INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UM films the text directly from the original or copy submitted. Thus, som thesis and dissertation copies are in typewriter face, while others ma be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of th copy submitted. Broken or indistinct print, colored or poor qualit illustrations and photographs, print bleedthrough, substandard margin and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complet manuscript and there are missing pages, these will be noted. Also, unauthorized copyright material had to be removed, a note will indicat the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced b sectioning the original, beginning at the upper left-hand corner an continuing from left to right in equal sections with small overlaps. Eac original is also photographed in one exposure and is included i reduced form at the back of the book.

Photographs included in the original manuscript have been reproduce xerographically in this copy. Higher quality 6" x 9" black and whit photographic prints are available for any photographs or illustration appearing in this copy for an additional charge. Contact UMI directl to order.

UMI

University Microfilms International
A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
313/761-4700 800/521-0600

Order Number 9503581

Generalized knowledge-based semantics for multi-valued logic programs

Mobasher, Bamshad, Ph.D.

Iowa State University, 1994

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

Generalized knowledge-based semantics for multi-valued logic programs

by

Bamshad Mobasher

A dissertation Submitted to the
Graduate Faculty in Partial Fulfillment of the
Requirements for the Degree of
DOCTOR OF PHILOSOPHY
Department: Computer Science
Major: Computer Science

Approved:

Signature was redacted for privacy.

Signature was redacted for privacy.

In Charge of Major Work

Signature was redacted for privacy.

For the Major Department

Signature was redacted for privacy.

For the Graduate College

Members of the Committee:

Signature was redacted for privacy.

Signature was redacted for privacy.

Signature was redacted for privacy.

Iowa State University
Ames, Iowa
1994

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGEMENTS

I would like to express my appreciation to my advisors, Giora Slutzki and Don Pigozzi, for their help in bringing this dissertation to fruition. Throughout the years, both of these individuals have been much more than advisors to me; they have been my mentors. I will be eternally grateful for their irreplacable technical assistance and their unending devotion and dedication.

I am also grateful to the other members of my committee, Gary Leavens, Les Miller, and Vasant Honavar, for their helpful suggestions and assistance in this work and throughout my career as a graduate student. I would like to give special thanks to Jacek Leszczylowski for his many helpful suggestions and important contributions to this work.

Last, but not least, I would like to thank my parents, my wife Angela, and Dorinne Dekrey, without whose love and support none of this would have been possible.

CHAPTER 1. INTRODUCTION

1.1 Logic Programming and Revisable reasoning

Many areas of Artificial Intelligence rely heavily on representing the knowledge an entity has about its environment in an explicit or symbolic form. Knowledge represented in this fashion is often referred to as *declarative knowledge* because it is contained in declarations about the world. There are many reasons to prefer declarative representation of knowledge when designing intelligent systems. One advantage is that it is relatively easy to make changes to the knowledge present in the system: this can be done by modifying a small subset of the statements that make up the system's knowledge base. Furthermore, declarative knowledge can usually be extended, beyond the explicit set of declarations, by the virtue of some *reasoning* process that can derive additional knowledge. The issue of formalizing knowledge has thus been one of the central building blocks and research areas of Artificial Intelligence. The purpose of formalizing declarative knowledge is to provide intelligent machines with a mathematically precise definition of the knowledge that they possess, in a manner which is independent of procedural considerations and easy to manipulate.

For many years, particularly in the 70's, it was believed by many AI practitioners that first-order logic can serve as an adequate framework and a universal language for knowledge representation. The power of first-order predicate calculus as a language

for expressing declarative knowledge in AI systems cannot be disputed. However, it later became clear to most adherents of this view that first-order logic suffers from certain inadequacies, especially when trying to represent or reason with inexact, incomplete, or contradictory information. This kind of reasoning process is, of course, one that faces intelligent systems in many problem domains.

Presumably, a typical AI system using first-order logic would work in the following manner. The knowledge that the system has about the problem domain is given as a finite set of axioms (formulas of first-order logic), say Γ. To answer a query or take some action, the system would then try to determine whether a formula F is a logical consequence of Γ. This task is accomplished by using the deduction rules of first-order logic to derive F from Γ (in most theorem proving systems, for instance, this is done by using the resolution rule of inference on the clausal form of $\Gamma \cup \neg F$ to derive a contradiction). It was research in the area of theorem proving that eventually led to logic programming.

The main tenet of logic programming, the idea that first-order logic could be used as a programming language, was revolutionary because it not only capitalized on the ability of logic as a way to specify knowledge, but also it showed that first-order logic has a *procedural interpretation*. The idea here was that given a program clause $A \leftarrow B_1, \cdots, B_n$, one could interpret this clause as giving a definition for a *procedure* A. Now, if a goal $\leftarrow B'_1, \cdots, B'_k$ is given to the system, each B'_i can be interpreted as a *procedure call*. The computation, then, would be comprised of a sequence of steps each involving an attempt to unify one of the B'_i with A. The unification process records bindings obtained for the variables of A in the form of *substitutions*. Thus unification will become a uniform mechanism for parameter passing, data selection, and data

construction. This procedural interpretation emanates from the proof theory of first-order logic, and computation of the resulting substitution essentially amounts to a constructive proof of an existential formula.

Logic programming is based on the idea of declarative programming stemming from Kowalski's principle of separation of logic and control [28]. Ideally, the programmer should be only concerned about the declarative meaning of the program (i.e., *what* needs to be done), while the procedural aspects of the execution (i.e., *how* it needs to be done) are handled automatically. Unfortunately, this idealized view of logic programming has not yet been realized. One reason is those same limitations of first-order logic as a knowledge representation language, described earlier. A particular instance of this problem is the lack of clarity as to what should be the proper declarative semantics for negation within logic programs. Let us examine some of the limitations mentioned above more closely.

One important limitation of the aforementioned model of AI systems using first-order logic (even in the context of logic programming) is that often the rules specified within the system are subject to an indefinite number of exceptions and qualifications. In a sense our set of axioms and rules are but an approximate description of the real world. Ideally, the representation language should be robust enough to deal with conclusions that will need to be further revised and specified. Systems based on classical logic are inherently *monotonic* in the sense that addition of new information to a base set of beliefs will never result in inferences that contradict what was originally known . The type of reasoning in which an intelligent agent engages, however, often involves revising or retracting old conclusion as a result of expanding the set of axioms by newly obtained information. This type of reasoning is called

nonmonotonic.

Another limitation of first-order logic stems from the fact that statements are evaluated to be either true or false. Intelligent agents, however must often deal with information which is *uncertain*, or incomplete. This suggests that certain non-classical logics which allow statements to be evaluated by truth values other than true or false may be more suitable for knowledge representation in some AI systems.

There have been several approaches for formalizing nonmonotonic and revisable reasoning. These include *default logic* [44], *nonmonotonic modal logics* [33], and *Autoepistemic logics* [34, 35]. For a review of these approaches see [54]. These and other logical approaches to nonmonotonic reasoning and reasoning with uncertain beliefs have been studied in [22].

The kind of "brittleness" resulting from the limitations of the underlying first-order logic within the logic programming systems, has caused many AI practitioners to shy away from using logic programming languages as the knowledge representation language in AI systems. It is therefore desirable to construct logic programming systems that can overcome the difficulties mentioned above. The work presented here is an attempt to provide a general framework for such logic programming languages. The above brief discussion suggests that such systems must have two common characteristics: they must rely on the expressive power of an underlying multi-valued logic which can deal with contradictory as well as incomplete or uncertain information, and secondly, such systems should be able to interpret statements not based on their truth or falsity, but based on some measure of the knowledge contained within those statements.

1.2 Semantics of Negation and Knowledge-based Logic Programming

In the context of logic programming and deductive databases, much of the research along the aforementioned lines has manifested itself in attempts to deal with the issue of adequately representing negative or conflicting information. The presence of negation within logic programs, however, causes certain semantic problems [49, 50, 29], and the full inclusion of classical negation in logic programs and queries is generally thought to be infeasible for computational reasons.

Negation as Failure is the most common treatment of negation in logic programming. It is essentially a rule of inference stating that if A is a ground atom, then the goal $\neg A$ succeeds if A fails, and the goal $\neg A$ fails if A succeeds. However, it is well known that Negation as Failure is not sound with respect to classical semantics for programs [49, 50]. There have been many attempts to give a reasonable declarative semantics with respect to which Negation as Failure is sound, including Reiter's Closed World Assumption [43] and Clark's program completion [8]. Unfortunately, while these and other approaches have resulted in various declarative semantics with respect to which Negation as Failure is sound, the corresponding completeness results hold only for restricted classes of logic programs.

These semantic problems are also present when the declarative semantics for logic programs involving negation is specified using fixpoints. Fixpoint semantics for logic programs were originally developed by van Emden and Kowalski [56] in the context of logic programs without negation. The idea is to associate, with each program P, a natural closure operator T_P on interpretations and to identify models of P with fixpoints of T_P. The interpretation given by the least fixpoint of T_P is generally taken to be the intended model for the program. When negations are present, however, the

T_P operator is generally not monotonic and T_P may have no least fixpoint.

Much of the literature about negation in logic programming examines the ramifications of choosing non-classical semantics based on multi-valued logics [14, 15, 29]. From a related but different point of view, several approaches have proposed dealing with negation by ordering statements and formulas not according to the degree of truth or falsity, but according to the degree of knowledge "present in the system" about these statements and formulas. Ginsberg [23, 24] introduced a family of multi-valued logics based on certain algebraic structures called *bilattices*, which combined the two aforementioned approaches. Ginsberg's work focuses on those logics that have a knowledge dimension as well as a truth dimension and thus can be used to model the connection between truth and knowledge in a particular logic program or deductive database. The first logic of this kind originated with Belnap [3]. It is based on the idea that information in a database can have both a positive and a negative content with regard to the truth of a particular event.

The two situations in which only positive or only negative information is available give rise to two truth values that can be identified with classical true and false, respectively. But there are two other situations: when the information has both a positive and a negative content, and where there is no information of either kind. These lead to a third and fourth "truth value" that are denoted respectively by $\top$ and $\bot$. Part of the motivation here is that, in a distributed database, information about a given event is collected from various sources at various times and some of it might be contradictory. So the truth value of the event can be viewed as representing our state of knowledge about the classical truth or falsity of the event rather than its actual truth or falsity, and as we have seen there are four possibilities for this state

of knowledge. This 4-element Belnap logic is the simplest example of a bilattice.

Fitting [16, 17, 18, 19] further studied properties of bilattices. He expanded the theory of bilattices into a full fledged mathematical theory of the truth-knowledge interaction and made this the basis of a well behaved fixpoint semantics for a knowledge-based logic programming system. For logic programs based on the class of distributive bilattices, he developed a fixpoint semantics and a procedural semantics based on Smullyan style semantic tableaux.

The significant feature of knowledge-based logic programming is that logical negation is monotonic with respect to knowledge: Having more information about event E_1 than event E_2 means we have more positive and more negative information. Hence we also have more information about $\neg E_1$ than $\neg E_2$ since logical negation simply switches the polarity of the information. In contrast, logical negation is anti-monotonic with respect to classical truth. Bilattices provide a setting in which one can successfully deal with negation in programs, at least when programs are interpreted according to their knowledge content. Many of the problems that arise in classical truth-based logic programming that are due to the anti-monotonic nature of negation, such as the nonexistence of least fixpoints, can be avoided in a knowledge-based system.

1.3 An Overview

The results presented in this thesis also use bilattices as the underlying framework for the logic programming language. We use a fixpoint semantics similar to the one proposed by Fitting, but we develop a new procedural semantics based, partly, on resolution. We present a generalized framework for knowledge-based logic programs.

Depending on the choice of the underlying bilattice, the resulting knowledge-based logic programming language can be used to model many useful logics such as probabilistic logics, intuitionistic logics, and modal logics based on the possible-worlds semantics. The procedural semantics presented in this work, generalizes the ideas behind SLDNF-resolution to provide a natural and efficient computational model for logic programming. The results presented in the sequel are organized as follows.

In Chapter 2, we provide some necessary background information on lattices. Lattices provide the underlying structure of bilattices. They are also used as the basis of fixpoint semantics which we use to characterize the declarative semantics of logic programs. An important part of this chapter is the discussion of the join-irreducible elements of a lattice and their role as a representative subset of elements in distributive lattices which satisfy certain finiteness conditions. These ideas are later extended to bilattices.

In Chapter 3, we present some general background material on logic programming. In particular, we discuss various elements of logic programming systems based on classical first-order logic, including the syntax of the underlying language, declarative semantics, and procedural semantics of logic programs. We also introduce the basic concepts of unification theory. Finally, we give a brief discussion of the role of negation in logic programming and some of the pitfalls and problems that arise in obtaining adequate semantics for negation.

The new results presented in this work are discussed in Chapters 4 through 7. In Chapter 4 we present a general theory of bilattices. We provide the basic definitions and some motivating examples of bilattices. We also present the basic construction and representation theorems for bilattices originally due to Ginsberg. For reasons

that will become clear later, the definition of bilattice given here is somewhat different from those found in the works by Ginsberg and Fitting mentioned earlier. There are two different mechanisms by which the relationship between the truth and the knowledge dimensions can be specified. One is the requirement that the lattice theoretic operators for each ordering be monotonic with respect to the other ordering. This called the *interlacing condition.* We call structures that have this property *pre-bilattices.*

Another way of capturing the connection between the two orderings is by means of a negation operator which is monotonic with respect to the knowledge ordering but not with respect to the truth ordering. Ginsberg's definition of bilattices consists of structures (with two complete lattices) which have such an idempotent negation operator. We, however, insist that a bilattice also satisfy the interlacing condition. In other words, every bilattice is a pre-bilattice. In this way we can discuss more general results in the context of the weaker notion of a pre-bilattice, whereas for the results specific to bilattices, we can focus on the properties of the negation operator. It must be noted, however, that for the class of bilattices in which we are most interested, namely, distributive bilattices, the alternative definitions are equivalent as the distributivity laws imply the interlacing condition.

For our purposes, the most important part of this chapter is the discussion of elements of a bilattice that are join-irreducible in the knowledge ordering. We extend many of the properties of the join irreducible elements to bilattices. We study these properties in detail and present several new results which are original contributions to the theory of bilattices. In particular, we prove several results which show that the knowledge join-irreducible elements provide a representing set for distributive

bilattice that have the descending chain property in the knowledge ordering. In the logic programming context, these are precisely the type of bilattices in which we are the most interested.

In Chapter 5 we extend the notion of unifiers to substitutions themselves and use this concept of *substitution unification* to provide the necessary machinery for parallel evaluation of queries in our procedural semantics. Our procedural semantics uses an AND- and OR-parallel interpretation model (see [52, 9, 10]), in which the AND-parallel component is independent. In an independent AND-parallel model, even when subgoals share variables, they are solved independently. After termination, however, answer substitutions obtained independently for shared variables are tested for consistency. Substitution unification is used to ensure the consistency of answer substitutions for shared variables. In Chapter 5 we present an in-depth treatment of substitution unifiers and study many of their interesting properties. The results about substitution unifiers are significant in two respects. First, they represent an interesting contribution to the theory of unification, and secondly, they may be found useful in the development of parallel interpretation models for logic programs.

Similar ideas have been studied before in the literature, particularly in the context of parallel logic programming [26, 38]. Most of these approaches reduce the problem of unification to that of finding a solution to a system of equations or to other methods which involve dealing with substitutions in the context of their application to sets of expressions. Our development provides an expression and equation free treatment of substitutions and their unifiers. This allows us to develop a full fledged algebraic theory of substitution unification.

In Chapter 6, we carefully study an important special case of our generalized

knowledge-based logic programming framework, namely, one based on a four-valued bilattice. This special case serves as the basis of many of the concepts which we later generalize to arbitrary bilattices. It is significant in its own right since the resulting logic programming system can be used as a representation language in AI systems that have to deal with contradictory or incomplete information. In particular, we will show how this particular system can resolve many of the problems associated with negation.

For this purpose we use a four-valued logic of Belnap in which the space of truth values includes not only `true` and `false`, but also two other truth values which represent degrees of knowledge about the truth or falsity of a particular statement. in particular, no information and conflicting information. The space of truth values, and by extension, the space of all interpretations, is now partially ordered in two dimensions using two separate orderings. One is called the *truth dimension* and the other is called the *knowledge dimension.* In the truth direction we have all of the machinery of classical logics. In the knowledge dimension, interpreting a program according to its knowledge content gives a monotonic operator associated with that program. When programs are interpreted according to their knowledge content, a statement can potentially be evaluated as both true and false, suggesting the existence of conflicting information. In this sense these programs have the *paraconsistency* property introduced in [5]. For instance, interpreting the program clause $A \leftarrow$`true` according to its knowledge content, does not mean that A is true, but rather that there is evidence suggesting that A is true.

We fully utilize the self-duality of knowledge operators under negation allowing us to evaluate negative queries with variables without encountering the usual seman-

tic problems associated with negation in standard truth-based logic programming. It is worth emphasizing that interpretation of programs under the knowledge ordering gives rise to a monotonic logic which has potential for efficient implementation. The procedural semantics presented here treats the notions of success and failure symmetrically, and thus it is can be used to deduce both negative and positive information in a uniform manner. Furthermore, this procedural semantics lends itself to parallel and distributed evaluation of queries, and thus it can serve as the basis for implementation of parallel knowledge-based logic programming languages.

In the last part of this chapter we extend the bilattice-based fixpoint and procedural semantics to incorporate a version of Closed World Assumption (CWA). This allows inference of negative information when no information is present. We give soundness and completeness results, with and without the presence of CWA. Our soundness and completeness results are general and are not restricted to ground atomic goals.

In Chapter 7 we generalize the four-valued semantics to arbitrary distributive bilattices. As mentioned above, a novel feature of our operational semantics based on the 4-element Belnap bilattice is the introduction of completely symmetric notions of *proof* and *refutation*. Roughly speaking, the existence of a proof (respectively refutation) for a given goal, corresponds to having positive (respectively negative) information about it.

In this chapter the operational semantics is generalized to an arbitrary distributive bilattice. We introduce the notion of a *b-proof* for each element of the bilattice except $\top$ and $\bot$. (In the 4-element case **true**-proofs coincide with proofs and a **false**-proofs with refutations.) We prove a soundness and completeness theorem for this

procedural semantics, again with respect to the declarative fixpoint semantics.

Although the resulting logic programming system is quite satisfactory in some respects, for example the symmetry between truth and refutation in the 4-element case is carried over and the mathematical theory is quite smooth, it has some serious defects. For a given truth value b, the search for a b-proof of a complex goal G may entail searches for c-proofs of the subformulas of G for a large number of truth values c that are only remotely related to b; moreover, this *complexity* ramifies as we pass down the parse tree of G. It turns out that for finite distributive bilattices (more generally, bilattices with the *descending chain condition*) we can essentially restrict our attention to b-proofs where b ranges over a relatively small subset of special truth-values. Moreover, in the search for a b-proof for G, we need only look for b-proofs of the subformulas of G. These special truth values turn out to be the so-called *join irreducible* elements of knowledge part of the bilattice. We present a *join irreducible* operational semantics as an alternative to the standard one, and prove the connection between the two in the main result of the chapter. This allows us to obtain a completeness theorem for the join-irreducible operational semantics (with respect to Fitting's fixpoint semantics) as a corollary of our first completeness theorem.

CHAPTER 2. LATTICE BACKGROUND

Before introducing the notion of a bilattice, which is central to the work presented in this thesis, we need to provide some background material on lattices. Lattices not only provide the underlying structure of bilattices, but they also are used as the foundation of logic programming semantics based on fixpoints which we shall use as the declarative semantics for logic programming system studied in this work.

Lattices are a special class of ordered sets which can be represented by certain algebraic axioms based on the existence of lower and upper bounds of subsets of the given ordered set. In the following, we will make these notions precise. We will also introduce and study a special subset of lattice elements, namely the join-irreducible ones, which will play an important role in the operational semantics of our knowledge-based logic programs.

2.1 Some Basic Concepts in Lattice Theory

Definition 2.1 Let P be a set. A *partial order* on P is a binary relation $\leq$ on P such that, for all $x, y, z \in P$,

1. $x \leq x$ (reflexivity);

2. $x \leq y$ and $y \leq x$ imply $x = y$ (anti-symmetry);

3. $x \leq y$ and $y \leq z$ imply $x \leq z$ (transitivity).

A set P on which a partial order $\leq$ is defined is called a *partially ordered set* (or simply an ordered set), denoted by $(P, \leq)$. The relation $\leq$ is called a *full* (or *total* or *linear*) *order* on P, if $\leq$ is a partial order on P and for any $x, y \in P$, either $x \leq y$ or $y \leq x$. In this case, we say that P is a *fully* (or *totally* or *linearly*) *ordered* set.

Definition 2.2 Let P be a partially ordered set and let $S \subseteq P$. An element $x \in P$ is an *upper bound* of S if $s \leq x$ for all $s \in S$. A *lower bound* is defined dually. The element $x \in P$ is the *least upper bound* of S if x is an upper bound of S, and $x \leq y$ for all upper bounds y of S. The notion of *greatest lower bound* is defined dually. The greatest element of P, if it exists, is called the *top element* of P and denoted by $\top$. Similarly, the least element of P, if it exists, is called the *bottom element* of P, and denoted by $\bot$.

Note that if a partially ordered set P has a top element, then $\top$ is the unique least upper bound for P. Similarly, the bottom element, $\bot$, if it exists, is the unique greatest lower bound for P.

We usually denote the least upper bound of x and y by $x \vee y$ (read "*x join y*") and the greatest lower bound of x and y by $x \wedge y$ (read "*x meet y*"). Accordingly, for a set S, we write $\bigvee S$ and $\bigwedge S$ to denote the least upper bound (join) and the greatest lower bound (meet) of S, respectively.

Definition 2.3 Let P be a non-empty partially ordered set.

1. If $x \vee y$ and $x \wedge y$ exist for all $x, y \in P$, then P is called a *lattice*.

2. If $\bigvee S$ and $\bigwedge S$ exist for all $S \subseteq P$, then P is called a *complete lattice*.

If P is a lattice, then $\vee$ and $\wedge$ are binary operations on P and we have an algebraic structure $\langle P, \wedge, \vee \rangle$ (we leave out the relation $\leq$ since its association with P is generally clear from context).

It is clear from the above definitions that any complete lattice is *bounded*, that is, it has top and bottom elements. Furthermore, if P is a lattice, we can easily verify the following relationships between its ordering relation and its binary operators, $\wedge$ and $\vee$.

Theorem 2.4 *Let L be a lattice and let $x, y \in L$. Then the following are equivalent:*

1. $x \leq y$;
2. $x \vee y = y$;
3. $x \wedge y = x$.

Hence, to show that P is a lattice it suffices to show that $x \vee y$ and $x \wedge y$ exist in P for all non-comparable pairs $x, y \in P$.

It is well known for an ordered set P that for every non-empty subset S of P which has an upper bound in P, if $\bigwedge S$ exists in P , then $\bigvee S$ exists in P; in fact,

$$\bigvee S = \bigwedge \{x \mid x \geq y \text{ for every } y \in S\}.$$

In other words, $\bigvee S$ is the meet of the set of all upper bounds of S. We thus have the following result.

Theorem 2.5 *Let P be a non-empty partially ordered set. Then the following are equivalent:*

1. *P is a complete lattice;*

2. $\bigwedge S$ *exists in* P *for every subset* $S \subseteq P$;

3. P *has a top element,* $\top$, *and* $\bigwedge S$ *exists in* P *for every non-empty* $S \subseteq P$.

By a similar argument we can also establish the dual the above theorem:

Theorem 2.6 *Let* P *be a non-empty partially ordered set. Then the following are equivalent:*

1. P *is a complete lattice;*

2. $\bigvee S$ *exists in* P *for every subset* S *of* P;

3. P *has a bottom element,* $\bot$, *and* $\bigvee S$ *exists in* P *for every non-empty subset* S *of* P.

The following properties of lattices are often used when studying lattices as algebraic structures. They provide an alternative definition for lattices. We often use these properties without explicit mention.

Theorem 2.7 *Let* L *be a lattice. Then, for all* $a, b, c \in L$, $\wedge$ *and* $\vee$ *satisfy the following conditions.*

1. *(Associativity)* $(a \vee b) \vee c = a \vee (b \vee c)$ *and* $(a \wedge b) \wedge c = a \wedge (b \wedge c)$

2. *(Commutativity)* $a \vee b = b \vee a$ *and* $a \wedge b = b \wedge a$

3. *(Idempotency)* $a \vee a = a$ *and* $a \wedge a = a$

4. *(Absorption)* $a \vee (a \wedge b) = a$ *and* $a \wedge (a \vee b) = a$.

