LEARNING AND DEDUCTION IN NEURAL NETWORKS AND LOGIC

A thesis submitted to the National University of Ireland, Cork
for the degree of Doctor of Philosophy

Ekaterina Komendantskaya

Supervisor: Doctor Anthony Seda
Head of department: Professor Jürgen Berndt

Department of Mathematics
College of Science, Engineering and Food Science
National University of Ireland, Cork

May 2007

Table of Contents

NATIONAL UNIVERSITY OF IRELAND, CORK

Date: **May 2007**

Author: **Ekaterina Komendantskaya**

Title: **Learning and Deduction in Neural Networks and Logic**

Department: **Mathematics**

Degree: **Ph.D.** Convocation: **May** Year: **2007**

Signature of Author

Abstract

We show how Logic Programming *deduction* and Neural Network *learning* can be integrated within the field of Neural-Symbolic Integration. In order to show this, we combine methods of Connectionism and Neurocomputing.

We introduce Bilattice-Based Annotated Logic Programs (BAPs) which are suitable for reasoning with uncertainty and incomplete/inconsistent databases. We propose a fixed-point semantics for them, and in particular, we propose a semantic operator capable of computing the least Herbrand models of BAPs as its least fixed points. We define an inference algorithm, called SLD-resolution, for BAPs and prove that it is sound and complete with respect to the fixed-point semantics.

We show that the semantic operator for BAPs can be simulated in Connectionist Neural Networks that embed Hebbian Learning functions. The learning neural networks we construct improve computational complexity comparing with other classes of non-learning neural networks that simulate many-valued semantic operators. However, similarly to other Connectionist Neural Networks, in the case of first-order Bilattice-Based Annotated Logic Programs, this construction would require an infinite number of neurons in order to perform computations of the semantic operator. We prove an approximation theorem showing that finite neural networks can approximate such computations.

Finally, we build artificial neural networks that perform the SLD-resolution algorithm for conventional Horn-clause Logic Programs. The resulting neural networks are finite and embed six learning functions. They can be practically implemented as a neural network interpreter for SLD-resolution.

We conclude that the integrative approach using techniques from both Connectionism and Neurocomputing is very effective. We suggest further work to be done in this field. In particular, we outline possible future developments of the neural network interpreter for SLD-resolution.

Acknowledgements

I am grateful to my supervisor, Dr Anthony Seda, for organising the work of the research group "Nature Inspired Models of Computation" and for his supervision of my PhD studies.

I would also thank Dr Anthony Seda and Dr John Power for supervising the joint research project "Category Theory and Nature Inspired Models of Computation" between the universities of Cork and Edinburgh. This project had an immense influence on my research. I thank Dr John Power for very useful discussions on various topics.

Two Schools for Postgraduate Students MATHLOGAPS (Mathematical Logic and Applications), University of Leeds, August 2006, and MODNET (Model theory), University of Leeds, December 2005, helped me to form a very general understanding of major techniques and problems solved in Mathematical Logic. It is my pleasure to thank Dr Jim Grannell, the chairman of the School of Mathematics, Applied Mathematics and Statistics, University College Cork, for generous funding of my visits to Leeds; and Professor Dugald MacPherson, School of Mathematics, University of Leeds, for organising the splendid training courses for PhD students.

I thank Dr Roy Dyckhoff for his useful suggestions concerning my work, and in particular, for his useful comments concerning the first three chapters of this thesis. I am also grateful to Dr Dyckhoff for organising a splendid academic visit for me at the School of Computer Science, University of St Andrews in January 2007. The summary of some of our January discussions constituted the Further Work section of the thesis.

I thank Dr Damien Woods for supplying me with up-to-date references concerning complexity characterisation of neural networks.

I thank the School of Mathematics, Applied Mathematics and Statistics, University College Cork, for the tuition bursary I was offered for the time of my studies in Cork. Apart from providing funding, this gave me valuable experience of teaching mathematics at university level.

I thank the Boole Centre for Research In Informatics, University College Cork, for funding the second year of my PhD studies and for partial funding of my visits to conferences.

I acknowledge the "Fee-Waiver for Non-EU Nationals" scholarship provided by the Postgraduate Studies Board, University College Cork.

Several different organisations funded my visits to international conferences, and I acknowledge their kind support: the American Association for Symbolic Logic (ASL) grant for presenting a paper at CIE'06; travel grant from the Organisers of CIE'06, University of Swansea, Wales; student travel grant from the organisers of LATA'07, University of Tarragona, Spain.

I thank Paul Keegan, the System Administrator in the School of Mathematics, UCC, for presenting me with a very reliable laptop. This gave me a happy chance to study at home when my daughter was born. Every line of this thesis and every line of each of my papers published while in Cork was typed on this laptop.

I thank my husband, Dr Vladimir Komendantsky, for bringing me to Cork, and for encouraging me to start my PhD studies there. I am infinitely grateful to my mother, Anna Gubanova, and to my parents in law, Evgeniy Komendantsky and Zoya Komendantskaya, for jointly paying the fees for the first year of my studies in UCC. Without their support, my arrival in Cork would have never been possible.

I am grateful to Martine Seda for her support and caring attention to me and all my family during the time of my studies in Cork.

I thank all my office mates, and in particular Máire Lane, Clodagh Carroll, Marina Yegorova-Dorman, Jackquie Kirrane, and Julie O'Donovan, for creating the unique friendly and warm atmosphere in our office.

Every day of my life, I thank my two Annas – my mother and my daughter — for loving me, for being with me, for being two symmetric parts of my heart.

Cork Ekaterina Komendantskaya

Declaration

This thesis was composed by myself and the work reported in it is my own except where otherwise stated.

Introduction

"The human mind... infinitely surpasses the powers of any finite machine". This manifest by Kurt Gödel, proclaimed in his Gibbs lecture to the American Mathematical Society in 1951, has been challenging researchers [21, 27, 149, 151] in many areas of computer science and mathematics to build a device that will prove to be as powerful as the human brain. The very titles speak for themselves: "A logical calculus of the ideas immanent in nervous activity"[151] (1944), "Probabilistic Logics and the Synthesis of Reliable Organisms from Unreliable Components"[165] (1956), "The Algebraic Mind"[149] (2001), "Brain Calculus"[27] (2001), "Brane Calculi"[21] (2004), this is just to mention some of the work done in this field.

This fundamental problem of creating (and then evaluating) automated reasoning system based upon some formally defined logical calculus is, however, at least one century older than Gödel manifest cited above. One can argue that the problem is as old as mathematical logic and even computational mathematics.

Thus, a century before Gödel, George Boole, one of the fathers of mathematical logic, was driven by the same problem of how one can formalise human reasoning, and his famous *"An Investigation of the Laws of Thought on Which are Founded the Mathematical Theories of Logic and Probabilities"* [18] is often cited as the first book on mathematical logic. Several years later, Peano [169] was trying to introduce a calculus capable of deriving

mathematically valid statements. Hilbert, in his attempts to find proper foundations of mathematics and a proper formal calculus for it, published "Axiomatic thought" [87] in 1918 and announced the programme of formalising mathematics using some logical calculus. This program is now commonly called "Hilbert's Programme".

And thus, at the beginning of the 20th century, Mathematical Logic became an independent discipline of Mathematics. There were three results that determined the birth and development of the discipline:

- The Completeness Theorem of Gödel [1929] establishing that in first-order predicate calculus every logically valid formula is provable.

- The Incompleteness Theorem of Gödel [1931] showing incompleteness of any axiomatic arithmetic. More precisely, Gödel proved that there are undecidable propositions in every axiomatic theory which is at least as rich as Peano Arithmetic.

- The existence of algorithmically undecidable problems, in particular, the Theorem of Church [1936] about algorithmic undecidability of first-order predicate calculus and the equivalent theorem of Turing [1936] concerning undecidability of the halting problem for Turing machines.

The three former results gave evidence that Hilbert's programme cannot be accomplished.

However, even after the famous theorems of Gödel, Church and Turing, the major question of how one can create some kind of automated reasoning, or, as it was later called, artificial intelligence, remained as appealing as in the times of Boole. Turing's machines stimulated the creation of digital computers; biology and neuroscience became proper scientific disciplines. And all this progress only increased interest in the general

problem of creating an artificial intelligence.

Returning to the citation of Gödel we started with, it is worth mentioning that, as advocated in [36], Gödel specifically claimed that Turing, who created and promoted the use of Turing machines, overlooked that the human mind "in its use, is not static but constantly developing". This particular concern of Gödel found a further development in connectionism and neurocomputing [151, 164, 165, 83, 159, 179]. A very delicate discussion of Gödel's and Turing's arguments, and of the general question of whether the human mind acts algorithmically, and whether the soundness of such algorithm would be provable, can be found in [171].

Connectionism is a movement in the fields of artificial intelligence, cognitive science, neuroscience, psychology and philosophy of mind which hopes to explain human intellectual abilities using artificial neural networks. Neural networks are simplified models of the brain composed of large numbers of units (the analogs of neurons) together with connections between the units and weights that measure the strength of these connections. Neural networks demonstrate an ability to learn such skills as face recognition, reading, and the detection of simple grammatical structure. We will pay special attention to one of the topics of connectionism, namely, the so-called topic of *neuro-symbolic integration*, which investigates ways of integrating logic and formal languages with neural networks in order to better understand the essence of symbolic (deductive) and human (developing, spontaneous) reasoning, and to show interconnections between them.

Neurocomputing is defined in [85] as a technological discipline concerned with information processing systems (neural networks) that autonomously develop operational capabilities in adaptive response to an information environment. Moreover, neurocomputing is often seen as an alternative to *programmed computing*: the latter is based on the

notion of some fixed *algorithm* (a set of rules) which must be performed by a machine; the former does not necessarily require an algorithm or rule development. Note that in particular, neurocomputing is specialised on the creation of machines implementing neural networks, among them are digital, analog, electronic, optical, electro-optic, acoustic, mechanical, chemical and some other types of neurocomputers; see [85] for further details about physical realizations of neural networks.

In this thesis we will use **methods** and achievements of both connectionism and neurocomputing: namely, we take the connectionist neural networks of [98, 99] as a starting point, and show how they can be enriched by learning functions and algorithms implemented in neurocomputing.

We believe that this sort of synthetic methodology is novel in its own right, as we have found a gap between the methods of connectionism and neurocomputing. The two areas seem to be largely isolated at the moment. Thus, although learning functions are the main topic within Neurocomputing, the text book on neuro-symbolic integration entitled "Neural-Symbolic *Learning* Systems" [30] does not employ to any significant extent most of the learning functions used in Neurocomputing and surveyed in Section 5.4. From the side of Neurocomputing, one argues that *nine*-neuron networks are sufficient to simulate Universal Turing Machines [190], see also Section 5.1. From the side of Neuro-Symbolic Integration, one approximates the work of *infinite* neural networks in order to simulate the Turing-computable least fixed points of the semantic operator T_P for first-order logic programs [98]. And this method of [98] has been developed for more than a decade, inspiring a number of papers about approximations of the infinite neural networks [99, 94, 10, 184]; see Chapter 5 and Section 6.3 for further details.

The **main question** we address is: how can learning in neural networks and deduction

in logic complement each other? The main question splits into two particular questions: Is there something in deductive, automated reasoning that corresponds (or may be brought into correspondence) to learning in neural networks? Can first-order deductive mechanisms be realized in neural networks, and, if they can, will the resulting neural networks be learning or static?

First we fix precise meanings of the terms "deduction" and "learning".

Deductive reasoning is inference in which the conclusion is necessitated by, or deduced from, previously known facts or axioms.

The notion of learning is often seen as opposite to the notion of deduction. We adopt the definition of learning from neurocomputing, see [82]. *Learning is a process by which the free parameters of a neural network are adapted through a continuing process of simulation by the environment in which the network is embedded. The type of learning is determined by the manner in which the parameter changes take place.* The definition of the learning process implies the following sequence of events:

- The neural network is *stimulated* by an environment.
- The neural network *undergoes changes* as a result of this stimulation.
- The neural network *responds in a new way* to the environment, because of the changes that have occurred in its internal structure.

In this thesis we will borrow techniques from both supervised and unsupervised learning paradigms, and in particular we will use such algorithms as error-correction learning (see Section 5.4.2), Hebbian learning (see Section 5.4.1), competitive learning (see Section 5.4.4) and filter learning (see Section 5.4.3).

The field of *neuro-symbolic integration* is stimulated by the fact that formal theories (as studied in mathematical logic and used in automated reasoning) are commonly recognised as deductive systems which lack such properties of human reasoning as adaptation, learning and self-organisation. On the other hand, neural networks, introduced as a mathematical model of neurons in the human brain, are claimed to possess all of the mentioned abilities, and moreover, they provide parallel computations and hence can perform certain calculations faster than classical algorithms [190]; see also Section 5.1. As a step towards integration of the two paradigms, there were built connectionist neural networks [98, 99] which can simulate the work of the semantic operator T_P for propositional and (function-free) first-order logic programs. These neural networks, however, were essentially deductive and could not learn or perform any form of self-organisation or adaptation. Moreover, in the first-order case, when the Herbrand base of a given logic can be infinite, the neural networks of [98, 99] would require an infinite number of neurons to compute the least fixed point of the semantic operator. This problem was tackled by proposing finite neural networks that approximate the computations performed in the infinite neural networks [94, 184, 10].

Therefore, there are at least two disconcerting conclusions that can be made about the construction of neural networks of [98, 99].

1. The first negative conclusion is that computations performed by the neural networks simulating T_P do not require learning. This leads to a general conclusion that conventional logic, being implemented in neural networks, is incapable of showing any sort of learning or self-organisation. This conclusion is especially disconcerting, if we recall that one of the main benefits of neural networks that Neurocomputing is proud of is the use of learning.

2. The second conclusion concerns the very method of implementing logic in neural networks. The fact that *infinite* neural networks would be required to simulate the conventional semantic operator suggests that the method of building the neural networks is not optimal. Indeed, neural networks are, by definition, finite. (Although, similarly to the Turing machines, one can imagine them working for an infinitely long time and making infinitely many iterations.) So, the very notion of infinite neural networks violates the definitions and the common practice of neurocomputing. Moreover, finite neural networks in general were shown to be as powerful and effective as universal Turing machines [190]. However, the neural networks of [98, 99] were incapable of computing the least fixed point of the T_P operator using a finite number of neurons and required approximations whereas this can be easily done using (finite) Turing machines.

In this thesis we tackled both of the listed problems. First we used *extensive* methods of improving the neural networks of [98, 99]. That is, first we relied on the methods of [98, 99], but tried to extend them to non-classical logic programs, hoping that this would bring learning into the picture. This extension was indeed useful in this respect, and allowed us to bring learning into neural networks. (See also Section 5.3.1 for an outline of other possible extensions of the neural networks of [98].) However, the resulting neural networks inherited the second of the two problems listed above, that is, they still required an infinite number of neurons in the first-order case.

This is why we turned to the second, *intensive*, method of improving the neuro-symbolic networks of [98, 99]. We completely abandoned the methods used in [98, 99] and built neural networks that simulate the work of SLD-resolution, as opposed to the simulation of the semantic operator in [98, 99]. It is worth mentioning here that the

problem of establishing a method of performing first-order inference in neural networks has been tackled by many researchers over the last 30 years; see [72, 188, 136, 95], for example. The resulting SLD neural networks that we obtained showed the following properties: they were inherently finite, and embedded six learning functions, some of which represented supervised, and some unsupervised learning. Therefore, these novel neural networks solved both of the main problems listed above.

The structure of the thesis is as follows.

Chapters 1 and 2 are expository, and are used to give basic definitions and introduce major techniques that we use in later sections. In Chapter 1 we give an introduction to conventional logic programming and discuss its model-theoretic and proof-theoretic properties, mainly following [138]. Thus, we define "first-order language and theory", and give interpretations for the language. We then define conventional logic programming, its syntax and semantics. We outline the main methods and techniques that are used to characterise logic programs semantically and computationally. Namely, we define the semantic operator for logic programs, and show that it characterises models of logic programs. We outline the main definitions and results concerning SLD-resolution that is used to run logic programs.

In Chapter 2 we discuss many-valued logic programming and the three main ways in which many-valued logic programs extend conventional logic programs. We discuss the nature and properties of many-valued interpretations. We also use this chapter to survey the relevant literature in the subject.

In the remaining chapters we develop our Thesis concerning deduction and learning in logic and neural networks.

Chapters 3, 4 are devoted to Bilattice-Based Annotated Logic Programs (BAPs) in

order to show, in Chapter 6, that the semantic operator for BAPs can be simulated by learning neural networks. Therefore, these chapters serve to support the *extensive* method of developing the neural networks of [99].

The extensive method was chosen to make use of some non-classical formal theory, similar to *logic programs for reasoning with inconsistencies* [107], for example, and adapt the connectionist neural networks of [98, 99] to these logic programs in order to investigate if this adaptation brings learning into the picture. We chose *many-valued logic programs* for this purpose. And in particular, we chose bilattices as a suitable structure in which to take interpretations for the logic programs. Bilattices were advertised in [65, 49, 50], as structures which are particularly suitable for formalising human reasoning and serving for different purposes of Artificial Intelligence. And so, we chose bilattices to underly the many-valued logic programs.

In Chapter 3 we introduce BAPs, and in Chapter 4 we characterise them from the model-theoretic and proof-theoretic point of view. Namely, we define a semantic operator $\mathcal{T}_P$ for them, give declarative and fixpoint semantics based on properties of $\mathcal{T}_P$, propose the method of SLD-resolution for these logic programs, and show that this SLD-resolution is sound and complete.

The two Chapters 3 and 4 characterise BAPs as a very interesting class of logic programs, both from the mathematical and computational points of view. Mathematically, they turn out to be a very general formalism that allows us to handle uniformly different many-valued logic programs within the context of BAPs; see Section 4.4 for further discussions. Also, we show in Section 4.2 that they possess some non-trivial model-theoretic properties. Computationally, they provide a sound and complete proof procedure that logic programs of this generality [108] fail to provide, see Section 4.3.

In Chapter 5, we give a very broad introduction to the history, background definitions, and main problems of Neurocomputing and Connectionism. We outline the relevant literature in these two areas.

In Chapter 6, we use BAPs and build neural networks in the style of [98, 99] to simulate the computations of the semantic operator $\mathcal{T}_P$. As we hoped from the start, the resulting neural networks embed learning functions. Thus, we show that the connectionist neural networks need to embed learning functions in order to reflect some non-trivial properties of $\mathcal{T}_P$ and perform its iterations. We also show in Section 6.4 that some fragments of BAPs can be simulated by different, non-learning, neural networks of Lane and Seda, proposed in [135]. But the learning functions used in the neural networks that we build in Chapter 6 improve the time and space complexity of computations compared with the neural networks of [185, 135], see [119]. Also, the neural networks for BAPs turn out to be more expressive than the neural networks of [185, 135]. And this proves that in general, learning methods taken from Neurocomputing are well worth implementing in the area of neuro-symbolic integration.

However, we were not completely satisfied with the obtained results, for two major reasons. The first, and the main, reason was that the neural networks we built in the style of [98, 99] inherited all the main characteristic properties (see Section 5.3) of the neural networks of [98, 99]. And in particular, in the first-order case the mentioned above neural networks would become infinite, and would require the use of some non-constructive approximation theorem similar to the one we prove in Section 6.3. So, the neural networks we have constructed are still less effective than (finite) Turing machines which are capable of computing the least fixed point of $\mathcal{T}_P$.

The second reason for our scepticism concerning the neural networks for BAPs was that

Section 6.4 provided evidence that, although learning improves the complexity of neural networks, it still can be avoided in favour of more bulky but non-learning networks.

This is why, in Chapter 7, we propose a novel method of building neural networks that can perform the algorithm of SLD-resolution for classical neural networks. This method solves intensively both of the main problems we mentioned above. That is, the resulting neural networks are finite, and embed six learning functions recognised in neurocomputing.

Thus, we show that both negative conclusions made from considering the neural networks of [98, 99] can be effectively solved:

1. We show that classical (conventional) deduction requires the use of learning when implemented in neural networks, and the learning functions play a crucial role in computations. For example, these learning functions characterise the unification algorithm as an adaptive process, and use competitive learning when choosing the next goal from a given list of goals in the process of refutation.

2. There exist finite neural networks that computationally characterise first-order deduction.

These new conclusions provide us with non-trivial answers to the **main question** we posed at the beginning of the Introduction. We conclude the following: learning neural networks are shown to be able to perform *deductive* reasoning. On the other hand, the *learning* neural networks of Chapters 6, 7 are built to simulate their *logic* counterparts. This is why we can make the conclusion that both the semantic operator for BAPs and classical SLD-resolution bear certain properties which correspond to learning functions in neurocomputing. This conclusion sounds like an apology for symbolic (deductive) theories because it suggests that symbolic reasoning which we normally call "deductive" does not

necessarily imply the lack of learning and spontaneous self-organisation in the sense of neurocomputing.

Practically, SLD neural networks provide us with an interpreter which is capable of performing first-order deduction in finite and learning neural networks.

To summarise, the thesis contains the following novel methodological, theoretical and practical results.

- The **methodological** novelty of the thesis is that we combine the methods of Connectionism and Neurocomputing, in that we make use of the learning techniques applied in Neurocomputing when developing Connectionist neural networks. In particular, the learning techniques surveyed in Section 5.4 have never been implemented in the field of Neuro-Symbolic Integration before.

- **Theoretically**, we show that the common distinction *deduction versus learning* cannot be supported by neurocomputing, and we establish that the SLD neural networks performing classical SLD-resolution are inherently learning. Therefore, one can talk about *learning in deduction*, rather than *learning versus deduction*.

- **Practically**, we propose finite neural networks that can run the SLD-resolution algorithm. This open problem has been tackled by many researchers over the last 30 years, [72, 188, 136, 95]. Due to their inherent finiteness, the SLD-neural networks are implementable and applicable in the area of automated reasoning. We outline future work to be done with this respect in Section 8.2.

All the main results of the thesis were published and presented in international conferences.

Thus, the contents of Chapter 3 first appeared in [129], and were presented at the

International Conference "First-Order Theorem Proving" (FTP'05), Koblenz, Germany. But this material was further refined in [119, 127, 128, 126, 127]; and especially [123].

The contents of Chapter 4 were published in [119, 127, 128, 126, 127, 123], and presented at the Fourth Irish Conference on the Mathematical Foundations of Computer Science and Information Technology (MFCSIT'06), Cork, Ireland.

The material of Chapter 6 appeared in [120, 127, 119] and was presented at the International Conference Computability in Europe (CiE'06), Swansea, Wales.

The main results of Chapter 7 appeared in [120, 122] and were presented during the 1st International Conference on Language and Automata Theory and Applications (LATA'07), Tarragona, Spain.

There were results that we obtained while working on this thesis, but that are not included in the text of this thesis. This is done to keep the current text focused on the single subject that we announced in the title: "Learning and Deduction in Neural Networks and Logic". Nevertheless, these "abandoned" papers influenced very much our progress and understanding of the subject we worked on, and so deserve to be mentioned in this introductory chapter.

In [124], we developed non-Horn clause logic programs, and the results were presented at the 1st World Congress on Universal Logic (UNILOG-05), Montreux, Switzerland.

In [121], [123] we obtained a many-sorted representation of many-valued logics and logic programs, and the results were presented at the Fourth Irish Conference on the Mathematical Foundations of Computer Science and Information Technology (MFCSIT'06), Cork, Ireland and the International Conference on Automated Reasoning with Analytic Tableaux and Related Methods, TABLEAUX'07, Aix en Provance, France. We briefly survey some of these results in Chapters 3 and 4.

Finally, our work within the project "Category Theory and Nature Inspired Models of Computation" between the universities of Cork and Edinburgh, was very exciting and inspiring, and the results will appear in [125, 118].

Chapter 1

Logic Programming

1.1 Introduction

Logic Programming is one of the major subjects of this thesis. In this Chapter we give an introduction to the subject. We will briefly outline the history of Logic Programming, then we will focus on the precise background definitions concerning Logic Programs; and, finally, we will use different sections of this chapter to survey the relevant literature on the topic.

Logic Programming can be seen as a combination of two major trends. On the one hand, it is supported by logicians, whose research contributes to the *Logic* component of *Logic* Programming. The other area is developed by practical programmers, who rather emphasise the *Programming* part of it. This thesis tends to contribute to the former area: the author sees *Logic* Programming as a rightful successor and branch of mathematical logic as it was understood by its fathers Boole [18], Frege [57, 56], Peano [169], Hilbert [87, 88, 89, 90, 91, 92, 93], Gentzen [63], Gödel [67], Turing [201], Kleene [111], Church [22].

So, we start this first chapter with a short survey of the history and range of problems

stated and solved in Mathematical Logic. This discusssion will help us to evaluate the contribution of Logic Programming and Connectionism to the subject.

The first attempts to build formal mathematical theories were made by Boole [18], Frege [56, 57], Peano [169], Peirce [170] and some other mathematicians. The major goal was to establish solid mathematical foundations, and find axioms and inference rules sufficient for proving any mathematically valid proposition. It was hoped that these propositions could be proven automatically. This program was associated with the name of Hilbert [87, 88, 89, 90, 91, 92, 93] and was shown not to be accomplishable: Kurt Gödel [66] showed that every sufficiently rich formal system, such as Peano Arithmetic, is not complete, that is, there are mathematically true statements which are not provable in the system. This result, together with the corresponding results of Church and Turing mentioned in the Introduction, raised a wide range of questions concerning computability/uncomputability, model properties, barriers of computations, adequacy of inference techniques, etc. that have been tackled by researchers for more than 70 years.

Mathematical Logic has become a dynamically growing field of mathematics. Since it was established as an independent discipline, Mathematical Logic has developed different disciplines within itself; among them are Model Theory [148], Computability Theory (Theory of Recursion) [194], Proof Theory [175]; some disciplines of mathematics, such as Set Theory, Category Theory [146] (see also [8] for some discussion of relations between logic and category theory) or Type Theory [207, 150] are often seen as disciplines of Logic; see, for example, [14] for a wide range of *Mathematical Logic* Disciplines.

Mathematical Logic has found numerous applications in Computer Science, see [102, 1]; and [132, 112] are good examples of such interdisciplinary research. This perfect match of Mathematical Logic and Computer Science gave birth to a new discipline, which

is now commonly called *Theoretical Computer Science*. Theoretical Computer Science has been extensively developed, and it includes various subjects, some of which did not necessarily originate in mathematical logic; for example, analysis of algorithms, semantics of programming languages, etc.

Logic Programming is one of those subjects where Logic and Computing meet. The idea of formal logical inference, or deduction, formed the basis for creation of a universal programming language Prolog. It is significant that almost every field of Mathematical Logic has found its application within the theory of logic programming. Thus, Sections 1.4, 1.5 will give a model-theoretic account of logic programming, Section 1.6 and [155, 156, 158, 124] give its proof-theoretic account. A computability, or rather, complexity, analysis of logic programming is given in [40, 70], a categorical account of logic programming is given in [173, 110, 46, 125], and type theory was widely implemented in higher-order logic programming and λPROLOG, see, for example, [162, 158].

Before we continue discussions of current advances in Logic Programming, we must discuss in some detail its history and its background notions.

Logic Programming as an independent field of research began in the early 1970s as a direct outgrowth of earlier work on logic, automated theorem proving and artificial intelligence. Constructing automated deduction system was and remains central to the aim of achieving artificial intelligence. Built upon work of Herbrand [86] in 1930, there was much activity in theorem proving in the early 1960's by Prawitz [174, 175], Gilmore [64], Davis, Putnam [35] and others. These efforts culminated in 1965 with the publication of the landmark paper by Robinson [178], which introduced the resolution rule. Resolution is an inference rule which is particularly well-suited for automation on a computer.

The credit for the introduction of logic programming goes mainly to Kowalski [131]

and Colmerauer [26], although Green [69] and Hayes [81] should be mentioned in this regard. The famous *Warren Abstract Machine (WAM)* proposed by Warren [206] is one of the famous early PROLOG compilers. In 1972, Kowalski and Colmerauer were led by the fundamental idea that *logic can be used as a programming language.* The acronym PROLOG (PROgramming in LOGic) was conceived, and the first PROLOG interpreter [26] was implemented in the language ALGOL-W by Roussel, in Marseille in 1972.

The idea that first-order logic, or at least a substantial subset of it, could be used as a programming language was revolutionary because, until 1972, logic had only been used as a specification of declarative language in computer science. However, what [131] shows, is that logic has a *procedural interpretation*, which makes it very effective as a programming language. See also Section 1.6 and 4.3.

Most current logic programming systems are resolution theorem provers. However, logic programming systems need not necessarily be based on resolution. They can be non-clausal systems with many inference rules [19, 79, 158, 156, 124]. But here we will work only with resolution-based logic programming systems.

The remaining sections of this chapter present the basic concepts and results which constitute the theoretical foundations of logic programming. We discuss first-order syntax, theories, interpretations and models, unification and SLD-resolution. We refer to classical books on the subjects, such as [138], [197], [25].

1.2 First-Order Language and Theory

As we mentioned in the previous section, the very notion of a logic program was built upon the notions of a first-order language and a first-order theory. And so we start discussions of the syntax of Logic Programs with a brief outline of first-order languages and theories.

We refer to classical books on mathematical logic, such as [22, 43, 111, 138, 153, 189].

A *first-order theory* **T** consists of an alphabet **A**, a first-order language **L**, a set of axioms and a set of inference rules. The first-order language consists of the well-formed formulae of the theory. The axioms are a designated subset of well-formed formulae. The axioms and rules of inference are used to derive the theorems of the theory.

Definition 1.2.1. *We fix the alphabet* **A** *to consist of*

- *constant symbols* $a_1, a_2, \ldots,$
- *variables* $x_1, x_2, \ldots,$
- *function symbols* $f_1, f_2, \ldots,$
- *predicate symbols* $Q_1, Q_2, \ldots,$
- *connectives* $\neg$, $\wedge$, $\vee$, *also called* negation, conjunction *and* disjunction.
- *quantifiers* $\forall, \exists$ *and*
- *punctuation symbols* "(", "," ")".

We will sometimes denote finite sequences of symbols $x_1, \ldots, x_n$ by $\overline{x}$.

We follow the conventional (inductive) definition of a term and a formula. Namely, every constant symbol is a term, every variable is a term, and if f_i is a function symbol of arity n and $t_1, \ldots, t_n$ are terms, then $f_i^n(t_1, \ldots, t_n)$ is a term.

Let Q_i^n be a predicate symbol of arity n and $t_1, \ldots, t_n$ be terms. Then $Q_i(t_1, \ldots, t_n)$ is a formula (also called an *atomic formula* or an *atom*). If F_1, F_2 are formulae and $\overline{x}$ are variables, then $\neg F_1$, $F_1 \wedge F_2$, $F_1 \vee F_2$, $\forall \overline{x} F_1$ and $\exists \overline{x} F_1$ are formulae.

Definition 1.2.2. *The first-order language* **L** *given by the alphabet A consists of the set of all formulae constructed from the symbols of the alphabet.*

Example 1.2.1. $\forall x_1 \forall x_2 (Q_1(x_1, x_2) \vee \neg Q_1(x_1, x_2))$ *is a formula of the language* **L**.

The next several definitions serve to fix the conventional terminology for bound and free variables, closed and ground formulae.

Definition 1.2.3. *The* scope *of* $\forall x$ *(respectively* $\exists x$*) in* $\forall x F$ *(respectively* $\exists x F$*) is* F*. A* bound occurrence *of a variable in a formula is an occurrence immediately following a quantifier or an occurrence within the scope of a quantifier, which has the same variable immediately after the quantifier. Any other occurrence of a variable is* free*.*

Definition 1.2.4. *A* closed formula *is a formula with no free occurrences of any variable.*

Example 1.2.2. *In Example 1.2.1 all the occurrences of all the variables are bound, so the formula* $\forall x_1 \forall x_2 (Q_1(x_1, x_2) \vee \neg Q_1(x_1, x_2))$ *is closed. The formula* $\forall x_1 (Q_1(x_1, x_2))$ *contains one free variable* x_2*, and so it is not closed.*

Definition 1.2.5. *A* ground term *is a term not containing variables. Similarly, a* ground atom *is an atom not containing variables.*

Definition 1.2.6. *A literal is an atom or the negation of an atom. A* positive literal *is an atom. A* negative literal *is the negation of an atom.*

Now we are ready to define the syntax of logic programs.

1.3 Syntax of First-Order Logic Programs

Logic Programs are finite sets of clauses, that is, first-order formulae satisfying the following definition.

Definition 1.3.1. *A closed formula of the form* $\forall\overline{x}(L_1 \vee \ldots \vee L_n)$*, where each* L_i *is a literal, is called a* clause.

Following classical books [25, 197, 138] on Logic Programming, we assume that each logic program contains a finite number of clauses. Note that even finite logic programs can have infinite models, and infinite search trees generated by SLD-resolution algorithm. From the point of view of implementations, the use of infinite sets of clauses is not practical, and would allow undersirable non-determinism in both model theoretical and proof theoretical characterisations of logic programs.

In the subsequent sections, we will need some infinite constructions. Thus, we will discuss infinite sets of clauses when talking about *models* of logic programs. We will use, for example, the notion of a set ground(P) of all ground instances of a logic program P. This will not contradict to our assumption about inherent finiteness of logic programs.

Example 1.3.1. *[138] The formula* $\forall x_1 \forall x_2 \forall x_3 (Q_1(x_1, x_3) \vee \neg Q_2(x_1, x_2) \vee \neg Q_3(x_2, x_3))$ *is a clause.*

Throughout, we will denote the clause

$$\forall x_1, \ldots x_n (A_1 \vee \ldots \vee A_m \vee \neg B_1 \vee \ldots \vee \neg B_k),$$

where $A_1, \ldots, A_m, B_1, \ldots, B_k$ are atoms and $x_1, \ldots x_n$ are the variables occurring in these atoms, by

$$A_1, \ldots, A_m \leftarrow B_1, \ldots, B_k.$$

Following traditional convention, we will call $A_1, \ldots, A_m$ a consequent of the clause, and $B_1, \ldots, B_k$ an antecedent of the clause.

In the subsequent chapters, we will work with a particular kind of clauses, called definite clauses. We define them as follows:

Definition 1.3.2. *[138] A* definite *program clause is a clause of the form*

$$A \leftarrow B_1, \ldots, B_n,$$

which contains at most one atom in its consequent. A is called the head *and atoms* $B_1, \ldots, B_n$ *are called the* body *of the program clause.*

Definition 1.3.3. *A* unit clause *is a clause of the form*

$$A \leftarrow,$$

that is, a definite clause with empty body. A definite goal *is a clause of the form*

$$\leftarrow B_1, \ldots, B_n,$$

that is, a definite clause with empty head.

Definition 1.3.4. *[138] A* definite program *is a finite set of definite program clauses.*

We introduce here some examples of definite logic programs, which will be used as running examples throughout the thesis.

Example 1.3.2. *Consider the following logic program* P_1*, which determines, for each pair of integer numbers* x_1 *and* x_2 *whether* $\sqrt[x_1]{x_2}$ *is defined.*

$$\begin{aligned} \mathit{defined}(\sqrt[x_1]{x_2}) &\leftarrow \mathit{even}(x_1), \mathit{nonnegative}(x_2) \\ \mathit{defined}(\sqrt[x_1]{x_2}) &\leftarrow \mathit{odd}(x_1) \end{aligned}$$

Let Q_1 *denote the property to be "defined",* $(f_1(x_1, x_2))$ *denote* $\sqrt[x_1]{x_2}$*;* Q_2*,* Q_3 *and* Q_4 *denote, respectively, the property of being an even number, nonnegative number and odd*

number. Then the program above can be represented as follows:

$$\begin{aligned} Q_1(f_1(x_1, x_2)) &\leftarrow Q_2(x_1), Q_3(x_2) \\ Q_1(f_1(x_1, x_2)) &\leftarrow Q_4(x_1) \end{aligned}$$

We will use the latter representation in the future.

To work properly, this program should be enriched with the program which determines whether a given number is odd or even, we will not address this issue here, but remark that such program can be found, for example, in [197].

Example 1.3.3. *[197] In this example we slightly abuse the notation, and use predicates "connected" and "edge" instead of symbols Q_1 and Q_2 fixed in Definition 1.2.1.*

Let GC (for graph connectivity) denote the logic program

$$\begin{aligned} \mathit{connected}(x_1, x_1) &\leftarrow \\ \mathit{connected}(x_1, x_2) &\leftarrow \mathit{edge}(x_1, x_3), \mathit{connected}(x_3, x_2). \end{aligned}$$

This program can determine, for all nodes x_1 and x_2, whether they are connected. We just need to add unit clauses describing a particular graph:

$$\begin{aligned} \mathit{edge}(a_1, a_2) &\leftarrow \\ \mathit{edge}(a_2, a_3) &\leftarrow \\ &\vdots \end{aligned}$$

And then the program will be able to answer queries of the form $\leftarrow \mathit{connected}(a_1, a_3)$ or $\leftarrow \mathit{connected}(x_1, x_2)$, given by definite goals.

Example 1.3.4. *This is an example of a* ground *definite logic program, that is, a program that does not contain variables.*

$Q_1(a_1) \leftarrow$

$Q_2(a_1) \leftarrow$

$Q_3(a_1) \leftarrow Q_1(a_1), Q_2(a_1)$

Definition 1.3.5. *A* Horn clause *is a clause which is either a definite program clause or a definite goal.*

Although syntactically we consider mainly definite logic programs and Horn clauses in this thesis, it is worth mentioning that there exist other classes of programs, with more complicated syntax; and we list some of them in Table 1.3.1. Note that all these languages can be seen as containing the logic of Horn clauses.

Table 1.3.1 represents the most popular syntactical extensions of Horn Clauses, but it is also possible to talk about semantical extensions of logic programming, for example, we can talk about modal and multimodal Horn clauses [11], about the probabilistic languages which use Kolmogorov's axioms describing probabilities [76, 44, 77, 167, 166]; or we can work with many-valued Horn clauses, as we do in Chapters 2, 3.

In any case, because of its minimality, Logic Programming based on Horn clauses is fundamental for us.

1.4 Interpretation and Herbrand Interpretation

The declarative semantics of a logic program is given by the usual (model-theoretic) semantics of formulae in first-order logic. This section discusses interpretations and models,

Type of Program Clauses	Syntax	Example
Horn clauses (PROLOG) [131]	Unrestricted uses of $\{\wedge\}$, but $\{\forall, \leftarrow\}$ are restricted to the top level only.	$\forall \overline{x}(A_0 \leftarrow A_1 \wedge \ldots \wedge A_n)$.
Normal clauses (PROLOG) [23, 137]	Unrestricted uses of $\{\wedge, \neg\}$, but $\{\forall, \leftarrow\}$ are restricted to the top level only.	$\forall \overline{x}(A_0 \leftarrow \neg A_1 \wedge \ldots \wedge A_n)$.
Clauses (PROLOG) [139, 138]	Unrestricted uses of $\{\wedge, \neg, \vee, \leftarrow, \forall, \exists\}$.	$\forall x_1 Q_1(x_1) \leftarrow \exists x_2(\neg Q_2(x_1, x_2) \vee (Q_3(x_2) \leftarrow Q_2(x_1, x_2))$.
Hereditary Harrop formulae (λPROLOG) [161, 154, 162]	Unrestricted uses of $\{\forall, \wedge, \leftarrow\}$. Restricted uses of $\{\exists, \vee\}$.	$\forall x_1(Q_3(x_1) \leftarrow (\forall x_2(Q_3(x_2) \leftarrow Q_2(x_1, x_2)) \wedge Q_1(x_1)))$
Lolli (a refinement of λPROLOG) [4]	Unrestricted uses of $\{\forall, \wedge, \leftarrow, \multimap\}$.	$\forall x_1(Q_3(x_1) \leftarrow (Q_1(x_1) \multimap \forall x_2(Q_3(x_2) \leftarrow Q_2(x_1, x_2))))$.
Linear Objects (LO) [3, 80]	Unrestricted uses of $\{\wedge, \otimes\}$, but $\{\forall, \multimap\}$ are restricted to the top level only.	$\forall \overline{x}(A_0 \multimap A_1 \otimes \ldots \otimes A_n)$.
Forum [157]	Unrestricted uses of $\{\forall, \wedge, \leftarrow, \multimap, \otimes\}$.	$\forall x_1(Q_1(x_1) \multimap \forall x_2(((Q_3(x_1) \otimes Q_1(x_1)) \leftarrow Q_3(x_2)) \leftarrow Q_2(x_1, x_2)))$.

Table 1.3.1: Syntactical Extensions of Horn-Clause Logic Programming

concentrating particularly on the important class of Herbrand interpretations.

First-order logic provides methods for deducing the theorems of a theory. These can be characterised, by Gödel's completeness theorem [67, 153], as the formulae which are logical consequences of the axioms of the theory, that is, they are true in every interpretation of the theory. In the logic programming systems which we are going to consider the resolution rule is the only inference rule.

We fix the definitions of pre-interpretation, interpretation and model, they are standard and can be found in [138, 153, 22], for example.

Definition 1.4.1. *A* pre-interpretation *for a first-order language* **L** *consists of the following:*

- *A non-empty set D, called the* domain *of the pre-interpretation.*
- *For each constant of* **L**, *the assignment of an element in D.*
- *For each n-ary function symbol in* **L**, *the assignment of a mapping from D^n to D.*

Definition 1.4.2. *An* interpretation *I for a first-order language* **L** *consists of a pre-interpretation J with domain D of* **L** *together with the following: For each n-ary predicate symbol in L, the assignment of a mapping from D^n into* {true, false}.

Definition 1.4.3. *Let J be a pre-interpretation for a first-order language* **L**. *A* variable assignment (with respect to J) *is an assignment to each variable in* **L** *of an element in the domain of J.*

Definition 1.4.4. *Let J be a pre-interpretation with domain D for a first-order language* **L** *and let V be a variable assignment. The* term assignment (with respect to J and V) *of the terms in* **L** *is defined as follows:*

- *Each variable is given its assignment according to V.*

- *Each constant is given its assignment according to J.*

- *If $t'_1, \ldots, t'_n$ are the term assignments for $t_1, \ldots, t_n$ and f' is the assignment of the n-ary function symbol f, then $f'(t'_1, \ldots, t'_n)$ is the term assignment for $f(t_1, \ldots, t_n)$.*

Definition 1.4.5. *Let J be a pre-interpretation of a first-order language L, V be a variable assignment with respect to J, and A an atom. Suppose A is $Q_1(t_1, \ldots, t_n)$ and $d_1, \ldots, d_n$ in the domain of J are the term assignments for $t_1, \ldots, t_n$ with respect to J and V. We call $A_{J,V} = Q_1(d_1, \ldots, d_n)$ the* J-instance of A with respect to V.
Let $\{A\}_J = \{A_{J,V} : V$ is a variable assignment with respect to $J\}$.

Definition 1.4.6. *Let I be an interpretation with domain D of a first-order language* **L** *and let V be a variable assignment. Then a formula in* **L** *can be given a* truth value true *or* false*, as follows:*

- *If the formula is an atom $Q(t_1, \ldots, t_n)$, then the truth value is obtained by calculating the value of $Q'(t'_1, \ldots, t'_n)$ where Q' is the mapping assigned to Q by I and $t'_1, \ldots, t'_n$ are the term assignments for $t_1, \ldots, t_n$ with respect to I and V.*

- *If the formula has the form $\neg F$, $F \wedge G$, $F \vee G$, then the truth value of the formula is given by the following table:*

F	G	$\neg F$	$F \wedge G$	$F \vee G$
true	*true*	*false*	*true*	*true*
true	*false*	*false*	*false*	*true*
false	*true*	*true*	*false*	*true*
false	*false*	*true*	*false*	*false*

- *If the formula has the form $\exists x F$, then the truth value of the formula is* true *if there exists $d \in D$ such that F has the truth value* true *with respect to I and $V(x/d)$, where $V(x/d)$ is V except that x is assigned d; otherwise its truth value is* false.

- *If the formula has the form $\forall x F$, then the truth value of the formula is* true *if, for all $d \in D$ we have that F has the truth value* true *with respect to I and $V(x/d)$; otherwise its truth value is* false.

Definition 1.4.7. *Let I be an interpretation for a first-order language* **L** *and let F be a formula in* **L**.

We say F is satisfiable in I *if $\exists(F)$ is true with respect to I.*

We say F is valid in I *if $\forall(F)$ is true with respect to I.*

We say F is unsatisfiable in I *if $\exists(F)$ is false with respect to I.*

We say F is nonvalid in I *if $\forall(F)$ is false with respect to I.*

Definition 1.4.8. *Let I be an interpretation for a first-order language* **L** *and let F be a closed formula of* **L**. *Then I is a* model *for F if F is true with respect to I.*

Example 1.4.1. *Consider the formula $\forall x_1 \exists x_2 Q_1(x_1, x_2)$ and the following interpretation I. Let the domain D be the non-negative integers and let Q_1 be assigned the relation $<$. Then I is a model for the formula.*

The concept of a model for a closed formula can easily be extended to a model for a set of closed formulae.

The axioms of a first-order theory are a designated subset of closed formulae in the language of the theory. For logic programs, the clauses of a program are its axioms.

Definition 1.4.9. *Let* **T** *be a first-order theory and let* **L** *be the language of T. A* model *for* **T** *is an interpretation for* **L** *which is a model for each axiom of T.*

Definition 1.4.10. *Let S be a set of closed formulae and F be a closed formula of a first-order language* $\mathbf{L}$. *We say that F is a* logical consequence *of S if, for every interpretation I of* $\mathbf{L}$, *I is a model for S implies I is a model for F.*

Proposition 1.4.1. *[138] Let S be a set of closed formulae and F be a closed formula of a first-order language* $\boldsymbol{L}$. *Then F is a logical consequence of S iff $S \cup \{\neg F\}$ is unsatisfiable.*

Applying these definitions and Proposition 1.4.1 to programs, we see that when we give a goal $G = \leftarrow B_1, \ldots, B_n$ to the system, with program P loaded, we are asking the system to show that the set of clauses $P \cup \{\neg((B_1 \wedge \ldots \wedge B_n))\}$ is unsatisfiable.

Thus, the basic problem is that of determining the unsatisfiability of $P \cup \{\neg(B_1 \wedge \ldots \wedge B_n)\}$, or, equivalently, of $P \cup \{G\}$. According to Definitions 1.4.7, 1.4.9, this is equivalent to determining whether or not *every* interpretation of the language is a model for $P \cup \{G\}$. Needless to say, this seems to be a formidable problem. Luckily, it turns out that there is a much smaller class of interpretations which suffice to test for unsatisfiability. These are the so-called *Herbrand interpretations*, which we now proceed to study.

Definition 1.4.11. *Let* $\mathbf{L}$ *be a first-order language. The* Herbrand Universe $U_{\mathbf{L}}$ *for* $\mathbf{L}$ *is the set of all ground terms, that can be formed out of the constants and function symbols appearing in* $\mathbf{L}$. *(In case* $\mathbf{L}$ *has no constants, we add some constant, say, a, to form ground terms.)*

Example 1.4.2. *Consider the program P_1 from Example 1.3.2. The Herbrand Universe* $U_{P_1} =$
$\{a_1, f(a_1, a_1), f(f(a_1, a_1), a_1), f(a_1, f(a_1, a_1)), f(f(a_1, a_1), f(a_1, a_1)),$
$f(f(f(a_1, a_1), a_1), a_1), \ldots\}$

Definition 1.4.12. *Let* $\mathbf{L}$ *be a first-order language. The* Herbrand Base $B_{\mathbf{L}}$ *for* $\mathbf{L}$ *is the set of all ground atoms which can be formed by using predicate symbols from* $\mathbf{L}$ *with terms from the Herbrand Universe as arguments.*

Example 1.4.3. *Continuing the previous example,*

$$B_{P_1} = \{Q_1(f(a_1, a_1)), Q_2(a_1), Q_3(a_1), Q_4(a_1), Q_1(f(f(a_1, a_1), a_1)), \ldots\}$$

Definition 1.4.13. *Let* $\mathbf{L}$ *be a first-order language. The* Herbrand pre-interpretation *for* $\mathbf{L}$ *is the pre-interpretation given by the following:*

- *The domain of the pre-interpretation is the Herbrand Universe* $U_{\mathbf{L}}$.
- *Constants in* $\mathbf{L}$ *are assigned themselves in* $U_{\mathbf{L}}$.
- *If* f *is* n*-ary function symbol in* $\mathbf{L}$*, then the mapping from* $(U_{\mathbf{L}})^n$ *into* $U_{\mathbf{L}}$ *defined by* $(t_1, \ldots, t_n) \rightarrow f(t_1, \ldots, t_n)$ *is assigned to* f.

An Herbrand interpretation *for* $\mathbf{L}$ *is any interpretation based on the Herbrand pre-interpretation for* $\mathbf{L}$.

Definition 1.4.14. *Let* $\mathbf{L}$ *be a first-order language and* S *a set of closed formulae of* $\mathbf{L}$. *An* Herbrand model *for* S *is an Herbrand interpretation for* $\mathbf{L}$ *which is a model for* S.

Applying all these definitions to logic programs, we will usually regard a program P as a set S of formulae, and then we will use the notation U_P, B_P and define the Herbrand universe, Herbrand base and Herbrand model for P, using the symbol P in subscripts instead of $\mathbf{L}$, as we did in Examples 1.4.2 and 1.4.3.

Next we state that in order to prove unsatisfiability of a set of clauses, it suffices to consider only Herbrand interpretations.

Proposition 1.4.2. *[138] Let S be a set of clauses and suppose S has a model. Then S has an Herbrand model.*

Proof. Let I be an interpretation for S. We define an Herbrand interpretation I' of S as follows:

$$I' = \{Q(t_1, \ldots, t_n) \in B_S : Q(t_1, \ldots, t_n) \text{ is true wrt } I\}.$$

It is straightforward to show that if I is a model, then I' is also a model. □

In any section where we work only with Herbrand interpretations and models, we will normally identify I and I' and always assume that an Herbrand model is a set of formulae satisfying the conditions given in the construction of I'.

Proposition 1.4.3. *[138] Let S be a set of clauses. Then S is unsatisfiable iff S has no Herbrand models.*

We have outlined the main semantical notions needed to characterise logic programs. In the next section, we will talk about the computational content of this semantics, mainly, we will show how the least Herbrand models of logic programs can be computed by the semantic operator.

1.5 Declarative Semantics and the T_P operator

The semantic operator T_P, introduced in [204], gives a precise characterisation of the least Herbrand model for any given definite logic program P. It was therefore widely used as a semantical counterpart of SLD-resolution. In Chapters 2, 3 we will discuss in detail different non-classical semantic operators. In this chapter we recall some of the conventional definitions to which we will refer later.

The following proposition and theorem [138, 204] characterise the minimal Herbrand models of logic programs.

Proposition 1.5.1 (Model Intersection Property)**.** *Let P be a definite program and $\{M_i\}_{i\in I}$ be a non-empty set of Herbrand models for P. Then $\cap_{i\in I} M_i$ is an Herbrand model for P.*

The least Herbrand model of a program P is traditionally denoted by M_P.

Theorem 1.5.1. *Let P be a definite logic program. Then*

$$M_P = \{A \in B_P : \ A \text{ is a logical consequence of } P\}.$$

Let P be a definite logic program. Then, by Proposition 1.4.2 and the remark following it, 2^{B_P} is the set of all Herbrand interpretations of P. It is a complete lattice under the partial order of set inclusion. The top element of the lattice is B_P and the bottom element is $\emptyset$. See also Chapter 3.2, and Example 3.2.1.

Definition 1.5.1. *[138] Let P be a definite logic program. The mapping $T_P : 2^{B_P} \rightarrow 2^{B_P}$ is defined as follows. Let I be an Herbrand interpretation. Then $T_P(I) = \{A \in B_P : A \leftarrow A_1, \ldots, A_n$ is a ground instance of a clause in P and $\{A_1, \ldots, A_n\} \subseteq I\}$.*

T_P provides both declarative and procedural semantics for a given P.

Example 1.5.1. *Consider the logic program from Example 1.3.4.*
Let $I_1 = \emptyset$, $I_2 = \{Q_1(a_1),\ Q_2(a_1)\}$.
Then $T_P(I_1) = \{Q_1(a_1),\ Q_2(a_1)\}$, and $T_P(I_2) = \{Q_1(a_1),\ Q_2(a_1),\ Q_3(a_1)\}$.

The semantic operator provides the Least Herbrand Model characterisation for each definite logic program. Namely, T_P computes the least Herbrand Model of a logic program

P as its least fixed point. Because much of the declarative, or fixpoint, semantics of this chapter and Chapter 3 is based upon this fact, we will recall briefly the definitions of monotonic and continuous mappings, and will give the definition of fixed points as well. We postpone giving basic definitions concerning lattices till Chapter 3, Section 3.2, where we will focus in particular on lattice and bilattice structures. Some of the definitions below use the standard definitions of a lattice, the least upper bound (lub), and the lower greatest bound (glb) of a lattice; these are formally defined in Section 3.2.

Definition 1.5.2. *Let $(L, \leq)$ be a complete lattice and $T : L \longrightarrow L$ be a mapping. We say T is* monotonic*, if $T(x) \leq T(y)$ whenever $x \leq y$.*

Definition 1.5.3. *Let $(L, \leq)$ be a complete lattice and $X \subseteq L$. We say that X is* directed *if every finite subset of X has an upper bound in X.*

Definition 1.5.4. *Let $(L, \leq)$ be a complete lattice and $T : L \longrightarrow L$ be a mapping. We say that T is* continuous *if $T(\text{lub}(X)) = \text{lub}(T(X))$, for every directed subset X of L.*

Our interest in these definitions arises from the fact that, as already mentioned above, the collection of all Herbrand interpretations for a given definite logic program P forms a complete lattice in a natural way (see also Example 3.2.1) and also because T_P associated with P was shown to be monotonic and continuous, [138].

Monotonic and continuous mappings can be computationally characterised by their fixed points, and we define the notion of the least fixed point next.

Definition 1.5.5. *Let $(L, \leq)$ be a complete lattice and $T : L \longrightarrow L$ be a mapping. We say that $a \in L$ is a* fixed point *of T if $T(a) = a$, and we call a the* least fixed point *of T if, for all fixed points b of T, we have $a \leq b$.*

The next theorem, known as the Knaster-Tarski theorem, establishes the existence of least fixed points for monotonic mappings.

Theorem 1.5.2. *[199] Let $(L, \leq)$ be a complete lattice and $T: L \longrightarrow L$ be monotonic. Then T has a least fixed point,* $\mathrm{lfp}(T)$. *Furthermore,* $\mathrm{lfp}(T) = \mathrm{glb}\{x : T(x) = x\} = \mathrm{glb}\{x : T(x) \leq x\}$.

For an arbitrary set X, we will denote the least upper bound of X by $\bigcup X$.

Definition 1.5.6. *Let $(L, \leq)$ be a complete lattice and $T: L \longrightarrow L$ be monotonic. Then we define*

$T \uparrow 0 = \emptyset$*;*

$T \uparrow \alpha = T(T(\alpha - 1))$*, if α is a successor ordinal; and*

$T \uparrow \alpha = \bigcup(T \uparrow \beta : \beta < \alpha)$*, if α is a limit ordinal.*

The next result is due to Kleene:

Theorem 1.5.3. *Let $(L, \leq)$ be a complete lattice and $T: L \longrightarrow L$ be continuous. Then* $\mathrm{lfp}(T) = T \uparrow \omega$.

We apply Theorems 1.5.2 and 1.5.3 to logic programs. The following theorem establishes a correspondence between the Least Herbrand Model of P and the least fixed point of T_P.

Theorem 1.5.4. *[204] Let M_P denote the* Least Herbrand Model *for a definite logic program P. Then*

$$M_P = \mathrm{lfp}(T_P) = T_P \uparrow \omega.$$

Example 1.5.2. *The following set is the Least Herbrand Model for the logic program P from Example 1.3.4: $T_P \uparrow 2 = \{Q_1(a_1), Q_2(a_1), Q_3(a_1)\}$.*

The last definition of this section gives the notion of a correct answer.

Definition 1.5.7. *Let P be a definite logic program and G a definite goal, $\leftarrow A_1, \ldots, A_n$. An* answer *for $P \cup \{G\}$ is a substitution θ for the free variables in G. We say that θ is a* correct answer *if $\forall((A_1, \ldots, A_n)\theta)$ is a logical consequence of P.*

One would wish to relate the semantical notion of a correct answer with the syntactical notion of a computed answer, and we pass to the proof theoretic background of logic programming in the next section.

1.6 Operational Semantics and SLD-Resolution

This section introduces the operational semantics of Logic Programming. We briefly survey the notions of unification and SLD-resolution as they were introduced by [178] and [131]; see also [138].

Definition 1.6.1. *Let S be a finite set of atoms. A substitution θ is called a unifier for S if $S\theta$ is a singleton. A unifier θ for S is called a* most general unifier *(mgu) for S if, for each unifier σ of S, there exists a substitution γ such that $\sigma = \theta\gamma$.*

Given a clause C from a definite logic program P, and substitution θ, we will call $C\theta$ a *variant of C*.

Given substitutions $\theta_1, \ldots, \theta_n$, we can compose them, and we will denote their composition by $\theta_1 \ldots \theta_n$.

Definition 1.6.2 (Disagreement set)**.** *Let S be a finite set of atoms. To find the disagreement set D_S of S locate the leftmost symbol position at which not all atoms in S have the same symbol and extract from each atom in S the subexpression beginning at that symbol position. The set of all such terms is the* disagreement set*.*

Unification algorithm[178]:

1. Put $k = 0$ and $\theta_0 = \varepsilon$.

2. If $S\theta_k$ is a singleton, then stop; θ_k is an mgu of S. Otherwise, find the disagreement set D_k of $S\theta_k$.

3. If there exist a variable v and a term t in D_k such that v does not occur in t, then put $\theta_{k+1} = \theta_k\{v/t\}$, increment k and go to 2. Otherwise, stop; S is not unifiable.

It remains to prove that indeed the Unification algorithm computes mgus.

Theorem 1.6.1. *[178][Unification Theorem] Let S be a finite set of atoms. If S is unifiable, then the unification algorithm terminates and gives an mgu for S. If S is not unifiable, then the unification algorithm terminates and reports this fact.*

Note that, at some stage of the unification algorithm, the disagreement set D_S can contain atoms, which effectively means that the set S contains atoms built from different predicates. In this case, it is clear that S is not unifiable. Nevertheless, the unification algorithm tries all the available substitutions for the variables in D_S before it comes to the conclusion that the set S is not unifiable. One can optimise this algorithm, and check D_S for the presence of atoms before searching for variable substitutions.

Thus, we will use the following reformulation of the algorithm of unification:

Definition 1.6.3 (Unification Algorithm (Alternative Definition))**.**

1. *Put $k = 0$ and $\theta_0 = \varepsilon$.*

2. *If $S\theta_k$ is a singleton, then stop; θ_k is an mgu of S. Otherwise, find the disagreement set D_k of $S\theta_k$.*

3. *If D_k contains atoms, stop; S is not unifiable.*

4. *If there exist a variable v and a term t in D_k such that v does not occur in t, then put $\theta_{k+1} = \theta_k\{v/t\}$, increment k and go to 2. Otherwise, stop; S is not unifiable.*

In Chapter 7, we will use Definition 1.6.3 when implementing the unification algorithm in Neural Networks.

The unification algorithm is embedded into the algorithm of SLD-refutation, which is described in the following definitions:

Definition 1.6.4. *Let a goal G be $\leftarrow A_1, \ldots, A_m, \ldots, A_k$ and a clause C be $A \leftarrow B_1, \ldots, B_q$. Then G' is* derived *from G and C using mgu θ if the following conditions hold:*

- *A_m is an atom, called the* selected *atom, in G.*
- *θ is an* mgu *of A_m and A.*
- *G' is the goal $\leftarrow (A_1, \ldots, A_{m-1}, B_1, \ldots, B_q, A_{m+1}, \ldots, A_k)\theta$.*

An *SLD-derivation* of $P \cup \{G\}$ consists of a sequence of goals $G = G_0, G_1, \ldots$, a sequence $C_1, C_2, \ldots$ of variants of program clauses of P and a sequence $\theta_1, \theta_2, \ldots$ of mgus such that each G_{i+1} is derived from G_i and C_{i+1} using θ_{i+1}. An *SLD-refutation* of $P \cup \{G\}$ is a finite SLD-derivation of $P \cup \{G\}$ which has the empty clause $\Box$ as its last goal. If $G_n = \Box$, we say that the refutation has length n.

The *success set* of P is the set of all $A \in B_P$ such that $P \cup \{\leftarrow A\}$ has an SLD-refutation.

If $\theta_1, \ldots, \theta_n$ is the sequence of *mgu*s used in an SLD-refutation of $P \cup \{G\}$, then a *computed answer* θ for $P \cup \{G\}$ is obtained by restricting the composition $\theta_1 \ldots \theta_n$ to the variables of G.

SLD-resolution is *sound and complete*, which means that the success set of a definite program is equal to its least Herbrand model.

The *soundness and completeness* of SLD-resolution can alternatively be stated as follows.

Theorem 1.6.2 (Soundness)**.** *[6] Let P be a definite logic program and G a definite goal $\leftarrow A_1, \ldots, A_k$. Suppose that $P \cup \{G\}$ has an SLD-refutation of length n and $\theta_1, \ldots, \theta_n$ is the sequence of* mgu*s of the SLD-refutation.*
Then we have that $\cup_{j=1}^{k}[A_j\theta_1 \ldots \theta_n] \subseteq T_P \uparrow n$.

Theorem 1.6.3 (Completeness)**.** *Let P be a definite logic program and G be a definite goal. For every correct answer θ for $P \cup \{G\}$, there exists a computed answer σ for $P \cup \{G\}$ and a substitution γ such that $\theta = \sigma\gamma$.*

Example 1.6.1. *Consider the logic program from Example 1.3.2.*

To keep computations simple, we chose $G_0 = \leftarrow Q_1(f_1(a_1, a_2))$, where $a_1 = 2$ and $a_2 = 3$, and add $Q_2(a_1) \leftarrow$ and $Q_3(a_2) \leftarrow$ to the database. Now the process of SLD-refutation proceeds as follows:

1. *$G_0 = \leftarrow Q_1(f_1(a_1, a_2))$ is unifiable with $Q_1(f_1(x_1, x_2))$, and the algorithm of unification can be applied as follows:*
   Form $S = \{Q_1(f_1(a_1, a_2)), Q_1(f_1(x_1, x_2))\}$ and form $D_S = \{x_1, a_1\}$. Put $\theta_1 = x_1/a_1$. Now $S\theta_1 = \{Q_1(f_1(a_1, a_2)), Q_1(f_1(a_1, x_2))\}$. Find $D_{S\theta} = \{x_2, a_2\}$ and put $\theta_2 = x_2/a_2$. Now $S\theta_1\theta_2$ is a singleton.

2. *Form the next goal $G_1 = \leftarrow (Q_2(x_1), Q_3(x_2))\theta_1\theta_2 = \leftarrow Q_2(a_1), Q_3(a_2)$. $Q_2(a_1)$ can be unified with the clause $Q_2(a_1) \leftarrow$ and no substitutions are needed.*

3. *Form the goal $G_2 = \leftarrow Q_3(a_2)$; it is unifiable with the clause $Q_3(a_2) \leftarrow$.*

4. Form the goal $G_3 = \square$.

There is a refutation of $P_1 \cup G_0$ *of length* 3*, the answer is* $\theta_1\theta_2$*, and, because the goal* G_0 *is ground, the computed answer is empty.*

1.7 Conclusions

In this Chapter, we have given a very broad introduction to Logic Programming, we discussed its place in the development of Mathematical Logic and Computer Science. We defined the syntax of Definite Logic Programs and gave a brief survey of extensions of Horn clause logic programs. We introduced some examples of definite Logic Programs that we are going to use in the subsequent chapters. Finally, we gave a formal description of the Declarative and Operational Semantics for Logic Programming.

Chapter 2

Many-Valued Logic Programming

In the previous chapter, we formally defined the syntax of definite logic programs. In Section 1.3, we also mentioned some possible extensions of these logic programs. In particular, Table 1.3.1 showed some *syntactical* extensions; it listed classes of logic programs whose syntax is enriched with some additional connectives and quantifiers. But we also mentioned that one can think about *semantical* extensions of Horn clauses; the latter way is mainly focused on extending the possible interpretations for logic programs. A good example of such semantical extension of logic programming is the area of *many-valued logic programming*. We will work with many-valued (bilattice-based) logic programs in Chapters 3 and 4.

In this chapter we give a general introduction to many-valued logic programming, and explain three main ways in which conventional, i.e. two-valued, logic programming has been extended to many-valued logic programming. This will enable us to analyse properties of the bilattice-based logic programs that we develop in the following chapters, and compare the logic programs we introduce with existing many-valued logic programs.

