implied by a table like the one above can be characterized as the unique fixpoint of the contraction $T_{\mathcal{P}} : [0,1]^k \to [0,1]^k$ which assigns new probabilistic truth values to every predicate of $\mathcal{P}$ (here $k = 4$), given a set of old ones.

$(\mathbf{P}_n(\cdot))_{n\in\omega}$ behaves better for the rest of the symbols, as it obtains a constant value in a finite number of steps. We now collect all those values we obtained as limits for the "bad-behaving" sequences into a set U. Collapsing all truth values below $\inf U$ to $\mathbf{F}$, those above $\sup U$ to $\mathbf{T}$ and those in U to $\mathbf{0}$, we get the well-founded model.

Another "desired fact" is that there is a partition of $[0, \inf U)$ into consecutive intervals such that collapsing the αth interval to $\mathbf{F}_\alpha$, will agree with the infinite-valued semantics, and similarly for the values above $\sup U$. In that way, we can consider this approach a refinement of the infinite-valued model, as it allows us to further distinguish between otherwise incomparable **0**-valued predicates.

A.2 Symbolic semantics for disjunctive programs

This method lies in between denotational and operational semantics. Each DLP program $\mathcal{P}$ is iteratively transformed into a disjunction-free program $\ddot{\mathcal{P}}$, in which disjunctions are "hidden" behind a convenient symbolic notation. This way, the program becomes disjunctionless and the problem of assigning semantics is reduced to what we have already solved (or not) so far for LP and LPN.

I present a way to reduce a disjunctive program to a general program, so that the reduced program will have the intended semantics. The transformation of $\mathcal{P}$ is done in two steps, the first resembling Clark's completion:

(i) make each head appear exactly once, by joining bodies disjunctively; and
(ii) replace each disjunction by a fresh, representative "atom-set", adding all possible instances of the two rule-schemata:

$$\boldsymbol{\{}p_1 \vee \cdots \vee p_n\boldsymbol{\}} \leftarrow p_1 \vee \overset{-i}{\cdots} \vee p_n, \tag{A.1}$$

$$p_1 \vee \overset{-i}{\cdots} \vee p_n \leftarrow \boldsymbol{\{}p_1 \vee \cdots \vee p_n\boldsymbol{\}}, {\sim}p_i, \tag{A.2}$$

where i ranges over the set $\{1, \ldots, n\}$.

short version $\dot{\mathcal{P}}$

Definition A.2. Let $\mathcal{P}$ be a logic program, possibly involving disjunctions. Then define its *short version* $\dot{\mathcal{P}}$ by the equation

$$\dot{\mathcal{P}} =_{\mathrm{df}} \{H \leftarrow C_1 \vee \cdots \vee C_n \mid \mathbf{def}_{\mathcal{P}}(H) = \{C_1, \ldots, C_n\}\}\,.$$

Definition A.3. Let $D = \{D_1, \ldots, D_n\}$ be a finite set of atomic formulæ. Then,

$$\begin{aligned}
\mathsf{ESA}(D) &=_{\mathrm{df}} \bigcup\nolimits_i \left\{\boldsymbol{\{}D_1 \vee \cdots \vee D_n\boldsymbol{\}} \leftarrow D_1 \vee \overset{-i}{\cdots} \vee D_n\right\}\\
\mathsf{ESB}(D) &=_{\mathrm{df}} \bigcup\nolimits_i \left\{D_1 \vee \overset{-i}{\cdots} \vee D_n \leftarrow \boldsymbol{\{}D_1 \vee \cdots \vee D_n\boldsymbol{\}}, {\sim}D_i\right\}\\
\mathsf{ES}(D) &=_{\mathrm{df}} \mathsf{ESA}(D) \cup \mathsf{ESB}(D).
\end{aligned}$$

> ☡ REMARK A.2. Note that there is always a way to order the atoms of a language. Hence, to avoid commutativity woes, we always sort the disjuncts of a disjunction before encoding it into an atom-set.
>
> ☡ REMARK A.3. Once a disjunction is encoded into an atom-set, we are no longer able to "look inside" it. This is very important, and guarantees that there is no cheating involved in the semantics proposed.