There are a number of ways to construct new lattices (or more generally, ordered sets) from existing ones. We are particularly interested in two of these constructions which will be useful in subsequent discussions.

Let L and K be lattices. Consider the ordered set $L \times K$ with $\vee$ and $\wedge$ defined coordinatewise, as follows:

$$\begin{aligned} \langle l_1, k_1 \rangle \vee \langle l_2, k_2 \rangle &= \langle l_1 \vee l_2, k_1 \vee k_2 \rangle \\ \langle l_1, k_1 \rangle \wedge \langle l_2, k_2 \rangle &= \langle l_1 \wedge l_2, k_1 \wedge k_2 \rangle. \end{aligned}$$

It is easy to check that $L \times K$ is also a lattice with the partial order $\leq$ defined coordinatewise. Furthermore, we have:

$$\begin{aligned} \langle l_1, k_1 \rangle \vee \langle l_2, k_2 \rangle = \langle l_2, k_2 \rangle &\Leftrightarrow l_1 \vee l_2 = l_2 \text{ and } k_1 \vee k_2 = k_2 \\ &\Leftrightarrow l_1 \leq l_2 \text{ and } k_1 \leq k_2 \\ &\Leftrightarrow \langle l_1, k_1 \rangle \leq \langle l_2, k_2 \rangle. \end{aligned}$$

Hence, the lattice obtained by taking the product of lattices L and K is the same as the one obtained by defining $\vee$ and $\wedge$ as above. If L and K are complete lattices, then $L \times K$ is a complete lattice (with joins and meets formed coordinatewise).

Another important class of lattices is obtained by considering the set of all mappings from a set to a lattice.

Definition 2.8 Let P and Q be ordered sets.

1. A map $\phi : P \to Q$ is said to be *order-preserving* (or *monotone*) if $x \leq y$ in P implies that $\phi(x) \leq \phi(y)$ in Q. ϕ is an *order-embedding* if $x \leq y$ in P if and only if $\phi(x) \leq \phi(y)$ in Q. If, in addition, ϕ is onto, then it is called an *order-isomorphism*.

2. The set of all maps from P to Q is denoted by Q^P.

Let P be any set and Q a (complete) lattice. Then, under the usual pointwise order, the set Q^P of maps from P to Q is also a (complete) lattice with join and meets formed in a pointwise fashion. Therefore, the join ϕ, of $\{\phi_i \in Q^P\}_{i \in I}$ is given by

$$\text{for all } x \in P,\ \phi(x) = \bigvee_{i \in I} \phi_i(x),$$

and similarly for meet. When P is an ordered set and all the maps ϕ_i are order preserving, then $\bigvee\{\phi_i \mid i \in I\}$ and $\bigwedge\{\phi_i \mid i \in I\}$ are also order-preserving. Hence, the resulting ordered set of mappings is a (complete) lattice.

Definition 2.9 Let L and K be lattices. A map $\phi : L \to K$ is said to be a *(lattice-) homomorphism* if ϕ is *join-preserving* and *meet-preserving*, that is, for all $a, b \in L$,

$$\phi(a \vee b) = \phi(a) \vee \phi(b) \quad \text{and} \quad \phi(a \wedge b) = \phi(a) \wedge \phi(b).$$

A bijective homomorphism is a *(lattice-) isomorphism*. If $\phi : L \to K$ is a one-to-one homomorphism, then the sublattice $\phi(L)$ of K is isomorphic to L and we refer to ϕ as an *embedding (of L into K)*.

In general, an order-preserving (monotonic) map may not be a homomorphism. But, a stronger relationship holds between order-isomorphisms and lattice isomorphisms, as the following result indicates [11].

Lemma 2.10 *Let L and K be lattices and $\phi : L \to K$ a map.*

1. *The following are equivalent:*

   (a) ϕ is order-preserving;

(b) for all $a, b \in L$, $\phi(a \vee b) \geq \phi(a) \vee \phi(b)$;

(c) for all $a, b \in L$, $\phi(a \wedge b) \leq \phi(a) \wedge \phi(b)$.

In particular, if ϕ *is a homomorphism, then* ϕ *is order-preserving.*

2. *The following are equivalent:*

   (a) ϕ *is an order-isomorphism;*

   (b) ϕ *is bijective and an order-embedding;*

   (c) ϕ *is a lattice-isomorphism.*

Complete lattices play an essential role in our discussion since they constitute the building blocks for bilattices, which we introduce in the next section. It is easy to verify that every finite lattice is complete. There are, however, other weaker finiteness conditions which guarantee that a lattice is complete. We will now state some of these conditions.

Definition 2.11 Let P be an ordered set.

1. A subset S of P is a *chain* in P, if for all $x, y \in P$, either $x \leq y$ or $y \leq x$. We usually represent a chain as a (possibly infinite) sequence $\langle c_0, c_1, \cdots \rangle$, where $c_i \leq c_{i+1}$.

2. If $C = \langle c_0, c_1, \cdots, c_n \rangle$ is a finite chain in P, then we say that the *length* of C is n.

3. P has *length* n if the length of the longest chain in P is n.

4. P is of *finite length* if it has length n for some $n < \omega$.

5. P has *no infinite chains* if every chain in P is finite.

6. P has the *ascending chain property* (ACP), if given any infinite sequence $x_1 \leq x_2 \leq \cdots \leq x_n \leq \cdots$ of elements of P, there exists a $k < \omega$ such that $x_k = x_{k+1} = \cdots$. The dual of the ACP is the *descending chain property* (DCP) and is defined accordingly.

The following theorems provide other characterizations of the ACP and the DCP. We state them without proof (see [11]).

Theorem 2.12 *An ordered set P satisfies the ACP if and only if every non-empty subset A of P has a maximal element ($a \in A$ is a maximal element of A, if $a \leq x \in A$ implies $a = x$; minimal element of A is defined dually). P satisfies the DCP if and only if every non-empty subset A of P has a minimal element.*

Theorem 2.13 *An ordered set P has no infinite chains if and only if it satisfies both the ACP and the DCP.*

Lattices with no infinite chains are complete as the following more general result indicates.

Theorem 2.14 *Let P be a lattice.*

1. *If P satisfies the ACP, then for every non-empty subset A of P there exists a finite subset F of A such that $\bigvee A = \bigvee F$ (which exists in P), and similarly for the DCP and $\bigwedge$.*

2. *If P has a bottom element, $\bot$, and satisfies the ACP, then P is complete, and similarly for $\top$ and the DCP.*

3. If P has no infinite chains, then P is complete.

Definition 2.15 Let Q be an ordered set and let $P \subseteq Q$. Then P is *join-dense* in Q if for every element $s \in Q$ there is a subset A of P such that $s = \bigvee_Q A$. The notion of *meet-density* is defined in a dual manner.

The next theorem shows the relationship between complete lattices and the join-dense subsets of an ordered set. Let $(\downarrow s)_P$ denote the set $\{y \in P \mid y \leq s\}$, for every $s \in Q$.

Theorem 2.16 *Let Q be an ordered set and let $P \subseteq Q$. Then the following are related by (1) $\Leftrightarrow$ (2) $\Rightarrow$ (3) in general and are equivalent if Q is a complete lattice:*

1. P is join-dense in Q;

2. $s = \bigvee_Q(\downarrow s)_P$ for all $s \in Q$;

3. for all $s, t \in Q$ with $t < s$ there exists a $y \in P$ with $y \leq s$ and $y \not\leq t$.

Another very important class of lattices, with useful algebraic properties, is the class of *distributive* lattices. The study of the distributivity conditions on lattices will be of particular importance to us. We shall see later that distributive bilattices, are taken as the basis for the procedural semantics of knowledge-based logic programs, due to the special properties they exhibit.

Definition 2.17 Let L be a lattice. The L is *distributive* if it satisfies the distributive law, that is, for all $a, b, c \in L$,

$$a \wedge (a \vee c) = (a \wedge b) \vee (a \wedge c)$$

or (equivalently)

$$a \vee (a \wedge c) = (a \vee b) \wedge (a \vee c).$$

It is well known that if L is a distributive lattice, then every subset of L and every image of L under a homomorphism is also a distributive lattice. Furthermore, if L and K are distributive lattices, then $L \times K$ is distributive.

2.2 Join-Irreducible Elements

There is a rich subclass of the class of distributive lattices which will play an important role in our discussion of procedural semantics for knowledge-based logic programs. These are distributive lattices which satisfy certain finiteness conditions, as described above. A special subset of the elements, namely the join-irreducible elements, provide a representing or characteristic set for this class of lattices. In the subsequent sections, we will extend these notions to bilattices and further study the properties of the join-irreducible elements.

Definition 2.18 Let $(L, \leq)$ be a lattice with $\wedge$, $\vee$, $\bot$, and $\top$. Then $a \in L$ is *join-irreducible*, if $a \neq \bot$ and $a = b \vee c$ implies that $b = a$ or $c = a$, for every $b, c \in L$. We denote the set of join-irreducible elements of L by $JIR(L)$. A *meet-irreducible* element of a lattice is defined dually.

The next few well known results demonstrate the importance of join-irreducible elements in the context of distributive lattices.

Lemma 2.19 *Let L be a distributive lattice and let $x \in L$, with $x \neq \bot$. Then the following are equivalent:*

1. *x is join-irreducible;*

2. *if $a, b \in L$ and $x \leq a \vee b$ then $x \leq a$ or $x \leq b$;*

Proof: To show that $1 \Rightarrow 2$, let $x \in JIR(L)$ and suppose that $a, b \in L$ are such that $x \leq a \vee b$. By distributivity we have $x = x \wedge (a \vee b) = (x \wedge a) \vee (x \wedge b)$. Since x is join-irreducible, $x = x \wedge a$ or $x = x \wedge b$. Hence, $x \leq a$ or $x \leq b$.

It remains to show that $2 \Rightarrow 1$. Suppose that 2 holds and that $x = a \vee b$. Then obviously, $x \leq a \vee b$, and so $x \leq a$ or $x \leq b$. But, $x = a \vee b$ implying that $a \leq x$ and $b \leq x$. Hence, $x = a$ or $x = b$. ∎

Based on the above lemma we have a more general result.

Theorem 2.20 *Let L be a distributive lattice and let $x \in L$, with $x \neq \perp$. $x \in JIR(L)$ if and only if, for $a_1, \cdots, a_k \in L$, $x \leq a_1 \vee \cdots \vee a_k$ implies $x \leq a_i$ for some i $(1 \leq i \leq k)$.*

The following theorem provides a representation theorem for distributive lattices which have the DCP [4].

Theorem 2.21 *Let L be a distributive lattice satisfying the DCP. Then for every $a \in L$, there exists an irredundant decomposition of a as a finite join of join-irreducible elements in L (that is $a = b_1 \vee \cdots \vee b_n$, where $b_i \in JIR(L)$, and none of the b_i can be removed). Furthermore, if $b_1 \vee \cdots \vee b_n = c_1 \vee \cdots \vee c_m$ are two irredundant decompositions of a as joins of join-irreducibles, then $n = m$ and $b_i = c_i$ $(1 \leq i \leq n = m)$, up to renumbering.*

Proof: Let $a \in L$. If $a \in JIR(L)$, then trivially a is an irredundant decomposition of a as a finite join of join-irreducibles. So, suppose that $a \notin JIR(L)$. Then there exist $a_1, a_2 \in L$, such that $a = a_1 \vee a_2$ and $a_1, a_2 < a$. If $a_1, a_2 \in JIR(L)$, then we are done. Otherwise (if a_1, a_2 or both are not join-irreducible), there exist $c_1, c_2, d_1, d_2 \in L$, such that $a_1 = c_1 \vee c_2$, $a_2 = d_1 \vee d_2$. Now, $c_1, c_2 < a_1 < a$ and $d_1, d_2 < a_2 < a$. Hence, $a = c_1 \vee c_2 \vee d_1 \vee d_2$. Continuing in this manner, we obtain a decomposition of a as a join of join irreducible elements, and since L has the DCP, this join must be finite. It can be easily verified that the smallest such decomposition for a will be an irredundant one.

Now suppose that $b_1 \vee b_2 \vee \cdots \vee b_n = c_1 \vee c_2 \vee \cdots \vee c_m = a$ be irredundant decompositions of a as joins of elements in $JIR(L)$. Assume, without loss of generality, that $n \leq m$. Since L is distributive, we can write:

$$\begin{aligned} b_1 &= b_1 \wedge (c_1 \vee c_2 \vee \cdots \vee c_m) \\ &= (b_1 \wedge c_1) \vee \cdots \vee (b_1 \wedge c_m). \end{aligned}$$

But, since $b_1 \in JIR(L)$, $b_1 = b_1 \wedge c_k$, for some k $(1 \leq k \leq m)$. Hence, $b_1 \leq c_k$. By a similar argument, we can conclude that $c_k \leq b_l$, for some l $(1 \leq l \leq n)$. Hence, $b_1 \leq c_k \leq b_l$. However, since $b_1 \vee \cdots \vee b_n$ is an irredundant decomposition of a, it must be the case that $l = 1$. Therefore, $b_1 = c_k$. By reordering, we can assume that $k = 1$ and so $b_1 = c_1$.

Continuing in this way, we have:

$$b_1 = c_1, b_2 = c_2, \cdots, b_n = c_n.$$

Hence,

$$\begin{aligned} a &= b_1 \vee \cdots \vee b_n \\ &= b_1 \vee \cdots \vee b_n \vee c_{n+1} \vee \cdots \vee c_m. \end{aligned}$$

Now, by the irredundancy of $c_1 \vee \cdots \vee c_m$, we can conclude that $n = m$, which establishes the theorem. ∎

In the case of finite distributive lattices we have a stronger representation result with which we end our discussion of lattices.

Definition 2.22 Let P be an ordered set and $Q \subseteq P$. Q is a *down-set* (or *order ideal*) of P if, whenever $x \in Q$, $y \in P$, and $y \leq x$, we have $y \in Q$. We denote the family of all down-sets of P by $\mathcal{O}(P)$.

Note that $\mathcal{O}(P)$ is itself a lattice partially ordered by $\subseteq$. The following theorem is known as *Birkhoff's Representation Theorem for Finite Distributive Lattices* [4].

Theorem 2.23 *Let L be a finite lattice. Then L is distributive if and only if L is isomorphic to $\mathcal{O}(JIR(L))$.*

Proof: First suppose that L is a finite distributive lattice. We show that the map $\eta : L \to \mathcal{O}(JIR(L))$ defined by

$$\eta(a) = \{x \in JIR(L) \mid x \leq a\} \quad (= JIR(L) \cap \downarrow a)$$

is an isomorphism of L onto $\mathcal{O}(JIR(L))$.

By transitivity of $\leq$, it is immediate that $\eta(a) \in \mathcal{O}(JIR(L))$. By Lemma 2.10, we only need to show that η is an order-isomorphism. It is easy to verify that $a \leq b$

implies $\eta(a) \subseteq \eta(b)$. To show that $\eta(a) \subseteq \eta(b)$ implies $a \leq b$, we can use Lemma 2.21 to obtain

$$a = \bigvee \eta(a) \leq \bigvee \eta(b) = b.$$

Finally, we need to show that η is onto. Let $U = \{a_1, a_2, \cdots, a_n\} \in \mathcal{O}(JIR(L))$, and let $a = a_1 \vee \cdots \vee a_n$. We show that $U = \eta(a)$. Suppose that $x \in U$. So, $x = a_i$ for some $i (1 \leq i \leq n)$. Then x is join-irreducible and $x \leq a$, and hence $x \in \eta(a)$. On the other hand, if $x \in \eta(a)$, then $x \leq a = a_1 \vee \cdots \vee a_n$ and by Lemma 2.19 we have $x \leq a_i$, for some i. Since U is an ideal and $a_i \in U$, we have $x \in U$.

The other direction of this theorem follows from the fact that $\mathcal{O}(JIR(L))$ is always distributive (details omitted here), and hence L must also be distributive. ∎

2.3 Lattices and Fixpoints

In this section we study fixpoints of mappings defined on lattices. Our interest in fixpoints arises from the fact that the declarative semantics of logic programs can be characterized by fixpoints of mappings defined over the lattice of interpretations. This characterization will be discussed in subsequent sections.

Definition 2.24 Let L be a complete lattice and $\phi : L \rightarrow L$ a mapping. Then $a \in L$ is a *fixpoint* of ϕ if $\phi(a) = a$. The element a is a *least fixpoint* of ϕ if it is a fixpoint of ϕ and for all fixpoints b of ϕ, we have $a \leq b$. The notion of *greatest fixpoint* of ϕ is defined dually. The least fixpoint of ϕ is denoted by $lfp(\phi)$ and the greatest fixpoint of ϕ is denoted by $gfp(\phi)$.

The following result is due to Tarski [53] and generalizes an earlier result due to Knaster and Tarski.

Theorem 2.25 *Let L be a complete lattice and $\phi : L \rightarrow L$ a monotonic mapping. Then ϕ has a least fixpoint and a greatest fixpoint. Furthermore, $lfp(\phi) = \bigwedge\{x \mid \phi(x) = x\} = \bigwedge\{x \mid \phi(x) \leq x\}$ and $gfp(\phi) = \bigvee\{x \mid \phi(x) = x\} = \bigvee\{x \mid \phi(x) \leq x\}$.*

Proof: Let $S = \{x \mid \phi(x) \leq x\}$ and $g = \bigwedge S$. Note that $g \leq x$ for all $x \in S$. So by monotonicity of ϕ we have $\phi(g) \leq \phi(x)$, for all $x \in S$. Thus, $\phi(g) \leq x$, for all $x \in S$, and hence $\phi(g) \leq g$. Hence, $g \in S$. Now, to show that g is a fixpoint, it only remains to show that $g \leq \phi(g)$. Note that $\phi(g) \leq g$ implies that $\phi(\phi(g)) \leq \phi(g)$, which in turn implies that $\phi(g) \in S$. Hence, $g \leq \phi(g)$ and so g is a fixpoint of ϕ.

Now let $g' = \bigwedge\{x \mid \phi(x) = x\}$. Since g is a fixpoint we have $g' \leq g$. On the other hand, $\{x \mid \phi(x) = x\} \subseteq S$ and so $g \leq g'$. Hence, $g = g'$ and the proof is complete for $lfp(\phi)$. The proof for $glb(\phi)$ is similar. ∎

Theorem 2.26 *Let L be a complete lattice and $\phi : L \rightarrow L$ a monotonic mapping. Suppose that $a \in L$ and $a \leq \phi(a)$. Then there exists a fixpoint a' of ϕ such that $a \leq a'$. Similarly, if $b \in L$ and $\phi(b) \leq b$, then there exists a fixpoint b' of ϕ such that $b' \leq b$.*

Proof: Let $a' = gfp(\phi)$ and $b' = lfp(\phi)$ and then use Theorem 2.25. ∎

We can now define the notion of ordinal powers of ϕ.

Definition 2.27 Let L be a complete lattice and $\phi : L \to L$ a monotonic mapping. Then we define

$$\begin{aligned}
\phi \uparrow 0 &= \bot \\
\phi \uparrow \alpha &= \phi(\phi \uparrow (\alpha - 1)), \text{ if } \alpha \text{ is a successor ordinal} \\
\phi \uparrow \alpha &= \bigvee\{\phi \uparrow \beta \mid \beta < \alpha\}, \text{ if } \alpha \text{ is a limit ordinal} \\
\phi \downarrow 0 &= \top \\
\phi \downarrow \alpha &= \phi(\phi \downarrow (\alpha - 1)), \text{ if } \alpha \text{ is a successor ordinal} \\
\phi \downarrow \alpha &= \bigwedge\{\phi \downarrow \beta \mid \beta < \alpha\}, \text{ if } \alpha \text{ is a limit ordinal.}
\end{aligned}$$

The following result is a well-known characterization of $lfp(\phi)$ and $gfp(\phi)$ in terms of ordinal powers of ϕ. We state it without proof (see [37]).

Theorem 2.28 *Let L be a complete lattice and $\phi : L \to L$ a monotonic mapping. Then, for any ordinal α, $\phi \uparrow \alpha \leq lfp(\phi)$ and $\phi \downarrow \alpha \geq gfp(\phi)$. Furthermore, there exists ordinals β_1 and β_2 such that $\gamma_1 \geq \beta_1$ implies $\phi \uparrow \gamma_1 = lfp(\phi)$ and $\gamma_2 \geq \beta_2$ implies $\phi \downarrow \gamma_2 = gfp(\phi)$.*

Definition 2.29 Let S be an ordered set and $A \subseteq S$. Then A is said to be a *directed set* if every finite subset of A has an upper bound in A.

Definition 2.30 Let L and P be two complete lattices. A mapping $\phi : L \to P$ is *continuous*, if for every directed set D of L, $\phi(\bigvee(D)) = \bigvee(\phi(D))$.

The least ordinal α such that $\phi \uparrow \alpha = lfp(\phi)$ is called the *closure ordinal* of ϕ. The following theorem shows that under certain conditions, namely when ϕ is continuous, the closure ordinal of ϕ is at most ω.

Theorem 2.31 *Let L be a complete lattice and $\phi : L \to L$ a continuous mapping. Then $lfp(\phi) = \phi \uparrow \omega$.*

Proof: By Theorem 2.28, we only need to show that $\phi \uparrow \omega$ is a fixpoint. Note that $S = \{\phi \uparrow n \mid n < \omega\}$ is directed, since ϕ is continuous and hence monotonic. Thus,

$$\phi(\phi \uparrow \omega) = \phi(\bigvee\{\phi \uparrow n \mid n < \omega\}) = \bigvee\{\phi(\phi \uparrow n) \mid n < \omega\} = \phi \uparrow \omega,$$

using continuity of ϕ. ∎

CHAPTER 3. LOGIC PROGRAMMING BACKGROUND

In this section we will provide some background material about logic programming which we need in our subsequent developments. This material is not intended to be a tutorial, but to provide a frame of reference with respect to which we can discuss extensions and generalizations of some of the standard notions. For most of our discussion in this section, we adopt the notation and definitions used by Lloyd in [37].

The main ideas behind logic programming emerged from the realization that logic can be used as a programming language. This discovery has been, for the most part, attributed to Kowalski [28]. The effectiveness of logic as a specification and declarative language has been known and studied for a long time. What Kowalski showed, however, was that logic also had a procedural interpretation, based on its proof theory, which made it effective as a programming language.

As is the case with any formal system, when studying logic programming languages, we must consider the syntax and semantics of the language. There are two separate but related notions of semantics which must be considered in this context. One is the declarative semantics which is derived from the model theory of the underlying logic, and the other is the procedural or operational semantics which derived from the logic's proof theory. It is the latter which defines the computational model

of the logic programs. The soundness and completeness of such formal systems are generally obtained by verifying the correspondence between the declarative and the procedural semantics.

3.1 Syntax for Standard Logic Programming

A *first order language* consists of an alphabet and the set of all formulas defined over that alphabet. The *alphabet* consists of the following classes of symbols:

1. *variables*, usually denoted by $x, y, z, u, v, \cdots$;
2. *constants*, usually denoted by $a, b, c, d, \cdots$;
3. *function symbols*, usually denoted by $f, g, h, \cdots$;
4. *predicate (relation) symbols*, usually denoted by $p, q, r, \cdots$;
5. *connectives* $\neg$ (negation), $\wedge$ (conjunction), $\vee$ (disjunction), $\rightarrow$ (implication), and $\leftrightarrow$ (equivalence);
6. *quantifiers* $\exists$ (there exists) and $\forall$ (for all); and
7. *punctuation symbols* "(",")", and ",".

So the set of connectives, quantifiers, and punctuation symbols are fixed. Usually, the set of variables, constants, function symbols, and predicate symbols vary from language to language. We assume, however, that the set of variables is infinite and fixed. Each function and predicate symbol has a fixed arity. Constant symbols can be viewed as function symbols of arity 0. Furthermore, we assume the existence of two special constants, called *propositional constants*. These are `true` and `false`.

Definition 3.1 The class of *terms* is defined inductively as follows.

1. a variable is a term;
2. a constant is a term;
3. if f is an n-ary function and $t_1, \cdots, t_n$ are terms, then so is $f(t_1, \cdots, t_n)$.

Definition 3.2 The class of *formulas* is defined as follows.

1. if p is an n-ary predicate symbol and $t_1, \cdots, t_n$ are terms, then $p(t_1, \cdots, t_n)$ is a formula (also called an *atomic formula* or an *atom*).
2. The propositional constants`true` and `false` are formulas;
3. if F and G are formulas, then so are $\neg F$, $F \wedge G$, $F \vee G$, $F \rightarrow G$, and $F \leftrightarrow G$. The formula $F \rightarrow G$ is sometimes written $G \leftarrow F$;
4. if F is a formula and x is a variable, then $\exists x F$ and $\forall x F$ are formulas. The *scope* of $\forall x$ and $\exists x$ is the formula F. An occurrence of x is said to be *bound* if it is within the scope of a quantifier or immediately to its right, otherwise, it is *free*.