And so, we take conventional Horn-clause logic programming as a starting point. Recall that in Chapter 1, in Definitions 1.4.2, 1.4.6, we defined the interpretation for each

definite logic program to be an assignment to each atomic ground formula of either *false* (0) or *true* (1) such that, for each unit clause of the form $A \leftarrow$, the value of A is equal to 1 and for each clause of the form $A \leftarrow A_1, \ldots, A_n$, if the value of $A_1, \ldots, A_n$ is equal to 1, then the value of A must be equal to 1. The definite logic programs are run by SLD-resolution, and, as we have stated in Section 1.6, SLD-resolution is sound and complete with respect to the interpretation. We will use the notation $|A|_I$ or just $|A|$ when talking about an interpretation of A.

The idea of enriching the set of truth values $\{0, 1\}$ to some more complicated structure was first proposed by logicians many centuries before logic programming was invented. It is curious that the idea of many-valued interpretations was first suggested by Aristotle [7]. But this idea was not technically developed and implemented until the 20th century, when the rapid development of mathematical logic and the birth of computer science brought this idea into the context of formal theories and programming.

There exists a variety of many-valued logics, see [177] for a survey. In the first half of the 20th century, Łukasiewicz [144] and Kleene [111] introduced three-valued logics. Then infinite-valued Lukasiewicz logic appeared, and there are several other many-valued logics, such as fuzzy logic [210] and intuitionistic logic. Belnap [15] studied lattice-based logics for reasoning with uncertainties and his work was further developed in [65, 107, 39, 52]. Most of these logics have been adapted to logic programming, see, for example, [141] for logic programs interpreted by arbitrary sets, [48] for applications of Kleene logic to logic programming, [205] and [186] for fuzzy logic programming and [49, 50, 51] for (bi)lattice-based logic programming.

There have been three main approaches to many-valued logic programming: annotation-free logic programming, implication-based logic programming, and annotated logic programming.

Annotation-free logic programs introduced by Fitting in [48] and further developed in [49, 50, 196], are formally the same as two-valued logic programs. But while each atomic ground formula of a two-valued logic program is given an interpretation in $\{0, 1\}$, an atomic formula of an annotation-free logic program receives an interpretation in an arbitrary specified preorder Ω with finite meets. This structure of a preorder with finite meets is the minimal requirement needed to give an interpretation to clauses which have only conjunctions of atoms in their bodies. As we mentioned above, many authors work with structures more sophisticated than just a preorder with finite meets, but we have chosen Ω as the minimal possible structure to talk about in this introductory chapter. Very often, Ω is required to have all finite joins in order to guarantee computations of all the models for a given many-valued logic program. As we will show in this chapter, the semantic operators $T_P^{\blacklozenge}$ and R_P for implication-based [203] and annotated logic programs [49, 50] require the existence of joins in Ω.

Till the end of this chapter, we use symbols $\cap$ and $\cup$ to denote finite meet and finite join in Ω. We will give formal definitions of $\cap$ and $\cup$ in the next Chapter, where we formally describe lattices and lattice operations.

See also [125, 118] for some discussion of optimal "additional" conditions on Ω. In general, Ω does not need to be a complete Heyting algebra, see [125]. Thus Ω can be, for example, a semilattice, as in the weak three-valued logic of Kleene [111] or as in the generalised logic programming of [108]. The former logic was initially introduced by Kleene in order to give an adequate formal account of partially recursive predicates,

but later found numerous applications in philosophical logic [133], in areas of computer science working with incomplete databases, see, for example [52], [51], and in particular in many-valued logic programming [48].

We are unaware of an account of SLD-resolution for annotation-free logic programming prior to the one we have proposed in [125], but see [186] and [17] for related work. In general, the literature ([48, 49, 50, 54, 28, 198]) restricts itself to a semantic operator, thus lacking the efficiency of an algorithm based on SLD-resolution.

We proceed by means of a running example. Consider the logic program GC from Example 1.3.3.

Example 2.0.1. *Our leading example of an annotation-free logic program is as follows. Let* Ω *be the unit interval* $[0,1]$*, and we will use symbol* $\cap$ *to denote finite meets defined on* Ω*. The logic program GC is, by definition, also an annotation-free (*$[0,1]$*-based) logic program. But each ground atom, e.g.,* $edge(a,b)$ *or* $connected(a,b)$*, is assigned a truth value from* $[0,1]$*, cf. the notion of probabilistic graph [202], [176], where edges and connections in a graph exist with some probability. These probabilistic graphs can be used to describe the behaviour of internet connections, for example.*

If we have a ground clause of the form

$$connected(a,b) \leftarrow edge(a,c), connected(c,b)$$

we say that the clause is true *relative to an interpretation if*

$$|edge(a,c)| \cap |connected(c,b)| \leq |connected(a,b)|$$

in $[0,1]$.

The choice of the structure Ω from which one takes truth values can determine the properties of many-valued semantic operators. In general, for each annotation-free clause

$A \leftarrow A_1, \ldots, A_n$, the semantic operator will pick the minimal value $|A_i|$, $(i = 1, \ldots, n)$ and assign this value to A. But sometimes this simple generalisation of classical semantic operator is not sufficient for the computation of all the models for many-valued logic programs. For example, the semantic operator of Fitting applied to bilattice-based logic programs requires the "completion" in order to compute all the logical consequences of a logic program, see [54] or Section 4.4 for further details.

Implication-based logic programs introduced by Van Emden in [203] were designed in order to obtain a simple and effective proof procedure. Van Emden considered the particular case of $\Omega = [0, 1]$ but this has since been extended, see e.g. [134, 129] or Section 4.4. Much of the work extends with little effort to an arbitrary preorder Ω with finite meets and finite joins.

Van Emden used the conventional first-order syntax for logic programs, except for having clauses of the form:

$$A \leftarrow\!\boxed{f}\!- B_1, \ldots, B_n,$$

where f is a *factor* or *threshold* taken from the interval $[0, 1]$ of reals. The atoms A, $B_1, \ldots, B_n$ received their interpretation from the interval $[0, 1]$, such that the value of the head of a clause A was to be greater than or equal to $f \times \min(|B_1|, \ldots, |B_n|)$.

The semantic operator for these logic programs is defined as follows:

Definition 2.0.1. *[203] Let P be an implication-based logic program. For every $A \in B_P$, $T_P^{\blacklozenge}(I)(A) = \cup\{f \times \min\{|L_i| : i \in \{1, \ldots, n\}\}$ and $A \leftarrow\!\boxed{f}\!- L_1, \ldots, L_n$ is a variable-free instance of a clause in $P\}$.*

Note that the existence of finite joins in Ω is required in the Definition of the semantic operator $T_P^{\blacklozenge}$.

Van Emden gave an account of SLD-resolution for $[0,1]$-valued implication-based logic programs, elegantly extending that for two-valued logic programming, with a soundness and completeness theorem for implication-based SLD-resolution relative to the models computed by the $T_P^{\blacklozenge}$ operator.

Example 2.0.2. *Consider an implication-based variant of Example 2.0.1 for* $[0,1]$*:*

$$\begin{aligned}
\mathit{connected}(x,x) \quad &\leftarrow\boxed{1}- \\
\mathit{connected}(x,y) \quad &\leftarrow\boxed{0.5}- \quad \mathit{edge}(x,z), \mathit{connected}(z,y) \\
\mathit{edge}(a,b) \quad &\leftarrow\boxed{0.75}- \\
\mathit{edge}(b,c) \quad &\leftarrow\boxed{0.25}- \\
&\vdots
\end{aligned}$$

As in Example 2.0.1, an interpretation $|\ |$ *assigns a value in* $[0,1]$ *to each ground atom. But here, the threshold in the implication is used to compute the interpretation of the clauses. Thus, for example,* $|\mathit{connected}(a,b)| \geq 0.5 \times (|\mathit{edge}(a,c)| \cap |\mathit{connected}(c,b)|)$.

This differs from Example 2.0.1 in that we have a factor 0.5 *here that the setting of Example 2.0.1 does not admit. So elements of* Ω *do not appear in the syntax of an annotation-free logic program, but typically do appear in the syntax of an implication-based logic program.*

Annotated (or signed) logic programs require each atom in a given clause to be annotated (or signed) by a truth value.

Most commonly, an annotated clause has the following representation ([108]):

$$A : \tau \leftarrow B_1 : \tau_1, \ldots, B_n : \tau_n,$$

where each τ_i is an *annotation term*, which means that it is either an annotation constant, or an annotation variable, or a function over annotation constants and/or variables. An

interpretation $|\ |$ assigns a value in Ω to each ground atom. Then, a pair of the form $A : \tau$, with A being a ground atom, is assigned 1 if $|A| \geq \tau$ and 0 otherwise. So, the analysis of annotated logic programs restricts to that of two-valued logic programs.

There have been many accounts of SLD-resolution for annotated logic programs derived from that for two-valued logic programming. Because of the ease of implementation, this approach has been very popular and many variations of annotated and signed logic programs and provers for them have appeared, see, for example, [20, 107, 108, 141, 142, 143, 166, 198, 129] and Section 4.4.

Example 2.0.3. *An annotated variant of the program GC from Example 2.0.1 with* $\Omega = [0, 1]$ *is as follows.*

$$\begin{aligned} \mathit{connected}(x, x) : (1) &\leftarrow \\ \mathit{connected}(x, y) : (\mu_1 \cap \mu_2) &\leftarrow \mathit{edge}(x, z) : (\mu_1), \mathit{connected}(z, y) : (\mu_2) \\ \mathit{edge}(a, b) : (0.75) &\leftarrow \\ \mathit{edge}(b, c) : (0.25) &\leftarrow \\ \vdots & \end{aligned}$$

In the program above, μ_1 *and* ν_1 *are annotation variables,* $\cap$ *is an annotation function, and* $(\mu_1 \cap \mu_2)$ *is a complex annotation term.*

As in Examples 2.0.1 and 2.0.2, an interpretation assigns a value from $[0, 1]$ *to each ground atom. Then each ground annotated atom, for example,* $\mathit{edge}(a, b) : (0.75)$ *("there is an edge between* a *and* b *with probability* 0.75*") is evaluated as* 0 *or* 1*. Thus we return to the classical meaning of a program clause and of implication.*

When one works with annotated logic programs, there can be several definitions of semantic operators; for example, in the classical paper of [108], one can find the following

definition of a semantic operator:

Definition 2.0.2. *Given an annotated logic program P, interpretation $|\ |$ for P and $A \in B_P$, the operator R_P is defined by*
$R_P(|A|) = \cup\{f(\mu_1, \ldots, \mu_n) \mid A : (f(\mu_1, \ldots, \mu_n)) \leftarrow B_1 : \mu_1, \ldots, B_n : \mu_n$ *is a ground instance of a clause in P and* $(|B_1 : \mu_1, \ldots, B_n : \mu_n| = 1)\}$.

It is shown in [108] that the propositional annotation-free logic programs of Fitting [50] and the implication-based logic programs of Van Emden [203] can be fully expressed in terms of the annotated logic programs of [108], while [129] and Section 4.4 herein gives an algorithm for translating the first-order bilattice-based annotation-free logic programs of Fitting [50] and implication-based logic programs interpreted in bilattices into Bilattice-Based Annotated Logic Programs (BAPs). These results are of particular importance for the theory of many-valued logic programming, as they allow us to handle all the three types of many-valued logic programs described above within the formal framework of annotated logic programs. Thus, every result obtained for annotated logic programs will immediately hold for annotation-free and implication-based logic programs; see also Section 4.4 for further discussion of this. We will use this fact in our discussion concerning expressiveness of different artificial neural networks in Section 6.4.

However, both annotation-free and implication-based logic programs remain attractive and convenient for certain purposes; for example, because of their elegant syntactic representation or because of the sound and complete SLD-resolution [203] of the latter.

In the next chapter, we will concentrate in particular on bilattice-based annotated logics, but in Section 4.4 we will compare their expressiveness with annotation-free and implication-based logic programs interpreted in bilattices.

Chapter 3

Bilattice-Based Logic Programming

3.1 Introduction

In this chapter we make use of bilattice structures in the context of logic programming.

Our interest in bilattices and logic programs based on bilattices arose, as we mentioned in Introduction, from the desire to find a class of non-classical logic programs which would be the most suitable for working with uncertainty, and thus would be the most suitable for employing the process of learning in corresponding neural networks. Bilattices were very much advertised [65, 49, 50, 55] as a very general mathematical structure that captures well the notions of uncertainty, reasoning with probabilities, and allows one to formalise uniformly various inference systems capable of non-monotonic reasoning. And in fact, as we show in Chapter 6, the neural networks of [98], when adapted to bilattice-based logic programs, prove to be very useful in this respect: they embed learning functions. Before turning to formal discussions of neural networks for bilattice-based programs, we will develop declarative and operational semantics for logic programs of this kind. In this chapter and Chapter 4 we describe bilattices, introduce bilattice-based languages and logic programs, and also develop declarative and operational semantics for these programs.

Historically, the notion of a bilattice generalises Belnap's lattice [15] with four values - *true*, *false*, *none*, *both*. This lattice of Belnap suggested that we can compare facts not only from the point of view of them being true or false, but we can also question how much information about facts is available to us. This gave rise to the notion of a bilattice - a structure with two orderings, usually called the *truth* and *knowledge* orderings.

Bilattice structures were introduced in order to formalise hypothetical and uncertain reasoning. In Chapter 2 we considered a logic program that is capable of working with probabilistic graphs. As we show in Example 3.3.1, one can work with probabilistic graphs within bilattice-based logic programs.

The next example describes yet another situation where uncertain reasoning can be useful.

Example 3.1.1. *Hypothetical reasoning can be useful if we learn or investigate some new area. For example, scientists may try to solve some open problem which is important enough to be tackled for many years; and in the meantime, they obtain some side results which advance the solution of the main problem. Consider, for example the famous "$P \neq NP$?" problem. Imagine two bright scientists Dr. N and Dr. M employed by a university to solve it. The scientists consider some related problems which may lead to the final proof of "$P \neq NP$"; for example, "$NP \neq coNP$?" and "$NC \neq NP$?". After a while, both give proofs (which are very long and need to be checked by someone): Dr. M has proven that "$NP \neq coNP$", and Dr. N has proven that "$NP = coNP$". If a two-valued logic program receives the data, it will report a contradiction and will stop further derivations. But humans can, for example, make the following conclusions. If "$NP \neq coNP$" and "$NP = coNP$" were proven, then no conclusions yet can be made about the "$P \neq NP$" problem. If "$P \neq NP$" is proven, we can conclude that "$NC \neq NP$". We will see in later sections how*

this sort of reasoning can be formally handled in bilattice-based logic programming. And in particular, Example 3.5.1 formalises this Example about "$P \neq NP$" problem. Example 6.2.1 shows how the program of Example 3.5.1 can be processed in neural networks.

The notion of a bilattice was first introduced by M. Ginsberg [65]. He defined bilattice structures in general, characterised some of their properties, and suggested some implementations of bilattices in artificial intelligence. Fitting further contributed to the development of bilattice structures in the context of logic programming semantics, see [49, 50, 51]. In [54, 49, 50], Fitting introduced quite general consequence operators for logic programs whose semantics are based on four-valued bilattices. He also obtained fixpoint semantics for four-valued interpretations, and described connections between the least and the greatest fixed points of the consequence operators with respect to the k- and t-orderings.

However, bilattices with structures more complicated then those using four truth values have not been widely implemented in logic programming, but see [61, 107, 134]. Some of the operations which are well-defined in the case of four truth values do not extend easily to more than four truth values, and therefore need to be modified.

Annotated languages and programs which we have described in general in Chapter 2 have been used as a formal tool for handling, for example, the semantics of logic programs over many-valued logics and probabilistic programs, see [108, 167, 145]. As we argued in Chapter 2, they are particularly valuable because of their expressiveness, see also Section 4.4. Their use, however, gives rise to the obvious question of how annotated logic programming can incorporate logic programming based on bilattices, see [107, 108, 129, 134] and Section 4.4. In this chapter, we contribute to this discussion by combining the two

approaches in that we make use of bilattice structures within the framework of annotated logic programs. Specifically, we introduce bilattice-based annotated logic programs (BAPs). BAPs, being many-valued and quantitative in nature, enable us to work with statistical knowledge and databases which can be incomplete or inconsistent, and thereby introduce monotonic extensions of two-valued negation. But what is particularly important, is that they possess many of the desirable properties of classical logic. Thus, their model-theoretic properties can be nicely described by analogy with the model-theoretic properties of classical logic programming and, if some additional computational rules are introduced into the proof theory, sound and complete proof procedures based on classical binary resolution methods can be developed, see Section 4.3. Also BAPs are expressive enough to formalise the fixed-point theory of the annotation-free logic programs of Fitting [49] and of implication-based quantitative annotated programs *à la* Van Emden.

The structure of the chapter is as follows. In Section 3.2, we describe bilattices. In Section 3.3, we describe in detail the syntax of Bilattice-Based Annotated Languages. In Section 3.4 we discuss semantics suitable for these languages. And in Section 3.5 we restrict these languages to logic programs (BAPs). The Section 3.6 about unification will provide a link between the current chapter and Chapter 4. We will make some conclusions in Section 3.7.

The material of this chapter first appeared in [129], but then was used, for different purposes, and further refined in [119, 127, 128, 126, 127, 123]. The paper [123] is particularly useful in this respect.

3.2 Bilattices

3.2.1 Basic definitions

Bilattices were first introduced by M. Ginsberg in [65]. We follow Ginsberg's definition of bilattices, as given in [65], and also include some useful lattice-theoretic definitions from [68, 16].

We start with conventional lattice-theoretic definitions.

Definition 3.2.1. *Let S be a set. A binary* relation *R on S is a subset of $S \times S$.*

We usually use infix notation writing $(x, y) \in R$ as xRy.

Definition 3.2.2. *A relation R on a set S is a* partial order *if R is reflexive, antisymmetric and transitive, that is, the following conditions are satisfied:*

- *xRx, for all $x \in S$;*
- *xRy and yRx imply $x = y$, for all $x, y \in S$;*
- *xRy and yRz imply xRz, for all $x, y, z \in S$.*

Example 3.2.1. *In Chapter 1, Section 1.5 we worked with the lattice 2^{B_P} of all possible interpretations for a logic program P. We actually used the fact that, given any set S, one can form the set 2^S of all subsets of S. The set theoretic inclusion $\subseteq$ defined on 2^S satisfies all the conditions of Definition 3.2.2, and thus 2^S is a partial order.*

We will use the symbol $\leq$ to denote partial orders.

Definition 3.2.3. *Let S be a set, and let $\leq$ be a binary relation on S. The pair $(S, \leq)$ is called a* partially ordered set *if $\leq$ is a partial order on S.*

Definition 3.2.4. *Let $(S, \leq)$ be a partially ordered set and let $A \subseteq S$. An element $a \in S$ is called an* upper bound *for A if $x \leq a$ for all $x \in A$. An upper bound a for A is called the* least upper bound *for A, written* $\operatorname{lub} A$*, if for any upper bound a' for A we have $a \leq a'$. An element $b \in S$ is called a* lower bound *for A if $b \leq x$ for all $x \in A$. A lower bound b for A is called the* greatest lower bound *for A, written* $\operatorname{glb} A$*, if for any lower bound b' for A we have $b' \leq b$.*

It can be easily seen that the greatest and the least upper bounds for any set A are unique whenever they exist.

Definition 3.2.5. *A* lattice *L is a pair $(\mathcal{L}, \leq)$, where $\mathcal{L}$ is a set and $\leq$ is a partial order defined on $\mathcal{L}$ such that both* $\operatorname{lub}\{a, b\}$ *and* $\operatorname{glb}\{a, b\}$ *exist for all elements $a, b \in \mathcal{L}$. We usually write* $\operatorname{lub}\{a, b\}$ *and* $\operatorname{glb}\{a, b\}$ *as* $\operatorname{lub}(a, b)$ *respectively* $\operatorname{glb}(a, b)$*. We call a lattice* $(\mathcal{L}, \leq)$ complete *if both* $\operatorname{lub} A$ *and* $\operatorname{glb} A$ *exist for all subsets A of $\mathcal{L}$.*

Given a lattice $L = (\mathcal{L}, \leq)$, we can associate with L two binary operations $\cup$ and $\cap$ from $\mathcal{L} \times \mathcal{L}$ to $\mathcal{L}$ by setting $a \cap b = \operatorname{glb}(a, b)$ and $a \cup b = \operatorname{lub}(a, b)$. The following theorem can then be established, see [16, 68].

Theorem 3.2.1. *Let L be a lattice, and let $a \cap b = \operatorname{glb}(a, b)$ and $a \cup b = \operatorname{lub}(a, b)$. Then $\cup$ and $\cap$ are* idempotent*,* commutative *and* associative *binary operations from $\mathcal{L} \times \mathcal{L}$ to $\mathcal{L}$. That is, for any elements $a, b, c \in \mathcal{L}$ the following axioms hold:*

1. $a \cap a = a \cup a = a$,

2. $a \cap b = b \cap a$,
   $a \cup b = b \cup a$,

3. $(a \cap b) \cap c = a \cap (b \cap c)$,
   $(a \cup b) \cup c = a \cup (b \cup c)$.

4. *Additionally, in each lattice the* absorption identities *hold, that is,* $a \leq c$ *if and only if* $a \cap c = \text{glb}(a, c) = a$ *and* $a \cup c = \text{lub}(a, c) = c$.

   The absorption law can equivalently be captured by the following identities: $a \cup (a \cap b) = a$, $a \cap (a \cup b) = a$.

We call $\cup$ the *join*, and $\cap$ the *meet* operation of the lattice.

Definition 3.2.6. *A lattice* L *is called* distributive *if* $\cup$ *and* $\cap$ *distribute with respect to each other, that is, for all* $a, b, c \in \mathcal{L}$ *the following identities hold:*
$a \cap (b \cup c) = (a \cap b) \cup (a \cap c)$
$a \cup (b \cap c) = (a \cup b) \cap (a \cup c)$.

Notice that there are two ways of defining a lattice, see [16, 68]. The first of them starts with Definition 3.2.5. From this definition, operations $\cup, \cap$ are defined and their properties derived as stated in Theorem 3.2.1. The other way is to define two idempotent, commutative and associative binary operations on a set. These operations induce a partial order in which glb and lub are defined for any two elements of the set, giving us a lattice in the sense of Definition 3.2.5.

Theorem 3.2.2. *[68]*

1. *Let a partially ordered set* $L = (\mathcal{L}, \leq)$ *be a lattice, and let* $a \cap b = \text{glb}(a, b)$, $a \cup b = \text{lub}(a, b)$. *Then an algebra* $L^a = (\mathcal{L}, \cap, \cup)$ *is a lattice.*

2. *Let an algebra* $L = (\mathcal{L}, \cap, \cup)$ *be a lattice. And let* $a \leq b$ *iff* $a \cup b = a$. *Then* $L^p = (\mathcal{L}, \leq)$ *is a partially ordered set and the partially ordered set* L^p *is a lattice.*

3. *If a partially ordered set* $L = (\mathcal{L}, \leq)$ *is a lattice, then* $(L^{a)^p} = L$.

4. *If an algebra* $L = (\mathcal{L}, \cap, \cup)$ *is a lattice, then* $(L^{p)^a} = L$.

The same observations apply to bilattices. Originally, Ginsberg [65] introduced bilattices as sets carrying an operation of negation together with four binary operations corresponding to meet and join with respect to the two different orderings. On the other hand, Fitting defined bilattices as sets with two partial orderings and an operation of negation, see [49]. We give here both definitions, and refer either to orderings or to operations when giving particular examples of bilattices.

Definition 3.2.7. *A* bilattice **B** *is a sextuple* $(\mathcal{B}, \vee, \wedge, \oplus, \otimes, \neg)$ *such that*

1. $(\mathcal{B}, \vee, \wedge)$ *and* $(\mathcal{B}, \oplus, \otimes)$ *are both complete lattices,*

2. $\neg : \mathcal{B} \to \mathcal{B}$ *is a mapping with*

   - $\neg^2 = Id_{\mathcal{B}}$, *and*
   - $\neg$ *is a dual lattice homomorphism from* $(\mathcal{B}, \vee, \wedge)$ *to* $(\mathcal{B}, \wedge, \vee)$ *and a mapping from* $(\mathcal{B}, \oplus, \otimes)$ *to itself.*

Definition 3.2.8. *A* bilattice **B** *is a structure* $(\mathcal{B}, \leq_t, \leq_k, \neg)$ *consisting of a nonempty set* $\mathcal{B}$ *together with two partial orderings* $\leq_t, \leq_k$ *on* $\mathcal{B}$ *and a mapping* $\neg$ *from* $\mathcal{B}$ *to itself such that for all* $x, y \in \mathcal{B}$ *the following axioms hold.*

1. $(\mathcal{B}, \leq_t)$ *and* $(\mathcal{B}, \leq_k)$ *are both complete lattices.*

2. $x \leq_t y$ *implies* $\neg y \leq_t \neg x$.

3. $x \leq_k y$ *implies* $\neg x \leq_k \neg y$.

4. $\neg\neg x = x$.

Note 3.2.1. *Note that* $(\mathcal{B}, \wedge, \vee)$ *corresponds to* $(\mathcal{B}, \leq_t)$ *and* $(\mathcal{B}, \oplus, \otimes)$ *corresponds to* $(\mathcal{B}, \leq_k)$.

The following definitions will be useful in the next section.

Definition 3.2.9. *Bilattices* $\mathbf{B}_1 = (\mathcal{B}_1, \leq_k^1, \leq_t^1, \neg_1)$ *and* $\mathbf{B}_2 = (\mathcal{B}_2, \leq_k^2, \leq_t^2, \neg_2)$ *are called* isomorphic *iff there exists a surjective function* φ*, mapping* $\mathcal{B}_1$ *onto* $\mathcal{B}_2$ *such that*

$$a \leq_k^1 b \in \mathbf{B}_1 \iff \varphi(a) \leq_k^2 \varphi(b) \in \mathbf{B}_2,$$
$$a \leq_t^1 b \in \mathbf{B}_1 \iff \varphi(a) \leq_t^2 \varphi(b) \in \mathbf{B}_2,$$

and φ *preserves properties of* $\neg$ *from Definition 3.2.8 when mapping* $\neg_1$ *to* $\neg_2$*. The function* φ *is called an isomorphism from* $\mathbf{B}_1$ *to* $\mathbf{B}_2$.

We may equivalently define isomorphism as follows:

Definition 3.2.10. *Bilattices* $\mathbf{B}_1 = (\mathcal{B}_1, \vee_1, \wedge_1, \oplus_1, \otimes_1, \neg_1)$ *and* $\mathbf{B}_2 = (\mathcal{B}_2, \vee_2, \wedge_2, \oplus_2, \otimes_2, \neg_2)$ *are called* isomorphic *iff there exists a surjective function* φ*, mapping* $\mathcal{B}_1$ *onto* $\mathcal{B}_2$ *such that*

$$\begin{aligned} a \wedge_1 b &= \varphi(a) \wedge_2 \varphi(b), \\ a \vee_1 b &= \varphi(a) \vee_2 \varphi(b), \\ a \otimes_1 b &= \varphi(a) \otimes_2 \varphi(b), \\ a \oplus_1 b &= \varphi(a) \oplus_2 \varphi(b), \\ \neg_1 a &= \neg_2 \varphi(a). \end{aligned}$$

The function φ is called an isomorphism from $\mathbf{B}_1$ to $\mathbf{B}_2$.

We have defined bilattices in general, and now we are ready to discuss properties of two particular classes of bilattices that we will use when building bilattice-based logic programs.

3.2.2 Interlaced and Distributive Bilattices

Analogously to conventional lattice theory, where distributive lattices are distinguished for their nice and well-defined properties, distributive bilattices also have some properties which distinguish them as an interesting class of bilattices to consider.

A bilattice satisfying the condition that the meet and join operations for each partial ordering are monotonic with respect to the other ordering is called an *interlaced bilattice*. Interlaced bilattices share many properties with distributive bilattices, but satisfy somewhat weaker conditions. This is why we start with the definition of interlaced bilattices.

Monotonicity of the bilattice operations will play an important role when we give an interpretation for bilattice-based languages, and in particular, we will use monotonicity when proving a very important Proposition 3.4.1.

Recall that, in each lattice, the operations $\cup$ and $\cap$ are monotonic. In [16], it is proven that $\cup$ and $\cap$ are isotone, that is, $a \leq b$ implies $a \cap c \leq b \cap c$ and $a \cup c \leq b \cup c$. We will prove monotonicity here:

Proposition 3.2.1. *In each lattice $(L, \leq)$, both $\cap$ and $\cup$ are monotonic with respect to $\leq$, that is, for all $a, b, c, d \in L$, if $a \leq b$ and $c \leq d$, then*

$$a \cap c \leq b \cap d, \text{ and } a \cup c \leq b \cup d.$$

Proof. We give a proof for $\cap$.

If $a \leq b$, then, according to the absorption law, see Theorem 3.2.1, Item 4, $a \cap b = a$. If $c \leq d$, then, according to the absorption law, $c \cap d = c$. Then, $a \cap c = (a \cap b) \cap (c \cap d)$. By associativity and commutativity, $(a \cap b) \cap (c \cap d) = (a \cap c) \cap (b \cap d)$. And so, $a \cap c = (a \cap c) \cap (b \cap d)$. And, by the absorption law, the latter implies that $(a \cap c) \leq (b \cap d)$.

The proof for $\cup$ is dual. □

If we require monotonicity of bilattice operations with respect to both bilattice orderings, we will obtain an interlaced bilattice.

Definition 3.2.11. *An* interlaced bilattice *is a structure* $\langle \mathcal{B}, \leq_t, \leq_k, \neg \rangle$ *where:*

- *both* $\leq_t$ *and* $\leq_k$ *give* $\mathcal{B}$ *the structure of a lattice; and for all* $a, b, c, d \in \mathcal{B}$,
- $a \leq_t b$ *and* $c \leq_t d$ *implies* $a \otimes c \leq_t b \otimes d$ *and* $a \oplus c \leq_t b \oplus d$;
- $a \leq_k b$ *and* $c \leq_k d$ *implies* $a \wedge c \leq_k b \wedge d$ *and* $a \vee c \leq_t b \vee d$.

We will return to this definition later, when we establish a uniform way of how interlaced bilattices can be generated. The next useful property of bilattices is distributivity.

Definition 3.2.12. *The bilattice* $\mathbf{B}$ *will be called* t-distributive *if the* t*-lattice* $(\mathcal{B}, \vee, \wedge)$ *is distributive, and it will be called* k-distributive *if the* k*-lattice* $(\mathcal{B}, \oplus, \otimes)$ *is distributive. It will be called* cross-distributive *if each of* $\wedge$ *and* $\vee$ *distributes with respect to* $\otimes$ *or* $\oplus$. *A bilattice which is k-distributive, t-distributive and cross-distributive will be called* distributive. *Twelve distributivity laws hold in any distributive bilattice, that is, for any three elements* a, b, c *of the bilattice* $\mathbf{B}$, *the following statements hold:*

$$
\begin{aligned}
a \wedge (b \vee c) &= (a \wedge b) \vee (a \wedge c), \\
a \vee (b \wedge c) &= (a \vee b) \wedge (a \vee c), \\
a \oplus (b \otimes c) &= (a \oplus b) \otimes (a \oplus c), \\
a \otimes (b \oplus c) &= (a \otimes b) \oplus (a \otimes c), \\
a \oplus (b \vee c) &= (a \oplus b) \vee (a \oplus c), \\
a \oplus (b \wedge c) &= (a \oplus b) \wedge (a \oplus c), \\
a \otimes (b \vee c) &= (a \otimes b) \vee (a \otimes c), \\
a \otimes (b \wedge c) &= (a \otimes b) \wedge (a \otimes c), \\
a \vee (b \oplus c) &= (a \vee b) \oplus (a \vee c), \\
a \vee (b \otimes c) &= (a \vee b) \otimes (a \vee c), \\
a \wedge (b \oplus c) &= (a \wedge b) \oplus (a \wedge c), \\
a \wedge (b \otimes c) &= (a \wedge b) \otimes (a \wedge c).
\end{aligned}
$$

The smallest nontrivial bilattice arises from Belnap's four-valued truth set, see figure 3.1.

This bilattice can be defined as $B = (\{0, 1, \bot, \top\}, \leq_t, \leq_k, \neg)$, with $\leq_t$ and $\leq_k$ such that: $0 \leq_t \bot \leq_t 1$, $0 \leq_t \top \leq_t 1$, $\bot \leq_k 0 \leq_k \top$, $\bot \leq_k 1 \leq_k \top$. Then $\oplus, \otimes$ are meet and join with respect to the k-ordering, and $\vee, \wedge$ are meet and join with respect to the t-ordering. This bilattice is interlaced and distributive; its properties and applications are very carefully examined in [49, 50, 51, 54].

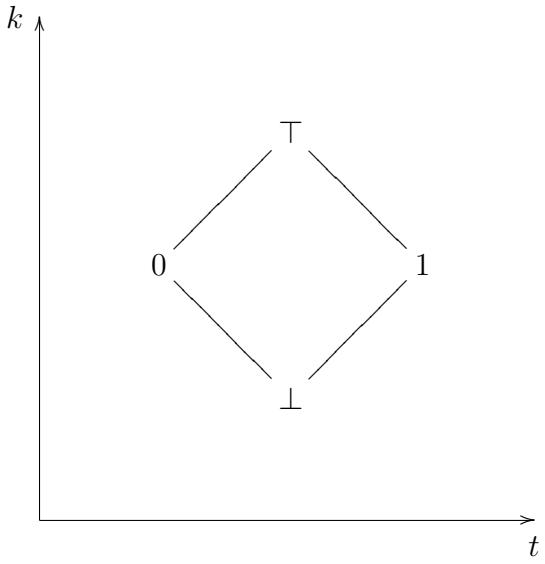


Figure 3.1: Belnap's bilattice

In general, the number of elements of a bilattice is arbitrary. In particular, an infinite bilattice consists of an infinite number of elements, with two partial orderings and negation defined on it. We denote the infinite operations as follows: $\bigvee, \bigwedge$ denote infinite join and meet with respect to the t-ordering, $\sum, \prod$ denote infinite join and meet with respect to the k-ordering.

Definition 3.2.13. *A bilattice satisfies the* infinite distributivity conditions *if the following equalities hold.*

$$a \wedge (\bigcirc_i b_i) = \bigcirc_i (a \wedge b_i),$$

$$a \vee (\bigcirc_i b_i) = \bigcirc_i (a \vee b_i),$$

$$a \otimes (\bigcirc_i b_i) = \bigcirc_i (a \otimes b_i),$$

$$a \oplus (\bigcirc_i b_i) = \bigcirc_i (a \oplus b_i),$$

where $\bigcirc$ *is any one of the four operations* $\bigwedge$, $\bigvee$, $\sum$, $\prod$.

There exists a very convenient method to generate distributive bilattices, and we will devote the rest of this subsection to discussions of this method and its applications.

The method is very simple: one needs to define a bilattice as a product of two distinct lattices. We can regard one lattice as denoting our measure of confidence for a fact, and the other as denoting our measure of confidence against it. The resulting bilattice structure may provide a space of interpretations appropriate for world-based semantics [65], many-valued semantics [49], probabilistic semantics [167, 166]. The bilattices of this type were proven to be interlaced and, if both lattices are distributive, the bilattice formed out of them is also distributive.

The details are as follows.

Let $L_1 = (\mathcal{L}_1, \leq_1)$ and $L_2 = (\mathcal{L}_2, \leq_2)$ be two lattices, and let x_1, x_2 denote arbitrary elements of the lattice L_1, and y_1, y_2 denote arbitrary elements of the lattice L_2. Let $\cup_1, \cap_1$ be join and meet defined on the lattice L_1, and $\cup_2, \cap_2$ be join and meet defined on the lattice L_2.

Definition 3.2.14. *Suppose $L_1 = (\mathcal{L}_1, \leq_1)$ and $L_2 = (\mathcal{L}_2, \leq_2)$ are complete lattices. Form the set of points $\mathcal{L}_1 \times \mathcal{L}_2$, and define the two orderings $\leq_t$ and $\leq_k$ on $\mathcal{L}_1 \times \mathcal{L}_2$ as follows.*
$\langle x_1, y_1\rangle \leq_t \langle x_2, y_2\rangle$ iff $x_1 \leq_1 x_2$ and $y_2 \leq_2 y_1$.
$\langle x_1, y_1\rangle \leq_k \langle x_2, y_2\rangle$ iff $x_1 \leq_1 x_2$ and $y_1 \leq_2 y_2$.

We denote this structure by $L_1 \odot L_2 = (\mathcal{L}_1 \times \mathcal{L}_2, \leq_t, \leq_k) = (\mathcal{B}, \leq_t, \leq_k)$, where $\mathcal{B}$ denotes $\mathcal{L}_1 \times \mathcal{L}_2$.

Having defined $L_1 \odot L_2$, we go on to define the bilattice operations as follows.

Definition 3.2.15. *The four bilattice operations associated with $\leq_t$ and $\leq_k$ are:*
$\langle x_1, y_1\rangle \wedge \langle x_2, y_2\rangle = \langle x_1 \cap_1 x_2, y_1 \cup_2 y_2\rangle$,
$\langle x_1, y_1\rangle \vee \langle x_2, y_2\rangle = \langle x_1 \cup_1 x_2, y_1 \cap_2 y_2\rangle$,

$\langle x_1, y_1\rangle \otimes \langle x_2, y_2\rangle = \langle x_1 \cap_1 x_2, y_1 \cap_2 y_2\rangle$,

$\langle x_1, y_1\rangle \oplus \langle x_2, y_2\rangle = \langle x_1 \cup_1 x_2, y_1 \cup_2 y_2\rangle$.

Now, using Definitions 3.2.14 and 3.2.15, one can prove that in the product bilattices, all the four bilattice operations are monotonic with respect to both bilattice orderings, in other words, we can show that the product bilattices are interlaced. This result was first stated by Fitting in [49] with no proof given, and we cite his proposition and then give our own proof of it.

Proposition 3.2.2. *[49] If L_1 and L_2 are lattices, then $L_1 \odot L_2$ is an interlaced bilattice. Furthermore, if $L_1 = L_2$, then the operation given by $\neg\langle x, y\rangle = \langle y, x\rangle$ satisfies the conditions of negation in Definition 3.2.7 and Definition 3.2.8.*

Proof. We give the proof of the first statement saying that $L_1 \odot L_2$ is interlaced.

We will give a proof that $\wedge$ is monotonic with respect to $\leq_k$, the proofs that $\vee$ is monotonic with respect to $\leq_k$, and $\oplus, \otimes$ are monotonic with respect to $\leq_t$ are similar to it.

Let $a_1, b_1, c_1, d_1 \in L_1$ and $a_2, b_2, c_2, d_2 \in L_2$. Suppose $(a_1, a_2) \leq_k (b_1, b_2)$ and $(c_1, c_2) \leq_k (d_1, d_2)$. By Definition 3.2.14, this means that $a_1 \leq_1 b_1$, $a_2 \leq_2 b_2$, $c_1 \leq_1 d_1$, and $c_2 \leq_2 d_2$. By monotonicity of $\cup$ and $\cap$ in L_1 and L_2, we can conclude that $(a_1 \cap_1 c_1) \leq_1 (b_1 \cap_1 d_1)$, and $(a_2 \cup_2 c_2) \leq_2 (b_2 \cup_2 d_2)$. But then, using Definition 3.2.14, we conclude that $\langle (a_1 \cap_1 c_1), (a_2 \cup_2 c_2)\rangle \leq_k \langle (b_1 \cap_1 d_1), (b_2 \cup_2 d_2)\rangle$. But then, by Definition 3.2.15, this means that $(a_1, a_2) \wedge (c_1, c_2) \leq_k (b_1, b_2) \wedge (d_1, d_2)$. □

Proposition 3.2.2 will be very useful in Section 3.4, and in particular we will use it when proving Proposition 3.4.1.

Proposition 3.2.3 ([65, 49]). *Suppose* $\mathbf{B}$ *is a distributive bilattice. Then there are distributive lattices* L_1 *and* L_2 *such that* $\mathbf{B}$ *is isomorphic to* $L_1 \odot L_2$.

Thus, every distributive bilattice can be represented as a product of two lattices. In this thesis, we will consider only logic programs whose underlying bilattices are distributive, and therefore formed as the product of two lattices.

The smallest distributive bilattice of Figure 3.1, also known as Belnap's bilattice, is isomorphic to the bilattice $\mathbf{B} = L_1 \odot L_2$, where $L_1 = L_2 = (\{0, 1\}, 0 \leq 1)$. The least and greatest elements with respect to the t-ordering are $\langle 0, 1\rangle$ and $\langle 1, 0\rangle$, the least and greatest elements with respect to the k-ordering are $\langle 0, 0\rangle$, $\langle 1, 1\rangle$. The greatest distributive bilattice we consider in this thesis is $L_1 \odot L_2$ based on the infinite lattices $L_1 = L_2 = [0, 1]$, where $[0, 1]$ is the unit interval of reals ordered as usual. There is an infinite number of intermediate bilattices. Consider, for example, the following bilattices.

Example 3.2.2. *Let* $L_1 = L_2 = (\{0, \frac{1}{2}, 1\}, \leq)$, *with* $0 \leq \frac{1}{2} \leq 1$. *The set* $\mathcal{B}_9 = \mathcal{L}_1 \times \mathcal{L}_2$ *consists of 9 elements:*

$$\{\langle 0, 0\rangle, \langle 0, \frac{1}{2}\rangle, \langle 0, 1\rangle, \langle \frac{1}{2}, 0\rangle, \langle \frac{1}{2}, \frac{1}{2}\rangle, \langle \frac{1}{2}, 1\rangle, \langle 1, 0\rangle, \langle 1, \frac{1}{2}\rangle, \langle 1, 1\rangle\}.$$

See figure 3.2.

Example 3.2.3. *Let* $L_1 = L_2 = (\{0, \frac{1}{3}, \frac{2}{3}, 1\}, \leq)$, *with* $0 \leq \frac{1}{3} \leq \frac{2}{3} \leq 1$. *The set* $\mathcal{B}_{16} = \mathcal{L}_1 \times \mathcal{L}_2$ *consists of 16 elements:*

$$\begin{aligned}
&\{\langle 0, 0\rangle, \langle 0, \frac{1}{3}\rangle, \langle 0, \frac{2}{3}\rangle, \langle 0, 1\rangle,\\
&\langle \frac{1}{3}, 0\rangle, \langle \frac{1}{3}, \frac{1}{3}\rangle, \langle \frac{1}{3}, \frac{2}{3}\rangle, \langle \frac{1}{3}, 1\rangle,\\
&\langle \frac{2}{3}, 0\rangle, \langle \frac{2}{3}, \frac{1}{3}\rangle, \langle \frac{2}{3}, \frac{2}{3}\rangle, \langle \frac{2}{3}, 1\rangle,\\
&\langle 1, 0\rangle, \langle 1, \frac{1}{3}\rangle, \langle 1, \frac{2}{3}\rangle, \langle 1, 1\rangle\}.
\end{aligned}$$

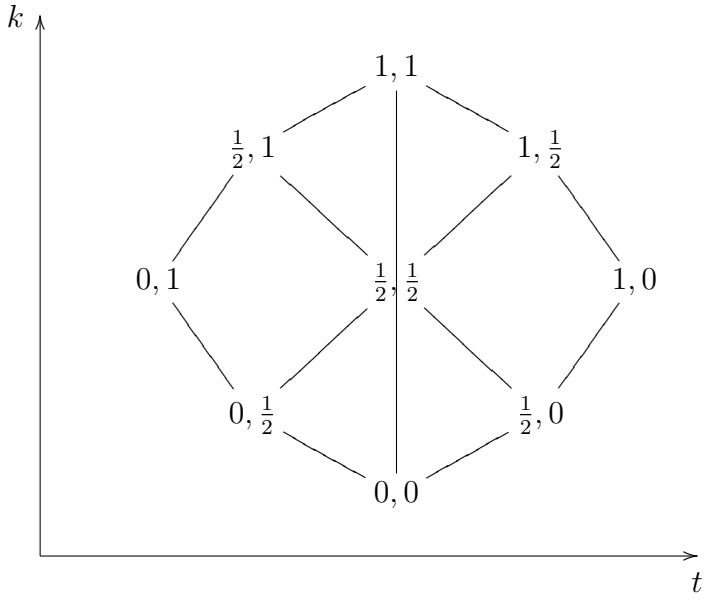


Figure 3.2: Bilattice $\mathbf{B_9}$

See figure 3.3.

Example 3.2.4. *Finally, consider the bilattice, consisting of* 25 *elements. Let* $L_1 = L_2 = (\{0, \frac{1}{4}, \frac{1}{2}, \frac{3}{4}, 1\}, \leq)$, *with* $0 \leq \frac{1}{4} \leq \frac{1}{2} \leq \frac{3}{4} \leq 1$. *The set* $\mathcal{B}_{25} = \mathcal{L}_1 \times \mathcal{L}_2$ *consists of the following elements:*

$$\begin{aligned}
&\{\langle 0,0\rangle, \langle 0,\frac{1}{4}\rangle, \langle 0,\frac{1}{2}\rangle, \langle 0,\frac{3}{4}\rangle, \langle 0,1\rangle,\\
&\langle \frac{1}{4},0\rangle, \langle \frac{1}{4},\frac{1}{4}\rangle, \langle \frac{1}{4},\frac{1}{2}\rangle, \langle \frac{1}{4},\frac{3}{4}\rangle, \langle \frac{1}{4},1\rangle,\\
&\langle \frac{1}{2},0\rangle, \langle \frac{1}{2},\frac{1}{4}\rangle, \langle \frac{1}{2},\frac{1}{2}\rangle, \langle \frac{1}{2},\frac{3}{4}\rangle, \langle \frac{1}{2},1\rangle,\\
&\langle \frac{3}{4},0\rangle, \langle \frac{3}{4},\frac{1}{4}\rangle, \langle \frac{3}{4},\frac{1}{2}\rangle, \langle \frac{3}{4},\frac{3}{4}\rangle, \langle \frac{3}{4},1\rangle,\\
&\langle 1,0\rangle, \langle 1,\frac{1}{4}\rangle, \langle 1,\frac{1}{2}\rangle, \langle 1,\frac{3}{4}\rangle, \langle 1,1\rangle\}.
\end{aligned}$$

See Figure 3.4.

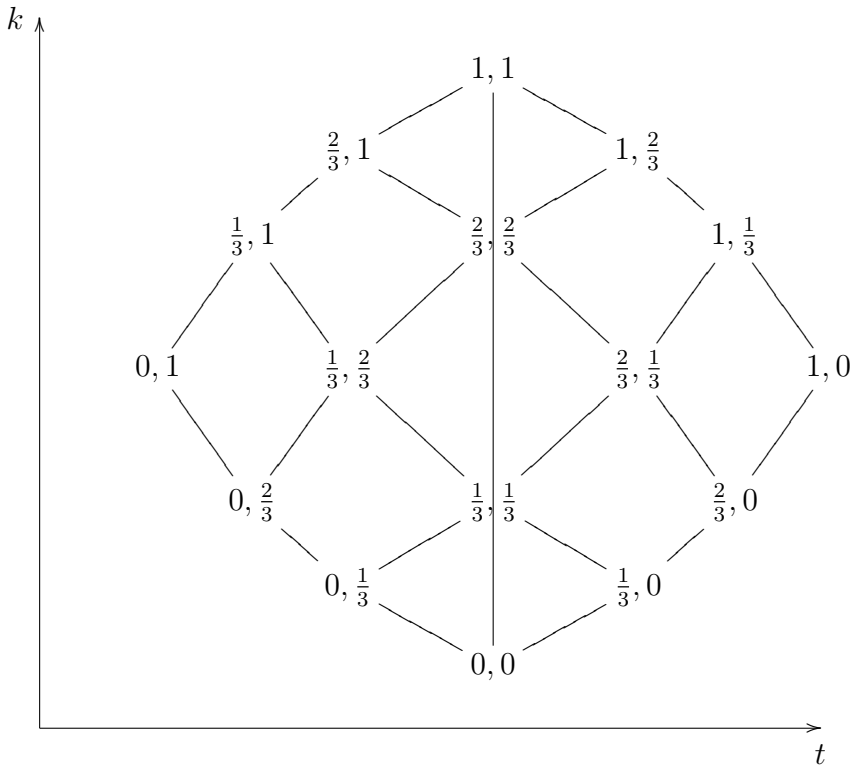


Figure 3.3: Bilattice $\mathbf{B}_{16}$

All three examples display distributive and interlaced bilattices formed from lattices which are subsets of the unit interval. Each carries two orderings $\leq_k$ and $\leq_t$ as given in Definition 3.2.14. For any element $\langle a, b\rangle$ of such a bilattice, $\neg\langle a, b\rangle = \langle b, a\rangle$.

Negation as it was given in Definitions 3.2.7 and 3.2.8 forces us to consider only structures where $L_1 = L_2$, and which are symmetric with respect to the $\langle 0, 0\rangle - \langle 1, 1\rangle$-axis. However, if a structure $L_1 \odot L_2$ is defined without negation, we can use asymmetric bilattice-like structures with orderings $\leq_k$, $\leq_t$ and operations $\oplus, \otimes, \vee, \wedge$ as given in Definitions 3.2.14 and 3.2.15, see §4.2.

Example 3.2.5. *Let* L_1 *be* $\{0, \frac{1}{2}, 1\}$ *with the ordering* $0 \leq \frac{1}{2} \leq 1$*, and let* L_2 *be* $\{0, \frac{1}{3}, \frac{2}{3}, 1\}$

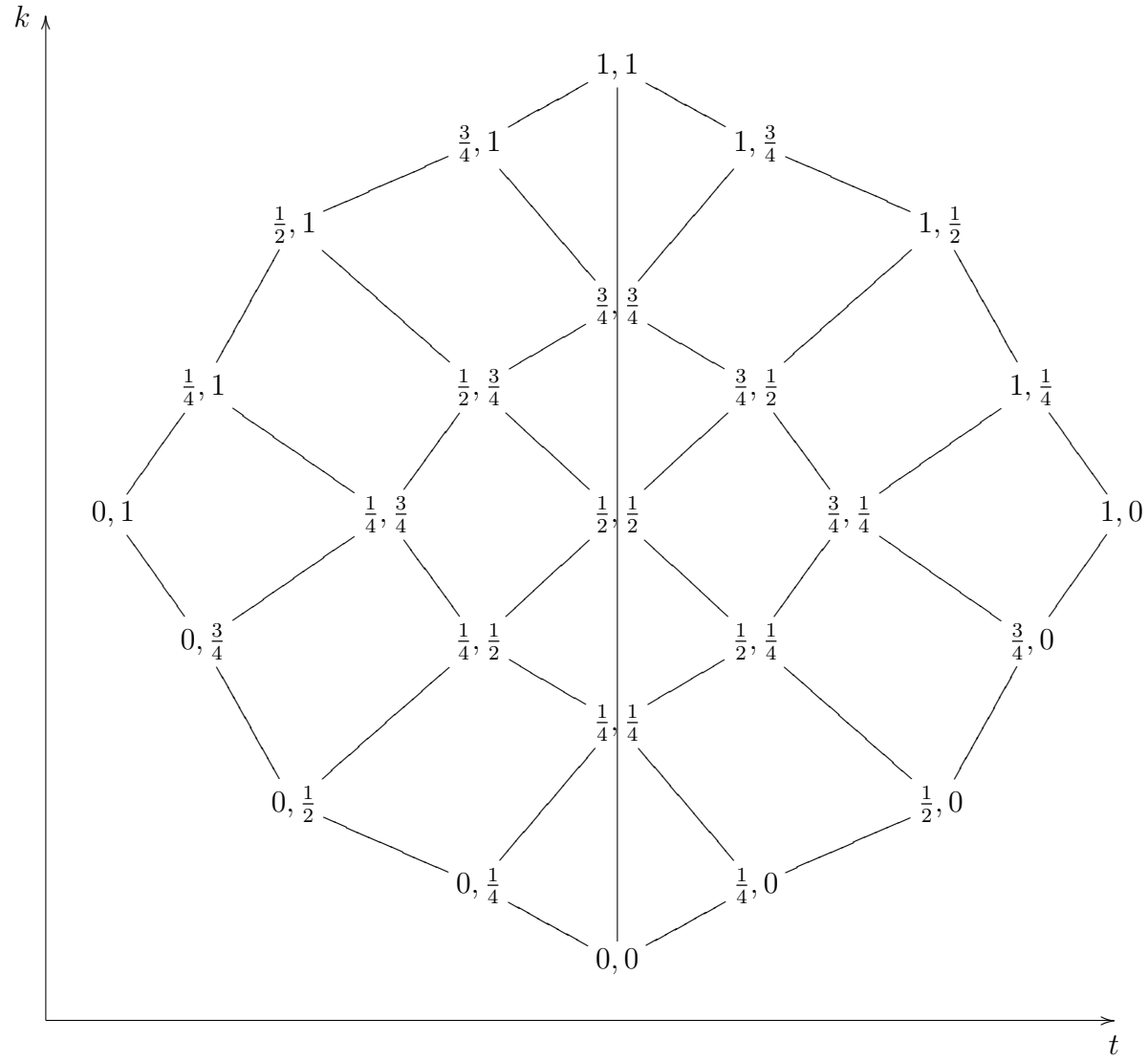


Figure 3.4: Bilattice $\mathbf{B}_{25}$

with the ordering $0 \leq \frac{1}{3} \leq \frac{2}{3} \leq 1$. *The set* $\mathcal{B}_{12} = \mathcal{L}_1 \times \mathcal{L}_2$ *consists of the following 12 elements:*

$$\{\langle 0,0\rangle, \langle 0,\frac{1}{3}\rangle, \langle 0,\frac{2}{3}\rangle, \langle 0,1\rangle, \langle \frac{1}{2},0\rangle, \langle \frac{1}{2},\frac{1}{3}\rangle, \langle \frac{1}{2},\frac{2}{3}\rangle, \langle \frac{1}{2},1\rangle, \langle 1,0\rangle, \langle 1,\frac{1}{3}\rangle, \langle 1,\frac{2}{3}\rangle, \langle 1,1\rangle\}.$$

See Figure 3.5.

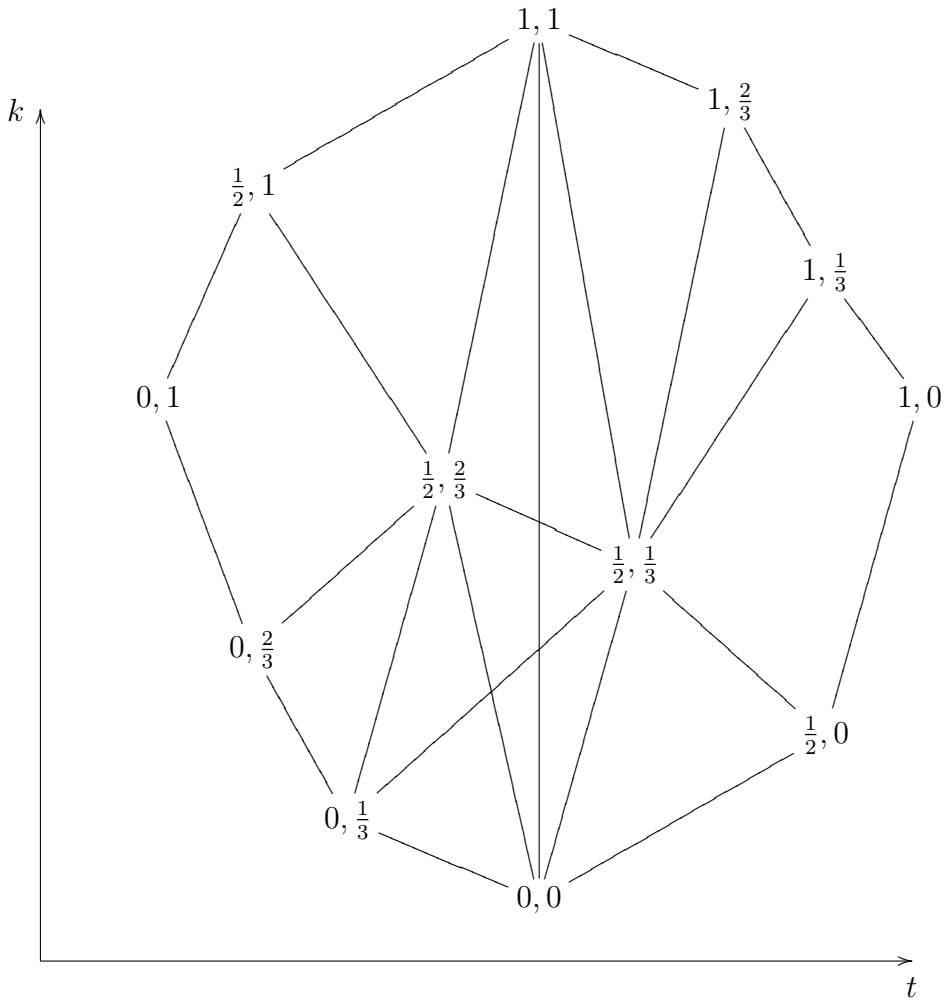


Figure 3.5: Bilattice $\mathbf{B}_{12}$

The negation as given by Definitions 3.2.7 and 3.2.8 cannot be defined on this bilattice-like structure. Thus, this asymmetric structure does not satisfy the definition of a bilattice as given by Ginsberg. But the question of different types of complements defined on

product bilattice-like structures has attracted some attention, see [50, 55].

The asymmetric bilattice-like structures can still capture some intuitions. For example, we may wish to work with non-equal measures of confidence and doubt. We will, therefore, consider one example of definite program based on an asymmetric bilattice, see Section 3.5, Example 3.5.4.

3.3 First-Order Annotated Languages Based on Bilattice Structures

In Chapter 2 we showed three ways in which conventional logic programming was extended to many-valued logic programming. And in particular, we emphasised the fact that annotated programs were proven to be in general more expressive than annotation-free or implication-based programs.

In this Section we develop the approach of [107, 108, 141, 142, 74, 183] and attach annotations to first-order formulae, such that a typical annotated atom of the language will be of the form $A : \tau$, where A is a first-order atom, and τ represents some annotation taken from a set of truth-values. Complex formulae can be built up using these atoms.

We propose a first-order bilattice-based annotated language (BAL). Although the papers we mentioned above provided us with many interesting insights and techniques, the language we define here is the first language for a full fragment of bilattices.

Syntactically, we follow [108] and allow not only constant annotations, as in signed logics ([141, 142, 74, 182, 183]), but also annotation variables and terms. We develop the annotated syntax of [108] and enrich it by new bilattice connectives and quantifiers and also we in general allow annotations over complex first-order formulae, which is a

reasonable option for annotated logics.

Semantically, we work with arbitrary distributive bilattices defined in [55, 65]. We believe that a one-lattice fragment of BAL can be described, with some syntactical restrictions, in terms of lattice-based annotated logics [74, 107, 108, 182, 183]. However, the second lattice constituting the underlying bilattice of BAL determines some novel and non-trivial properties of its models and theory, that are captured in Proposition 3.4.1 and further used in the next chapter, in Definitions 4.2.6 and 4.3.1.

We proceed as follows.

Let $\mathbf{B} = L_1 \odot L_2$ denote a bilattice given as the product of two complete lattices L_1, L_2, such that $L_1 = L_2$ and both are sublattices of the lattice $([0,1], \leq)$, where $[0,1]$ is the unit interval of real numbers and $\leq$ is the usual linear ordering on it. We define the annotated bilattice-based first-order language $\mathcal{L}$ as follows.

Definition 3.3.1 (Cf. Definition 1.2.1)**.** *An* annotated alphabet*, or just* alphabet*, contains*

1. *individual variable symbols* $x_1, x_2, \ldots$.
2. *individual constant symbols* $a_1, a_2, \ldots$.
3. *function symbols* $f_1, f_2, \ldots$ *of various arities greater than* 0.
4. *predicate symbols* $R_1, R_2, \ldots$ *of various arities greater than* 0.
5. *annotation constants* $(\alpha_1, \beta_1), (\alpha_2, \beta_2), \ldots \in \mathbf{B}$,
6. *annotation variables* $(\mu_1, \nu_1), (\mu_2, \nu_2), \ldots$ *taking values in* $\mathbf{B}$,
7. *total continuous annotation functions* $\vartheta_1, \vartheta_2, \ldots$ *of type* $\mathbf{B}^i \to \mathbf{B}$.

   We assume that each function ϑ_k *is computable, i.e., there is a uniform procedure*

P_{ϑ_k}, such that if ϑ_k is n-ary and $((\alpha_1, \beta_1), \ldots (\alpha_n, \beta_n))$ are given as an input to P_{ϑ_k}, then $\vartheta_k((\alpha_1, \beta_1), \ldots (\alpha_n, \beta_n))$ is an output of P_{ϑ_k} obtained in a finite amount of time.

8. *connectives $\vee, \wedge, \oplus, \otimes$, which correspond to join and meet with respect to the t- and k- orderings; $\neg$, which correspond to $\neg$ defined on bilattices, and $\sim$.*

9. *quantifiers $\forall, \exists, \Pi, \Sigma$, which correspond to infinite meet and join with respect to t- and k- orderings respectively.*

10. *punctuation symbols "(", "," and ")".*

Annotation constants and variables are taken as pairs of constants respectively pairs of variables interpreted in bilattices; the first and second elements of a pair accumulate belief for respectively against a fact.

Note that $\sim$ is alternative to bilattice negation $\neg$, the former is usually called "ontological" negation in comparison with the latter negation, which is called "epistemic". Epistemic negation ($\neg(A : (\alpha, \beta) = A : (\beta, \alpha)$) switches degrees of belief and doubt in annotations and thus enables us to work with degrees of incomplete or inconsistent knowledge. This is why it is essentially many-valued. Ontological negation ($\sim (A : (\alpha, \beta))$ asserts that the negated and already annotated expression, i.e., a proposition about the degrees of confidence and doubt, does not hold. This annotated proposition, which is a kind of a fact itself, can only be true or false, and thus $\sim$ is essentially two-valued. For careful examination of the properties of both negations see, for example, [107]. In our language $\sim$ is defined as the restriction of $\neg$ to the set $\{\langle 1, 0\rangle, \langle 0, 1\rangle\}$.

Definition 3.3.2. *A* term *is defined inductively as follows:*

- *An individual variable symbol is a term.*

- *An individual constant symbol is a term.*
- *If f is an n-ary function symbol and $t_1, \ldots, t_n$ are terms, then $f(t_1, \ldots, t_n)$ is a term.*

Definition 3.3.3. *An* annotation term *is defined inductively as follows:*

- *an annotation constant is an annotation term,*
- *an annotation variable is an annotation term,*
- *if ϑ is an n-ary annotation function and $\tau_1, \ldots, \tau_n$ are annotation terms, then $\vartheta(\tau_1, \ldots, \tau_n)$ is an annotation term.*

Definition 3.3.4. *We will call formulae of the form $R(t_1, \ldots, t_n)$* atomic formulae *or* atoms *if R is an n-ary predicate symbol and $t_1, \ldots, t_n$ are terms.*

Definition 3.3.5. *An* annotated formula *is defined inductively as follows:*

- *If R is an n-ary predicate symbol, $t_1, \ldots, t_n$ are terms, and τ is an annotation term, then $R(t_1, \ldots, t_n) : (\tau)$ is an* annotated formula *(called an* annotated atomic formula *or an* annotated atom*).*
- *If F and G are annotated formulae, then so are $(F \vee G)$, $(F \wedge G)$, $(F \otimes G)$, $(F \oplus G)$, $(\neg F)$, $(\sim F)$.*
- *If F and G are formulae and τ is an annotation term, then $(F \vee G) : (\tau)$, $(F \wedge G) : (\tau)$, $(F \otimes G) : (\tau)$, $(F \oplus G) : (\tau)$ are annotated formulae.*
- *If F is an annotated formula and x is a variable symbol, then $(\forall x F)$, $(\exists x F)$, $(\Pi x F)$ and $(\Sigma x F)$ are annotated formulae.*

When working with annotation terms, we will sometimes use the notation $(\overline{\mu, \nu})$ instead of the symbol τ that we used to denote terms in the previous definitions. Whenever we do this, we will assume that we talk about the annotation term τ with free annotation variables $\overline{\mu, \nu}$.

Definition 3.3.6. *Let* $\mathbf{B}$ *be a distributive bilattice. The first order language* $\mathcal{L}$ *given by an alphabet consists of the set of all annotated formulae constructed from the symbols of the alphabet. We will refer to* $\mathbf{B}$ *as the* underlying bilattice *of the language* $\mathcal{L}$*. We will sometimes use the notation* $\mathcal{L}^{\mathbf{B}}$ *when we want to specify a particular choice of a bilattice underlying* $\mathcal{L}$*.*

Example 3.3.1. *Consider a binary predicate connected, which describes the fact of existence of an edge in a probabilistic graph, see also Example 1.3.3 for the two-valued and Example 2.0.1 for many-valued interpretations of it. Suppose we use probabilistic graphs to describe probabilities of establishing different connections in the internet. Then connected*$(a, b) : (\frac{1}{3}, \frac{2}{3})$ *will describe the fact that the probability of establishing a connection between nodes* a *and* b *is equal to* $\frac{1}{3}$*, while the probability of losing this connection is* $\frac{2}{3}$*. Then, for example, connected*$(a, b) : (\frac{1}{3}, \frac{2}{3}) \wedge (\mu, \nu)$ *is also a formula which contains a function* $\wedge$ *and a free variable* (μ, ν) *in its annotation.*

The language defined above can also be regarded as many-sorted or multimodal.