Future work

It remains to verify the claim that the symbolic semantics assign a true/false value to an atom p iff the all the models of the minimal model semantics agree with that value.

A.3 Gossip programming

This alternate view of programming is too far from logic programming to be considered a part of it. The motivation behind it seems clearer if instead of rules, one thinks each program consist of rumours; hence the name.

In the following example, a party of three friends is deciding on where to go to on vacations. They have the following demands:

> «*I'll be happy if the hotel has a swimming pool, or if there is a beach nearby.*»
> «*I'll be happy if the hotel has a swimming pool, or if there is a mountain nearby.*»
> «*I'll be happy if the hotel has a swimming pool, or if there are good taverns nearby.*»

Breaking down these demands, let

$$\begin{aligned} a &= \text{the hotel must have a swimming pool} \\ b &= \text{there must be a beach nearby} \\ c &= \text{there must be a mountain nearby} \\ d &= \text{there must be good taverns nearby.} \end{aligned}$$

Now, we have a set of rules to satisfy, namely

$$\begin{Bmatrix} a \vee b \\ a \vee c \\ a \vee d \end{Bmatrix}.$$

Provided that each demand is of equal difficulty, it is obvious that the *easier* way to satisfy all of these demands is to satisfy a.[a)] Looking at these as programming rules, it seems reasonable to say that a is in some sense *truer* than b, c, or d. But just how much truer?

Semantics for gossip programming

Looking at just the first rule, we can assign the truth value 2/3 to a, and another 2/3 to b. But if we look at the bigger picture, it seems reasonable to ask: "what is the probability that a fails to be true?" This is obviously equal to the probability that all of b, c and d are satisfied (and not a) which boils down to $(1/3)^3 = 1/27$.

↯

[a)] This wouldn't be as obvious if a stood for the demand "there must be an alien spaceship nearby" instead. In this case it would have been easier to simultaneously satisfy all other three demands: just go to Crete.

> «Ne pleure pas, Alfred! J'ai besoin de tout mon courage pour mourir à vingt ans!»
>
> Evariste Galois

Appendix B

Mathematical preliminaries

B.1 Set & order theory

Excellent sources for the basic notions of set theory are [Hal60, Mos05], while for lattices and order, [DP02].

partial order

Definition B.1. A *partial order* on a set S is a reflexive, antisymmetric and transitive relation $\mathrm{R} \subseteq S \times S$, i.e.,

(i) $x \mathrel{\mathrm{R}} x$,

(ii) $x \mathrel{\mathrm{R}} y \ \&\ y \mathrel{\mathrm{R}} x \implies x = y$,

(iii) $x \mathrel{\mathrm{R}} y \ \&\ y \mathrel{\mathrm{R}} z \implies x \mathrel{\mathrm{R}} z$.

poset

bottom element

$\bot$

top element

$\top$

Definition B.2. A *poset* (partially ordered set) is a structured set $\langle P; \leq \rangle$ such that $\leq$ is a partial order on P. If there is an element a such that $a \leq x$ for every $x \in P$, we call it the *bottom element* and denote it by $\bot$. The *top element* is defined analogously and denoted by $\top$.

lattice

complete lattice

Definition B.3. A poset L is a *lattice* if $x \vee y$ and $x \wedge y$ exist for any x and y in L. A lattice L is *complete* if both $\operatorname{lub} X$ and $\operatorname{glb} X$ exist for any $X \subseteq L$. A side-effect of this definition is the existence of a bottom element $\bot = \operatorname{lub} \emptyset$ and a top element $\top = \operatorname{glb} \emptyset$.