A *closed formula* is a formula with no free occurrences of any variables. If F is a quantifier-free formula with variables $x_1, \cdots, x_n$, we write $\forall F$ for $\forall x_1 \cdots \forall x_n F$ and $\exists F$ for $\exists x_1 \cdots \exists x_n F$, called the *universal closure* and the *existential closure* of F, respectively. Formulas of the form $\forall F$ are called *universal formulas* and those of the form $\exists F$ are called *existential formulas*.

Definition 3.3 A *literal* is either an atom or the negation of an atom. A *positive literal* is an atom, and a *negative literal* is the negation of an atom.

Definition 3.4 A *clause* is a formula of the form

$$\forall x_1 \cdots \forall x_n (L_1 \vee \cdots \vee L_m),$$

where $L_1, \cdots, L_m$ are literals and $x_1, \cdots, x_n$ are variables occurring in $L_1 \vee \cdots \vee L_m$.

Since logic programs are generally expressed as a collection of clauses, it will be convenient to adopt a special notation, called the *clausal notation*. In general, we will write the formula $\forall x_1 \cdots \forall x_n (L_1 \vee \cdots \vee L_m)$ in clausal form as

$$A_1, \cdots, A_k \leftarrow B_1, \cdots, B_s,$$

where $A_1, \cdots, A_k$ is the list of all positive literals among $L_1, \cdots, L_m$, called *conclusions*, and $B_1, \cdots, B_s$ are the remaining (negative) literals without the negation symbol, called *premises*. Note that in clausal form all variables are assumed to be universally quantified. The commas in $B_1, \cdots, B_s$ denote conjunctions and the commas in $A_1, \cdots, A_k$ denote disjunctions.

If a clause has only one conclusion ($k = 1$), then it is called a *definite clause*. In that case the conclusion is called the *head* of the clause and the list of premises is called the *body* of the clause. If a clause has no premises ($s = 0$), then it is a *unit clause* and it is written as $A \leftarrow$. A clause with no conclusions and no premises is called the *empty clause* and is denoted by $\square$. The empty clause is interpreted as a contradiction. Informally, a definite clause $A \leftarrow B_1, \cdots, B_s$ is interpreted as "for each assignment of each variable, if $B_1, \cdots, B_s$ is true, then A is true".

Definition 3.5 A *general logic program* is a set of clauses. A *logic program* or *definite program* is a set of definite clauses. In a logic program, the set of all clauses with the same predicate symbol p in the head is called the *definition* of p.

Definition 3.6 A *goal* is a clause of the form

$$\leftarrow B_1, \cdots, B_s.$$

In other words, a goal is a clause with no conclusions. Each B_i $(i = 1, \cdots, s)$ is called a *subgoal* of the goal.

If $y_1, \cdots, y_r$ are the variables occurring in the goal $\leftarrow B_1, \cdots, B_s$, then the goal is the clausal notation for the formula

$$\forall y_1 \cdots \forall y_r (\neg B_1 \vee \cdots \vee \neg B_s),$$

or equivalently,

$$\neg \exists y_1 \cdots \exists y_r (B_1 \wedge \cdots \wedge B_s).$$

Definition 3.7 A *Horn clause* is either a definite clause or a goal.

3.2 Declarative Semantics of Logic Programs

In order to be able to consider the truth or falsity of formulas in any first-order language, we must first attach some meaning to all the symbols appearing in such formulas. Of course, the quantifiers and connectives have a fixed meaning, however, the meaning attached to constants, function symbols, and predicate symbols can vary. The assignment of meaning to these symbols is accomplished by means of an *interpretation*. We are specially interested in interpretations for which the formula expresses a true statement in that interpretation. Such an interpretation is called a *model* of the formula. The main objective in study of interpretations is to find some distinguished interpretation, called the *intended* interpretation, which provides the

"desired" meaning for formulas. We now make these notions precise. The following definitions are given in the context of a particular first-order language $\mathcal{L}$.

Definition 3.8 An *interpretation* of $\mathcal{L}$ consists of the following components:

1. a non empty set D called the *domain of discourse* (or simply the *domain*),
2. for each constant in $\mathcal{L}$, an assignment of an element in D,
3. for each n-ary function symbol, an assignment of a mapping from D^n to D,
4. for each n-ary predicate symbol in $\mathcal{L}$, the assignment of a mapping from D^n to the set $\{\texttt{true}, \texttt{false}\}$.

Definition 3.9 Let I be an interpretation of $\mathcal{L}$. A *variable assignment* with respect to I is an assignment of an element in the domain of I to each variable.

Definition 3.10 Let I be an interpretation of $\mathcal{L}$ and let ν be a variable assignment. The *term assignment with respect to I and ν* of the terms in $\mathcal{L}$ is defined as follows:

1. each variable is given its assignment according to ν,
2. each constant is given its assignment according to I,
3. if $t'_1, \cdots, t'_n$ are term assignments of terms $t_1, \cdots, t_n$ with respect to I and ν and f' is the assignment of the function symbol f, then $f'(t'_1, \cdots, t'_n)$ is the term assignment of $f(t_1, \cdots, t_n)$ with respect to I and ν.

Definition 3.11 Let I be an interpretation of $\mathcal{L}$ with the domain D and let ν be a variable assignment. Then a formula is given a *truth value* with respect to I and ν as follows.

1. If the formula is an atom $p(t_1, \cdots, t_n)$, then the truth value is obtained by evaluating $p'(t'_1, \cdots, t'_n)$, where p' is the mapping assigned to p by I and $t'_1, \cdots, t'_n$ are term assignments of $t_1, \cdots, t_n$ with respect to I and ν.

2. If the formula is of the form $\neg F$, $F \wedge G$, $F \vee G$, $F \rightarrow G$, or $F \leftrightarrow G$, then the truth value is given according to the standard truth tables for logical connectives.

3. If the formula is of the form $\exists x F$, then the formula has the truth value `true` if there exists a $d \in D$ such that F has the truth value `true` with respect I and $\nu[x/d]$, where $\nu[x/d]$ is the same as ν with d assigned to x; otherwise the truth value is `false`.

4. If the formula is of the form $\forall x F$, then the formula has the truth value `true` if for all $d \in D$ such that F has the truth value `true` with respect I and $\nu[x/d]$; otherwise the truth value is `false`.

Note that the truth value of a closed formula does not depend on any variable assignments. This means that we can unambiguously speak about the truth value of a closed formula with respect to an interpretation.

Definition 3.12 Let I be an interpretation of $\mathcal{L}$ and let F be a closed formula. The interpretation I is called a *model* for F if the truth value of F with respect to I is `true`. If S is a set of closed formulas, then I is a *model* for S if it is a model for each $F \in S$.

Definition 3.13 Let S be a set of closed formulas. Then a closed formula F is a *logical consequence* of S if every model for S is also a model for F. Alternatively, we say that F logically follows from S, denoted by $S \models F$.

We sometimes write $S \models G$ for an arbitrary formula G to mean that $S \models \forall(G)$.

The following well-known result provides the declarative basis for the methodology used in the procedural semantics in standard logic programming systems. This methodology is often referred to as *refutation mechanism* and will be discussed later.

Theorem 3.14 *Let S be a set of formulas and F a closed formula of the first-order language $\mathcal{L}$. Then F is a logical consequence of S if and only if $S \cup \{\neg F\}$ has no model.*

Since logic programs are basically a set of Horn clauses, the basic problem of logic programming is verifying that a given goal (usually an atom) A, is the logical consequence of a program P. As the above theorem suggests, this problem reduces to the problem of showing that $P \cup \{\neg A\}$ has no models. In other words, the system must prove that *every* interpretation of $P \cup \{\neg A\}$ is not a model. Fortunately, it turns out that only a small subclass of interpretations called *Herbrand interpretations* needs to be investigated, thus qualitatively reducing the complexity of the above task. We now consider Herbrand interpretations and their role in the semantics of logic programs.

Definition 3.15 1. A *ground term* is a term in which no variables occur.

2. A *ground atom* is an atom in which no variables occur.

3. A *ground instance* of a clause C is a clause obtained from C by removing all the quantifiers and replacing each variable in C by a ground term.

4. The *Herbrand universe* $U_{\mathcal{L}}$ is the set of all ground terms.

5. The *Herbrand base* $B_{\mathcal{L}}$ is the set of all ground atoms.

Definition 3.16 An interpretation I for $\mathcal{L}$ is a *Herbrand interpretation* if it satisfies the following conditions:

1. The domain of I is the Herbrand universe $U_{\mathcal{L}}$;
2. constant in $\mathcal{L}$ are assigned to themselves in $U_{\mathcal{L}}$;
3. if f is an n-ary function symbol in $\mathcal{L}$, then f is assigned to the mapping from $(U_{\mathcal{L}})^n$ to $U_{\mathcal{L}}$ defined by $(t_1, \cdots, t_n) \rightarrow f(t_1, \cdots, t_n)$.

Note that no restriction is placed on the assignment of predicate symbols in $\mathcal{L}$, and thus different assignments of predicate symbols can give rise to different interpretations. Furthermore, it is convenient to identify a Herbrand interpretation by the set of all ground atoms which are true with respect to the interpretation. This is possible, since for Herbrand interpretations, the assignment of constants and function symbols is fixed. Thus we can identify Herbrand interpretations as subsets of the Herbrand base.

Definition 3.17 A *Herbrand model* for a set of closed formulas S of $\mathcal{L}$ is a Herbrand interpretation of $\mathcal{L}$ which is a model of S.

It is the following well-known result which verifies that when considering the semantics of logic programs, we need only be concerned with Herbrand interpretations.

Theorem 3.18 *Let S be a set of clauses. Then S has a model if and only if S has a Herbrand model.*

For convenience, when considering Herbrand interpretations of a set of formulas S, we often restrict our attention to constants, function symbols, and predicate symbols that appear in S rather than the underlying first-order language. In this case, we

may refer to the Herbrand universe U_S and Herbrand base B_S of S and identify the Herbrand interpretations of S with subsets of the Herbrand base of S. In particular, for a program P, we can now consider the Herbrand universe U_P and the Herbrand base B_P of P.

We can observe that for a program P, the power set of B_P, $\mathcal{P}(B_P)$, which is the set of all Herbrand interpretations of P, is a complete lattice under set inclusion $\subseteq$. The top element of the lattice is B_P and the bottom element is $\emptyset$. Hence, the least upper bound of any set of Herbrand interpretations is the union of all Herbrand interpretations in the set. This union is, of course, itself a Herbrand interpretation. The greatest lower bound of the set is the intersection of Herbrand interpretations.

As discussed earlier, we are interested in a particular model, called the intended model, which is regarded as the canonical interpretation of a program. The intended model, in standard logic programming is taken to be the least Herbrand model. Let us now formalize these notions [37].

Theorem 3.19 *Let P be a program and $\{M_i\}_{i\in I}$ be a non-empty set of Herbrand models for P. Then $\cup_{i\in I}M_i$ is a Herbrand model for P.*

Definition 3.20 Let P be a program. Suppose that P has a Herbrand model and let $\{M_i\}_{i\in I}$ be the set of all Herbrand models for P. Then $\cup_{i\in I}M_i$ is called the *least Herbrand model* for P, denoted by M_P.

Theorem 3.21 *Let P be a definite program. Then $M_P = \{A \in B_P \mid P \models A\}$.*

In order to capture the notion of computation within the declarative semantics for logic programs, it is necessary to give a characterization of the least Herbrand

model based on the concept of fixpoints. This characterization is called the *fixpoint semantics.*

Definition 3.22 Let P be a definite program. The mapping $T_P : \mathcal{P}(B_P) \rightarrow \mathcal{P}(B_P)$ is defined as follows. Let I be an Herbrand interpretation. Then $T_P(I) = \{A \in B_P \mid A \leftarrow A_1, \cdots, A_n$ is a ground instance of a clause in P and $\{A_1, \cdots, A_n\} \subseteq I\}$.

The T_P operator provides a link between the declarative and the procedural semantics for logic programs. Note that T_P is a monotonic mapping. Furthermore, it can be easily shown that T_P is continuous (see [37]). Hence, as indicated earlier in our discussion of lattices, T_P has a least fixpoint and a greatest fixpoint, and furthermore, the least fixpoint can be reached in ω steps. In fact, the Herbrand interpretations that are models can be characterized in terms of T_P.

Theorem 3.23 *Let P be a definite program and I a Herbrand interpretation of P. Then I is a (Herbrand) model for P if and only if $T_P(I) \subseteq I$.*

Now we state a well-known result, providing the fixpoint characterization of the least Herbrand model. Both this and the aforementioned result are originally due to van Emden and Kowalski [56].

Theorem 3.24 *Let P be a definite program. Then $M_P = lfp(T_P) = T_P \uparrow \omega$.*

Before discussing the procedural semantics of logic programs we present some of the preliminary concepts that we will need in subsequent developments.

3.3 Substitutions and Unification

In logic programming variables are assigned values by means of substitutions, and if viewed procedurally, special type of substitutions called unifiers provide the primary mechanism for parameter passing. Substitutions, renamings, composition of substitutions, unifiers, and most general unifiers are defined in the standard manner as detailed in [37].

Definition 3.25 A *substitution* θ is a mapping from variables to terms such that $v\theta \neq v$ for only finitely many v. Every substitution is uniquely represented by a finite set of the form $\{v_1/t_1, \cdots, v_n/t_n\}$, where each v_i is a variable, each t_i is a term distinct from v_i, and the variables $v_1, \cdots, v_n$ are distinct. Each element v_i/t_i is called a *binding* for v_i. The empty substitution is called the *identity* substitution and is denoted by ε. θ is *ground* if each t_i is a ground term, and it is *variable-pure* if each t_i is a variable. θ in *injective* if $t_i \neq t_j$ whenever $i \neq j$. The domain of θ is $\{v_1, \cdots, v_n\}$ and is denoted by $dom(\theta)$. The set of variables occurring in the range of θ, i.e., $\cup_{i=1}^{n} vars(t_i)$, is denoted by $vrange(\theta)$.

Definition 3.26 An *expression* is either a term or a formula. A *simple expression* is either a term or an atom. For an expression E, $vars(E)$ denotes the set of all variables that occur in E.

Definition 3.27 A substitution θ is further extended to a mapping from the set of expressions into itself in the following way: Let $\theta = \{v_1/t_1, \cdots, v_n/t_n\}$ and let E be an expression. Then $E\theta$, the *instance of E by θ*, is the expression obtained from E by simultaneously replacing each occurrence of the variable v_i in E by the term t_i,

$i = 1, \cdots, n$. For a set S of expressions, $S\theta = \{E\theta \mid E \in S\}$. For two expressions E_1 and E_2, we write $E_1 \leq E_2$, if $E_2 = E_1\sigma$ for some substitution σ. If $E\theta$ is ground, then $E\theta$ is called a *ground instance* of E.

Definition 3.28 Let θ and σ be substitutions. By the *composition* $\theta\sigma$ of θ and σ we mean their composition as transformations of the set of expressions in the usual sense of functional composition. If $\theta = \{u_1/s_1, \cdots, u_m/s_m\}$ and $\sigma = \{v_1/t_1, \cdots, v_n/t_n\}$, then the representation of $\theta\sigma$ is obtained from the set

$$\{u_1/s_1\sigma, \cdots, u_m/s_m\sigma, v_1/t_1, \cdots, v_n/t_n\}$$

by deleting any binding $u_i/s_i\sigma$ for which $u_i = s_i\sigma$ and deleting any binding v_j/t_j for each $v_j \in \{u_1, \cdots, u_m\}$.

Definition 3.29 Let $\sigma = \{v_1/t_1, \ldots, v_n/t_n\}$ be a substitution and U a set of variables.

1. $\sigma_U = \{v_{i_1}/t_{i_1}, \ldots, v_{i_k}/t_{i_k}\}$, where $\{v_{i_1}, \ldots, v_{i_k}\} = U \cap dom(\sigma)$.

2. $\sigma_{-U} = \sigma_{(dom(\sigma)-U)}$.

Let σ and τ be substitutions and E an expression.

3. $\sigma_E = \sigma_{vars(E)}$, $\quad \sigma_{-E} = \sigma_{-vars(E)}$;

4. $\sigma_\tau = \sigma_{dom(\tau)}$, $\quad \sigma_{-\tau} = \sigma_{-dom(\tau)}$;

5. (*Application*) $\sigma \circ \tau = (\sigma\tau)_\sigma$;

Note that for any set U of variables,

$$\sigma = \sigma_U \uplus \sigma_{-U},$$

where $\uplus$ denotes disjoint union. In particular, $\sigma = \sigma_E \uplus \sigma_{-E}$ for every expression E, and $\sigma = \sigma_\tau \uplus \sigma_{-\tau}$ for every substitution τ. Note also that the usual composition of substitutions can be expressed in terms of application as follows:

$$\sigma\tau = \sigma \circ \tau \uplus \tau_{-\sigma}.$$

Let A be a set of substitutions and τ any substitution. Then $\tau A = \{\tau\alpha \mid \alpha \in A\}$ and $A\tau = \{\alpha\tau \mid \alpha \in A\}$. The following theorem states some of the well-known elementary properties of substitutions. We state it without proof [37].

Theorem 3.30 *Let θ, σ, and γ be substitutions. Then*

1. $\theta\varepsilon = \varepsilon\theta = \theta$.
2. $(E\theta)\sigma = E(\theta\sigma)$, *for all expressions* E.
3. $(\theta\sigma)\gamma = \theta(\sigma\gamma)$.

The first part of the above theorem illustrates that ε acts as the left and right identity for composition. Furthermore, the associativity of substitutions allows us to omit parentheses when writing a composition $\theta_1\theta_2\cdots\theta_n$ of substitutions.

Definition 3.31 Let θ and η be substitutions.

1. θ is an *instance of* η, in symbols $\eta \leq \theta$, if $\theta = \eta\alpha$ for some substitution α. In this case we say that θ is an *instance of* η *by* α.
2. Let U be a set of variables. We say that θ is a *variant of* η *with respect to* U if there are instances θ' and η' of θ and η, respectively, such that $\theta_U = \eta'_U$ and $\eta_U = \theta'_U$.

3. We say that θ is a *variants* of η, in symbols $\theta \equiv \eta$, if it is a variant of η with respect to the set of all variables, i.e., if each is an instance of the other ($\eta \leq \theta$ and $\theta \leq \eta$).

It is easy to see that $\leq$ is a preordering on substitutions and $\equiv$ is an equivalence relation. In particular it is symmetric, so that θ is a variant of η w.r.t U iff η is a variant of θ w.r.t. U. We say that θ is a *variant of* η *w.r.t.* a given expression E if θ is a variant of η w.r.t. $vars(E)$, i.e., if $E\theta = E\eta'$ and $E\eta = E\theta'$ for some θ' and η' such that $\theta \leq \theta'$ and $\eta \leq \eta'$.

The key feature of the variant relation is the fact that, given any expression E, any substitution θ, and any finite set V of variables, there is a variant η of θ w.r.t. E such that $vars(E\eta) \cap V = \emptyset$. For substitutions that are variants in the absolute sense we can be even more explicit about their connection using the notion of a renaming.

Definition 3.32 Let U be a set of variables. A substitution α is a *renaming w.r.t.* U if α is injective and variable-pure, and

$$vrange(\alpha) \cap (U - dom(\alpha)) = \emptyset.$$

Definition 3.33 A substitution θ is a *renaming* of substitution η, denoted by $\eta \leq_r \theta$, if $\theta = \eta\alpha$, where α is some renaming with respect to $vrange(\eta)$. Wanting to be specific, we sometimes say that $\eta \leq_r \theta$ via α. Thus, $\eta \leq_r \theta$ if and only if for some injective, variable pure substitution α,

1. $\eta \leq \theta$ via α, and

2. $vrange(\alpha) \cap (vrange(\eta) - dom(\alpha)) = \emptyset$.

We write $\theta \equiv_r \eta$ just when $\theta \leq_r \eta$ and $\eta \leq_r \theta$.

In fact, the notions of $\equiv_r$ and $\equiv$ are equivalent, as we shall prove in the following lemma. First, we need to make some easily verified observations about substitutions.

Definition 3.34 Let θ be a substitution. The *discriminant* of θ, denoted by $D(\theta)$, is the set of variables

$$D(\theta) = dom(\theta) - vrange(\theta).$$

Let θ, η, and γ be substitutions such that $\theta = \eta\gamma$. We have the following facts:

A. Without loss of generality we may assume that $dom(\gamma) \cap D(\eta) = \emptyset$ (or else we may choose $\gamma' \subseteq \gamma$ that also satisfies $\theta = \eta\gamma'$ and does not have this property).

B. $vrange(\eta) - dom(\gamma) \subseteq vrange(\theta)$.

C. $D(\gamma) \cap vrange(\eta) \cap vrange(\theta) = \emptyset$.

D. $\theta = \theta\gamma$ if and only if $dom(\gamma) \subseteq D(\theta)$.

Lemma 3.35 *Let θ and η be substitutions. Then $\theta \equiv \eta$ if and only if $\theta \equiv_r \eta$.*

Proof: obviously, $\theta \leq_r \eta$ ($\theta \equiv_r \eta$) implies $\theta \leq \eta (\theta \equiv \eta)$. In the following we will prove that $\theta \equiv \eta$ implies $\theta \equiv_r \eta$.

Suppose that $\theta \equiv \eta$. By fact A, let α and β be substitutions such that $\theta = \eta\alpha$, $\eta = \theta\beta$, and $dom(\alpha) \cap D(\eta) = \emptyset = dom(\beta) \cap D(\theta)$. Thus, $\theta = \eta\alpha = \theta(\beta\alpha)$, and $\eta = \theta\beta = \eta(\alpha\beta)$.

By fact D, $dom(\beta\alpha) \subseteq D(\theta)$ and $dom(\alpha\beta) \subseteq D(\eta)$. Hence,

$$\begin{aligned} dom(\beta \circ \alpha) &\subseteq dom(\beta) \cap dom(\beta\alpha) \\ &\subseteq dom(\beta) \cap D(\theta) \\ &= \emptyset, \end{aligned}$$

and similarly, $dom(\alpha \circ \beta) = \emptyset$. These imply that $\alpha \circ \beta = \beta \circ \alpha = \varepsilon$ and so α and β must be variable-pure and one-to-one. Moreover, it is immediate that $dom(\alpha) = vrange(\beta)$ and $dom(\beta) = vrange(\alpha)$.

It remains to show that α and β are renamings with respect to $vrange(\eta)$ and $vrange(\theta)$, respectively. In other words, we have to argue that

$$\begin{aligned} vrange(\alpha) \cap (vrange(\eta) - dom(\alpha)) &= \emptyset, \text{ and} \\ vrange(\beta) \cap (vrange(\theta) - dom(\beta)) &= \emptyset. \end{aligned}$$

Indeed, given facts C and D, we have

$$\begin{aligned} & vrange(\alpha) \cap (vrange(\eta) - dom(\alpha)) \\ &= (vrange(\alpha) \cap (vrange(\eta) - dom(\alpha))) - vrange(\hat{\theta}) \\ &= (dom(\beta) \cap (vrange(\eta) - vrange(\beta))) \cap vrange(\theta) \\ &= D(\beta) \cap vrange(\eta) \cap vrange(\theta) \\ &= \hat{\emptyset} \quad (\text{since } \eta = \hat{\theta}\beta). \end{aligned}$$

The second claim is proved similarly. ■

In the sequel we often use the expression "θ is a renaming of η" as a synonym for "θ is a variant of η". Also, when we say that a substitution α is "unique up to renaming", we mean that α can be any member of a $\equiv$-equivalence class.

Definition 3.36 A substitution θ is *idempotent* if $\theta\theta = \theta$.

The class of idempotent substitutions exhibits some interesting properties which have been extensively studied [32, 38, 13]. In particular, it can be easily verified that σ is an idempotent substitution if and only if $dom(\sigma) \cap vrange(\sigma) = \emptyset$.

We also use the standard notion of unification due originally to Herbrand [25] and later to Robinson [45]. See also [37].

Definition 3.37 Let S be a finite set of simple expressions. A substitution θ is called a *unifier* for S if $S\theta$ is a singleton. A unifier θ of S is called a *most general unifier* (*mgu*) for S if $\theta \leq \sigma$ for each unifier σ of S. The set S is called *unifiable* if it has a unifier.

The following algorithm, called the *unification algorithm*, takes a finite set of simple expressions as input and outputs an mgu if the set is unifiable. If the set is not unifiable, then the algorithm reports failure. Before giving the unification algorithm we need to define the notion of the disagreement set of a set of expressions.