For the classical definition of a sorted first-order language, see, for example, [43]. For more detailed discussion of many-sorted languages, their expressiveness, applications and relations to first- and second-order logic, see [152] and in particular [147]. Many-sorted languages were introduced in order to allow constants and variables of different sorts, that is, from different domains of objects, to appear in formal languages. Once the domains of objects are chosen, the sorts of functions and predicates are computed in the natural way:

the sort of a function or of a predicate is determined using sequences of sorts of variables appearing in this function or predicate.

When defining the language $\mathcal{L}$, we allowed symbols of two sorts. One sort corresponds to the first-order component of our language and the other sort gives an account for annotation symbols which represent truth values. The latter sort enables us to work with degrees of belief and doubt within our language. However, because of the particular way in which annotated formulae were formed, the bilattice-based language $\mathcal{L}$ does not comply with the conventional definition of a sorted language.

This simple observation about inherent sortedness of the annotated language led us to establish that the bilattice-based first-order language (BAL) is representable in the conventional (two-valued) many-sorted language in the style of Manzano [147]. See [123] for a detailed discussion of it. Moreover, the method of the representation of BAL in many-sorted logic, which we introduced in [123], can be applied, with minor modifications, to any many-valued logic mentioned in Chapter 2. This result of [123] is the first formal proof that many-valued annotated languages are representable in the conventional two-valued many-sorted logic of [41].

Note that a similar division of expression of a language into two sorts can be found within a framework of probabilistic logic. See, for example, [9] for further discussion of it.

On the other hand, the annotations in the language play the role of modalities. That is, an annotated proposition of the form $R(x) : (\mu, \nu)$ receives its interpretation through interpreting the proposition $R(x)$. Thus, if the proposition $R(x)$ expresses information that x possesses the property R, the proposition $R(x) : (\mu, \nu)$ says that x is known to possess this property with degree of belief μ, and it is known to not satisfy this property with degree of belief ν. Since annotations in the language are taken from the set of

elements of the underlying bilattice, the number of these modalities will vary according to the chosen bilattice. As a consequence, the language $\mathcal{L}$ we have defined may be regarded as multimodal. For descriptions of different multimodal logics see, for example, [12, 11, 37, 60, 62]. But this relation of annotated and multi-modal logics has not yet received a formal account. One has to relate many-valued and modal semantics in order to establish the formal representation of annotated many-valued logics as multi-modal.

3.4 Interpretations

Let $\mathbf{B}$ denote the bilattice underlying the first-order annotated language $\mathcal{L}$; it will provide the set of truth values for $\mathcal{L}$.

Following the conventional definition of an interpretation (see Chapter 1 or [22], for example), we fix a domain D, where individual constants and individual variables and function symbols receive interpretation. A *variable assignment* V is an assignment, to each variable in $\mathcal{L}$, of an element in the domain D. An interpretation for constants, variables, and function symbols in D will be called a pre-interpretation J, we will denote it by $|\ |$, see Chapter 1, Definition 1.4.1.

Furthermore, given a variable assignment V, by an abuse of notation which will not cause confusion, we denote by $|t| = |t|_{J,V}$ the term assignment in D of the term t in $\mathcal{L}$ with respect to J and V, see Chapter 1.

Definition 3.4.1 (Interpretation $\mathcal{I}$)**.** *An* interpretation $\mathcal{I}$ *for* $\mathcal{L}$ *consists of a pre-interpretation* J *for* $\mathcal{L}$ *together with the assignment of a mapping* $|R| = |R|_{\mathcal{I},V} : D^n \longrightarrow \mathbf{B}$ *for each* n*-ary predicate symbol* R *in* $\mathcal{L}$*.*

Throughout this chapter, we will use the notation $\langle \alpha, \beta \rangle$ to denote elements of $\mathbf{B}$, and

the notation (α, β) to denote annotation constants from $\mathcal{L}^{\mathbf{B}}$. Mathematically, both $\langle \alpha, \beta \rangle$ and (α, β) are elements of $\mathbf{B}$.

One further piece of notation we need is as follows: for each element $\langle \alpha, \beta \rangle$ of $\mathbf{B}$, we denote by $\chi_{\langle \alpha, \beta \rangle} : \mathbf{B} \longrightarrow \mathbf{B}$ the mapping defined by $\chi_{\langle \alpha, \beta \rangle}(\langle \alpha', \beta' \rangle) = \langle 1, 0 \rangle$ if $\langle \alpha, \beta \rangle \leq_k \langle \alpha', \beta' \rangle$ and $\chi_{\langle \alpha, \beta \rangle}(\langle \alpha', \beta' \rangle) = \langle 0, 1 \rangle$ otherwise.

We will use the two functions $\mathcal{I}$ and χ to define an interpretation I for annotated atoms. Given an annotated atom $A : (\alpha', \beta')$ with constant annotation (α', β'), an interpretation $\mathcal{I}$ for the first-order formula A, and a value $\langle \alpha, \beta \rangle$ from $\mathbf{B}$ assigned to A using $\mathcal{I}$, we use χ as follows: if the value $\langle \alpha', \beta' \rangle \leq_k \langle \alpha, \beta \rangle$, then $I(A : (\alpha', \beta')) = \langle 1, 0 \rangle$, and $I(A : (\alpha', \beta')) = \langle 0, 1 \rangle$ otherwise. If the annotated term τ attached to an annotated atom $A : \tau$ contains free annotation variables $\overline{\mu, \nu}$, we use the existential quantifier Σ when applying χ as follows: $\chi_\tau(\langle \alpha, \beta \rangle) = \langle 1, 0 \rangle$ if $\Sigma(\overline{\mu, \nu})(\langle \alpha, \beta \rangle \leq_k \tau(\overline{\mu, \nu}))$. We will assume this quantification when giving the next definition.

Example 3.4.1. *Let $R(a_1, \ldots, a_k) : (0, \frac{1}{2})$ and $R(a_1, \ldots, a_k) : (\frac{1}{2}, 1)$ be annotated formulae of the language $\mathcal{L}_{B_9}$. And let the Bilattice B_9 from Example 3.2.2 be the underlying bilattice of $\mathcal{L}_{B_9}$.*

Let $|R(a_1, \ldots, a_k)| = \langle 1, \frac{1}{2} \rangle$. Element $\langle 0, \frac{1}{2} \rangle \leq_k \langle 1, \frac{1}{2} \rangle$, but element $\langle \frac{1}{2}, 1 \rangle$ is not k-comparable with $\langle 1, \frac{1}{2} \rangle$. That is why $|R(a_1, \ldots, a_k) : (0, \frac{1}{2})| = \langle 1, 0 \rangle$ and $|R(a_1, \ldots a_k) : (\frac{1}{2}, 1)| = \langle 0, 1 \rangle$.

When giving interpretation to the language $\mathcal{L}$, we will distinguish the connectives $\otimes$, $\oplus$, $\wedge$ and $\vee$ of $\mathcal{L}$, from the corresponding operations $\widetilde{\otimes}$, $\widetilde{\oplus}$, $\widetilde{\wedge}$ and $\widetilde{\vee}$ of the underlying bilattice $\mathbf{B}$ by using the symbol $\widetilde{\ }$.

Definition 3.4.2 (Interpretation I built on $\mathcal{I}$)**.** *Let $\mathcal{I}$ be an interpretation with domain D for a first order annotated language $\mathcal{L}$ and let V be a variable assignment. Then an*

annotated formula F in $\mathcal{L}$ can be given a truth value $|F| = |F|_{I,V}$ *in* $\mathbf{B}$ *as follows.*

- *If F is an annotated atom $R(t_1, \ldots, t_n) : (\tau)$, then the value of $|F|_{I,V}$ is given by $|F|_{I,V} = \chi_\tau(|R|_{\mathcal{I}}(|t_1|, \ldots, |t_n|))$.*

- *If an annotated formula has the form $(\neg F_1)$, $(\sim F_1)$, $(F_1 \vee F_2)$, $(F_1 \wedge F_2)$, $(F_1 \otimes F_2)$, $(F_1 \oplus F_2)$, where F_1 and F_2 are annotated atoms then the truth value of the formula is given as follows:*

- $|\neg F : (\overline{\mu, \nu})|_{I,V} = |F : (\overline{\nu, \mu})|_{I,V}$[1]*;*

- $| \sim F : (\overline{\mu, \nu})|_{I,V} =< 0, 1 >$*, if* $|F : (\overline{\mu, \nu})|_{I,V} =< 1, 0 >$*, and* $| \sim F : (\overline{\mu, \nu})|_{I,V} =< 1, 0 >$*, if* $|F : (\overline{\mu, \nu})|_{I,V} =< 0, 1 >$.

- $|F_1 \otimes F_2|_{I,V} = |F_1|_{I,V} \widetilde{\otimes} |F_2|_{I,V}$*;*

- $|F_1 \oplus F_2|_{I,V} = |F_1|_{I,V} \widetilde{\oplus} |F_2|_{I,V}$

- $|F_1 \wedge F_2|_{I,V} = |F_1|_{I,V} \widetilde{\wedge} |F_2|_{I,V}$

- $|F_1 \vee F_2|_{I,V} = |F_1|_{I,V} \widetilde{\vee} |F_2|_{I,V}$

- *If an annotated formula has the form $(F_1 \vee F_2) : (\tau)$, $(F_1 \wedge F_2) : (\tau)$, $(F_1 \otimes F_2) : (\tau)$, $(F_1 \oplus F_2) : (\tau)$, where F_1 and F_2 are atoms and τ is an annotation term, then the truth value of each formula of this kind is computed in two steps as follows:*

- $|(F_1 \otimes F_2) : (\tau)|_{I,V} = \chi_\tau(|F_1|_{\mathcal{I},V} \widetilde{\otimes} |F_2|_{\mathcal{I},V})$

[1] *The formula $|\neg F : (\overline{\mu, \nu})|_{I,V} = |F : (\overline{\nu, \mu})|_{I,V}$ is a short notation for an inductive definition of the interpretation for negation, as follows:*
If the annotation term is a constant (α, β), then $|\neg F : (\alpha, \beta)|_{I,V} = |F : (\beta, \alpha)|_{I,V}$;
If the annotation term is a variable (μ, ν), then $|\neg F : (\mu, \nu)|_{I,V} = |F : (\nu, \mu)|_{I,V}$;
If the annotation term is a function $\vartheta(\overline{\mu, \nu})$, then, for each ground instance $[\vartheta(\overline{\mu, \nu})]^g$ of $\vartheta(\overline{\mu, \nu})$, if $[\vartheta(\overline{\mu, \nu})]^g = \langle \alpha, \beta \rangle$, then $|\neg F : ([\vartheta(\overline{\mu, \nu})]^g)|_{I,V} = |F : (\beta, \alpha)|_{I,V}$.

- $|(F_1 \oplus F_2) : (\tau)|_{I,V} = \chi_\tau(|F_1|_{\mathcal{I},V} \widetilde{\oplus} |F_2|_{\mathcal{I},V})$
- $|(F_1 \wedge F_2) : (\tau)|_{I,V} = \chi_\tau(|F_1|_{\mathcal{I},V} \widetilde{\wedge} |F_2|_{\mathcal{I},V})$
- $|(F_1 \vee F_2) : (\tau)|_{I,V} = \chi_\tau(|F_1|_{\mathcal{I},V} \widetilde{\vee} |F_2|_{\mathcal{I},V})$
- *If a formula has the form* $\Sigma x R_n(x) : (\tau)$, $\Pi x R_n(x) : (\tau)$, $\forall x R_n(x) : (\tau)$ *and* $\exists x R_n(x) : (\tau)$,

$$|\Sigma x R_n(x) : (\tau)|_{I,V} = \chi_\tau \left(\sum_{d \in D} |R_n(d)|_{\mathcal{I},V} \right)$$

$$|\Pi x R_n(x) : (\tau)|_{I,V} = \chi_\tau \left(\prod_{d \in D} |R_n(d)|_{\mathcal{I},V} \right)$$

$$|\forall x R_n(x) : (\tau)|_{I,V} = \chi_\tau \left(\bigwedge_{d \in D} |R_n(d)|_{\mathcal{I},V} \right)$$

$$|\exists x R_n(x) : (\tau)|_{I,V} = \chi_\tau \left(\bigvee_{d \in D} |R_n(d)|_{\mathcal{I},V} \right)$$

where $\sum$, $\prod$, $\bigvee$ *and* $\bigwedge$ *are the infinite joins and meets with respect to the k- and t-orderings in the bilattice* $\mathbf{B}$, $|R_n(d)|_{\mathcal{I},V}$ *receives interpretation with respect to* $\mathcal{I}$ *and* $V(x/d)$, *where* $V(x/d)$ *is* V *except that* x *is assigned* d.

Note that the connectives $\oplus$, $\otimes$, $\vee$, $\wedge$ and the quantifiers Σ, Π, $\exists$, $\forall$ are interpreted by finite and infinite operations defined on bilattices as in Section 3.2.

In general, the analogs of the classical truth values true and false are represented by $\langle 1, 0 \rangle$ and $\langle 0, 1 \rangle$ - the greatest and least elements of the bilattice with respect to $\leq_t$.

The following properties of I are very important for the development of the declarative and operational semantics for BAPs.

Proposition 3.4.1 (Properties of I)**.** *Let* $F, F_1, \ldots F_k$ *be first-order formulae, and fix a first-order interpretation* $\mathcal{I}$ *for them. Then any interpretation* I *built upon* $\mathcal{I}$ *as in Definition 3.4.2 has the following properties:*

1. *If* $I(F : (\alpha, \beta)) = \langle 1, 0\rangle$, *then* $I(F : (\alpha', \beta')) = \langle 1, 0\rangle$ *for all* $\langle \alpha', \beta'\rangle \leq_k \langle \alpha, \beta\rangle$.

2. *If* $I(F : (\alpha, \beta)) = \langle 0, 1\rangle$, *then* $I(F : (\alpha', \beta')) = \langle 0, 1\rangle$, *for all* (α', β') *such that* $\langle \alpha, \beta\rangle \leq_k \langle \alpha', \beta'\rangle$.

3. $I(F_1 : (\tau_1) \otimes \ldots \otimes F_k : (\tau_k)) = \langle 1, 0\rangle \iff I(F_1 : (\tau_1) \oplus \ldots \oplus F_k : (\tau_k)) = \langle 1, 0\rangle \iff I(F_1 : (\tau_1) \wedge \ldots \wedge F_k : (\tau_k)) = \langle 1, 0\rangle \iff I(F_i : (\tau_i)) = \langle 1, 0\rangle$, *for each* $i \in \{1, \ldots, k\}$.

4. $I(F_1 : (\tau_1) \otimes \ldots \otimes F_k : (\tau_k)) = \langle 0, 1\rangle \iff I(F_1 : (\tau_1) \oplus \ldots \oplus F_k : (\tau_k)) = \langle 0, 1\rangle \iff I(F_1 : (\tau_1) \vee \ldots \vee F_k : (\tau_k)) = \langle 0, 1\rangle \iff I(F_i : (\tau_i)) = \langle 0, 1\rangle$, *for each* $i \in \{1, \ldots, k\}$.

5. *If* $I(F_1 : (\tau_1) \odot \ldots \odot F_k : (\tau_k)) = \langle 1, 0\rangle$, *then* $I((F_1 \odot \ldots \odot F_k) : ((\tau_1) \odot \ldots \odot (\tau_k))) = \langle 1, 0\rangle$, *where* $\odot$ *is any one of the connectives* $\otimes, \oplus, \wedge$.

6. *If* $I(F_1 : (\tau_1) \odot \ldots \odot F_k : (\tau_k)) = \langle 0, 1\rangle$, *then* $I((F_1 \odot \ldots \odot F_k) : ((\tau_1) \odot \ldots \odot (\tau_k))) = \langle 0, 1\rangle$, *where* $\odot$ *is any one of the connectives* $\otimes, \oplus, \vee$.

7. *If* $I(F_1 : (\tau)) = \langle 1, 0\rangle$, *then* $I((F_1 \oplus F_2) : (\tau)) = \langle 1, 0\rangle$, *for any formula* F_2.

8. *If* $I(F_1 : (\tau)) = \langle 0, 1\rangle$, *then* $I((F_1 \otimes F_2) : (\tau)) = \langle 0, 1\rangle$, *for any formula* F_2.

9. $I(F : (\alpha, \beta)) = \langle 1, 0\rangle \iff I(\neg F : (\beta, \alpha)) = \langle 1, 0\rangle$.

10. $I(F : (\alpha, \beta)) = \langle 0, 1\rangle \iff I(\neg F : (\beta, \alpha)) = \langle 0, 1\rangle$.

11. *For every formula* F, $I(F : (0, 0)) = \langle 1, 0\rangle$.

Proof. Because in Proposition 3.4.1, all the annotated formulae are evaluated as true ($\langle 1, 0\rangle$) or false ($\langle 0, 1\rangle$), we assume throughout the proof that all the individual and annotation terms are either ground or contain only bound variables. Therefore, we freely

operate with bilattice values and annotations of formulae, assuming them fully determined by ground and/or bound annotation terms.

We will give proofs of Items 1, 3, 5, the dual assertions are proven similarly. Items 7 - 11 are straightforward.

Item 1:

According to Definition 3.4.2, $|F : (\alpha, \beta)| = \langle 1, 0\rangle$ if and only if $\langle \alpha, \beta\rangle \leq_k |F|$. But then, for all $\langle \alpha', \beta'\rangle \leq_k \langle \alpha, \beta\rangle$, we also have $\langle \alpha', \beta'\rangle \leq_k |F|$. Then, using Definition 3.4.2, we conclude that $|F : (\alpha', \beta')| = \langle 1, 0\rangle$.

Item 3:

The proof that if $I(F_i : (\tau_i)) = \langle 1, 0\rangle$, for each $i \in \{1, \ldots, k\}$, then $I(F_1 : (\tau_1) \otimes \ldots \otimes F_k : (\tau_k)) = \langle 1, 0\rangle$; $I(F_1 : (\tau_1) \oplus \ldots \oplus F_k : (\tau_k)) = \langle 1, 0\rangle$ and $I(F_1 : (\tau_1) \wedge \ldots \wedge F_k : (\tau_k)) = \langle 1, 0\rangle$ is trivial, and follows from the fact that $\otimes, \oplus, \wedge$ are idempotent.

So, we concentrate on the $\Longrightarrow$ part of the item 3. Note that according to the definition of interpretation, each atomic annotated formula $F_i : (\tau_i)$ receives its value from the set $\{\langle 1, 0\rangle, \langle 0, 1\rangle\}$. Using properties of $\otimes$ and the properties of the four-valued bilattice we use at meta-level, we conclude that $|F_1 : (\tau_1) \otimes \ldots \otimes F_k : (\tau_k)| = \langle 1, 0\rangle$ only if each $|F_i : (\tau_i)| = \langle 1, 0\rangle$. Otherwise, $F_1 : (\tau_1) \otimes \ldots \otimes F_k : (\tau_k)$ will receive the value $\langle 0, 0\rangle$ or $\langle 0, 1\rangle$.

The same observations are valid for $\oplus$ and $\wedge$.

The formula $|F_1 : (\tau_1) \oplus \ldots \oplus F_k : (\tau_k)|$ will receive the value $\langle 1, 1\rangle$ or $\langle 0, 1\rangle$, if at least one of $F_1 : (\tau_1), \ldots, F_k : (\tau_k)$ receives the value $\langle 0, 1\rangle$.

The formula $F_1 : (\tau_1) \wedge \ldots \wedge F_k : (\tau_k)$ receives the value $\langle 0, 1\rangle$ if at least one of its atomic formulae receives $\langle 0, 1\rangle$. Consequently, all the complex formulae receive the value $\langle 1, 0\rangle$ if and only if each annotated atom, occurring in these complex formulae, receives

the value $\langle 1, 0\rangle$.

Item 5:

Proof is given by induction on k. Note that we use the definition of $|\ |_{I,V}$ that assigns to annotated expressions only the values $\langle 0, 1\rangle$ and $\langle 1, 0\rangle$. We give here the proof for the cases when $\odot$ is one of the following connectives: $\otimes, \oplus$, and omit the similar proof for $\wedge$.
Basic step.(k=1) The basic step is similar for all the connectives. If $|F_1 : (\mu_1, \nu_1)| = \langle 1, 0\rangle$, then $|F_1 : (\mu_1, \nu_1)| = \langle 1, 0\rangle$.

We continue the proof for $\otimes$.

Inductive step. Assume for $k-1$ the proposition holds. Let $|F_1 : (\tau_1) \otimes \ldots \otimes F_{k-1} : (\tau_{k-1}) \otimes F_k : (\tau_k)|_{I,V} = \langle 1, 0\rangle$. Using Item 3, we can conclude that $|F_1 : (\tau_1) \otimes \ldots \otimes F_{k-1} : (\tau_{k-1})|_{I,V} = \langle 1, 0\rangle$ and $F_k : (\tau_k)|_{I,V} = \langle 1, 0\rangle$. This means that $\langle \tau_k\rangle \leq_k |F_k|_{\mathcal{I},V}$. Now, using the induction hypothesis, we obtain $|(F_1 \otimes \ldots \otimes F_{k-1}) : (\tau_1 \otimes \ldots \otimes \tau_{k-1})|_{I,V} = \langle 1, 0\rangle$. And thus, $\langle \tau_1 \widetilde{\otimes} \ldots \widetilde{\otimes} \tau_{k-1}\rangle \leq_k |(F_1 \otimes \ldots \otimes F_{k-1})|_{I,V}$. Consider $|((F_1 \otimes \ldots \otimes F_{k-1}) \otimes F_k) : ((\tau_1 \otimes \ldots \otimes \tau_{k-1}) \otimes (\tau_k))|_{I,V}$. By Definition 3.4.2, $|(F_1 \otimes \ldots \otimes F_{k-1}) \otimes F_k|_{\mathcal{I},V} = |(F_1 \otimes \ldots \otimes F_{k-1})|_{\mathcal{I},V} \widetilde{\otimes} |F_k|_{\mathcal{I},V}$. Since $\langle \tau_1 \widetilde{\otimes} \ldots \widetilde{\otimes} \tau_{k-1}\rangle \leq_k |(F_1 \otimes \ldots \otimes F_{k-1})|_{\mathcal{I},V}$ and $\langle \tau_k\rangle \leq_k |F_k|_{\mathcal{I},V}$, using the monotonicity property of the operation $\widetilde{\otimes}$ (see Proposition 3.2.1), we conclude that $(\langle \tau_1 \widetilde{\otimes} \ldots \widetilde{\otimes} \tau_{k-1}\rangle \widetilde{\otimes} \langle \tau_k\rangle \leq_k |(F_1 \otimes \ldots \otimes F_{k-1})|_{\mathcal{I},V} \widetilde{\otimes} |F_k|_{\mathcal{I},V}$, which means that $|((F_1 \otimes \ldots \otimes F_{k-1}) \otimes F_k) : ((\tau_1 \otimes \ldots \otimes \tau_{k-1}) \otimes (\tau_k))|_{I,V} = \langle 1, 0\rangle$.

The following is the proof of the inductive step for the case when $\circ = \oplus$.

Inductive step. Assume for $k-1$ the proposition holds. Let $|F_1 : (\tau_1) \oplus \ldots \oplus F_{k-1} : (\tau_{k-1}) \oplus F_k : (\tau_k)|_{I,V} = \langle 1, 0\rangle$. Using Item 3, we conclude that $|F_1 : (\tau_1) \oplus \ldots \oplus F_{k-1} : (\tau_{k-1})|_{I,V} = \langle 1, 0\rangle$ and $F_k : (\tau_k)|_{I,V} = \langle 1, 0\rangle$. This means that $\langle \tau_k\rangle \leq_k |F_k|_{\mathcal{I},V}$. Now, using the induction hypothesis, we obtain $|(F_1 \oplus \ldots \oplus F_{k-1}) : (\tau_1 \oplus \ldots \oplus \tau_{k-1})|_{I,V} = \langle 1, 0\rangle$. And thus, $(\tau_1 \widetilde{\oplus} \ldots \widetilde{\oplus} \tau_{k-1}) \leq_k |(F_1 \oplus \ldots \oplus F_{k-1})|_{I,V}$. Consider $|((F_1 \oplus \ldots \oplus F_{k-1}) \oplus F_k) :$

$((\tau_1 \oplus \ldots \oplus \tau_{k-1}) \oplus (\mu_k, \nu_k))|_{I,V}$. By Definition 3.4.2, $|(F_1 \oplus \ldots \oplus F_{k-1}) \oplus F_k|_{\mathcal{I},V} = |(F_1 \oplus \ldots \oplus F_{k-1})|_{\mathcal{I},V} \widetilde{\oplus} |F_k|_{\mathcal{I},V}$. Since $\langle \tau_1 \widetilde{\oplus} \ldots \widetilde{\oplus} \tau_{k-1} \rangle \leq_k |(F_1 \oplus \ldots \oplus F_{k-1})|_{\mathcal{I},V}$ and $\langle \mu_k, \nu_k \rangle \leq_k |F_k|_{\mathcal{I},V}$, using the monotonicity property of the operation $\widetilde{\oplus}$ (see Proposition 3.2.1), we conclude that $(\langle \tau_1 \widetilde{\oplus} \ldots \widetilde{\oplus} \tau_{k-1} \rangle \widetilde{\oplus} \langle \mu_k, \nu_k \rangle \leq_k |(F_1 \oplus \ldots \oplus F_{k-1})|_{\mathcal{I},V} \widetilde{\oplus} |F_k|_{\mathcal{I},V}$, which means that $|((F_1 \oplus \ldots \oplus F_{k-1}) \oplus F_k) : ((\mu_1, \nu_1 \widetilde{\oplus} \ldots \widetilde{\oplus} \mu_{k-1}, \nu_{k-1}) \widetilde{\oplus} (\mu_k, \nu_k))|_{I,V} = \langle 1, 0 \rangle$.

The same proof can be given for $\wedge$, using the properties of $\widetilde{\wedge}$ defined on the bilattice $\mathbf{B}$ and Item 3. We use here the fact that all four operations defined on the bilattice are monotonic with respect to both orderings (see Definition 3.2.11 and Proposition 3.2.2), and, in particular, if $a \leq_k b$, $c \leq_k d$, then $a \widetilde{\wedge} c \leq_k b \widetilde{\wedge} d$. □

The converse of the Item 5 is not true in the general case. Consider the following counterexample.

Example 3.4.2. *Let the bilattice* $\mathbf{B}_9$ *from Example 3.2.2 be the underlying bilattice of the language* $\mathcal{L}$. *Consider the following formulae* $F_1 : (\frac{1}{2}, \frac{1}{2}) \otimes F_2 : (\frac{1}{2}, 1)$ *and* $(F_1 \otimes F_2) : ((\frac{1}{2}, \frac{1}{2}) \otimes (\frac{1}{2}, 1))$. *Consider the following interpretation* $\mathcal{I}$: $|F_1|_{\mathcal{I},V} = \langle \frac{1}{2}, \frac{1}{2} \rangle$ *and* $|F_2|_{\mathcal{I},V} = \langle 1, \frac{1}{2} \rangle$. *The interpretation* $|(F_1 \otimes F_2) : ((\frac{1}{2}, \frac{1}{2}) \otimes (\frac{1}{2}, 1))|_{I,V} = |(F_1 \otimes F_2) : (\frac{1}{2}, \frac{1}{2})|_{I,V} = \langle 1, 0 \rangle$. *But* $|F_1 : (\frac{1}{2}, \frac{1}{2}) \widetilde{\otimes} F_2 : (\frac{1}{2}, 1)|_{I,V} = \langle 0, 0 \rangle$, *because* $|F_1 : (\frac{1}{2}, \frac{1}{2})|_{I,V} = \langle 1, 0 \rangle$ *and* $|F_2 : (\frac{1}{2}, 1)|_{I,V} = \langle 0, 1 \rangle$, *and, consequently,* $|F_1 : (\frac{1}{2}, \frac{1}{2})|_{I,V} \widetilde{\otimes} |F_2 : (\frac{1}{2}, 1)|_{I,V} = \langle 0, 0 \rangle$.

Note also that the proof of Item 5 cannot be extended to the case when $\circ = \vee$. Consider the following example.

Example 3.4.3. *Let the bilattice* $\mathbf{B}_{16}$ *from Example 3.2.3 be the underlying bilattice of* $\mathcal{L}$. *Let* $|F_1|_{\mathcal{I},V} = \langle 0, \frac{1}{3} \rangle$, $|F_2|_{\mathcal{I},V} = \langle \frac{1}{3}, \frac{1}{3} \rangle$. *The annotated expression* $|F_1 : (\frac{2}{3}, \frac{2}{3}) \vee F_2 : (\frac{1}{3}, \frac{1}{3})|_{I,V} = \langle 1, 0 \rangle$, *because* $|F_2 : (\frac{1}{3}, \frac{1}{3})|_{I,V} = \langle 1, 0 \rangle$. *But* $|(F_1 \vee F_2) : ((\frac{2}{3}, \frac{2}{3}) \widetilde{\vee} (\frac{1}{3}, \frac{1}{3}))|_{I,V} = |(F_1 \vee F_2) : (\frac{2}{3}, \frac{1}{3})|_{I,V} = \langle 0, 1 \rangle$, *because* $|F_1 \vee F_2|_{\mathcal{I},V} = \langle \frac{1}{3}, \frac{1}{3} \rangle$ *and* $\langle \frac{2}{3}, \frac{1}{3} \rangle > \langle \frac{1}{3}, \frac{1}{3} \rangle$.

Proposition 3.4.1 plays an important role when discussing the set of all logical consequences of a set of annotated formulae. We will use these properties when discussing declarative and operational semantics for bilattice-based annotated logic programs.

Because the syntax of Horn clauses restricts us only to formulae of a certain kind, we will use only Items 1, 3, 5, 11, 9 in subsequent sections. But in [123] we used all the properties stated in Proposition 3.4.1 to give a sequent calculus for the full fragment of the bilattice-based annotated language $\mathcal{L}$.

Note that if restricted to one lattice, for example, to the classical lattice {0,1}, the connectives $\otimes$ and $\wedge$ ($\oplus$ and $\vee$) are the same, $\neg$ corresponds to classical negation, and the quantifiers $\exists$ and $\forall$ are the same as in classical logic.

The definitions of satisfiable, unsatisfiable, valid and non-valid formulae and models are standard, if classical values true and false in these definitions are replaced by the greatest and the least elements of the bilattice with respect to the t-ordering, that is, by $\langle 1, 0\rangle$ and $\langle 0, 1\rangle$. The following definitions and proposition are straightforward generalisations of their classical counterparts.

Definition 3.4.3 (Cf. Definition 1.4.9)**.** *Let I be an interpretation for $\mathcal{L}$ and let F be a closed annotated formula of $\mathcal{L}$. Then I is a* model *for F if $|F|_{I,V} = \langle 1, 0\rangle$. We say that I is a model for the set of annotated formulae S if it is a model for each annotated formula of S.*

Definition 3.4.4 (Cf. Definition 1.4.10)**.** *Let S be a set of closed annotated formulae and F be a closed annotated formula of $\mathcal{L}$. We say that F* is a logical consequence of *S if, for every interpretation I of $\mathcal{L}$, I is a model for S implies I is a model for F.*

Proposition 3.4.2 (Cf. Proposition 1.4.1)**.** *Let S and F be defined as in the previous definitions. Then F is a logical consequence of S if and only if $S \cup \{\sim F\}$ is unsatisfiable.*

Proof. The proof is a straightforward adaptation of the analogous proof given for the two-valued first-order case in [138]. □

Now we are ready to introduce Horn-clause logic programs based on the bilattice-based languages we have defined.

3.5 Bilattice-Based Annotated Logic Programs

In this section we introduce Bilattice-Based Annotated Logic Programs (BAPs). These programs extend the Horn clause logic programming carefully examined in Chapter 1. Some notions that we introduce in this section bear strong resemblance to the conventional notions. To make this relation between BAPs and Horn clause logic programming of Chapter 1 clear, we will refer to the corresponding Definitions of Chapter 1 whenever it makes sense.

Let $\mathbf{B} = L_1 \odot L_2$ and $\mathcal{L}$ be the language defined in Section 3.3. We require $\mathbf{B}$ to be distributive, and have only *finite* joins.

Definition 3.5.1 (Cf. Definition 1.2.6)**.** An annotated literal *is an annotated atom or the (epistemic) negation of an annotated atom. A* positive annotated literal *is an annotated atom. A* negative annotated literal *is the (epistemic) negation of an annotated atom.*

We choose the k-ordering to take the main connectives and quantifiers from. The next Definition introduces clauses.

Definition 3.5.2 (Cf. Definitions 1.3.1, 1.3.2)**.** *An* annotated bilattice-based clause *is an annotated formula of the form*

$$\Pi\overline{\mu,\nu},\Pi\overline{x}(A_1:(\tau_1)\oplus\ldots\oplus A_k:(\tau_k))\oplus\sim(\Sigma\overline{\mu',\nu'},\Sigma\overline{x'}(L_1:(\tau_1')\otimes\ldots\otimes L_n:(\tau_n')))$$

where each $A_i : (\tau_i)$ *is an annotated atom and each* $L_i : (\tau'_i)$ *is an annotated literal, and* $\overline{\mu, \nu}$, $\overline{x}$ *are all the annotation and individual variables appearing in* $A_1 : (\tau_1) \oplus \ldots \oplus A_k : (\tau_k)$, $\overline{\mu', \nu'}$ *and* $\overline{x'}$ *are all the annotation and individual variables appearing in* $L_1 : (\tau'_1) \otimes \ldots \otimes L_n : (\tau'_n)$.

Following [138] we adopt the following notation for the clauses. We denote the clauses

$$\Pi\overline{\mu, \nu}, \Pi\overline{x}(A_1 : (\tau_1) \oplus \ldots \oplus A_k : (\tau_k)) \oplus \sim (\Sigma\overline{\mu', \nu'}, \Sigma\overline{x'}(L_1 : (\tau'_1) \otimes \ldots \otimes L_n : (\tau'_n)))$$

defined above by

$$A_1 : (\tau_1), \ldots, A_k : (\tau_k) \leftarrow L_1 : (\tau'_1), \ldots, L_n : (\tau'_n).$$

Note that unlike the conventional clauses of Definitions 1.3.1, 1.3.2, these annotated clauses cannot be represented as simple disjunctions of literals.

Definition 3.5.3 (Cf. Definitions 1.3.1, 1.3.2.)**.** *A* bilattice-based annotated normal logic program (BAP) *P consists of a finite set of* program clauses *of the form*

$$A : (\tau) \leftarrow L_1 : (\tau_1), \ldots, L_n : (\tau_n)$$

where $A : (\tau)$ *is an annotated atomic formula called the* head *of the clause and* $L_1 : (\tau_1), \ldots, L_n : (\tau_n)$ *are annotated literals, called the* body *of the clause.*

Note that in this setting, where the epistemic negation $\neg$ is monotonic with respect to the k-ordering, the distinction between definite (positive) and normal (allowing negation) programs is completely diminished. Indeed, according to the Definition 3.4.2, each annotated negative literal of the form $\neg L : (\alpha, \beta)$ can be equivalently substituted by $L : (\beta, \alpha)$. And thus, each normal annotated program transforms easily into a definite annotated

program. We will use this fact throughout without further mention, and develop all the model theoretic and proof theoretic techniques assuming that we work with definite BAPs.

The definitions of a unit clause and of a program goal are straightforward generalisations of [138], see also Definition 1.3.3.

The language described in Subsection 3.3 contains six possible connectives and four quantifiers. Consequently, one in principle can think about all possible combinations of meets, joins, and quantifiers with respect to both the t- and k- orderings when defining clauses of BAPs. However, in Definition 3.5.2 of a BAP, we assumed the use of only two connectives: $\otimes$ and $\neg$. From the enumerated range of quantifiers we pick only Σ. In [123] we used all the connectives and quantifiers listed in Definition 3.3.1, and proposed a sequent calculus for the full fragment of bilattice-based annotated logic. However, we will not develop this theme any further in this thesis.

Consider some examples of BAPs, based on the language $\mathcal{L}^B$.

Example 3.5.1. *Consider the (propositional) logic program based on the four-valued bilattice from Example 3.1.*

$$
\begin{aligned}
B : (0,1) \quad &\leftarrow \\
B : (1,0) \quad &\leftarrow \\
A : (1,1) \quad &\leftarrow \quad B : (1,1) \\
C : (1,0) \quad &\leftarrow \quad A : (1,0)
\end{aligned}
$$

We can regard this program as formalising Example 3.1.1. Let B stand for "NP $\neq$ coNP", A stand for "P $\neq$ NP", C stand for "NP $\neq$ NC"; annotations $(0,0)$, $(0,1)$, $(1,0)$, $(1,1)$ express, respectively, "no proof/refutation is given", "proven", "proven the opposite"

and "contradictory, or proven both the statement and the opposite".

Example 3.5.2. *Let* $\mathbf{B}_{25}$ *from Example 3.2.4 be the chosen bilattice and let* $\mathcal{L}^{B_{25}}$ *be the language with underlying bilattice* $\mathbf{B_{25}}$.

We can have the following program:

$$
\begin{aligned}
R_1(a_1) : (\mu, \nu) &\leftarrow \\
R_2(f(x)) : (0, \frac{1}{4}) &\leftarrow \\
R_3(a_2) : (\frac{3}{4}, 1) &\leftarrow \\
R_4(a_1) : (\frac{1}{2}, \frac{1}{2}) &\leftarrow \\
R_3(x) : (\mu, \nu) &\leftarrow R_2(x) : (\mu, \nu) \\
R_4(x) : (1, 0) &\leftarrow R_3(f(x)) : (\mu, \nu) \\
R_5(f(x)) : ((\mu_1, \nu_1) \widetilde{\wedge} (\mu_2, \nu_2)) &\leftarrow \neg R_1(x) : (\frac{1}{2}, \frac{1}{4}), R_4(f(x)) : (1, 0)
\end{aligned}
$$

Note that $\widetilde{\wedge}$ *is the meet with respect to the* t*-ordering defined on the underlying bilattice (see §3.2).* $R_1(a_1) : (\mu, \nu) \leftarrow$, $R_2(f(x)) : (0, \frac{1}{4}) \leftarrow$, $R_3(a_2) : (\frac{3}{4}, 1) \leftarrow$, $R_4(a_1) : (\frac{1}{2}, \frac{1}{2}) \leftarrow$ *are unit clauses, or facts. Some of them contain bilattice constants, e.g., the fact* $R_3(a_2) : (\frac{3}{4}, 1)$ *is known to be true with measures of confidence* $\frac{3}{4}$ *for the fact and* 1 *against it. The clause* $R_1(a_1) : (\mu, \nu)$ *contains bilattice variables. Annotations in the heads* $R_3(x) : (\mu, \nu)$ *and* $R_5(f(x)) : ((\mu_1, \nu_1) \widetilde{\wedge} (\nu_2, \mu_2))$ *receive their values in accordance with the values of the literals from their bodies. We treat* $\neg R_1(x) : (\frac{1}{2}, \frac{1}{4})$ *as* $R_1(x) : (\frac{1}{4}, \frac{1}{2})$.

Example 3.5.3. *Suppose we want to implement a program* P *which is able to make decisions about delaying and cancelling airport flights without human supervision. Let this program work with databases which are obtained through accumulating information from weather forecasts taken from different sources. These sources may give inconsistent*

or incomplete information which is collected in the database of P. The first and the second elements of each annotation denote evidence for, respectively against, a given fact. Let $\mathbf{B}_{25}$ from Example 3.2.4 be the chosen bilattice. Let individual variables x, y receive, respectively, values from the set $\{Monday, Tuesday\}$ and the set with the numbers of flights $\{1, 2\}$. A suitable program for processing the database may contain the following fragment:

$$\begin{aligned}
Storm(Monday) : (\frac{3}{4}, \frac{1}{2}) &\leftarrow \\
Storm(x) : (\frac{1}{2}, \frac{3}{4}) &\leftarrow \\
Delay(y, x) : ((\frac{3}{4}, \frac{1}{2}) \vee (\frac{1}{2}, \frac{1}{2})) &\leftarrow Storm(x) : (\frac{3}{4}, \frac{3}{4}), Storm(Tuesday) : (\frac{1}{2}, \frac{1}{2}) \\
Cancel(y, x) : (1, 0) &\leftarrow Delay(y, x) : (\frac{3}{4}, \frac{1}{2})
\end{aligned}$$

Note that this program is able to make quantitative (fuzzy) conclusions, as in the third clause, as well as qualitative (two-valued) conclusions, as in the last clause. It can work with conflicting sources of information, see, for example, the two unit clauses.

We need to mention here that each particular BAP contains only a finite set of annotation constants. That is why the programs which do not contain annotation functions, and the programs containing only those functions which do not generate new annotation constants through the process of computing, can always be interpreted by finite bilattices. Examples 3.5.2, 3.5.3, 3.5.1 present programs which can be interpreted by finite bilattices.

Example 3.5.4. *Consider the following infinitely-interpreted logic program:*

$$\begin{aligned}
R_1(a_1) : (1, 0.5) &\leftarrow \\
R_2(f(x)) : (\frac{\mu}{2}, \frac{\nu}{3}) &\leftarrow R_1(x) : (\mu, \nu) \\
R_1(f(x)) : (\frac{\mu}{3}, \frac{\nu}{3}) &\leftarrow R_2(x) : (\mu, \nu)
\end{aligned}$$

Annotation terms of this program receive their values from the countable bilattice whose underlying set of elements contains 0, 1, 0.5 *and all the numbers which can be generated from* 0, 1, 0.5 *by iterating the functions* $\frac{\mu}{2}$, $\frac{\mu}{3}$, $\frac{\nu}{3}$.

Note that since this program does not contain negation, it can be interpreted by asymmetric bilattices. We discussed the possibility of using these bilattices in Section 3.2.

3.6 Unification

We refer to Section 1.6 and [138] for a description of the unification process (originally defined in [86] and later refined in [178]) for classical logic programming.

The unification algorithm for BAPs is essentially the unification algorithm of [24, 178], but it additionally involves annotation variables in the process of unification. The unification process for the individual variables remains unchanged, and, as in two-sorted languages, unification for the first-order and annotation parts of an annotated formula are handled independently. Following conventional notation, we denote the disagreement set by S, and substitutions by $\theta\lambda$, possibly with subscripts, where θ denotes a first-order substitution, and λ denotes a substitution involving annotations.

Definition 3.6.1 (Cf. Definition 1.2.5.)**.** *A* strictly ground atom *is an annotated atom containing no free individual and no free annotation variables.*

Definition 3.6.2. *Let* $\theta = (v_1/t_1, \ldots, v_m/t_m)$ *be a substitution, where each* v_k *is an individual variable and each* t_k *is a ground term distinct from* v_k*.*
Let $\lambda = ((\mu_1, \nu_1)/\tau_1, \ldots, ((\mu_n, \nu_n)/\tau_n)$ *be a substitution, where each* (μ_i, ν_i) *is an annotation variable and each* τ_k *is an annotation term, and let* E *be an expression. Then* $E\theta\lambda$ *is*

the instance obtained from E by simultaneously replacing each occurrence of the individual variable v_j in E by the term t_j, $j = 1, \ldots, m$, and each occurrence of the annotation variable (μ_k, ν_k) in E by the term τ_k, $k = 1, \ldots, n$. If $E\theta\lambda$ does not contain any individual nor any annotation variables, then $E\theta\lambda$ is called a strictly ground instance *of E.*

Definition 3.6.3. *A* strictly ground instance of a clause $A : (\tau) \leftarrow L_1 : (\tau_1), \ldots, L_n : (\tau_n)$ *is a clause obtained from $A : (\tau) \leftarrow L_1 : (\tau_1), \ldots, L_n : (\tau_n)$ through ground substitution $\theta\lambda$.*

As can be seen from these definitions, the substitutions in BAPs treat the annotation variables similarly to individual variables. The unification as it is defined in this section extends naturally the classical algorithm of unification applied in SLD-resolution, see, for example, [138] or Section 1.6.

Definition 3.6.4 (Cf. Definition 1.6.2)**.** *Let S be a finite set of atomic annotated formulae. The* disagreement set *of S is defined as follows. Locate the leftmost symbol position at which not all the first-order formulae in S have the same symbol, do the same for the annotation symbols, and extract from each such expression in S the subexpression beginning at the leftmost individual and annotation symbols. The set of all such symbols is the disagreement set D_S.*

Example 3.6.1. *Let* $S = \{R_1(f_1(x), f_2(y), a) : (\vartheta((\alpha_1, \nu_1), (\alpha_2, \nu_2))),$
$R_1(f_1(x), z, a) : (\vartheta((\alpha_1, \beta_1), (\alpha_2, \beta_2))),$
$R_1(f_1(x), f_2(y), b) : (\vartheta((\alpha_1, \nu_1), (\alpha_2, \beta_3)))\}.$
The disagreement set of S is $\{f_2(y), z; \nu_1, \beta_1\}$.

Definition 3.6.5. *Let S be a finite set of annotated formulae. A substitution $\theta\lambda$ is called a* unifier *for S if $S\theta\lambda$ is a singleton. A unifier $\theta\lambda$ is called a* most general unifier (mgu) *for*

S if for each unifier $\theta'\lambda'$ of S there exist substitutions γ_1 and γ_2 such that $\theta'\lambda' = \theta\gamma_1\lambda\gamma_2$.

Definition 3.6.6 (Cf. Unification algorithm of Section 1.6.)**.** *Let S be a set of annotated formulae, $\theta\lambda$ with indices be substitutions, and ε be the identity substitution. We define the* annotation unification algorithm *as follows.*

1. *Put $k = 0$ and $\theta_0 = \varepsilon$, $\lambda_0 = \varepsilon$.*

2. *If $S\theta_k\lambda_k$ is a singleton, then stop; $\theta_k\lambda_k$ is an mgu of S. Otherwise, find the disagreement set D_k of $S\theta_k\lambda_k$.*

3. *If there exist v, t, (μ, ν), τ in D_k such that v is an individual variable that does not occur in an individual term t, (μ, ν) is an annotation variable that does not occur in an annotation term τ, then put $\theta_{k+1}\lambda_{k+1} = \theta_k\lambda_k\{v/t; ((\mu, \nu)/\tau\}$, increment k and go to 2. Otherwise, stop. S is not unifiable.*

Example 3.6.2. *Let* $S = R(a, x_1, f_1(f_2(x_2))) : (\vartheta_1(\vartheta_2((\mu_1, \nu_1), (\alpha_1, \beta_1)), \vartheta_2((\mu_2, \nu_2), (\mu_1, \nu_1)))),$
$R(x_2, f_1(x_3), f_1(x_3)) : (\vartheta_1(\mu_3, \nu_3))$. *The unification will be held as follows:*

1. $\theta_0 = \varepsilon$, $\lambda_0 = \varepsilon$.
2. $D_0 = \{a, x_2; (\vartheta_2((\mu_1, \nu_1), (\alpha_1, \beta_1)), \mu_3\}$.

$\theta_1\lambda_1 = \{x_2/a; \mu_3/(\vartheta_2((\mu_1, \nu_1), (\alpha_1, \beta_1))\}$.
$S\theta_1\lambda_1 = \{R(a, x_1, f_1(f_2(a))) : (\vartheta_1(\vartheta_2((\mu_1, \nu_1), (\alpha_1, \beta_1)), \vartheta_2((\mu_2, \nu_2), (\mu_1, \nu_1)))),$
$R(a, f_1(x_3), f_1(x_3)) : (\vartheta_1(\vartheta_2((\mu_1, \nu_1), (\alpha_1, \beta_1)), \nu_3))\}$.

3. $D_1 = \{x_1, f(x_3); \nu_3, \vartheta_2((\mu_2, \nu_2), (\mu_1, \nu_1))\}$.

$\theta_1\lambda_1\theta_2\lambda_2 = \{x_2/a, x_1/f(x_3); \mu_3/(\vartheta_2((\mu_1, \nu_1), (\alpha_1, \beta_1)), \nu_3/\vartheta_2((\mu_2, \nu_2), (\mu_1, \nu_1))\}$.
$S\theta_1\lambda_1\theta_2\lambda_2 = \{R(a, f_1(x_3), f_1(f_2(a))) : (\vartheta_1(\vartheta_2((\mu_1, \nu_1), (\alpha_1, \beta_1)), \vartheta_2((\mu_2, \nu_2), (\mu_1, \nu_1)))),$
$R(a, f_1(x_3), f_1(x_3)) : (\vartheta_1(\vartheta_2((\mu_1, \nu_1), (\alpha_1, \beta_1)), \vartheta_2((\mu_2, \nu_2), (\mu_1, \nu_1))))\}$.

4. $D_2 = \{f_2(a), x_3\}$.

$\theta_1\lambda_1\theta_2\lambda_2\theta_3\lambda_3 =$

$\{x_2/a, x_1/f(x_3), x_3/f_2(a); \mu_3/(\vartheta_2((\mu_1,\nu_1),(\alpha_1,\beta_1)), \nu_3/\vartheta_2((\mu_2,\nu_2),(\mu_1,\nu_1))\}$.

$S\theta_1\lambda_1\theta_2\lambda_2\theta_3\lambda_3 =$

$\{R(a, f_1(f_2(a)), f_1(f_2(a))) : (\vartheta_1(\vartheta_2((\mu_1,\nu_1),(\alpha_1,\beta_1)), \vartheta_2((\mu_2,\nu_2),(\mu_1,\nu_1))))\}$.

The set $S\theta_1\lambda_1\theta_2\lambda_2\theta_3\lambda_3$ *is a singleton, so the algorithm stops.*

Theorem 3.6.1. *Let* S *be a finite set of simple annotated formulae. If* S *is unifiable, then the unification algorithm terminates and gives an mgu for* S*. If* S *is not unifiable, then the unification algorithm stops and reports this fact.*

Proof. Proof is similar to the proof of the analogous theorem in [138], but is given separately for unification of atoms and annotation terms. □

3.7 Conclusions

In this section we described bilattice structures, and discussed the ways in which they can be applied in computer science and logic. We introduced the first-order annotated languages $\mathcal{L}$ based on bilattices, and Bilattice-Based Annotated Logic Programs (BAPs) built using $\mathcal{L}$. We linked BAPs with the conventional logic programs of Chapter 1 throughout the chapter.

We also used this section to introduce some useful and interesting examples of bilattice-based logic programs. These examples will serve us in the next chapter, where we develop declarative and operational semantics for BAPs, and in Chapter 6, where we introduce neural networks for BAPs.

Chapter 4

Declarative and Operational Semantics for Bilattice-Based Annotated Logic Programs

4.1 Introduction

In this chapter, we give declarative and operational semantics for BAPs, annotated logic programs based on distributive bilattices with arbitrary numbers of elements. In particular, we extend the definitions of Chapter 1, such as *Herbrand interpretation, Herbrand base and Herbrand universe, semantic operator*, and *SLD-resolution* given in Chapter 1 for two-valued logics, to the case of BAPs.

The results which we obtain here can be seen as a further development of lattice-based logic programming for handling inconsistencies, see, for example, [54, 107, 108, 167, 28, 198]. However, there are several original results in this chapter, as follows.

First of all, we show that logic programs based on bilattices (and on lattices whose elements are not linearly ordered) may have logical consequences which cannot be computed by the semantic operators defined in the papers just cited. In fact, most authors who have considered lattice and bilattice based logic programs follow the tradition of

classical logic programming and require their semantic operators to work by propagating interpretation from the body of each clause to its head. We use Proposition 3.4.1 and show that this straightforward generalisation is not sufficient when one works with lattice and bilattice-based logic programs. As a result, we define a new semantic operator which guarantees computation of all the logical consequences of bilattice-based annotated logic programs. Our results agree with the paper [107] in which it was first suggested that lattice-based interpretations require some additional inference rules. Unlike the case of [107], we allow infinite (but countable) interpretations for logic programs.

Throughout this chapter, we work with first-order bilattice-based logic programs interpreted by arbitrary (possibly infinite) distributive bilattices having only *finite* joins with respect to the k-ordering. And this framework considerably enriches that of propositional-based logic programs [167, 28, 198, 134] based on finite sets or finite lattices [73, 141, 205, 107, 20, 142, 143]. Moreover, we allow annotations to be variables or to contain function symbols unlike, for example [73, 141, 205, 107, 20, 142, 143].

We prove that the enriched semantic operator we introduce is continuous. As usual, continuity of the semantic operator ensures that it reaches its least fixed point in at most ω iterations. This property is crucial for the whole theory. It makes possible computer implementations of the operator and it also makes it possible to introduce sound and complete proof procedure for BAPs. Note that the semantic operators introduced earlier for annotated logic programs of this generality do not possess this important property, see [108, 28].

Finally, we establish sound and complete SLD-resolution for BAPs. As far as we know, this is the first sound and complete proof procedure for first-order infinitely interpreted (bi)lattice-based annotated logic programs. Compare, for example, our results with those

obtained for constrained resolution for GAPs (Generalised Annotated Logic Programs), which was shown to be incomplete, see [108], or with sound and complete (SLD)- resolutions for finitely-interpreted annotated logic programs (these logic programs do not contain annotation variables and annotation functions) [107, 141, 20, 205, 73, 142, 143].

Note also that [198] introduced computer programs which compute consequence operators for (propositional) bilattice-based logic programs. However, this work cannot be seen as a proof procedure for the simple reason that the semantic operator of [198] is capable only of producing sets of logical consequences of a program. But we look for a proof procedure that can respond to given goals and compute correct answers that are suitable substitutions for individual and/or annotation variables in the goal.

Thus, in summary, we fully describe declarative and operational semantics for a broad class of first-order infinitely-valued bilattice-based annotated logic programs and propose suitable proof procedures and implementations for them.

The structure of the chapter is as follows. In Section 4.2, we develop the fixed-point theory for BAPs and relate it to declarative semantics. In Section 4.3, we define SLD-resolution for BAPs and prove its soundness and completeness. Finally, in Section 4.4 we compare our approach with the work of others in this area including Fitting, Kifer and Subrahmanian, Lakshmanan and Sadri, and Van Emden. We establish that annotation-free and implication-based logic programs can be fully expressed by means of BAPs.

The results of this chapter appeared in [129, 127, 126, 128].

4.2 Declarative Semantics for Bilattice-Based Logic Programs (BAPs)

Let P be a BAP, and let $\mathcal{L}$ be its underlying language based on arbitrary distributive bilattice $\mathbf{B}$ with *finite* joins with respect to $\leq_k$.

Definition 4.2.1 (Cf. Definition 1.4.11)**.** *Similarly to Definition 1.4.11, the* Herbrand Universe $U_{\mathcal{L}}$ *for* $\mathcal{L}$ *(or for* P*) is the set of all ground terms which can be formed out of the constants and function symbols appearing in* $\mathcal{L}$*. (In the case that* $\mathcal{L}$ *has no constants, we add some constant, say* a*, to form ground terms.)*

Example 4.2.1. *Consider the BAP from Example 3.5.2. The Herbrand universe for* $\mathcal{L}$ *is*

$$a_1, a_2, f(a_1), f(a_2), f(f(a_1), f(f(a_2)), \ldots$$

Definition 4.2.2 (Cf. Definition 1.4.12)**.** *The* annotation Herbrand base $B_{\mathcal{L}}$ *for* $\mathcal{L}$ *is the set of all strictly ground atoms formed from predicate symbols of the language* $\mathcal{L}$ *with ground terms from the Herbrand universe as arguments and constants from* $\mathbf{B}$ *as annotations. (In case* $\mathcal{L}$ *has no annotation constants, we add some annotation constant,* $(1,1)$ *say, to form the strictly ground atoms.) Finally, we let ground*(P) *denote the set of all strictly ground instances of clauses of* P*.*

Example 4.2.2. *Consider the BAP from Example 3.5.2. The Herbrand base for* $\mathcal{L}$ *is*

$$R_1(a_1) : (0,0), R_1(a_2) : (0,0), R_1(f(a_1)) : (0,0), R_1(f(a_2)) : (0,0), \ldots$$

$$R_1(a_1) : (0, \frac{1}{4}), R_1(a_2) : (0, \frac{1}{4}), R_1(f(a_1)) : (0, \frac{1}{4}), R_1(f(a_2)) : (0, \frac{1}{4}), \ldots$$

$$R_1(a_1) : (\frac{1}{4}, 0), R_1(a_2) : (\frac{1}{4}, 0), R_1(f(a_1)) : (\frac{1}{4}, 0), R_1(f(a_2)) : (\frac{1}{4}, 0), \ldots$$

$$R_1(a_1) : (\frac{1}{2}, \frac{1}{2}), R_1(a_2) : (\frac{1}{2}, \frac{1}{2}), R_1(f(a_1)) : (\frac{1}{2}, \frac{1}{2}), R_1(f(a_2)) : (\frac{1}{2}, \frac{1}{2}) \ldots$$

$$\vdots$$

and so on for the rest of the annotation and predicate symbols.

An *Herbrand pre-interpretation* J for $\mathcal{L}$ is the same as in Definition 1.4.13.

Definition 4.2.3. *An* annotation Herbrand Interpretation HI for $\mathcal{L}$ *consists of the Herbrand pre-interpretation J with domain D for $\mathcal{L}$ together with the following: for each n-ary predicate symbol in $\mathcal{L}$, the assignment of a mapping from $U_{\mathcal{L}}^n$ into $\mathbf{B}$ as given in Definition 3.4.2.*

Note 4.2.1. *In common with conventional logic programming, each annotation Herbrand interpretation* HI *for P can be identified with the subset* $\{R(t_1, \ldots, t_k) : (\tau) \in B_P \mid R(t_1, \ldots, t_k) : (\tau)$ *receives the value* $\langle 1, 0 \rangle$ *with respect to* HI$\}$ *of B_P it determines, where $R(t_1, \ldots, t_k) : (\tau)$ denotes a typical element of B_P. In future, this identification will be made without further mention. We let* $\mathrm{HI}_{P,\mathbf{B}} = 2^{B_P}$ *denote the set of all annotation Herbrand interpretations for P, and we order* $\mathrm{HI}_{P,\mathbf{B}}$ *by subset inclusion, see Example 3.2.1.*

The following definition and propositions generalise the conventional definition and propositions concerning Herbrand models, as follows.

Definition 4.2.4 (Cf. Definition 1.4.14)**.** *Let $\mathcal{L}$ be an annotated first-order language, S be a set of closed formulae of $\mathcal{L}$. An* annotation Herbrand model *for S is an annotation Herbrand interpretation for $\mathcal{L}$ which is a model for S.*

Proposition 4.2.1. *[Cf. Proposition 1.4.2] Let S be a set of annotated clauses and suppose S has a model. Then S has an annotation Herbrand model.*

Proof. The proof is straightforward consequence of the Note 4.2.1, and is essentially the same as the proof of Proposition 1.4.2. □

Proposition 4.2.2. *[Cf. Proposition 1.4.3] Let S be a set of annotated clauses. Then S is unsatisfiable if and only if S has no annotation Herbrand models.*

Proof. The "only if" part is trivial.

The "if" part. If S is satisfiable, then, according to Proposition 4.2.1, it has an annotation Herbrand model. □

We call the intersection of all Annotation Herbrand models for P the least annotation Herbrand model and denote it by M_P, similarly to Chapter 1.

The following theorem is a generalisation of the theorem which is due to van Emden and Kowalski [204]:

Theorem 4.2.1 (Cf. Theorem 1.5.1.)**.** *Let P be a BAP. Then $M_P = \{A : (\tau) \mid A : (\tau)$ is a logical consequence of $P\}$.*

Proof. Proof is essentially the same as in [204].
$A : (\tau)$ is a logical consequence of $P \iff P \cup \{\sim A : (\tau)\}$ is unsatisfiable (by Proposition 3.4.2) $\iff P \cup \{\sim A : (\tau)\}$ has no annotation Herbrand models (by Proposition 4.2.2) $\iff \sim A : (\tau)$ receives value $\langle 0, 1\rangle$ with respect to all annotation Herbrand models of P $\iff A : (\tau)$ receives value $\langle 1, 0\rangle$ with respect to all annotation Herbrand models of P $\iff A : (\tau) \in M_P$. □

We define here two consequence operators, $\widehat{\mathcal{T}_P}$ and $\mathcal{T}_P$, and both compute the logical consequences of BAPs. The operator $\widehat{\mathcal{T}_P}$ is a straightforward generalisation of T_P from Definition 1.5.1, but it does not compute all the logical consequences of P in the general case, and therefore we call it the restricted semantic operator. We will use its properties when discussing the relationship of BAPs to annotation-free and implication-based bilattice logic programs. The semantic operator $\mathcal{T}_P$ is an extended version of $\widehat{\mathcal{T}_P}$. It reflects properties established in Proposition 3.4.1, items 1, 3, 5 and this guarantees that $\mathcal{T}_P$ computes all the logical consequences of a given program.

Definition 4.2.5. *We define the mapping* $\widehat{\mathcal{T}}_P : \mathrm{HI}_{P,\mathbf{B}} \to \mathrm{HI}_{P,\mathbf{B}}$ *by* $\widehat{\mathcal{T}}_P(\mathrm{HI}) = \{A : (\tau) \in B_P | A : (\tau) \leftarrow L_1 : (\tau_1), \ldots, L_n : (\tau_n)$ *is a strictly ground instance of a clause in* P*, and* $\{L_1 : (\tau_1'), \ldots, L_n : (\tau_n')\} \subseteq \mathrm{HI}\}$*, such that* $\tau_i \leq_k \tau_i'$*, for each* $i \in \{1, \ldots, n\}$.

Definition 4.2.6. *We define the mapping* $\mathcal{T}_P : \mathrm{HI}_{P,\mathbf{B}} \to \mathrm{HI}_{P,\mathbf{B}}$ *as follows:* $\mathcal{T}_P(\mathrm{HI}) =$
$= \{A : (\tau) \in B_P$ *such that*

1. *either* $A : (\tau) \leftarrow L_1 : (\tau_1), \ldots, L_n : (\tau_n)$ *is a strictly ground instance of a clause in* P *and* $\{L_1 : (\tau_1'), \ldots, L_n : (\tau_n')\} \subseteq \mathrm{HI}$*, and for each* (τ_i')

$$(\tau_i) \leq_k (\tau_i'),$$

2. *or there are annotated strictly ground atoms* $A : (\tau_1^*), \ldots, A : (\tau_k^*) \in \mathrm{HI}$ *such that* $(\tau) \leq_k (\tau_1^*) \oplus \ldots \oplus (\tau_k^*)\}$.

Note 4.2.2. *By Proposition 3.4.1, whenever* $F : (\tau) \in \mathrm{HI}$ *and* $(\tau') \leq_k (\tau)$*, then* $F : (\tau') \in \mathrm{HI}$.

Also, for each formula F*,* $F : (0, 0) \in \mathrm{HI}$.

Note that this holds in particular for $\widehat{\mathcal{T}}_P(\mathrm{HI})$ *and for* $\mathcal{T}_P(\mathrm{HI})$*. Definitions 4.2.5 and 4.2.6 use* HI *and thus ensure that at any step of derivation, the presence of ground atoms annotated by* $(0,0)$ *will be taken into account. Also, in both definitions the property that whenever* $F:(\tau)\in \mathrm{HI}$ *and* $(\tau')\leq_k(\tau)$*, then* $F:(\tau')\in \mathrm{HI}$ *is reflected.*

This definition is based on the definition of a BAP as a set of Horn clauses and reflects the results established in Proposition 3.4.1, items 1, 3, 5. For example, from Proposition 3.4.1 it follows that if $|A:(\tau)\oplus A:(\tau')|=\langle 1,0\rangle$, then $|A:(\tau\widetilde{\oplus}\tau')|=\langle 1,0\rangle$. So, the use of $\oplus$ is needed in the definition of $\mathcal{T}_P$. Consider, for example the situation when different clauses have the same head:

$$\begin{aligned} A:(\mu,\nu) &\leftarrow B:(1,0)\\ A:(\mu,\nu) &\leftarrow B:(0,1). \end{aligned}$$

In case either $B:(1,0)\in \mathrm{HI}$ or $B:(0,1)\in \mathrm{HI}$, the semantic operator will add $A:(0,1)$ and $A:(1,0)$ to HI. But, by Definition 3.4.2, the latter is possible only if $(0,1)\leq_k \mathcal{I}(A)$ and $(1,0)\leq_k \mathcal{I}(A)$, but then, by monotonicity of $\oplus$ and the fact that $\oplus$ is idempotent, $(0,1)\oplus(1,0)\leq_k \mathcal{I}(A)$, and thus, $(1,1)\leq_k \mathcal{I}(A)$. This means, in turn, that $A:(1,1)\in \mathrm{HI}$. The same argument would hold for $\otimes$, but, item 1 from Definition of 4.2.6 has already taken care of such cases.