$\uparrow$-notation

Definition B.4. Let L be a complete lattice and $T : L \to L$ be monotone. We define $\uparrow$ by

$$T \uparrow 0 = \bot$$

$$T \uparrow \alpha = \begin{cases} T(T \uparrow (\alpha - 1)) & \text{if } \alpha \text{ is a successor ordinal,} \\ \operatorname{lub}\{T \uparrow \beta \mid \beta < \alpha\} & \text{if } \alpha \text{ is a limit ordinal,} \end{cases}$$

$\downarrow$-notation

and *mutatis mutandis* for $\downarrow$,

$$
\begin{aligned}
T \downarrow 0 &= \top \\
T \downarrow \alpha &= \begin{cases} T(T \downarrow (\alpha - 1)) & \text{if } \alpha \text{ is a successor ordinal,} \\ \mathrm{glb}\{T \downarrow \beta \mid \beta < \alpha\} & \text{if } \alpha \text{ is a limit ordinal.} \end{cases}
\end{aligned}
$$

closure ordinal

The least ordinal α such that $T \uparrow \alpha = \mathrm{lfp}(T)$ is the *closure ordinal* of T.

B.2 Mathematical analysis

contraction

Definition B.5. Let $\langle R; d \rangle$ be metric space. A mapping $F : R \to R$ is called a *contraction* if there exists a number $\alpha < 1$ such that

$$d(Fx, Fy) \leq \alpha d(x, y), \qquad (\forall x, y \in R). \tag{B.1}$$

Note that every contraction is continuous. In fact, by virtue of (B.1), $x_n \to x$ implies $Fx_n \to Fx$. The following theorem is due to Banach, and was first stated in 1920.

Banach Fixpoint Theorem. *Every contraction mapping F defined on a complete metric space R has one and only one fixed point. Moreover, this fixpoint is the limit of the sequence $F^n(x_0)$, where x_0 is any point of R.*

Proof. See [KF57]. ▮

Bibliography

[Abr97] Samson Abramsky. Game semantics for programming languages. In Igor Prívara and Peter Ruzicka, editors, *Mathematical Foundations of Computer Science 1997*, volume 1295 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 1997. 10.1007/BFb0029944.

[ABW88] K. R. Apt, H. A. Blair, and A. Walker. Towards a theory of declarative knowledge. pages 89–148, 1988.

[Apt90] Krzysztof R. Apt. Logic programming. pages 493–574, 1990.

[AvE82] Krzysztof R. Apt and M. H. van Emden. Contributions to the theory of logic programming. *J. ACM*, 29(3):841–862, 1982.

[BCLR10] Piero Bonatti, Francesco Calimeri, Nicola Leone, and Francesco Ricca. Answer set programming. In Agostino Dovier and Enrico Pontelli, editors, *A 25-Year Perspective on Logic Programming*, volume 6125 of *Lecture Notes in Computer Science*, pages 159–182. Springer Berlin / Heidelberg, 2010.

[BF91] Nicole Bidoit and Christine Froidevaux. General logical databases and programs: Default logic semantics and stratification. *Information and Computation*, 91(1):15 – 54, 1991.

[Cla78] Keith Clark. Negation as failure. *Logic and Databases*, pages 293–322, 1978.

[Cla79] Keith Clark. Predicate logic as a computational formalism. 1979.

[CPRW07] Pedro Cabalar, David Pearce, Panos Rondogiannis, and William Wadge. A purely model-theoretic semantics for disjunctive logic programs with negation. In Chitta Baral, Gerhard Brewka, and John Schlipf, editors, *Logic Programming and Nonmonotonic Reasoning*, volume 4483 of *Lecture Notes in Computer Science*, pages 44–57. Springer Berlin / Heidelberg, 2007.