Definition 3.38 Let S be a finite set of simple expressions. The *disagreement set* of S is defined as follows. Locate the leftmost symbol position at which not all expressions in S have the same symbol and extract from each expression in S the subexpression beginning at that position. The set of all such leftmost subexpressions is the disagreement set for S.

In the unification algorithm, S denotes a finite set of simple expressions:

1. Let $k = 0$ and $\sigma_0 = \varepsilon$.

2. If $S\sigma_k$ is a singleton, then stop and output σ_k as an mgu of S. Otherwise, find the disagreement set D_k of $S\sigma_k$.

3. If there exists a variable v and a term t in D_k such that v does not occur in t, then let $\sigma_{k+1} = \sigma_k\{v/t\}$; increment k and go back to step 2. Otherwise, stop; S is not unifiable.

The above algorithm is non-deterministic since there may be several choices of v and t in step 3. But, the application of two mgu's to an expression leads to expressions that are variants. It is easy to see that the algorithm terminates since S contains finitely many variables and each pass through step 3 eliminates one variable. It is also easy to verify that each mgu produced by the unification algorithm is idempotent.

The following result, which is due to Robinson [45], guarantees the existence of an mgu, if a set of expressions is unifiable.

Theorem 3.39 (Unification Theorem) *Let S be a finite set of simple expressions. 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 failure.*

3.4 Procedural Semantics for Logic Programs

The procedural semantics of logic programs is rooted in the proof theory of the underlying first-order language. It provides the means of transforming a set of Horn clauses into an executable program. There are several approaches for characterizing the operational semantics of logic programs. Most commonly, they are based on the resolution principle originally introduced by Robinson [45], in the more general context of first-order logic. Later, Kowalski [28] employed these notions along with a

procedural interpretation of Horn clauses to provide the basis of what is now known as logic programming. In general, resolution theorem provers are refutation systems. In order to prove a formula, the negation of the formula is added to the set of axioms (or, in the context of logic programming, to the program) and an attempt is made to derive a contradiction. Recall that a in a logic programming system, a goal is defined to be the negation of an existential formula. From a programming point of view, given an existential formula $\neg\exists y_1 \cdots \exists y_r(B_1 \wedge \cdots \wedge B_s)$, we are not only interested in proving that it is a logical consequence of a program, but also, in constructively obtaining bindings for the variable $y_1, \cdots, y_n$ which result from such computation. The refutation procedure which is commonly used in logic programming is based on SLD-resolution first introduced by Kowalski [28]. We now study these concepts more closely. First we define the notion of SLD-refutation.

Definition 3.40 Let G be the goal $\leftarrow A_1, \cdots, A_m, \cdots, A_k$ and let C be a clause $A \leftarrow B_1, \cdots, B_r$. Then G' is *derived* from G and C using mgu θ if the following conditions hold:

1. A_m is an atom, called the *selected* atom, in G;
2. θ is an mgu of A_m and A;
3. G' is the goal $\leftarrow (A_1, \cdots, A_{m-1}, B_1, \cdots, B_r, A_{m+1}, \cdots, A_k)\theta$. G' is called the *resolvant* of G and C.

Definition 3.41 Let P be a definite program and G a definite goal. An *SLD-derivation* of $P \cup \{G\}$ consists of a (finite or infinite) sequence $G_0 = G, G_1, \cdots$ of goals, a sequence $C_1, C_2, \cdots$ of variants of program clauses of P and a sequence

$\theta_1, \theta_2, \cdots$ of mgu's such that for each $i \geq 0$, G_{i+1} is derived from G_i and C_{i+1} using θ_{i+1}.

In the above definition, the C_i are taken to be suitable variants of program clause so that C_i does not have any variables appearing in the derivation up to G_i. This process is called *standardizing variables apart.* It is necessary to avoid variable conflicts when unifying the subgoals with heads of clauses.

Definition 3.42 An *SLD-refutation* of $P \cup \{G\}$ is a finite SLD-derivation of $P \cup \{G\}$ which has the empty clause $\square$ as the last goal in the derivation.

Clearly, it is possible that an SLD-derivation is infinite. Finite, (SLD-)derivations may be *successful* or *failed.* A successful derivation is simply an SLD-refutation and a failed derivation is one that ends in a subgoal which does not unify with the head of any clause. The *success set* of a program P is the set of all $A \in B_P$ such that $P \cup \{\leftarrow A\}$ has a successful derivation. The notion of the success set is the procedural counterpart of the least Herbrand model. In fact we have the following well-known result [37].

Theorem 3.43 *The success set of a definite program is contained in its least Herbrand model.*

We also have the following notion of a computed answer which represents the binding for variables of the goal obtained in the derivation.

Definition 3.44 Let P be a definite program and G a definite goal. A *computed answer* θ for $P \cup \{G\}$ is the substitution obtained by restricting the composition

$\theta_1\theta_2\cdots\theta_n$ to the variables of G, where $\theta_1,\cdots,\theta_n$ is the sequence of mgu's used in an SLD-refutation of $P \cup \{G\}$.

When considering SLD-refutations for a program, the system must be able to choose the selected atom in a deterministic fashion. This goal is obtained by what is called the *computation rule*. For example, the computation rule used in the programming language Prolog is to take the leftmost atom in the body of a subgoal as the selected atom (the one to be unified with the head of some clause).

The search space (possible computation paths) in an SLD-refutation can be viewed as a tree called the *SLD-tree*. Each branch of the SLD-tree represents a derivation of $P \cup \{G\}$, for a program P and a goal G. For example [37], consider the program P with the following three clauses:

1. $p(x,z) \leftarrow q(x,y), p(y,z)$

2. $p(x,x) \leftarrow$

3. $q(a,b) \leftarrow$

and the goal $\leftarrow p(x,b)$. Figure 3.1 depicts an SLD-tree for for this program and goal. The computation rule used for the derivations represented by the tree is the "leftmost" computation rule described above. Note that the tree has two *success branches* corresponding the computed answers $\{x/a\}$ and $\{x/b\}$. The tree in this example also has a failure branch. A different choice of computation rule would result in a different SLD-tree, although, it can be shown that all such trees will have two success branches corresponding to the above computed answers.

We end this section with the statements of the Soundness and Completeness theorems for SLD-resolution. These results establish the correspondence between the

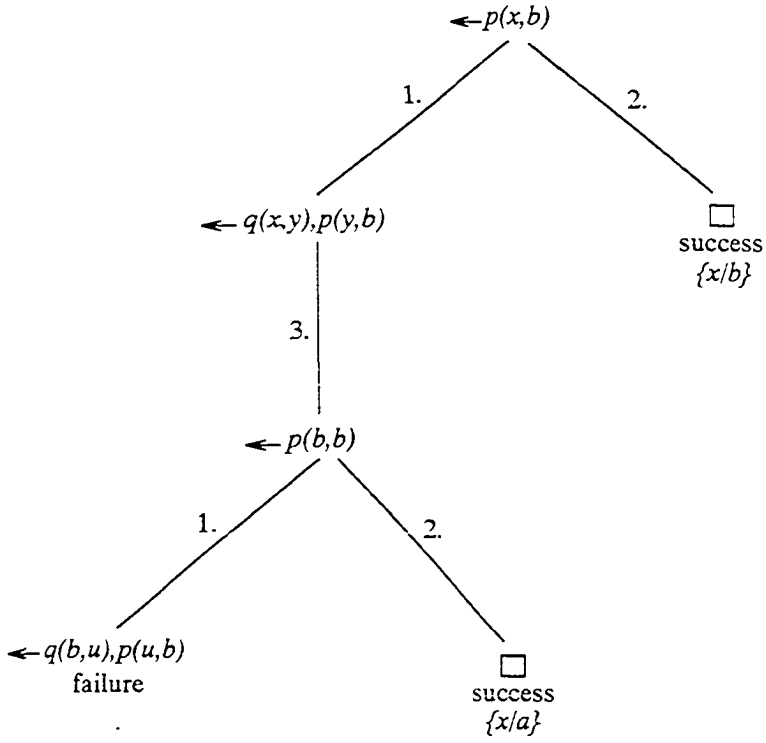


Figure 3.1: An SLD-tree

declarative and procedural semantics of logic programs. The proofs can be found, among other places, in [37].

Theorem 3.45 (Soundness of SLD-resolution) *Let P be a definite program and G a definite goal. If θ is a computed answer for $P \cup \{G\}$, then $G\theta$ is a logical consequence of P.*

Theorem 3.46 (Completeness of SLD-resolution) *Let P be a definite program and G a definite goal. If $G\theta$ is a logical consequence of P, then $P \cup \{G\}$ has an SLD-refutation with the computed answer σ such that for some substitution γ, $G\theta = G\sigma\gamma$.*

3.5 The Role of Negation in Logic Programming

SLD-resolution only applies to sets of Horn clauses with exactly one goal clause. Since only positive information can be a logical consequence of a program, it is not possible to deduce negative information using SLD-resolution. The problem is that if P is a definite program, and $A \in B_P$, then we cannot prove that $\neg A$ is a logical consequence of P since $P \cup \{\leftarrow A\}$ has a model, namely B_P itself. It is therefore necessary to invoke special rules of inference which allow for the deduction of negative information. The most well-known and commonly used of these rules are the Closed World Assumption (CWA) and its procedural counterpart, the Negation as Finite Failure (or simply Negation as Failure) rule.

The Closed World Assumption, introduced by Reiter [43], is a special rule of inference stating that if a ground atom A is not a logical consequence of a program, then infer $\neg A$. To introduce the Negation as Failure rule, we first need to define the notion of the finite failure set of a program.

Definition 3.47 Let P be a definite program. Then the *finite failure set* of P is the set of all atoms $A \in B_P$ for which there exists a finitely failed SLD-tree for $P \cup \{\leftarrow A\}$, that is, one which is finite and contains no

Clearly, if A is in the finite failure set of P, then A is not a logical consequence of P. This means that every SLD-tree for $P \cup \{\leftarrow A\}$ contains only infinite or failure branches. The Negation as Failure rule , originally introduced by Clark [8], states that if A is in the finite failure set of P, then infer $\neg A$. Procedurally, this is accomplished by starting with the goal $\leftarrow A$. If the system can construct a finitely failed SLD-tree, then this is interpreted as the deduction of $\neg A$. Of course, it is

possible that every SLD-tree has at least one infinite branch, in which case (in the absence of a mechanism for detecting infinite branches). the system will never be able to infer $\neg A$, eventhough A may, in fact, not be a logical consequence of the program. Hence, the finite failure set is a subset of the complement of the success set, indicating that Negation as Failure rule is less powerful than the CWA.

The above mechanism suggests that we can extend SLD-resolution to be able to deal with negated subgoals, in the context of programs which may have negation in the body of their clauses. The procedural semantics obtained by combining SLD-resolution with the Negation as Failure rule is called SLDNF-resolution. It works essentially as follows: If during the derivation a negated subgoal $\leftarrow \neg A$ is reached, the system will try the goal $\leftarrow A$. If $\leftarrow A$ succeeds (using SLDNF-resolution), then $\leftarrow \neg A$ fails, and if it fails, then $\leftarrow \neg A$ succeeds. If the program is definite (does not contain negation in the body of its clauses), then SLDNF-resolution simply reduces to SLD-resolution.

SLDNF-resolution allows us to extend the notion of a program so that the clause bodies can contain negation. This is desirable because, in practice, definite programs often lack the necessary expressive power in many contexts. For example, consider the following two clauses:

1. $diff(x, y) \leftarrow member(z, x), \neg member(z, y)$

2. $diff(x, y) \leftarrow \neg member(z, x), member(z, y)$

Given appropriate definition for the predicate *member*, the above clauses define when two sets are different. Without the use of negation it would be difficult to capture the intuitive declarative meaning of this relation in a logic program.

The following example illustrates the use of SLDNF-resolution. Consider the following program P.

1. $p(x) \leftarrow q(x), \neg r(x)$

2. $q(a) \leftarrow$

3. $r(a) \leftarrow$

Suppose that we are interested in establishing that $p(a)$ is not a logical consequence of P. This is in fact verified using SLDNF-resolution, since starting with the goal $\leftarrow p(a)$, we can construct a finitely failed derivation. This derivation is depicted in Figure 3.2. The dotted line pointing from $\leftarrow \neg r(a)$ to $\leftarrow r(a)$ indicates that upon reaching the negated subgoal, the system will attempt an independent evaluation of its positive component. Since $\leftarrow r(a)$ succeeds, the derivation for $\leftarrow p(a)$ fails finitely. It is worth noting that if we add a new clause, $p(a) \leftarrow$, to P, then we can no longer deduce $p(a)$, since in that case not all branches of the SLDNF-tree for $P \cup \{\leftarrow p(a)\}$ will be finitely failed.

Unfortunately, as soon as negation is introduced into the body of program clauses, certain semantic problems arise. In particular, Negation as Failure is not sound with respect to classical semantics of first-order logic. For example, consider a program $P = \{p \leftarrow \neg q\}$. Given this program, the goal $\leftarrow q$ fails, and so using Negation as Failure, $\leftarrow p$ succeeds. However, p is not a logical consequence of the program P. This situation is reflected in different characterizations of the declarative semantics for logic programs. For instance, note that the Herbrand models of P are $\{p, q\}$, $\{p\}$, and $\{q\}$. Hence, P has no minimal models, which could be taken as the intended model for the program. When considering the fixpoint semantics of

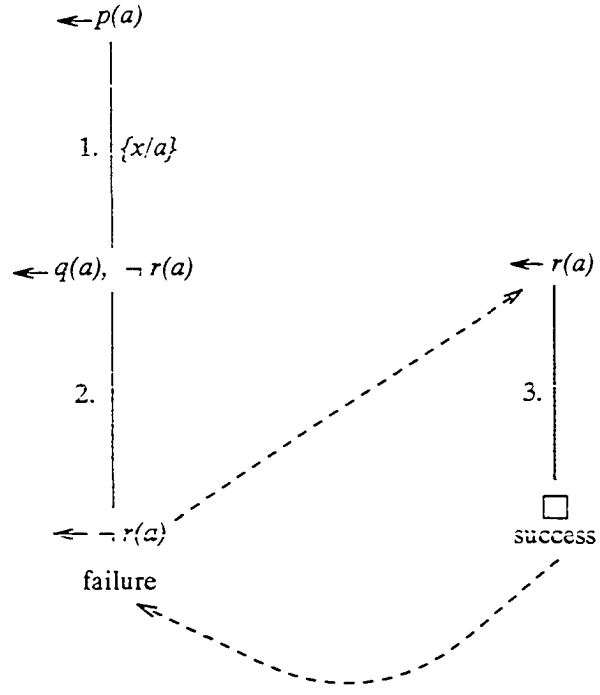


Figure 3.2: Example of SLDNF-resolution

logic programs, the difficulty with negation is reflected in the fact that the semantic operator T_P of a program P is not monotonic in general. Thus, the existence of a least fixpoint can no longer be guaranteed.

The SLDNF-resolution also suffers from a lack of completeness. This is because it cannot in general deal with non-ground negative subgoals. For example, given a goal $\leftarrow p(x)$, the formula to be proved as a logical consequence of the program is actually the existential formula $\exists p(x)$. So the goal $\leftarrow \neg p(x)$ is actually taken to mean $\leftarrow \exists \neg p(x)$. Now, if the goal $\leftarrow \exists p(x)$ succeeds, it is not in general sound to allow $\leftarrow \exists \neg p(x)$ to fail (both $\exists p(x)$ and $\exists \neg p(x)$ may be logical consequences of the program). This is why in SLDNF-resolution, Negation as Failure is only allowed on ground negative subgoals.

Another problem with SLDNF-resolution is that the choice of the computation rule may affect the behavior of programs. In SLD-resolution, if a goal succeeds with answer θ using one computation rule, then it does so using any other computation rule. This is no longer the case with SLDNF-resolution. For example, consider the program:

1. $p \leftarrow p, q$

2. $r \leftarrow \neg p$

The goal $\leftarrow r$ succeeds if we use the "rightmost literal rule", but it will fail if the "leftmost literal rule" is taken as the computation rule. These examples illustrate the difficulty of finding a sufficiently simple declarative semantics with respect to which Negation as Failure is sound and yet it is complete.

The task of providing appropriate semantics for negation has been subject of considerable research and study in logic programming. There have been many attempts to provide declarative semantics with respect to which Negation as Failure is sound. Some of these approaches involve replacing a program P with another program obtained by applying some transformation to P. The idea behind such a transformation is that when a program P is given, it should be taken as a shorthand notation for another set of formulas. Examples of these approaches include the notion of *completed programs* of Clark [8], and Reiter's Closed World Assumption.

Another approach involves redefining the notion of logical consequence with respect to some non-classical logic. Examples of these are logic programming systems based on three-valued logics [31, 14, 29, 30], Autoepistemic logics [20, 39, 42], and linear logics [7].

Yet another approach of dealing with semantics of negation interprets the notion of logical consequence not as being true in all models of a program P, but only in models of certain kind. Examples of such specialized or restricted models include minimal models [36, 49, 6], perfect models [41, 40], well-founded models [46, 55], and stable models [21].

A detailed study of these approaches is outside the scope of this thesis. The interested reader is referred to [49, 50]. As discussed in the introduction, part of the motivation behind this work, and especially, Section 6, is to give semantics for logic programming based on multi-valued logics, which can adequately and effectively deal with negation within programs.

CHAPTER 4. A THEORY OF BILATTICES

The idea of constructing multi-valued logics by choosing truth values assigned to statements from a partially ordered set of truth values is not a new one. Some examples include approaches by Scott [48] and Sandewall [47]. Scott, for instance, condsidered the idea of partially ordering statements according to their truth or falsity, where the truth values assigned to these statements come from a lattice L with an ordering relation $\leq_t$. In fact, the classical two valued logic itself can be considered to be based on a two element set of truth values, namely, $\{\texttt{true}, \texttt{false}\}$, with the ordering $\leq_t$, where $\texttt{false} \leq_t \texttt{true}$.

In his approach, Sandewall suggested ordering the truth values, not based on truth or falsity, but rather based on the amount of information or knowledge they represent. Specifically, he suggested allowing truth values to be subsets of the unit interval $[0, 1]$ indicating that the probability that a given statement is true, is known to fall within the associated interval.

Ginsberg [23, 24] pointed out that the partial order given to truth values based on degrees of knowledge is conceptually different from the one based on the degrees of truth. In the case of the knowledge-based ordering on the set of truth values (denoting the ordering relation by $\leq_k$), one can informally interpret $p \leq_k q$ to mean that more is known about a statement with a truth value q than one with a truth

value p. Accordingly, Ginsberg introduced and algebraic structure called a *bilattice*, in order to combine the two types of orderings on a set of truth values.

4.1 Definitions and Motivation

Bilattices were considered as the basis for a family of multi-valued logics with certain desirable algebraic properties suitable for combining the notions of truth and knowledge. A bilattice is a space of generalized truth values with two lattice orderings, one measuring degrees of truth, and the other measuring degrees of knowledge. The relationships between the two separate orderings could be captured in several ways. For example, as we will see below, a negation operator can provide a strong connection between the two orderings. We will now discuss these notions more formally. The reader is cautioned that the definitions presented below are similar but not identical to those introduced elsewhere in the literature. In particular, our notion of a bilattice always includes a negation operator which, in the context of logics based on bilattices, has the properties of classical negation.

Let $\langle \mathcal{B}, \leq_t, \leq_k \rangle$ be a structure consisting of a nonempty set $\mathcal{B}$ and two partial orderings, $\leq_t$ and $\leq_k$ on $\mathcal{B}$. If $\leq_t$ is a lattice ordering, let `true` and `false` denote the top and bottom elements, $\wedge$ and $\vee$ the meet and join, and $\bigwedge$ and $\bigvee$ the infinitary meet and join (if they exist). Similarly, if $\leq_k$ is a lattice ordering, the corresponding notions are denoted respectively by $\top$, $\bot$, $\otimes$, $\oplus$, $\prod$, and $\sum$. Note that $\wedge$ and $\vee$ are monotonic with respect to $\leq_t$ and $\otimes$ and $\oplus$ are monotonic with respect to $\leq_k$.

Definition 4.1 A *pre-bilattice* is a structure $\langle \mathcal{B}, \leq_t, \leq_k \rangle$ consisting of a nonempty set $\mathcal{B}$ and partial orderings $\leq_t$ and $\leq_k$, each of which gives $\mathcal{B}$ the structure of a

complete lattice. Moreover, $\otimes$ and $\oplus$ are both monotonic with respect to $\leq_t$, and $\wedge$ and $\vee$ are both monotonic with respect to $\leq_k$, that is, for all $x, y, z \in \mathcal{B}$,

1. $x \leq_t y \Longrightarrow x \oplus z \leq_t y \oplus z$ and $x \otimes z \leq_t y \otimes z$;

2. $x \leq_k y \Longrightarrow x \vee z \leq_k y \vee z$ and $x \wedge z \leq_k y \wedge z$.

The monotonicity conditions in the definition of a pre-bilattice are called the *interlacing conditions* [18]. For a pre-bilattice, the interlacing conditions provide the connection between the two orderings. This connection can be made stronger by adding negation.

Definition 4.2 A *bilattice* is a structure $\langle \mathcal{B}, \leq_t, \leq_k, \neg \rangle$ consisting of a non-empty set $\mathcal{B}$, partial orderings $\leq_t$ and $\leq_k$, and a mapping $\neg : \mathcal{B} \rightarrow \mathcal{B}$, such that:

1. $\langle \mathcal{B}, \leq_t, \leq_k \rangle$ is a pre-bilattice;

2. $x \leq_t y \Longrightarrow \neg y \leq_t \neg x$, for all $x, y \in \mathcal{B}$;

3. $x \leq_k y \Longrightarrow \neg x \leq_k \neg y$, for all $x, y \in \mathcal{B}$;

4. $\neg\neg x = x$, for all $x \in \mathcal{B}$.

In the above definitions, $\leq_t$ represents the truth ordering and $\leq_k$ the knowledge ordering. Informally, we interpret $p \leq_k q$ to mean that the evidence underlying an assignment of the truth value p is subsumed by the evidence underlying an assignment of q. The lattice operations for the $\leq_t$ ordering are natural generalizations of the familiar classical ones. Note that $\neg$ reverses the $\leq_t$-ordering, like classical negation, but preserves the $\leq_k$-ordering. Thus it is an automorphism of the lattice $\langle \mathcal{B}, \leq_k \rangle$.

Furthermore, De Morgan laws hold for $\vee$ and $\wedge$, while $\oplus$ and $\otimes$ are self-dual under negation. The last condition means that if a and b are elements of the bilattice, then $\neg(a \oplus b) = \neg a \oplus \neg b$ and $\neg(a \otimes b) = \neg a \otimes \neg b$.

Belnap's four-valued logic [3] is an example of a logic based on the simplest non-trivial bilattice. It will serve as the basis and the setting for the logic presented in the next section. The underlying bilattice for this logic, called $\mathcal{FOUR}$, consists of the four truth values `true`, `false`, $\bot$, $\top$. The bilattice $\mathcal{FOUR}$ is depicted in Figure 4.1. In the $\leq_t$ ordering, if $\wedge, \vee$, and $\neg$ are restricted to the two classical truth values `true` and `false`, then they will behave according to the usual two-valued truth table semantics. If they are restricted to the truth values `false`, `true`, and $\bot$, then the behavior is that of Kleene's strong three-valued logic [14, 27]. In the $\leq_k$ ordering, $\otimes$ represents the *consensus* operator which takes the most information consistent with the two arguments. For example `true` $\otimes$ `false` $= \bot$. Similarly, $\oplus$ represents the *accept everything* operator. For instance, `false` $\oplus$ `true` $= \top$.

Reiter's default logic [44] can also be represented as a logic based on a pre-bilattice. In addition to the truth values `true`, `false`, $\top$, and $\bot$, we now have three new truth values. These are *dt* (true by default), *df* (false by default), and *d* (which is assigned to statements that are both true and false by default). The pre-bilattice for the default logic is depicted in Figure 4.2.

Note that the truth values `true`, `false`, $\top$, and $\bot$ are present in both $\mathcal{FOUR}$ and the pre-bilattice of the default logic. In fact, any pre-bilattice will share these same distinguished elements as they represent the maximal and minimal elements in each of the orderings. In general we have the following result.