Note also that item 1 in Definition 4.2.6 is the same as given in Definition 4.2.5, and reflects the property stated in Item 1 of Proposition 3.4.1; item 2 in Definition 4.2.6 extends the Definition of $\widehat{\mathcal{T}_P}$ in order to reflect the results established in items 3 and 5 of Proposition 3.4.1.

As can be seen from Definitions 4.2.5 and 4.2.6, the sets of formulae computed by $\widehat{\mathcal{T}_P}$

are subsets of the sets of formulae computed by $\mathcal{T}_P$ at each iteration of the two semantic operators. We will further illustrate this in Examples 4.2.4, 4.2.5.

The operator $\mathcal{T}_P$ is monotonic, which follows from monotonicity of the annotation functions $\oplus$, $\otimes$ and $\neg$ with respect to $\leq_k$. Since every bilattice $\mathbf{B}$ we consider is a complete lattice and $\mathcal{T}_P$ is monotonic, by the Knaster-Tarski theorem (see Theorem 1.5.2), $\mathcal{T}_P$ has a *least fixed point*. And we can use the transfinite sequence (see Definition 1.5.6) to compute the least fixed point, similarly to Theorem 1.5.2. This sequence increases with increasing ordinals, eventually becoming constant, settling on the least fixed point of $\mathcal{T}_P$. The smallest ordinal α such that $\mathcal{T}_P \uparrow \alpha$ is the least fixed point of $\mathcal{T}_P$ is called the closure ordinal of $\mathcal{T}_P$.

We will illustrate how to compute the least fixed points of $\mathcal{T}_P$ for the annotated logic programs we introduced in Section 3.5.

For the next program, the semantic operators $\widehat{\mathcal{T}_P}$ and $\mathcal{T}_P$ compute the same Herbrand models.

In all the examples we assume the properties of $\mathcal{T}_P$ and $\widehat{\mathcal{T}_P}$ stated in Note 4.2.2. Thus, for example, when writing $B : (1, 0) \in \mathcal{T}_P \uparrow k$ we assume that $B : (0, 0) \in \mathcal{T}_P \uparrow k$.

Example 4.2.3. *Consider the BAP from Example 3.5.2, and the following process of computing $\mathcal{T}_P$.*

$\mathcal{T}_P \uparrow 0 = \emptyset$;

$\mathcal{T}_P \uparrow 1 = \mathcal{T}_P(\mathcal{T}_P(\emptyset)) = \{R_1(a_1) : (1,1), R_2(f(a_1)) : (0, \frac{1}{4}), R_2(f(a_2)) : (0, \frac{1}{4}), R_3(a_2) : (\frac{3}{4}, 1), R_4(a_1) : (\frac{1}{2}, \frac{1}{2})\}$;

$\mathcal{T}_P \uparrow 2 = \mathcal{T}_P\left(\mathcal{T}_P\left(\mathcal{T}_P\left(\emptyset\right)\right)\right) =$

$\mathcal{T}_P\{R_1(a_1) : (1,1), R_2(f(a_1)) : (0, \frac{1}{4}), R_2(f(a_2)) : (0, \frac{1}{4}), R_3(a_2) : (\frac{3}{4}, 1), R_4(a_1) : (\frac{1}{2}, \frac{1}{2})\} =$

$\{R_1(a_1) : (1,1), R_2(f(a_1)) : (0, \frac{1}{4}), R_2(f(a_2)) : (0, \frac{1}{4}), R_3(a_2) : (\frac{3}{4}, 1), R_4(a_1) : (\frac{1}{2}, \frac{1}{2}),$

$R_3(f(a_1)) : (0, \frac{1}{4}), R_3(f(a_2)) : (0, \frac{1}{4})\}$;

$\mathcal{T}_P \uparrow 3 = \mathcal{T}_P(\mathcal{T}_P \uparrow 2) = \{R_1(a_1) : (1,1), R_2(f(a_1)) : (0, \frac{1}{4}), R_2(f(a_2)) : (0, \frac{1}{4}), R_3(a_2) : (\frac{3}{4}, 1), R_4(a_1) : (\frac{1}{2}, \frac{1}{2}), R_3(f(a_1)) : (0, \frac{1}{4}), R_3(f(a_2)) : (0, \frac{1}{4}), R_4(f(a_1)) : (1,0), R_4(f(a_2)) : (1,0)\}$

$\mathcal{T}_P \uparrow 4 = \mathcal{T}_P(\mathcal{T}_P \uparrow 3) = \{R_1(a_1) : (1,1), R_2(f(a_1)) : (0, \frac{1}{4}), R_2(f(a_2)) : (0, \frac{1}{4}), R_3(a_2) : (\frac{3}{4}, 1), R_4(a_1) : (\frac{1}{2}, \frac{1}{2}), R_3(f(a_1)) : (0, \frac{1}{4}), R_3(f(a_2)) : (0, \frac{1}{4}), R_4(f(a_1)) : (1,0), R_4(f(a_2)) : (1,0), R_5(f(a_1)) : ((\frac{1}{4}, 1)\widetilde{\wedge}(\frac{1}{2}, 0))\}$.

Therefore, $\mathcal{T}_P \uparrow 4$ is the least fixed point of $\mathcal{T}_P$.

The next example displays the difference between $\widehat{\mathcal{T}_P}$ and $\mathcal{T}_P$.

Example 4.2.4. *Consider the logic program from Example 3.5.1, based on the four-valued bilattice. And consider the computations of the least fixed point for $\widehat{\mathcal{T}_P}$ and $\mathcal{T}_P$. We put in square brackets the numbers of the rules from the definition of $\mathcal{T}_P$ which enabled us to derive that element.*

Iteration	$\mathcal{T}_P$	$\widehat{\mathcal{T}_P}$
1	$B : (0,1)$*[1]*, $B : (1,0)$*[1]*	$B : (0,1)$, $B : (1,0)$
2	$B : (0,1)$*[1]*, $B : (1,0)$*[1]*, $A : (1,1)$ *[2]*	–
3	$C : (1,0)$ *[1]*	–

Example 4.2.5. *Consider the logic program from Example 3.5.3, and the least fixed points of $\mathcal{T}_P$ and $\widehat{\mathcal{T}_P}$. We put in square brackets the numbers of the rules from the definition of $\mathcal{T}_P$ which enabled us to derive that element.*

Iteration	$\mathcal{T}_P$	$\widehat{\mathcal{T}_P}$
1	$Storm(Monday) : (\frac{3}{4}, \frac{1}{2})$*[1]*, $Storm(Monday) : (\frac{1}{2}, \frac{3}{4})$*[1]*, $Storm(Tuesday) : (\frac{1}{2}, \frac{3}{4})$*[1]*	$Storm(Monday) : (\frac{3}{4}, \frac{1}{2})$, $Storm(Monday) : (\frac{1}{2}, \frac{3}{4})$, $Storm(Tuesday) : (\frac{1}{2}, \frac{3}{4})$
2	$Storm(Monday) : (\frac{3}{4}, \frac{3}{4})$*[2]*	—
3	$Delay(1, Monday) : ((\frac{3}{4}, \frac{1}{2}) \vee (\frac{1}{2}, \frac{1}{2}))$*[1]*, $Delay(2, Monday) : ((\frac{3}{4}, \frac{1}{2}) \vee (\frac{1}{2}, \frac{1}{2}))$*[1]*	—
4	$Cancel(1, Monday) : (1, 0)$*[1]*, $Cancel(2, Monday) : (1, 0)$*[1]*	—

Note that we use the fact that $((\frac{3}{4}, \frac{1}{2}) \vee (\frac{1}{2}, \frac{1}{2})) = (\frac{3}{4}, \frac{1}{2})$ *and* $((\frac{3}{4}, \frac{1}{2}) \oplus (\frac{1}{2}, \frac{3}{4})) = (\frac{3}{4}, \frac{3}{4})$.

Thus, in this example we can see that the program will definitely cancel both flights 1 *and* 2 *on Monday.*

The following example shows the fixed point of the semantic operators reached in ω steps.

Example 4.2.6. *Consider the logic program given in Example 3.5.4. The least fixed point of this program (for* $\widehat{\mathcal{T}_P}$ *and for* $\mathcal{T}_P$*) is*

$\{R_1(a_1) : (1, 0.5)$,

$R_2(f(a_1)) : (\vartheta_1(1), \vartheta_2(0.5))$,

$R_1(f(f(a_1))) : (\vartheta_3(\vartheta_1(1)), \vartheta_4(\vartheta_2(0.5)))$,

$\vdots$

$R_1(f^{n-1}(a_1)) : (\vartheta_3^n(\vartheta_1^{n-1} \ldots (\ldots((1))\ldots)), \vartheta_4^n(\vartheta_2^{n-1} \ldots (\ldots((0.5))\ldots)))$,

$R_2(f^n(a_1)) : (\vartheta_1^n(\vartheta_3^{n-1}(\vartheta_1^{n-2} \ldots (\ldots((1))\ldots))), \vartheta_2^n(\vartheta_4^{n-1}(\vartheta_2^{n-2} \ldots (\ldots((0.5))\ldots)))), \ldots\}$,

where $n \in \omega$. Here, ϑ_1, ϑ_2, ϑ_3, ϑ_4 stand for the functions $\frac{\mu}{2}$, $\frac{\nu}{3}$, $\frac{\mu}{3}$, $\frac{\nu}{3}$ respectively.

4.2.1 Continuity of $\mathcal{T}_P$.

Continuity of $\mathcal{T}_P$ is the property that helps to characterise computationally the least fixed point of $\mathcal{T}_P$, and also ensures that sound and complete SLD-resolution can be introduced relative to the models computed by $\mathcal{T}_P$. In this subsection we show that $\mathcal{T}_P$ is continuous, and characterise its fixed points.

Results concerning the continuity of semantic operators determined by bilattice structures first appeared in [49]. The following definition and theorem are due to Fitting [49].

Definition 4.2.7. *A logic program is called* k-existential *provided* Σ *is the only infinite operation symbol allowed to appear in program clause bodies. That is, none of* Π, $\forall$ *or* $\exists$ *are allowed.*

Clearly, BAPs defined in Chapter 3 and Definition 3.5.2 are k-existential programs.

Theorem 4.2.2. *Suppose* $\mathbf{B}$ *is a bilattice that meets the infinite distributivity conditions of Definition 3.2.13, and P is a k-existential program. Then the semantic operator defined on P is continuous and its closure ordinal is ω.*

Since we did not follow Fitting's definitions of an annotation-free logic program and gave an original definition of the $\mathcal{T}_P$ operator that differs from Fitting's semantic operator, we need to present our own proof of continuity of $\mathcal{T}_P$ here, as follows.

Theorem 4.2.3. *Let P be a BAP, then the mapping $\mathcal{T}_P$ is continuous.*

Proof. Let X be a directed subset of 2^{B_P}. In order to show that $\mathcal{T}_P$ is continuous, we have to show that $\mathcal{T}_P(\bigcup X) = \bigcup \mathcal{T}_P(X)$ for each directed subset X of $\mathrm{HI}_{P,\mathbf{B}}$. By directedness, it

follows that $\{L_1 : (\tau_1), \ldots, L_n : (\tau_n)\} \subseteq \bigcup X$ if and only if $\{L_1 : (\tau_1), \ldots, L_n : (\tau_n)\} \subseteq \text{HI}$, for some $\text{HI} \in X$.

Assume $A : (\tau) \in \mathcal{T}_P(\bigcup X)$

$\iff$ $A : (\tau) \leftarrow L_1 : (\tau_1), \ldots, L_n : (\tau_n)$ is a strictly ground instance of a clause in P and $\{L_1 : (\tau_1), \ldots, L_n : (\tau_n)\} \subseteq \bigcup X$

$\iff$ $A : (\tau) \leftarrow L_1 : (\tau_1), \ldots, L_n : (\tau_n)$ is a strictly ground instance of a clause in P and $\{L_1 : (\tau_1), \ldots, L_n : (\tau_n)\} \subseteq \text{HI}$ for some $\text{HI} \in X$

$\iff$ $A : (\tau) \in \mathcal{T}_P(\text{HI})$, for some $\text{HI} \in X$

$\iff$ $A : (\tau) \in \bigcup \mathcal{T}_P(X)$. □

Note that $\mathcal{T}_P$ works with sets of strictly ground formulae, and not with interpretations themselves and this is why it is continuous, comparing, for example, with the analogous semantic operators defined, for example, in [108, 28]. This is possible because the underlying bilattice of each BAP is required to have only *finite* k-joins. Because the continuity of semantic operators is crucial for us, we will discuss in some detail why $\mathcal{T}_P$ differs from the analogous semantic operator of Kifer and Subrahmanian [108] in this respect.

We cited the definition of the restricted semantic operator of [108] in Chapter 2, in Definition 2.0.2. We will use an example from [108] showing that the (restricted) semantic operator for Generalised Annotated Logic Programs (GAPs) is not continuous. This example can be extended to the language of bilattice-based annotated logic programs we introduce as follows.

Example 4.2.7.

$$\begin{aligned} Q_1 : (\frac{\mu+1}{2}, 0) &\leftarrow Q_1 : (\mu, 0) \\ Q_2 : (1, 0) &\leftarrow Q_1 : (1, 0) \\ Q_1 : (0, 0) &\leftarrow \end{aligned}$$

Moreover, if we simply generalise the (restricted) semantic operator of Definition 2.0.2 to the bilattice case, this example will show that this restricted semantic operator reaches its least fixed point in $\omega + 1$ steps, and, consequently, is not continuous. Consider, for example, the following generalisation of the (restricted) operator given in Definition 2.0.2 to the bilattice-based case: $T_P(I)(A) = \sum\{\vartheta((\tau_1), \ldots, (\tau_n)) | A : \vartheta((\tau_1), \ldots, (\tau_n)) \leftarrow B_1 : (\tau_1), \ldots, B_n : (\tau_n)$ is a strictly ground instance of a clause in P and $I \models (B_1 : (\tau_1), \ldots, B_n : (\tau_n))\}$. This semantic operator reaches $Q_1 : (1, 0)$ in $\omega + 1$ steps. The semantic operator $\mathcal{T}_P$ we have defined will reach neither $Q_1 : (1, 0)$ nor $Q_2 : (1, 0)$, because only finite operation $\oplus$ (as opposed to infinite $\sum$) is used in item 2 of Definition 4.2.6. And thus neither $Q_1 : (1, 0)$ nor $Q_2 : (1, 0)$ will be computed as logical consequences of the given program. This is the main reason for restricting the choice of underlying bilattices only to those which have only *finite* joins with respect to the k-ordering. If we assumed that a bilattice had infinite joins with respect to the k-ordering, then, for example $Q_1 : (1, 0)$ and $Q_2 : (1, 0)$ would have been logical consequences of the logic program from Example 4.2.7. However, $\mathcal{T}_P$ would not compute these logical consequences. Thus, we exclude logic programs similar to the one from Example 4.2.7 from our analysis, and take only logic programs that can be interpreted by bilattices with *finite* joins.

Having defined a continuous semantic operator, we wish to prove that, for a given BAP P, its least fixed point characterises the least Herbrand model of P. We proceed as follows. The next two propositions prepare us to prove the theorem concerning the least

fixed point characterisation of the least Herbrand models of BAPs.

Proposition 4.2.3. *Let* $A : (\tau_1)$ *and* $A : (\tau_2)$ *be strictly ground annotated atoms. Then* $A : (\tau_1) \in \mathrm{HI}$ *and* $A : (\tau_2) \in \mathrm{HI}$ *implies that* $A : (\tau_1 \oplus \tau_2) \in \mathrm{HI}$.

Proof. The proof is a simple generalisation of item 5 from Proposition 3.4.1. It uses the monotonicity of $\oplus$ and Definition 3.4.2, as follows. If $A : \tau_1 \in \mathrm{HI}$ and $A : \tau_2 \in \mathrm{HI}$, this means that $|A : \tau_1|_{\mathrm{HI}} = \langle 1, 0 \rangle$ and $|A : \tau_2|_{\mathrm{HI}} = \langle 1, 0 \rangle$. But then, for the $\mathcal{I}$ used to define HI, $\tau_1 \leq_k |A|_{\mathcal{I}}$ and $\tau_2 \leq_k |A|_{\mathcal{I}}$. By monotonicity of $\oplus$, $\tau_1 \oplus \tau_2 \leq_k |A|_{\mathcal{I}} \oplus |A|_{\mathcal{I}}$. By idempotency of $\oplus$, $\tau_1 \oplus \tau_2 \leq_k |A|_{\mathcal{I}}$. But then, by Definition 3.4.2, $|A : (\tau_1 \oplus \tau_2)|_{\mathrm{HI}} = \langle 1, 0 \rangle$. Thus means that $A : (\tau_1 \oplus \tau_2) \in \mathrm{HI}$.

□

It is possible to prove a similar proposition using $\otimes$, but such a proposition would be trivial, and would just duplicate Item 1 of Proposition 3.4.1.

Proposition 4.2.4. *Let* HI *be an annotation Herbrand interpretation for* P. *Then* HI *is a model for* P *if and only if* $\mathcal{T}_P(\mathrm{HI}) \subseteq \mathrm{HI}$.

Proof. HI is a model for P if and only if,

1. (Cf. Proposition 3.4.1, item 1) for each strictly ground instance of $A : (\tau) \leftarrow A_1 : (\tau_1), \ldots, A_k : (\tau_k)$ of each clause in P we have

   $\{A_1 : (\tau_1^*), \ldots, A_k : (\tau_k^*)\} \subseteq \mathrm{HI}$, and $(\tau_i^*) \leq_k (\tau_i)$, for each $i = 1, \ldots, k$ implies $A : (\tau) \in \mathrm{HI}$.

2. (Cf. Proposition 3.4.1, item 5 and Proposition 4.2.3)

   $\{A_j : (\tau_1), \ldots, A_j : (\tau_n)\} \subseteq \mathrm{HI}$ implies $\{A_j : ((\tau_1) \oplus \ldots \oplus (\tau_j))\} \in \mathrm{HI}$, for any formulae $\{A_j : (\tau_1), \ldots, A_j : (\tau_n)$ in HI.

3. (Cf. Proposition 3.4.1, item 11) For each ground A, $A : (0, 0) \in \text{HI}$.

4. (Cf. Proposition 3.4.1, item 1) If $A : (\tau) \in \text{HI}$, then $A : (\tau') \in \text{HI}$, for every $\tau' \leq_k \tau$.

5. (Cf. Proposition 3.4.1, item 5 and Proposition 4.2.3)
   $\{A_j : (\tau_1), \ldots, A_j : (\tau_n)\} \subseteq \text{HI}$ implies $\{A_j : ((\tau_1) \otimes \ldots \otimes (\tau_j))\} \in \text{HI}$, for any formulae $\{A_j : (\tau_1), \ldots, A_j : (\tau_n)$ in HI. Similar statement holds for the infinite meet Π.

All these cases hold if and only if $\mathcal{T}_P(\text{HI}) \subseteq \text{HI}$: indeed, the first case corresponds to item 1 of Definition 4.2.6 of $\mathcal{T}_P$; and the second case corresponds to item 2 in Definition of $\mathcal{T}_P$. By Note 4.2.2, the third case is covered by Definition 4.2.6. The fourth and fifth cases are covered in item 1 of Definition 4.2.6, see also Note 4.2.2 for explanations.

□

Now, using Kleene's theorem, Theorem 1.5.3, we can assert that $\text{lfp}(\mathcal{T}_P) = \mathcal{T}_P \uparrow \omega$. We have the following generalisation of a theorem which is due to van Emden and Kowalski [204].

Theorem 4.2.4 (Cf. Theorem 1.5.4)**.**

$$M_P = \text{lfp}(\mathcal{T}_P) = \mathcal{T}_P \uparrow \omega.$$

Proof. $M_P = \text{glb}\{\text{HI} \mid \text{HI is an Annotation Herbrand model for } P\} = \text{glb}\{\text{HI} \mid \mathcal{T}_P(\text{HI}) \subseteq \text{HI}\}$ (by Proposition 4.2.4)$= \text{lfp}(\mathcal{T}_P)$ (by definition of the least fixed point) $= \mathcal{T}_P \uparrow \omega$ (by Theorem 4.2.3 and Theorem 1.5.3). □

This theorem completes the section about declarative semantics for BAPs.

The next definition will provide a link between this section and the next section about SLD-resolution.

Definition 4.2.8 (Cf. Definition 1.5.7)**.** *Let P be a BAP, G a goal $\leftarrow A_1 : (\tau_1), \ldots, A_k : (\tau_k)$. An* answer *for $P \cup \{G\}$ is a substitution $\theta\lambda$ for individual and annotation variables of G. We say that $\theta\lambda$ is a* correct answer *for $P \cup \{G\}$ if $\Pi((A_1 : (\tau_1), \ldots, A_k : (\tau_k))\theta\lambda)$ is a logical consequence of P.*

The next section will provide a procedural counterpart of the notion of a correct answer.

4.3 SLD-resolution for Bilattice-Based Logic Programs

The resolution method was first introduced by Robinson, adapted to logic programming by Kowalski [131] and was implemented (as SLD-resolution) in two-valued logic programming by Colmerauer et al. A detailed exposition of this can be found in [138] and Chapter 1, and for many-valued resolution procedures see [75, 107, 203, 108, 141, 186], for example.

SLD-resolution for lattice-based logic programs with annotation functions and annotation variables was first introduced in [108] and was shown to be incomplete.

Kifer and Lozinskii showed in [107] that unlike classical refutation procedures, where only the resolution rule is applied, lattice-based theories need to incorporate four procedures: resolution, factorisation, reduction and elimination in order to be sound and complete. This enriches the set of resolutive inference rules for many-valued logics which have linearly ordered sets of values, as was defined, for example, in [75, 205]. Some very interesting ideas about the relationship between resolutions for languages with ordered and non-ordered annotations can be found in [142, 143].

Unlike all the mentioned papers, this chapter works with bilattices, and not just lattice-based programs. Also, comparing with all these papers, we allow variables and functions

in annotations and adopt additional refutation rules (which correspond to some of the rules of [107]) in order to obtain soundness and completeness of SLD-resolution for BAPs. Note that in [107, 142, 143] only constant annotations are allowed in the language and therefore each logic program becomes finitely interpreted in these settings. We extend all our results to infinitely interpreted programs with functions and variables in annotations. Finally, we establish an operational semantics for BAPs and prove its soundness and completeness.

Throughout this section, we denote a BAP by P, and a BAP goal by G. We refer to [138] and Section 1.6 and 3.6 herein for a description of the unification algorithm first defined in [24] for classical logic programming. As we illustrated in Section 3.6, the unification algorithm for BAPs is essentially the unification algorithm of [24], but it additionally involves annotation variables in the process of unification. The unification process for the individual variables remains unchanged, and, as in two-sorted languages, unification for the first-order and annotation parts of an annotated formula are handled independently. Following conventional notation, we denote the disagreement set by S, and substitutions by $\theta\lambda$, possibly with subscripts, where θ denotes a first-order substitution, and λ denotes a substitution involving annotations.

4.3.1 SLD-refutation for BAPs

Definition 4.3.1. *[SLD-derivation] Let G_i be the annotated goal $\leftarrow B_1 : (\tau_1), \ldots, B_k : (\tau_k)$, and let $C_1, \ldots, C_l$ be the annotated clauses $A_1 : (\tau_1^*) \leftarrow body_1, \ldots, A_l : (\tau_l^*) \leftarrow body_l$, where each $body_i$ from $body_1, \ldots, body_l$ denotes a body of the clause C_i. Then the goal G_{i+1} is* derived *from G_i and $C_1, \ldots, C_l$ using mgu (see Definition 3.6.5) $\theta\lambda$ if the following conditions hold:*

1. $B_m : (\tau_m)$ *is an annotated atom, called the selected atom, in* G_i *and*

   (a) θ *is an* mgu *of* B_m *and* A_1*, and one of the following conditions holds: either* λ *is an* mgu *of* (τ_m) *and* (τ_1^*)*; or* $(\tau_m)\lambda$ *and* $(\tau_1^*)\lambda$ *receive constant values such that* $(\tau_m)\lambda \leq_k (\tau_1^*)\lambda$*;*

   or

   (b) θ *is an* mgu *of* B_m *and* $A_1, \ldots, A_l$*, and either* λ *is an* mgu *of* (τ_m) *and* $(\tau_1^*), \ldots, (\tau_l^*)$ *or* $(\tau_m)\lambda$ *and* $(\tau^*)\lambda, \ldots, (\tau_l^*)\lambda$ *receive constant values such that* $(\tau_m)\lambda \leq_k ((\tau_1^*)\lambda \oplus \ldots \oplus (\tau_l^*)\lambda)$*.*

2. *in case 1a,* G_{i+1} *is the goal* $(\leftarrow B_1 : (\tau_1), \ldots, B_{m-1} : (\tau_{m-1}), body_1, B_{m+1} : (\tau_{m+1}), \ldots, B_k : (\tau_k))\theta\lambda$*. In this case,* G_{i+1} *is said to be derived from* G_i *and* C_1 *using* $\theta\lambda$*.*

3. *in case 1b,* G_{i+1} *is the goal* $(\leftarrow B_1 : (\tau_1), \ldots, B_{m-1} : (\tau_{m-1}), body_1, \ldots, body_l, B_{m+1} : (\tau_{m+1}), \ldots, B_k : (\tau_k))\theta\lambda$*. In this case,* G_{i+1} *is said to be derived from* G_i*,* $C_1, \ldots, C_l$ *using* $\theta\lambda$*.*

4. *Whenever a goal* G_i *contains a formula of the form* $F : (0, 0)$*, then remove* $F : (0, 0)$ *from the goal and form the next goal* G_{i+1} *that is* G_i *except that it does not contain* $F : (0, 0)$*.*

Note that certain items in the definition of derivation correspond to certain items in Definition 4.2.6 of $\mathcal{T}_P$, for example, item 1a corresponds to the item 1 in Definition 4.2.6, item 2 corresponds to the item 2 in Definition 4.2.6. And, as we have noticed before in relation to the definition of $\mathcal{T}_P$, all these items serve to reflect the model properties of BAPs captured in Proposition 3.4.1, see also Propositions 4.2.3 and 4.2.4 for more detailed discussion.

Definition 4.3.2. *Suppose that P is a BAP and G_0 is a goal. An* SLD-derivation *of $P \cup \{G_0\}$ consists of a sequence $G_0, G_1, G_2 \ldots$ of BAP goals, a sequence of finite sets of clauses $C_1, C_2, \ldots$ of BAP clauses and a sequence $\theta_1\lambda_1, \theta_2\lambda_2, \ldots$ of* mgus *such that each G_{i+1} is derived from G_i and C_{i+1} using $\theta_{i+1}\lambda_{i+1}$.*

Note that $C_1, C_2, \ldots$ is defined to be a sequence of *finite sets* of clauses, and not just a sequence of clauses, as in classical SLD-resolution. This happens because item 1b admits the use of a finite set of clauses at each step of derivation. This item was not included in the classical definition of SLD-resolution.

In [121], we gave a many-sorted representation of BAPs. This translation allowed us to apply the classical definition of SLD-resolution to many-sorted translations of BAPs. This result showed that in principle, one can use just *sequences of clauses*, and not *sequences of finite sets of clauses* when defining many-sorted *SLD-derivation* for BAPs.

Definition 4.3.3. *An* SLD-refutation *of $P \cup \{G_0\}$ is a finite SLD-derivation of $P \cup \{G\}$ which has the empty clause $\Box$ as the last goal of the derivation. If $G_n = \Box$, we say that the refutation has length n.*

Definition 4.3.4. *An* unrestricted SLD-refutation *is an SLD-refutation, except that we drop the requirement that the substitutions $\theta_i\lambda_i$ be most general unifiers. They are only required to be unifiers.*

Definition 4.3.5. *The success set of P is the set of all $A : (\tau) \in B_P$ such that $P \cup \{\leftarrow A : (\tau)\}$ has an SLD-refutation.*

Definition 4.3.6. *A* computed answer $\theta\lambda$ *for $P \cup \{G_0\}$ is the substitution obtained by restricting the composition of $\theta_1, \ldots, \theta_n, \lambda_1, \ldots, \lambda_k$ to the variables of G_0, where $\theta_1, \ldots, \theta_n, \lambda_1, \ldots, \lambda_k$ is the sequence of* mgus *used in the SLD-refutation of $P \cup \{G_0\}$.*

We will give two examples of how this SLD-resolution is run, using the logic programs from Examples 3.5.2 and 3.5.3.

Example 4.3.1. *Let P be the program from Example 3.5.2. And G_0 be a goal $\leftarrow R_3(f(a_1)) : (0, \frac{1}{4}), R_5(f(a_1)) : ((\frac{1}{4}, 1)\widetilde{\wedge}(\frac{1}{2}, 0))$.*

1. *Let $R_3(f(a_1)) : (0, \frac{1}{4})$ be a selected atom. We have a clause $R_3(x) : (\mu, \nu) \leftarrow R_2(x) : (\mu, \nu)$ in P. Form the set $S = \{R_3(f(a_1)) : (0, \frac{1}{4}), R_3(x) : (\mu, \nu)\}$. Get its mgu:*

   (a) $D_0 = \{f(a_1), x; 0, \mu\}$.

   (b) $\theta_0\lambda_0 = \{x/f(a_1); \mu/0\}$.

   (c) $S\theta_0\lambda_0 = \{R_3(f(a_1)) : (0, \frac{1}{4}), R_3(f(a_1)) : (0, \nu)\}$.

   (a) $D_1 = \{\frac{1}{4}, \nu\}$.

   (b) $\theta_0\lambda_0\theta_1\lambda_1 = \{x/f(a_1); \mu/0, \nu/\frac{1}{4}\}$.

   (c) $S\theta_0\lambda_0\theta_1\lambda_1 = \{R_3(f(a_1)) : (0, \frac{1}{4})\}$, *which means that $\theta_1\lambda_1$ is the mgu of S.*

2. *By item 2 of Definition 4.3.1, G_1 is $\leftarrow (R_2(x) : (\mu, \nu), R_5(f(a_1)) : ((\frac{1}{4}, 1)\widetilde{\wedge}(\frac{1}{2}, 0)))\theta_1\lambda_1$, that is, $\leftarrow (R_2(f(a_1)) : (0, \frac{1}{4}), R_5(f(a_1)) : ((\frac{1}{4}, 1)\widetilde{\wedge}(\frac{1}{2}, 0))$. Let $(R_2(f(a_1)) : (0, \frac{1}{4})$ be a selected atom. We have a clause $R_2(f(x)) : (0, \frac{1}{4}) \leftarrow$. Form the set $S_1 = (R_2(f(a_1)) : (0, \frac{1}{4}), R_2(f(x)) : (0, \frac{1}{4})$. Get its mgu:*

   (a) $D_2 = \{x, a_1\}$.

   (b) $\theta_2\lambda_2 = \{x/a_1\}$.

   (c) $S_1\theta_2\lambda_2 = \{R_2(f(a_1)) : (0, \frac{1}{4})\}$ *Thus, $\theta_2\lambda_2$ is the mgu of S_1.*

3. *By item 2 of Definition 4.3.1, G_2 is $\leftarrow (R_5(f(a_1)) : ((\frac{1}{4}, 1)\widetilde{\wedge}(\frac{1}{2}, 0)))\theta_2\lambda_2$, that is, $\leftarrow R_5(f(a_1)) : ((\frac{1}{4}, 1)\widetilde{\wedge}(\frac{1}{2}, 0))$. Now $R_5(f(a_1)) : ((\frac{1}{4}, 1)\widetilde{\wedge}(\frac{1}{2}, 0))$ is a selected atom.*

We have a clause $R_5(f(x)) : ((\mu_1, \mu_2)\widetilde{\wedge}(\nu_1, \nu_2)) \leftarrow R_1(x) : (\frac{1}{4}, \frac{1}{2}), R_4(f(x)) : (1, 0)$. *Form the set* $S_2 = \{R_5(f(a_1)) : ((\frac{1}{4}, 1)\widetilde{\wedge}(\frac{1}{2}, 0)), R_5(f(x)) : ((\mu_1, \mu_2)\widetilde{\wedge}(\nu_1, \nu_2))\}$. *Get its mgu.*

(a) $D_3 = \{a_1, x; \frac{1}{4}, \mu_1\}$.

(b) $\theta_3\lambda_3 = \{x/a_1, \mu_1/\frac{1}{4}\}$.

(c) $S_2\theta_3\lambda_3 = \{R_5(f(a_1)) : ((\frac{1}{4}, 1)\widetilde{\wedge}(\frac{1}{2}, 0)), R_5(f(a_1)) : ((\frac{1}{4}, \mu_2)\widetilde{\wedge}(\nu_1, \nu_2))\}$.

$\vdots$

(a) $D_6 = \{\nu_2, 0\}$.

(b) $\theta_6\lambda_6 = \{x/a_1, \mu_1/\frac{1}{4}, \mu_2/\frac{1}{2}, \nu_1/1, \nu_2/0\}$

(c) $S_2\theta_6\lambda_6 = \{R_5(f(a_1)) : ((\frac{1}{4}, 1)\widetilde{\wedge}(\frac{1}{2}, 0))\}$, *which means that* $\theta_6\lambda_6$ *is the mgu of* S_2.

4. *Now we obtain that* G_3 *is* $\leftarrow (R_1(x) : (\frac{1}{4}, \frac{1}{2}), R_4(f(x)) : (1, 0))\theta_6\lambda_6$, *that is,* $\leftarrow (R_1(a_1) : (\frac{1}{4}, \frac{1}{2}), R_4(f(a_1)) : (1, 0)$. *Let* $R_1(a_1) : (\frac{1}{4}, \frac{1}{2})$ *be the selected atom. We have a clause* $R_1(a_1) : (1, 1) \leftarrow$. *Form the set* $S_3 = \{R_1(a_1) : (\frac{1}{4}, \frac{1}{2}), R_1(a_1) : (1, 1)\}$. *Get its mgu.*

   (a) $D_7 = \{\frac{1}{4}, 1\}$. S_3 *is not unifiable. But* $\langle \frac{1}{4}, \frac{1}{2} \rangle \leq_k \langle 1, 1 \rangle$, *and so we use Item 1a from Definition 4.3.1.*

5. *By item 2 of Definition 4.3.1,* G_4 *equals* $\leftarrow R_4(f(a_1)) : (1, 0)$ *with* $R_4(f(a_1)) : (1, 0)$ *as the selected atom. We have a clause* $R_4(a_1) : (\frac{1}{2}, \frac{1}{2}) \leftarrow$. *Form the set* $S_4 = \{R_4(f(a_1)) : (1, 0), R_4(a_1) : (\frac{1}{2}, \frac{1}{2})$. *Get its mgu.*

   (a) $D_8 = \{(f(a_1), (a_1); 1, \frac{1}{2}\}$. D_8 *is not unifiable. Stop.*

Consider the clause $R_4(x) : (1,0) \leftarrow R_3(f(x)) : (\mu,\nu)$. *Form the set* $S_5 = \{R_4(f(a_1)) : (1,0), R_4(x) : (1,0)\}$. *Find its mgu.*

(a) $D_9 = \{x, f(a_1)\}$.

(b) $\theta_7\lambda_7 = \{x/f(a_1); \varepsilon\}$.

(c) $S_5\theta_7\lambda_7 = \{R_4(f(a_1)) : (1,0)\}$, *which means that* $\theta_7\lambda_7$ *is the mgu of* S_5.

6. *Form* G_5, *according to item 2 of Definition 4.3.1,* G_5 *is* $\leftarrow (R_3(f(x)) : (\mu,\nu))\theta_7\lambda_7$, *that is,* $\leftarrow R_3(f(f(a_1))) : (\mu,\nu)$, *and* $R_3(f(f(a_1))) : (\mu,\nu)$ *is a selected atom. We have a clause* $R_3(a_2) : (\frac{3}{4}, 1) \leftarrow$. *Form the set* $S_6 = \{R_3(f(f(a_1))) : (\mu,\nu), R_3(a_2) : (\frac{3}{4}, 1)\}$. *Get its mgu.*

   (a) $D_{10} = \{(f(f(a_1)), a_2; \mu, \frac{3}{4}\}$. *Clearly,* $R_3(f(f(a_1)))$ *and* $R_3(a_2)$ *are not unifiable. Stop.*

   Consider the clause $R_3(x) : (\mu,\nu) \leftarrow R_2(x) : (\mu,\nu)$. *Form the set* $S_7 = \{R_3(f(f(a_1))) : (\mu,\nu), R_3(x) : (\mu,\nu)\}$. *Find its mgu.*

   (a) $D_{11} = \{(f(f(a_1))), x; \varepsilon\}$.

   (b) $\theta_8\lambda_8 = \{x/(f(f(a_1))); \varepsilon\}$.

   (c) $S_7\theta_8\lambda_8 = \{R_3(f(f(a_1))) : (\mu,\nu)\}$, *which means* $\theta_8\lambda_8$ *is the mgu for* S_7.

7. *By item 2 of Definition 4.3.1,* G_6 *equals to* $\leftarrow (R_2(x) : (\mu,\nu))\theta_8\lambda_8$, *that is,* $\leftarrow R_2(f(f(a_1))) : (\mu,\nu)$. *We have a clause* $R_2(f(x)) : (0, \frac{1}{4}) \leftarrow$. *Form the set* $S_8 = \{R_2(f(f(a_1))) : (\mu,\nu), R_2(f(x)) : (0, \frac{1}{4})\}$. *Find its mgu.*

   (a) $D_{12} = \{x, fa_1; \mu, 0\}$.

   (b) $\theta_9\lambda_9 = \{x/f(a_1); \mu/0\}$.

(c) $S_8\theta_9\lambda_9 = \{R_2(f(f(a_1))) : (0,\nu), R_2(f(f(a_1))) : (0,\frac{1}{4})\}$.

(a) $D_{13} = \{\nu, \frac{1}{4}\}$.

(b) $\theta_{10}\lambda_{10} = \{\nu/\frac{1}{4}\}$.

(c) $S_8\theta_{10}\lambda_{10} = \{R_2(f(f(a_1))) : (0,\frac{1}{4})\}$, *which means that* $\theta_{10}\lambda_{10}$ *is the mgu for* S_8.

8. *Now we form the goal* $G_7 = \square$, *using item 2 of Definition 4.3.1,.*

Thus, we conclude that we obtained an SLD-refutation of $P \cup \{G_0\}$ *of length* 7, *with computed answer* $\theta_1\lambda_1\theta_2\lambda_2\theta_6\lambda_6\theta_7\lambda_7\theta_8\ \lambda_8\theta_{10}\lambda_{10}$, *which is restricted to the variables of* G_0 *(in our case, the computed answer is the identity substitution because* G *didn't contain variables). Moreover, we conclude that* $R_3(f(a_1)) : (0,\frac{1}{4}), R_5(f(a_1)) : ((\frac{1}{4},1)\widetilde{\wedge}(\frac{1}{2},0))$ *are in the success set of* P. *As can be seen from Example 4.2.3, these formulae are contained in the least fixed point of the* $\mathcal{T}_P$*-operator applied to* P.

The previous example gave a tedious illustration of how unification and SLD-resolution algorithms work. In the next example, we will save space by omitting the details of the unification process, and at each step of the derivation, mention only the final mgu.

Example 4.3.2. *Let* P *be the program from Example 3.5.3 and let* G_0 *be the goal* $\leftarrow$ *Cancel*$(1, Monday) : (\mu,\nu)$, *that is, we want to know the probability of cancelling flight number* 1 *on Monday.*

1. *We have a clause* $Cancel(y,x) : (1,0) \leftarrow Delay(y,x) : (\frac{3}{4},\frac{1}{2})$ *in* P. *Form the set* $S = \{Cancel(1, Monday) : (\mu,\nu), Cancel(y,x) : (1,0)\}$, *find its disagreement set, and apply item 1a from Definition 4.3.1 to get its* mgu*:* $\theta_0\lambda_0 = \{y/1, x/Monday, \mu/1, \nu/0\}$.

2. *Now* G_1 *is* $\leftarrow Delay(y,x) : (\frac{3}{4},\frac{1}{2})\theta_0\lambda_0 = Delay(1, Monday) : (\frac{3}{4},\frac{1}{2})$. *We have a clause* $Delay(y,x) : ((\frac{3}{4},\frac{1}{2}) \vee (\frac{1}{2},\frac{1}{2})) \leftarrow Storm(x) : (\frac{3}{4},\frac{3}{4})$, $Storm(Tuesday) : (\frac{1}{2},\frac{1}{2})$, *whose*

head contains annotations which are unifiable with $(\frac{3}{4}, \frac{1}{2})$*. This means that item 1a from Definition 4.3.1 can be applied here.*

3. $G_2 = \leftarrow (Storm(Monday) : (\frac{3}{4}, \frac{3}{4}), Storm(\,Tuesday) : (\frac{1}{2}, \frac{1}{2}))\theta_1\lambda_1 = \leftarrow Storm(Monday) : (\frac{3}{4}, \frac{1}{2}), Storm(\,Tuesday) : (\frac{1}{2}, \frac{1}{2})$.

   Let $Storm(Monday) : (\frac{3}{4}, \frac{3}{4})$ *be the selected atom. We see that it is not unifiable with any input clause so we apply item 1b from Definition 4.3.1. So, we find the mgu* θ_2 *for* $Storm(x) : (\frac{1}{2}, \frac{3}{4})$, $Storm(Monday) : (\frac{3}{4}, \frac{1}{2})$ *and* $Storm(Monday) : (\frac{3}{4}, \frac{3}{4})$ *such that* $\theta_2 = \{x/Monday\}$, *and* $(\frac{3}{4}, \frac{3}{4}) \leq_k (\frac{1}{2}, \frac{3}{4}) \oplus (\frac{3}{4}, \frac{1}{2})$*. So, we can form the next goal according to item 3 from Definition 4.3.1.*

4. *The goal* $G_3 = \leftarrow Storm(\,Tuesday) : (\frac{1}{2}, \frac{1}{2})$*. Now* $Storm(\,Tuesday) : (\frac{1}{2}, \frac{1}{2})$ *is a selected atom and, choosing the input clause* $Storm(x) : (\frac{1}{2}, \frac{3}{4}) \leftarrow$*, we apply item 1a from Definition 4.3.1: the* mgu *for* $Storm(\,Tuesday)$ *and* $Storm(x)$ *is* $\theta_2 = x/\,Tuesday$*, and* $(\frac{1}{2}, \frac{1}{2}) \leq_k (\frac{1}{2}, \frac{3}{4})$.

5. *Use item 2 and form the last goal* $G_4 = \square$.

Thus, we conclude that we have obtained an SLD-refutation of $P \cup \{G_0\}$ *of length* 4 *with computed answer* $\theta_0\lambda_0\theta_1\lambda_1\theta_2\lambda_2$ $(\lambda_2 = \lambda_1 = \varepsilon)$*, which is restricted to the variables of* G_0 *(in our case, the computed answer is* $\{\mu/1, \nu/0\}$*, that is, the flight number* 1 *will definitely be cancelled on Monday). Moreover, we conclude that* $Cancel(1, Monday) : (1, 0)$ *is in the success set of* P*. As can be seen from Example 4.2.5, this formula is contained in the least fixed point of the* $\mathcal{T}_P$ *operator applied to* P.

Theorem 4.3.1 (Soundness of SLD-resolution for BAPs)**.** *Every computed answer for* $P \cup \{G\}$ *is a correct answer for* $P \cup \{G\}$.

Proof. Let G be $\leftarrow B_1 : (\tau_1), \ldots, B_k : (\tau_k)$ and $\theta_1\lambda_1, \ldots, \theta_n\lambda_n$ be the sequence of *mgu*s used in a refutation of $P \cup \{G\}$. We have to show that $\Pi(B_1(\tau_1) \otimes \ldots \otimes B_k : (\tau_k))\theta_1\lambda_1 \ldots \theta_n\lambda_n$ is a logical consequence of P.

We prove this by induction on the length n of the refutation.

Basis step Suppose first that $n = 1$. This means that G is a goal of the form $\leftarrow B_1 : (\tau_1)$, and one of the following conditions holds:

1. P has a unit program clause of the form $A_1 : (\tau_1^*) \leftarrow$ and $B_1\theta = A_1\theta$ and

   (a) either $(\tau_1)\lambda = (\tau_1^*)\lambda_1$,

   (b) or $(\tau_1)\lambda$ and $(\tau_1^*)\lambda$ receive constant values and $(\tau_1^*)\lambda \leq_k (\tau_1)\lambda$.

2. According to the definition of refutation, we have the third case in which P has clauses $A_1 : (\tau_1^*) \leftarrow$, ..., $A_l : (\tau_l^*) \leftarrow$ such that $B_1\theta = A_1\theta = \ldots = A_l\theta$, and

   (a) either $(\tau_1)\lambda = (\tau_1^*)\lambda = \ldots = (\tau_l^*)\lambda$,

   (b) or $(\tau_1)\lambda$ and $(\tau_1^*)\lambda, \ldots, (\tau_l^*)\lambda$ are constants, and $(\tau_1)\lambda \leq_k ((\tau_1^*)\lambda \oplus \ldots \oplus (\tau_l^*))\lambda$.

3. $G = \leftarrow A_1 : (0, 0)$.

Suppose 1a holds. Since $(A_1 : (\tau_1^*))\theta\lambda$ is an instance of a unit clause in P, we conclude that $(\Pi(B_1 : (\tau_1)))\theta\lambda$ is a logical consequence of $P \cup \{G\}$.

Suppose 1b holds. Since $A_1\theta = B_1\theta$ and $(\tau_1)\lambda \leq_k (\tau_1^*)\lambda$, $A_1 : (\tau_1^*)\theta\lambda$ is an instance of a unit clause in P and, using Proposition 3.4.1, item 1, we conclude that $(\Pi(B_1 : (\tau_1)))\theta\lambda$ is a logical consequence of P.

Suppose 2 holds. The case 2a can be proved analogously to the proof of 1a. Consider the case 2b. Since all of $(A_1 : (\tau_1^*) \leftarrow)\theta\lambda, \ldots, (A_l : (\tau_l^*) \leftarrow)\theta\lambda$ are instances of unit clauses in P, all these formulae are logical consequences of P. But then, using the fact that

$A_1^*\theta = \ldots = A_l^*\theta$ and Proposition 4.2.3, we conclude that $A((\tau_1^*) \oplus \ldots \oplus (\tau_l^*))$ is a logical consequence of P. Now, using Proposition 3.4.1, item 1 and the fact that $B_1\theta = A_1\theta$, we have that $(\Pi(B_1 : (\tau_1)))\theta\lambda$ is a logical consequence of P.

Suppose 3 holds. According to Proposition 3.4.1, item 11, $\Pi(A_1 : (0, 0))$ is a logical consequence of P.

Inductive step. Suppose that the result holds for computed answers which come from refutations of length $n - 1$. Suppose $\theta_1\lambda_1, \ldots, \theta_n\lambda_n$ is a sequence of *mgu*s used in a refutation of $P \cup \{G\}$ of length n. Suppose further that the first refutation step in the refutation of length n was made in accordance with item 1a in Definition 4.3.1, and let $A_1 : (\tau_1^*) \leftarrow \text{body}_1$ be the first input clause, such that $A_1 : (\tau_1^*)\theta_1 \ldots \theta_n = B_m : (\tau_m)\theta_1 \ldots \theta_n$ for some $B_m : (\tau_m)$ in G. By the induction hypothesis, $\Pi(B_1 : (\tau_1) \otimes \ldots \otimes B_{m-1} : (\tau_{m-1}) \otimes \text{body}_1 \otimes B_{m+1} : (\tau_{m+1}) \otimes \ldots \otimes B_k : (\tau_k))\theta_1\lambda_1 \ldots \theta_n\lambda_n$ is a logical consequence of P. (This is because only $n - 1$ steps are needed to obtain a refutation for the latter formula.) But then $(\text{body}_1)\theta_1\lambda_1 \ldots \theta_n\lambda_n$ is a logical consequence of P. This means that $A_1 : (\tau_1^*)\theta_1\lambda_1 \ldots \theta_n\lambda_n$ is a logical consequence of P. Additionally, we use the fact that $(\tau_m)\lambda_1 \ldots \lambda_n \leq_k (\tau_1^*)\lambda_1 \ldots \lambda_n$ and apply Proposition 3.4.1, item 1 to conclude that $B_m : (\tau_m)\theta_1\lambda_1 \ldots \theta_n\lambda_n$ is a logical consequence of P.

Suppose the first refutation step in the refutation of length n was taken in accordance with item 1b in Definition 4.3.1. Consider input clauses $A_1 : (\tau_1^*) \leftarrow \text{body}_1, \ldots, A_l : (\tau_l^*) \leftarrow \text{body}_l$; and selected atom $B_m : (\tau_m)$ of G at this step of the refutation. By the induction hypothesis, $\Pi(B_1 : (\tau_1) \otimes \ldots \otimes B_{m-1} : (\tau_{m-1}) \otimes (\text{body}_1) \otimes \ldots \otimes (\text{body}_l) \otimes B_{m+1} : (\tau_{m+1}) \otimes \ldots \otimes B_k : (\tau_k))\theta_1\lambda_1 \ldots \theta_n\lambda_n$ is a logical consequence of P. (This is because only $n - 1$ steps are needed to obtain a refutation for the latter formula.) Consequently, $(\text{body}_1), \ldots, (\text{body}_l))\theta_1\lambda_1 \ldots \theta_n\lambda_n$ is a logical consequence of P, which means

that $(A_1 : (\tau^*))\theta_1\lambda_1 \ldots \theta_n\lambda_n, \ \ldots, A_l : (\tau_l^*))\theta_1\lambda_1 \ldots \theta_n\lambda_n$ are logical consequences of P. But since $A_1\theta_1 \ldots \theta_n = \ldots = A_l\theta_1 \ldots \theta_n = B_m\theta_1 \ldots \theta_n$, using Proposition 4.2.3, we see that $(A_1 : ((\tau_1^*) \oplus \ldots \oplus (\tau_l^*)))\theta_1 \ldots \theta_n$ is a logical consequence of P. Moreover, according to Definition 4.3.1, item 1b that we have taken as assumption, we have $(\tau_m)\lambda_1 \ldots \lambda_n \leq_k (\tau_1^*)\lambda_1 \ldots \lambda_n, \oplus \ldots \oplus (\tau_l^*)\lambda_1 \ldots \lambda_n$. Now, using Proposition 3.4.1, item 1, we conclude that $(B_m : (\tau_m))\theta_1\lambda_1 \ldots \theta_n\lambda_n$ is a logical consequence of P.

We can apply the same arguments for the rest of the atoms $(B_1 : (\tau_1), \ldots, B_k : (\tau_k))$ from G. Thus, $\Pi((B_1 : (\tau_1), \ldots, B_k : (\tau_k)))\theta_1\lambda_1 \ldots \theta_n\lambda_n$ is a logical consequence of P.

□

Corollary 4.3.1. *The success set of P is contained in its least annotation Herbrand model.*

Proof. Let $A : (\tau) \in B_P$ and suppose that $P \cup \{\leftarrow A : (\tau)\}$ has a refutation. By Theorem 4.3.1, $A : (\tau)$ is a logical consequence of P. Thus $A : (\tau)$ is in the least annotation Herbrand model for P. □

So, we conclude that the SLD-resolution for BAPs introduced in Definition 4.3.1 is sound with respect to the Herbrand models computed by $\mathcal{T}_P$.

It remains to show that our SLD-resolution is complete.

4.3.2 Completeness of SLD-resolution for BAPs

Some proofs in this section require the use of lemmas which are straightforward generalisations of the so-called *mgu lemma* [138] and the *lifting lemma* to the case when unification is allowed with respect to annotation variables as well as individual variables. We omit their proofs because they remain essentially the same as in the classical, annotation-free case. These proofs can be found, for example, in [138].

Lemma 4.3.1 (Mgu lemma)**.** *Let $P \cup \{G\}$ have an unrestricted SLD-refutation. Then $P \cup \{G\}$ has an SLD-refutation of the same length such that if $\theta_1, \ldots, \theta_n, \lambda_1, \ldots, \lambda_n$ are the unifiers from the unrestricted SLD-refutation of $P \cup \{G\}$, and $\theta'_1, \ldots, \theta'_n, \lambda'_1, \ldots, \lambda'_n$ the mgus from the SLD-refutation of $P \cup \{G\}$, then there exist substitutions γ and ζ such that $\theta_1 \ldots \theta_n = \theta'_1 \ldots \theta'_n \gamma$ and $\lambda_1 \ldots \lambda_n = \lambda'_1 \ldots \lambda'_n \zeta$.*

Proof. The proof is essentially the same as in [138], and is given separately for individual and annotation substitutions. □

Lemma 4.3.2 (Lifting lemma)**.** *Suppose there exists an SLD-refutation of $P \cup \{G\theta\lambda\}$. Then there exists an SLD-refutation of $P \cup \{G\}$ such that, if $\theta_1, \ldots, \theta_n, \lambda_1, \ldots, \lambda_n$ are the mgus from the SLD-refutation of $P \cup \{G\theta\lambda\}$, and $\theta'_1, \ldots, \theta'_n, \lambda'_1, \ldots, \lambda'_n$ are the mgus from the SLD-refutation of $P \cup \{G\}$, then there exist substitutions γ, ζ such that $\theta\theta_1 \ldots \theta_n = \theta'_1 \ldots \theta'_n \gamma$ and $\lambda\lambda_1 \ldots \lambda_n = \lambda'_1 \ldots \lambda'_n \zeta$.*

Proof. The proof is essentially the same as in [138]. □

The following completeness theorem relates the computations by the semantic operator $\mathcal{T}_P$ and SLD-resolution. It extends the corresponding theorem for two-valued propositional logic programming which is due to Apt and Van Emden [6]. The many-valued interpretation adds substantial complication to the proof of the corresponding completeness theorem for BAPs. Namely, we do not simply assume, as in the classical case, that $A : (\tau) \in \mathcal{T}_P \uparrow n$ whenever $A : (\tau) \leftarrow B_1 : (\tau_1), \ldots, B_n : (\tau_n)$ and $B_1 : (\tau_1), \ldots, B_k : (\tau_k) \in \mathcal{T}_P \uparrow (n-1)$. According to the Definition of $\mathcal{T}_P$, we have to give an account of the cases when $B_1 : (\sigma_1), \ldots, B_k : (\sigma_k) \in \mathcal{T}_P \uparrow (n-1)$, where each $\tau_i \leq_k \sigma_i$, $i = 1, \ldots, k$. Item 2 from Definition 4.2.6 of $\mathcal{T}_P$ needs to be taken into consideration as well.

Theorem 4.3.2. *The success set of P is equal to its least annotation Herbrand model.*

Proof. By Corollary 4.3.1, it suffices to show that the least Herbrand model for P is contained in its success set.

Suppose that $A : (\tau)$ is in the least annotation Herbrand model for P. By Theorem 4.2.4, $A : (\tau) \in \mathcal{T}_P \uparrow n$, for some $n \in \omega$. We claim that $A : (\tau) \in \mathcal{T}_P \uparrow n$ implies that $P \cup \{\leftarrow A : (\tau)\}$ has a refutation, and hence that $A : (\tau)$ is in the success set; we prove this claim by induction on n.

Basis step. $n = 1$. Then $A : (\tau) \in \mathcal{T}_P \uparrow 1$, which means that either $A : (\tau) \leftarrow$ is a strictly ground instance of a clause in P or $(\tau) = (0, 0)$. And in both cases $P \cup \{\leftarrow A : (\tau)\}$ has a refutation (see items 1a and 4 in Definition 4.3.1).

Inductive step. Suppose the claim holds for $n - 1$. Let $A : (\tau) \in \mathcal{T}_P \uparrow n$. By the definition of $\mathcal{T}_P$, one of the following holds:

1. there exists a strictly ground instance of a clause $B : (\tau') \leftarrow B_1 : (\tau_1), \ldots, B_k : (\tau_k)$ such that $A = B\theta$, $(\tau) = (\tau')\lambda$, and $B_1 : (\tau_1')\theta, \ldots, B_k : (\tau_k')\theta \in \mathcal{T}_P \uparrow (n-1)$ with $\tau_1', \ldots, \tau_k'$ such that:

$$(\tau_i)\lambda \leq_k (\tau_i'), \text{ for } i \in \{1, \ldots, k\}.$$

2. There are strictly ground atoms $A : (\tau_1), \ldots, A : (\tau_k) \in \mathcal{T}_P \uparrow (n-1)$ such that $(\tau) \leq_k ((\tau_1) \oplus \ldots \oplus (\tau_k))$.

Suppose 1 holds. By the induction hypothesis, each $P \cup \{\leftarrow B_i : (\tau_i')\theta\}$ has a refutation, for $i \in \{1, \ldots, k\}$. We want to show that then $P \cup \{\leftarrow B_i : (\tau_i)\theta\lambda\}$ has a refutation, for $i \in \{1, \ldots, k\}$.

Consider the refutation of $G_0 = \{\leftarrow B_i : (\tau_i')\theta\}$. According to Definition 4.3.1, there are two ways in which $\{\leftarrow B_i : (\tau_i')\theta\}$ can be derived.

Case 1.

There is a clause $C : (\tau^*) \leftarrow \text{body}$ in P such that $B_i\theta = C\theta$, and $(\tau_i') \leq_k (\tau^*)\lambda$. Taking into account that $(\tau_i)\lambda \leq_k (\tau_i')$, by transitivity of $\leq_k$, we conclude that $(\tau_i)\lambda \leq_k (\tau^*)\lambda$. But then, by Definition 4.3.1, item 1a, the goal $\leftarrow B_i(\tau_i)\theta\lambda$ will receive a refutation.

Case 2.

There are clauses $C_1 : (\tau_1^*) \leftarrow \text{body}_1, \ \ldots, \ C_m : (\tau_1^*) \leftarrow \text{body}_m$ in P such that $B_i\theta = C_1\theta = \ldots = C_m\theta$, and $\tau_i' \leq_k (\tau_1^*\lambda \oplus \ldots \oplus \tau_m^*\lambda)$. But then, because $(\tau_i)\lambda \leq_k (\tau_i')$, we have $\tau_i\lambda \leq_k (\tau_1^*\lambda \oplus \ldots \oplus \tau_m^*\lambda)$. And, by Definition 4.3.1, item 1b the goal $\leftarrow (B_i : \tau_i)\theta\lambda$ will receive refutation as well.

Suppose 2 holds, that is, there are strictly ground atoms $A : (\tau_1), \ldots, A : (\tau_k) \in \mathcal{T}_P \uparrow (n-1)$ such that $(\tau) \leq_k ((\tau_1) \oplus \ldots \oplus (\tau_k))$. We want to show that then $A : (\tau)$ has a refutation.

Using the induction hypothesis, each of $P \cup \{\leftarrow A : (\tau_1)\}, \ldots, P \cup \{\leftarrow A : (\tau_k)\}$ has a refutation. This means that, according to Definition 4.3.1, items 1a and 1b, there are clauses $A : (\tau_1^*) \leftarrow \text{body}_1^*, \ldots, A : (\tau_n^*) \leftarrow \text{body}_n^*$, such that for each $A : (\tau_i)$ one of the following holds:

- $(\tau_i)\lambda \leq_k (\tau_j^*)\lambda$. In this case, $P \cup \{\leftarrow \text{body}_j^*\}$ has a refutation.

- $(\tau_i)\lambda \leq_k ((\tau_j^*) \oplus \ldots \oplus (\tau_l^*))$. Then $P \cup \{\leftarrow \text{body}_j^*, \ldots, \text{body}_l^*\}$ has a refutation, for some $j, l \in \{1, \ldots, n\}$.

Combining all these refutations for $i \in \{1, \ldots, n\}$, we obtain a refutation of $P \cup \{\leftarrow (\text{body}_1^*, \ldots, \text{body}_n^*)\theta\lambda\}$. But then, according to Definition 4.3.1, items 1a and 1b, there is

a refutation for $P \cup \{\leftarrow A : (\tau')\theta\lambda\}$, where $\tau'\lambda$ is some annotation such that

$$(\tau')\lambda \leq_k ((\tau_1^*)\lambda \oplus \ldots \oplus (\tau_n^*)\lambda) \quad (*)$$

for each $(\tau')\lambda \leq_k (\tau_j^*)\lambda$ (in case of $\bullet$) or for each $(\tau')\lambda \leq_k ((\tau_j^*) \oplus \ldots \oplus (\tau_l^*))$ (in case of $\bullet\bullet$), for $j, l \in \{1, \ldots, n\}$. Now, having that either each $(\tau_i)\lambda \leq_k (\tau_j^*)\lambda$ or each $(\tau_i)\lambda \leq_k ((\tau_j^*) \oplus \ldots \oplus (\tau_l^*))$, for $i \in \{1, \ldots, k\}$ and $j, l \in \{1, \ldots, n\}$, we conclude, by monotonicity of $\oplus$, that

$$((\tau_1)\lambda \oplus \ldots \oplus (\tau_k)\lambda) \leq_k ((\tau_1^*)\lambda \oplus \ldots \oplus (\tau_n^*)\lambda).$$

But then, by Definition 4.3.1, item 1a, the condition $(*)$ implies that, for each annotation constant $(\tau^\Diamond)$ such that $(\tau^\Diamond) \leq_k ((\tau_1^*) \oplus \ldots \oplus (\tau_n^*))\lambda$, there exists a refutation for $P \cup \{\leftarrow A : (\tau^\Diamond)\theta\lambda\}$. This holds in particular for all the $\tau^\Diamond$ such that $(\tau^\Diamond) \leq_k ((\tau_1) \oplus \ldots \oplus (\tau_k))\lambda$. According to item 2, among such $(\tau^\Diamond)$ will be $(\tau^\Diamond) = (\tau)\lambda$. Thus, we conclude that $P \cup \{\leftarrow A : (\tau)\theta\lambda\}$ has an unrestricted refutation. Finally, we apply the *mgu* lemma to obtain a refutation for $P \cup \{\leftarrow A : (\tau)\}$. □

Next, we need to obtain completeness with respect to correct answers. As in classical two-valued logic programming, it is impossible to prove the exact converse of Theorem 4.3.1. However, we can extend the classical result, that every correct answer is an instance of a computed answer, to the case of BAPs. The next lemma and associated theorem are straightforward; recall that we allow refutation to work over individual and annotation variables independently.

Lemma 4.3.3. *Let $\Pi(A : (\tau))$ be a logical consequence of P. Then there exists an SLD-refutation for $P \cup \{\leftarrow A : (\tau)\}$ with the identity substitution as a computed answer.*

Proof. Suppose $A : (\tau)$ has individual variables $x_1, \ldots, x_n$ and annotation variables $\mu_1, \nu_1, \ldots, \mu_m, \nu_m$. Let $a_1, \ldots, a_n$ be distinct individual constants and $\alpha_1, \ldots, \alpha_m, \beta_1, \ldots, \beta_l$

be distinct annotation constants, which do not appear in P or $A : (\tau)$ and let θ be the substitution $\{x_1/a_1, \ldots, x_n/a_n\}$, and λ be substitution $\{\mu_1/\alpha_1, \ldots, \mu_m/\alpha_m, \nu_1/\beta_1, \ldots \nu_l/\beta_l\}$. Then $A : (\tau)\theta\lambda$ is a logical consequence of P. Since $A : (\tau)\theta\lambda$ is ground, Theorem 4.3.2 shows that $P \cup \{A : (\tau)\theta\lambda\}$ has a refutation. Since all the a_i, α_j, β_k do not appear in P or $A : (\tau)$, we replace all the a_i, α_j, β_k by x_i, μ_j and ν_k respectively, ($i \in \{1, \ldots, n\}$, $j \in \{1, \ldots, m\}$, $k \in \{1, \ldots, l\}$) in this refutation. And thus we obtain a refutation of $P \cup \{\leftarrow A : (\tau)\}$ with the identity substitution as the computed answer. □

The next completeness result is a generalisation of the analogous theorem of Clark [24].

Theorem 4.3.3. *For every correct answer $\theta\lambda$ for $P \cup \{G\}$, there exist a computed answer $\theta^*\lambda^*$ for $P \cup \{G\}$ and substitutions φ, ψ such that $\theta = \theta^*\varphi$ and $\lambda = \lambda^*\psi$.*

Proof. Suppose G is the goal $\leftarrow A_1 : (\tau_1), \ldots, A_k : (\tau_k)$. Since $\theta\lambda$ is correct, $\Pi((A_1 : (\tau_1), \ldots, A_k : (\tau_k))\theta\lambda)$ is a logical consequence of P. By Lemma 4.3.3, there exist a refutation of $P \cup \{\leftarrow A_i : (\tau_i)\theta\lambda\}$ such that the computed answer is the identity, for all $i \in \{1, \ldots, k\}$. We can combine these refutations into a refutation of $P \cup \{G\theta\lambda\}$ such that the computed answer is the identity.