[DCLN98] Roberto Di Cosmo, Jean-Vincent Loddo, and Stephane Nicolet. A game semantics foundation for logic programming. In Catuscia Palamidessi, Hugh Glaser, and Karl Meinke, editors, *Principles of Declarative Programming*, volume 1490 of *Lecture Notes in Computer Science*, pages 355–373. Springer Berlin / Heidelberg, 1998. 10.1007/BFb0056626.

[DP02] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order (2nd ed.).* Cambridge University Press, 2002.

[Dra95] Wlodzimierz Drabent. What is failure? an approach to constructive negation. *Acta Informatica*, 32:27–59, 1995. 10.1007/BF01185404.

[Fit90] Melvin Fitting. Bilattices in logic programming, 1990.

[Fit99] Melvin Fitting. Fixpoint semantics for logic programming - a survey. *Theoretical Computer Science*, 278:25–51, 1999.

[GL88] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. pages 1070–1080. MIT Press, 1988.

[GRW08] Chrysida Galanaki, Panos Rondogiannis, and William W. Wadge. An infinite-game semantics for well-founded negation in logic programming. *Annals of Pure and Applied Logic*, 151(2-3):70 – 88, 2008. First Games for Logic and Programming Languages Workshop.

[Hal60] Paul R. Halmos. *Naive set theory.* Litton Educational Publishing, Inc., 1960.

[JlLL83] Joxan Jaffar, Jean louis Lassez, and John Lloyd. Completeness of the negation as failure rule. In *In Proceedings of the 8th International Joint Conference on Artificial Intelligence IJCAI-83*, pages 500–506, 1983.

[KF57] A. N. Kolmogorov and S. V. Fomin. *Elements of the Theory of Functions and Functional Analysis*, volume 1. Graylock Press, Rochester, New York, 1957.

[Kun89] Kenneth Kunen. Signed data dependencies in logic programs. *J. Log. Program.*, 7(3):231–245, 1989.

[Llo93] John Wylie Lloyd. *Foundations of Logic Programming.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1993.

[LMR92] Jorge Lobo, Jack Minker, and Arcot Rajasekar. *Foundations of disjunctive logic programming.* MIT Press, Cambridge, MA, USA, 1992.

[Mos05] Yiannis N. Moschovakis. *Notes on set theory (2nd ed.).* Springer-Verlag New York, Inc., New York, NY, USA, 2005.

[NM00] Ulf Nilsson and Jan Małuszyński. *Logic, Programming, and Prolog (2nd ed.)*. Nilsson & Małuszyński, 2000.

[PP90] Halina Przymusinska and Teodor Przymusinski. Semantic issues in deductive databases and logic programs. In *Formal Techniques in Artificial Intelligence*, pages 321–367. North-Holland, 1990.

[Rei78] R Reiter. On closed world data bases. *Logic and Databases*, pages 55–76, 1978.

[Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, 1965.

[RW05a] Panos Rondogiannis and William W. Wadge. An infinite-game semantics for negation in logic programming. In *In Proceedings of Games for Logic and Programming Languages (GaLoP)*, pages 77–91, 2005.

[RW05b] Panos Rondogiannis and William W. Wadge. Minimum model semantics for logic programs with negation-as-failure. *ACM Trans. Comput. Logic*, 6(2):441–467, 2005.

[SS94] Leon Sterling and Ehud Shapiro. *The art of Prolog (2nd ed.): advanced programming techniques.* MIT Press, Cambridge, MA, USA, 1994.

[vE86] M. H. van Emden. Quantitative deduction and its fixpoint theory. *Journal of Logic Programming*, 3:37–53, 1986.

[vEK76] M. H. van Emden and R. A. Kowalski. The semantics of predicate logic as a programming language. *Journal of the ACM*, 23:569–574, 1976.

[VGRS91] Allen Van Gelder, Kenneth A. Ross, and John S. Schlipf. The well-founded semantics for general logic programs. *J. ACM*, 38(3):619–649, 1991.

Index of symbols

Index of names

General index

Written with love, in vim.

```
:wq
```