Theorem 4.3 *Let $\mathcal{B}$ be a pre-bilattice. Then*

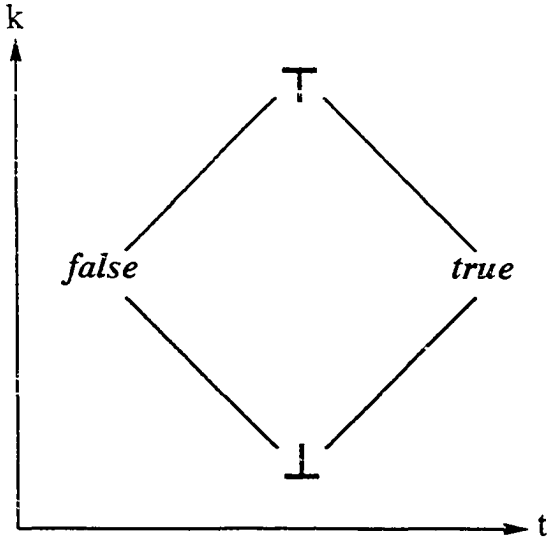


Figure 4.1: The bilattice $\mathcal{FOUR}$

1. $\texttt{true} \ominus \texttt{false} = \top$ *and* $\texttt{true} \otimes \texttt{false} = \bot$.

2. $\top \vee \bot = \texttt{true}$ *and* $\top \wedge \bot = \texttt{false}$.

Proof: Since $\top$ is the largest element of $\mathcal{B}$ under the $\leq_k$-ordering, $a \ominus \top = \top$, for all $a \in \mathcal{B}$. Now, since $\mathcal{B}$ is a pre-bilattice and since $\texttt{false} \leq_t \top$, we have $a \ominus \texttt{false} \leq_t a \ominus \top$. Hence,

$$\texttt{true} \ominus \texttt{false} \leq_t \texttt{true} \ominus \top = \top.$$

Similarly, since true is the largest element under the $\leq_t$-ordering, $\top \leq_t \texttt{true}$, and thus, for any $a \in \mathcal{B}$, $a \ominus \top \leq_t a \ominus \texttt{true}$. Now,

$$\top = \texttt{false} \ominus \top \leq_t \texttt{false} \ominus \texttt{true}.$$

The other parts of the theorem are proved using similar arguments. ∎

For another example consider the pre-bilattice $\mathcal{SIX}$ depicted in Figure 4.3. This

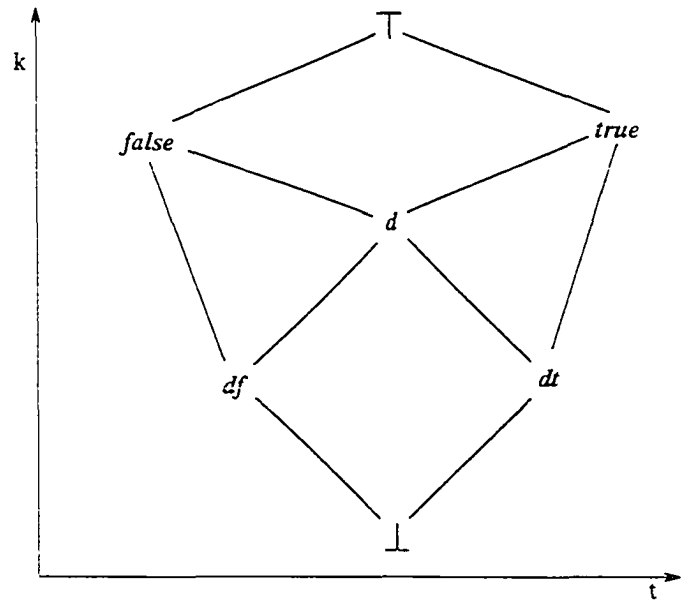


Figure 4.2: The pre-bilattice for default logic

is an example of a pre-bilattice which does not have a negation operator as requireded in the definition of a bilattice. However, one may note that, aside from the last condition of the bilattice definition (i.e., that $\neg\neg x = x$, for all x), the other three conditions hold in $\mathcal{SIX}$ for a negation operator defined by $\neg b = \top$, $\neg a = \bot$, $\neg\texttt{false} = \texttt{true}$, $\neg\texttt{true} = \texttt{false}$, $\neg\top = \top$, and $\neg\bot = \bot$. This provides a motivation for the following definition.

Definition 4.4 A *pre-bilattice with weak negation* is a structure $\langle \mathcal{B}, \leq_t, \leq_k, \neg \rangle$ consisting of a non-empty set $\mathcal{B}$, partial orderings $\leq_t$ and $\leq_k$, and a mapping $\neg : \mathcal{B} \rightarrow \mathcal{B}$, such that:

1. $\langle \mathcal{B}, \leq_t, \leq_k \rangle$ is a pre-bilattice;

2. $x \leq_t y$ implies $\neg y \leq_t \neg x$, for all $x, y \in \mathcal{B}$;

3. $x \leq_k y$ implies $\neg x \leq_k \neg y$, for all $x, y \in \mathcal{B}$.

A stronger type of negation operator may exist in some pre-bilattices, which is stronger than the weak negation, but not quite as strong as the classical one. This is sometimes referred to as the *intuitionistic negation* [19].

Definition 4.5 A pre-bilattice $\mathcal{B}$ has *intuitionistic negation*, if:

1. $\mathcal{B}$ is a pre-bilattice with a weak negation $\neg$;

2. $\neg\neg x \leq_t x$, for all $x \in \mathcal{B}$.

It is easy to verify that $\mathcal{SIX}$ does not have intuitionistic negation, however, there are many interesting examples of pre-bilattices which, while not having classical negation, do satisfy the requirements for an intuitionistic negation operator. We will look at one such example in the following section.

There are many other interesting non-classical logics that can be represented using bilattices or pre-bilattices. Examples include probabilistic logics [57], Kripke's intuitionistic logic model [19], and modal logics based on the many-worlds semantics [24]. For a more detailed discussion see [19, 24, 1, 12].

The special class of distributive pre-bilattices are particularly important in developing semantics for logic programming. There are twelve distributive laws associated with the four operations $\wedge$, $\vee$, $\ominus$, and $\otimes$, for example:

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

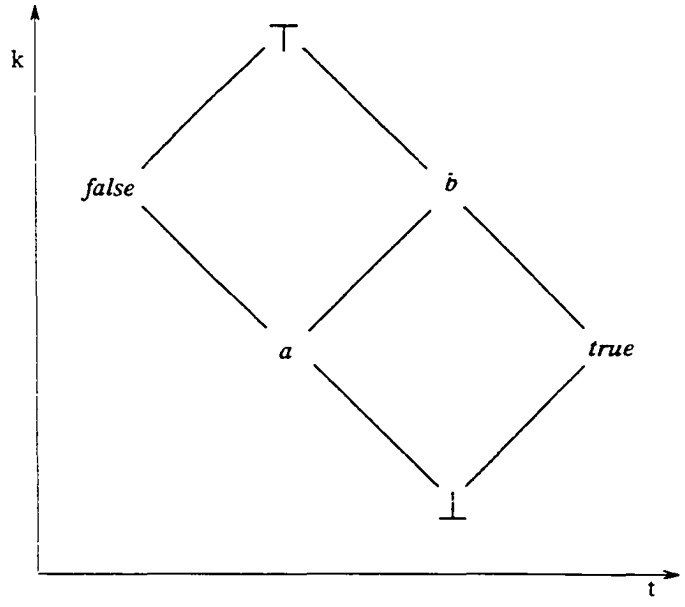


Figure 4.3: The pre-bilattice $\mathcal{SIX}$

Definition 4.6 A pre-bilattice is *distributive* if all twelve distributivity laws hold. A pre-bilattice satisfies the *infinite distributivity condition* if all of the infinitary distributive laws such as $a \otimes \bigvee_i b_i = \bigvee_i (a \otimes b_i)$ and $a \wedge \prod_i b_i = \prod_i (a \wedge b_i)$ hold.

Distributivity by itself is a stronger condition than the interlacing conditions in the definition of pre-bilattice. In fact, distributivity implies the interlacing conditions. Suppose that $\mathcal{B}$ is a structure identical to a pre-bilattice except without the interlacing conditions, but that all twelve distributivity laws hold for $\mathcal{B}$. Then, for instance, we can argue that

$$a \leq_k b \Rightarrow a \wedge c \leq_k b \wedge c,$$

as follows:

$$a \leq_k b \quad \Leftrightarrow \quad a \ominus b = b$$

$$\Rightarrow \quad (a \ominus b) \wedge c = b \wedge c$$

$$\Leftrightarrow \quad (a \wedge c) \ominus (b \wedge c) = b \wedge c \quad \text{(by distributivity)}$$

$$\Leftrightarrow \quad a \wedge c \leq_k b \wedge c.$$

There are many natural examples distributive pre-bilattices. For instance, the bilattice $\mathcal{FOUR}$ is distributive. In our discussion these bilattices will serve as the basis for the logic programming semantics. Ginsberg [24] showed that every distributive pre-bilattice can be represented as a special kind of direct product of two lattices. We use this representation to characterize the join-irreducible elements of a distributive bilattice. One of our main results is that, by imposing certain finiteness conditions on distributive bilattices, we can restrict our attention to the join-irreducible elements in the bilattice without losing any power in the procedural semantics. Let us now make these notions more precise.

4.2 Construction and Representation

As mentioned above, we are particularly interested in the class of distributive bilattices since they have some useful algebraic properties. The results in this section will provide a methodology for construction and representation of pre-bilattices based on the underlying lattice structures.

Definition 4.7 Let $\langle L_1, \leq_1 \rangle$ and $\langle L_2, \leq_2 \rangle$ be two lattices. Define $\leq_t$ and $\leq_k$ on $L_1 \times L_2$ by:

1. $\langle x_1, x_2 \rangle \leq_t \langle y_1, y_2 \rangle$ if $x_1 \leq_1 y_1$ and $y_2 \leq_2 x_2$,

2. $\langle x_1, x_2 \rangle \leq_k \langle y_1, y_2 \rangle$ if $x_1 \leq_1 y_1$ and $x_2 \leq_2 y_2$.

The structure $\langle L_1 \times L_2, \leq_t, \leq_k \rangle$ is denoted by $\mathcal{B}(L_1, L_2)$. If $L = L_1 = L_2$, define a mapping $\neg : L \times L \rightarrow L \times L$ by $\neg\langle x, y\rangle = \langle y, x\rangle$. In this case $\mathcal{B}(L)$ will denote the structure $\langle L \times L, \leq_t, \leq_k, \neg\rangle$.

The following theorem is essentially due to Ginsberg [24] and provides a method for construction of bilattices.

Theorem 4.8 *Let L_1 and L_2 be complete lattices. Then $\mathcal{B}(L_1, L_2)$ is a pre-bilattice and is distributive if L_1 and L_2 are distributive. Furthermore, if $L = L_1 = L_2$, then $\mathcal{B}(L)$ is a bilattice.*

Proof: We need to show that each of the orderings ($\leq_t$ and $\leq_k$) gives $L_1 \times L_2$ the structure of a complete lattice, and that the interlacing conditions hold. It is straightforward to verify that the completeness conditions hold. We will verify that the interlacing and distributivity conditions are satisfied (we denote the ordering in both lattices by $\leq$, since the meaning will be clear from context, furthermore, we will denote the join and meet operations in both lattices by $+$ and $\cdot$, respectively). Let $c_1, c_2 \in L_1$ and $d_1, d_2 \in L_2$. It is easy to check that the following identities hold.

$$\begin{aligned}
\langle c_1, d_1\rangle \otimes \langle c_2, d_2\rangle &= \langle c_1 \cdot c_2, d_1 \cdot d_2\rangle \\
\langle c_1, d_1\rangle \oplus \langle c_2, d_2\rangle &= \langle c_1 + c_2, d_1 + d_2\rangle \\
\langle c_1, d_1\rangle \wedge \langle c_2, d_2\rangle &= \langle c_1 \cdot c_2, d_1 + d_2\rangle \\
\langle c_1, d_1\rangle \vee \langle c_2, d_2\rangle &= \langle c_1 + c_2, d_1 \cdot d_2\rangle.
\end{aligned}$$

We will verify that $\langle c_1, d_1\rangle \wedge \langle c_2, d_2\rangle = \langle c_1 \cdot c_2, d_1 + d_2\rangle$. The other identities can be proved similarly. By the properties of meet and join, it is clear that $c_1 \cdot c_2 \leq c_1, c_2$

and $d_1, d_2 \leq d_1 + d_2$, and hence

$$\langle c_1 \cdot c_2, d_1 + d_2 \rangle \leq_t \langle c_1, d_1 \rangle, \langle c_2, d_2 \rangle.$$

Furthermore, let $\langle c_3, d_3 \rangle \leq_t \langle c_1, d_1 \rangle, \langle c_2, d_2 \rangle$. Clearly $c_3 \leq c_1 \cdot c_2$ and $d_1 + d_2 \leq d_3$. Thus $\langle c_1 \cdot c_2, d_1 + d_2 \rangle$ is the greatest lower bound of $\langle c_1, d_1 \rangle$ and $\langle c_2, d_2 \rangle$ in the $\leq_t$-ordering, proving the above identity.

Now, let $x = \langle c_1, d_1 \rangle, y = \langle c_2, d_2 \rangle$, and $z = \langle c_3, d_3 \rangle$. Assume that $x \leq_t y$. Hence, $\langle c_1, d_1 \rangle \leq_t \langle c_2, d_2 \rangle$ which by the definition implies that $c_1 \leq c_2$ and $d_2 \leq d_1$. Thus,

$$\begin{aligned} x \otimes z &= \langle c_1, d_1 \rangle \otimes \langle c_3, d_3 \rangle \\ &= \langle c_1 \cdot c_3, d_1 \cdot d_3 \rangle \\ &\leq_t \langle c_2 \cdot c_3, d_2 \cdot d_3 \rangle \\ &= \langle c_2, d_2 \rangle \otimes \langle c_3, d_3 \rangle \\ &= y \otimes z. \end{aligned}$$

The case with $\ominus$ is shown similarly.

For the second interlacing condition, suppose that $x \leq_k y$. Hence, $\langle c_1, d_1 \rangle \leq_k \langle c_2, d_2 \rangle$ which by the definition implies that $c_1 \leq c_2$ and $d_1 \leq d_2$. Thus,

$$\begin{aligned} x \wedge z &= \langle c_1, d_1 \rangle \wedge \langle c_3, d_3 \rangle \\ &= \langle c_1 \cdot c_3, d_1 + d_3 \rangle \\ &\leq_k \langle c_2 \cdot c_3, d_2 + d_3 \rangle \\ &= \langle c_2, d_2 \rangle \wedge \langle c_3, d_3 \rangle \\ &= y \wedge z. \end{aligned}$$

The case with $\vee$ is shown similarly.

It is easy to verify that the distributivity conditions hold. We will prove the following case: the other cases are proved similarly.

$$\begin{aligned} a \vee (b \oslash c) &= \langle a_1, a_2 \rangle \vee (\langle b_1, b_2 \rangle \oslash \langle c_1, c_2 \rangle) \\ &= \langle a_1, a_2 \rangle \vee \langle b_1 \cdot c_1, b_2 \cdot c_2 \rangle \\ &= \langle a_1 + (b_1 \cdot c_1), a_2 \cdot (b_2 \cdot c_2) \rangle \\ &= \langle (a_1 + b_1) \cdot (a_1 + c_1), (a_2 \cdot b_2) \cdot (a_2 \cdot c_2) \rangle \\ &= \langle a_1 + b_1, a_2 \cdot b_2 \rangle \otimes \langle a_1 + c_1, a_2 \cdot c_2 \rangle \\ &= (a \vee b) \otimes (a \vee c). \end{aligned}$$

It remains to show that if $L = L_1 = L_2$ then $\mathcal{B}(L)$ is a bilattice. Let $\neg : \mathcal{B}(L) \rightarrow \mathcal{B}(L)$ be defined by $\neg\langle c, d \rangle = \langle d, c \rangle$. We will only check the condition that if $x \leq_t y$, then $\neg y \leq_t \neg x$. The other conditions are proved similarly. Suppose that $\langle c_1, d_1 \rangle \leq_t \langle c_2, d_2 \rangle$. Then, $c_1 \leq c_2$ and $d_2 \leq d_1$. Clearly, $\langle d_2, c_2 \rangle \leq_t \langle d_1, c_1 \rangle$. Hence, $\neg y \leq_t \neg x$. ∎

The next lemma [18] will be used in the following representation theorem.

Lemma 4.9 *Let $\mathcal{B}$ be a distributive pre-bilattice, then for every $x \in \mathcal{B}$,*

$$x = (x \wedge \bot) \ominus (x \vee \bot)$$

Proof: We use the distributivity and the absorption laws for lattices.

$$\begin{aligned} (x \wedge \bot) \ominus (x \vee \bot) &= [x \ominus (x \vee \bot)] \wedge [\bot \ominus (x \vee \bot)] \\ &= [(x \ominus x) \vee (x \ominus \bot)] \wedge [(\bot \ominus x) \vee [\bot \ominus \bot)] \\ &= (x \vee x) \wedge (x \vee \bot) \end{aligned}$$

$$= x \wedge (x \vee \perp)$$

$$= x. \blacksquare$$

The following result [24, 18] suggests a representation method for distributive pre-bilattices.

Theorem 4.10 *Let $\mathcal{B}$ be a distributive pre-bilattice. Then there exist complete distributive lattices L_1 and L_2 such that $\mathcal{B}$ is isomorphic to $\mathcal{B}(L_1, L_2)$.*

Proof: Let $L_1 = \{x \vee \perp \mid x \in \mathcal{B}\}$ with an ordering $\leq_1$ defined by $x \leq_1 y$ if and only if $x \leq_t y$. Also, let $L_2 = \{x \wedge \perp \mid x \in \mathcal{B}\}$ with the ordering $\leq_2$ defined by: $x \leq_2 y$ if and only if $y \leq_t x$. Using the distributivity laws it is easy to verify that L_1 and L_2 are closed under $\wedge$, $\vee$, $\oplus$, and $\otimes$. Thus, L_1 is a sublattice of $\langle \mathcal{B}, \leq_t \rangle$ and L_2 is a sublattice of $\langle \mathcal{B}, \leq_t \rangle$. The completeness of L_1 and L_2 follow immediately. Furthermore, the distributivity of L_1 and L_2 follow from the distributivity of $\mathcal{B}$.

Next, we must show that there exists an order-preserving bijection from $\mathcal{B}$ to $\mathcal{B}(L_1, L_2)$. Define a mapping $\phi : \mathcal{B} \rightarrow \mathcal{B}(L_1, L_2)$, defined by:

$$\phi(x) = \langle x \vee \perp, x \wedge \perp \rangle.$$

The fact that ϕ is one-to-one is an immediate consequence of Lemma 4.9. To show that ϕ is onto, suppose that $\langle a \vee \perp, b \wedge \perp \rangle \in \mathcal{B}(L_1, L_2)$, and let $c = (b \wedge \perp) \oplus (a \vee \perp)$. By definition of ϕ, $\phi(c) = \langle c \vee \perp, c \wedge \perp \rangle$. Consider the first component of this pair:

$$\begin{aligned} c \vee \perp &= [(b \wedge \perp) \oplus (a \vee \perp)] \vee \perp \\ &= [(b \wedge \perp) \vee \perp] \oplus (a \vee \perp \vee \perp) \\ &= \perp \oplus (a \vee \perp) \\ &= a \vee \perp. \end{aligned}$$

Similarly, we can show that $c \wedge \bot = b \wedge \bot$. Hence, $\phi(c) = \langle a \vee \bot, b \wedge \bot \rangle$.

Next we would like to show that for $x, y \in L_1$, $x \leq_t y$ if and only if $x \leq_k y$, and that for $x, y \in L_2$, $x \leq_t y$ if and only if $y \leq_k x$. Since the proofs are similar, we show the second equivalence. Let $x = a \wedge \bot$ and $y = b \wedge \bot$, and suppose that $a \wedge \bot \leq_t b \wedge \bot$. Then $a \wedge \bot = a \wedge b \wedge \bot$. But, $\bot \leq_k a \wedge \bot$ and so we have:

$$b \wedge \bot \leq_k b \wedge a \wedge \bot = a \wedge \bot.$$

Now, suppose that $b \wedge \bot \leq_k a \wedge \bot$. In this case, $a \wedge b \wedge \bot \leq_k a \wedge \bot$. Since, $\bot \leq_k b \wedge \bot$, we can conclude that $a \wedge \bot \leq_k a \wedge b \wedge \bot$. Hence, we have $a \wedge \bot = a \wedge b \wedge \bot$ and hence $a \wedge \bot \leq_t b \wedge \bot$.

Finally, to show that ϕ is order-preserving, suppose that $a \leq_k b$. Then $a \wedge \bot \leq_k b \vee \bot$ and hence, $b \wedge \bot \leq_t a \wedge \bot$. Therefore, $a \wedge \bot \leq_2 b \wedge \bot$. Also, $a \vee \bot \leq_k b \vee \bot$, and so $a \vee \bot \leq_t b \wedge \bot$. Hence, $a \vee \bot \leq_1 b \vee \bot$. Thus, $\phi(a) \leq_k \phi(b)$. Conversely, suppose that $\phi(a) \leq_k \phi(b)$. Then $a \vee \bot \leq_k b \vee \bot$ and $a \wedge \bot \leq_k b \wedge \bot$. Therefore, by Lemma 4.9, we have:

$$a = (a \wedge \bot) \odot (a \vee \bot) \leq_k (b \wedge \bot) \oplus (b \vee \bot) = b.$$

Hence, we conclude that $a \leq_k b$ if and only if $\phi(a) \leq_k \phi(b)$. In a similar manner, we can show that $a \leq_t b$ if and only if $\phi(a) \leq_t \phi(b)$. ∎

One can think of the components x and y of a pair $\langle x, y \rangle$ as summarizing the evidence for and the evidence against an assertion, respectively. In general, the pair $\langle x, y \rangle$ is codifying two independent judgments regarding the truth or falsity of some statement. Since the corresponding lattices need not be the same, these judgments can be measured in different ways.

Belnap's four-valued logic, for instance, can be represented by the above construction if one takes both L_1 and L_2 to be the 2-element lattice $\{0,1\}$. The probabilistic bilattice can formed by taking, for both lattices, the interval [0,1]. In the latter logic, each truth value can represent the degrees of belief and doubt.

As shown by Ginsberg, the notion of Kripke models, for the possible worlds semantics of modal logics, can also be obtained using the above construction. One can think of the set of possible worlds in which a formula is true as the evidence for the statement represented by the formula, and the set of possible worlds in which the formula is false, as evidence against it. In this case, given the Kripke model with W as the set of possible worlds, we can use the power set lattice $P = \langle \mathcal{P}(W), \subseteq \rangle$ to construct the bilattice $\mathcal{B}(P,P)$. A truth value relative to this Kripke model would be a pair $\langle F, A \rangle$, where $F, A \subseteq W$. Based on the above construction, it is easy to see that $\langle F_1, A_1 \rangle \leq_t \langle F_2, A_2 \rangle$ if $F_1 \subseteq F_2$ and $A_2 \subseteq A_1$. Similarly, $\langle F_1, A_1 \rangle \leq_k \langle F_2, A_2 \rangle$ if $F_1 \subseteq F_2$ and $A_1 \subseteq A_2$. Furthermore, in the resulting bilattice we have $\texttt{false} = \langle \emptyset, W \rangle$, that is, no evidence for but total evidence against. Analogously, $\texttt{true} = \langle W, \emptyset \rangle$, $\top = \langle W, W \rangle$, and $\bot = \langle \emptyset, \emptyset \rangle$. The negation is defined by $\neg(\langle F, A \rangle) = \langle A, F \rangle$.

For another example, consider the probabilistic logic obtained by using as the underlying lattice, the interval $[0,1]$ of reals with the usual ordering $\leq$ of reals. The resulting bilattice, then would be the structure $\mathcal{B}([0,1],[0,1])$. A probabilistic logic based on the lattice $[0,1]$ was considered by van Emden in [57]. Logics based on $\mathcal{B}([0,1],[0,1])$, however, have the additional machinary to naturally accommodate conflicting probabilistic information.

An interesting family of pre-bilattices can be obtained from the topological spaces arising from the Kripke intuitionistic logic models [16]. Let T be a topological space.

The family O of open sets is a complete lattice under $\subseteq$. In this case, join will be the union, and meet will be the interior of the intersection. Also, the family C of closed sets is a complete lattice under $\subseteq$. The resulting bilattice will be obtained by considering the structure $\mathcal{B}(O, C)$. Note that this is an example of a pre-bilattice in which the two underlying lattices are different, and hence it does not have classical negation. However, for such a bilattice, if we take $\neg\langle o, c\rangle$, where o is an open set and c is a closed set, to be $\langle \mathit{interior}(c), \mathit{closure}(o)\rangle$, then we can get an intuitionistic negation.