Suppose the sequence of mgus of the refutation of $P \cup \{G\theta\lambda\}$ is $\theta_1, \ldots, \theta_n, \lambda_1, \ldots, \lambda_m$. Then $G\theta\theta_1 \ldots \theta_n\lambda\lambda_1 \ldots \lambda_m = G\theta\lambda$. By the lifting lemma, there exist a refutation of $P \cup \{G\}$ with mgus $\theta'_1 \ldots \theta'_n, \lambda'_1 \ldots \lambda'_m$ such that $\theta\theta_1 \ldots \theta_n = \theta'_1 \ldots \theta'_n\varphi'$ and $\lambda\lambda_1 \ldots \lambda_m = \lambda'_1 \ldots \lambda'_m\psi'$ for some substitutions φ', ψ'. Let θ^* be $\theta'_1 \ldots \theta'_n$ and λ^* be $\lambda'_1 \ldots \lambda'_m$ restricted to the individual and annotation variables of G. Then $\theta = \theta^*\varphi$ and $\lambda = \lambda^*\psi$. □

We have defined an SLD-resolution for BAPs and proved that it is sound and complete relative to the declarative semantics of BAPs.

We will conclude this chapter with a section showing the relationship of BAPs to bilattice-based annotation-free and implication-based bilattice logic programs.

4.4 Relation to other Kinds of Bilattice-Based Logic Programs

In Chapter 2 we outlined the following three approaches to many-valued logic programming: annotation-free, implication-based and annotated approaches.

The annotated approach was shown to be the most expressive of all in the case of *lattice-based* interpretations, see [108]. The BAPs we have introduced here are annotated logic programs, and we need to discuss their relations with annotation-free and implication-based *bilattice-based* logic programs.

We consider two alternative approaches to bilattice-based logic programming:

- Fitting's language (see, for example, [49]) does not contain any annotations but contains all possible connectives and quantifiers from Definition 3.3.1.

- The implication-based logic programs contain annotated implication arrows instead of having all the literals annotated. These programs may be seen as generalisations of Van Emden's programs [203], to the bilattice-based case.

The results of this section serve not only for the purpose of comparing the expressiveness of BAPs and other bilattice-based logic programs, but they also help to evaluate the two semantic operators we have introduced in Definitions 4.2.5, 4.2.6. Namely, we show here the relations of the semantic operators of Fitting and Van Emden to $\widehat{\mathcal{T}_P}$ and $\mathcal{T}_P$, and in particular, we prove that the semantic operators of Fitting and Van Emden can be

simulated by means of $\widehat{\mathcal{T}_P}$, and thus do not guarantee the computation of all the logical consequences of a given program.

In this section we revise, formalise, and further develop some ideas of Kifer and Subrahmanian [108] on the relation of General Annotated Programs (GAP's) to annotation-free and implication-based logic programs. For example, we extend their results concerning the translation of Fitting's connectives into GAP's by allowing Σ in BAPs, which gives us a translation of the k-existential fragment of Fitting's logic programs into BAPs. We also show that despite the conjecture of [108], it is not always the case that the least fixed point of the $T_P^{\blacklozenge}$ operator of Van Emden from Definition 2.0.1 is finite, and that is why we give a novel proof of the fact that the semantic operator for the implication-based logic programs can be simulated by $\mathcal{T}_P$.

Our first target is to prove that Fitting's bilattice-based (annotation-free) logic programs [49, 50] can be fully translated into BAPs, and the semantic operator of Fitting can be simulated by $\widehat{\mathcal{T}_P}$. And we start by giving formal definitions of these programs and the semantic operator, as follows.

Definition 4.4.1. *We call a set of clauses of the form $A \leftarrow F$, where all individual variables appearing in F are quantified by Σ and F is a formula consisting of first-order literals connected by $\vee$, $\wedge$, $\oplus$ and $\otimes$ a* Fitting bilattice-based (k-existential) logic program.

Propositional symbols denoting elements of the underlying bilattice are allowed in the language, that is, Fitting's programs can contain clauses of the form $A \leftarrow (\alpha, \beta)$, where $(\alpha, \beta) \in \mathbf{B}$. Clauses of the form $A \leftarrow$ with an empty body are thought of as being completed as follows: $A \leftarrow (1, 1)$.

Definition 4.4.2. *We call the set of clauses of the form*

$$A \leftarrow \Sigma x_1, \ldots, \Sigma x_k(L_1 \otimes \ldots \otimes L_n),$$

where each L_i *is a literal and* $x_1, \ldots, x_k$ *are all the individual variables appearing in* $(L_1 \otimes \ldots \otimes L_n)$ *a* Horn-clause Fitting bilattice-based logic program.

In different papers Fitting worked with either full fragments of the bilattice-based logic programs, or with the Horn-clause fragment of them. This is why, although Fitting did not put this terminological distinction between the two classes of bilattice-based logic programs, we distinguish them here.

Example 4.4.1. *Consider the following simple example of a Horn-clause Fitting program* P*:*

$$\begin{aligned} B &\leftarrow (1,0) \\ B &\leftarrow (0,1) \\ A &\leftarrow B \end{aligned}$$

All Fitting's formulae receive interpretation from the set of elements of a chosen bilattice $\mathbf{B}$, using the interpretation function $\mathcal{I}$ from Definition 3.4.1. See [49, 50] for further explanations and definitions. The immediate consequence operator for Fitting's logic programs is defined as follows:

Definition 4.4.3. *[49] Let* $\mathcal{I}$ *be an interpretation of the annotation-free language of Fitting in* $\mathbf{B}$*,* P *be a Fitting program, and* $A \in B_P$*. Then*

$$\Phi_P(\mathcal{I})(A) = \begin{cases} \mathcal{I}(F) & \text{if } A \leftarrow F \in \textit{ground}(P) \\ A & \text{if } A \in \mathbf{B} \end{cases}$$

The letter F *in this definition denotes any formula admissible in the bodies of clauses, and its form will depend on which fragment of Fitting's programs - full or Horn clause - we chose to consider.*

The main difference between $\mathcal{T}_P$ defined for BAPs and Φ_P is that the former works with annotated formulae, while the latter works directly with interpretations of annotation-free formulae.

The transfinite sequence of Definition 1.5.6 has to be reformulated as follows.

Definition 4.4.4.

$\Phi_P \uparrow 0 = \bot$, *where* $\bot$ *is the least element of* $\mathbf{B}$ *with respect to the* k*-ordering,*

$\Phi_P \uparrow \alpha = \Phi_P(\Phi_P(\alpha - 1))$, *if* α *is a successor ordinal,*

$\Phi_P \uparrow \alpha = \bigvee\{\Phi_P \uparrow \beta : \beta < \alpha\}$, *if* α *is a limit ordinal.*

Fitting established in [49] that Φ_P is continuous in the case of k-existential bilattice-based logic programs P, see also Theorem 4.2.2 herein. Therefore, the least fixed point of Φ_P will always be reached in at most ω steps, that is, $\mathrm{lfp}(\Phi_P) = \Phi_P \uparrow \omega$ characterises models of Fitting's k-existential bilattice-based logic programs.

Example 4.4.2. *Consider the logic program from Example 4.4.1. The least fixed point of* Φ_P *gives us* $\{\mathcal{I}(B) = \langle 1, 0\rangle, \mathcal{I}(B) = \langle 0, 1\rangle, \mathcal{I}(A) = \langle 1, 0\rangle, \mathcal{I}(A) = \langle 0, 1\rangle\}$. *(We can think that valuation* $\mathcal{I}(B) = \langle 1, 0\rangle$ *subsumes* $\mathcal{I}(B) = \langle 0, 0\rangle$*; similarly for A.)*

Definition 4.4.5. *We say that the BAP clause*

$$A : ((\mu_1, \nu_1)\widetilde{\otimes} \ldots \widetilde{\otimes}(\nu_n, \mu_n)) \leftarrow \Sigma x_1, \ldots, \Sigma x_k(L_1 : (\mu_1, \nu_1), \ldots, L_n : (\mu_n, \nu_n)),$$

where each (μ_i, ν_i) *is an annotation variable, translates the Horn clause*

$$A \leftarrow \Sigma x_1, \ldots, \Sigma x_k(L_1 \otimes \ldots \otimes L_n)$$

from the Fitting logic program.

Whenever bilattice elements are used as propositional symbols in Fitting's logic programs, we translate them using annotation constants. Thus, $A \leftarrow L_1, \ldots, (\alpha, \beta), \ldots, L_n$ *will be translated by*

$$A : ((\mu_1, \nu_1) \otimes \ldots \otimes (\alpha, \beta) \otimes \ldots \otimes (\mu_n, \nu_n)) \leftarrow L_1 : (\mu_1, \nu_1), \ldots, L_n : (\mu_n, \nu_n).$$

In accordance with the notational convention used in classical logic programming, we will sometimes omit writing $\Sigma x_1, \ldots, \Sigma x_k$ in the bodies of clauses.

Example 4.4.3. *Consider the annotation-free logic program* P *from Example 4.4.1. According to Definition 4.4.5, we have the following translation of this program into a BAP* P^F*:*

$$\begin{aligned} B : (1, 0) \quad &\leftarrow \\ B : (0, 1) \quad &\leftarrow \\ A : (\mu, \nu) \quad &\leftarrow \quad B : (\mu, \nu) \end{aligned}$$

Next we relate the semantic operators Φ_P and $\widehat{\mathcal{T}_P}$.

Lemma 4.4.1. *Let* P *be a Fitting Horn-clause bilattice-based program, and let* P^F *be the BAP obtained from* P *using Definition 4.4.5. Then the following holds.*

If a formula A *receives interpretation* $\langle \alpha, \beta \rangle$ *on the least fixed point of* Φ_P*, then* $A : (\alpha, \beta) \in \mathrm{lfp}(\widehat{\mathcal{T}_{P^F}})$.

If $A : (\alpha, \beta) \in \mathrm{lfp}(\widehat{\mathcal{T}_{P^F}})$*, then* $\mathcal{I}(A) = \langle \alpha', \beta' \rangle$ *in the least fixed point of* Φ_P*, where* $\langle \alpha, \beta \rangle \leq_k \langle \alpha', \beta' \rangle$.

Proof. The proof proceeds by induction on the number of iterations of Φ_P and $\widehat{\mathcal{T}_{P^F}}$. Note that throughout this proof, $\langle \alpha, \beta \rangle$ denotes an element of the bilattice, and (α, β) denotes an annotation constant.

We first prove that if a formula A receives interpretation $\langle \alpha, \beta \rangle$ in the least fixed point of Φ_P for a Fitting Horn-clause bilattice program P, then $A : (\alpha, \beta) \in \text{lfp}(\widehat{\mathcal{T}_{P^F}})$ for the BAP P^F.

Basis step. Assume $\text{lfp}(\Phi_P) = \Phi_P \uparrow 1$ and $(\Phi_P \uparrow 1)(\mathcal{I})(A) = \langle \alpha, \beta \rangle$. This means $A \leftarrow (\alpha, \beta) \in \text{ground}(P)$. But then there is a clause $A : (\alpha, \beta) \leftarrow \in \text{ground}(P^F)$, and hence $A : (\alpha, \beta) \in \widehat{\mathcal{T}_{P^F}} \uparrow 1$.

Inductive step. Suppose that whenever $(\Phi_P \uparrow k)(\mathcal{I})(B) = \langle \alpha, \beta \rangle$, then the annotated formula $B : (\alpha, \beta) \in \widehat{\mathcal{T}_{P^F}} \uparrow k$.

Consider $\Phi_P \uparrow (k+1)$. Assume $(\Phi_P \uparrow (k+1))(\mathcal{I})(A) = \langle \alpha, \beta \rangle$. This is possible only if $A \leftarrow B_1 \otimes \ldots \otimes B_m \in \text{ground}(P)$ and $(\Phi_P \uparrow k)(\mathcal{I})(B_1 \otimes \ldots \otimes B_m) = \langle \alpha, \beta \rangle$, where

$$\langle \alpha, \beta \rangle = (\langle \alpha_1, \beta_1 \rangle \otimes \ldots \otimes \langle \alpha_m, \beta_m \rangle) \quad (*);$$

for each $(\Phi_P \uparrow k)(\mathcal{I})(B_i) = (\alpha_i, \beta_i)$, for $i = 1, \ldots, m$.

Since $A \leftarrow B_1 \otimes \ldots \otimes B_m \in \text{ground}(P)$, by Definition 4.4.5, $A : ((\alpha_1, \beta_1) \otimes \ldots \otimes (\alpha_m, \beta_m)) \leftarrow B_1 : (\alpha_1, \beta_1), \ldots, B_m : (\alpha_m, \beta_m) \in \text{ground}(P^F)$. Using the induction hypothesis, we conclude that each $B_i : (\alpha_i, \beta_i) \in \widehat{\mathcal{T}_{P^F}} \uparrow k$. Hence, by Definition 4.2.5, we obtain

$$A : ((\alpha_1, \beta_1) \otimes \ldots \otimes (\alpha_m, \beta_m)) \in \widehat{\mathcal{T}_{P^F}} \uparrow (k+1)$$

and, using $(*)$, we have that

$$A : (\alpha, \beta) \in \widehat{\mathcal{T}_{P^F}} \uparrow (k+1).$$

Now we need to prove that if $A : (\alpha, \beta) \in \text{lfp}(\widehat{\mathcal{T}_{P^F}})$ for the BAP P^F obtained from P using Definition 4.4.5, then A receives interpretation $\langle \alpha', \beta' \rangle$ in the least fixed point of Φ_P, where $\langle \alpha, \beta \rangle \leq_k \langle \alpha', \beta' \rangle$.

Basis step. Let $\mathrm{lfp}(\widehat{\mathcal{T}_{P^F}})$ be obtained at the first iteration of $\widehat{\mathcal{T}_{P^F}}$, and $A : (\alpha, \beta) \in \widehat{\mathcal{T}_{P^F}} \uparrow 1$. Then there is a clause $A : (\alpha, \beta) \leftarrow \in \mathrm{ground}(P^F)$. But then, according to the definition of P^F, $A \leftarrow (\alpha, \beta) \in \mathrm{ground}(P)$. Hence, $(\Phi_P \uparrow 1)(\mathcal{I})(A) = \langle \alpha, \beta \rangle$.

Inductive step. Suppose for k we have that if $B_i : (\alpha_i, \beta_i) \in \widehat{\mathcal{T}_{P^F}} \uparrow k$, then $(\Phi_P \uparrow k)(\mathcal{I})(B_i) = \langle \alpha_i, \beta_i \rangle$, $(i \in \{1 \ldots m\})$.

Let $A : (\alpha, \beta) \in \widehat{\mathcal{T}_{P^F}} \uparrow (k{+}1)$. Then there is a clause $A : (\alpha, \beta) \leftarrow B_1 : (\alpha_1, \beta_1), \ldots, B_m : (\alpha_m, \beta_m) \in \mathrm{ground}(P^F)$, $(\alpha_i, \beta_i) \leq_k (\alpha'_i, \beta'_i)$, and $B_i : (\alpha'_i, \beta'_i) \in \widehat{\mathcal{T}_{P^F}} \uparrow k$, for $i \in \{1, \ldots, m\}$. Because *all* clauses in P^F are obtained as described in Definition 4.4.5 of P^F,

$$(\alpha, \beta) = (\alpha_1, \beta_1) \otimes \ldots \otimes (\alpha_m, \beta_m) \quad (**).$$

But then both $A : (\alpha, \beta) \leftarrow B_1 : (\alpha_1, \beta_1), \ldots, B_m : (\alpha_m, \beta_m)$ and $A : (\alpha', \beta') \leftarrow B_1 : (\alpha'_1, \beta'_1), \ldots, B_m : (\alpha'_m, \beta'_m)$, $((\alpha', \beta') = (\alpha'_1, \beta'_1) \otimes \ldots \otimes (\alpha'_m, \beta'_m))$, are strictly ground instances of a clause that translates the clause $A \leftarrow B_1 \otimes \ldots \otimes B_m$. Now, using our induction hypothesis, for each B_i we have that $(\Phi_P \uparrow k)(\mathcal{I})(B_i) = \langle \alpha'_i, \beta'_i \rangle$. But then, according to the definitions of $\mathcal{I}$ and of Φ_P, $(\Phi \uparrow (k+1))(\mathcal{I})(A) = \langle \alpha'_1, \beta'_1 \rangle \otimes \ldots \otimes \langle \alpha'_m, \beta'_m \rangle = \langle \alpha', \beta' \rangle$. By monotonicity of $\oplus$, $\langle \alpha_1, \beta_1 \rangle \otimes \ldots \otimes \langle \alpha_m, \beta_m \rangle \leq_k \langle \alpha'_1, \beta'_1 \rangle \otimes \ldots \otimes \langle \alpha'_m, \beta'_m \rangle$. And, since we have $(**)$, it follows that $(\Phi \uparrow (k+1))(\mathcal{I})(A) = \langle \alpha', \beta' \rangle$, where $\langle \alpha, \beta \rangle \leq_k \langle \alpha', \beta' \rangle$. □

Example 4.4.4. *We continue to develop Examples 4.4.1, 4.4.2, 4.4.3. Given the annotated logic program P^F from Example 4.4.3, the least fixed point of $\widehat{\mathcal{T}_{P^F}}$ will compute the following set:* $\{B : (0,0), B : (1,0), B : (0,1), A : (0,0), A : (1,0), A : (0,1)\}$. *It precisely corresponds to the computations of Φ_P from Example 4.4.2 performed for the annotation-free Fitting logic program P, modulo our assumption that whenever Φ_P computes $\mathcal{I}(A) = \langle 1, 0 \rangle$, it subsumes $\mathcal{I}(A) = \langle 0, 0 \rangle$, and similarly for B.*

We can establish a stronger translation result, that is, we can show that not only Horn clause bilattice-based logic programs, but also the full fragment of k-existential bilattice-based logic programs can be translated into BAPs.

Definition 4.4.6. *Let F be an annotation-free formula of a bilattice-based language of Fitting, and let $A \leftarrow F$ be a clause of a k-existential Fitting logic program.*

We define the following process of translation of Fitting's clauses into annotated clauses.

1. *If F consists of literals $L_1, \ldots, L_n$, substitute them by annotated literals $L_1 : (\mu_1, \nu_1), \ldots, L_n : (\mu_n, \nu_n)$, where each (μ_i, ν_i) is a variable annotation. We call the resulting formula F'.*

2. *Each clause of the form $A \leftarrow F'$ should be transformed into*

$$A : (\vartheta((\mu_1, \ldots, \mu_n), (\nu_1, \ldots, \nu_n)) \leftarrow F''$$

using the following rules:

- *if $A \leftarrow F'$ is*

  $A \leftarrow L_1 : (\mu_1, \nu_1), \ldots, L_k : (\mu_k, \nu_k) \vee L_{k+1} : (\mu_{k+1}, \nu_{k+1}), \ldots, L_n : (\mu_n, \nu_n)$,

  then substitute it by

  $A : (((\mu_1, \nu_1), \ldots, (\mu_k, \nu_k))\widetilde{\vee}((\mu_{k+1}, \nu_{k+1}), \ldots (\mu_n, \nu_n)))$

  $\leftarrow L_1 : (\mu_1, \nu_1), \ldots, L_k : (\mu_k, \nu_k), L_{k+1} : (\mu_{k+1}, \nu_{k+1}), \ldots, L_n : (\mu_n, \nu_n)$;

- *if $A \leftarrow F'$ is*

  $A \leftarrow L_1 : (\mu_1, \nu_1), \ldots, L_k : (\mu_k, \nu_k) \wedge L_{k+1} : (\mu_{k+1}, \nu_{k+1}), \ldots, L_n : (\mu_n, \nu_n)$,

  then substitute it by

  $A : (((\mu_1, \nu_1), \ldots, (\mu_k, \nu_k))\widetilde{\wedge}((\mu_{k+1}, \nu_{k+1}), \ldots, (\mu_n, \nu_n)))$

  $\leftarrow L_1 : (\mu_1, \nu_1), \ldots,$

  $L_k : (\mu_k, \nu_k), L_{k+1} : (\mu_{k+1}, \nu_{k+1}), \ldots, L_n : (\mu_n, \nu_n)$;

- *if* $A \leftarrow F'$ *is*

  $A \leftarrow L_1 : (\mu_1, \nu_1), \ldots, L_k : (\mu_k, \nu_k) \oplus L_{k+1} : (\mu_{k+1}, \nu_{k+1}), \ldots, L_n : (\mu_n, \nu_n)$,

  then substitute it by

  $A : (((\mu_1, \nu_1), \ldots, (\mu_k, \nu_k))\widetilde{\oplus}((\mu_{k+1}, \nu_{k+1}), \ldots, (\mu_n, \nu_n)))$

  $\leftarrow L_1 : (\mu_1, \nu_1), \ldots, L_k : (\mu_k, \nu_k), L_{k+1} : (\mu_{k+1}, \nu_{k+1}), \ldots, L_n : (\mu_n, \nu_n)$.

Note that symbols $\vee$, $\wedge$, $\oplus$ *appearing in the bodies of the clauses denote connectives of the language, and the symbols* $\widetilde{\vee}$, $\widetilde{\wedge}$, $\widetilde{\oplus}$ *appearing in the heads of the clauses denote operations defined on the underlying bilattice. We say that the resulting BAP clause*

$$A : (\vartheta((\mu_1, \nu_1,) \ldots, (\mu_n, \nu_n)) \leftarrow F''$$

translates the Fitting (k-existential) clause $A \leftarrow F$.

It remains to show that this translation allows us to compute the logical consequences of the translated programs.

Theorem 4.4.1. *Let A be an annotation-free ground formula of Fitting k-existential logic program P, and* P^F *be the BAP translating P using Definition 4.4.6. Then the following holds.*

If A receives the value $\langle \alpha, \beta \rangle$ *in the least fixed point of* Φ_P*, then* $A : (\alpha, \beta) \in \mathrm{lfp}(\widehat{\mathcal{T}_{P^F}})$.

If $A : (\alpha, \beta) \in \mathrm{lfp}(\widehat{\mathcal{T}_{P^F}})$*, then A receives the value* $\langle \alpha', \beta' \rangle$ *in the least fixed point of* Φ_P*, where* $\langle \alpha, \beta \rangle \leq_k \langle \alpha', \beta' \rangle$.

Proof. The proof is given by induction on the number of iterations of the semantic operators, and is similar to the proof of Lemma 4.4.1, but includes similar proofs for the three additional cases when $\vee, \wedge, \oplus$ are used instead of $\otimes$. We use Definition 4.4.6 here in addition to Definition 4.4.5. □

This theorem concludes the discussion of the relationship between BAPs and the annotation-free bilattice-based logic programs of Fitting.

Next we establish a translation of implication-based logic programs into BAPs. First, we extend Van Emden's implication-based approach to the case when a bilattice instead of the unit interval of reals is taken as the underlying structure of the logic program. Then we show that the resulting extension can be translated into a BAP.

For the definitions of the implication-based logic programs of Van Emden [203], examples and further motivation, see Chapter 2. Here we consider only bilattice-based implication-based logic programs.

We also refer to the paper of Lakshmanan and Sadri [134] for a detailed discussion of normal implication-based logic programs that extend Van Emden's logic programs to the bilattice-based case. Unlike the programs we consider in this chapter, the programs of Lakshmanan and Sadri have interval-based annotations and do not contain function symbols. That is why we need to give our own definitions of bilattice-based implication-based logic programs, as follows.

Definition 4.4.7. *An* Implication-based Bilattice Normal Logic Program P *consists of a finite set of* program clauses *of the form*

$$A \leftarrow \boxed{f, g} - L_1, \ldots, L_n,$$

where A is a first-order atomic formula called the head *of the clause and $L_1, \ldots, L_n$ are first-order literals, called the* body *of the clause and (f, g) is a factor or threshold taken from the bilattice $[0, 1] \times [0, 1]$. Literals in the body are thought of as being connected using $\otimes$.*

Example 4.4.5. *The following program P is an implication-based logic program:*

$$B \leftarrow\boxed{1,0}\!\!-$$
$$B \leftarrow\boxed{0,1}\!\!-$$
$$A \leftarrow\boxed{1,1}\!\!- \quad B$$

All first-order formulae receive their interpretation from the bilattice $\mathbf{B} = L_1 \odot L_2$, where $L_1 = L_2 = ([0,1], \leq_k, \leq_t)$, using the interpretation function $\mathcal{I}$ from Definition 3.4.1. Up to the end of this section we use $|\ |$ to denote $|\ |_{\mathcal{I}}$. The value of the head A contained in a clause $A \leftarrow\boxed{f,g}\!\!- L_1, \ldots, L_n$, is computed as $(f \times \min(|L_1|_1, \ldots, |L_n|_1), g \times \min(|L_1|_2, \ldots, |L_n|_2))$, where $\min(|L_1|^i, \ldots, |L_n|^i)$ is the minimum value of the sentences $L_1, \ldots, L_n$, indices $i \in \{1,2\}$ denote the first and the second elements constituting a pair of values from $\mathbf{B}$. Empty bodies are thought of as having interpretation $\langle 1,1 \rangle$.

Definition 4.4.8. *Let $A \leftarrow\boxed{f,g}\!\!- L_1, \ldots, L_n$ be a clause of an Implication-based Bilattice logic program.*

We say that the BAP clause

$$A : (\vartheta \times (\mu_1 \cap \ldots \cap \mu_n), g \times (\nu_1 \cap \ldots \cap \nu_n)) \leftarrow L_1 : (\mu_1, \nu_1), \ldots, L_n : (\mu_n, \nu_n),$$

where each (μ_i, ν_i) is an annotation variable, translates the implication-based clause

$$A \leftarrow\boxed{f,g}\!\!- L_1, \ldots, L_n.$$

Example 4.4.6. *Consider the implication-based logic program P from Example 4.4.5. According to Definition 4.4.8, we have the following translation of this program into a*

BAP P^{VE}*:*

$$\begin{aligned} B : (1,0) &\leftarrow \\ B : (0,1) &\leftarrow \\ A : (\mu,\nu) &\leftarrow B : (\mu,\nu) \end{aligned}$$

In the last clause, we write $A : (\mu,\nu)$ *instead of* $A : ((1 \times \mu),(1 \times \nu))$.

We give here a definition of the immediate consequence operator T'_P for Implication-based Bilattice Logic Programs which generalises Van Emden's T_P operator from Definition 2.0.1 to the bilattice case in the obvious way.

Definition 4.4.9. $T'_P(\mathcal{I})(A) = \{\oplus\langle(f \times \cap\{|L_i|_1\}),(g \times \cap\{|L_i|_2\})\rangle : i \in \{1,\ldots,n\}$ *and* $A \leftarrow \boxed{f,g} - L_1,\ldots,L_n$ *is a variable-free instance of an implication-based clause in* $P\}$, $|L_i|_1$ *and* $|L_i|_2$ *denote the first and second elements in the pair of values taken from a product bilattice.*

Similarly to the Φ_P operator of Fitting, this T'_P operator works directly with values of first-order atoms. This distinguishes Φ_P and T'_P from $\mathcal{T}_P$ and $\widehat{\mathcal{T}_P}$; the latter, similarly to the classical operator T_P from Definition 1.5.1, work with sets of annotated atoms, and not with values.

The transfinite sequence of Definition 1.5.6 needs to be reformulated similarly to the Definition of $\Phi_P \uparrow \alpha$.

Definition 4.4.10.

$T'_P \uparrow 0 = \bot$, *where* $\bot$ *is the least element of* $\mathbf{B}$ *with respect to the k-ordering,*

$T'_P \uparrow \alpha = T'_P(T'_P(\alpha - 1))$, *if* α *is a successor ordinal,*

$T'_P \uparrow \alpha = \bigsqcup\{T'_P \uparrow \beta : \beta < \alpha\}$, *if* α *is a limit ordinal.*

Van Emden showed in [203] that implication-based logic programs have continuous semantic operators, and so the least fixed point of T'_P will be reached in at most ω steps and will compute the least model of the given implication-based logic program P.

Example 4.4.7. *Consider the implication-based logic program* P *from Example 4.4.5. The least fixed point of* T'_P *will give* $\{\mathcal{I}(B) = \langle 1, 0\rangle, \mathcal{I}(B) = \langle 0, 1\rangle, \mathcal{I}(A) = \langle 1, 1\rangle\}$. *(The valuation* $\mathcal{I}(A) = \langle 1, 1\rangle$ *subsumes* $\mathcal{I}(A) = \langle 0, 1\rangle$, $\mathcal{I}(A) = \langle 1, 0\rangle$ *and* $\mathcal{I}(A) = \langle 0, 0\rangle$.*)*

Note that $\mathcal{I}(A) = \langle 1, 1\rangle$ *was not computed by* $\widehat{\mathcal{T}_{P^F}}$ *in Example 4.4.4.*

Next we show that the translation algorithm of Definition 4.4.8 preserves the model properties of T'_P.

That is, we wish to relate the semantic operators T'_P and $\widehat{\mathcal{T}_P}$.

But first consider the following example.

Example 4.4.8. *Let* P *be the implication-based logic program from Example 4.4.5. We determined the least fixed point of* T'_P *for this program in Example 4.4.7. Also, in Example 4.4.6 we found the annotation translation* P^{VE} *of this implication-based logic program* P.

Now we compute the least fixed point of $\widehat{\mathcal{T}_{P^{VE}}}$ *as follows:* $\{B : (1, 0), B : (0, 1), A : (1, 0), A : (0, 1)\}$. *Despite the fact that* $\mathrm{lfp}(T'_P)(\mathcal{I})(A) = \langle 1, 1\rangle$, $\widehat{\mathcal{T}_{P^{VE}}}$ *does not compute* $A(1, 1)$ *at its least fixed point. Thus,* $\widehat{\mathcal{T}_{P^{VE}}}$ *cannot fully simulate* T'_P.

But $\mathrm{lfp}(\mathcal{T}_{P^{VE}}) = \{B : (0, 0), B : (1, 0), B : (0, 1), A : (0, 0), A : (1, 0), A : (0, 1), B : (1, 1), A : (1, 1)\}$. *And thus, this set contains* $A : (1, 1)$. *And we will show in the next theorem that* $\mathcal{T}_{P^{VE}}$ *from Definition 4.2.6 can fully simulate* T'_P.

However, as we can easily see, despite the fact that $B : (1,1) \in \text{lfp}(\mathcal{T}_{P^{VE}})$*, we do not have* $\text{lfp}(T'_P)(\mathcal{I})(B) = \langle 1,1\rangle$*. And thus,* $\mathcal{T}_P^{VE}$ *cannot be fully simulated by* T'_P*.*

Therefore, T'_P is intermediate between $\widehat{\mathcal{T}_P}$ and $\mathcal{T}_P$: it can compute more than $\widehat{\mathcal{T}_P}$ but less than $\mathcal{T}_P$. In the next theorem we formally prove this relation between T'_P, $\mathcal{T}_P$ and $\widehat{\mathcal{T}_P}$, as follows.

Theorem 4.4.2. *Let* P *be an implication-based bilattice logic program, and* A *be a ground instance of a first-order atom appearing in* P*. And let* P^{VE} *be the BAP obtained from* P *using Definition 4.4.8. Then the following holds.*

(1)*. If* $A : (\alpha,\beta) \in \text{lfp}(\widehat{\mathcal{T}_{P^{VE}}})$ *for the BAP* P^{VE}*, then the formula* A *receives the value* $\langle \alpha',\beta'\rangle$ *in the least fixed point of* (T'_P) *where* $\langle \alpha,\beta\rangle \leq_k \langle \alpha',\beta'\rangle$*.*

(2)*. If a formula* A *receives the value* $\langle \alpha,\beta\rangle$ *in the least fixed point of* (T'_P) *then* $A : (\alpha,\beta) \in \text{lfp}(\mathcal{T}_{P^{VE}})$*.*

Note that the proof of the analogous theorem relating $T_P^{\blacklozenge}$ from Definition 2.0.1 and R_P from Definition 2.0.2 in [108] uses the argument that $\text{lfp}(T_P^{\blacklozenge})$, the analogue of $\text{lfp}(T'_P)$, is always finite. This is not always the case. Consider Examples 3.5.4 and 4.2.6: this annotated logic program can be seen as a translation of Van Emden's bilattice-based logic program into an annotated program, but it can be interpreted only by an infinite bilattice.

Fact 4.4.1. *To prove the stated theorem, we use the fact that, in programs* P^{VE}*, annotation functions are not contained in the bodies of clauses and, as a consequence, the annotation function "*$\times$*" in heads of clauses will allow* $\mathcal{T}_P$ *to generate only a descending (relative to the k-ordering) sequence of values from* $\mathbf{B}$ *which tends to* $\langle 0,0\rangle$*. Any such sequence has an upper bound given by finite join with respect to the k-ordering in* $\mathbf{B}$*.*

Proof. The proof proceeds by induction on the number of iterations of T'_P, $\widehat{\mathcal{T}_{P^{VE}}}$ and $\mathcal{T}_{P^{VE}}$ needed to reach the least fixed point.

(1).

First we need to prove that if $A : (\alpha, \beta) \in \mathrm{lfp}(\widehat{\mathcal{T}_{P^{VE}}})$ for the BAP P^{VE}, then A receives the value $\langle \alpha', \beta' \rangle$ in the least fixed point of (T'_P) for P, where $\langle \alpha, \beta \rangle \leq_k \langle \alpha', \beta' \rangle$.

Basis step. Let $\mathrm{lfp}(\widehat{\mathcal{T}_{P^{VE}}})$ be obtained on the first iteration of $\widehat{\mathcal{T}_{P^{VE}}}$ and $A : (\alpha, \beta) \in \mathrm{lfp}(\widehat{\mathcal{T}_{P^{VE}}}) \uparrow 1$. Thus, $A : (\alpha, \beta) \leftarrow \in \mathrm{ground}(P^{VE})$. But then

$$A \leftarrow \boxed{\alpha, \beta} - \ \in \mathrm{ground}(P),$$

and $\mathrm{lfp}(T'_P)(\mathcal{I})(A) = \langle \alpha, \beta \rangle$.

Inductive step.

Induction hypothesis: Suppose the theorem holds for k, that is, if $B_j : (\alpha_j, \beta_j) \in \widehat{\mathcal{T}_{P^{VE}}} \uparrow k$, then $(T'_P \uparrow k)(\mathcal{I})(B_j) = \langle \alpha_j, \beta_j \rangle$ for $j = 1, \ldots, n$.

Let $A : (\alpha, \beta) \in \widehat{\mathcal{T}_{P^{VE}}} \uparrow (k+1)$. Then, by Definition 4.2.5 of $\widehat{\mathcal{T}_P}$, and by Definition 4.2.2 of $\mathrm{ground}(P^{VE})$, there exists

$$A : (f \times (\alpha_1 \cap \ldots \cap \alpha_n), g \times (\beta_1 \cap \ldots \cap \beta_n)) \leftarrow B_1 : (\alpha_1, \beta_1), \ldots, B_n : (\alpha_n, \beta_n) \in \mathrm{ground}(P^{VE})$$

(we label the latter proposition by $(***)$), such that

$$(\alpha, \beta) = (f \times (\alpha_1 \cap \ldots \cap \alpha_n), g \times (\beta_1 \cap \ldots \cap \beta_n)) \quad (****),$$

where $(\alpha, \beta) \leq_k (\alpha', \beta')$, and each $B_j : (\alpha'_j, \beta'_j) \in \widehat{\mathcal{T}_{P^{VE}}} \uparrow k$, $j = 1, \ldots, n$.

In order to proceed with the proof, we must characterise (α, β) first:

The clause $(***)$ is a ground instance of the clause

$$A : (f \times (\mu_1 \cap \ldots \cap \mu_n), g \times (\nu_1 \cap \ldots \cap \nu_n)) \leftarrow B_1 : (\mu_1, \nu_1), \ldots, B_n : (\mu_n, \nu_n).$$

In general, there can be infinitely many ground instances of this clause. But not all of these ground instances will be used in computations at $\widehat{\mathcal{T}_{PVE}} \uparrow (k+1)$. Consider those instances that will be used: they are the ground clauses containing precisely those $B_i : (\alpha'_i, \beta'_i)$, $(i = 1, \ldots, n)$, that are contained in $\widehat{\mathcal{T}_{PVE}} \uparrow k$, but also the instances $B_i : (\alpha^*_i, \beta^*_i)$ such that $(\alpha^*_i, \beta^*_i) \leq_k (\alpha'_i, \beta'_i)$. This is guaranteed by Definitions 4.2.2 and 4.2.5. However, these definitions do not guarantee that the least upper bound $(\alpha^{\blacktriangle}_i, \beta^{\blacktriangle}_i)$, of all such annotations (α^*_i, β^*_i) for B_i, $(i = 1, \ldots, n)$, will appear in $\widehat{\mathcal{T}_{PVE}} \uparrow k$ and participate in computations of $\widehat{\mathcal{T}_{PVE}} \uparrow (k+1)$.

We will consider the two cases where $(\alpha, \beta) = (f \times (\alpha_1 \cap \ldots \cap \alpha_n), g \times (\beta_1 \cap \ldots \cap \beta_n))$ can be obtained: the unique case when each $(\alpha_i, \beta_i) = (\alpha^{\blacktriangle}_i, \beta^{\blacktriangle}_i)$, and then the case when each $(\alpha_i, \beta_i) \leq_k (\alpha^{\blacktriangle}_i, \beta^{\blacktriangle}_i)$.

Consider the first case.

The clause $(***)$ is a ground instance of the clause translating the implication-based clause

$$A \leftarrow \boxed{f, g} - B_1, \ldots, B_n$$

from ground(P). By the induction hypothesis, we know that each
$(T'_P \uparrow k)(\mathcal{I})(B_j) = \langle \alpha_j, \beta_j \rangle$.

But $(T'_P \uparrow (k+1))(\mathcal{I})(A) = \langle \cup[(f \times (|B_1|_1 \cap \ldots \cap |B_n|_1))], \cup[(g \times (|B_1|_2 \cap \ldots \cap |B_n|_2))] \rangle = \oplus[\langle (f \times (|B_1|_1 \cap \ldots \cap |B_n|_1)), (g \times (|B_1|_2 \cap \ldots \cap |B_n|_2)) \rangle]$, for all possible values $|B_1|, \ldots, |B_n|$. But then, using the assumption we have made that each (α_i, β_i) is the join of all other values for B_i, we rewrite $(T'_P \uparrow (k+1))(\mathcal{I})(A) = \langle (f \times \cap(\alpha_1, \ldots, \alpha_n)), (g \times \cap(\beta_1, \ldots, \beta_n)) \rangle$. Now we use $(****)$ and obtain $T'_P \uparrow (k+1))(\mathcal{I})(A) = \langle \alpha, \beta \rangle$.

In the case when each $(\alpha_i, \beta_i) \leq_k (\alpha^{\blacktriangle}_i, \beta^{\blacktriangle}_i)$, the values $(\alpha_i, \beta_i) <_k (\alpha^{\blacktriangle}_i, \beta^{\blacktriangle}_i)$ may never be used by T'_P. So, it is possible that, as in Example 4.4.7, $\widehat{\mathcal{T}_{PVE}}$ will compute only

$A : (\alpha^*, \beta^*)$, with $(\alpha^*, \beta^*) \leq_k (\alpha, \beta)$, whereas T'_P computes $\mathcal{I}(A) = \langle \alpha, \beta \rangle$. But note that, whenever T'_P computes some value (α, β) for A, one usually interprets this as "the value of A is at least $\langle \alpha, \beta \rangle$", and assumes the lower interpretations, similarly to item 1 of Proposition 3.4.1.

And thus, we have a slight asymmetry in the statement **(1)**. That is, we can assert only that $A : (\alpha, \beta) \in \text{lfp}(\widehat{\mathcal{T}_{P^{VE}}})$ implies that $\text{lfp}(T'_P)(\mathcal{I})(A) = \langle \alpha', \beta' \rangle$, where $(\langle \alpha, \beta \rangle) \leq_k (\langle \alpha', \beta' \rangle)$.

(2).

Next, we need to prove that if a formula A receives the value $\langle \alpha, \beta \rangle$ in the least fixed point of (T'_P) for an Implication-based Bilattice Normal Logic Program P, then $A : (\alpha, \beta) \in \text{lfp}(\mathcal{T}_{P^{VE}})$ for the BAP P^{VE}.

Basis step. Let $\text{lfp}(T'_P) = T'_P \uparrow 1$ and suppose that $(T'_P \uparrow 1)(\mathcal{I})(A) = \langle \alpha, \beta \rangle$. Thus, $A \leftarrow \boxed{\alpha, \beta} - \in \text{ground}(P)$. But then there is a clause $A : (\alpha, \beta) \leftarrow$ in $\text{ground}(P^{VE})$, and hence $A : (\alpha, \beta) \in \mathcal{T}_{P^{VE}} \uparrow 1$, and $A : (\alpha, \beta) \in \text{lfp}(\mathcal{T}_{P^{VE}})$.

Inductive step. Suppose the theorem holds for k, that is, for any $B_i \in P$, if $(T'_P \uparrow k)(\mathcal{I})(B_i) = \langle \alpha_i, \beta_i \rangle$, then $B_i : (\alpha_i, \beta_i) \in \mathcal{T}_{P^{VE}} \uparrow k$.

Let $(T'_P \uparrow (k+1))(\mathcal{I})(A) = \langle \alpha, \beta \rangle$. This means that

$$A \leftarrow \boxed{f, g} - B_1, \ldots, B_n \in \text{ground}(P), \qquad (*)$$

and

$$\langle \alpha, \beta \rangle = \oplus[\langle (f \times (|B_1|_1 \cap \ldots \cap |B_n|_1)), (g \times (|B_1|_2 \cap \ldots \cap |B_n|_2)) \rangle] \ (\bigstar),$$

for all the possible values

$(f \times (|B_1|_1 \cap \ldots \cap |B_n|_1))$, $(g \times (|B_1|_2 \cap \ldots \cap |B_n|_2))$. Moreover, each B_j, $j = 1, \ldots, n$, receives a value in $(T'_P \uparrow k)$. In other words, for each $j = 1, \ldots, n$,

$$(T'_P \uparrow k)(\mathcal{I})(B_j) = \langle \alpha_j, \beta_j \rangle, \qquad (**)$$

where $\langle \alpha_j, \beta_j \rangle$ is some value from $\mathbf{B}$. We can use the induction hypothesis, and conclude that each $B_j : (\alpha_j, \beta_j) \in (\mathcal{T}_P^{VE}) \uparrow k$.

Let us consider these values (α_j, β_j). Because each value $|B_j|$, $j = 1, \ldots, n$ was computed at $(T'_P \uparrow k)(\mathcal{I})$ according to Definition 4.4.9, we conclude that the value $\langle \alpha_j, \beta_j \rangle$ is the k-join of all the possible values $\langle \alpha_j^1, \beta_j^1 \rangle, \langle \alpha_j^2, \beta_j^2 \rangle, \ldots$ for B_j computed in $k-1$ iterations of T'_P. When T'_P computes the value $\langle \alpha, \beta \rangle$ for A, it collects all the values for each B_j, $j = 1, \ldots, n$ computed in k iterations, and it uses the join $(\alpha_j^{\blacktriangle}, \beta_j^{\blacktriangle})$ of all such values for B_j to compute the value $\langle \alpha, \beta \rangle$ for A.

Thus, the equality ($\bigstar$) can be rewritten as

$$\langle \alpha, \beta \rangle = \langle (f \times (\alpha_1^{\blacktriangle} \cap \ldots \cap \alpha_n^{\blacktriangle})), (g \times (\beta_1^{\blacktriangle} \cap \ldots \cap \beta_n^{\blacktriangle})) \rangle \quad (\bigstar\bigstar).$$

By Definition of $\mathcal{T}_P$, we know that if there are several $B_j : (\alpha_j^*, \beta^*), B_j : (\alpha_j^{**}, \beta_j^{**}), \ldots \in \mathcal{T}_{P^{VE}} \uparrow k$, then the finite joins of these annotations will be attached to B_j at $\mathcal{T}_{P^{VE}} \uparrow (k+1)$. By Fact 4.4.1, finite joins will be sufficient to compute the (lower) upper bound $(\alpha_j^{\blacktriangle}, \beta_j^{\blacktriangle})$ for all $(\alpha_j^*, \beta^*), (\alpha_j^{**}, \beta_j^{**}), \ldots$.

By Definition 4.4.8, the following clause translates the clause $(*)$:

$$A : (f \times (\mu_1 \cap \ldots \cap \mu_n), g \times (\nu_1 \cap \ldots \cap \nu_n)) \leftarrow B_1 : (\mu_1, \nu_1), \ldots, B_n : (\mu_n, \nu_n).$$

Then, the set ground(P^{VE}), containing all the ground instances of this clause will be used for computations of $\mathcal{T}_{P^{VE}}$. By Definition 4.2.2, among other ground clauses, ground(P^{VE}) will contain the clause

$$A : (f \times (\alpha_1^{\blacktriangle} \cap \ldots \cap \alpha_n^{\blacktriangle}), g \times (\beta_1^{\blacktriangle} \cap \ldots \cap \beta_n^{\blacktriangle})) \leftarrow B_1 : (\alpha_1^{\blacktriangle}, \beta_1^{\blacktriangle}), \ldots, B_n : (\alpha_n^{\blacktriangle}, \beta_n^{\blacktriangle}),$$

where each $(\alpha_j^{\blacktriangle}, \beta_j^{\blacktriangle})$, $j = 1, \ldots, n$ is as defined as above.

Then, since such a clause is in ground(P^{VE}), and by the discussion above each $B_j(\alpha_j^{\blacktriangle}, \beta_j^{\blacktriangle})$ is in $\mathcal{T}_{P^{VE}} \uparrow (k+1)$, then $A : (f \times (\alpha_1^{\blacktriangle} \cap \ldots \cap \alpha_n^{\blacktriangle}), g \times (\beta_1^{\blacktriangle} \cap \ldots \cap \beta_n^{\blacktriangle})) \in \mathcal{T}_{P^{VE}} \uparrow (k+2)$. And thus, using ($\bigstar\bigstar$), $A : (\alpha, \beta) \in \text{lfp}(\mathcal{T}_{P^{VE}})$. □

The Lemma 4.4.1, Theorems 4.4.1 and 4.4.2 show that annotation-free and implication-based bilattice-based logic programs can be translated into BAPs. Once these programs are translated into BAPs, all the inference techniques we proposed for BAPs can be applied to these logic programs, among them will be the SLD-resolution defined in Section 4.3, and neural networks for BAPs defined in Chapter 6.

There is another interesting consequence of Lemma 4.4.1, Theorems 4.4.1 and 4.4.2. Since Theorems 4.4.1, 4.4.2 were proven using the restricted operator $\widehat{\mathcal{T}_P}$ and we showed in Section 4.2 that, unlike the extended semantic operator $\mathcal{T}_P$, $\widehat{\mathcal{T}_P}$ does not compute all the logical consequences of P, it is straightforward to show that Φ_P and T'_P do not compute all the logical consequences of a given P that $\mathcal{T}_P$ is capable of computing. And we used a simple example of an annotated logic program $P^F = P^{VE}$ to illustrate this fact in Examples 4.4.2, 4.4.4, 4.4.7, 4.4.8.

The expressiveness of the least fixed points of the semantic operators considered in this chapter can be compared using the following diagram (the lower least fixed points are less expressive than the higher ones):

$$\text{lfp}(\mathcal{T}_P)$$
$$\Uparrow$$
$$\text{lfp}(T'_P)$$
$$\Uparrow$$
$$\text{lfp}(\widehat{\mathcal{T}_P}) = \text{lfp}(\Phi_P)$$

In the case of Fitting's logic programs, the so-called completion introduced in [54] can improve the expressiveness of Φ_P. Each Fitting logic program can be completed as follows. Given a function-free annotation-free logic program P, form a set of all ground instances of clauses. Then all the ground clauses $A \leftarrow \text{body}_1, A \leftarrow \text{body}_2, \ldots$ having the same head A must be replaced by a single clause $A \leftarrow \text{body}_1 \oplus \text{body}_2 \oplus \ldots$. In this case, Φ_P computes all the logical consequences of the program P and the translation of P into a BAP always gives a program whose semantic operator $\mathcal{T}_{P^F}$ never uses the rules captured in item 2 of Definition 4.2.6. This means that for such programs, the computations performed by $\mathcal{T}_{P^F}$ and $\widehat{\mathcal{T}_{P^F}}$ are the same, and all the results established for $\mathcal{T}_{P^F}$ hold for $\widehat{\mathcal{T}_{P^F}}$.

4.5 Conclusions

We have carefully examined the declarative and operational semantics of annotated logic programs interpreted in arbitrary bilattices. The only restriction on the type of a bilattice was that only finite, but not infinite, meets can be defined with respect to $\leq_k$ in a given bilattice.

In Section 4.2, we defined the two semantic operators, $\mathcal{T}_P$ and $\widehat{\mathcal{T}_P}$ suitable for computations of the logical consequence of BAPs, we compared the two operators and showed that unlike $\mathcal{T}_P$, $\widehat{\mathcal{T}_P}$ does not compute *all* the logical consequences of a given BAP. We developed the fixed point semantics for $\mathcal{T}_P$ and proved continuity of $\mathcal{T}_P$.

This declarative semantics for BAPs allowed us to propose an SLD-resolution for BAPs and prove its soundness and completeness relative to the declarative semantics. Like the resolution procedures given in [107] for lattice-based logics, this SLD-resolution is enriched with additional rules reflecting the properties of the extended semantic operator for BAPs, and the SLD resolution we have proposed is alternative to the constrained resolution for

the Generalised Annotated Logic Programs (GAPs) of Kifer and Subrahmanian ([108]) and to resolutions for logics which are interpreted by linearly ordered sets [141, 205, 75] and/or finite sets [108, 141, 142, 143]. This is the first sound and complete SLD-resolution for annotated logic programs of this generality, cf., for example, the constrained SLD-resolution of [108] which was shown to be incomplete.

We analysed the relations between BAPs and $\mathcal{T}_P$ defined for BAPs with other classes of bilattice-based logic programs. In particular, we have shown that unlike the usual approach to many-valued logic programming semantics (see, for example, [50], [54], [203], [108] and many others), the immediate consequence operators for (bi)lattice-based annotated logic programs cannot be obtained as a trivial extension of the classical semantic operator, but need to include some additional conditions. We have given some examples displaying the immediate consequence operators for annotation-free and implication-based logic programs following [50], [54], [203], [108], and discussed the expressiveness of the least fixed points computed by these semantic operators relative to $\widehat{\mathcal{T}_P}$ or $\mathcal{T}_P$. In fact, we have shown that $\mathcal{T}_P$ is expressive enough to simulate the semantic operators of all the logic programs we have mentioned.

We will show in the next section that the semantic operator $\mathcal{T}_P$ for BAPs can be computed by learning artificial neural networks in the style of [99], but with learning functions embedded into connections between the layers.

Chapter 5

Neural Networks: Connectionism or Neurocomputing?

5.1 Introduction

In this chapter we introduce neural networks, give some historical introduction to the subject, outline the state of the art, and survey the relevant bibliography, paying special attention to the results we are going to use in the subsequent chapters. When surveying other papers, we motivate the importance of the novel results obtained in this thesis.

In the Introduction we have already distinguished the three main notions were are going to use when talking about Artificial Neural Networks, and these three notions were Neurocomputing, Connectionism and Neuro-Symbolic Integration.

Because the results of the remaining chapters are indisciplinary and related to each of the mentioned fields, we will briefly outline the history and achievements of each.

We start with Neurocomputing which is the most established of the three disciplines. We consulted [85, 2, 82].

Definition 5.1.1. ***Neurocomputing*** *is defined in [85] as a technological discipline concerned with information processing systems (neural networks) that autonomously develop*

operational capabilities in adaptive response to an information environment.

Moreover, neurocomputing is often seen as an alternative to *programmed computing*: the latter is based on the notion of some fixed *algorithm* (a set of rules) which must be performed by a machine; the former does not necessarily require an algorithm or rule development. Note that in particular, neurocomputing is specialised on the creation of machines implementing artificial neural networks, among them are digital, analog, electronic, optical, electro-optic, acoustic, mechanical, chemical and some other types of neurocomputers.

The beginnings of neurocomputing are often taken to be the 1943 paper of W. McCulloch and W. Pitts [151]. This paper, which showed that even simple types of neural networks could, in principle, compute any arithmetic or logical function, was widely read and had a great influence. Other researchers, principally Norbert Weiner and John von Neumann, wrote books and papers [164, 165] in which the suggestion was made that research into the design of brain-like or brain-inspired computers might be interesting.

In 1949 Donald Hebb wrote a book entitled "The Organisation of Behaviour" [83], that proposed a specific learning law for the synapses of neurons. Hebb used this learning law to explain some experimental results from psychology. This bold step served to inspire many other researchers to pursue the same theme - which further laid the groundwork for the advent of neurocomputing.

In 1951, the first Neurocomputer (*Snark*) by M. Minsky [159] was constructed. The Snark did operate successfully from a technical standpoint (it adjusted its weights automatically), but it never actually carried out any particularly interesting information processing function. Nonetheless, it provided design ideas that were used later by other investigators.

The first successfull neurocomputer (the Mark I Perceptron) was developed during 1957 and 1958 by F. Rosenblatt, C. Wightman, and others [179]. Many people see Rosenblatt as the founder of neurocomputing as we know it today. His primary interest was pattern recognition. Besides inventing Perceptron, Rosenblatt also wrote an early book on neurocomputing, "Principles of Neurodynamics" [180].

B.Widrow developed a different type of neural network processing element called ADALINE which was equipped with a powerful new learning law which, unlike the perceptron learning law, is still in widespread use. Widrow and his students applied the ADALINE successfully to a large number of toy problems. Widrow also founded the first neurocomputer hardware company (the Memistor Corporation), which actually produced neurocomputers and neurocomputer components for commercial sale during the early to mid 1960s.

In 1969, Minsky and Papert published the book "Perceptrons" [160]. The book proved mathematically that a Perceptron could not implement the EXCLUSIVE OR (XOR) logical function ($f(0,0) = f(1,1) = 0,\ f(0,1) = f(1,0) = 1$), nor many other such predicate functions. The implicit thesis of "Perceptrons" was that essentially all neural networks suffer from the same "fatal flaw" as the perceptron; namely, the inability to compute certain predicates such as XOR. To make this point, the authors reviewed several proposed improvements to the perceptron and showed that these were also unable to perform well.

These results had a great influence, and not much research in neurocomputing was being done during 1967–1982.

In 1983–1986, John Hopfield, an established physicist, became interested in neural

networks. He wrote two highly readable papers on neural networks [100, 101] and persuaded hundreds of highly qualified scientists, mathematicians and technologists to join the emerging field of neural networks.

In 1986, with the publication of "Parallel Distributed Processing (PDP)" by D. Rumelhart and J. McClelland [181], the field exploded. The PDP group rediscovered powerful learning rules which transcended the limitations discovered by Minsky and Papert.

At the moment, there is much activity all over the world concerning both theory of neurocomputing, and the implementation of neural networks.

From the point of view of applications, the utility of artificial neural networks lies in the fact that they can be used to infer a function from observations. This is particularly useful in applications where the complexity of the data or task makes the design of such a function by hand impractical. Real-life applications mainly lie within the three spheres:

1. Function approximation, or regression analysis, including time series prediction and modelling.

2. Classification, including pattern recognition, novelty detection and sequential decision making.

3. Data processing, including filtering, clustering, blind source separation and compression.

Application areas include system identification and control (vehicle control, process control), game-playing and decision making (backgammon, chess, racing), pattern recognition (radar systems, face identification, object recognition and more), sequence recognition (gesture, speech, handwritten text recognition), medical diagnosis, financial applications, data mining, visualisation and e-mail spam filtering. See also [85, 82, 59] for more details

concerning implementation of neural networks.

The so-called *Neural Networks software* has been developed and used to simulate, research, develop and apply neural networks. And there are several Neural Network Simulators available at the moment, such as *Stuttgart Neural Network Simulator (SNNS)* [212], *Parallel Distributed System Processing (PDP++)* [168], *JavaNNs* [47], and some others.

From the point of view of theoretical research, neural networks are being developed in various fields of computability theory and logic.

It is worth mentioning that there has been an interdisciplinary research involving model theory and neural networks, [106, 105].

There have been series of publications concerning computational complexity of neural networks and their relation to Turing computability. Pollack [172] showed that a particular type of heterogeneous processor network is Turing Universal. Siegelmann and Sontag [191] showed the universality of homogeneous networks of first-order neurons having piecewise-linear activation functions. Their results were generalised by Kilian and Siegelmann [109] to include various sigmoidal activation functions. One of the interesting questions concerned the smallest number of neurons that allows one to reach Turing computability and Turing universality. The number of the neurons required for universality with first-order neurons was estimated at 886 [192], and in the later papers was reduced to 96 [114] and down to 25 [103]. In [190], it was shown that there is a universal neural network with nine switch-affine neurons which is Turing universal, see also [209]. These results are very important, and we will often refer to them in the subsequent sections.

Another field where neural networks are receiving theoretical development is *Connectionism*.

Connectionism is a movement in the fields of artificial intelligence, cognitive science, neuroscience, psychology and philosophy of mind which hopes to explain human intellectual abilities using artificial neural networks. Connectionism is focused on the fact that neural networks are simplified models of the brain. It uses basic definitions of a neuron and of a neural network that are conventionally used in Neurocomputing, and therefore, many authors who work within Connectionism do not distinguish themselves from Neurocomputing.

However, the range of problems that connectionism solves and the methodology it uses differs from those of Neurocomputing. The latter is machine-oriented and is focused on computability issues, whereas Connectionism is looking for formalising and modelling psychological behaviour. Thus, in [72] we find the following definition of Connectionism.

"Connectionist systems aim at modelling aspects of the animal and human nervous system on an abstract computational level". And the same understanding of the aims of connectionism can be found in [45, 181]. (Notice the difference between how Neurocomputing formulates its subject in [85], see Definition 5.1.1: there is no particular emphasis on biological or psychological motivation.)

As we mentioned in the Introduction, there is an even bigger gap between the methodologies of Neurocomputing and Connectionism. Connectionism, in its attempts to give a neuro-symbolic account of human reasoning, develops its own methods of constructing neural networks, and does not effectively uses the learning techniques developed in Neurocomputing. Thus, classical connectionist papers [72, 97, 95, 98, 99, 188, 136, 94], do not develop the learning techniques of Section 5.4 to any significant extent. The textbook "Neural-Symbolic Learning Systems" [30] brings backtracking into the picture, but largely ignores the discussions of various learning functions of Neurocomputing surveyed

in [82, 85] and in Section 5.4 herein.

The computability (and "Turing Universality") results obtained in [172, 191, 109, 192, 114, 103, 190] are ignored in Connectionist literature [72, 95, 98, 99, 188, 136, 94].

Connectionism focuses in particular on the so-called topic of **neuro-symbolic integration**, which investigates ways of the integration of logic and formal languages with neural networks in order to better understand the essence of symbolic (deductive) and human (developing, spontaneous) reasoning, and to show interconnections between them. The books [30], [149] are good examples of this approach.

The field of neuro-symbolic integration is stimulated by the fact that formal theories (as studied in mathematical logic and used in automated reasoning) are commonly recognised as deductive systems which lack such properties of human reasoning, as adaptation, learning and self-organisation. On the other hand, neural networks, introduced as a mathematical model of neurons in the human brain, claim to possess all of the mentioned abilities, and moreover, they provide parallel computations and hence can perform certain calculations faster than classical algorithms, [190]. There were several attempts to combine the two paradigms, such as [188, 136, 95], and some others, they are all carefully surveyed in [72].

One of the most influential results in this area was obtained by Hölldobler et al. As a step towards the integration of logic and neural networks, there were built connectionist neural networks [98, 99] which can simulate the work of the semantic operator T_P for propositional and (function-free) first-order logic programs. We will call these neural networks T_P-neural networks. Those neural networks, however, were essentially deductive and could not learn or perform any form of self-organisation or adaptation. Moreover, in the first-order case, when the Herbrand base of a given logic program can be infinite, the

neural networks of [98, 99] require an infinite number of neurons to compute the least fixed point of T_P. Note that $\mathrm{lfp}(T_P) = T_P \uparrow n$ is Turing computable. Comparing with results about Turing universality of finite neural networks [172, 191, 109, 192, 114, 103, 190], this result of [98] seems to be very disappointing.

The starting point of this dissertation was the result of Hölldobler et al. that we briefly outlined above. And our first goal was to enrich T_P-neural networks with learning functions that reflect non-trivial properties of bilattice-based logic programs. We used Hebbian learning recognised in Neurocomputing, see Section 5.4 for the definition of it, in order to reach this goal in Chapter 6. However, we soon discovered that this extension of T_P-neural networks inherits the infiniteness of architecture from T_P-neural networks. This inspired us to make a more radical move from Connectionism towards Neurocomputing, and we construct in Chapter 7 SLD neural networks, that effectively simulate the algorithm of SLD-resolution using finite (and quite small) numbers of neurons and six learning functions recognised in Neurocomputing.

In this Chapter, we proceed as follows. Section 5.2 gives some basic definitions concerning neural networks. Section 5.3 focuses on T_P-neural networks, their main features, state-of-the-art, and open problems. We also give some more motivation for the results of Chapters 6 and 7, while surveying the existing literature. Section 5.4 defines several kinds of learning that are used in Neurocomputing and that will be used in the Chapters 6 and 7.

5.2 Neural Networks

In this section, we give formal definitions of neural networks.

We follow the definitions of a connectionist neural network given in [99], see also [30]

and [94]. This basic definition is no different from the analogous definition of a neural network accepted in Neurocomputing [85, 82].

A *connectionist network* is a directed graph. A *unit* k in this graph is characterised, at time t, by its *input vector* $(v_{i_1}(t), \ldots v_{i_n}(t))$, its potential $p_k(t)$, its *threshold* Θ_k, and its *value* $v_k(t)$. Note that in general, all v_i, p_i and Θ_i, as well as all other parameters of a neural network can be performed by different types of data, the most common of which are real numbers, rational numbers [99], fuzzy (real) numbers [163], complex numbers, numbers with floating point, and some others, see [85] for more details. In Chapters 6 and 7 we use rational numbers and Gödel numbers.

Units are connected via a set of directed and weighted connections. If there is a connection from unit j to unit k, then w_{kj} denotes the *weight* associated with this connection, and $i_k(t) = w_{kj}v_j(t)$ is the *input* received by k from j at time t. In each update, the potential and value of a unit are computed with respect to an *activation* and an *output function* respectively. Most units considered in this thesis compute their potential as the weighted sum of their inputs minus their threshold:

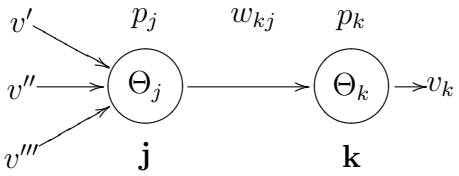
$$p_k(t) = \left(\sum_{j=1}^{n_k} w_{kj}v_j(t) \right) - \Theta_k.$$

The units are updated synchronously, time becomes $t + \Delta t$, and the output value for k, $v_k(t + \Delta t)$, is calculated from $p_k(t)$ by means of a given *output function* F, that is, $v_k(t + \Delta t) = F(p_k(t))$. For example, the output function used in [98] is the binary threshold function H, that is, $v_k(t + \Delta t) = H(p_k(t))$, where $H(p_k(t)) = 1$ if $p_k(t) > 0$ and 0 otherwise. Units of this type are called *binary threshold units*.

A unit is said to be a *linear unit* if its output function is the identity and its threshold Θ is 0. A unit is said to be a *sigmoidal* or *squashing unit* if its output function ϕ is non-decreasing and is such that $\lim_{t\to\infty}(\phi(p_k(t))) = 1$ and $\lim_{t\to-\infty}(\phi(p_k(t))) = 0$. Such

functions are called *squashing functions*.

Example 5.2.1. *Consider two units, j and k, having thresholds Θ_j, Θ_k, potentials p_j, p_k and values v_j, v_k. The weight of the connection between units j and k is denoted w_{kj}. Then the following graph shows a simple neural network consisting of j and k. The neural network receives input signals v', v'', v''' and sends an output signal v_k.*



We will mainly consider connectionist networks where the units can be organised in layers. A *layer* is a vector of units. An *n-layer network* $\mathcal{F}$ consists of the *input* layer, $n-2$ *hidden* layers, and the *output* layer, where $n \geq 2$. Each unit occurring in the i-th layer is connected to each unit occurring in the $(i+1)$-st layer, $1 \leq i < n$. Neural networks consisting of layers are sometimes called *associative neural networks* [85].

The primary classification of associative neural networks is into *feedforward* and *recurrent* classes. In feedforward neural network connections between units do not form a directed cycle. In recurrent neural networks, on the contrary, connections between units form a directed cycle.