As a final example, note that the pre-bilattice $\mathcal{SIX}$ described above (see Figure 4.3), can be represented by the structure $\mathcal{B}(L, K)$, where L is the lattice $\{0, 1\}$ and K is the lattice $\{0, 1, 2\}$, each with the usual ordering. In this case the truth value a in Figure 4.3 would be represented by the pair $\langle 0, 1\rangle$, b would be represented by $\langle 1, 1\rangle$. Again, $\mathcal{SIX}$ does not have classical negation, but it does have a weak negation operator defined earlier. Incidentally, according to the above representation theorem, this construction verifies that $\mathcal{SIX}$ is in fact a distributive pre-bilattice.

Before leaving this section, we need also consider another way to construct pre-bilattices, namely the one formed by taking the set of all maps from a set to a pre-bilattice.

Definition 4.11 Suppose $\mathcal{B}$ is a pre-bilattice and S is any set. Let $\mathcal{B}^{\langle S\rangle}$ denote the set of all mappings from S to $\mathcal{B}$, with the induced orderings $\leq_k$ and $\leq_t$ defined coordinatewise on $\mathcal{B}^{\langle S\rangle}$. In other words, for $f, g \in \mathcal{B}^{\langle S\rangle}$, $f \leq_k g$, if $f(a) \leq_k g(a)$, for all $a \in S$, and similarly for $\leq_t$. Furthermore, if $\mathcal{B}$ is a bilattice (has the negation operator), then a negation operator is correspondingly induced on $\mathcal{B}^{\langle S\rangle}$ by $(\neg f)(a) = \neg f(a)$.

Theorem 4.12 ([19]) *Let $\mathcal{B}$ be a pre-bilattice and let S be any set. Then $\mathcal{B}^{\langle S \rangle}$ is a pre-bilattice. In addition, if $\mathcal{B}$ is a bilattice, then $\mathcal{B}^{\langle S \rangle}$ is a bilattice.*

In the above theorem, the operations on $\mathcal{B}^{\langle S \rangle}$ are determined in a coordinatewise manner. For example, $(f \otimes g)(a) = f(a) \otimes g(a)$. The importance of this kind of construction will become clear when we discuss the fixpoint semantics of bilattice-based logic programs. Specifically, the space of interpretations mapping formulas to truth values (elements of a bilattice), will itself become a bilattice.

4.3 Bilattices and Join-Irreducible Elements

In this section, we extend the concept of join-irreducible elements of lattices to the realm of bilattices. This will allow us, under certain conditions, to concentrate on a small representative set of bilattice elements with a rich set of algebraic properties. Based on these notions, we will introduce an alternative procedural semantics for logic programs in Chapter 7.

Definition 4.13 Let $\mathcal{B} = \langle \mathcal{B}, \leq_t, \leq_k \rangle$ be a pre-bilattice, and let $a \in \mathcal{B}$. We say that a is *k-join-irreducible* (*t-join-irreducible*) if a is join-irreducible in the $\leq_k$-ordering ($\leq_t$-ordering). The set of all k-join-irreducible elements of $\mathcal{B}$ is denoted by $JIR_k(\mathcal{B})$, and the set of all t-join-irreducible elements of $\mathcal{B}$ is denoted by $JIR_t(\mathcal{B})$.

Since in a bilattice $\neg$ is an automorphism of the knowledge lattice, $\neg b$ will be k-join-irreducible whenever b is k-join-irreducible.

The following lemma provides a characterization of the k-join-irreducible elements of a bilattice of the form $\mathcal{B}(L_1, L_2)$ in terms of the join-irreducible elements of

the underlying lattice structures. The bottom elements of L_1 and L_2 will be denoted by $\bot_1$ and $\bot_2$, but we will drop the subscripts when they are clear from context.

Lemma 4.14 *Let* $\mathcal{B} = \mathcal{B}(L_1, L_2) = \langle L_1 \times L_2, \leq_t, \leq_k \rangle$ *be a pre-bilattice, where* $\langle L_i, \leq_i, +, \cdot \rangle$ *is a complete lattice, for* $i = 1, 2$. *Then*

$$JIR_k(\mathcal{B}) = \{\langle a_1, \bot \rangle, \langle \bot, a_2 \rangle \mid a_i \in JIR(L_i), i = 1, 2\}.$$

Proof: Let $a = \langle a_1, a_2 \rangle \in \mathcal{B}$. Suppose that $a_1 \neq \bot$ and $a_2 \neq \bot$. Since

$$\langle a_1, \bot \rangle \oplus \langle \bot, a_2 \rangle = \langle a_1 + \bot, \bot + a_2 \rangle = \langle a_1, a_2 \rangle = a.$$

we have $a \notin JIR_k(\mathcal{B})$. Conversely, let $\langle a_1, \bot \rangle \in \mathcal{B}$, such that $a_1 \in JIR(L_1)$, and suppose that $\langle a_1, \bot \rangle = \langle b_1, b_2 \rangle \oplus \langle c_1, c_2 \rangle$. Then $a_1 = b_1 + c_1$ and $\bot = b_2 + c_2$. Clearly, $b_2 = c_2 = \bot$. Since $a_1 \in JIR(L_1)$, either $a_1 = b_1$ or $a_1 = c_1$. In either case, $\langle a_1, \bot \rangle \in JIR_k(\mathcal{B})$.

Using a similar argument, we can conclude that $\langle \bot, a_2 \rangle \in JIR_k(\mathcal{B})$. ∎

This allows us to a classify the join-irreducible elements in the knowledge ordering. As we shall see later, the following classification will shed light on the behavior of bilattice-based logic programs.

Definition 4.15 Let $\mathcal{B} = \mathcal{B}(L_1, L_2)$ be a pre-bilattice. An element $c \in \mathcal{B}$ is a *positive k-join-irreducible* element if $c \in JIR_k(\mathcal{B})$ and $c = \langle c_1, \bot \rangle$, where $c_1 \in JIR(L_1)$. c is a *negative k-join-irreducible* element if $c \in JIR_k(\mathcal{B})$ and $c = \langle \bot, c_2 \rangle$, where $c_2 \in JIR(L_2)$. The set of all positive k-join-irreducible elements of $\mathcal{B}$ is denoted by $JIR_k^+(\mathcal{B})$, and the set of all negative k-join-irreducible elements of $\mathcal{B}$ is denoted by $JIR_k^-(\mathcal{B})$.

Clearly, by the previous lemma, we have:

$$JIR_k(\mathcal{B}) = JIR_k^+(\mathcal{B}) \cup JIR_k^-(\mathcal{B}).$$

To illustrate the above concepts, consider the bilattice $\mathcal{NINE}$, depicted in Figure 4.4. This bilattice can be constructed by taking the set $P = \{0, b, 1\}$ with the ordering $0 \leq b \leq 1$, and obtaining the structure $\mathcal{B}(P, P)$. Then

$$JIR_k^+(\mathcal{NINE}) = \{\langle 0,0\rangle, \langle b,0\rangle, \langle 1,0\rangle\},$$

and

$$JIR_k^-(\mathcal{NINE}) = \{\langle 0,0\rangle, \langle 0,b\rangle, \langle 0,1\rangle\}.$$

Note that in this bilattice, $\langle 0,1\rangle$ represents `false` while $\langle 1,0\rangle$ represents `true`. In fact, in any pre-bilattice $\mathcal{B}$, the elements `false` and `true` are among the k-join-irreducible elements (`false` $\in JIR_k^-(\mathcal{B})$ and `true` $\in JIR_k^+(\mathcal{B})$), if the top element in both of the underlying lattices is join-irreducible. Furthermore, note that in $\mathcal{NINE}$ and in fact in any bilattice $\mathcal{B}$, we have:

$$\neg(JIR_k^+(\mathcal{B})) = JIR_k^-(\mathcal{B}) \text{ and } \neg(JIR_k^-(\mathcal{B})) = JIR_k^+(\mathcal{B}).$$

The following lemma provides a basis for the join-irreducible procedural semantics defined in Chapter 7.

Lemma 4.16 *Let $\mathcal{B} = \mathcal{B}(L_1, L_2) = \langle L_1 \times L_2, \leq_1, \leq_k \rangle$ be a distributive pre-bilattice, where $\langle L_i, \leq_i, +, \cdot \rangle$ is a complete distributive lattice, for $i = 1, 2$. Suppose that $a, b \in \mathcal{B}$ and let $c \in JIR_k(\mathcal{B})$. Then:*

1. *If $c \in JIR_k^+(\mathcal{B})$, then $a \vee b \geq_k c$ if and only if $a \geq_k c$ or $b \geq_k c$;*

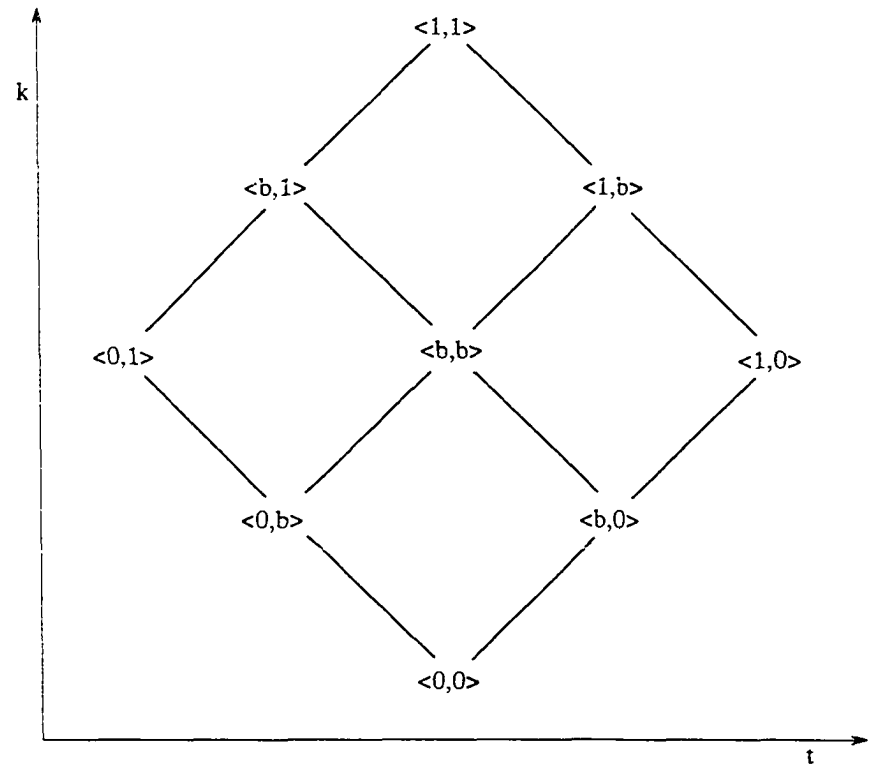


Figure 4.4: The bilattice $\mathcal{NINE}$

2. *If* $c \in JIR_k^+(\mathcal{B})$, *then* $a \wedge b \geq_k c$ *if and only if* $a \geq_k c$ *and* $b \geq_k c$;

3. *If* $c \in JIR_k^-(\mathcal{B})$, *then* $a \vee b \geq_k c$ *if and only if* $a \geq_k c$ *and* $b \geq_k c$;

4. *If* $c \in JIR_k^-(\mathcal{B})$, *then* $a \wedge b \geq_k c$ *if and only if* $a \geq_k c$ *or* $b \geq_k c$.

Proof: Let $a = \langle a_1, a_2 \rangle$ and $b = \langle b_1, b_2 \rangle$, and note that

$$a \vee b = \langle a_1, a_2 \rangle \vee \langle b_1, b_2 \rangle = \langle a_1 + b_1, a_2 \cdot b_2 \rangle, \text{ and } a \wedge b = \langle a_1, a_2 \rangle \wedge \langle b_1, b_2 \rangle = \langle a_1 \cdot b_1, a_2 + b_2 \rangle.$$

1. Suppose that $c = \langle c_1, \bot \rangle$. Now we have:

$$a \vee b \geq_k c \text{ iff } \langle a_1 + b_1, a_2 \cdot b_2 \rangle \geq_k \langle c_1, \bot \rangle$$

iff $a_1 + b_1 \geq_1 c_1$ and $a_2 \cdot b_2 \geq_2 \bot$

iff $a_i + b_i \geq_i c_i$

iff $a_1 \geq_1 c_1$ or $b_1 \geq_1 c_1$ (by Lemma 2.19)

iff $\langle a_1, a_2 \rangle \geq_k \langle c_1, \bot \rangle$ or $\langle b_1, b_2 \rangle \geq_k \langle c_1, \bot \rangle$

iff $a \geq_k c$ or $b \geq_k c$.

2. Suppose that $c = \langle c_1, \bot \rangle$. Now we have:

$a \wedge b \geq_k c$ iff $\langle a_1 \cdot b_1, a_2 + b_2 \rangle \geq_k \langle c_1, \bot \rangle$

iff $a_1 \cdot b_1 \geq_1 c_1$ and $a_2 + b_2 \geq_2 \bot$

iff $a_1 \cdot b_1 \geq_1 c_1$

iff $a_1 \geq_1 c_1$ and $b_1 \geq_1 c_1$

iff $\langle a_1, a_2 \rangle \geq_k \langle c_1, \bot \rangle$ and $\langle b_1, b_2 \rangle \geq_k \langle c_1, \bot \rangle$

iff $a \geq_k c$ and $b \geq_k c$.

3. Suppose that $c = \langle \bot, c_2 \rangle$. Now we have:

$a \vee b \geq_k c$ iff $\langle a_1 + b_1, a_2 \cdot b_2 \rangle \geq_k \langle \bot, c_2 \rangle$

iff $a_1 + b_1 \geq_1 \bot$ and $a_2 \cdot b_2 \geq_2 c_2$

iff $a_2 \cdot b_2 \geq_2 c_2$

iff $a_2 \geq_2 c_2$ and $b_2 \geq_2 c_2$

iff $\langle a_1, a_2 \rangle \geq_k \langle \bot, c_2 \rangle$ and $\langle b_1, b_2 \rangle \geq_k \langle \bot, c_2 \rangle$

iff $a \geq_k c$ and $b \geq_k c$.

4. Suppose that $c = \langle \bot, c_2 \rangle$. Now we have:

$a \wedge b \geq_k c$ iff $\langle a_1 \cdot b_1, a_2 + b_2 \rangle \geq_k \langle \bot, c_2 \rangle$

iff $a_1 \cdot b_1 \geq_1 \bot$ and $a_2 + b_2 \geq_2 c_2$

iff $a_2 + b_2 \geq_2 c_2$

iff $a_2 \geq_2 c_2$ or $b_2 \geq_2 c_2$ by Lemma 2.19)

iff $\langle a_1, a_2 \rangle \geq_k \langle \bot, c_2 \rangle$ or $\langle b_1, b_2 \rangle \geq_k \langle \bot, c_2 \rangle$

iff $a \geq_k c$ or $b \geq_k c$. ■

The above results provide the basis and the motivation behind the join-irreducible procedural semantics presented in subsequent sections. To make to correspondence more clear, we presented the following reformulation of above lemma and the corresponding results for the knowledge operators.

Corollary 4.17 *Let $\mathcal{B} = \langle \mathcal{B}, \leq_t, \leq_k \rangle$ be a distributive pre-bilattice. Suppose that $a, b, c \in \mathcal{B}$.*

1. *If $c \in JIR_k(\mathcal{B})$, then*

   (a) *$c \leq_k a \oplus b$ if and only if $c \leq_k a$ or $c \leq_k b$;*

   (b) *$c \leq_k a \otimes b$ if and only if $c \leq_k a$ and $c \leq_k b$ (this is, in fact, true for any $c \in \mathcal{B}$);*

2. *If $c \in JIR_k^+(\mathcal{B})$, then*

   (a) *$c \leq_k a \vee b$ if and only if $c \leq_k a \oplus b$;*

   (b) *$c \leq_k a \wedge b$ if and only if $c \leq_k a \otimes b$;*

3. *If $c \in JIR_k^-(\mathcal{B})$, then*

   (a) *$c \leq_k a \vee b$ if and only if $c \leq_k a \otimes b$;*

(b) $c \leq_k a \wedge b$ *if and only if* $c \leq_k a \oplus b$.

Proof: The proof is immediate from Lemmas 4.16, 2.19. ∎

We will also find the following lemmas useful in the proofs of the main results in subsequent chapters. The next lemma is simply a reformulation of Lemma 2.19.

Lemma 4.18 *Let* $\mathcal{B} = \langle \mathcal{B}, \leq_t, \leq_k \rangle$ *be a pre-bilattice,* $c_1, c_2, \cdots, c_n \in \mathcal{B}$, *and let* $c \in JIR_k(\mathcal{B})$. *Then* $c_1 \oplus c_2 \oplus \cdots \oplus c_n \geq_k c$ *if and only if* $c_j \geq_k c$ *for some* j $(1 \leq j \leq n)$.

Lemma 4.19 *Let* $\mathcal{B} = \mathcal{B}(L_1, L_2) = \langle L_1 \times L_2, \leq_t, \leq_k \rangle$ *be a pre-bilattice, where* $\langle L_i, \leq_i, +, \cdot \rangle$ *is a complete lattice, for* $i = 1, 2$. *Let* $b, c \in JIR_k(\mathcal{B})$.

1. *If* $c \in JIR_k^+(\mathcal{B})$ *and* $c \geq_k b$, *then* $b \in JIR_k^+(\mathcal{B})$.

2. *If* $c \in JIR_k^-(\mathcal{B})$ *and* $c \geq_k b$, *then* $b \in JIR_k^-(\mathcal{B})$.

Proof: Suppose that $c \in JIR_k^+(\mathcal{B})$ and $c \geq_k b$. Then $c = \langle c_1, \bot \rangle$, where $c_1 \in JIR(L_1)$. Now, $b \in JIR_k(\mathcal{B})$, hence $b = \langle b_1, \bot \rangle$ or $b = \langle \bot, b_2 \rangle$, where $b_i \in JIR(L_i)$, for $i = 1, 2$. But, $b \neq \langle \bot, b_2 \rangle$, since otherwise $c \geq_k b$ implies that $\langle c_1, \bot \rangle \geq_k \langle \bot, b_2 \rangle$, which is impossible because $b \neq \bot$. Hence $b = \langle b_1, \bot \rangle \in JIR_k^+(\mathcal{B})$.

The case when $c \in JIR_k^-(\mathcal{B})$ is proved similarly. ∎

CHAPTER 5. UNIFICATION AND SUBSTITUTION UNIFIERS

Our procedural semantics uses an AND- and OR-parallel interpretation model (see [52, 9, 10]), in which the AND-parallel component is independent. In an independent AND-parallel model, even when subgoals share variables, they are solved independently. After termination, however, answer substitutions obtained independently for shared variables are tested for consistency. We use the notion of substitution unification to ensure the consistency of bindings obtained for shared variables during the computation. In this section we introduce the notions of substitution unification and substitution unifiers, and we prove several results which are useful in proving the completeness results in the subsequent sections. Many of these results may also be of independent interest in unification theory.

We begin by extending the notion of unification to substitutions themselves. The notion of unifiable substitutions has been used in concurrent logic programming systems which use AND-parallelism [26, 38]. These substitutions also play an essential role in our procedural semantics.

Definition 5.1 Let S be a set of substitutions. Then a substitution γ is called a *substitution unifier (s-unifier)* of S, if $S\gamma$ is a singleton. If such a substitution γ exists, then we say that S is *unifiable*. An s-unifier γ of S is a *most general substitution unifier* (mgsu) of S if $\gamma \leq \delta$ for every s-unifier δ of S. We denote the set of all

s-unifiers of S by $su(S)$ and the set of all most general s-unifiers of S by $mgsu(S)$.

That $mgsu(S)$ exists whenever S is unifiable follows from the easily verified fact that $mgsu(S) = mgu(S')$ for some finite set S' of atomic formulas.

Definition 5.2 Let S be a unifiable set of substitutions. A substitution δ is a *substitution unification* of S, if $\delta = \sigma\gamma$, for some $\gamma \in mgsu(S)$ and some $\sigma \in S$. The set of all substitution unifications of S, is denoted by $\odot S$. Clearly, for any $\sigma \in S$:

$$\begin{aligned} \odot S &= \{\sigma\tau \mid \tau \in mgsu(S)\} \\ &= \sigma\, mgsu(S). \end{aligned}$$

When dealing with a pair of substitutions σ and τ, we often use a shorthand notation and denote the set of s-unifiers and the set of mgsu's of σ and τ by $su(\sigma, \tau)$ and $mgsu(\sigma, \tau)$, respectively. Similarly, we denote the set of all substitution unifications of σ and τ by $\sigma \odot \tau$.

It follows from Lemma 3.35 that *mgu*'s and *mgsu*'s are unique up to renaming [27]. We will sometimes slightly abuse the notation and use this fact to treat $mgu(E_1, E_2)$, where E_1 and E_2 are expressions, as a function returning a unique mgu. In other words, we interpret the equality $\gamma = mgu(E_1, E_2)$ to mean equality up to renaming. We'll adopt the same convention for mgsu's. It follows that the substitution unification $\sigma_1 \odot \sigma_2$ of any pair of substitutions σ_1 and σ_2 is also unique up to renaming. In the sequel we sometimes interpret $\sigma_1 \odot \sigma_2$ as a function returning a unique substitution unification of σ_1 and σ_2 up to renaming.

Substitution unifications are useful in parallel evaluation of queries, since they provide a mechanism for ensuring consistency of the bindings obtained concurrently

during the derivation process. Furthermore, they display certain algebraic characteristics which may be of independent interest in unification theory. Here we present some of the properties of substitution unifiers.

Lemma 5.3 *Let θ, η_1, η_2, γ_1, and γ_2 be substitutions. Then:*

1. $su(\eta_1, \eta_2) = su(\gamma_1, \gamma_2)$ *iff* $mgsu(\eta_1, \eta_2) = mgsu(\gamma_1, \gamma_2)$.

2. $su(\eta_1, \eta_2) \subseteq su(\theta\eta_1, \theta\eta_2)$.

3. *If* $dom(\eta_1) \cup dom(\eta_2) \subseteq vrange(\theta)$ *then* $su(\eta_1, \eta_2) = su(\theta\eta_1, \theta\eta_2)$.

Proof: 1 and 2 are easy. We will prove part 3. According to 2 it suffices to show that $su(\theta\eta_1, \theta\eta_2) \subseteq su(\eta_1, \eta_2)$. Let $\gamma \in su(\theta\eta_1, \theta\eta_2)$. We first show that $dom(\eta_1\gamma) = dom(\eta_2\gamma)$. Suppose $x \in dom(\eta_1\gamma) - dom(\eta_2\gamma)$. If $x \notin vrange(\theta)$ then, by assumption, $x \notin dom(\eta_1) \cup dom(\eta_2)$. However, since $x \in dom(\eta_1\gamma)$, we must have $x \in dom(\gamma)$. But this implies that $x \in dom(\eta_2\gamma)$, contradicting the choice of x. By symmetry, $dom(\eta_1\gamma) = dom(\eta_2\gamma)$.

It remains to show that $\eta_1\gamma$ and $\eta_2\gamma$ have the same bindings. Let $x \in dom(\eta_1\gamma) = dom(\eta_2\gamma)$ such that $x/e_1 \in \eta_1\gamma$ and $x/e_2 \in \eta_2\gamma$. We will show that $e_1 = e_2$. Suppose that $e_1 \neq e_2$. If $x \notin vrange(\theta)$ then $x \notin dom(\eta_1) \cup dom(\eta_2)$. This implies that $x/e_1, x/e_2 \in \gamma$ and hence $e_1 = e_2$, contradicting the assumption. If $x \in vrange(\theta)$, then $\theta\eta_1\gamma \neq \theta\eta_2\gamma$, contradicting the choice of γ. Hence, $e_1 = e_2$ and the proof is complete. ∎

Corollary 5.4 *Let θ, η_1, η_2 be substitutions such that $dom(\eta_1), dom(\eta_2) \subseteq vrange(\theta)$. Then,*

$$mgsu(\eta_1, \eta_2) = mgsu(\theta\eta_1, \theta\eta_2).$$

Proof: Follows from parts 1 and 3 of the previous lemma. ∎

Corollary 5.5 *Let θ, η_1, and η_2 be substitutions such that $dom(\eta_i) \subseteq vrange(\theta)$. Then*

$$\theta(\eta_1 \odot \eta_2) = \theta\eta_1 \odot \theta\eta_2.$$

Proof:

$$\begin{aligned} \theta(\eta_1 \odot \eta_2) &= \theta\eta_1\ mgsu(\eta_1, \eta_2) \\ &= \theta\eta_1\ mgsu(\theta\eta_1, \theta\eta_2) \qquad \text{by Corollary 5.4} \\ &= \theta\eta_1 \odot \theta\eta_2. \ \blacksquare \end{aligned}$$

Lemma 5.6 *Let σ_1, σ_2, τ be substitutions. Then*

1. *$\tau\ su(\sigma_1\tau, \sigma_2\tau) \subseteq su(\sigma_1, \sigma_2)$.*
2. *For every $\alpha \in \sigma_1\tau \odot \sigma_2\tau$ and for every $\beta \in \sigma_1 \odot \sigma_2$, $\beta \leq \alpha$.*

Proof: Part 1 is easy. For part 2, first note that by definition, $\alpha = \sigma_1\tau\delta$ and $\beta = \sigma_1\rho$, for some $\delta \in mgsu(\sigma_1\tau, \sigma_2\tau)$ and $\rho \in mgsu(\sigma_1, \sigma_2)$. Then, by part 1, $\tau\delta \in \tau\ (mgsu(\sigma_1\tau, \sigma_2\tau)) \subseteq su(\sigma_1, \sigma_2)$. Hence, $\alpha = \sigma_1\tau\delta = \sigma_1\rho\gamma = \beta\gamma$, for some substitution γ. ∎

Lemma 5.7 (Left-invariance of $\leq$) *Let σ and τ be substitutions. Then*

$$\sigma \leq \tau \Leftrightarrow \textit{ for all } \pi, \pi\sigma \leq \pi\tau.$$

Proof: ($\Leftarrow$): Trivial.