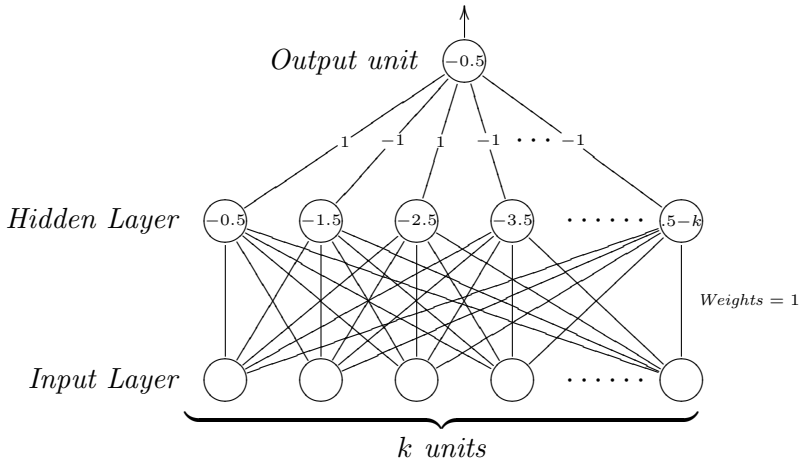
In many neural networks input units can receive only single inputs that arrive from the outside world. They typically have no function other than to distribute the signals to other layers of the neural networks. Such units are called *fanout units*.

There are interesting and useful results concerning the optimal number of layers needed for computations, and we will briefly outline them in the next subsection.

5.2.1 Three-Layer Neural Networks

At the end of the previous section we defined n-layer neural networks. Throughout this thesis, we work only with three-layer neural networks. This short subsection serves to motivate the use of three-layer neural networks.

Example 5.2.2. *A typical three-layer neural network can have the following architecture:*



In 1957 the mathematician Andrei Kolmogorov published an astounding theorem concerning the representation of arbitrary continuous functions from the n-dimensional cube $[0,1]^n$ to the real numbers R in terms of functions of only one variable [115]. This theorem intrigued a number of mathematicians and over the next twenty years several improvements to it were discovered, notably those by G.G. Lorentz [195, 140].

Although Kolmogorov's theorem is both powerful and shocking (most mathematicians do not believe it can be true when they first see it), it has not been found to be of much value in mathematics in terms of utility for proving other important theorems. However, this is *not* the case in neurocomputing!

Kolmogorov's mapping neural network existence theorem is stated below, this particular reformulation is due to [84].

Theorem 5.2.1 (Kolmogorov's Mapping Neural Network Existence Theorem)**.** *Given any continuous function $f : [0,1]^n \rightarrow R^m$, $f(x) = y$, f can be implemented exactly by a three-layer feedforward neural network having n fanout processing units in the first (input) layer, $(2n+1)$ processing units in the hidden layer, and m processing elements in the output layer.*

The proof can be found in [84].

Kolmogorov's Mapping Neural Network Existence Theorem is a statement that a quest for approximations of functions by neural networks is, at least in theory, sound. However, the direct usefulness of this result is doubtful, because no constructive method for developing these neural networks is known.

Some of the results in Connectionism and Neuro-Symbolic Integration, for example, approximation results of [99, 94, 184], are based on a reformulation of Kolmogorov's theorem due to Funahashi:

Theorem 5.2.2. *[58] Suppose that $\phi : \mathbb{R} \rightarrow \mathbb{R}$ is a non-constant, bounded, monotone increasing and continuous function. Let $K \subseteq \mathbb{R}^n$ be compact, let $f : K \rightarrow \mathbb{R}$ be a continuous mapping and let $\epsilon > 0$. Then there exists a 3-layer feedforward network with squashing function ψ whose input-output mapping $\overline{f} : K \rightarrow \mathbb{R}$ satisfies $max_{x \in K} d(f(x), \overline{f}(x)) < \epsilon$, where d is a metric which induces the natural topology on $\mathbb{R}$.*

Because of the theorems stated above, from the theoretical point of view it is preferable to research into neural networks having precisely three layers.

Note that it was shown in [98, 94] that two-layer neural networks simulating T_P are incapable of computing logical consequences of a logic program expressing XOR. And thus, three-layer neural networks were chosen to simulate T_P.

All the neural networks we consider in the following chapters are three-layer neural networks.

5.3 T_P-Neural Networks

We call the neural networks of [104, 98, 99] simulating the semantic operator T_P, *T_P-neural networks*. We will discuss the construction of T_P-neural networks and their computational abilities next.

The next theorem establishes a correspondence between the semantic operator T_P for definite logic programs and the three-layer connectionist neural networks and is taken from [98, 99]. It was originally formulated to incorporate Normal logic programs, that is, logic programs containing negation. However, the definition of the T_P operator for this class of logic programs differs from the one given for definite logic programs in Section 1.5. In order to be mathematically precise, we restrict the original theorem to definite logic programs. (The detailed discussion of definite logic programs and the semantic operator for them are given in Chapter 1.)

Theorem 5.3.1. *[98] For each definite propositional logic program P, there exists a 3-layer recurrent neural network built of binary threshold units that computes T_P.*

Proof. The core of the proof is the translation algorithm from a logic program P into a corresponding neural network, which can be briefly described as follows.

- The input and output layer are vectors of binary threshold units of length m, where the i-th unit in the input and output layer represents the i-th proposition. All units in the input and output layers have thresholds equal to 0.5.
- For each clause $A \leftarrow B_1, \ldots, B_l$, do the following.

- Add a binary threshold unit c to the hidden layer.
- Connect c to the unit representing A in the output layer with weight 1.
- For each atom B_j, $1 \leq j \leq l$, connect the unit representing B_j to c with weight 1.
- Set the threshold Θ_c of c to $l - 0.5$, where l is the number of atoms occurring in $B_1, \ldots, B_l$.

For each proposition A, connect the unit representing A in the output layer to the unit representing A in the input layer via a connection with weight 1. These are *recurrent* connections. □

We will call the period of time during which the T_P-neural network receives a signal at the input layer and propagates it through the hidden and output layer an *iteration* of the neural network.

Note 5.3.1. *This theorem first appeared in [98] and the neural networks constructed in the proof above were called "feedforward" despite of having recurrent connections. In later papers [99, 94, 119] containing the same theorem and the same construction, the recurrent nature of the neural networks was emphasised, but the theorem was stated as follows: "For each definite propositional logic program P, there exists a 3-layer* feedforward *neural network built of binary threshold units that computes T_P". Theorem 5.3.1 we have stated here is taken from [98, 99, 94] modulo this small terminological modification concerning recurrent nature of the networks. We do not change the construction of neural networks of [98] here.*

Theorem 5.3.1 easily generalises to the case of function-free first-order logic programs, because the Herbrand Base for these logic programs is finite and hence it is possible

to work with a finite number of ground atoms as with propositions. However, if we wish to use the construction of Theorem 5.3.1 to compute the semantic operator for conventional first-order logic programs with function symbols, we will need to use some kind of approximation theorems as in [94],[184] or [10] to give an account of cases when the Herbrand Base is infinite, and hence an infinite number of neurons is needed to simulate the semantic operator for these logic programs. We will further illustrate the work of the approximation method in Section 6.3.

Note 5.3.2. *There are* ***four characteristic properties*** *that distinguish T_P-neural networks. We summarise them as follows.*

1. *The number of neurons in the input and output layers is the number of atoms in the Herbrand base B_P of a given program P.*

2. *The number of iterations of T_P (for a given logic program P) corresponds to the number of iterations of the neural network built upon P.*

3. *Signals of T_P-neural networks are binary, and this is achieved by using binary threshold activation functions. This provides the computations of truth value functions $\wedge$ and $\leftarrow$ that are used in program clauses.*

4. *As a consequence of the property (3), first-order atoms are not presented in the neural network directly, and only truth values* 1 *and* 0*, that are the same for all the atoms, are propagated.*

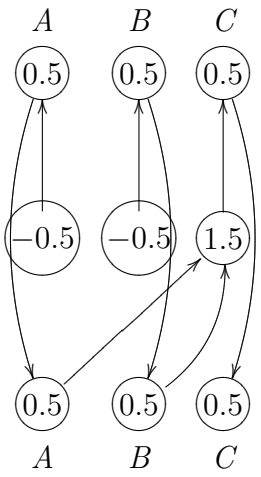
Example 5.3.1. *Consider the ground logic program from Example 1.3.4. For simplicity of notation, we will reformulate it as follows:*
$B \leftarrow$

$A \leftarrow$

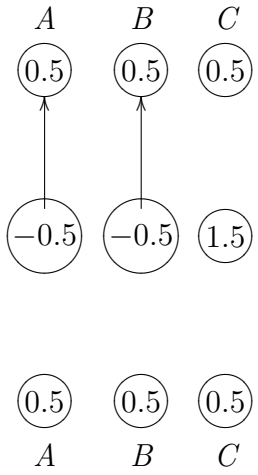
$C \leftarrow A, B.$

The neural network below computes $T_P \uparrow 2$ *in two iterations, and we illustrate the iterations in a series of diagrams.*

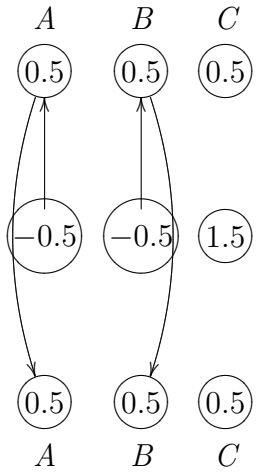
Each of the atoms A, B, C *is presented in the output and input layers. Connections between the hidden layer and both outer layers are fixed in order to reflect the structure of clauses. The weights of the recurrent connections between the output and the input layer are set to* 1*. That is, all the connections* $\longrightarrow$ *on the following diagram have weights which are set to* 1*:*



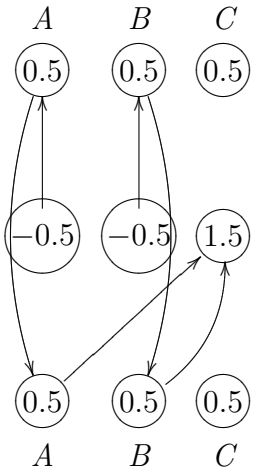
Time $t = 0$*. Because the thresholds of the two leftmost hidden units were defined to be negative, at time* $t = 0$ *these units become excited and they send signals* 1 *to the output neurons that are connected to them:*



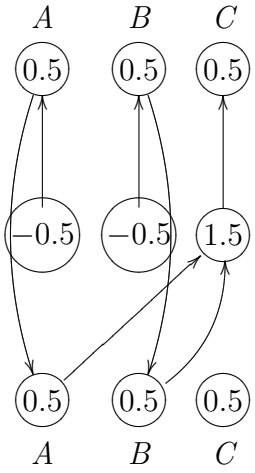
Time $t = 1$. *The signals equal to* 1 *are sent from the output layer to the input layer:*



Because units representing A *and* B *in the input layer are connected to the rightmost hidden unit, the signals* 1 *are sent from the units representing* A *and* B *in the input layer to this hidden unit at time* $t = 1$*:*



Time $t = 2$. *The excited hidden unit is connected only to the rightmost output unit, and so signal* 1 *is sent there, and at time* $t = 2$ *the unit "representing"* C *in the output layer emits the signal* 1.



The animated version of this example can be found in [116].

There are several immediate conclusions to be made from the construction of T_P-neural networks. We have described a procedure of how the Least Herbrand model of a function-free logic program can be computed by a three-layer recurrent connectionist

neural network. However, the neural networks do not possess some essential and most beneficial properties of the artificial neural networks used in Neurocomputing, and among such are learning and self-adaptation.

The T_P-neural networks are essentially truth tables computing classical truth values of program clauses. This is why, atoms are not encoded directly, units are just "thought of" as representing certain atoms. Furthermore, different atoms can be represented by the same parameters in order to compute the same truth values 0 and 1.

Moreover, the T_P-networks depend on ground instances of clauses, and in the case of logic programs containing function symbols, will require infinitely long layers to compute the least fixed point of T_P. This property does not agree with the very idea of neurocomputing, which advocates another principle of computation: effectiveness of neural networks depends primary on their architecture, which is finite, but allows very sophisticated and "well-trained" interconnections between neurons. (It is curious that in human brains the overall number of neurons constantly decreases during the lifetime, but the neurons which remain active are able to increase and optimise interconnections with other neurons, see [82, 59] for more details.) The way of approximation introduced in [99, 94, 184] supports the opposite direction of increasing the number of units in the artificial neural network (in order to approximate its infinite counterpart) rather than of improving connections within a given neural network.

Recall that the original goal of connectionism was to combine the most beneficial properties of automated reasoning and neural networks in order to build more effective and "wiser" algorithms. The natural questions is: why the Connectionist neural networks of [99] do not possess the essential properties of Neurocomputing neural networks and what are the ways to improve them and bring learning and self-organisation into their

computations? Are there ways to realise logical deduction, and not just propagation of truth values, in neural networks? There may be two alternative answers to the question.

1. The connectionist neural networks we have described simulate classical two-valued deductive system, and they inherit the style of processing a given database from classical logic. If this is the case, we can adapt the algorithm for building a T_P-neural network to non-classical logic programs. And different kinds of logic programs for reasoning with uncertainties and incomplete/inconsistent databases are natural candidates for this. In Section 6.2 we show that connectionist neural networks computing the Least Herbrand Model for a given bilattice-based annotated logic program use unsupervised (Hebbian) learning. These neural networks, however, still require approximation theorems when working with logic programs containing function symbols. And, similarly to T_P-neural networks, they work with truth values, rather than with symbols of the language.

These problems led us to yet another answer:

2. It may be the choice of the T_P operator which has led us in the wrong direction. What if we try to simulate classical SLD-resolution in connectionist neural networks? The main benefit of it is that we will not need to work with infinite neural network architectures anymore, and will not depend on truth values. So, we introduce an algorithm for constructing neural networks simulating the work of SLD-resolution in Section 7.2. In fact, the neural networks of this type possess many more useful properties than just finite architecture. In particular, we show that these neural networks incorporate six learning functions which are recognised in neurocomputing, and can perform parallel computations for certain types of program goals.

In the Introduction, we have already characterised the approaches 1) and 2): we called the first approach of developing T_P-neural networks *extensive*, and the second -

intensive. The choice of this terminology is obvious: the extensive way simply extends T_P-neural networks to different non-classical logic programs, and this was pursued by [5, 34, 33, 29, 30, 32, 135, 200]. We will survey their main results in the next subsection. The intensive way suggested in this thesis completely rebuilds T_P-neural networks in order to encode first-order atoms directly in the networks. This way we obtain an SLD-resolution interpreter working by means of Neural networks. The resulting neural networks require a finite number of neurons and several learning functions.

5.3.1 Extensions of T_P-Neural Networks: Outline of the Relevant Literature

T_P-neural networks have had an immense influence on the research in the field of Neuro-Symbolic Integration, and several (extensive) improvements to T_P- neural networks have been suggested. The book [78] contains an up-to-date collection of papers extending and generalising T_P-neural networks. All the extensions of T_P-neural networks we are aware of inherit the four major properties of T_P-neural networks summarised in Note 5.3.2. We will give more details as follows.

Learning The T_P-neural networks have not been trained by the usual learning methods applied in Neurocomputing. However, there were attempts to develop an original Learning Theory within Neuro-Symbolic integration. In [34, 33, 30], binary threshold units in the hidden layer were replaced by sigmoidal units. Such networks allow back-propagation which can be used to train neural networks. These neural networks copied exactly the features (1), (2), (4) stated in Note 5.3.2. The property (3) was slightly modified, as it allowed signals from the interval $[-1, 1]$ to be sent from the hidden layer. This was applied to inductive logic programming, where facts in a database can be assigned

some measures of probabilities. The values from the interval $[-1, 0]$ were recognised by the output unit as "false", and the values from the interval $[0, 1]$ - as "true". Then the two classical truth values were used to compute $\wedge$ and $\leftarrow$, precisely as in item (3) of Note 5.3.2. Sigmoidal units allow back-propagation, which can be seen as a step towards learning.

Rule Extraction This area of development was very much stimulated by the Neural networks of [34, 33, 30] with sigmoidal hidden units. The computations performed by such neural networks required some post-processing in order to be adequately interpreted by a human. So, the problem of how to extract the rules that are being used by these neural networks was tackled in [5, 29, 200]. This sort of research could be aimed at reformulation of the property stated in Item (4) of Note 5.3.2, but unfortunately, the cited papers all dealt only with truth values, and not with encoded atoms.

Propositional Modal Logic Programs [32] Construction of T_P-neural networks was exactly reproduced in this approach, with the only difference that it was adapted to Kripke semantics. That is, in each *possible world* (by Kripke) one could have a separate T_P-neural network computing classical values.

Fibrational Neural Networks Some research was done on creating networks of T_P-neural networks, they were called *Fibring Neural Networks* in [31]. Alternative definition of fibring neural networks was later given in [130]. The latter paper shows that the fibring neural networks can approximate the computation of models for normal first-order logic programs.

Many-Valued Logic Programs This approach was developed in [185, 135, 119]. The resulting neural networks did not replace binary units by fuzzy or many-valued units, as one could expect. Instead, binary units were "thought of" as representing some atom

together with its value, and several additional layers were needed in order to reflect some additional properties of many-valued semantic operators. This approach is very close to the results of Chapter 6, and we will discuss it in some detail in Section 6.4. This many-valued approach incorporated all the properties of T_P-neural networks listed in Note 5.3.2. It is curious that the construction of the 5-layer neural networks obtained in [185, 135] does not follow the general tendency to develop 3-layer neural networks, motivated in Section 5.2.1.

There have been several papers relating Fuzzy Prolog and Fuzzy Neural Networks, [38, 163, 211]. These authors call the resulting neural networks *Fuzzy-Logical Neural Networks*. The authors of [163] show that Fuzzy Logic Programs can be simulated by feedforward neural networks; moreover, they use learning algorithms when working with fuzzy numbers. This is a very interesting piece of research. This work is not extending the work of Hölldobler et al. [98, 99], and seems to be an alternative direction in Neuro-Symbolic Integration. The authors from the two alternative approaches do not cross-reference each other, from which we conclude that there is little collaboration between those who develop *T_P-neural networks* and *Fuzzy-Logical Neural Networks*. However, there is a strong resemblance in the constructions developed in both groups. Comparing with the many-valued extensions of T_P-neural networks [185, 135, 119] we have mentioned, the *Fuzzy-Logical Neural Nets* are capable of propagating fuzzy signals, and not just 0, 1. If we examine *Fuzzy-Logical Neural Networks* relative to the properties listed in Note 5.3.2, we will notice that they are not correlated with semantic operators any more, however, they still maintain the properties (3)-(4) from Note 5.3.2, in that they work with truth values and functions over truth values, and do not encode first-order atoms directly.

5.4 Learning in Neural Networks

The following four subsections support the Neurocomputing part of this Chapter. We define here the types of learning which are widely recognised and applied in Neurocomputing and which will be used for simulations of the semantic operator for BAPs and conventional SLD-resolution in neural networks in Chapters 6 and 7.

It is common to distribute various learning laws and techniques of neurocomputing into three major groups: Supervised Learning, Unsupervised Learning and Reinforcement Learning. The latter form of learning is not discussed in this thesis.

We will start with Unsupervised (Hebbian) Learning.

5.4.1 Hebbian Learning

The aim of unsupervised learning is to present an artificial neural network with raw data and allow the network to make its own representation of the data - hopefully retaining all the important information.

Unsupervised learning in artificial neural networks is generally recognised by using a form of Hebbian learning which is based on a proposal by Donald Hebb who wrote [83]: "When an axon of cell A is near enough to excite a cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells such that A's efficiency, as one of the cells firing B, is increased." Since this principle was proposed, many functions which increase and bound the growth of reaction in artificial neural networks were introduced. The main problem was to fix not only the "growth" function, but its bounds as well, because finite biological or artificial neural networks cannot present unlimited growth of reaction.

The general idea behind Hebbian learning is that positively correlated activities of two

neurons strengthen the weight of the connection between them and that uncorrelated or negatively correlated activities weaken the weight of the connection (the latter form is known as Anti-Hebbian learning).

The conventional definition of Hebbian learning is given as follows, see [82] for further details. Let k and j denote two units and w_{kj} denote a weight of the connection from j to k. We denote the value of j at time t by $v_j(t)$ and the value of k at time t by $v_k(t)$. Then the rate of change in the weight between j and k is expressed in the form

$$\Delta w_{kj}(t) = F(v_j(t), v_k(t)),$$

where F is some function. As a special case of this formula, it is common to write

$$\Delta w_{kj}(t) = \eta(v_j(t))(v_k(t)),$$

where η is a constant that determines the *rate of learning* and is positive in case of Hebbian learning and negative in case of Anti-Hebbian learning. Finally, we update the weight of the connection $w_{kj}(t+1) = w_{kj}(t) + \Delta w_{kj}(t)$ at the next step of the computation.

Example 5.4.1. *Consider the neural network from Example 5.2.1. The Hebbian learning functions described above can be used to train the weight w_{kj}.*

5.4.2 Error-Correction Learning

We will use the algorithm of error-correction learning to simulate the process of unification described in the previous section.

Error-correction learning is one of the algorithms among the paradigms which advocate *supervised learning*. Supervised learning is the most popular type of learning implemented in artificial neural networks, and we give a brief sketch of error-correction algorithm in this subsection; see, for example, [82] for further details.

Let $d_k(t)$ denote some *desired response* for unit k at time t. Let the corresponding value of the *actual response* be denoted by $v_k(t)$. The response $v_k(t)$ is produced by a *stimulus* (vector) $v_j(t)$ applied to the input of the network in which the unit k is embedded. The input vector $v_k(t)$ and desired response $d_k(t)$ for unit k constitute a particular *example* presented to the network at time t. It is assumed that this example and all other examples presented to the network are generated by an environment. We define an *error signal* as the difference between the desired response $d_k(t)$ and the actual response $v_k(t)$ by $e_k(t) = d_k(t) - v_k(t)$.

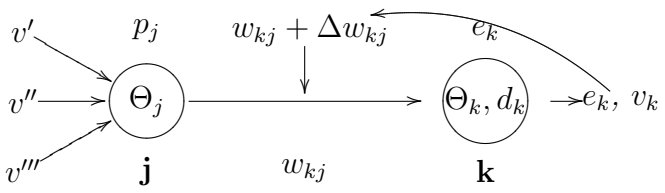
The *error-correction learning rule* is the adjustment $\Delta w_{kj}(t)$ made to the weight w_{kj} at time n and is given by

$$\Delta w_{kj}(t) = \eta e_k(t) v_j(t),$$

where η is a positive constant that determines the rate of learning.

Finally, the formula $w_{kj}(t+1) = w_{kj}(t) + \Delta w_{kj}(t)$ is used to compute the updated value $w_{kj}(t+1)$ of the weight w_{kj}. We use formulae defining v_k and p_k as in Section 5.2.

Example 5.4.2. *The neural network from Example 5.2.1 can be transformed into an error-correction learning neural network as follows. We introduce the* desired response *value d_k into the unit k, and the error signal e_k computed using d_k must be sent to the connection between j and k to adjust w_{kj}.*



5.4.3 Filter Learning and Grossberg's Law

Filter learning, similar to Hebbian learning, is a form of unsupervised learning, see [85] for further details. Filter learning and in particular Grossberg's law will be used in simulations of SLD-resolution at stages when a network, acting in accordance with the SLD-resolution algorithm, must choose, for each atom in the goal, with which particular clause in the program it will be unified at the next step.

Consider the situation when a unit receives multiple input signals, $v_1, v_2, \ldots, v_n$, with v_n a distinguished signal. In Grossberg's original neurobiological model [71], the v_i, $i \neq n$, were thought of as "conditioned stimuli" and the signal v_n was an "unconditioned stimulus". Grossberg assumed that v_i, $i \neq n$ was 0 most of the time and took large positive value when it became active.

Choose some unit c with incoming signals $v_1, v_2, \ldots, v_n$. Grossberg's law is expressed by the equation

$$w_{ci}^{\text{new}} = w_{ci}^{\text{old}} + a[v_i v_n - w_{ci}^{\text{old}}]U(v_i), (i \in \{1, \ldots, n-1\}),$$

where $0 \leq a \leq 1$ and where $U(v_i) = 1$ if $v_i > 0$ and $U(v_i) = 0$ otherwise.

Example 5.4.3. *Consider the unit j from Example 5.2.1. It receives multiple input values, and this means that we can apply Grossberg's learning law to the weights connecting v', v'' and v''' with j.*

In the next section we will also use *the inverse form of Ginsberg's law* and apply the equation

$$w_{ic}^{\text{new}} = w_{ic}(t)^{\text{old}} + a[v_i v_n - w_{ic}^{\text{old}}]U(v_i), (i \in \{1, \ldots, n-1\})$$

to enable (unsupervised) change of weights of the connections going from some unit c

which sends outcoming signals $v_1, v_2, \ldots v_n$ to units $1, \ldots, n$ respectively. This will enable outcoming signals of one unit to compete with each other.

5.4.4 Competitive Learning and Kohonen's Layer

Competitive approach to building neural networks is often seen as an alternative to the Hebbian model of learning. In books of Edelman [42], it was suggested that rather than having rules of Hebbian type, a form of "Darwinian" principle operates within the brain, enabling it to improve its performance continually by means of a kind of natural selection principle that governs these connections. Edelman and his colleagues have constructed series of computationally controlled devices (DARWIN I, II, III IV) that are intended to simulate the procedures that he is proposing lie at the basis of mental action.

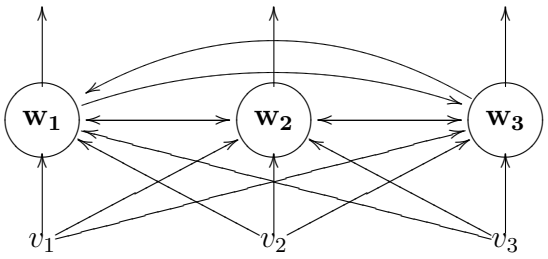
In this thesis, we will use Kohonen's [113] type of competitive learning. The definition of a *Kohonen Layer* was introduced in conjunction with the famous *Kohonen learning law* [113], and both notions are usually thought of as fine examples of *competitive learning*. We are not going to use the latter notion in this thesis and will concentrate on the notion of *Kohonen layer* which is defined as follows.

The Kohonen layer consists of N units, each receiving n input signals $v_1, \ldots, v_n$ from another layer of units. The v_j input signal to a Kohonen unit i has a weight w_{ij} assigned to it. We denote by $\mathbf{w_i}$ the vector of weights $(w_{i1}, \ldots, w_{in})$, and we use $\mathbf{v}$ to denote the vector of input signals $(v_1, \ldots, v_n)$.

Each Kohonen unit i calculates its *input intensity* I_i in accordance with the following formula: $I_i = D(\mathbf{w_i}, \mathbf{v})$, where $D(\mathbf{w_i}, \mathbf{v})$ is the distance measurement function. The common choice for $D(\mathbf{w_i}, \mathbf{v})$ is the Euclidian distance $D(\mathbf{w_i}, \mathbf{v}) = |w_i - v|$. We will use a constant function to compute I_i in the proof of Theorem 7.4.1.

Once each Kohonen unit has calculated its input intensity I_i, a competition takes place to see which unit has the smallest input intensity. Once the winning Kohonen unit is determined, its output v_i is set to 1. All the other Kohonen unit output signals are set to 0.

Example 5.4.4. *The neural network below consists of three units organised in one layer; the competition according to the algorithm described above will take place between those three units, and only one of them will eventually emit the signal.*



5.5 Conclusions

We have discussed the scope of problems being solved in Neurocomputing, Connectionism, and Neuro-Symbolic Integration. We have given definitions of a neuron and an artificial neural network. We motivated why the use of the 3-layer architecture is preferable.

We then split the background material into two parts. In the first part we described the T_P-neural networks of [104, 98, 99]. We gave analysis of their main features, and listed the range of problems that occur when constructing T_P-neural networks. We outlined some of the extensions of T_P-neural networks that have been obtained so far.

We have given motivation leading to establishing the results of Chapters 6 and 7.

In the second part, we described and illustrated the learning techniques used in Neurocomputing. We are going to use them in the subsequent chapters.

Chapter 6

$\mathcal{T}_P$-Neural Networks for BAPs

6.1 Introduction

This chapter supports the extensive approach to developing the T_P-Neural Networks of [98, 99, 104]. That is, we take Bilattice-Based Annotated Logic Programs as developed in Chapters 3 and 4, and we extend T_P-neural networks in order to simulate the semantic operator $\mathcal{T}_P$ for BAPs. To reflect the strong relation of the new neural networks we build to T_P-neural networks, we will call them $\mathcal{T}_P$-neural networks. As we have shown in Chapter 4, $\mathcal{T}_P$ incorporated some additional conditions comparing with conventional T_P. This is why the $\mathcal{T}_P$-neural networks had to be enriched with some additional functions. Similarly to all the extensive developments of the T_P-networks that we outlined in 5.3.1, the $\mathcal{T}_P$-neural networks incorporate all the main properties that T_P-neural networks possess: see items (1)-(4) in Note 5.3.2.

As we hoped, $\mathcal{T}_P$-neural networks use learning in order to compute $\mathcal{T}_P$. This allows us to introduce hypothetical and uncertain reasoning expressed by BAPs into the framework of neural-symbolic computation. Moreover, it allows us to claim that learning is required for such a task. The form of learning that we use can be seen as a form of unsupervised

Hebbian learning which is widely implemented in Neurocomputing, its definition is given in Section 5.4.

We show in Section 6.4 that the neural networks of [185, 135] that are capable of computing semantic operators of various annotation-free many-valued logic programs, compensate the lack of learning by two additional layers. The neural networks with two additional layers require more time and space resources comparing with T_P- and $\mathcal{T}_P$-neural networks. We show that $\mathcal{T}_P$-neural networks can compute all that the networks of [185, 135] can, but improve time and space complexity of computations. This shows that learning techniques are extremely effective, not only in Neurocomputing, but also in the field of Neuro-Symbolic Integration.

The structure of this Chapter is as follows. In Section 6.2, we construct the neural networks computing the $\mathcal{T}_P$-operator for propositional BAPs. In Section 6.3, we show how the results can be extended to first-order logic programs, and apply the approximation techniques of [184] to $\mathcal{T}_P$-neural networks. In Section 6.4 we relate our results with those of [185, 135], and in Section 6.5, we conclude the discussion.

The construction of $\mathcal{T}_P$-neural networks was first presented in [127, 120, 119], and the approximation results of Section 6.3 were published in [127, 120]. Section 6.4 about the relations between the two types of neural networks computing the semantic operators of many-valued logic programs appeared in [119].

6.2 $\mathcal{T}_P$-Neural Networks for BAPs

In this Section, we prove a theorem establishing that the semantic operator $\mathcal{T}_P$ for BAPs with no function symbols occurring in either individual or annotation terms can be simulated by three-layer learning neural networks built in the style of [98]. (Since the annotation Herbrand base for these programs is finite, they are computationally equivalent to propositional BAPs with no functions allowed in the annotations.)

The resulting neural networks employ the Hebbian learning functions defined in Section 5.4.1. We shall compare the use of learning functions in $\mathcal{T}_P$-neural networks with the conventional use of Hebbian learning. But first we will give technical details, as follows.

Theorem 6.2.1. *Let P be a function-free BAP. Then there exists a 3-layer recurrent learning neural network which computes $\mathcal{T}_P$.*

Proof. Let m and n be the number of strictly ground annotated atoms from the annotation Herbrand base B_P and the number of clauses occurring in P respectively. Without loss of generality, we may assume that the annotated atoms are ordered. The network associated with P can now be constructed by the following translation algorithm.

1. The input and output layers are vectors of binary threshold units of length k, $1 \leq k \leq m$, where the i-th unit in the input and output layers represents the i-th strictly ground annotated atom. The threshold of each unit occurring in the input or output layer is set to 0.5.

2. For each clause of the form $A : (\alpha, \beta) \leftarrow B_1 : (\alpha_1, \beta_1), \ldots, B_l : (\alpha_l, \beta_l)$, $l \geq 0$, in P do the following.

   2.1 Add a binary threshold unit c to the hidden layer.

2.2 Connect c to the unit representing $A : (\alpha, \beta)$ in the output layer with weight 1. We will call connections of this type 1-*connections*.

2.3 For each atom $B_j : (\alpha_j, \beta_j)$ in the input layer, connect the unit representing $B_j : (\alpha_j, \beta_j)$ to c and set the weight to 1. (We will call these connections 1-connections also.)

2.4 Set the threshold Θ_c of c to $l - 0.5$, where l is the number of atoms in $B_1 : (\alpha_1, \beta_1), \ldots, B_l : (\alpha_l, \beta_l)$.

3. For each annotated atom $A : (\alpha, \beta)$, connect the unit o representing $A : (\alpha, \beta)$ in the output layer to the unit i representing it in the input layer via weight 1; and in the other direction, connect i to o via a connection with weight 1. These are recurrent connections. Set all the weights which are not covered by these rules to 0.

4. For the input units $i_f, \ldots i_j$ and the output units $o_f, \ldots, o_j$ representing some atoms $A_f : (\alpha_f, \beta_f)$ and $A_j : (\alpha_j, \beta_j)$, with $A_i = A_j$, do the following. Connect each of the output units $o_f, \ldots, o_j$ with each of the input units $i_f, \ldots i_j$ via a $\oplus$-connection. These connections have initial weights 0, but a learning function ϕ is embedded into them.

The function ϕ is embedded only into $\oplus$-connections, as follows.

Let o^* be the output unit representing an annotated atom $A : (\alpha_i, \beta_i)$, and v_{o^*} be its value. Let v_{i^*} be the value of the input unit i^* representing the atom $A : (\alpha_i, \beta_i)$ in the input layer; o^* is connected to i^* via a 1-connection, by item 3. To each $\oplus$-connection from o^* to each input unit i representing some $A : (\alpha_d, \beta_d)$ in the input layer, apply $\phi = \Delta w_{io^*}(t+1) = (v_{o^*}(t))(v_{i^*}(t+1))$ to change w_{io^*} at time $t+1$ using the formula $w_{io^*}(t+1) = w_{io^*}(t) + \Delta w_{io^*}(t+1)$, if the following conditions are satisfied.

★ there are output units $o_j, \ldots, o_k$ representing annotated atoms $A_j : (\alpha_j, \beta_j), \ldots, A_k : (\alpha_k, \beta_k)$, with $A = A_j = \ldots = A_k$,

★★ $(\alpha_d, \beta_d) \leq_k (\alpha_i, \beta_i) \oplus (\alpha_j, \beta_j) \oplus \ldots \oplus (\alpha_k, \beta_k)$;

★★★ these output units $o^*, o_j, \ldots, o_k$ become activated at time t.

When ϕ is computed, it will always give $\Delta w_{io^*}(t) = 1$. Indeed, $(v_{o^*}(t)) = 1$, because of the Item ★★★ in the list of conditions above. Because o^* and i^* are connected via a 1-connection, $(v_{i^*}(t+1)) = 1$.

Note that the three conditions above cover the case when $A : (\alpha_i, \beta_i) = A_j : (\alpha_j, \beta_j) = \ldots = A_k : (\alpha_k, \beta_k)$, that is, when only the unit representing $A : (\alpha_i, \beta_i)$ is excited, and $(\alpha_d, \beta_d) \leq_k (\alpha_i, \beta_i)$. That is, the three conditions above guarantee that all the units representing atoms with lower annotations than $A : (\alpha_i, \beta_i)$ will be activated, if the unit representing $A : (\alpha_i, \beta_i)$ is activated.

Note 6.2.1. *In order to guarantee that all the new signals obtained via the function ϕ will be read by an external recipient at the output layer, we require each input unit representing some atom $A : (\alpha, \beta)$ to be connected to the output unit representing $A : (\alpha, \beta)$ via a* 1*-connection; see item 3.*

Each annotation Herbrand interpretation HI for P can be represented by a binary vector $(v_1, \ldots, v_m)$. Such an interpretation is given as an input to the network by externally activating corresponding units of the input layer at time $t = 0$. It remains to show that $A : (\alpha, \beta) \in \mathcal{T}_P \uparrow n$ for some n if and only if the output unit representing $A : (\alpha, \beta)$ becomes active at time t, for some non-zero t.

The proof that this is so proceeds by routine induction, as follows.

Subproof 1. We prove that if some annotated atom $A : (\mu, \nu) \in \mathcal{T}_P \uparrow n$, then it is computed by a neural network described in the construction of Theorem 6.2.1 at some non-zero time t.

Basic step. Suppose $A : (\mu, \nu) \in \mathcal{T}_P \uparrow 1$. Then, by construction of the neural networks, there will be an output unit i representing $A : (\mu, \nu)$, and it is connected via a 1-connection to some hidden unit c having threshold -0.5. But then, the hidden unit c will emit the signal 1 at time $t = 0$. This means, by construction of the neural networks, and by the fact that $\Theta_i = 0.5$, that the output unit i will emit a signal 1 at time $t = 1$.

Induction Hypothesis. Suppose the statement of the Subproof 1 holds for all formulae in $\mathcal{T}_P \uparrow (n-1)$.

If $A : (\alpha, \beta) \in \mathcal{T}_P \uparrow n$, then one of the following holds:

i. there is a clause $A : (\alpha, \beta) \leftarrow B_1 : (\alpha_1, \beta_1), \ldots, B_l : (\alpha_l, \beta_l)$ in ground(P) such that $\{B_1 : (\alpha'_1, \beta'_1), \ldots, B_l : (\alpha'_l, \beta'_l)\} \subseteq \mathcal{T}_P \uparrow (n-1)$ and for each (α'_i, β'_i) $(i = 1, \ldots, l)$,

$$(\alpha_i, \beta_i) \leq_k (\alpha'_i, \beta'_i).$$

ii. there are annotated strictly ground atoms $A : (\alpha^*_1, \beta^*_1), \ldots, A : (\alpha^*_l, \beta^*_l) \in \mathcal{T}_P \uparrow (n-1)$ such that $(\alpha, \beta) \leq_k (\alpha^*_1, \beta^*_1) \oplus \ldots \oplus (\alpha^*_l, \beta^*_l)$.

Consider the case i. Let c be the unit in the hidden layer associated with this clause according to item 2.1 of the construction in the translation algorithm in the proof of this theorem.

The part of the proof for i. corresponds to the proof of the similar statement by Hölldobler and al., [98, 99]. The argument is as follows. By the conditions of i above and item [2.1] in the construction of these neural networks, we conclude that there exists

a hidden unit c associated with the clause $A : (\alpha, \beta) \leftarrow B_1 : (\alpha_1, \beta_1), \ldots, B_l : (\alpha_l, \beta_l)$ in ground(P). According to item [2.2] of the construction of a $\mathcal{T}_P$-neural network, the unit c is connected to the unit representing $A : (\alpha, \beta)$ in the output layer by a 1-connection. By item [2.3] we conclude that c is connected via a 1-connection with the units representing $B_1 : (\alpha_1, \beta_1), \ldots, B_l : (\alpha_l, \beta_l)$ in the input layer.

Because $\{B_1 : (\alpha'_1, \beta'_1), \ldots, B_l : (\alpha'_l, \beta'_l)\} \subseteq \mathcal{T}_P \uparrow (n-1)$, we conclude, by the induction hypothesis, that the units representing $B_1 : (\alpha'_1, \beta'_1), \ldots, B_l : (\alpha'_l, \beta'_l)$ became activated in the output layer at some time t. But then, by definition of $\oplus$-connections, items $\bigstar$ – $\bigstar\bigstar\bigstar$, the input units representing $B_1 : (\alpha_1, \beta_1), \ldots, B_l : (\alpha_l, \beta_l)$ will be activated at time $t + 1$.

From this we conclude that c becomes active at time $t + 2$. Consequently, [2.2] and the fact that units occurring in the output layer have thresholds 0.5 (see Item 1) ensure that the unit representing $A : (\alpha, \beta)$ in the output layer becomes active at time $t + 3$.

Consider the case ii. Since $A : (\alpha^*_1, \beta^*_1), \ldots, A : (\alpha^*_l, \beta^*_l) \in \mathcal{T}_P \uparrow (n-1)$, there are units representing these annotated strictly ground atoms in the input and output layers, and, by the induction hypothesis, these units in the output layer became active at time t. Moreover, according to item 4 in the description of the neural network architecture, there are $\oplus$ -connections between these output and input units. This is why we can conclude that ϕ is activated at time t, and therefore the input unit representing $A : (\alpha, \beta)$ receives the signal 1 in the input layer at time $t + 1$. Then, according to Note 6.2.1 and item 3, at time $t + 3$ this signal is read from the output unit representing $A : (\alpha, \beta)$.

Subproof 2. We prove that, if the neural network described in the construction of the current proof computes the value 1 for some annotated atom $A : (\alpha, \beta)$ at time t, then $A : (\alpha, \beta) \in \text{lfp}(\mathcal{T}_P)$.

Basic step. Suppose the output unit o representing $A : (\alpha, \beta)$ emits the signal 1 at time $t = 1$. This is possible only if there exists a hidden unit c that sends a signal to o at time $t = 0$. By construction of the neural network, this is possible only if c is not connected to any input unit, and its threshold is equal to -0.5. From this we conclude that c represents the unit clause $A : (\alpha, \beta) \leftarrow$. But then, by definition of $\mathcal{T}_P$, $A : (\alpha, \beta) \in \mathrm{lfp}(\mathcal{T}_P)$.

Induction hypothesis. We assume that the statement of the Subproof 2 holds for the iterations performed at times $t' < t$.

Suppose that the unit representing the annotated atom $A : (\alpha, \beta)$ in the output layer becomes active at time t. From the construction of the network, we have two situations when this can happen because the signal could come either from a regular hidden unit or directly from an input unit as described in Note 6.2.1.

(1) Consider the situation when the signal comes from a regular hidden unit c which became active at time $t-1$ and c is connected to l units in the input layer and has threshold $l - 0.5$. By construction of these neural networks, this hidden unit c is associated with some clause $A : (\alpha, \beta) \leftarrow B_1 : (\alpha_1, \beta_1), \ldots, B_l : (\alpha_l, \beta_l)$.

The hidden unit c could be activated at time $t - 1$ only if each of these l input units was activated at time $t - 2$. By construction of these neural networks, these l input units represent strictly ground annotated atoms from the body of the clause $A : (\alpha, \beta) \leftarrow B_1 : (\alpha_1, \beta_1), \ldots, B_l : (\alpha_l, \beta_l)$. Because the l input units representing $B_1 : (\alpha_1, \beta_1), \ldots, B_l : (\alpha_l, \beta_l)$ were activated at time $t - 2$, then we conclude that the output units representing these atoms or some atoms $B_1 : (\alpha'_1, \beta'_1), \ldots, B_l : (\alpha'_l, \beta'_l)$, where $(\alpha_1, \beta_1) \leq_k (\alpha'_1, \beta'_1), \ldots, (\alpha_l, \beta_l) \leq_k (\alpha'_l, \beta'_l)$, were activated at time $t - 3$ at the output layer. By the induction hypothesis, $(\alpha'_1, \beta'_1), \ldots, (\alpha'_l, \beta'_l) \in \mathrm{lfp}(\mathcal{T}_P)$. Hence, we have found a strictly ground clause $A : (\alpha, \beta) \leftarrow B_1 : (\alpha_1, \beta_1), \ldots, B_l : (\alpha_l, \beta_l)$ such that for all

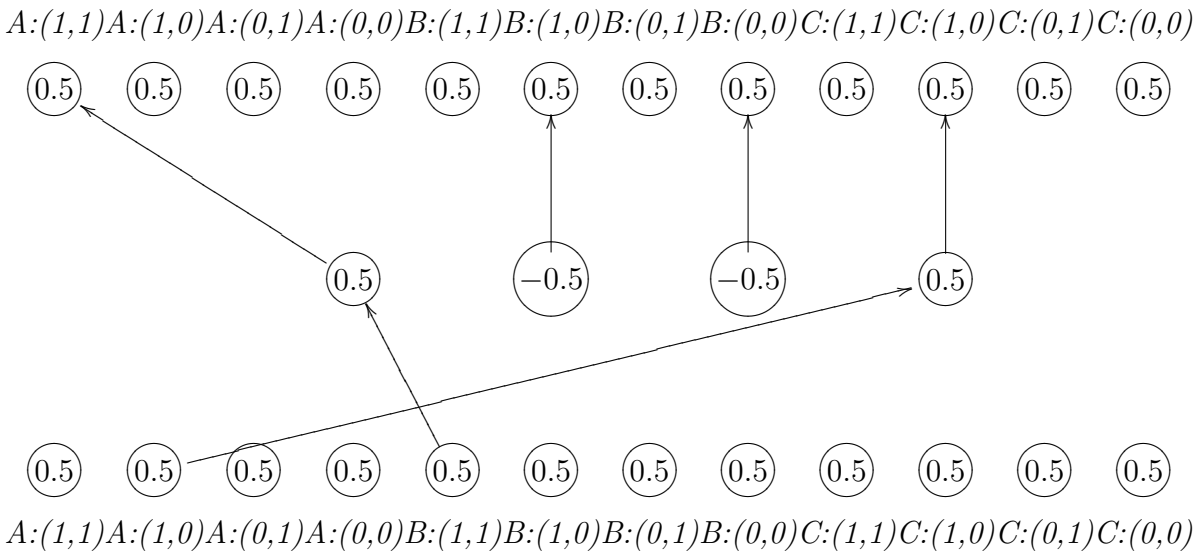
$i \in \{1, \ldots, l\}$ we have $B_i : (\alpha_i', \beta_i') \in \mathcal{T}_P \uparrow (n-1)$, and consequently, by Definition of $\mathcal{T}_P$, $A : (\alpha, \beta) \in \mathcal{T}_P \uparrow n$.

(2) Suppose the output unit representing $A : (\alpha, \beta)$ is activated via 1-connection at time t as described in Note 6.2.1. This means that the learning function ϕ is activated at time $t-2$ via the $\oplus$ -connection between the output and input layers, and at time $t-1$, the input unit representing $A : (\alpha, \beta)$ emitted the signal. Then, by the construction of the network (see item 4) and the definition of ϕ in the proof of Theorem 6.2.1), there are units representing annotated atoms $A : (\alpha_i^*, \beta_i^*), \ldots, A : (\alpha_l^*, \beta_l^*)$ in the output layer such that they are connected to the input unit representing $A : (\alpha, \beta)$ via $\oplus$ -connections, they became active at time $t-3$, and $(\alpha, \beta) \leq_k (\alpha_i^*, \beta_i^*) \oplus \ldots \oplus (\alpha_k^*, \beta_k^*)$. By induction hypothesis, this means that $A : (\alpha_i^*, \beta_i^*), \ldots, A : (\alpha_k^*, \beta_k^*) \in \mathcal{T}_P \uparrow (n-1)$. Then, according to the definition of $\mathcal{T}_P$ (see Definition 4.2.6, item 2), $A : (\alpha, \beta) \in \mathcal{T}_P \uparrow n$.

This completes the proof. □

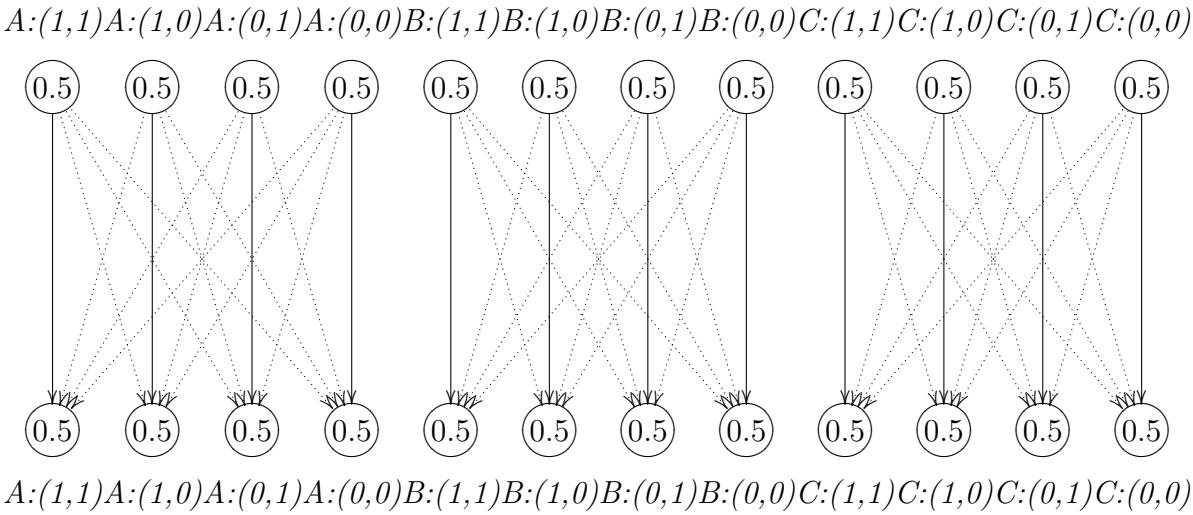
Example 6.2.1. *The following diagram displays the neural network which computes* $\mathcal{T}_P \uparrow 3$ *from Example 4.2.4. Without the function* ϕ*, the neural network will compute only* $\mathcal{T}_P \uparrow 1 = \{B : (0,1), B : (1,0)\}$*, explicit logical consequences of the program. But it is the use of* ϕ *that allows the neural network to compute* $\mathcal{T}_P \uparrow 3$*. Note that arrows* $\longrightarrow$ *,* $\cdots\cdots\succ$ *denote respectively* 1*-connections and* $\oplus$*-connections, and we have marked by* ϕ *the connections which are activated by the learning function. According to the conventional definition of recurrent neural networks, each output unit denoting some atom is in its turn connected to the input unit denoting the same atom via a* 1*-connection and this forms a loop. We assume but do not draw these connections in the next diagram. (The diagram below will be followed by a thorough illustration of the recurrent connections in*

this network.)



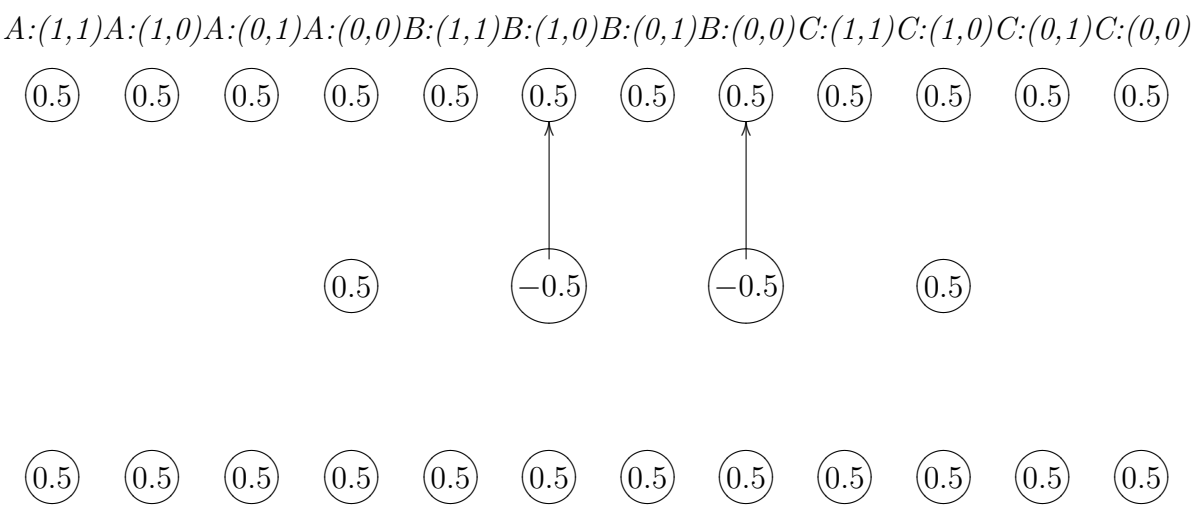
The arrows $\longrightarrow$ *reflect the structure of the clauses from Example 3.5.1, similarly to* T_P*-neural networks of Section 5.3.*

We will use the next diagram to illustrate the recurrent connections between output and input layers in the network we have drawn above. The arrows $\cdots\!\!\succ$ *represent* $\oplus$*-connection between the output and input neurons representing annotated atoms with the same first-order part.*

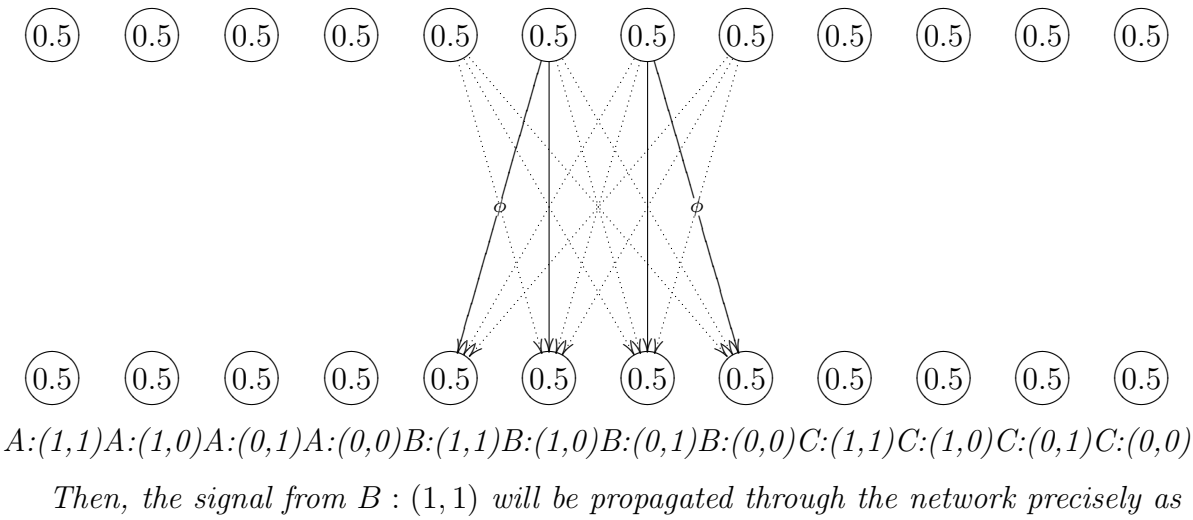


And thus units representing one and the same first-order atom form a single multineuron. And we may as well say that the neural network above consists of three multineurons A, B *and* C.

Computations in this neural network bear a strong resemblance to those of Example 5.3.1. First inputs representing $B : (1, 0)$ *and* $B : (0, 1)$ *will be excited. Then they will send signals to the input layer:*

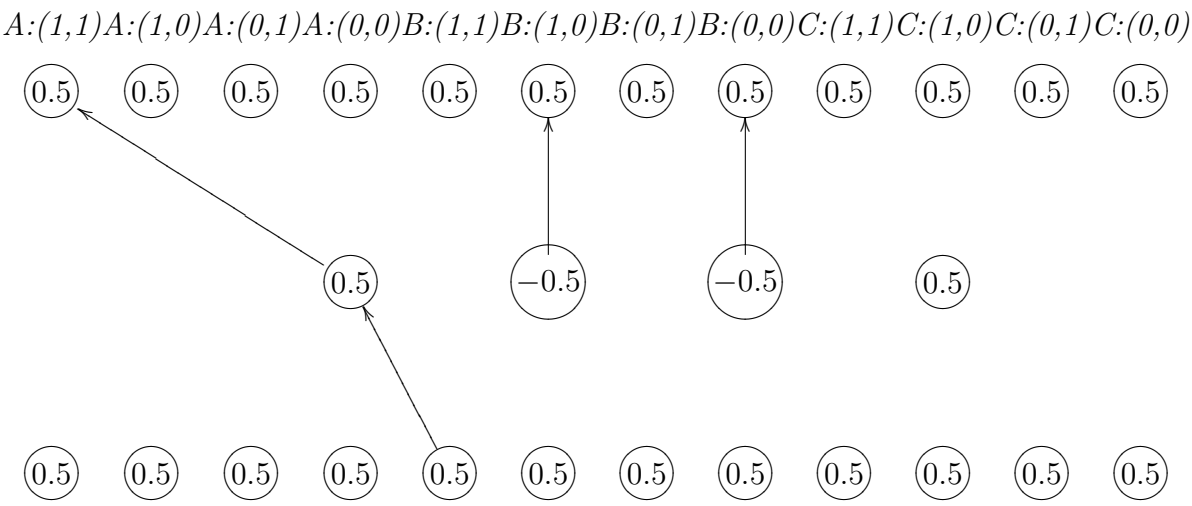


At this stage, the learning function ϕ will be activated, and the unit representing $B : (1, 1)$ and $B : (0, 0)$ in the input layer of multineuron B will become excited at the next iteration.



Then, the signal from $B : (1, 1)$ will be propagated through the network precisely as it would be handled in the T_P-neural network. Finally, the unit representing $A : (1, 1)$ in the

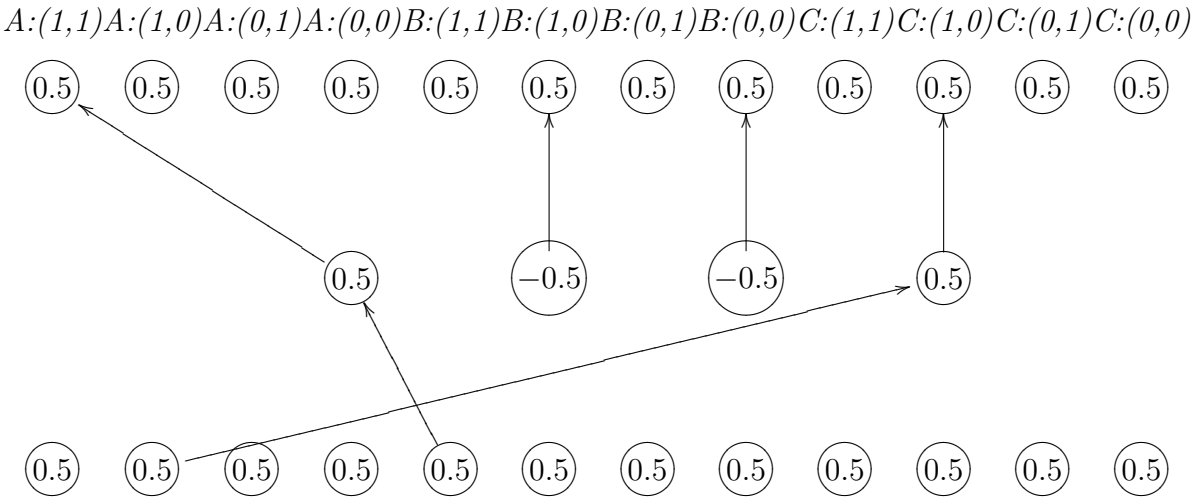
output layer will become excited:



Next, the signal from the unit representing $A : (1, 1)$ *in the output layer, sends its signal to the input layer via a* 1*-connection. But also, by definition of* $\oplus$*-connections and* ϕ*, the unit representing* $A : (1, 1)$ *will activate all the units representing atoms with lower annotations:*

Next, the unit representing $A : (1, 1)$ *in the input layer is not connected to any hidden*

unit, but the unit representing $A : (1, 0)$ *is. And so, this unit further participates in computations.*



The direct signals from input units representing all the annotated atoms $A : (1, 1), \ldots, A : (0, 0)$ *are sent to the output layer, but we omitted these connections on the digaram above.*

And so finally, the last logical consequence $C(1, 0)$ *of the given logic program is computed. In two more steps, the output layer will activate the unit representing* $C : (0, 0)$.

We can make several conclusions from the construction of Theorem 6.2.1.

- Neurons representing annotated atoms with the identical first-order (or propositional) component are joined to multineurons in which units are correlated using $\oplus$-connections.

- The learning function ϕ roughly corresponds to Hebbian learning defined in Section 5.4.1, with the rate of learning $\eta = 1$. This function is used to train the multineurons: if a multineuron in the output layer repeatedly excites the corresponding input

multineuron, then the weights of the connections between the two multineurons increase.

- The main problem Hebbian learning causes is that the weights of connections with embedded learning functions tend to grow exponentially, which cannot fit into the model of biological neurons. This is why traditionally some functions are introduced to bound the growth. In the neural networks we have built some of the weights may grow with iterations, but the growth will always be bounded by the activation functions, namely, binary threshold functions used in the computation of each v_i.

6.3 Approximation of $\mathcal{T}_P$-Neural Networks in the First-Order case

Since the neural networks of [99] were proven to compute the least fixed points of the semantic operator defined for propositional logic programs, many attempts have been made to extend this result to first-order logic programs. In this section we will briefly describe the essence of the approximation result of [184], but we apply it to BAPs. Some alternative results to the approximation theorem of [184] appeared in [94, 10]. But we chose [184] to illustrate the general idea behind the approximation.

We consulted [208] for basic topological definitions.

We first give some technical details, and then we will summarise the conclusions to be made from the approximation theorem.

Some of the following definitions are due to Fitting [53], others can be found in [184]. Let $l : B_P \to \mathbb{N}$ be a *level mapping* with the property that, given $n \in \mathbb{N}$, we can effectively find the set of all $A : (\tau) \in B_P$ such that $l(A : (\tau)) = n$.

Such a mapping always exists for $\mathcal{L}$, and for example l may be defined by taking $l(A : (\tau))$ be the depth of $A : (\tau)$, i.e., the number of brackets used in forming $A : (\tau)$ according to the inductive definition of an annotated formula. Notice that in this case each of the sets $l^{(-1)}(n)$ is finite. If $l(A : (\tau)) = n$, we say that n is a level of $A : (\tau)$.

Definition 6.3.1 ([53]). *Let* $\mathrm{HI}_{P,\mathbf{B}}$ *be the set of all interpretations* $HI : B_P \to \{\langle 0,1\rangle, \langle 1,0\rangle\}$. *We define the ultrametric* $d : \mathrm{HI}_{P,\mathbf{B}} \times \mathrm{HI}_{P,\mathbf{B}} \to \mathbb{R}$ *as follows: if* $\mathrm{HI}_1 = \mathrm{HI}_2$, *we set* $d(\mathrm{HI}_1, \mathrm{HI}_2) = 0$, *and if* $\mathrm{HI}_1 \neq \mathrm{HI}_2$, *we set* $d(\mathrm{HI}_1, \mathrm{HI}_2) = 2^{-n}$, *where* n *is such that* HI_1 *and* HI_2 *differ on some ground atom of level* n *and agree on all atoms of level less then* n.

We have fixed an interpretation HI from elements of the Herbrand base for a given program P to the set of values $\{\langle 1,0\rangle, \langle 0,1\rangle\}$. We assume further that $\langle 1,0\rangle$ is encoded by 1 and $\langle 0,1\rangle$ is encoded by 0. $HI_{P,\mathbf{B}}$ denotes the set of all such interpretations, and, as in the previous section, we work with the semantic operator $\mathcal{T}_P$ from Definition 4.2.6.

Let $\mathcal{F}$ denote a 3-layer feedforward learning neural network with m units in the input and output layers. The input-output mapping $f_{\mathcal{F}}$ is a mapping $f_{\mathcal{F}} : HI_{P,\mathbf{B}} \to HI_{P,\mathbf{B}}$ defined as follows. Given $HI \in HI_{P,\mathbf{B}}$, we present the vector $(HI(B_1 : (\alpha_1, \beta_1)), \ldots, HI(B_m : (\alpha_m, \beta_m)))$, to the input layer; after propagation through the network, we determine $f_{\mathcal{F}}(HI)$ by taking the value of $f_{\mathcal{F}}(HI)(A_j : (\alpha_j, \beta_j))$ to be the value in the jth unit in the output layer, $j = 1, \ldots, m$, and taking all other values of $f_{\mathcal{F}}(HI)(A_j : (\alpha_j, \beta_j))$ to be 0.

Suppose that M is a fixed point of $\mathcal{T}_P$. Following [184], we say that a family $\mathcal{F} = \{\mathcal{F}_i : i \in \mathcal{I}\}$ of 3-layer feedforward learning networks $\mathcal{F}_i$ computes M if there exists $HI \in HI_P$ such that the following holds: given any $\varepsilon > 0$, there is an index $i \in \mathcal{I}$ and a natural number m_i such that for all $m \geq m_i$ we have $d(f_i^m(HI), M) < \varepsilon$, where f_i denotes $f_{\mathcal{F}_i}$ and $f_i^m(HI)$ denotes the mth iterate of f_i applied to HI.

The following Theorem and its proof effectively reproduce the statement and proof

of the analogous Theorem in [184]. Some minor modifications were needed in order to reflect the additional properties of $\mathcal{T}_P$.

Theorem 6.3.1. *Let P be a BAP, let* HI *denote the least fixed point of $\mathcal{T}_P$ and suppose that we are given $\varepsilon > 0$. Then there exists a finite program $\overline{P} = \overline{P}(\varepsilon)$ (a finite subset of ground(P)) such that $d(\overline{\mathrm{HI}}, \mathrm{HI}) < \varepsilon$, where $\overline{\mathrm{HI}}$ denotes the least fixed point of $\mathcal{T}_{\overline{P}}$. Therefore, the family $\{\mathcal{F}_n | n \in \mathbb{N}\}$ computes* HI*, where $\mathcal{F}_n$ denotes the neural network obtained by applying the algorithm of Theorem 6.2.1 to P_n, and P_n denotes $\overline{P}(\varepsilon)$ with ε taken as 2^{-n} for $n = 1, 2, 3, \ldots$.*

Proof. Since $\mathcal{T}_P$ is continuous, we have
$\mathcal{T}_P \uparrow 0 \subseteq \mathcal{T}_P \uparrow 1 \subseteq \ldots \subseteq \mathcal{T}_P \uparrow n \subseteq \ldots \subseteq \mathrm{HI} = \bigcup_{n=1}^{\infty} \mathcal{T}_P \uparrow n$,
where $\mathcal{T}_P \uparrow n$ denotes the n-th upward power of $\mathcal{T}_P$.