($\Rightarrow$):

$$\begin{aligned}
\sigma \leq \tau &\Leftrightarrow \exists\delta[\sigma\delta = \tau] \\
&\Leftrightarrow \exists\delta\forall\pi[\pi\sigma\delta = \pi\tau] \\
&\Rightarrow \forall\pi\exists\delta[\pi\sigma\delta = \pi\tau] \\
&\Leftrightarrow \forall\pi[\pi\sigma \leq \pi\tau]. \ \blacksquare
\end{aligned}$$

Lemma 5.8 *Let σ and τ be substitutions.*

1. *σ is idempotent $\Rightarrow \sigma(\sigma \odot \tau) = \sigma \odot \tau$.*

2. *τ is idempotent $\Rightarrow \tau(\sigma \odot \tau) = \sigma \odot \tau$.*

Proof: We prove part 1; part 2 is proved in a similar manner. We have:

$$\begin{aligned}
\sigma(\sigma \odot \tau) &= \sigma\sigma(mgsu(\sigma, \tau)) \\
&= \sigma(mgsu(\sigma, \tau)) \quad \text{(since } \sigma \text{ is idempotent)} \\
&= \sigma \odot \tau. \ \blacksquare
\end{aligned}$$

Lemma 5.9 *Let σ, τ, γ be substitutions.*

1. *If σ is idempotent, then $\sigma \leq \gamma \Rightarrow \sigma\gamma = \gamma$.*

2. *If σ and τ are both idempotent, and $\sigma, \tau \leq \gamma$, then σ and τ are unifiable.*

Proof: (1) Let $\gamma = \sigma\delta$. Then $\sigma\gamma = \sigma\sigma\delta = \sigma\delta = \gamma$.

(2) By part (1), $\sigma\gamma = \gamma$ and $\tau\gamma = \gamma$. So, γ is a substitution unifier of σ and τ. ∎

We can now prove the following weak distributivity result for $\odot$.

Lemma 5.10 *Let θ,η_1, and η_2 be substitutions, where η_1 and η_2 are unifiable. Then*

$$\theta\eta_1 \odot \theta\eta_2 \leq \theta(\eta_1 \odot \eta_2).$$

Proof: We know that $\theta\eta_1(mgsu(\eta_1, \eta_2)) = \theta\eta_2(mgsu(\eta_1, \eta_2))$. Hence, $mgsu(\eta_1, \eta_2)$ is a unifier of $\theta\eta_1$ and $\theta\eta_2$ and so:

$$mgsu(\theta\eta_1, \theta\eta_2) \leq mgsu(\eta_1, \eta_2).$$

Now, by Lemma 5.7,

$$\theta\eta_1(mgsu(\theta\eta_1, \theta\eta_2)) \leq \theta\eta_1(mgsu(\eta_1, \eta_2)).$$

Hence,

$$\theta\eta_1 \odot \theta\eta_2 \leq \theta(\eta_1 \odot \eta_2). \blacksquare$$

Lemma 5.11 *Let σ and τ be unifiable substitutions. Then,*

1. *$\sigma \leq \sigma \odot \tau$ and $\tau \leq \sigma \odot \tau$.*

2. *If σ and τ are idempotent, then, for every substitution γ,*

$$\sigma, \tau \leq \gamma \Rightarrow \sigma \odot \tau \leq \gamma.$$

Proof:

1. $\sigma \odot \tau = \sigma(mgsu(\sigma, \tau)) = \tau(mgsu(\sigma, \tau))$.

2. Suppose $\sigma, \tau \leq \gamma$. Then, by Lemma 5.9, γ is a substitution unifier of σ and τ. Hence, $mgsu(\sigma, \tau) \leq \gamma$ and so:

$$\begin{aligned} \sigma \odot \tau &= \sigma(mgsu(\sigma, \tau)) \\ &\leq \sigma\gamma \quad \text{(by Lemma 5.7)} \\ &= \gamma \quad \text{(by Lemma 5.9).} \blacksquare \end{aligned}$$

Lemma 5.12 *Let σ_1, σ_2, η_1, and η_2 be substitutions such that:*

1. *$\sigma_i \leq \eta_i$, for $i = 1, 2$, and*

2. *σ_1 and σ_2 are idempotent, and*

3. *η_1 and η_2 are unifiable.*

Then σ_1 and σ_2 are unifiable and $\sigma_1 \odot \sigma_2 \leq \eta_1 \odot \eta_2$.

Proof: By Lemma 5.11 and assumption 1, we have:

$$\sigma_i \leq \eta_i \leq \eta_1 \odot \eta_2, \quad \text{for } i = 1, 2.$$

Now, by Lemma 5.9 (taking $\gamma = \eta_1 \odot \eta_2$), σ_1 and σ_2 are unifiable. The second part follows immediately from Lemma 5.11. $\blacksquare$

Corollary 5.13 *Let σ_1, σ_2, η_1, η_2, and θ be substitutions such that:*

1. $\sigma_i \leq \theta\eta_i$, *for* $i = 1, 2$, *and*

2. σ_1 *and* σ_2 *are idempotent, and*

3. η_1 *and* η_2 *are unifiable.*

Then σ_1 *and* σ_2 *are unifiable and* $\sigma_1 \odot \sigma_2 \leq \theta(\eta_1 \odot \eta_2)$.

Proof: Since η_1 and η_2 are unifiable, then so are $\theta\eta_1$ and $\theta\eta_2$. Hence, by Lemma 5.12 and Lemma 5.10, σ_1 and σ_2 are unifiable and

$$\sigma_1 \odot \sigma_2 \leq \theta\eta_1 \odot \theta\eta_2 \leq \theta(\eta_1 \odot \eta_2). \blacksquare$$

Lemma 5.14 $su(\sigma, \tau) \subseteq su(\sigma_\tau, \tau_\sigma)$.

Proof: Let $\alpha \in su(\sigma, \tau)$. Then $\sigma\alpha = \tau\alpha$ and hence,

$$(\sigma_\tau \uplus \sigma_{-\tau}) \circ \alpha \uplus \alpha_{-\sigma} = (\tau_\sigma \uplus \tau_{-\sigma}) \circ \alpha \uplus \alpha_{-\tau},$$

which implies that $\sigma_\tau \circ \alpha \uplus \sigma_{-\tau} \circ \alpha \uplus \alpha_{-\sigma} = \tau_\sigma \circ \alpha \uplus \tau_{-\sigma} \circ \alpha \uplus \alpha_{-\tau}$. Since $dom(\sigma_\tau) = dom(\tau_\sigma)$, and because of the disjointness of substitutions on each side, we have $\sigma_\tau \circ \alpha = \tau_\sigma \circ \alpha$. Hence,

$$\sigma_\tau\alpha = \sigma_\tau \circ \alpha \uplus \alpha_{-(\sigma_\tau)} = \tau_\sigma \circ \alpha \uplus \alpha_{-(\tau_\sigma)} = \tau_\sigma\alpha,$$

implying that $\alpha \in su(\sigma_\tau, \tau_\sigma)$. ∎

Lemma 5.15 *Suppose that* $dom(\sigma) = dom(\tau)$. *Let* α *be any substitution. Then*

$$\sigma \circ \alpha = \tau \circ \alpha \quad \Leftrightarrow \quad \sigma\alpha = \tau\alpha.$$

Proof: Suppose that $\sigma \circ \alpha = \tau \circ \alpha$. We have:

$$\begin{aligned} \sigma\alpha &= \sigma \circ \alpha \uplus \alpha_{-\sigma} \\ &= \tau \circ \alpha \uplus \alpha_{-\tau} \\ &= \tau\alpha. \end{aligned}$$

The other direction follows from the proof of Lemma 5.14. ∎

We can also show the associativity of $\odot$ under certain conditions. First we need the following lemma.

Lemma 5.16 *Let σ, τ, ρ be unifiable substitutions, with σ idempotent. Then*

$$\sigma((\sigma \odot \tau) \odot \rho) = (\sigma \odot \tau) \odot \rho.$$

Proof:

$$\begin{aligned} \sigma((\sigma \odot \tau) \odot \rho) &= \sigma[(\sigma mgsu(\sigma, \tau)) \odot \rho] \\ &= \sigma[(\sigma mgsu(\sigma, \tau))(mgsu(\sigma mgsu(\sigma, \tau), \rho))] \\ &= (\sigma mgsu(\sigma, \tau))(mgsu(\sigma mgsu(\sigma, \tau), \rho)) \\ &= (\sigma mgsu(\sigma, \tau)) \odot \rho \\ &= (\sigma \odot \tau) \odot \rho. \ \blacksquare \end{aligned}$$

Lemma 5.17 *Let σ, τ, ρ be unifiable and idempotent substitutions. Then*

$$(\sigma \odot \tau) \odot \rho \equiv \odot\{\sigma, \tau, \rho\}.$$

Proof: Let $\alpha \in mgsu\{\sigma, \tau, \rho\}$ and $\beta \in mgsu(\sigma, \tau)$ be idempotent. Then $\beta \leq \alpha$ and hence, by Lemma 5.9, $\beta\alpha = \alpha$. Thus, $\alpha\beta\alpha = \sigma\alpha = \rho\alpha$, implying that $\alpha \in su(\sigma\beta, \rho)$. Then

$$\begin{aligned} mgsu(\sigma \odot \tau, \rho) &= mgsu(\sigma\beta, \rho) \\ &\leq \alpha \in mgsu\{\sigma, \tau, \rho\}, \end{aligned}$$

and hence, by Lemma 5.7,

$$\begin{aligned} (\sigma \odot \tau) \odot \rho &= \rho mgsu(\sigma \odot \tau, \rho) \\ &\leq \rho\alpha \\ &\in \rho mgsu\{\sigma, \tau, \rho\} \\ &= \odot\{\sigma, \tau, \rho\}. \end{aligned}$$

On the other hand, by Lemma 5.16, $(\sigma \odot \tau) \odot \rho \in su\{\sigma, \tau, \rho\}$ so $\alpha \leq (\sigma \odot \tau) \odot \rho$. This, by Lemmas 5.7 and 5.16, implies that

$$\odot\{\sigma, \tau, \rho\} = \sigma\alpha \leq \sigma((\sigma \odot \tau) \odot \rho) = (\sigma \odot \tau) \odot \rho,$$

which completes the proof. ∎

In a similar manner we can prove that

$$\sigma \odot (\tau \odot \rho) \equiv \odot\{\sigma, \tau, \rho\}$$

resulting in the following associativity result.

Lemma 5.18 *Let σ, τ, ρ be unifiable and idempotent substitutions. Then*

$$(\sigma \odot \tau) \odot \rho \equiv \sigma \odot (\tau \odot \rho).$$

The main results of this chapter are the following technical lemmas which are used in the proof of the Completeness Theorem.

Lemma 5.19 *Let G be an expression and σ and τ be substitutions such that:*

1. $dom(\sigma) \subseteq vars(G)$;
2. $vrange(\sigma) \cap vars(G) = \emptyset$;
3. $vrange(\sigma) \cap dom(\tau) = \emptyset$; *and*
4. $G\sigma \leq G\tau$.

Then $\sigma \leq \tau$.

Proof: Note that assumptions 1 and 2 together imply that σ is idempotent. Now, by assumptions 4 and 1 we have:

$$\sigma = \sigma_G \leq \tau_G,$$

and so, by Lemma 5.9 (since σ is idempotent),

$$\sigma\tau_G = \tau_G.$$

Hence,

$$\begin{aligned}
\tau &= \tau_G \uplus \tau_{-G} \\
&= \sigma\tau_G \uplus \tau_{-G} \\
&= (\sigma \circ \tau)_G \uplus (\tau_{-\sigma})_G \uplus \tau_{-G} \\
&= \sigma \uplus (\tau_{-\sigma})_G \uplus \tau_{-G}, \quad \text{(by assumption 2)}
\end{aligned}$$

$$
\begin{aligned}
&= \sigma \uplus (\tau_{-\sigma})_G \uplus (\tau_{-\sigma})_{-G}, \quad \text{(by assumption 1)} \\
&= \sigma \uplus \tau_{-\sigma} \\
&= \sigma \circ \tau \uplus \tau_{-\sigma}, \quad \text{(by assumption 3)} \\
&= \sigma\tau,
\end{aligned}
$$

which implies that $\sigma \leq \tau$. ∎

Lemma 5.20 *Let G_1 and G_2 be expressions and σ_i, η_i, and θ be substitutions, for $i = 1, 2$, such that:*

1. *$dom(\sigma_i) \subseteq vars(G_i)$, $i = 1, 2$;*
2. *$vrange(\sigma_i) \cap vars(G_i) = \emptyset$, $i = 1, 2$;*
3. *$vrange(\sigma_i) \cap dom(\theta\eta_i) = \emptyset$, $i = 1, 2$;*
4. *$G_i\sigma_i \leq G_i\theta\eta_i$, $i = 1, 2$; and*
5. *η_1 and η_2 are unifiable.*

Then σ_1 and σ_2 are unifiable and $\sigma_1 \odot \sigma_2 \leq \theta(\eta_1 \odot \eta_2)$.

Proof: By Lemma 5.19 (taking $\tau_i = \theta\eta_i, i = 1, 2$), we have that $\sigma_i \leq \theta\eta_i$, $i = 1, 2$. Now by Lemma 5.13, both conclusions follow immediately. ∎

Lemma 5.21 *Let G_1 and G_2 be expressions and σ, η, α, and θ be substitutions such that:*

1. $dom(\sigma) \subseteq vars(G_1)$;

2. $G_1\theta\eta = G_1\sigma\alpha$;

3. $vrange(\sigma) \cap Y = \emptyset$;

4. $dom(\alpha) \cap Y = \emptyset$;

where $Y = vars(G_2) - vars(G_1)$. *Define* $\gamma = \alpha \uplus (\theta\eta)_Y$. *Then* $G_i\theta\eta = G_i\sigma\gamma$. *for* $i = 1, 2$.

Proof: For $i = 1$, by the definition of Y and by assumption 3, $vars(G_1\sigma) \cap Y = \emptyset$, and hence,

$$G_1\sigma\gamma = G_1\sigma(\alpha \uplus (\theta\eta)_Y) = G_1\sigma\alpha = G_1\theta\eta.$$

Let $i = 2$. Define $X = vars(G_1) \cap vars(G_2)$. Since $X \subseteq vars(G_1)$, by the previous case, $(\sigma\gamma)_X = (\theta\eta)_X$. Also,

$$\begin{aligned}
(\sigma\gamma)_Y &= (\sigma \circ \gamma)_Y \uplus (\sigma_{-\sigma})_Y \\
&= (\sigma_{-\sigma})_Y \quad (\text{since } dom(\sigma \cap Y = \emptyset) \\
&= \gamma_Y \\
&= (\theta\eta)_Y \quad (\text{by the definition of } \gamma).
\end{aligned}$$

Hence,

$$\begin{aligned}
G_2\theta\eta &= G_2[(\theta\eta)_X \uplus (\theta\eta)_Y] \\
&= G_2[(\sigma\gamma)_X \uplus (\sigma\gamma)_Y] \\
&= G_2\sigma\gamma. \blacksquare
\end{aligned}$$

CHAPTER 6. KNOWLEDGE-BASED LOGIC PROGRAMMING

We now present a four-valued knowledge-based logic programming system based on the bilattice $\mathcal{FOUR}$. Later we will extend ideas presented in the subsequent sections to logics based on arbitrary distributive bilattices. The special four-valued case, however, is important on its merits and deserves detailed study. It provides the motivation behind the results presented in the sections on generalized knowledge-based logic programming. Furthermore, the notion of Closed World Assumption, which we discussed earlier, has a natural and useful counterpart in the context of the four-valued knowledge-based logic programs. Those results are presented in the last section of this chapter.

6.1 Logic Programming Syntax

Our logic programming language, denoted by $\mathcal{L}$, will have the bilattice $\mathcal{FOUR}$ as the underlying space of truth values. The alphabet of $\mathcal{L}$ consists of the usual sets of variables, constants, predicate symbols, and function symbols, similar to conventional logic programming. There is also an infinite number of new constants, called *generic constants*, that are distinct from the regular constants in the sense that they may not appear in any clause of a program over $\mathcal{L}$. In addition, $\mathcal{L}$ includes the connectives $\leftarrow$, $\neg$, $\wedge$, $\vee$, $\otimes$, and $\oplus$. $\wedge$ and $\vee$ represent the meet and join operations of the bilattice

in the truth ordering and $\otimes$ and $\oplus$ represent the meet and join in the knowledge ordering. The "quantifiers" $\prod, \sum$ represent the infinitary meet and join operations of the bilattice in the knowledge ordering.

The notions of *term* and *ground term* are defined in the usual way; they may contain generic constants. The set $U_{\mathcal{L}}$ of all ground terms in a language $\mathcal{L}$ is called the *Herbrand universe* of $\mathcal{L}$ [37]. An *atom* is either one of the constants `true` or `false` or an expression of the form $p(t_1, \cdots, t_n)$, where p is an n-ary predicate symbol and $t_1, \cdots, t_n$ are terms. An atom in which there are no occurrences of variables is called a *ground atom.*

Formulas are either atoms or expressions of the form $\neg A$, $A \oplus B$, $A \otimes B$, $A \wedge B$, or $A \vee B$, where A and B are formulas. A *complex formula* is a formula which is not an atom. A *normalized formula* is a formula in which the operator $\oplus$ does not occur. An *expression* is either a term or a formula. A *simple expression* is either a term or an atom. For an expression E, *vars*(E) denotes the set of all variables that occur in E.

A *clause* is an expression of the form:

$$\prod_{x_1 \cdots x_n} (A \leftarrow \sum_{y_1 \cdots y_m} (G)),$$

where A is an atom other than `true` and `false`, G is a formula, $x_1, \cdots, x_n$ are variables occurring in A, and $y_1, \cdots, y_m$ are variables occurring in G, but not in A. A is called the *head* and G is called the *body* of the clause. Normally, we drop the quantifiers from the clauses and simply write $A \leftarrow G$, where the variables occurring in the head of the clause are implicitly quantified by $\prod$, and the variables occurring in the clause body and not in the clause head are quantified by $\sum$. This convention is a standard practice in logic programming. Of course, in classical logic programming

the quantifiers are the truth quantifiers $\forall$ and $\exists$ which are assumed to implicitly quantify a clause. The choice of quantifiers $\prod$ and $\sum$ is motivated by our interest in the knowledge content of statements rather than their truth content.

Note that in conventional Prolog, a clause of the form $A \leftarrow$ is taken to stand for $A \leftarrow \texttt{true}$. So the empty clause body is the equivalent of a truth constant. In our language we designate symbols representing each element of the bilattice (with the exception of $\top$ and $\bot$). Hence, the above definition of atomic formulas includes the constants `true` and `false`. As usual, a *program* is a finite set of clauses. A *goal* is simply a formula.

Definition 6.1 A clause $A \leftarrow G$ is *normalized* if G is a normalized formula and either $vars(A) \subseteq vars(G)$, or $G \in \{\texttt{true}, \texttt{false}\}$. A program is *normalized* if all of its clauses are normalized.

The *Herbrand Base* of a program P, denoted B_P, is the set of all ground atoms using only function or predicate symbols occurring in P, and constants that either occur in P or are generic constants.

The extension of the language by these generic constants is done for technical reasons which will become clear in the proof of our Completeness Theorem. We now define the notion of a generic constant substituion which we will also need in the proof the Completeness Theorem.

Definition 6.2 A substitution δ is a *generic constant substitution* or simply a *gc-substitution* if it is of the form

$$\{x_1/a_1, x_2/a_2, \cdots, x_n/a_n\},$$

where $a_1, \cdots, a_n$ are distinct generic constants. If E is an expression, then δ is a *g-substitution for* E, if $dom(\delta) = vars(E)$.

The following technical lemma connecting gc-substitutions and the most general unifier will be used in the proof of the Completeness Theorem.

Lemma 6.3 *Let A and B be atoms such that $vars(A) \cap vars(B) = \emptyset$ and neither A nor B contains a generic constant. Let δ be a gc-substitution for A. Assume $A\delta$ and B are unifiable and $\eta = mgu(A\delta, B)$. Then A and B are unifiable, and if $\mu = mgu(A, B)$, then*

1. *$\delta\eta = \mu\tau$, where τ is a gc-substitution for $B\mu$, and*

2. *μ_A is injective and variable-pure.*

Proof: $vars(A) \cap vars(B) = \emptyset$ implies $B\delta = B$. So $\eta = mgu(A\delta, B\delta)$, and hence $\delta\eta$ unifies A and B. Let $\mu = mgu(A, B)$. Then $\delta\eta = \mu\tau$ for some τ. Since δ is a gc-substitution for A, we have $A\delta = A\delta\eta$. Thus $A\delta = A\delta\eta = A\mu\tau$. Now $dom(\delta) = vars(A)$ and all constants of δ are generic. Thus, since A and B do not contain any generic constants and μ is their *mgu*, μ cannot contain generic constants. But $x\mu\tau = x\delta$, which is generic, for every $x \in vars(A)$. This implies that μ_A must be variable-pure, and it must also be injective since δ is injective. It then follows from the equality $A\delta = A\mu\tau$ that τ is a gc-substitution for $A\mu$ and hence also for $B\mu$. ∎

We further extend the notion of unification to substitutions themselves. The notion of unifiable substitutions has been used in concurrent logic programming systems which use AND-parallelism [26]. These substitutions also play an essential role in our procedural semantics.

6.2 Fixpoint Semantics

In the classical two-valued logic programming, a single step operator on interpretations, denoted T_P, is associated with a program. In the absence of negation, this operator is monotonic and has a natural least fixpoint. It is this fixpoint which serves as the denotational meaning of the program. However, in the presence of negation in the clause bodies, the T_P operator is no longer monotonic and may not have a fixpoint. The idea of associating such an operator with programs carries over in a natural way to logic programming languages with a distributive bilattice as the space of truth values. However, the ordering in which the least fixpoint is evaluated is the knowledge ordering ($\leq_k$) and not the truth ordering ($\leq_t$). Since, knowledge operators are self-dual under negation in the $\leq_k$-ordering, presence of negation in the body of a program clause does not pose any of the problems associated with classical logic programming. The fixpoint semantics presented in this section is essentially due to Fitting [18].

Definition 6.4

1. An *interpretation* for a program P is a mapping $I : B_P \rightarrow \mathcal{FOUR}$.

2. We extend the interpretation I to ground formulas as follows:

$$\begin{aligned} I(\neg A) &= \neg I(A); \\ I(A_1 \ominus A_2) &= I(A_1) \ominus I(A_2); \\ I(A_1 \otimes A_2) &= I(A_1) \otimes I(A_2); \\ I(A_1 \wedge A_2) &= I(A_1) \wedge I(A_2); \\ I(A_1 \vee A_2) &= I(A_1) \vee I(A_2). \end{aligned}$$

3. We further extend the interpretation I to nonground formulas. For a nonground formula G:

$$I(G) = \prod\{I(G\sigma) \mid \sigma \text{ is a ground substitution for the variables of } G\}.$$

The following lemma is an easy consequence of the definitions involved.