Choose $n \in \mathbb{N}$ so large that $2^{-n} < \varepsilon$. Then there are finitely many atoms $A_1 : (\tau_1), \ldots, A_m : (\tau_m) \in \mathrm{HI}$ with $l(A_g : (\tau_g)) \leq n$ for $g = 1, \ldots, m$ and, by directedness, there is $k \in \mathbb{N}$ such that $A_1 : (\tau_1), \ldots, A_m : (\tau_m) \in \mathcal{T}_P \uparrow k$ (and hence $d(\mathcal{T}_P \uparrow k, \mathrm{HI}) \leq 2^{-n} < \varepsilon$). Consider the atom $A_g : (\tau_g)$, where $1 \leq g \leq m$, and the following sequence of steps.

Step 1. We have $A_g : (\tau_g) \in \mathcal{T}_P \uparrow k = \mathcal{T}_P(\mathcal{T}_P \uparrow (k-1))$. Therefore, one of the following holds:

1. there is a clause

$$A_g : (\tau_g) \leftarrow A_g^1 : (\tau_g^1)(1), \ldots, A_g^{m(g)} : (\tau_g^{m(g)})(1)$$

   in ground(P) such that $A_g^1 : ((\tau_g^1)')(1), \ldots, A_g^{m(g)} : ((\tau_g^{m(g)})')(1) \in \mathcal{T}_P \uparrow (k-1)$, where each of $(\tau_g^{k(g)})(1) \leq_k (\tau_g^{k(g)})'(1)$, $1 \leq k \leq m$.

2. there are atoms $A_g : (\tau_g^1)(1), \ldots, A_g : (\tau_g^{l(g)})(1) \in (\mathcal{T}_P \uparrow (k-1))$ such that $(\tau_g) \leq_k (\tau_g^1)(1) \oplus \ldots \oplus (\tau_g^{l(g)})(1)$.

Note that there may be many such clauses with head $A_g(\tau_g)$ in ground(P), we just pick one.

Step 2. Because $A_g^1 : ((\tau_g^1)')(1), \ldots, A_g^{m(g)} : ((\tau_g^{m(g)})')(1) \in \mathcal{T}_P \uparrow (k-1) = \mathcal{T}_P(\mathcal{T}_P \uparrow (k-2))$, or, alternatively, $A_g : (\tau_g^1)(1), \ldots, A_g : (\tau_g^{l(g)})(1) \in (\mathcal{T}_P \uparrow (k-1)) = \mathcal{T}_P(\mathcal{T}_P \uparrow (k-2))$, one of the following holds:

1. there are clauses in ground(P) as follows:

$$\begin{aligned}
A_g^1 : ((\tau_g^1)')(1) &\leftarrow A_{g,1}^1 : (\tau_{g,1}^1)(2), \ldots, A_{g,1}^{m(g,1)} : (\tau_{g,1}^{m(g,1)})(2) \\
A_g^2 : ((\tau_g^2)')(1) &\leftarrow A_{g,2}^1 : (\tau_{g,2}^1)(2), \ldots, A_{g,2}^{m(g,2)} : (\tau_{g,2}^{m(g,2)})(2) \\
&\vdots \leftarrow \vdots \\
A_g^{m(g)} : ((\tau_g^{m(g)})')(1) &\leftarrow A_{g,m(g)}^1 : (\tau_{g,m(g)}^1)(2), \ldots, A_{g,m(g)}^{m(g,m(g))} : (\tau_{g,m(g)}^{m(g,m(g))})(2),
\end{aligned}$$

   and, according to the definition of $\mathcal{T}_P$, for each $A_{g,m(g)}^r : (\tau_{g,m(g)}^r)(2)$ in each of the bodies, $A_{g,m(g)}^r : (\tau_{g,m(g)}^r)'(2)$, with $(\tau_{g,m(g)}^r) \leq_k (\tau_{g,m(g)}^r)'$, belongs to $\mathcal{T}_P \uparrow (k-2)$.

   Note that each $A_g^{r(g)} : (\tau_g^{r(g)})(1)$ could alternatively be obtained in the same way as described in **Step 1**, item 2, using the definition of $\mathcal{T}_P$, namely, its item number 2.

2. Or there are the following clauses in ground(P):

$$\begin{aligned}
A_g : (\tau_g^1)(1) &\leftarrow A_{g,1} : (\tau_{g,1}^1)(2), \ldots, A_{g,1} : (\tau_{g,1}^{l(g,1)})(2) \\
A_g : (\tau_g^2)(1) &\leftarrow A_{g,2} : (\tau_{g,2}^1)(2), \ldots, A_{g,2} : (\tau_{g,2}^{l(g,2)})(2) \\
&\vdots \leftarrow \vdots \\
A_g : (\tau_g^{l(g)})(1) &\leftarrow A_{g,l(g)} : (\tau_{g,l(g)}^1)(2), \ldots, A_{g,l(g)} : (\tau_{g,l(g)}^{l(g,l(g))})(2),
\end{aligned}$$

where, for each of the annotated atoms $A_{g,l(g)} : (\tau^r_{g,l(g)})(2)$ in each of the bodies, $A_{g,l(g)} : (\tau^r_{g,l(g)})'(2)$, with $(\tau^r_{g,l(g)}) \leq_k (\tau^r_{g,l(g)})'$, belongs to $\mathcal{T}_P \uparrow (k-2)$, according to the definition of $\mathcal{T}_P$.

Or each $A^{r(g)}_g : (\tau^{r(g)}_g)(1)$ is obtained in the same way as described in **Step 1**, item 2.

We continue this process till it terminates. Moreover, it is also clear that on termination the clauses produced in the last step are unit clauses. We let P_g denote the finite subset of ground(P) consisting of all the clauses produced by this process. By construction, it is clear that $\mathcal{T}_{P_g} \uparrow k$ consists of the heads of all the clauses in P_g, and is such that $\mathcal{T}_{P_g}(\mathcal{T}_{P_g} \uparrow k) = (\mathcal{T}_{P_g} \uparrow k)$, so that $\mathcal{T}_{P_g} \uparrow k$ is the least fixed point of $\mathcal{T}_{P_g}$ by Theorem 1.5.3; and $A_g : (\tau_g) \in \mathcal{T}_{P_g} \uparrow k$. Furthermore, $\mathcal{T}_{P_g} \uparrow r \subseteq \mathcal{T}_P \uparrow r$ for all $r \in N$, and, in particular, $\mathcal{T}_{P_g} \uparrow k \subseteq \mathcal{T}_P \uparrow k$.

We carry out this construction for $g = 1, \ldots, m$ to obtain programs $P_1, \ldots, P_m$ such that, for $g = 1, \ldots, m$, $\mathcal{T}_{P_g} \uparrow k$ is the least fixed point of $\mathcal{T}_{P_g}$, $A_g : (\tau_g) \in \mathcal{T}_{P_g} \uparrow k$, and $\mathcal{T}_{P_g} \uparrow r \subseteq \mathcal{T}_P \uparrow r$ for all $r \in N$. Let $\overline{P}$ denote the program $P_1 \cup \ldots \cup P_m$. Then $\overline{P}$ is a finite subprogram of ground(P), and we have $\mathcal{T}_{P_g} \uparrow k \subseteq \mathcal{T}_{\overline{P}} \uparrow k \subseteq \mathcal{T}_P \uparrow k \subseteq \mathrm{HI}$ for $g = 1, \ldots, m$. Furthermore, we also have $A_1 : (\tau_1), \ldots, A_m : (\tau_m) \in \mathcal{T}_{\overline{P}} \uparrow k$, and hence $d(\mathcal{T}_{\overline{P}} \uparrow k, \mathrm{HI}) \leq 2^{-n} \leq \varepsilon$.

We show next that $\mathcal{T}_{\overline{P}} \uparrow k$ is the least fixed point of $\mathcal{T}_{\overline{P}}$. Suppose that $A : (\tau) \in \mathcal{T}_{\overline{P}}(\mathcal{T}_{\overline{P}} \uparrow k)$. Then there is a clause $A : (\tau) \leftarrow \texttt{body}$ in $\overline{P}$ with `body` true in $\mathcal{T}_{\overline{P}} \uparrow k$. By construction, the clause $A : (\tau) \leftarrow \texttt{body}$ belongs to P_l, say, and body is true in $\mathcal{T}_{P_l} \uparrow k$. But then we have $A : (\tau) \in \mathcal{T}_{P_l}(\mathcal{T}_{P_l} \uparrow k) = \mathcal{T}_{P_l} \uparrow k = \mathcal{T}_{\overline{P}} \uparrow k$ and therefore $\mathcal{T}_{\overline{P}}(\mathcal{T}_{\overline{P}} \uparrow k) \subseteq \mathcal{T}_{\overline{P}} \uparrow k$. Conversely, suppose that $A : (\tau) \in \mathcal{T}_{\overline{P}} \uparrow k$. Then there is a clause $A : (\tau) \leftarrow \texttt{body}$ in $\overline{P}$ with body true in $\in \mathcal{T}_{\overline{P}} \uparrow (k-1)$, and then clearly `body` is true in

$\mathcal{T}_{\overline{P}} \uparrow k$ also. Hence we have $A : (\tau) \in \mathcal{T}_{\overline{P}}(\mathcal{T}_{\overline{P}} \uparrow k)$ and so $\mathcal{T}_{\overline{P}}(\mathcal{T}_{\overline{P}} \uparrow k) \subseteq \mathcal{T}_{\overline{P}} \uparrow k$, and it follows that $\mathcal{T}_{\overline{P}} \uparrow k$ is a fixed point of $\mathcal{T}_{\overline{P}}$. It follows from Theorem 1.5.3 that $\mathcal{T}_{\overline{P}} \uparrow k$ is the least fixed point of $\mathcal{T}_{\overline{P}}$. Thus, on writing $\overline{\mathrm{HI}} = \mathcal{T}_{\overline{P}} \uparrow k$ we have $d(\overline{\mathrm{HI}}, \mathrm{HI}) < \varepsilon$, where $\overline{\mathrm{HI}}$ denotes the least fixed point of $\mathcal{T}_{\overline{P}}$.

This theorem clearly contains two results corresponding to the two separate statements made in it. The first concerns finite approximation of $\mathcal{T}_P$, and is a straightforward generalisation of a theorem and proof established in [184]. The second is an immediate consequence of the first conclusion and Theorem 6.2.1. □

Thus, we have shown that the learning neural networks we have built can approximate the least fixed point of the semantic operator defined for first-order BAPs.

There are two major problems - practical and theoretical - that arise from the approximation results described above.

1. This theorem is an attempt to improve the non-constructive approximation theorems of [99, 94]. These approximation theorems were based on the non-constructive Theorem 5.2.2 of Funahashi ([184] gives a detailed analysis of this).

   Theorem 6.3.1, as well as the corresponding Theorem of [184], shows the procedure which effectively produces, for any given ε, the finite program $\overline{P}$ for which a finite $\mathcal{T}_P$-neural network can be constructed. This approach was further implemented and extended in [10, 130, 129, 127].

   There is, however, certain non-constructivism in the Theorem 6.3.1. Namely, in general case we are not given the procedure which determines, for a given program P, which ε is sufficiently small for accurate approximation of the least fixed point of $\mathcal{T}_P$. So, practically, for a given BAP P, determining the size of a finite $\mathcal{T}_P$-neural network computing accurately the least fixed point of $\mathcal{T}_P$ remains an open question.

2. To our view, there is a serious theoretical problem that arises from these approximation results. By the discussions of Section 5.1, finite, and quite small, neural networks are sufficient to simulate computations performed by Universal Turing Machines [172, 191, 109, 192, 114, 103, 190]. For any given definite logic program P and for any given BAP P, the least fixed point of the semantic operators T_P and $\mathcal{T}_P$ are Turing computable. Therefore, there must exist a finite neural network that is capable of computing the least fixed points of T_P and $\mathcal{T}_P$, and not just "approximating" them.

   Note that there is a so-called area of "Approximation" within Neurocomputing that is working on approximations of real-valued functions by finite neural networks, but not vice versa - approximations of infinite neural networks that calculate computable functions!

   One hopes that in the future, these important results of Neurocomputing will find application and further development in the field of Neuro-Symbolic Integration.

The second problem is present in all the extensions of T_P-neural networks we have discussed in Section 5.3.1. The $\mathcal{T}_P$-neural networks we have introduced in this Chapter are not an exception.

The seriousness of this problem led us to the conclusion, that perhaps the architecture of T_P-neural networks is not an optimal construction to be used in either Connectionism or in Neurocomputing. This is why, in Chapter 7, we suggest a radically new solution to the task of implementing logic programs in Neural Networks.

6.4 Relation to other Many-Valued T_P-Neural Networks

In this section we will briefly discuss the relations between $\mathcal{T}_P$-neural networks and the neural networks of [185, 135]. As we mentioned in Section 5.3.1, Seda and Lane [185, 135] proposed extensions of T_P-neural networks that are capable of simulating the semantic operator for annotation-free many-valued logic programs. In [119] we proved a theorem relating $\mathcal{T}_P$-neural networks and those of [185, 135]. One would need to supply the current chapter with all the essential definitions and constructions from [185, 135] in order to reproduce the theorem and its proof here. But we prefer not to do this.

We will, therefore, give an informal discussion of the relation between the two approaches, and we refer to [119] for the formal explanations.

In [185, 135] the neural networks simulating the semantic operator $\mathfrak{T}_P$ for annotation-free logic programs were constructed. The neural networks extended the T_P-neural networks. To emphasise this strong relation to T_P-neural networks, we will call these neural networks of [185, 135] $\mathfrak{T}_P$-neural networks.

The $\mathfrak{T}_P$ operator generalised the semantic operator of Fitting from Definition 4.4.3, in that the $\mathfrak{T}_P$ operator included new conditions comparing with Definition 4.4.3, and the new conditions worked in conjunction with the *completion* for the annotation-free logic programs we have described in Section 4.4. As we mentioned in Section 4.4, the completion guaranteed that all the logical consequences of annotation-free logic programs were computed. (Furthermore, [135] contained an algebraic reformulation of $\oplus$ and $\otimes$ in terms of semiring operations $+$ and $\times$. The latter result had no particular effect on computational properties of $\mathfrak{T}_P$ and $\mathfrak{T}_P$-neural networks, so we will not go into the

details of this result here.) $\mathfrak{T}_P$-neural networks incorporated all the properties of T_P-neural networks we listed in Note 5.3.2, but required two additional layers in order to pre-process and post-process the signals of the three-layer network, and thus to reflect the additional conditions in definition of $\mathfrak{T}_P$.

Taking into consideration the discussions of Chapter 2 and the results of Section 4.4 showing that annotation-free logic programs can be soundly translated into annotated logic programs and the semantic operators for annotation-free logic programs can be simulated by annotated semantic operators, it is natural to ask whether there is the same relation between $\mathfrak{T}_P$-neural networks and $\mathcal{T}_P$-neural networks. Because both types of neural networks reflect *ad hoc* all the properties of the semantic operators they simulate, one expects a trivial result that $\mathfrak{T}_P$-neural networks can be simulated by $\mathcal{T}_P$-neural networks. And indeed, this follows from Lemma 4.4.1, Theorem 6.2.1 and the theorem analogous to Theorem 6.2.1 proven for $\mathfrak{T}_P$-neural networks in [135, 119]. See [119] for further details.

There is one curious consequence of this relation between $\mathfrak{T}_P$-neural networks and $\mathcal{T}_P$-neural networks. Because the $\mathfrak{T}_P$-neural networks constructed in [185, 135, 119] required two additional layers in order to compute $\mathfrak{T}_P$ and had no learning functions, whereas $\mathcal{T}_P$-neural networks have only three layers with a learning function used in order to perform $\mathcal{T}_P$, we conclude that learning functions compensate for the use of additional layers.

The details are as follows. The $\mathfrak{T}_P$-neural networks are applicable to annotation-free many-valued logic programs. Chapter 2 and Section 4.4 show that any annotation-free logic program can be soundly transformed into annotated logic program. Suppose we have an annotation-free logic program P and its annotated version P'. Consider the construction of $\mathcal{T}_P$-neural network built to simulate $\mathcal{T}_{P'}$ for P'. As we have shown in Section 6.2, $\mathcal{T}_P$-neural networks are essentially 3-layer T_P-neural networks of [98], but

with embedded learning functions. These learning functions give an account of the cases, when we have to manipulate not only with atoms, but also with annotations attached to them. For example, we need to give an account of the fact that if $A : (\alpha)$ is computed, then $A : (\alpha')$ will be computed, where $\alpha' \leq_k \alpha$; or we must formalise the fact that whenever both $A : (\alpha)$ and $A : (\beta)$ are computed, then $A : (\alpha \oplus \beta)$ will be computed. In such cases the learning function is activated, and certain connections between units become active. We carefully examined this role of learning functions in Example 6.2.1.

$\mathfrak{T}_P$-neural networks computing $\mathfrak{T}_P$ [135, 185] do essentially the same thing: they extend T_P-neural networks, in order to give an account to cases when values of atoms play substantial role in deduction. Similarly to the $\mathcal{T}_P$-neural networks, each input and output unit of a $\mathfrak{T}_P$-neural network represents an atom together with some truth value. And, similarly to $\mathcal{T}_P$-neural networks, it is the presence of truth values that brings complications to the construction of the $\mathfrak{T}_P$-neural networks. Consider the following example: whenever two values α and β are assigned by the $\mathfrak{T}_P$-neural network to some atom A, the network must be able to compute the value $\alpha \oplus \beta$ for A. But recall that the $\mathfrak{T}_P$-neural networks have no learning. The alternative solution to learning in such cases is to postprocess signals of the traditional 3-layer network, in order to compute the value $\alpha \oplus \beta$. And thus, one additional layer appears. For the similar reasons, some input signals must be preprocessed, and so, the second additional layer is needed. Overall, the $\mathfrak{T}_P$-neural networks have 5 layers instead of 3.

Note that, because P and P' express the same information, and both $\mathcal{T}_{P'}$ and $\mathfrak{T}_P$ compute the same values for the same atoms, the number of connections post-processed and pre-processed in the $\mathfrak{T}_P$-neural network and the number of connections activated by means of learning function ϕ in $\mathcal{T}_{P'}$-neural network will always be equal. But, as we stated

in [119], if $\mathcal{T}_{P'}$-neural network spend time t to compute one iteration of $\mathcal{T}_{P'}$, the $\mathfrak{T}_P$-neural network will spend $t+2$ time to compute one iteration of $\mathfrak{T}_P$. This way the learning function ϕ saves space (two layers) and time (-2) at each iteration.

This has two important consequences.

1. Practically, because the number of layers is reduced, the space complexity of computations is getting improved in $\mathcal{T}_P$-neural networks comparing with $\mathfrak{T}_P$-neural networks. Because the number of layers determines the time of computations in all these neural networks, the time complexity is getting improved, too. This supports the Neurocomputing point of view that learning functions are extremely effective in practice.

2. Theoretically, by the discussion of Section 5.2.1, the use of three layers is preferable, because of the fundamental Theorems 5.2.1 and 5.2.2, showing the astounding computational power of three-layer neural networks.

To conclude, we have shown that comparing with $\mathfrak{T}_P$-neural networks, $\mathcal{T}_P$-neural networks use Hebbian learning functions in the spirit of Neurocomputing. This shift towards Neurocomputing and learning brings important theoretical and practical improvements into the construction of the resulting neural networks.

6.5 Conclusions

We devoted this chapter to the discussion of $\mathcal{T}_P$-neural networks. We have shown that these neural networks can simulate the $\mathcal{T}_P$-operator for the BAPs introduced in Chapter

4. The resulting neural networks incorporated Hebbian learning recognised in Neurocomputing. The latter result has methodological, practical and theoretical value. Methodologically, this extension of T_P-neural networks uses learning methods of *Neurocomputing* when developing *Connectionist* neural networks. Practically, learning improves time and space complexity of computations comparing with analogous many-valued extensions of T_P-neural networks by Lane and Seda [185, 135]. Theoretically, it is consistent with discussions of Section 5.2.1 advocating the use of three-layer neural networks.

We devoted one section to the discussion of an approximation theorem in the style of Seda [184] applied to $\mathcal{T}_P$-neural networks. In Section 6.3 we have shown that this approximation theorem, as well as any other approximation result [94, 10], causes some serious problems. The first problem is that the proofs are often not constructive, and we discussed two sources of non-constructivism in them: some [99, 94] use Funahashi's theorem, some [184, 129, 127] do not determine the size of a network that is sufficient for the accurate approximation of the semantic operator. The second problem is that, as follows from the results of [172, 191, 109, 192, 114, 103, 190], it is possible to build finite (and quite small) neural networks that are capable of computing, and not just "approximating", the least fixed point of T_P and $\mathcal{T}_P$. These two problems gave a major motivation for the next Chapter.

Chapter 7

SLD Neural Networks

7.1 Introduction

In this chapter we construct neural networks which are capable of performing the conventional SLD-resolution algorithm for definite first-order Logic Programs. The conventional algorithm of SLD resolution is defined and illustrated in Section 1.6.

It is worth mentioning here that the problem of establishing a way of performing first-order inference in neural networks has been tackled by many researchers over the last 30 years, see [72, 188, 136, 96, 97, 95], for example.

Similarly to T_P-neural networks, all these approaches did not encode the symbols of the language directly, but propagated only truth values through the constructed neural network. This made it technically difficult to perform such inference algorithms as unification, or substitution. To illustate this, we just mention that in [97, 190, 136, 149], the substitution and variable binding algorithms are realised as follows. Some units are "thought of" as representing some constants, and if certain conditions are satisfied, these units send binary signals to the units that are "thought of" as representing predicates.

This indirect representation of symbols of the language makes questionable whether effective implementation of such neural networks is possible. Moreover, this approach, similar to the T_P-neural networks, would result in infinite neural networks, should one wish to have function symbols in the language. So, such neural networks do not advance the theory of first-order inference either.

In Section 7.4, we build SLD neural networks which simulate the work of SLD-resolution, as opposed to the computation of the semantic operators in [98, 99]. We show that SLD neural networks have several advantages comparing with T_P-neural networks. First of all, they embed several learning functions, and thus perform different types of supervised and unsupervised learning recognised in Neurocomputing. Furthermore, SLD neural networks encode symbols of first-order language directly, which makes their future implementation easier. As a result, SLD neural networks do not require infinite number of neurons, and are able to perform resolution for any first-order logic program using finite number of units. The two properties of the SLD neural networks - finiteness and ability to learn - bring the Neuro-Symbolic computations closer to the practically efficient methods of neurocomputing [85].

The fact that classical first-order derivations require the use of learning mechanisms if implemented in neural networks is very interesting in its own right and suggests that first-order deductive theories are in fact capable of acquiring new knowledge, at least to the extent of how this process is understood in neurocomputing.

We proceed as follows. In Section 7.2 we fix a Gödel numbering that we will use later in order to numerically encode first-order language in neural networks. In the same section, we define some simple operations on Gödel numbers. In Section 7.3, we show how the algorithm of Unification can be performed in two-neuron neural networks. In

Section 7.4, we build SLD neural networks that work as an interpreter for conventional SLD-resolution. In Section 7.5 we make some conclusions from the construction of SLD neural networks.

The results of this Chapter appeared in [120] and [122].

7.2 Gödel Enumeration

In order to perform SLD-resolution in neural networks, we will allow not only binary threshold units, as in T_P-neural networks, but also units which may receive and send Gödel numbers as signals. Comparing with the traditional approach of [98, 99], where first-order atoms are not represented numerically in the T_P-neural networks and only "thought of" as being ascribed one or other unit, we encode first-order atoms directly via parameters of neural networks, and this enables us to perform unification and resolution in terms of functions of neural networks.

We do not insist that only Gödel numbers can be used here to enumerate the language. In principle, any deterministic way of encoding the first-order language will be as good as Gödel numbering. So, one can think about binary, or digital, or any other encoding. Some sort of number representation is crucial because neural networks are numerical machines, and can process only numbers.

We will use the fact that a first-order language yields a Gödel enumeration, [66]. There are several ways of performing the enumeration, we just fix one as follows.

Each symbol of a first-order language receives a **Gödel number** as follows:

- variables $x_1, x_2, x_3, \ldots$ receive numbers $(01), (011), (0111), \ldots$;
- constants $a_1, a_2, a_3, \ldots$ receive numbers $(21), (211), (2111), \ldots$;

- function symbols $f_1, f_2, f_3, \ldots$ receive numbers $(31), (311), (3111), \ldots$;
- predicate symbols $Q_1, Q_2, Q_3, \ldots$ receive numbers $(41), (411), (4111), \ldots$;
- symbols (,) and , receive numbers 5, 6 and 7, respectively.

It is possible to enumerate connectives and quantifiers, but we will not need them here and so omit further enumeration.

Example 7.2.1. *In the following table we enumerate the atoms from Example 1.6.1, the rightmost column contains short abbreviations we use for these numbers in further examples:*

Atom	*Gödel Number*	*Label*
$Q_1(f_1(x_1, x_2))$	*41531501701166*	g_1
$Q_2(x_1)$	*4115016*	g_2
$Q_3(x_2)$	*411150116*	g_3
$Q_3(a_2)$	*411152116*	g_4
$Q_2(a_1)$	*4115216*	g_5
$Q_1(f_1(a_1, a_2))$	*41531521721166*	g_6
$Q_4(x_1)$	*411115017*	g_0

We will reformulate some major notions defined in Section 1.6 in terms of Gödel numbers. We will define some simple (but useful) operations on Gödel numbers in the meantime.

Disagreement set from Definition 1.6.2 can be reformulated in terms of Gödel numbers as follows. Let g_1, g_2 be Gödel numbers of two arbitrary atoms A_1 and A_2, respectively. Define the set $g_1 \ominus g_2$ as follows.

- Locate the leftmost symbols $j_{g_1} \in g_1$ and $j_{g_2} \in g_2$ which are not equal.

- If j_{g_i}, $i \in \{1, 2\}$ is 0, put 0 and all successor symbols $1, \ldots, 1$ into $g_1 \ominus g_2$.
- If j_{g_i} is 2, put 2 and all successor symbols $1, \ldots, 1$ into $g_1 \ominus g_2$.
- If j_{g_i} is 3, then extract the first two symbols after j_{g_i}. Continue extracting successor symbols until the number of occurrences of symbol 6 becomes equal to the number of occurrences of symbol 5; put the number starting with j_{g_i} and ending with the last such 6 in $g_1 \ominus g_2$.
- If j_{g_i} is 4, then extract the first two symbols after j_{g_i}. Continue extracting successor symbols until the number of occurrences of symbol 6 becomes equal to the number of occurrences of symbol 5; put the number starting with j_{g_i} and ending with the last such 6 in $g_1 \ominus g_2$.

It is a straightforward observation that $g_1 \ominus g_2$ is equivalent to the notion of the disagreement set D_S, for $S = \{A_1, A_2\}$ defined in Section 1.6.

We will also need the operation $\oplus$, **concatenation** of Gödel numbers, defined by $g_1 \oplus g_2 = g_1 8 g_2$, where g_1 and g_2 are arbitrary Gödel numbers.

Remark about notation. The symbol $\oplus$ used in this chapter is not related to the symbol $\oplus$ we used to denote the k-join defined on bilattices in Chapters 3 and 4. Because we do not talk about bilattices in the current chapter, this somewhat excessive use of the symbol $\oplus$ will not cause confusion. We use this symbol when defining concatenation of Gödel numbers in order to emphasise that this operation is analogous to the usual operation $+$ that can be used when one works with natural or rational numbers, for example.

Let g_1 and g_2 denote Gödel numbers of a variable x_i and a term t respectively. We use the number $g_1 9 g_2$ to describe the substitution $\sigma = \{x/t\}$, and we will call $g_1 9 g_2$ the

Gödel number of the substitution σ. If the substitution is obtained for $g_m \ominus g_n$, we will sometimes refer to its Gödel number as $s(g_m \ominus g_n)$.

If g_1 is the Gödel number of some atom A_1, and $s = g_1'9g_2'8g_1''9g_2''8 \ldots 8g_1'''9g'''$ is the concatenation of Gödel numbers of some substitutions $\sigma', \sigma'', \ldots, \sigma'''$, then $g_1 \odot s$ is defined as follows: whenever g_1 contains a substring $(g_1)^*$ such that $(g_1)^*$ is equivalent to some substring s_i of s such that either s_i contains the first several symbols of s up to the first symbol 9 or s_i is contained between 8 and 9 in s, but does not contain 8 or 9 itself, substitute this substring $(g_1)^*$ in g_1 by the substring s_i' of symbols which succeed $s_i 9$ up to the first 8. It easy to see that $g_1 \odot s$ reformulates the substitution $(A_1)\sigma_1\sigma_2 \ldots, \sigma_n$ in terms of Gödel numbers. In neural networks, Gödel numbers can be used as positive or negative signals, and we put $g_1 \odot s$ to be 0 if $s = -g_1$.

The unification algorithm from Definition 1.6.3 can be restated in terms of Gödel numbers as follows: Let g_1 and g_2 be the Gödel numbers of two arbitrary atoms A_1 and A_2.

1. Put $k = 0$ and the Gödel number s_0 of the substitution σ_0 equal to 0.

2. If $g_1 \odot s_k = g_2 \odot s_k$ then stop; s_k is an mgu of g_1 and g_2. Otherwise, find the disagreement set $(g_1 \odot s_k) \ominus (g_2 \odot s_k)$ of $g_1 \odot s_k$ and $g_2 \odot s_k$.

3. If the set $(g_1 \odot s_k) \ominus (g_2 \odot s_k)$ contains a Gödel number starting with 4, stop; g_1 and g_2 are not unifiable.

4. If $(g_1 \odot s_k) \ominus (g_2 \odot s_k)$ contains a number g' starting with 0 and a number g'' such that g' does not occur as a sequence of symbols in g'', then put $s_{k+1} = s_k \oplus g'9g''$, increment k and go to 2. Otherwise, stop; g_1 and g_2 are not unifiable.

We will show in the next section, that the unification algorithm can be simulated in neural networks using the *error-correction learning* defined in Section 5.4.2.

7.3 Unification in Neural Networks

We have devoted much space in the two preceeding chapters to discussions of the fact that T_P-neural networks would have to contain an infinite number of units in order to perform first-order computations that employ infinitely many ground instances of atoms appearing in the Herabrand base B_P. The infiniteness of the architecture of the T_P-neural networks was characterised as an undesirable property of the neural networks. There have been many attempts to perform unification and substitution in connectionist neural networks before.

The problem of performing substitutions and unification in connectionist neural networks is older than T_P-neural networks, see [72] for a survey. Since the 1980s, numerous discussions have been held about solutions to this problem. First, the problem of the variable binding was formulated in [13]. Because all the connectionist neural networks of that time propagated only truth values 0 and 1 assigned to ground instances of first-order atoms, the problem was put as follows: "How is it possible to dynamically bind values to variables?".

Connectionist inference systems were criticised by John McCarthy, who observed in a commentary to [193], that in connectionist models that he had seen *the basic predicates ... are applied to a fixed object, and a concept is a propositional function of these predicates.* To overcome this *propositional approach* and to allow multiple predicates, Shastri and Ajjanagadde [187] proposed neural networks that could perform some sort of a variable binding. A similar construction was reproduced in the inference system ROBIN developed

by Lange and Dyer [136].

However, the resulting neural networks maintained the main disadvantage of all the connectionist neural networks in that they could only process binary truth values. According to [72], they essentially did not unify terms, but propagated constants through a network; function symbols were not allowed, otherwise, similarly to the T_P-neural networks, these neural networks could become infinite.

The unification algorithm we build in this sections gives an easy solution to the problem. In order to perform the unification algorithm, one does not have to introduce an infinite number of units corresponding to ground instances of atoms in B_P. We propose to encode first-order atoms numerically via parameters of the neural networks, and thus to calculate numbers of substitutions using the computational resources of *finite* neural networks. This method of performing unification is novel, and has not been used in Connectionism before.

The next theorem and its proof present this novel construction. The construction is effectively based on the error-correction learning algorithm defined in Section 5.4, and makes use of the operations $\odot, \oplus, \ominus, s$ over Gödel numbers defined in Section 7.2.

Theorem 7.3.1. *Let k be a neuron with the desired response value $d_k = g_B$, where g_B is the Gödel number of a first-order atom B, and let $v_j = 1$ be a signal sent to k with weight $w_{kj} = g_A$, where g_A is the Gödel number of a first-order atom A. Let h be a unit connected to k. Then there exists an error signal function e_k and an error-correction learning rule Δw_{kj} such that the* ***unification algorithm for A and B*** *is performed by* ***error-correction learning*** *at unit k, and the unit h outputs the Gödel number of an mgu of A and B if such an mgu exists, and it outputs 0 if no mgu of A and B exists.*

Proof. The error-correction learning will perform the unification of variables. But, before

unifying variables, we should check that the atoms A and B are build from the same predicates; otherwise the unification of variables will not lead to unifying A and B. This is why we use the Predicate Threshold, as follows.

Predicate threshold. If g_A starts with 4 and has a string of 1s after 4 of the same length as the string of 1s succeeding 4 in g_B, then start computations at time t, and use the parameter $w_{kj}(t) = g_A$. Otherwise, set the weight $w_{kj}(t) = 0$. (The Predicate threshold requirement reproduces the item 3 from the unification algorithm of Section 7.2.) If A and B are built from the same predicates, we start computations.

We set $\Theta_k = \Theta_h = 0$, and the initial weight $w_{hk}(t) = 0$ of the connection between k and h. We use the standard formula to compute $p_k(t) = v_j(t)w_{kj}(t) - \Theta_k$, and put $v_k(t) = p_k(t)$ if $p_k(t) \geq 0$, and $v_k(t) = 0$ otherwise.

The error signal is defined by $e_k(t) = s(d_k(t) \ominus v_j(t))$. That is, $d_k(t) \ominus v_j(t)$ computes the disagreement set of g_B and g_A, and $s(d_k(t) \ominus v_j(t))$ computes the Gödel number of the substitution for this disagreement set, as described in item 4 of the unification algorithm. If $d_k(t) \ominus v_j(t) = \emptyset$, set $e_k(t) = 0$. This corresponds to item 1 of the unification algorithm. If $d_k(t) \ominus v_j(t) \neq \emptyset$, but $s(d_k(t) \ominus v_j(t))$ is empty, set $e_k(t) = -w_{kj}(t)$. The latter condition covers the case when g_A and g_B are not unifiable.

We use the error-correction learning rule defined in Section 5.4.2: $\Delta w_{kj}(t) = v_j(t)e_k(t)$. In our case $v_j(t) = 1$, at every time t, and so $\Delta w_{kj}(t) = e_k(t)$. We use $\Delta w_{kj}(t)$ to compute

$$w_{kj}(t+1) = w_{kj}(t) \odot \Delta w_{kj}(t) \text{ and } d_k(t+1) = d_k(t) \odot \Delta w_{kj}(t).$$

That is, at each new iteration of this unit, substitutions are performed in accordance with item 2 of the Unification algorithm.

We update the weight of the connection from the unit k to the unit h:
$w_{hk}(t+1) = w_{hk}(t) \oplus \Delta w_{kj}(t)$, if $\Delta w_{kj}(t) \geq 0$, and $w_{hk}(t+1) = 0$ otherwise. That is, the

Gödel numbers of the substitutions will be concatenated at each iteration, in accordance with item 4 of the unification algorithm, if an mgu exists; and the weight w_{hk} will be set to 0 if the unification fails.

It remains to shows how to read the Gödel number of the resulting mgu. Compute $p_h(t+\Delta t) = v_k(t+\Delta t) \odot w_{hk}(t+\Delta t)$. If $p_h(t+\Delta t) > 0$, put $v_h(t+\Delta t) = w_{hk}(t+\Delta t)$, and put $v_h(t+\Delta t) = 0$ otherwise. (Note that $p_h(t+\Delta t)$ can be equal to 0 only if $v_k(t+\Delta t) = 0$, and this is possible only if $w_{kj}(t+\Delta t) = 0$. But this, in turn, is possible only if $\Delta w_{kj}(t+\Delta t-1) = e_k(t+\Delta t-1) = -w_{kj}(t+\Delta t-1)$, that is, if some terms appearing in A and B are reported to be non-unifiable according to the unification algorithm.) Thus, if an mgu of A and B exists, it will be computed by $v_h(t+\Delta t)$, and if it does not exists, the unit h will give $v_h(t+\Delta t) = 0$ as its output.

If the signal $v_h(t+\Delta t) \neq 0$ and the first and the last symbols constituting the number $v_h(t+\Delta t)$ are 0, stop. The signal $v_h(t+\Delta t)$ is the Gödel number of mgu of A and B. □

For example, in case when $A = B$ and their mgu is ε, the neural network will output the signal $v_h(t+2) = 080$.

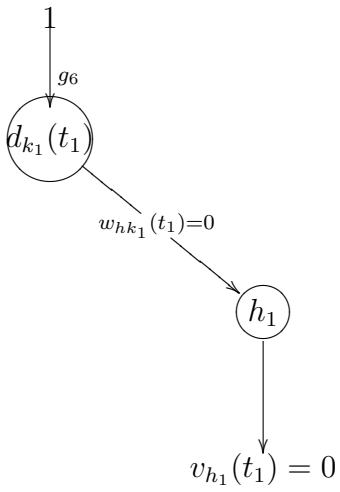
The next example illustrates how the construction of Theorem 7.3.1 works.

Example 7.3.1. *Let $S = \{Q(f(x_1, x_2)), Q(f(a_1, a_2))\}$ be a set of first-order atoms. These atoms were used in Examples 1.3.2 and 1.6.1. Then $\theta = \{x_1/a_1;\ x_2/a_2\}$ is the mgu for S.*

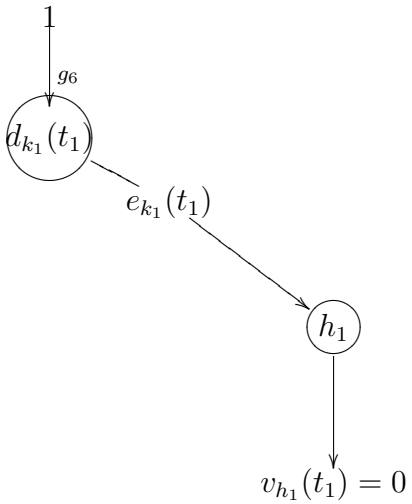
Next we show how this can be computed by neural networks. In Example 7.2.1, we encoded these two atoms using Gödel numbers, and we denoted the numbers of $Q(f(x_1, x_2))$ and $Q(f(a_1, a_2))$ by g_1 and g_6, respectively.

The neural network we build will consist of two units, k_1 and h_1. We send the signal 1 to the unit k. We use the numbers g_1 and g_6 as parameters of the neural network, as

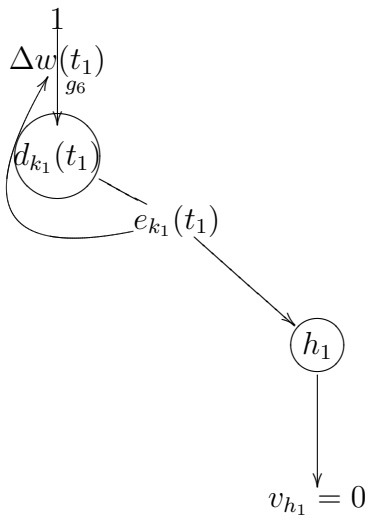
follows: $w_{k_1 i}(t_1) = g_6$, *and hence* $v_{k_1}(t_1) = 1 \,.\, g_6 = g_6$; $d_{k_1}(t_1) = g_1$. *See the next diagram:*



At time t_1, *this neural network computes* $e_{k_1}(t_1) = s(d_{k_1}(t_1) \ominus v_{k_1}(t_1))$ - *the Gödel number of the substitution for the disagreement set* $d_{k_1}(t_1) \ominus v_{k_1}(t_1)$:



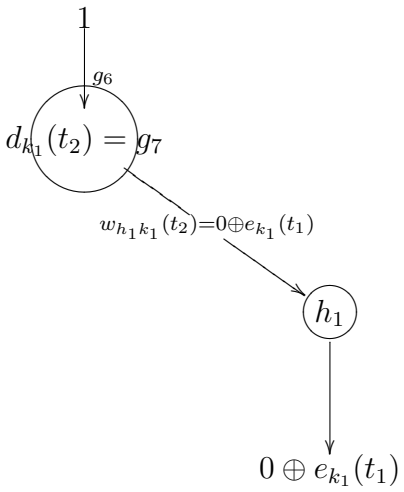
The error signal $e_{k_1}(t_1)$ *will then be used to change the weight of the outer connection* $w_{k_1 i}$ *to* k_1 *and some other parameters; and* $\Delta w(t_1)$ *is computed for this purpose as follows:* $\Delta w(t_1) = v_i(t_1) e_{k_1}(t_1) = e_{k_1}(t_1)$. *This is shown on the next diagram.*



At time t_2, the weight of the connection between i and k_1 is amended using $e_{k_1}(t_1)$, and the desired response value $d_{k_1}(t_2)$ is "trained", too: $w_{k_1 i}(t_2) = w_{k_1 i}(t_1) \odot \Delta w_{k_1 i}(t_1)$ and $d_{k_1}(t_2) = d_{k_1}(t_1) \odot \Delta w_{k_1 i}(t_1)$. Note that the operation $\odot$ was used in the unification algorithm of Section 7.2 to apply substitutions. So, informally speaking, at this stage the computed substitution $e_{k_1}(t_1)$ is applied to the numbers of both atoms $Q(f(x_1, x_2)), Q(f(a_1, a_2))$ we are unifying.

The weight between k_1 and h_1 is amended at this stage: $w_{h_1 k_1}(t_2) = w_{h_1 k_1}(t_1) \oplus \Delta w_{h_1 k_1}(t_1)$.

At time t_2, all the parameters are updated as follows: $w_{k_1 i}(t_2) = v_{k_1}(t_2) = g_6$ is the Gödel number of $Q_1(f(a_1, a_2))$; $d_{k_1}(t_2) = g_7$ is the Gödel number of $Q_1(f(a_1, x_2))$. And this is shown on the next diagram:

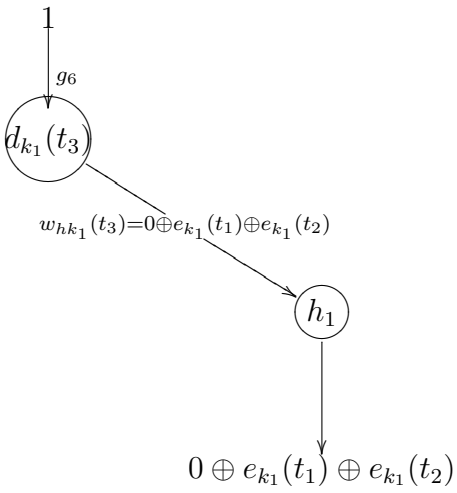


Note that on the diagram above, the unit h_1 *emits its first non-zero output signal, that is, the number of the substitution* $e_{k_1}(t_1)$.

Because the parameters $w_{k_1 i}(t_2) = g_6$ *and* $d_{k_1}(t_2) = g_7$ *are not equal numbers yet and the number* v_{h_1} *does not end with* 0, *the same process as we have described starts again. And at times* $t_2 - t_3$, *the Gödel number* $e_{k_1}(t_2)$ *of a new substitution* $\{x_2/a_2\}$ *is computed and applied, as follows:*

$e_{k_1}(t_2) = s(g_6 \ominus g_7)$; $w_{k_1 i}(t_3) = v_{k_1}(t_3) = g_6$ *is the Gödel number of* $Q_1(f(a_1, a_2))$;

$d_{k_1}(t_3) = d_{k_1}(t_2) \odot e_{k_1}(t_2) = g_6$ *is the Gödel number of* $Q_1(f(a_1, a_2))$:



At time t_4, a new iteration will be initialised. But, because $d_{k_1}(t_4) = v_{k_1}(t_4) = g_6$, $e_{k_1}(t_4) = 0$. Thus, $w_{h_1k_1}(t_4) = 0 \oplus e_{k_1}(t_1) \oplus e_{k_1}(t_2) \oplus 0$.

Note that the neuron h_1 finally emits the signal that contains both substitutions computed by the network. The fact that the last symbol of the number $0 \oplus e_{k_1}(t_1) \oplus e_{k_1}(t_2) \oplus 0$ is 0, tells the outer reader that the unification algorithm has finished its computations. It can be deterministically checked that the number $0 \oplus e_{k_1}(t_1) \oplus e_{k_1}(t_2) \oplus 0$ is the Gödel number of the substitution θ.

The animated version of this example can be found in [117].

The main conclusions to be made from the construction of Theorem 7.3.1 are as follows:

- First-order atoms are embedded directly into a neural network via Gödel numbers.
- The resulting neural network is finite and gives deterministic results, comparing with the infinite layers that would be needed to perform first-order substitutions in the neural networks of [187, 136, 96].
- The error-correction learning recognised in Neurocomputing is used to perform the unification algorithm.

- The unification algorithm is performed as an adaptive process, which corrects one piece of data relative to the other piece of data.

Note that in this section, as well as in the previous section, we used Definition 1.6.3 when constructing the neural networks, and not the original definition of the unification algorithm due to Robinson [178]. As we already explained in Section 1.6, the Definition 1.6.3 requires to check the predicates before initialising the unification of variables. It is not difficult to modify the neural networks of this section in order to bring them into correspondence with the original definition of the unification algorithm of [178]. If we did so, the proofs and examples would have become slightly more complex. Note that the simplicity of exposition was the main reason why we used Definition 1.6.3. The construction of neural networks we have proposed here can easily be modified to satisfy the original unification algorithm. Doing this is a technical rather than theoretical question.

7.4 SLD Neural Networks

We use the results and constructions of the previous section to state and prove the main Theorem of this chapter, and we construct the SLD neural networks next.

Note that in this section, we continue to impose the Predicate threshold which we used to perform the unification algorithm. One can, in principle, use the original definition of Robinson [178] here, and the Predicate threshold would not be needed in this case. We impose the Predicate threshold here in order to avoid excessive iterations of error-correction learning.

Theorem 7.4.1. *Let P be a definite logic program and G be a definite goal. Then there*

exists a 3-layer recurrent neural network which computes the Gödel number s of the substitution θ if and only if SLD-refutation derives θ as an answer for $P \cup \{G\}$.

Proof. Let P be a logic program and $C_1, \ldots, C_m$ be the definite clauses contained in P.

The SLD neural network consists of three layers, *Kohonen's layer* **k** (see the table below) of input units $k_1, \ldots, k_m$, layer **h** of output units $h_1, \ldots, h_m$ and layer **o** of units $o_1, \ldots, o_l$, where m is the number of clauses in the logic program P, and l is the number of all atoms appearing in the bodies of clauses in P.

Similarly to the connectionist T_P-neural networks, each input unit k_i represents the head of some clause C_i in P, and is connected to precisely one unit h_i, which is connected, in turn, to units $o_k, \ldots, o_s$ representing atoms contained in the body of C_i. This is the main similar feature of T_P- and SLD- neural networks. Note that in T_P-neural networks **o** was an output layer, and the layer **c** - the analogue of the layer **h** - was a hidden layer, whereas in our setting **h** will be an output layer and we require the reverse flow of signals comparing with T_P-networks.

The thresholds of all the units are set to 0.

The layers **k** and **h** will perform the unification algorithm described in Theorem 7.3.1. Thus, the input units $k_1, \ldots, k_m$ will be involved in the process of error-correction learning and this is why each of $k_1, \ldots, k_m$ must be characterised by the value of the *desired response* d_{k_i}, $i \in \{1, \ldots, m\}$, and each d_{k_i} is the Gödel number of the atom A_i which is the head of the clause C_i. Initially all weights between layer **k** and layer **h** are set to 0, but an error-correction learning function is introduced in each connection between k_i and h_i, as in Theorem 7.3.1. The weight from each h_i to some o_j is defined to be the Gödel number of the atom represented by o_j.

Consider a definite goal G that contains atoms $B_1, \ldots, B_n$, and let $g_1, \ldots g_n$ be the

Gödel numbers of $B_1, \ldots, B_n$. Then, for each g_l, do the following: at time t send a signal $v_l = 1$ to each unit k_i, $i \in \{1, \ldots, m\}$. Set the weight $w_{k_i l}(t) = g_l$.

Predicate threshold function will be assumed throughout the proof, and is stated as follows. Suppose $w_{k_i l}^{\texttt{old}} = g_l$ is the weight of the connection from some external source l to the input unit k_i. Set the weight of the connection $w_{k_i,l}^{\texttt{new}}(t)$ equal to g_l ($l \in \{1, \ldots, n\}$) if g_l starts with 4 and has a string of 1s after 4 of the same length as the string of 1s succeeding 4 in d_{k_i}. Otherwise, set the weight $w_{k_i l}^{\texttt{new}}(t)$ to 0. There may be several such signals from one g_l to each of the $k_1, \ldots, k_m$, and we denote them by $v_{l1}, \ldots, v_{lm}$.

The Predicate Threshold function is needed to block connections between external signals encoding goal atoms and input units encoding clause heads, in case if the given goal atom and the clause head are built from different predicates.

Step 1 shows how the input layer **k** filters excessive signals in order to process, according to the SLD-resolution algorithm, only one goal at a time, using only one clause at a time. This step will involve the use of *Kohonen competition* and *Grossberg's laws*, defined in Sections 5.4.3, 5.4.4.

Suppose several input signals $v_{l1}(t), \ldots, v_{lm}(t)$ were sent from one source l to units $k_1, \ldots, k_m$. At time t, only one of $v_{l1}(t), \ldots, v_{lm}(t)$ can be activated, and we apply the *inverse Grossberg's law* to filter the signals $v_{l1}(t), \ldots, v_{lm}(t)$ as follows. Fix the *unconditioned signal* $v_{l1}(t)$ and compute, for each $j \in \{2, \ldots, m\}$ and $i \in \{1, \ldots, m\}$, $w_{k_i l_j}^{\text{new}}(t) = w_{k_i l_j}^{\text{old}}(t) + [v_{l1}(t) v_{l_j}(t) - w_{k_i l_j}^{\text{old}}(t)] U(v_{l_j})$. We will also denote this function by $\psi_1(w_{k_i l_j}(t))$. Recall that $v_{l_1}(t) = v_{l_j}(t) = 1$ by the definitions above, and $U(v_{l_j}) = 1$ whenever $v_{l_j} > 0$; and, by construction above, each $v_{l_j}(t) > 0$. Therefore, this filter will set all the weights $w_{k_i l_j}(t)$, where $j \in \{2, \ldots, m\}$, to 1, and the *Predicate threshold* will ensure that those weights will be inactive.

The use of the inverse Grossberg's law here reflects the logic programming convention that each goal atom unifies only with one clause at a time. Yet several goal atoms may be unifiable with one and the same clause, and we use Grossberg's law *to filter signals of this type as follows.*

If an input unit k_i receives several signals $v_j(t), \ldots, v_r(t)$ from different sources $j, \ldots, r$, then fix an *unconditioned signal* $v_j(t)$ and apply, for all $m \in \{(j+1), \ldots, r\}$ the equation $w_{k_i m}^{\text{new}}(t) = w_{k_i m}^{\text{old}}(t) + [v_m(t)v_j(t) - w_{k_i m}^{\text{old}}(t)]U(v_m)$ at time t; we will denote this function by $\psi_2(w_{k_i m}(t))$. The function ψ_2 will have the same effect as ψ_1: all the signals except $v_j(t)$ will have to pass through connections with weights 1, and the *Predicate threshold* will make them inactive at time t.

The weights which were set to 0 by means of applying Grossberg's functions ψ_1 and ψ_2 and the Predicate threshold will retain their 0 weights until $e_{k_i}(t + \Delta t)$ computes, at time $t + \Delta t$, the value 0 or a negative value. In either of these two cases, at time $t + \Delta t + 2$, the old, non-zero weights, will participate in another turn of Grossberg's "filter" learning. We will give more details of it as we go along.

Functions ψ_1 and ψ_2 will guarantee that each input unit processes only one signal at a time.

At this stage we could start further computations independently at each input unit, but the algorithm of SLD-refutation treats each non-ground atom in a goal as dependent on others via variable substitutions, that is, if one goal atom unifies with some clause, the other goal atoms will be subjects to the same substitutions. This is why we must avoid independent, parallel computations in the input layer and we apply the principles of competitive learning as they are realized in Kohonen's layer*:*

At time $t+1$, compute $I_{k_i}(t+1) = i$, for each k_i, $i = 1, \ldots, m$. The unit with the least

$I_{k_i}(t+1)$ will proceed with the computations of $p_{k_i}(t+1)$ and $v_{k_i}(t+1)$, all the other units $k_j \neq k_i$ will automatically receive the value $v_{k_j}(t+1) = 0$.

Note that if neither of $w_{k_ij}(t+1)$ contains the symbol 0 (all goal atoms are ground), we don't have to apply Kohonen's competition and we can proceed with parallel computations for each input unit.

After applying the Grossberg's laws and Kohonen's competition, there will be only one non-zero signal which reaches the input layer **k**. For the next few paragraphs, we denote the source of this signal by j, and the value of the signal by v_j. The value v_j is set to 1.

Now, given an input signal $v_j(t+1)$, $j \in \{1, \ldots, n\}$, the potential $p_{k_i}(t+1)$ will be computed using the standard formula: $p_{k_i}(t+1) = v_j(t+1)w_{k_ij} - \Theta_k$, where, as we defined before, $v_j(t+1) = 1, w_{k_ij} = g_j$ and $\Theta_{k_i} = 0$. The output signal from k_i is computed as follows: $v_{k_i}(t+1) = p_{k_i}(t+1)$, if $p_{k_i}(t+1) > 0$, and $v_{k_i}(t+1) = 0$ otherwise; this agrees with the construction of Theorem 7.3.1.

At this stage the input unit k_i is ready to propagate the signal $v_{k_i}(t+1)$ further. However, the signal $v_{k_i}(t+1)$ may be different from the desired response $d_{k_i}(t+1)$, and the network initialises the *error-correction learning* in order to bring the signal $v_{k_i}(t+1)$ in correspondence with the desired response and compute the Gödel number of an mgu. *We use here the construction of Theorem 7.3.1, and conclude that at some time* $(t+\Delta t)$, $v_{k_i}(t+\Delta t) = d_{k_i}(t+\Delta t)$, *and the signal* $v_{h_i}(t+\Delta t+1)$ *(the Gödel number of the relevant mgu) is sent both as the input signal to the layer* **o** *and as an output signal of the network which can be read by an external recipient.*

The next two paragraphs describe amendments to the neural networks to be done in cases when either the mgu was obtained, or the unification algorithm of Theorem 7.3.1

reported that no mgu exists.

If $e_{k_i}(t+\Delta t) = 0$ ($\Delta t \geq 1$), then, for each $i \in \{1, \ldots, m\}$, set $w_{k_i j}(t+\Delta t+2) = 0$, where j is the input signal we have been using; change input weights leading from all other sources r, $r \neq j$ to units $k_1, \ldots, k_n$, using $w_{k_n r}(t+\Delta t+2) = w^{\mathtt{old}}_{k_n r}(t) \odot w_{h_i k_i}(t+\Delta t)$.

Whenever at time $t+\Delta t$ ($\Delta t \geq 1$), $e_{k_i}(t+\Delta t) \leq 0$, set the weight $w_{h_i k_i}(t+\Delta t+2) = 0$. At time $t+\Delta t+2$, initialise a new activation of *Grossberg's function* ψ_2 (for some fixed $v_m \neq v_j$); if $e_{k_i}(t+\Delta t) < 0$, initialise at time $t+\Delta t+2$ a new activation of *inverse Grossberg's function* ψ_1 (for some $v_{l_i} \neq v_{l_1}$). In both cases initialise Kohonen's layer competition at time $t+\Delta t+3$. If no such $v_m \neq v_j$ or no such $v_{l_i} \neq v_{l_1}$ are available, stop the computations at time $t+\Delta t+2$.

Step 2. As we defined already, h_i is connected to some units $o_l, \ldots, o_r$ in the layer $\mathbf{o}$ with weights $w_{o_l h_i} = g_{o_l}, \ldots, w_{o_r h_i} = g_{o_r}$. And v_{h_i} is sent to each $o_l, \ldots, o_r$ at time $t+\Delta t+1$. The network will now compute, for each o_l, $p_{o_l}(t+\Delta t+1) = w_{o_l h_i} \odot v_{h_i} - \Theta_{o_l}$, with $\Theta_{o_l} = 0$. Put $v_{o_l}(t+\Delta t+1) = 1$ if $p_{o_l}(t+\Delta t+1) > 0$ and $v_{o_l}(t+\Delta t+1) = 0$ otherwise.

At step 2 the network applies the substitutions to the atoms in the body of the clause whose head has been unified already.

Step 3. At time $t+\Delta t+2$, $v_{o_l}(t+\Delta t+1)$ is sent to the layer $\mathbf{k}$. Note that all weights $w_{k_j o_l}(t+\Delta t+2)$, $j \in \{1, \ldots, m\}$, $l \in \{l, \ldots, r\}$, were defined to be 0, and we introduce the learning function $\vartheta = \Delta w_{k_j o_l}(t+\Delta t+1) = p_{o_l}(t+\Delta t+1) v_{o_l}(t+\Delta t+1)$, which can be seen as a kind of Hebbian function, see Section 5.4.1. At time $t+\Delta t+2$ the network computes $w_{k_j o_l}(t+\Delta t+2) = w_{k_j o_l}(t+\Delta t+1) + \Delta w_{k_j o_l}(t+\Delta t+1)$.

At step 3, the new goals, which are the Gödel numbers of the body atoms (with applied substitutions) are formed and sent to the input layer.

Once the signals $v_{o_l}(t + \Delta t + 2)$ are sent as input signals to the input layer **k**, the *Grossberg's functions* will be activated at time $(t + \Delta t + 2)$, *Kohonen competition* will take place at time $(t+\Delta t+3)$ as described in Step 1 and thus the new iteration will start.

Computing and reading the answer. The signals $v_{h_1}, \ldots, v_{h_m}$ are read at different times from the hidden layer **h**, and by Theorem 7.3.1, are Gödel numbers of the relevant substitutions. We say that an SLD neural network *computes an answer for* $P \cup \{G\}$, if and only if, for each external source i of input signals $v_{i_1}(t), v_{i_2}(t), \ldots, v_{i_n}(t)$ and internal source o_s of input signals $v_{o_{s1}}(t), v_{o_{s2}}(t), \ldots, v_{o_{sm}}(t)$, the following holds. For at least one input signal $v_{i_l}(t)$ sent from the source i and for at least one input signal or $v_{o_{sk}}(t)$ sent from o_s, there exists $v_{h_j}(t + \Delta t)$, where j is the number of input neuron k_j to which the signal was sent; and $v_{h_j}(t + \Delta t)$ is such that $v_{h_j}(t + \Delta t)$ is a string of length $l \geq 2$ whose first and last symbol is 0. If, for all $v_{i_l}(t)$ ($v_{o_{sk}}(t)$ respectively), $(l \in \{1, \ldots, n\}$, $k \in \{1, \ldots, m\})$, the corresponding value $v_{h_j}(t + \Delta t) = 0$, we say that the computation failed.

The rest of the proof proceeds by routine induction.

Inductive part of the proof.

Let P be a logic program, G_0 a program goal and NN be an SLD neural network built as described in Theorem 7.4.1.

In the proof we assume that we work with the general case, when goals are non-ground, and therefore, we need to apply Kohonen's competition at each step of computation. We assume also that the SLD-refutation algorithm always chooses the leftmost goal atom from its list of goals, and the leftmost clause from its list of clauses. We assume that the input signals corresponding to program goals and units of layer **k** are enumerated in accordance with the enumeration of program goals and program clauses in P.

We call the period of time during which the NN receives a signal in the input layer **k**, and propagates it through the layers **h** and **o**, an *iteration* of the NN.

Subproof 1. We prove that, if there is an SLD-refutation for $P \cup G_0$ with the answer $\theta_1, \ldots, \theta_n$, then NN will *compute* the Gödel number of $\theta_1, \ldots, \theta_n$. We proceed with the proof by induction on the length n of the refutation $P \cup G_0$.

Basis step. Let $n = 1$. Then $G_0 = \leftarrow B$ and there exists a unit clause $A \leftarrow$, where A and B are first-order atomic formulae such that there exists θ making $A\theta = B\theta$. But then, by the construction of NN, there exists an input unit k_j with d_{k_j} such that d_{k_j} is the Gödel number of A and the unit k_j receives an input signal $v_i(t) = 1$ via $w_{k_j i}(t)$, such that $w_{k_j i}(t)$ is the Gödel number of B. If θ exists, then, by the unification algorithm of Section 1.6 and Theorem 7.3.1, $e(t + \Delta t) = 0$ will be reached by NN at some time $t + \Delta t$, and $v_{h_j}(t + \Delta t)$ will be computed, $v_{h_j}(t + \Delta t)$ being the Gödel number of θ.

Recall that the SLD-resolution algorithm chooses the leftmost (or the upper) clause to unify B with. And the function ψ_1 chooses the leftmost unit in the layer **k**. This guarantees that in case if B is unifiable with several unit clauses, NN will perform exactly the same unification that the SLD-resolution algorithm does.

Inductive step. Suppose the statement holds for refutations of length $n-1$. We will prove that then it holds for the length of refutation n. Consider some goal

$$G_{k-1} = \leftarrow (B_1, \ldots, B_l, \ldots B_s)\theta_1 \ldots \theta_{k-1},$$

which has a refutation of length n and the goal

$$G_k = \leftarrow (B_1, \ldots, B_{l-1}, C_1, \ldots, C_m, B_{l+1}, \ldots B_s)\theta_1 \ldots \theta_k$$

derived from G_{k-1} and some clause $A \leftarrow C_1, \ldots, C_m$, such that B_l is a selected goal atom, and there exist $\theta_1, \ldots, \theta_k$ making $A\theta_k = B_l\theta_1 \ldots \theta_k$. Clearly, G_k has the length of

refutation $n-1$, and, by induction hypothesis, NN computes the answer for G_k.

By the construction of NN, there exists a unit k_j, with d_{k_j} equal to the Gödel number of A and there exists a unit h_j connected both to k_j and some $o_{i_1}, \ldots, o_{i_m}$, such that each weight $w_{o_{i_j}h_j}$ is the Gödel number of C_j, $j = 1, \ldots, m$. Because G_{k-1} has a refutation, by the construction of NN, some input signal $v_f(t)$ will be sent to the layer **k** with weight $w_{kf}(t)$ equal to the Gödel number of $B_l\theta_1, \ldots, \theta_k$. And, because B_l has to be involved in further refutation and is unifiable with A, by the construction of NN we conclude that after applications of Predicate threshold, two Grossberg's functions and Kohonen's competition, $v_f(t)$ will reach some k_j and will initialise the error-correction learning using $d_{k_j}(t)$ at time t. The latter is guaranteed by the fact that the SLD-resolution algorithm always chooses the leftmots goals and clauses, and ψ_1, ψ_2, I are defined to choose those input signals and those units in the layer **k**, which have the least indices.

By Theorem 7.3.1, the fact that A and B are unifiable, suggests that at time $t+\Delta t$ NN will reach the state when $e_{k_j}(t+\Delta t) = 0$, and $v_{h_j}(t+\Delta t)$ will be read off the output layer **h** and, since, h_j is connected to $o_{i_1}, \ldots, o_{i_m}$, $v_{h_j}(t+\Delta t)$ will be sent to $o_{i_1}, \ldots, o_{i_m}$. Further, each $v_{o_{i_j}}(t+\Delta t+2) = 1$ will be sent to the input layer **k** with each $w_{ko_{i_j}}(t+\Delta t+2)$, $j = 1, \ldots, m$, equal to the Gödel number of $C_j\theta_k$. But, according to the induction hypothesis, NN will compute the Gödel numbers of answers for

$$\leftarrow B_1, \ldots, B_{l-1}, C_1, \ldots, C_m, B_{l+1}, \ldots, B_s.$$

Thus, we have proven that NN computes the Gödel numbers of answers for $P \cup G_{k-1}$.

Subproof 2. We prove that, if an SLD neural network NN computes $v_{h_1}, \ldots, v_{h_k}$ producing Gödel numbers $s_1, \ldots, s_n$, then $s_1, \ldots, s_n$ are Gödel numbers of substitutions which constitute an answer for $P \cup G$. We will prove this statement by induction on the number

n of iterations of NN.

Basis step. Suppose $n = 1$, that is, only one iteration was needed to compute an answer v_{h_j}. This means that there was only one external impulse $v_i(1)$ which activated, through $w_{k_j i}(1)$, some unit k_j in the input layer **k**. Since the answer of the SLD neural network is actually *computed*, no other signals are received by the input layer **k**, and from this we make the conclusion that h_j is not connected to any unit in the layer **o**. Therefore, by construction of NN, we conclude that there was a goal $G_0 = \leftarrow B$, with B being some atom in the first-order language such that the Gödel number of B is equal to $w_{k_j i}(1)$; and there was a unit clause $A \leftarrow$ in P, A being some atom in the first-order language, such that d_{k_j} is precisely the Gödel number of A, moreover, there exists θ such that $A\theta = B\theta$ with θ being such that its Gödel number is v_{h_j}. But then, clearly, $P \cup \{G_0\}$ has an SLD-refutation, with the answer θ.

Inductive step. Suppose the statement holds for $n - 1$ iterations. We prove that then it holds for n iterations. Consider the iteration n at which some $v_{h_j}(t)$ was computed (that is, has the form $08e_{k_j}(t-2)8 \ldots 8e_{k_j}(t-2-\Delta t)80$), such that $v_{h_1}, \ldots, v_{h_j}, \ldots, v_{h_k}$ constitute the final answer, with each v_{h_l} $(l \neq j)$ computed at one of the $n-1$ iterations.

By the construction of NN, the unit h_j was first excited by the unit k_j, which received, at some time $t - \Delta t - 2$ either an input signal $v_i(t - \Delta t - 2)$ (if it was an external impulse) or $v_{o_s}(t - \Delta t - 2)$ (if it was an internal signal).

By the construction of NN, we know that $w_{k_j o_s}(t - \Delta t - 2)$ (or $w_{k_j i}(t - \Delta t - 2)$) is the Gödel number of some first-order atomic formula B_l, and $d_{k_j}(t - \Delta t - 2)$ is the Gödel number of some first-order atomic formula A which is the head of some clause $A \leftarrow C_1, \ldots, C_m$. Moreover, B_l is a selected goal atom of some $G = \leftarrow B_1, \ldots, B_l, \ldots, B_n$.

Since $v_{h_j}(t)$ is of the form $08e_{k_j}(t-2)8 \ldots 8e_{k_j}(t-2+\Delta t)80$, we conclude, using the

definition of $\Delta w_{h_j k_j}$ and the definition of the unification algorithm given and Theorem 7.3.1, that B_i and A are unifiable via some θ, such that the Gödel number of θ is precisely $v_{h_j}(t)$.

Furthermore, $v_{h_j}(t)$ is sent to $o_{i_1}, \ldots, o_{i_m}$, with each $w_{o_{i_r} h_j}$ encoding the Gödel number of C_r, $r \in \{1, \ldots, m\}$, from $A \leftarrow C_1, \ldots, C_m$. The layer **o** then emits signals $v_{o_{i_1}}, \ldots, v_{o_{i_m}} = 1$ and the weight between each o_{i_j} and layer **k** is set to the Gödel number of $C_j\theta$.

Now $n-1$ iterations are left to be completed. And, according to the induction hypothesis, the signals emitted by NN during these $n-1$ iteration correspond to the Gödel numbers of relevant answers for the SLD-refutation of

$$P \cup \leftarrow (B_1, \ldots, B_{l-1}, C_1, \ldots, C_m, B_{l-1}, \ldots, B_m)\theta.$$

Thus, NN computes $v_{h_1}, \ldots, v_{h_j}, \ldots, v_{h_k}$ which constitute the Gödel number of an answer for $P \cup G$ in n iterations.

This completes the proof.

□

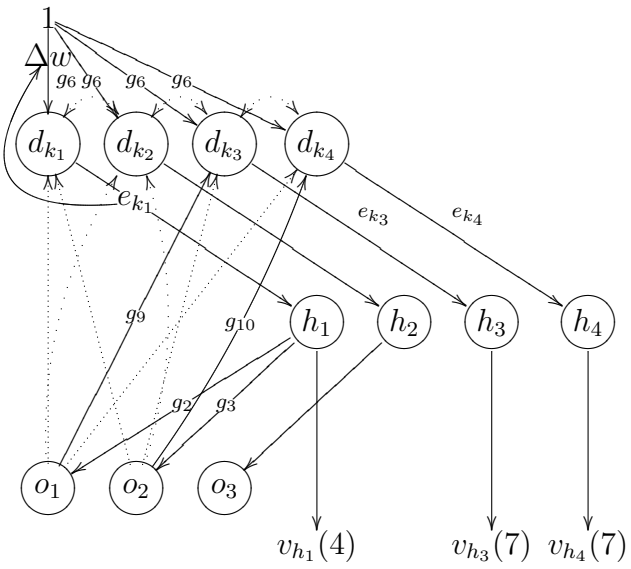
It follows from the constructions of the proof above, that the length of the SLD-refutation for $P \cup \{G\}$ and the number of iterations of the corresponding NN are equal. We omit stating this as a separate theorem or corollary, we only make a remark that such corollary is possible.

Backtracking is one of the major techniques in SLD-resolution. We formulate it in the SLD neural networks as follows. Let k_j be an input unit, and h_j be a unit from the layer **h** connected to k_j as described in Theorems 7.3.1 and 7.4.1. Let v_{ol} be a signal sent to the input unit k_j via $w_{k_j o_l}$. Whenever $v_{h_j}(t + \Delta t) = 0$, do the following.

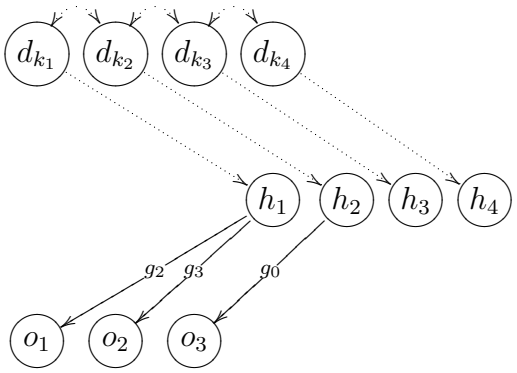
1. Find the corresponding unit k_j and the weight $w_{k_j o_l}$; apply the inverse Grossberg's function ψ_1 to some $v_{o_s} \neq v_{ol}$, such that v_{o_s} has not been an *unconditioned signal* before.

2. If there is no such v_{o_s}, find a unit h_f in the layer $\mathbf{h}$, such that h_f is connected to o_l, change the units and weights with indices j to the units and weights with indices f and go to item 1.

Example 7.4.1. *Consider the logic program P_1 from Example 1.3.2 and the process of SLD-refutation for P_1 and the goal $G_0 = \leftarrow Q_1(f_1(a_1, a_2))$ from Example 1.6.1. We will build an SLD neural network that performs this refutation. Gödel numbers $g_1, \ldots, g_6$ are taken from Example 7.2.1. Input layer $\mathbf{k}$ consists of units k_1, k_2, k_3 and k_4, representing heads of the four clauses in P_1, each with the* desired response value d_{k_i}, *($d_{k_1} = d_{k_2} = g_1$, $d_{k_3} = g_5$, $d_{k_4} = g_4$). The layer $\mathbf{o}$ consists of units o_1, o_2 and o_3, representing three body atoms contained in P_1. Then the steps of the computation of the answer for the goal $G_0 = \leftarrow Q_1(f_1(a_1, a_2))$ from Example 1.6.1 can be performed by the following neural network:*

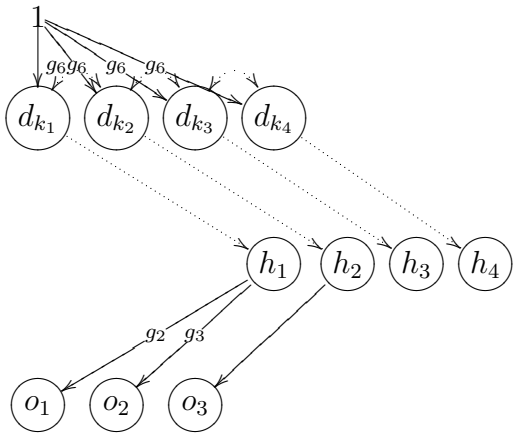


Step-by-step computations can be illustrated as follows. The network that we are going to use for computations must have the following architecture:



Gödel numbers of heads of clauses are encoded in the desired response values $d_{k_1} - d_{k_4}$; the Gödel numbers of atoms in the bodies are used as weights between layers $\mathbf{h}$ *and* $\mathbf{o}$*. Initial weights between layer* $\mathbf{k}$ *and layer* $\mathbf{h}$ *are set to* 0*. Units $k_1 - k_4$ are interconnected and form the Kohonen's layer.*

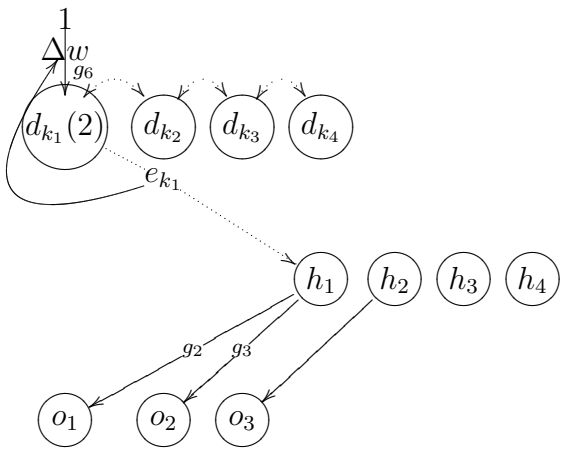
1. *Because we have only one goal* $G_0 = \leftarrow Q_1(f_1(a_1, a_2))$, *at time* $t = 1$, *four signals* $v_1 = v_2 = v_3 = v_4 = 1$ *are sent from one external source* i_1, *and* $w_{k_1 i_1}(1) = w_{k_2 i_1}(1) = w_{k_3 i_1}(1) = w_{k_4 i_1}(1) = g_6$, *the Gödel number of* $Q_1(f_1(a_1, a_2))$. *This is shown on the next diagram:*



Predicate threshold *function will set* $w_{k_3 i_1}(1) = w_{k_4 i_1}(1) = 0$. *Still two signals from one source are active and we apply* inverse Grossberg's law. *From the remaining active signals* v_1 *and* v_2 *we pick* v_1 *as unconditioned, and compute* $w^{new}_{k_2 i_1}(1) = w^{old}_{k_2 i_1}(1) + [v_1(1)v_2(1) - w^{old}_{k_2 i_1}(1)]U(v_2) = g_6 + [1 - g_6]1 = 1$. *Now we apply the* Predicate threshold *and get* $w^{new}_{k_2 i_1}(1) = 0$. *Because we have only one goal atom, at this stage neither of Grossberg's laws can be applied; Kohonen's competition is not applicable either. Thus, we proceed with* $p_{k_1}(1) = 1 g_6 = g_6$ *and* $v_{k_1} = g_6$.

Now error-correction learning can be initiated: $e_{k_1}(1) = \Delta w_{k_1 i_1}(1) = s(d_{k_1}(1) \ominus v_{k_1}(1)) = 01921$. *We are ready to update parameters using* $\Delta w_{k_1 i_1}(1) = e_{k_1}(1)$ *as follows:* $w_{k_1 i_1}(2) = w_{k_1 i_1}(1) \odot \Delta w_{k_1 i_1}(1) = g_6 \odot 01921 = g_6$; $w_{h_1 k_1}(2) = w_{h_1 k_1}(1) \oplus \Delta w_{k_1 i_1}(1) = 0 \oplus 01921 = 0801921$; $d_{k_1}(2) = d_{k_1}(1) \odot e_{k_1}(1) = g_1 \odot 01921 = 41531521701166$, *call the latter number* g_7.

This is shown on the next diagram:



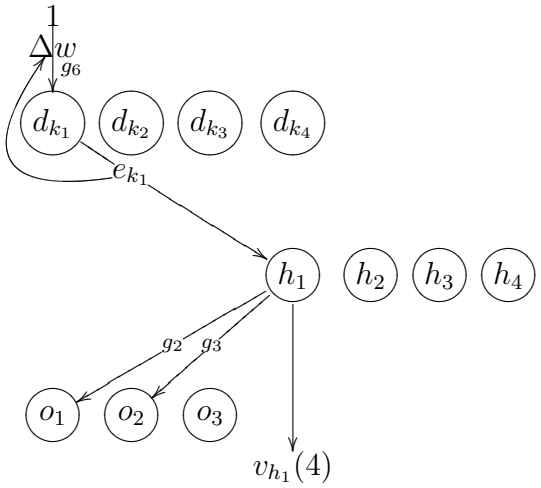
Next we increment time, put $t = 2$*, and compute* $p_{k_1}(2) = v_{k_1}(2) = g_6$*;* $\Delta w_{k_1 i_1}(2) = e_{k_1}(2) = s(d_{k_1}(2) \ominus v_{k_1}(2)) = 0119211$*. We update parameters using* $\Delta w_{k_1 i_1}(2) = e_{k_1}(2)$*, as follows:* $w_{k_1 i_1}(3) = w_{k_1 i_1}(2) \odot \Delta w_{k_1 i_1}(2) = g_6 \odot 0119211 = g_6$*;* $w_{h_1 k_1}(3) = w_{h_1 k_1}(2) \oplus \Delta w_{k_1 i_1}(2) = 080192180119211$*;*
$d_{k_1}(3) = d_{k_1}(2) \odot e_{k_1}(2) = g_7 \odot 0119211 = 41531521721166 = g_6$*.*

Increment t*, put* $t = 3$ *and compute* $p_{k_1}(3) = v_{k_1}(3) = g_6$*;* $\Delta w_{k_1 i_1}(3) = e_{k_1}(3) = 0$*. Thus,* $w_{k_1 i_1}(4) = w_{k_1 i_1}(3) = g_6$*;* $w_{h_1 k_1}(4) = w_{h_1 k_1}(3) \oplus e_{k_1}(3) = 080192180119211 80$*;* $d_{k_1}(4) = d_{k_1}(3) = g_6$*.*

Set $w_{k_1 i_1}(6) = 0$*,* $w_{h_1 k_1}(6) = 0$ *and apply* $w_{h_1 k_1}(4)$ *to change* $w_{k_2 i_1}(6) = w_{k_2 i_1}(1) \odot w_{h_1 k_1}(4)$*, do the same for* $w_{k_3 i_1}(6)$ *and* $w_{k_4 i_1}(6)$*. The details of the process of unification of* $Q_1(f_1(x_1, x_2))$ *and* $Q_1(f_1(a_1, a_2))$ *were also given in Example 7.3.1.*

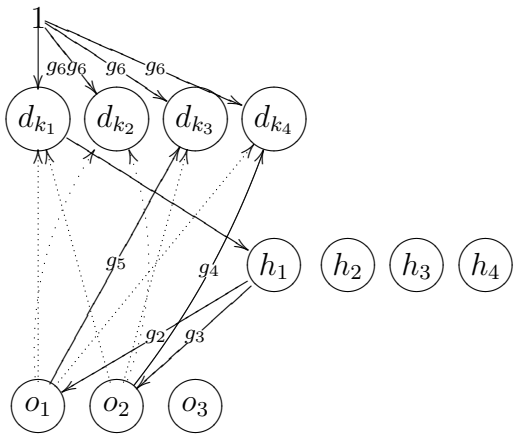
2. *Because* $e_{k_1}(3) = 0$*, at time* $t = 4$ *we compute the potential and value at the next level* $\mathbf{h}$*, as follows:* $p_{h_1}(4) = v_{k_1}(4) \odot w_{h_1 k_1}(4) = g_6 \odot 080192180119211 80 = g_6$*. And*

$v_{h_1}(4) = 08019218011921180$. *See the next diagram:*



3. *At time* $t = 5$ *we proceed with the computations at the next level* **o** *as follows:* $p_{o_1}(5) = w_{o_1h_1}(4) \odot v_{h_1}(4) = g_2 \odot 08019218011921180 = g_5$, *and compute* $v_{o_1}(5) = 1$. *Similarly for the unit* o_2, $p_{o_2}(5) = w_{o_2h_1}(4) \odot v_{h_1}(4) = g_3 \odot 08019218011921180 = g_4$, *and compute* $v_{o_2}(5) = 1$.

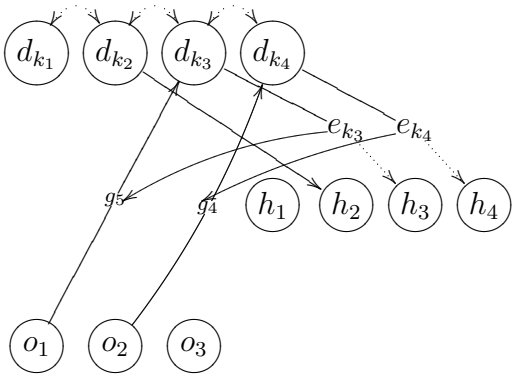
Weights between layers **o** *and* **k** *can be updated now as follows:* $w_{k_jo_1}(6) = w_{k_jo_1}(5) + (p_{o_1}(5)v_{o_1}(5)) = 0 + g_5 = g_5$ *and similarly for* o_2, $w_{k_jo_2}(6) = w_{k_jo_2}(5) + (p_{o_2}(5)v_{o_2}(5)) = 0 + g_4 = g_4$, *for each* $j \in \{1, \ldots, 4\}$. *It is shown on the next diagram:*



New iteration starts. *Apply* predicate threshold *and set* $w_{k_1o_1}(6) = w_{k_2o_1}(6) = w_{k_4o_1}(6) = w_{k_1o_2}(6) = w_{k_2o_2}(6) = w_{k_3o_2}(6) = 0$. *Neither of Grossberg's functions can be applied here; because* $w_{k_3o_1}(6) = g_5$ *and* $w_{k_4o_2}(6) = g_4$ *do not contain the symbol* 0 *(i.e, encode ground atoms), we do not impose Kohonen's competition anymore, and continue to propagate two input signals in parallel.*

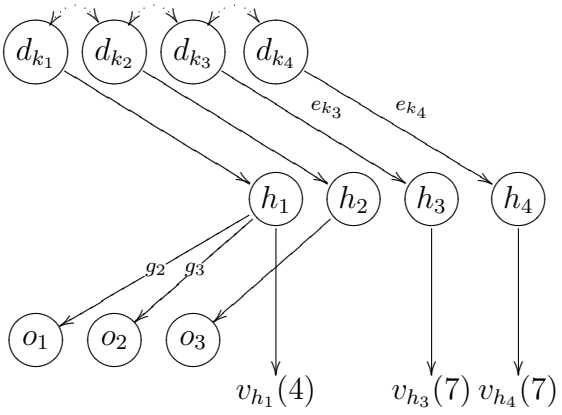
$p_{k_3}(6) = v_{k_3}(6) = g_5$;	$p_{k_4}(6) = v_{k_4}(6) = g_4$;
$e_{k_3}(6) = 0$;	$e_{k_4}(6) = 0$;
Update parameters:	*Update parameters:*
$w_{k_3o_1}(7) = w_{k_3o_1}(6) = g_5$;	$w_{k_4o_2}(7) = w_{k_4o_2}(6) = g_4$;
$w_{h_3k_3}(7) = 080$;	$w_{h_4k_4}(7) = 080$;
$p_{h_3}(7) = v_{k_3}(7) \odot w_{h_3k_3}(7) =$	$p_{h_4}(7) = v_{k_4}(7) \odot w_{h_4k_4}(7) =$
g_5, $v_{h_3}(7) = 080$;	g_4, $v_{h_4}(7) = 080$.

The next digram illustrates this table:



No further signals can be sent; no further learning functions can be applied, yet for every input signal v_i, we obtained the corresponding number v_{h_j} of length $l \geq 2$ which contains 0 *as its first and its last symbol. Computations stop. The answer is:* $v_{h_1}(4) =$ 08019218011921180, $v_{h_3}(7) = 080$, $v_{h_4}(7) = 080$. *It is easy to see that the output signals correspond to Gödel numbers of substitutions obtained as an answer for* $P_1 \cup G_0$ *in Example 1.6.1. Note that* $v_{h_3}(7) = v_{h_4}(7)$ *give the numbers of empty substitutions, and* $v_{h_1}(4)$ *gives the number of* $\theta_1\theta_2$.

The final stage of the computations is shown on the last diagram:



The animated version of this example can be found in [117].

Note that if we build a T_P-neural network for the logic program P_1 from Examples 1.3.2, 1.6.1, we would need to build it using infinitely many units in all the three layers, because U_{P_1} and B_{P_1} are infinite as shown in Examples 1.4.2 and 1.4.3. And, since such networks cannot be built in the real world, we would finally need to use some approximation theorem which is, in general, non-constructive.

7.5 Conclusions

Several conclusions can be made from Theorem 7.3.1 and Theorem 7.4.1.

SLD neural networks have finite architecture, but their effectiveness is due to several learning functions: two Grossberg's filter learning functions, error-correction learning functions, *Predicate threshold* function, Kohonen's competitive learning, and Hebbian learning function ϑ. The most important of these functions are those providing supervised learning and simulating the work of the algorithm of unification.

Learning laws implemented in SLD neural networks exhibit a "creative" component in the SLD-resolution algorithm. Indeed, the search for successful unification, choice of goal atoms and a program clause at each step of derivation are not fully determined by the resolution algorithm, but leave us (or a program interpreter) to make a personal choice, and in this sense, allow certain "creativity" in the decisions.

The fact that the process of unification is simulated by means of an error-correction learning algorithm reflects the fact that the unification algorithm is, in essence, a correction of one piece of data relative to another piece of data. This also suggests that unification is not a totally deductive algorithm, but an adaptive process.

Atoms and substitutions of the first-order language are represented in SLD neural networks internally via Gödel numbers of weights and other parameters. This distinguishes SLD neural networks from T_P-neural networks, where atoms are not encoded in the corresponding neural network directly, and only their truth values 0 and 1 are propagated. This suggests that SLD neural networks allow easier machine implementations comparing with T_P-neural networks.

The SLD neural networks described here work according to the depth-first search strategy. In general, these SLD neural networks can realize either depth-first or breadth-first search algorithms implemented in SLD-resolution, and this can be determined by imposing some conditions on the choice of unconditioned stimulus during the use of Grossberg's laws and Kohonen's competition in layer **k**.

The SLD neural networks we have presented here, unlike T_P-neural networks, allow almost straightforward generalisations to higher-order logic programs. Further extension of these neural networks to higher-order Horn logics, hereditary Harrop logics, linear logic programs, etc. may lead to other novel and interesting results.

Chapter 8

Conclusions and Further Work

8.1 Conclusions

In the Thesis, we have shown that Logic, Logic Programming and First-Order Deduction can be expressed in Connectionist neural networks using *learning functions* recognised in Neurocomputing.

Namely, we have constructed two types of *learning* neural networks: the first type performs computations of the semantic operator defined for logic programs for reasoning with uncertainty (BAPs); and the second type of neural networks is an interpreter of the algorithm of SLD-resolution for conventional definite logic programs. We called the first approach *extensive*, because it extended constructions and techniques introduced in [98, 99] to the new class of logic programs. We characterised the second approach as *intensive*, because it introduced an absolutely novel construction of neural networks, where learning techniques of Neurocomputing were effectively used in order to perform the algorithm of SLD-resolution by means of finite neural networks.

Chapters 1, 2, 5 gave background material and a literature survey on Logic, Logic Programming and Neural Networks sufficient for developing and evaluating the discussions

of the remaining chapters of the thesis.

The Chapters 3 and 4 supported the first, extensive, approach. Namely, they developed declarative and operational semantics for Bilattice-Based Annotated Logic Programs (BAPs) in order to use them for building neural networks simulating the $\mathcal{T}_P$-operator for BAPs. BAPs have proven to be a very general and interesting class of logic programs, well worthy to be studied for numerous theoretical and practical reasons. We showed in Section 4.4 that BAPs are expressive enough to incorporate many classes of annotation-free and implication-based logic programs interpreted in lattice and bilattice structures. We established a sound and complete SLD-resolution for BAPs, which is the first sound and complete proof procedure for (bi)lattice-based logic programs of this generality.

In Chapter 6 we have built Connectionist neural networks simulating the $\mathcal{T}_P$-operator for BAPs, we called them $\mathcal{T}_P$-neural networks. The resulting neural networks were capable of performing Hebbian learning. These neural networks supported the first, extensive, approach to developing the Connectionist neural networks of [98, 99]. The main virtue of this approach is that it exhibits some nontrivial properties of BAPs, and also suggests that different types of non-classical, many-valued and (bi)lattice-based logics and logic programs may have interesting implementations in Neurocomputing, and the neural networks built to simulate them can effectively employ learning techniques. Due to the generality and expressiveness of BAPs, the neural networks built in Chapter 6 can be used, with minor modifications, for different logic programs for reasoning with uncertainty described in [17, 50, 49, 203, 107, 108, 141, 143, 142].

This led us to compare, in Section 6.4, the neural networks for BAPs we introduced in Chapter 6 with the neural networks for many-valued annotation-free logic programs introduced in [185, 135]. This comparison showed that $\mathcal{T}_P$ neural networks can compute

all that the neural networks of [185, 135] could, but the Hebbian learning used to build $\mathcal{T}_P$-neural networks significantly improves computational complexity (both time and space) of these neural networks comparing with those of [185, 135]. This further supported the usefullness of learning techniques developed in Neurocomputing, and suggested that learning can be effectively used in the field of Neuro-Symbolic Integration.

Simplicity of architecture of $\mathcal{T}_P$-neural networks comparing with neural networks supporting the intensive approach, might be seen as their advantage, especially in case if they are built for propositional logic programs. However, for first-order logic programs with individual or annotation function symbols these neural networks are not practical, although the approximation theorem of Section 6.3 establishes that the approximation of the least fixed points of $\mathcal{T}_P$ by finite neural networks is possible.

Chapter 7 supported the second, intensive, approach to building Neural-Symbolic Networks. The intensive approach is more innovative in that it advocates the method of SLD-resolution, rather than fixpoint semantics, as a suitable formalism for Neural-Symbolic Integration. This approach seems to be closer to the ideas of Neurocomputing, because the resulting neural networks are finite, their size can be constructively determined, and they effectively use different types of learning. Moreover, this intensive approach resulted in the first Neural-Symbolic interpreter for *goal-oriented deduction*, namely, the SLD-refutation algorithm. SLD neural networks have computational benefits too: both the unification algorithm and breadth-first/depth-first search which are involved in SLD-refutation are known to be P-complete ([40],[70]), and SLD neural networks which simulate SLD-refutation will inherit these characteristics.

Returning to the **main question** we posed in the Introduction, we conclude the following: Connectionist neural networks supporting Neuro- Symbolic Integration were shown

to be able to perform deductive reasoning, see Chapters 6, 7. But, what is more important, the learning neural networks we have built in these chapters exhibit some non-trivial computational properties of the logic counterparts they simulate. Thus, we can conclude that both the semantic operator for BAPs and conventional SLD-resolution bear certain properties which can be realized by the learning algorithms of Neurocomputing. This conclusion sounds like an apology for symbolic (deductive) theories, because it suggests that symbolic reasoning which we normally call "deductive" does not necessarily imply the lack of learning and spontaneous self-organisation in the sense of Neurocomputing.

We have illustrated that methods of Neurocomputing and Connectionism can be effectively integrated, and this integration brings theoretical, practical, and methodological benefits. Therefore, we would propose to develop this integrative approach in the future, as follows.

8.2 Further Work

The results of the Chapter 7 allow us to talk about the practical use of artificial neural networks for purposes of automated reasoning and logic programming. The future work may be to develop and evaluate this use.

Particular research objectives may include:

1. To represent the *operational semantics* of a logic programming language L as an artificial neural network, where L is one of the following:

   - Hereditary Harrop Logic;
   - Higher-Order Horn Logic;
   - Linear Logic Programming (Lolli or Lygon);

and may be some others. See also Table 1.3.1 for the list of plausible logic programming languages.

2. To represent decision and semi-decision procedures of a logic L as a neural network, where L is one of the following:

   - Classical zero-order logic;
   - Classical first-order logic;
   - Intuitionistic logic;
   - Intermediate logics, such as Gödel-Dummett logic;
   - Modal and Description logic;
   - Fixed point logics (i.e., Dynamic logic);
   - Logics with Uncertainty, Probability, Fuzziness...

3. To evaluate the effectiveness of these representations compared with orthodox methods such as resolution and analytic tableaux.

4. To refine these representations in the light of evaluations.

5. To seek abstract and more general representations of which the above representations are special cases.

In the last paragraph of this thesis, we would like to return to the two citations of Kurt Gödel we started with, and to the distinction between *static and finite machine deduction* and *infinite and developing human mind* they supported. We would like this thesis to contribute to this long-lasting discussion in the style of George Boole. We believe that Boole, who proposed logic in order to formalise "*the laws of thought*" [18] did not

intend the strict distinction between learning and deduction. In Chapters 6 and 7 we have suggested two interesting and natural examples when logical *deduction* and neural network *learning* meet and complement each other.

Bibliography

[1] S. Abramsky, D. Gabbay, and T.S. Mabaum. *Handbook of Logic in Computer Science*. Oxford University Press, 1995.

[2] J.A. Anderson and E. Rosenfield. *Neurocomputing: Foundations of Research*. MIT Press, Camridge MA, 1988.

[3] J.-M. Andreoli and R. Pareschi. Linear objects: Logical processes with built-in inheritance. *New Generation Computing*, 9:445–473, 1991.

[4] Jean-Mark Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3):297–347, 1992.

[5] R. Andrews, J. Diedrich, and A.B. Tickle. A survey and critique of techniques for exatrcting rules from trained artificial neural networks. *Knowledge-Based Systems*, 8(6):373–389, 1995.

[6] K. R. Apt and M. van Emden. Contributions to the theory of logic programming. *Journal of the Association for Computational Mach.*, 29:841–862, 1982.

[7] Aristotle. *Organon*. Loeb Classical library, E.S. Forster, (ed)., 1960.

[8] Steve Awodey. In memoriam: Saunders Mac Lane (1909–2005). *The Bulletin of Symbolic Logic*, 13(1):115 – 119, 2007.

[9] Fahiem Bacchus. Lp, a logic for representing and reasoning with statistical knowledge. *Computational Intelligence*, 6:209–231, 1990.

[10] S. Bader, P. Hitzler, and A. Witzel. Integrating first-order logic programs and connectionist systems — a constructive approach. In A. S. d'Avila Garcez, J. Elman, and P. Hitzler, editors, *Proceedings of the IJCAI-05 Workshop on Neural-Symbolic Learning and Reasoning, NeSy'05*, Edinburgh, UK, 2005.

[11] M. Baldoni. *Normal Multimodal Logics: Automatic Deduction and Logic Programming extension.* PhD thesis, Torino, Italy, 2003.

[12] M. Baldoni, L. Giordano, and A. Martelli. A multimodal logic to define modules in logic programming. In D.Miller, editor, *Proc. of the International Logic Programming Symposium, ILPS'93*, pages 473–487. MIT Press, 1993.

[13] J.A. Barnden. On short term information processing in connectionist theories. *Cognition and Brain Theory*, 7:25–59, 1984.

[14] J. Barwise. *Handbook of Mathematical Logic.* North Holland, 1987.

[15] N. D. Belnap. A useful four-valued logic. In John Michael Dunn and G. Epstein, editors, *Modern Uses of Multiple-Valued Logic*, pages 8–37. Dordrecht, Reidel, 1977.

[16] G. Birkhoff. *Lattice Theory*, volume XXV. American Mathematical Society, 1973.

[17] S. Bistarelli, U. Montanari, and F.Rossi. Semiring-based constraint logic programming: Syntax and semantics. *ACM Transactions on Programming Languages and Systems*, 23(1):1–29, 2001.

[18] George Boole. *An investigation of the Laws of Thought on Which are Founded the Mathematical Theories of Logic and Probabilities.* Macmillan, 1854.

[19] K. A. Bowen. Programming with full first-order logic. *Machine Intelligence*, 10:421–440, 1982.

[20] Jacques Calmet, James J. Lu, Maria Rodriguez, and Joachim Schü. Signed formula logic programming: Operational semantics and applications. In *Proceedings of the Ninth International Symposium on Foundations of Intelligent Systems*, volume 1079 of *Lecture Notes in Artificial Intelligence*, pages 202–211, Berlin, June 9-13 1996. Springer.

[21] Luca Cardelli. Brane calculi. In V. Danos and V. Schachter, editors, *International Conference Computational Methods in Systems Biology CMSB'04, Paris, France, 2004*, volume 3082 of *LNCS*, pages 257–278. Springer, 2005.

[22] Alonzo Church. *Introduction to Mathematical Logic*. Princeton, 1944.

[23] K.L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, N.Y., 1978.

[24] K.L. Clark. Predicate logic as a computational formalism. Technical Report DOC 79/59, Department of Computing, Imperial College, 1979.

[25] W. F. Clocksin and C. S. Mellish. *Programming in Prolog*. Springer-Verlag, Berlin, West Germany, 1981.

[26] A. Colmerauer, H. Kanoui, P. Roussel, and R. Pasero. Un systeme de communications homme-machine en francais. Technical report, Groupe de Recherche en Intelligence Artificielle, Université d'Aix-Mareselle, 1973.

[27] David Mc. Cormick. Brain calculus: Neural integration and persistent activity. *Nature Neuroscience*, 4:113–114, 2001.

[28] Carlos Viegas Damásio and Lu'is Moniz Pereira. Sorted monotonic logic programs and their embeddings. In *Proceedings of the 10th International Conference on Information Processing and Management of Uncertainty in Knowledge-Based Systems (IPMU-04)*, pages 807–814, 2004.

[29] Arthur d'Avila Garcez, K. B. Broda, and D. M. Gabbay. Symbolic knowledge extraction from trained neural networks: A sound approach. *Artificial Intelligence*, 125:155–207, 2001.

[30] Arthur d'Avila Garcez, K. B. Broda, and D. M. Gabbay. *Neural-Symbolic Learning Systems: Foundations and Applications*. Springer-Verlag, 2002.

[31] Arthur d'Avila Garcez and D. M. Gabbay. Fibring neural networks. In *Proceedings of the 19th National Conference on Artificial Intelligence*, pages 342–347. AAAI Press/MIT Press, San Jose, California, USA, 2004.

[32] Arthur d'Avila Garcez, L.C. Lamb, and D. M. Gabbay. A connectionist inductive learning system for modal logic programming. In *Proc. of the IEEE International Conference on Neural Information Processing ICONIP'02, Singapore*. 2002.

[33] Arthur d'Avila Garcez and Gerson Zaverucha. The connectionist inductive learning and logic programming system. *Applied intelligence, Special Issue on Neural networks and Structured Knowledge*, 11(1):59–77, 1999.

[34] Arthur d'Avila Garcez, Gerson Zaverucha, and Luis A.V. de Carvalho. Logical inference and inductive learning in artificial neural networks. In C. Hermann, F. Reine, and A. Strohmaier, editors, *Knowledge Representation in Neural Networks*, pages 33–46. Logos Verlag, Berlin, 1997.

[35] Martin Davis and Hillary Putnam. A computing procedure for quantification theory. *ACM*, 7:201–215, 1960.

[36] John W. Dawson. Gödel and the origins of computer science. In A. Beckmann, U. Berger, B. Löwe, and J.V. Tucker, editors, *Logical Approaches to Computational Barriers, CiE'06*, volume 3988 of *LNCS*, pages 133–137, 2006.

[37] F. Debart, P. Enjabert, and M. Lescot. Multimodal logic programming using equational and order-sorted logic. *Theoretical computer science*, 105(1):141–166, 1992.

[38] Liya Ding. Neural prolog - the concepts, construction and mechanism. In *Proceedings of the 3rd Int. Conference Fuzzy Logic, Neural Nets, and Soft Computing*, pages 181–192, 1995.

[39] J. M. Dunn. Intuitive semantics for first-degree entailments and coupled trees. *Philosophical studies*, 29:149–168, 1976.

[40] C. Dwork, P.C. Kanellakis, and J.C. Mitchell. On the sequential nature of unification. *Journal of Logic Programming*, 1:35–50, 1984.

[41] H.D. Ebbinghaus, J. Flum, and W. Thomas. *Mathematical Logic*. Springer-Verlag, Berlin, 1984.

[42] G.M. Edelman. *Neural Darwinism, the Theory of Neuronal Group Selection*. Basic Books, 1987.

[43] H. B. Enderton. *A Mathematical Introduction to Logic*. Academic Press, 1972.

[44] Ronald Fagin, Joseph Y. Halpern, and Nimrod Megiddo. A logic for reasoning about probabilities. *Information and Computation*, 87(1,2):78–128, 1990.

[45] J.A. Feldman and D.H. Ballard. Connectionist models and their properties. *Cognitive Science*, 6(3):205 – 254, 1982.

[46] Stacy E. Finkelstein, Peter Freyd, and James Lipton. Logic programming in tau categories. In Leszek Pacholski and Jerzy Tiuryn, editors, *Computer Science Logic'94*, volume 933 of *Lecture Notes in Computer Science*. Springer, 1995.

[47] I. Fischer, F. Henneche, C. Bannes, and A. Zell. Java neural network simulator. user manual. Technical report, University of Tübingen, Wilhelm-Schickard Institute for Computer Science, Department of Computer Architecture, 2002. www.-ra.informatik.uni-tuebingen.de/software/JavaNNS.

[48] Melvin Fitting. A Kripke/Kleene semantics for logic programs. *Journal of logic programming*, 2:295–312, 1985.

[49] Melvin Fitting. Bilattices in logic programming. In G. Epstein, editor, *The twentieth International Symposium on Multiple-Valued Logic*, pages 238–246. IEEE, 1990.

[50] Melvin Fitting. Bilattices and the semantics of logic programming. *Journal of logic programming*, 11:91–116, 1991.

[51] Melvin Fitting. Kleene's logic, generalized. *Journal of Logic and Computation*, 1:797–810, 1992.

[52] Melvin Fitting. Kleene's three-valued logics and their children. *Fundamenta informaticae*, 20:113–131, 1994.

[53] Melvin Fitting. Metric methods: Three examples and a theorem. *The Journal of Logic Programming*, 21:113–127, 1994.

[54] Melvin Fitting. Fixpoint semantics for logic programming — a survey. *Theoretical computer science*, 278(1-2):25–51, 2002.

[55] Melvin Fitting. Bilattices are nice things. *Self-Reference*, pages 53–77, 2006.

[56] F. L. Gottlob Frege. *Begriffsschrift, eine der arithmetischen nachgebildete Formelsprac he des reinen Denkens (Concept Notation, the Formal Language of the Pure Thought like that of Arithmetic).* Halle a. S., 1879.

[57] F. L. Gottlob Frege. *Logical Investigations. (Collected works 1918–1923).* Blackwells, 1975 (collected works 1918–1923). P. Geach, editor.

[58] K.-I. Funahashi. On the approximate realisation of continuous mappings by neural networks. *Neural Networks*, 2:183–192, 1989.

[59] Colin Fyfe. Artificial neural networks and information theory. Lecture Course, 2000. Department of Computing and Information Systems, The University of Paisley, UK.

[60] D. M. Gabbay. Modal and temporal logic programming. In A. Galton, editor, *Temporal logics and their applications*, pages 197–237. Academic press, 1987.

[61] O. N. Garcia and M. Moussavi. A six-valued logic for representing incomplete knowledge. In G. Epstein, editor, *Proceedings of Twentieth International symposium on Multiple-valued logic*, pages 110–114. IEEE Computer Society Press, 1990.

[62] O. Gasquet. Optimization of deduction for multi-modal logics. In Masuch, Marx, and Plòs, editors, *Applied logic: How, What and Why?* Kluwer Academic Publishers, 1993.

[63] Gerhard Gentzen. Untersuchungen über das logische schliessen. *Math. Zeit.*, 39:176–210, 1934.

[64] P. C. Gilmore. A proof method for quantification theory. *IBM J. Res. Development*, 4:28–35, 1960.

[65] Mathew L. Ginsberg. Multivalued logics: a uniform approach to reasoning in artificial intelligence. *Computational Intelligence*, 4:265–316, 1988.

[66] Kurt Gödel. On formally undecidable propositions of *Principia Mathematica* and related systems (1931). In J. W. Dawson, S. Feferman, and S. C. Kleene, editors, *Collected Works: Volume 1.* Oxford University Press, 1986.

[67] Kurt Gödel. *Collected Works: Volume 1-4.* Oxford University Press, 1986–2003. J.W. Dawson, S. Feferman, S.C. Kleene, editors.

[68] George Grätzer. *General Lattice Theory.* Birkauser Verlag, Basel, Switzerland, 1978.

[69] C. Green. Applications of theorem proving to problem solving. In *IJCAI'69*, pages 219–239, Washington D.C., 1969.

[70] R. Greenlaw, H. J. Hoover, and W.L. Ruzzo. A compendium of problems complete for P. Technical Report TR 91-05-01, Department of Computer Science and Engineering, University of Washington, 1991.

[71] S. Grossberg. Embedding fields: A theory of learning with physiological implications. *J. Math. Psych.*, 6:209–239, 1969.

[72] Hans Werner Güsgen and Steffen Hölldobler. Connectionist inference systems. In B. Fronhöfer and G. Wrightson, editors, *Parallelization in Inference Systems*. Springer, LNAI *590*, 1992.

[73] Reiner Hähnle. Towards an efficient tableaux proof procedure for multiple-valued logics. In *Workshop in Computer Science and Logic, Heildelberg*, volume 533 of *lecture Notes in Computer Science*, pages 248–260, 1990.

[74] Reiner Hähnle. Commodious axiomatizations of quantifiers in multiple-valued logic. *Studia Logica*, 61(1):101–121, 1998.

[75] Reiner Hähnle and Ganzalo Escalado-Imaz. Deduction in many-valued logics: a survey. *Mathware and soft computing*, IV(2):69–97, 1997.

[76] Theodore Hailperin. Probability logic. *Notre Damme Journal of Formal Logic*, 25(3), July 1984.

[77] Joseph Halpern. *Reasoning about uncertainty*. MIT Press, 2003.

[78] B. Hammer and eds. P. Hitzler. *Perspectives of Neural-Symbolic Integration*. Studies in Computational Intelligence. Springer Verlag, 2007.

[79] A. Hansson, S. Haridi, and S.-A. Tärnlund. Properties of a logic programming language. *Logic Programming*, pages 267–280, 1982.

[80] J. A. Harland and D. J. Pym. A uniform proof-theoretic investigation of linear logic programming. *Journal of Logic and Computation*, 4(2):175–207, 1994.

[81] P.J. Hayes. Computation and deduction. In *Proceedings MFCS Conf.*, pages 105–118, Czechoslovak Academy of Sciences, 1973.

[82] Simon Haykin. *Neural Networks. A Comprehensive Foundation*. Macmillan College Publishing Company, 1994.

[83] D. Hebb. *The Organisation of Behaviour*. Wiley, New York, 1949.

[84] Robert Hecht-Nielsen. Counterpropagation networks. In *Proc. of the Int. Conf. on Neural Networks*, volume II, pages 19–32. IEEE Press, New York, 1987.

[85] Robert Hecht-Nielsen. *Neurocomputing*. Addison-Wesley, 1990.

[86] J. Herbrand. Investigations in proof theory. In J. van Heijenoort, editor, *From Frege to Gödel: A source book in Mathematical Logic, 1879-1931*, pages 525–581. Harvard University Press, Cambridge, Mass., 1967.

[87] David Hilbert. Axiomatic thought (1918). In William B. Ewald, editor, *From Kant to Hilbert: A Source Book in the Foundations of Mathematics, 2 vols.* Oxford Univ. Press, 1996.

[88] David Hilbert. The foundations of mathematics (1927). In William B. Ewald, editor, *From Kant to Hilbert: A Source Book in the Foundations of Mathematics, 2 vols.* Oxford Univ. Press, 1996.

[89] David Hilbert. The grounding of elementary number theory (1931). In William B. Ewald, editor, *From Kant to Hilbert: A Source Book in the Foundations of Mathematics, 2 vols.* Oxford Univ. Press, 1996.

[90] David Hilbert. Logic and knowledge of nature (1930). In William B. Ewald, editor, *From Kant to Hilbert: A Source Book in the Foundations of Mathematics, 2 vols.* Oxford Univ. Press, 1996.

[91] David Hilbert. The logical foundations of mathematics (1923). In William B. Ewald, editor, *From Kant to Hilbert: A Source Book in the Foundations of Mathematics, 2 vols.* Oxford Univ. Press, 1996.

[92] David Hilbert. The new grounding of mathematics: First report (1922). In William B. Ewald, editor, *From Kant to Hilbert: A Source Book in the Foundations of Mathematics, 2 vols.* Oxford Univ. Press, 1996.

[93] David Hilbert. On the foundations of logics and arithmetic (1904). In William B. Ewald, editor, *From Kant to Hilbert: A Source Book in the Foundations of Mathematics, 2 vols.* Oxford Univ. Press, 1996.

[94] Pascal Hitzler, S. Hölldobler, and Anthony Karel Seda. Logic programs and connectionist networks. *Journal of Applied Logic*, 2(3):245–272, 2004.

[95] S. Hölldobler and F. Kurfess. CHCL – A connectionist inference system. In B. Fronhöfer and G. Wrightson, editors, *Parallelization in Inference Systems*, pages 318 – 342. Springer, LNAI *590*, 1992.

[96] Stephen Hölldobler. A structured connectionist unification algorithm. In *Proceedings of the AAAI National Conference on Artificial Intelligence*, pages 587–593, 1990.

[97] Stephen Hölldobler. Towards a connectionist inference system. *Computational Intelligence*, III:25–38, 1991.

[98] Stephen Hölldobler and Yvonne Kalinke. Towards a massively parallel computational model for logic programming. In *Proceedings of the ECAI94 Workshop on Combining Symbolic and Connectionist Processing*, pages 68–77. ECCAI, 1994.

[99] Stephen Hölldobler, Yvonne Kalinke, and H. P. Storr. Approximating the semantics of logic programs by recurrent neural networks. *Applied Intelligence*, 11:45–58, 1999.

[100] J.J. Hopfield. Neural networks and physical systems with emergent collective computational abilities. In *Proc. Natl. Acad. Sci.*, volume 79, pages 2554–2558, 1982.

[101] J.J. Hopfield. Neurons with graded response have collective computational properties like those of two-state neurons. In *Proc. Natl. Acad. Sci.*, volume 81, pages 3088–3092, 1984.

[102] Michael Huth and Mark Ryan. *Logic in Computer Science*. Cambridge University Press, 2004.

[103] P. Indyk. Optimal simulation of automata by neural nets. In E.W. Mayr and C. Puech, editors, *Proc. of the Twelfth Annual Symposium on theoretical aspect of Computer Science*, LNCS, page 337, 1995.

[104] Y. Kalinke. *Ein Massive Parallels Berechningsmodell für Normale Logische Programme*. PhD thesis, Department of Computer Science, Dresden University of Technology, 1994.

[105] M. Karpinski and A. Macintyre. Polynomial bounds for vc dimension of sigmoidal and general pfaffian neural networks. *J. Comp. Syst. Sci.*, 54:169–176, 1997.

[106] M. Karpinski and Angus Macintyre. Polynomial bounds for vc dimension of sigmoidal neural networks. In *Proceedings of 27th ACM STOC*, pages 337–341, 1995.

[107] Michael Kifer and Eliezer L. Lozinskii. RI: A logic for reasoning with inconsistency. In *Proceedings of the 4th IEEE Symposium on Logic in Computer Science (LICS)*, pages 253–262, Asilomar, 1989. IEEE Computer Press.

[108] Michael Kifer and V. S. Subrahmanian. Theory of generalized annotated logic programming and its applications. *Journal of logic programming*, 12:335–367, 1991.

[109] J. Kilian and H.T. Siegelmann. The dynamic universality of sigmoidal neural networks. *Information and Computation*, 128(1):48–56, 1996.

[110] Y. Kinoshita and A. J. Power. A fibrational semantics for logic programs. In Roy Dyckhoff, Heinrich Herre, and Peter Schroeder-Heister, editors, *Proceedings of the Fifth International Workshop on Extensions of Logic Programming*, volume 1050 of *LNAI*, Leipzig, Germany, 1996. Springer.

[111] Stephen Cole Kleene. *Introduction to Metamathematics*. D. van Nostrand Company, Inc, New York, Toronto, 1952.

[112] Werner Kluge. *Abstract Computing Machines*. Springer, 2005.

[113] T. Kohonen. *Self-Organization and Associative memory*. Springer-Verlag, Berlin, second edition edition, 1988.

[114] P. Koiran, M. Cosnard, and M.Garzon. Computability with low-dimensional dynamic systems. *Theoretical Computer Science*, 132:113–128, 1994.

[115] A.N. Kolmogorov. On the representation of continuous functions of many variables by superposition of continuous functions of one variable and addition. In *Dokl. Akad. Nauk USSR*, volume 114, pages 953–956, 1957.

[116] E. Komendantskaya. Bilattice-based logic programs: Neural computations and automated reasoning. CiE'06 Presentation, available at `www.cs.ucc.ie/~ek1/slides.html`.

[117] E. Komendantskaya. First-order deduction in neural networks. LATA'07 Presentation, available at `www.cs.ucc.ie/~ek1/slides.html`.

[118] E. Komendantskaya. Categorical analysis of many-valued logic programming. Technical report, Boole Centre For Reasearch in Informatics, 2007. To appear as BCRI Preprint, 36 pages.

[119] E. Komendantskaya, M. Lane, and A. Seda. Connectionist representation of multi-valued logic programs. In Barbara Hammer and Pascal Hizler, editors, *Perspectives*

of Neural-Symbolic Integration, Computational Intelligence, pages 259–289. Springer Verlag, 2007. To appear.

[120] Ekaterina Komendantskaya. Learning and deduction in neural networks and logic. Technical report, Boole Centre for Research in Informatics, 2006. 34 pages. Appeared as BCRI preprint www.bcri.ucc.ie/BCRI_62.pdf.

[121] Ekaterina Komendantskaya. A many-sorted semantics for many-valued annotated logic programs. In *Proceedings of the Fourth Irish Conference on the Mathematical Foundations of Computer Science and Information Technology (MFCSIT)*, pages 225–229, Cork, Ireland, August 1– August 5 2006.

[122] Ekaterina Komendantskaya. First-order deduction in neural networks. In *Proceedings of the 1st International Conference on Language and Automata Theory and Applications (LATA'07)*, pages 307–319, Tarragona, Spain, 2007.

[123] Ekaterina Komendantskaya. A sequent calculus for bilattice-based logic and its many-sorted representation. In N. Olivetti, editor, *Proceedings of the International Conference on Automated Reasoning with Analytic Tableaux and Related Methods, TABLEAUX'07*, volume 4548 of *LNAI*, pages 165–182, Aix en Provance, 3– 6 July, 2007. Springer.

[124] Ekaterina Komendantskaya and Vladimir Komendantsky. On uniform proof-theoretical operational semantics for logic programming. In *Directions in Universal Logic. (Post-Proceedings volume of the 1st World Congress on Universal Logic (UNILOG-05), 26 March – 3 April, 2005, Montreux, Switzerland)*. Polymetrica, 2007. In print.

[125] Ekaterina Komendantskaya and John Power. Fibrational semantics for many-valued logic programs, 2007. Submitted.

[126] Ekaterina Komendantskaya and Anthony Karel Seda. Declarative and operational semantics for bilattice-based annotated logic programs. In *Proceedings of the Fourth Irish Conference on the Mathematical Foundations of Computer Science and Information Technology (MFCSIT)*, pages 229–233, Cork, Ireland, August 1– August 5 2006.

[127] Ekaterina Komendantskaya and Anthony Karel Seda. Logic programs with uncertainty: neural computations and automated reasoning. In *Proceedings of the International Conference Computability in Europe*, pages 170–182, Swansea, Wales, June 30– July 5 2006.

[128] Ekaterina Komendantskaya and Anthony Karel Seda. Sound and complete sld-resolution for bilattice-based annotated logic programs. In *Submitted to Post-Proceedings of the Fourth Irish Conference on the Mathematical Foundations of Computer Science and Information Technology (MFCSIT)*, page 18 pages, Cork, Ireland, August 1– August 5 2006.

[129] Ekaterina Komendantskaya, Anthony Karel Seda, and Vladimir Komendantsky. On approximation of the semantic operators determined by bilattice-based logic programs. In *Proceedings of the Seventh International Workshop on First-Order Theorem Proving (FTP'05)*, pages 112–130, Koblenz, Germany, September 15–17 2005.

[130] Vladimir Komendantsky and Anthony Karel Seda. Computation of normal logic programs by fibring neural networks. In *Proceedings of the Seventh International Workshop on First-Order Theorem Proving (FTP'05)*, pages 97–112, Koblenz, Germany, September 15–17 2005.

[131] R. A. Kowalski. Predicate logic as a programming language. In *Information Processing 74*, pages 569–574, Stockholm, North Holland, 1974.

[132] Dexter Kozen. *Theory of Computation*. Springer, 2006.

[133] S. Kripke. Outline of a theory of truth. *The Journal of Philosophy*, 72:690–716, 1975.

[134] Laks V. S. Lakshmanan and Fereidoon Sadri. On a theory of probabilistic deductive databases. *Theory and Practice of Logic Programming*, 1(1):5–42, January 2001.

[135] Máire Lane. $\mathfrak{C}$-*Normal Logic Programs and Semantic Operators: Their Simulation by Artificial Neural Networks*. PhD thesis, Department of Mathematics, University College Cork, Cork, Ireland, 2006.

[136] T. E. Lange and M. G. Dyer. High-level inferencing in a connectionist network. *Connection Science*, 1:181 – 217, 1989.

[137] J.-L. Lassez and M.J. Maher. Closures and fairness in the semantics of programming logic. *Theoretical Computer Science*, 29:167–184, 1984.

[138] J.W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 2nd edition, 1987.

[139] J.W. Lloyd and R.W. Topor. Making prolog more expressive. *Journal of Logic Programming*, 1(3):225 – 240, 1984.

[140] G.G. Lorentz. The 13-th problem of Hilbert. In *Proc. of Symposia in Pure Math.*, volume 28, pages 419–430. Am. Math. Soc., 1976.

[141] James J. Lu. Logic programming with signs and annotations. *Journal of Logic and Computation*, 6(6):755–778, 1996.

[142] James J. Lu, Neil V. Murray, and Erik Rosenthal. A framework for automated reasoning in multiple-valued logics. *Journal of Automated Reasoning*, 21(1):39–67, 1998.

[143] James J. Lu, Neil V. Murray, and Erik Rosenthal. Deduction and search strategies for regular multiple-valued logics. *Journal of Multiple-valued logic and soft computing*, 11:375–406, 2005.

[144] Jan Lukasiewicz. *Elements of Mathematical Logic*. Macmillan, NY, 1964.

[145] T. Lukasiewicz. Many-valued disjunctive logic programs with probabilistic semantics. In M. Gelfond, N. Leone, and G. Pfiefer, editors, *LPNMR'99*, volume 1730 of *LNAI*, pages 277–289, 1999.

[146] S. MacLane. *Categories for the working mathematician*. Graduate Texts in Mathematics. Springer-Verlag, Berlin, 1971.

[147] Maria Manzano. Introduction to many-sorted logic. In K. Meinke and J. V. Tucker, editors, *Many-Sorted logic and its Applications*, pages 3–88. John Wiley and Sons, UK, 1993.

[148] David Marker. *Model Theory: An Introduction*. Graduate Texts in Mathematics. Springer-Verlag, N.Y., 2002.

[149] Gary Markus. *The Algebraic Mind: Integrating Connectionism and Cognitive Science*. Cambridge, MA: MIT Press, 2001.

[150] Per Martin-Löf. *Intuitionistic Type Theory*. Napoli, Bibliopolis, 1984.

[151] W.S. McCulloch and W. Pitts. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Math. Bio.*, 5:115–133, 1943.

[152] K. Meinke and J. V. Tucker. *Many-Sorted logic and its Applications*. John Wiley and Sons, UK, 1993.

[153] E. Mendelson. *Introduction to Mathematical Logic*. Van Nostrand, Princeton, N.Y., 2nd edition, 1979.

[154] D. Miller and G. Nadathur. A logic-programming approach to manipulating formulas and programs. In Seif Haridi, editor, *IEEE Symposium on Logic Programming*, pages 379–388, San-Francisco, 1987.

[155] Dale Miller. A theory of modules for logic programming. In *IEEE Symposium on Logic Programming*, pages 106–114, 1986.

[156] Dale Miller. A logical analysis of modules in logic programming. *Journal of Logic Programming*, 6:79–108, 1989.

[157] Dale Miller. A multiple-conclusion meta-logic. *Theoretical Computer Science*, 165(1), 1996.

[158] Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. *Uniform Proofs as a Foundation for Logic Programming*, volume 51 of *Annals of Pure and Applied Logic*, pages 125–157. Elsevier, 1991.

[159] M. Minsky. *Neural Nets and the Brain - Model Problem*. PhD thesis, Princeton University, Princeton NJ, 1954.

[160] M. Minsky and S. Papert. *Perceptrons*. MIT Press, Cambridge, 1969.

[161] Gopalan Nadathur and Dale Miller. An overview of lambda-prolog. In *Fifth International Logic Programming Conference*, pages 810–827, Seattle, 1988. MIT Press.

[162] Gopalan Nadathur and Dale Miller. Higher-order Horn clauses. *Journal of the Association for Computing Machinery*, 37(4):777–814, October 1990.

[163] D. Nauck, F. Klawonn, R. Kruse, and F.Klawonn. *Foundations of Neuro-Fuzzy Systems*. John Wiley and Sons Inc., NY, 1997.

[164] J. Von Neumann. The general and logical theory of automata. *Cerebral Mechanisms and Behavior*, pages 1–41, 1951.

[165] J. Von Neumann. Probabilistic logics and the synthesis of reliable organisms from unreliable components. *Automata Studies*, pages 43–98, 1956.

[166] Raymond Ng. Reasoning with uncertainty in deductive databases and logic programs. *International Journal of Uncertainty, Fuzziness and Knowledge-based Systems*, 2(3):261–316, 1997.

[167] Raymond Ng and V. S. Subrahmanian. Probabilistic logic programming. *Information and computation*, 101(2):150–201, 1992.

[168] Randall C. O'Reilley and Yuko Munahata. *Computational Explorations in Cognitive Neuroscience: Understanding the Mind by Simulating the Brain*. MIT Press, 2000.

[169] Guiseppe Peano. The principles of arithmetic, presented by a new method. In Jean van Heijenoort, editor, *A Source Book in Mathematical Logic, 1879 – 1931*, pages 83–97. Harvard Univ. Press, 1967.

[170] C.S. Peirce. *Collected Papers of Charles Saunders Peirce, volumes 1–8, C. Hartshorne, P. Weiss, A. Burks (eds.)*. Harvard University Press, Cambridge MA, 1931–1935, 1958.

[171] R. Penrose. *Shadows of the Mind: A Search for the Missing Science of Consciousness*. Oxford University Press, 1994.

[172] J.B. Pollack. *On Connectionist Models of Natural Language Processing*. PhD thesis, Computer science Department, University of Illinois, Urbana, 1987.

[173] A. John Power and Leon Sterling. A notion of a map between logic programs. In *Logic Programming, Proceedings of the Seventh International Conference*, pages 390–404. MIT Press, 1990.

[174] D. Prawitz. An improved proof procedure. *Theoria*, 26:102–139, 1960.

[175] D. Prawitz. *Natural Deduction*. Almqvist & Wiksell, Uppsala, 1965.

[176] J. S. Provan and M. O. Ball. The complexity of counting cuts and of computing the probability that a graph is connected. *SIAM Journal of Computing*, 12(4), 1983.

[177] N. Resher. *Many-valued logic*. Mac Graw Hill, 1996.

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

[179] F. Rosenblatt. The perceptron: A probabilistic model for information storage and organisation in the brain. *Psychol. Rev.*, 65:386—408, 1958.

[180] F. Rosenblatt. *Principles of Neurodynamics*. Spartan Books, Washington DC, 1961.

[181] D.E. Rumelhart and J.L. McClelland. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, I & II*. MIT Press, Cambridge MA, 1986.

[182] Gernot Salzer. MUltlog 1.0: Towards an expert system for many-valued logics. In *Proc. 13th Int. Conf. on Automated Deduction (CADE'96)*, volume 1104 of *LNCS (LNAI)*, pages 226 – 230. Springer, 1996.

[183] Gernot Sazler. Optimal axiomatizations of finitely-valued logics. *Information and Computation*, 162(1 – 2):185 – 205, 2000.

[184] Anthony K. Seda. On the integration of connectionist and logic-based systems. In T. Hurley, M. Mac an Airchinnigh, M. Schellekens, A. K. Seda, and G. Strong, editors, *Proceedings of MFCSIT2004, Trinity College Dublin, July, 2004*, volume 161 of *Electronic Notes in Theoretical Computer Science*, pages 109–130. Elsevier, 2005.

[185] Anthony Karel Seda and Maire Lane. Some aspects of the integration of connectionist and logic-based systems. In L. Li and K. Yen, editors, *Proceedings of the Third International Conference on Information Information'04*, pages 297–300, Tokyo, November 2004.

[186] Maria I. Sessa. Approximate reasoning by similarity-based SLD-resolution. *Theoretical computer science*, 275:389–426, 2002.

[187] L. Shastri and V. Ajjanagadde. An optimally efficient limited inference system. In *Proceedings of the AAAI National Conference on Artificial Intelligence*, pages 563–570, 1990.

[188] L. Shastri and V. Ajjanagadde. From associations to systematic reasoning: A connectionist representation of rules, variables and dynamic bindings using temporal synchrony. *Behavioural and Brain Sciences*, 16(3):417–494, 1993.

[189] J. Shoenfield. *Mathematical Logic*. Addison-Wesley, Reading, Mass., 1967.

[190] H. Siegelmann and M. Margenstern. Nine switch-affine neurons suffice for Turing universality. *Neural Networks*, 12:593–600, 1999.

[191] H. Siegelmann and E.D. Sontag. Turing computability with neural nets. *Applied Mathematics Letters*, 4(6):77–80, 1991.

[192] H. Siegelmann and E.D. Sontag. On the computational power of neural nets. *J. of Computers and System Science*, 50(1):132–150, 1995.

[193] P. Smolensky. On the proper treatment of connectionism. *Behavioral and Brain Sciences*, 11:1–74, 1988.

[194] Robert Soare. *Recursively Enumerable Sets and Degrees: A Study of Computable Functions and Computably Generated Sets*. Springer-Verlag, Heidelberg, 1987.

[195] D.A. Sprecher. On the structure of continuous functions of several variables. *Trans. Am. Math. Soc.*, 115:340–355, 1965.

[196] Daniel Stamate. Quantitative datalog semantics for databases with uncertain information. In *Proceedings of the 4th Workshop on Quantitative Aspects of Programming Languages*, ENTCS. Elsevier, 2006.

[197] Leon Sterling and Ehud Shapiro. *The art of Prolog*. MIT Press, 1986.

[198] Umberto Straccia. Query answering in normal logic programs under uncertainty. In *8th European Conference on Symbolic and Quantitative Approaches to Reasoning with Uncertainty (ECSQARU-05)*, Lecture Notes in Computer Science, pages 687–700, Barcelona, Spain, 2005. Springer Verlag.

[199] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific J. Math.*, 5:285–309, 1955.

[200] G.G. Towell and J.W. Slavik. Knowledge-based artificial neural networks. *Artificial Intelligence*, 70(1–2):119–165, 1994.

[201] Alan Turing. *Collected Works: Mathematical Logic*. North-Holland, 2001.

[202] L. G. Valiant. The complexity of enumeration and reliability problems. *SIAM Journal of Computing*, 8(3), 1979.

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

[204] M. van Emden and R. Kowalski. The semantics of predicate logic as a programming language. *Journal of the Assoc. for Comp. Mach.*, 23:733–742, 1976.

[205] Peter Vojtás and Leonard Paulík. Soundness and completeness of non-classical extended sld-resolution. In Roy Dickhoff, H. Herre, and P. Shroeder-Heister, editors, *Extensions of Logic Programming, 5th International Workshop ELP'96, Leipzig, Germany, March 28-30, 1996*, volume 1050 of *Lecture notes in Computer Science*, pages 289–301. Springer, 1996.

[206] David H.D. Warren. An abstract Prolog instruction set. Technical Report 309, SRI International, Menlo Park, CA, 1983.

[207] Alfred Whitehead and Bertran Russel. *Principia Mathematica*. Cambridge Univ. Press, 1910, 1912, 1913.

[208] S. Willard. *General Topology*. Addison-Wesley, 1970.

[209] D. Woods and T. Neary. The complexity of small universal Turing machines. In *Proceedings of the International Conference Computability in Europe (CiE'07)*, LNCS, 2007. To appear.

[210] L. A. Zadeh. Fuzzy sets. *Information and Control*, 8:338–353, 1965.

[211] L.A. Zadeh. Interpolative reasoning in fuzzy logic and neural network theory. *Fuzzy Systems*, pages 1–20, 1992.

[212] A. Zell, C. Mamier, M. Vogt, N. Mache, R. Hübner, S. Döring, K.-W. Hermann, T. Soyez, M. Schmatzl, T. Sommer, A. Hatzigeorgionis, D. Dossett, T. Schreiner, B. Kett, G. Clemente, and J. Wieland. SNNS (Stuttgart Neural Network Simulator): User manual. Technical Report 6, University of Stuttgart, Institute for Parallel and Distributed High Performance Systems (IPVR), 1995. www.-ra.informatik.uni-tuebingen.de/SNNS.