Lemma 6.5 *Let I_1 and I_2 be two interpretations for a program P and let σ and τ be substitutions.*

1. *$I_1(G\sigma) \leq_k I_2(G\tau)$ for every formula G if and only if $I_1(A\sigma) \leq_k I_2(A\tau)$ for every atom A.*

2. *$I_1(G\sigma) = I_2(G\tau)$ for every formula G if and only if $I_1(A\sigma) = I_2(A\tau)$ for every atom A.*

Proof: The implication from left to right in part 1 is trivial. The opposite implication is proved by an easy induction on the structure of G, using the fact that the operations $\neg$, $\ominus$, $\oslash$, $\wedge$, and $\vee$ are monotone with respect to $\leq_k$.

Part 2 is an immediate consequence of part 1. ∎

Definition 6.6 The *initial interpretation* I_0 of a program P is defined as follows. For any atom $A \in B_P$:

$$I_0(A) = \begin{cases} \texttt{true} & \text{if } A = \texttt{true} \\ \texttt{false} & \text{if } A = \texttt{false} \\ \bot & \text{otherwise.} \end{cases}$$

Note that for any atomic formula G, if $I_0(G) \geq_k \texttt{true}$ (or $I_0(G) \geq_k \texttt{false}$), then $G = \texttt{true}$ (respectively $G = \texttt{false}$). Now we can associate a semantic operator with each program.

Definition 6.7 Let P be a program and let $A \in B_P$. The *semantic operator* Φ_P is a function mapping interpretations to interpretations, defined as follows:

$$\Phi_P(I)(A) = \begin{cases} \texttt{true} & \text{if } A = \texttt{true} \\ \texttt{false} & \text{if } A = \texttt{false} \\ \sum\{I(G\sigma) \mid A' \leftarrow G \in P \text{ and } A = A'\sigma\} & \text{otherwise} \end{cases}$$

Pointwise partial orderings are also defined on interpretations in the following manner:

1. $I_1 \leq_k I_2$ if $I_1(A) \leq_k I_2(A)$, for every ground atom $A \in B_P$.

2. $I_1 \leq_t I_2$ if $I_1(A) \leq_t I_2(A)$, for every ground atom $A \in B_P$.

Using this pointwise ordering, the space of interpretations itself becomes a distributive bilattice [18]. Furthermore, the Φ_P operator is monotonic with respect to the knowledge ordering [18]:

$$I_1 \leq_k I_2 \implies \Phi_P(I_1) \leq_k \Phi_P(I_2),$$

and hence, by the Knaster-Tarski theorem [53], Φ_P has a least fixpoint. It is this least fixpoint which provides the denotational meaning of the program P. In order to approximate the least fixpoint of the operator Φ_P, we use the following notion of upward iteration.

Definition 6.8 The *upward iteration* of Φ_P is defined as follows:

$$\Phi_P \uparrow \alpha = \begin{cases} I_0 & \text{if } \alpha = 0 \\ \Phi_P(\Phi_P \uparrow (\alpha - 1)) & \text{if } \alpha \text{ is a successor ordinal} \\ \sum\{\Phi_P \uparrow \beta \mid \beta < \alpha\} & \text{if } \alpha \text{ is a limit ordinal} \end{cases}$$

The smallest ordinal at which this sequence gives the least fixpoint of Φ_P is called the *closure ordinal.* In $\mathcal{FOUR}$ and in fact in any bilattice which satisfies the infinitary distributivity conditions, Φ_P is continuous and its closure ordinal is at most ω [18].

The following lemma shows that generic constants behave semantically like variables.

Lemma 6.9 *For every formula G and any gc-substitution σ for G,*

$$(\Phi_P \uparrow n)(G\sigma) = (\Phi_P \uparrow n)(G), \quad \textit{for every } n < \omega.$$

Proof: Consider the following property of an interpretation I.

$$I(G\sigma) = I(G), \quad \text{for every formula } G \text{ and gc-substitution } \sigma \text{ for } G. \tag{6.1}$$

We first prove that, for every interpretation I, if I has the property (6.1), then so does $\Phi_P(I)$. Note that, to show $\Phi_P(I)$ has the property, it suffices to prove that $\Phi_P(I)(G\sigma) \leq_k \Phi_P(I)(G\delta)$, every ground substitution δ. Furthermore, by Lemma 6.5, we can assume G is an atom.

Assume I in an arbitrary interpretation such that (6.1) holds and G is an atom A. $\Phi_P(I)(A\sigma)$ is the join of all elements of $\mathcal{B}$ of the form $I(G'\tau)$ such that $A\sigma = B\tau$ for some clause $B \leftarrow G'$ of P and some ground substitution τ. Let $x_1, \ldots, x_n$ be the

variables of A so that $A\sigma = A(x_1\sigma, \ldots, x_n\sigma) = B\tau$. Since the constants $x_1\sigma, \ldots, x_n\sigma$ are distinct and do not occur in B, there is a substitution τ' such that $A = B\tau'$. Since $A\sigma = B\tau'\sigma$, we can assume $\tau'\sigma = \tau$. We also have $A\delta = B\tau'\delta$ and hence $I(G'\tau'\delta)$ is one of the set of elements of $\mathcal{FOUR}$ whose join defines $\Phi_P(I)(A\delta)$. But $I(G'\tau) = I(G'\tau'\sigma) = I(G'\tau') \leq_k I(G'\tau'\delta)$; the second equality holds because of our assumption that I has the property (6.1). Thus (6.1) is preserved under Φ_P.

The conclusion of the lemma now follows by an easy induction on n and the fact that I_0 clearly has the property (6.1). ∎

Using the distributivity and infinitary distributivity properties of the bilattice $\mathcal{FOUR}$ we can establish the semantic equivalence of ordinary and normalized programs.

Theorem 6.10 *Let P be a program over the language $\mathcal{L}$. There exists a normalized program P' over a possibly larger language $\mathcal{L}'$ with the property that for every $A \in B_P$,*

$$(\Phi_P \uparrow \omega)(A) = (\Phi_{P'} \uparrow \omega)(A).$$

Proof: Since we have distributivity, every clause in the program P can be written as

$$A \leftarrow G_1 \oplus \cdots \oplus G_n,$$

where each G_i is a normalized formula (the infinitary distributivity conditions are needed since the clauses in the program body may contain variables which are implicitly quantified). P' is formed by first replacing each clause in the above form by the set of n clauses:

$$A \leftarrow G_1, A \leftarrow G_2, \cdots, A \leftarrow G_n.$$

For each i, if either $vars(A) \subseteq vars(G_i)$ or $G_i \in \{\texttt{true}, \texttt{false}\}$, then the clause $A \leftarrow G_i$ is not further transformed; otherwise it is replaced by $A \leftarrow G_i \otimes E(x_1, \ldots, x_n)$, where E is a new predicate symbol and $\{x_1, \ldots, x_n\} = vars(A) - vars(G_i)$. In this event the two clauses $E(x_1, \ldots, x_n) \leftarrow \texttt{true}$ and $E(x_1, \ldots, x_n) \leftarrow \texttt{false}$ are also added. Obviously, the transformed program P' is a normalized program. Furthermore, it should be clear from the definition of Φ_P that, for every interpretation I, $\Phi_P(I)(E(t_1, \ldots, t_n)) = \top$ for all terms $t_1, \ldots, t_n$. It is easy to check that for every $A \in B_P$, every interpretation I, and any extension I' of I to $B_{P'}$, we have

$$\Phi_P(\Phi_P(I))(A) = \Phi_{P'}(\Phi_{P'}(I'))(A).$$

Thus, $(\Phi_P \uparrow \alpha)(A) = (\Phi_{P'} \uparrow \alpha)(A)$, for any $A \in B_P$ and every $\alpha \geq 2$. ∎

Since, according to the above theorem, ordinary programs and normalized programs are equivalent, we will, from now on, consider only normalized programs.

Interpretations over distributive bilattices exhibit some interesting algebraic properties. In particular, we have found the following lemmas useful in the proofs of our the Soundness and Completeness Theorems.

Lemma 6.11 *Let I be an interpretation, G a formula, and θ, η substitutions. Suppose that θ is a variant of η w.r.t. G. Then $I(G\theta) = I(G\eta)$.*

Proof: The result follows immediately from the definitions of interpretation and variant. ∎

Lemma 6.12 *Let G_1 and G_2 be formulas, and suppose I is an interpretation. Then for $\Box \in \{\odot, \oplus, \wedge, \vee\}$, we have*

$$I(G_1 \Box G_2) \geq_k I(G_1) \Box I(G_2).$$

Proof:

$$\begin{aligned}
I(G_1 \Box G_2) &= \prod\{I(G_1 \Box G_2)\sigma \mid \sigma \textit{ is a ground substitution } \} \\
&= \prod\{I(G_1\sigma) \Box I(G_2\sigma) \mid \sigma \textit{ is a ground substitution } \} \\
&\geq_k \prod\{I(G_1\gamma) \Box I(G_2\delta) \mid \gamma, \delta \textit{ are ground substitutions } \} \\
&= \prod\{I(G_1\gamma) \mid \gamma \textit{ is ground } \} \Box \prod\{I(G_2\delta) \mid \delta \textit{ is ground } \} \\
&= I(G_1) \Box I(G_2). \blacksquare
\end{aligned}$$

Lemma 6.13 *Let α and β be substitutions, let I be an interpretation, and let F be a formula.*

1. *If $\alpha \leq \beta$, then $I(F\alpha) \leq_k I(F\beta)$.*

2. *$I(F\alpha) \leq_k I(F\alpha\beta)$.*

Proof: (1) The proof is straightforward from the definitions of interpretation and properties of $\prod$: Assume $\alpha \leq \beta$, i.e., $\alpha\sigma = \beta$ for some substitution σ. Then

$$\begin{aligned}
I(F\alpha) &= \prod\{ I(F\alpha\delta) | \delta \text{ a ground substitution for } F\sigma \} \\
&\leq_k \prod\{ I(F\alpha\sigma\delta) | \delta \text{ a ground substitution for } F\sigma\delta \} \\
&= I(F\alpha\sigma) \\
&= I(F\beta).
\end{aligned}$$

Part (2) is an immediate consequence of (1). ■

Corollary 6.14 *Let θ_1 and θ_2 be unifiable substitutions, let I be an interpretation, and let F be a formula. Then $I(F(\theta_1 \odot \theta_2)) \geq_k I(F\theta_i)$, for $i = 1, 2$.*

Proof: Since θ_1 and θ_2 are unifiable, $\theta_1 \odot \theta_2 = \theta_1\gamma$, where γ is an *mgsu* of θ_1 and θ_2. Now use Lemma 6.13. ■

Lemma 6.15 *Let $A \leftarrow G \in P$. Then for every interpretation I and every substitution θ,*

$$\Phi_P(I)(A\theta) \geq_k I(G\theta).$$

Proof: Recall that the interpretation of a nonground formula is defined to be the greatest lower bound in the knowledge ordering of the interpretations of all ground substitution instances of the formula. Hence, it suffices to show that $\Phi_P(I)(A\theta\delta) \geq_k I(G\theta\delta)$ for every ground substitution δ. But

$$\Phi_P(I)(A\theta\delta) = \bigsqcup\{\, I(G'\sigma) \mid A' \leftarrow G' \in P \text{ and } A\theta\delta = A'\sigma \,\}.$$

Taking $A \leftarrow G$ for $A' \leftarrow G'$ and $\theta\delta$ for σ, we see that $I(G\theta\delta)$ is actually a member of the set whose least upper bound is $\Phi_P(I)(A\theta\delta)$. ■

Lemma 6.16 *Let $A \leftarrow G \in P$. Then for any substitution θ,*

$$(\Phi_P \uparrow \omega)(A\theta) \geq_k (\Phi_P \uparrow \omega)(G\theta).$$

Proof: It follow from the last lemma by an easy induction on n that $(\Phi_P \uparrow n)(A\theta) \geq_k (\Phi_P \uparrow (n-1))(G\theta)$ for all $n \geq 1$. Using this fact we have

$$\begin{aligned}
(\Phi_P \uparrow \omega)(A\theta) &= \bigsqcup\{(\Phi_P \uparrow n)(A\theta) \mid 1 \leq n < \omega\} \\
&\geq_k \bigsqcup\{(\Phi_P \uparrow (n-1))(G\theta) \mid 1 \leq n < \omega\} \\
&= \bigsqcup\{(\Phi_P \uparrow n)(G\theta) \mid n < \omega\} \\
&= (\Phi_P \uparrow \omega)(G\theta). \blacksquare
\end{aligned}$$

6.3 Procedural Semantics

Fitting has proposed a bilattice-based procedural model [17, 18] based on a version of Smullyan style semantic tableaux [51]. Fitting's extension of semantic tableaux to bilattices (in this case $\mathcal{FOUR}$) involves using signed formulas of the form FX and TX, where X is a formula. Informally, FX (respectively, TX) says that X is either `false` or $\bot$ (respectively, `true` or $\bot$). Since tableaux are refutation arguments (as in resolution), if one arrives at a contradiction by starting with FX, it follows that X is either `true` or $\top$, in other words, X is at least `true`. Similarly, a contradiction deriving from TX will mean that X is at least `false`. Finally, if contradictions follow from both FX and TX, then X is $\top$, and if contradictions follow from neither then X is $\bot$.

In contrast, we use a resolution-based procedural semantics which will allow us to start with any formula as a goal and within a uniform framework derive both negative and positive information about that goal. In the context of the bilattice $\mathcal{FOUR}$ this means that if the derivation from a goal A leads to success, then A is at least `true`,

and if it leads to failure, then A is at least `false`. Informally, if a derivation from A is successful, we say that A has a *proof*, and if the derivation is failed, we say that A has a *refutation*. Note that we do not use the notion of *refutation* in the same way as it is used in resolution-based methods. Also, our notions of successful and failed derivations are quite different from those used in such methods.

The procedural model we propose is essentially an extension of the well-known operational semantics known as SLDNF-resolution which is based on SLD-resolution [28, 2] augmented with the *negation as failure* rule [8]. Negation as Failure uses the notion of *finite failure* to decide if the derivation of a goal has failed. For a definite program P, the *finite failure set* of P, is the set of all ground atoms A for which there exists a finitely failed resolution tree for $P \cup \{A\}$, that is, one which is finite and contains no success branches. A failure branch in such a tree, is one whose leaf node cannot unify with the head of any clause in P.

Our procedural model, called *SLDPF-resolution* (PF stands for Partial Failure) does not require that a finitely failed derivation tree have no success branches. In our approach, the notions of failure and success are treated in exactly the same manner. Thus, a derivation tree, which we call an *SLDPF-tree* for a given goal, can represent both failed and successful derivations (i.e., both refutations and proofs) of that goal. This feature, which we call *Negation as Partial Failure*, is one of the consequences of shifting our emphasis from truth to knowledge. In our derivation trees, for every subtree rooted at an internal node labeled by some atom, say A, each direct descendant of A corresponds to a clause whose head unifies with A. Each such clause is seen as contributing to the information the system has about the truth or falsity of A. All clauses with the same head can be combined using the $\ominus$ operator

which, as we explained earlier, is self-dual under negation. In the classical logic programming approach, clauses with the same head are combined using $\vee$, and since the dual of $\vee$ under negation is $\wedge$, a failed subgoal is one whose derivation tree has no success branches. In our approach, existence of only one failed branch is sufficient for failure. Thus, we may have goals whose SLDPF-tree has both success and failure branches.

One of the problems in dealing with negation in SLDNF-resolution is that in order to ensure soundness, only ground negative literals can be selected for resolution. This is sometimes referred to as the *safeness condition* [37]. The reason for this problem is that once a negative literal is reached during the derivation, the system must attempt to prove its positive component. This positive literal will be universally quantified, since $\exists$ will change to $\forall$ when negation is taken outside to establish a derivation for the positive counterpart of the subgoal. However, the system will actually attempt to prove an existentially quantified subgoal, since $\exists$ is implicitly assumed. On the other hand, we interpret free variables in the body of a clause as being quantified by $\sum$, which is its own dual under negation. Therefore, our procedural semantics will remain sound even in the presence of non-ground negative subgoals.

SLDPF-resolution also extends the treatment of $\neg$ to the operators $\wedge$, $\vee$, and $\otimes$. In other words, if during the derivation a subgoal is reached which contains one of these operators, then an attempt is made to establish appropriate derivations for the two operands based on the way these operators, viewed as lattice operations, act on the elements of the bilattice. This is precisely the point at which we need the notion of substitution unifiers. S-unifiers will ensure that the answer substitutions obtained

from the derivation trees of each operand will not contradict each other once they are finally applied to the formula itself.

Note that negative information can be derived through the explicit use of clauses of the form $A \leftarrow$ `false`. This allows us to treat success and failure in a completely symmetrical manner. Later we will describe how we can extend Negation as Partial Failure to incorporate the Closed World Assumption with only minor modifications to our procedural and fixpoint semantics. We now present the details formally in the following definitions.

Definition 6.17 An *SLDPF-tree for* $P \cup \{A\}$, where P is a normalized program and A is an atom, is a (possibly infinite) tree satisfying the following conditions:

1. The root of the tree is A.

2. Let G be a nonleaf node. Then G is an atom and for each clause $G' \leftarrow G'' \in P$, if G and G' are unifiable, then the node has a child $G''\gamma$, where $\gamma = mgu(G, G')$. We say that γ is the *substitution associated with the edge* between G and $G''\gamma$.

3. Let G be a leaf node. Then either G is an atom which does not unify with the head of any clause (in particular, G can be `true` or `false`), or G is a complex formula.

Let E be an expression and let σ be a substitution. Recall that the *restriction* of σ with respect to E (or, more precisely, to the variables of E) is denoted by σ_E.

Definition 6.18 Let P be a normalized program and G a normalized goal. Then

1. G has a *proof of rank 0 with answer* θ if $G =$ `true` and θ is the identity substitution ε. G has a *refutation of rank 0 with answer* θ if $G =$ `false` and θ is the identity substitution ε.

2. G has a *proof of rank* $k+1$ *with answer* θ if:

   (a) G is an atom different from both `true` and `false`, and $P \cup \{G\}$ has an SLDPF-tree with at least one leaf node G', such that G' has a proof of rank k with answer θ', and $\theta = (\sigma_1 \cdots \sigma_n \theta')_G$, where $\sigma_1, \cdots, \sigma_n$ are the substitutions associated with edges along the path from G to G'; or

   (b) $G = \neg G'$, and G' has a refutation of rank k with answer θ; or

   (c) $G = G_1 \odot G_2$ or $G = G_1 \wedge G_2$, and G_1 and G_2 have proofs of ranks k_1 and k_2 with answers θ_1 and θ_2, respectively, $k = max(k_1, k_2)$, and $\theta = (\theta_1' \odot \theta_2')_G$, where θ_i' is a variant of θ_i w.r.t G_i $(i = 1, 2)$; or

   (d) $G = G_1 \vee G_2$, and at least one of G_1 or G_2 has a proof of rank k with answer θ.

3. G has a *refutation of rank* $k+1$ *with answer* θ if:

   (a) G is an atom different from both `true` and `false`, and $P \cup \{G\}$ has an SLDPF-tree with at least one leaf node G' such that G' has a refutation of rank k with answer θ', and $\theta = (\sigma_1 \cdots \sigma_n \theta')_G$, where $\sigma_1, \cdots, \sigma_n$ are the substitutions associated with edges along the path from G to G'; or

   (b) $G = \neg G'$, and G' has a proof of rank k, with answer θ; or

(c) $G = G_1 \otimes G_2$ or $G = G_1 \vee G_2$, and G_1 and G_2 have refutations of ranks k_1 and k_2 with answers θ_1 and θ_2, respectively, $k = max(k_1, k_2)$, and $\theta = (\theta'_1 \odot \theta'_2)_G$, where θ'_i is a variant of θ_i w.r.t. G_i $(i = 1, 2)$; or

(d) $G = G_1 \wedge G_2$, and at least one of G_1 or G_2 has a refutation of rank k with answer θ.

Definition 6.19 Let P be a normalized program and G a normalized goal. Then G has a proof (respectively, a refutation) with answer θ, if G has a proof (respectively, a refutation) of rank k, with answer θ, for some $k \geq 0$.

In parts 2(c) and 3(c) of the above definition, the reason for allowing variants of answers before taking their substituion unification (e.g., $\theta'_1 \odot \theta'_2$), is to assure that variables do not conflict in independent derivations associated with complex subgoals. In order to select the appropriate variant, we can compose the answers with especial renaming substitutions which repalce that variables in the *vrange* of answers by variables which have not occurred in the derivation up to that point. The additional bindings which result from the composition ensure that the relationships between variables of independent derivations are preserved.

We also adopt the standard practice of assuming that suitable variants of program clauses are used at each step of a proof or refutation. This is to ensure that the variables used for the derivation do not already occur in the derivation up to that point. We will refer to this assumption as the *unique renaming assumption.*

Before proving the Soundness and Completeness Theorems for our knowledge-based logic programming system, let us illustrate the procedural semantics by an example. Consider the following the following normalized program P.

1. $p(u) \leftarrow q(f(u))$

2. $q(y) \leftarrow r(y) \otimes s(y)$

3. $r(z) \leftarrow \neg t(z)$

4. $s(f(a)) \leftarrow \texttt{false}$

5. $t(f(b)) \leftarrow \texttt{false}$

6. $r(f(a)) \leftarrow \texttt{false}$

7. $p(b) \leftarrow \texttt{true}$

8. $r(f(b)) \leftarrow \texttt{false}$

and the goal $p(x)$. The process of computation of this goal using SLDPF-resolution is depicted in Figure 6.1. Once again the dotted lines represent start of an independent computation for a subgoal. So, for example the dotted lines from the subgoal $r(f(u)) \otimes s(f(u))$ to each of $r(f(u))$ and $s(f(u))$ represent the fact that, according to the procedural semantics, in order to establish a proof (refutation) for $r(f(u)) \otimes s(f(u))$, the system must try to construct proofs (refutations) for each of the operands. The situation is similar for the subgoal $\neg t(f(u))$. Note that the in the latter case (when the operator is $\neg$), the computation follows a similar process to that of SLDNF-resolution.

Note that the subgoal $r(f(u))$ has refutations with answers $\{u/a\}$ and $\{u/b\}$. On the other hand, the subgoal $s(f(u))$ has a refutation with answer $\{u/a\}$. In this case, the substitution unification process will only accept the consistent bindings obtained for the refutations of each of the subgoals. Thus the subgoal $r(f(u)) \otimes s(f(u))$ has a

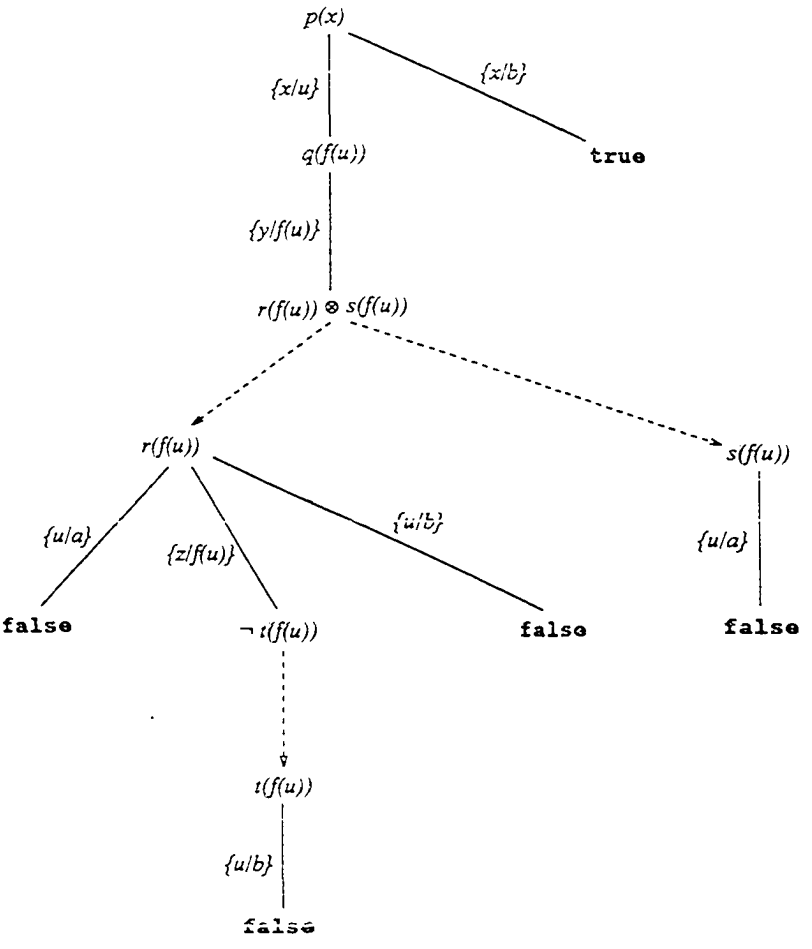


Figure 6.1: An Example of SLDPF-resolution

refutation with answer $\{u/a\}$. It can be seen that the goal $p(x)$ has both a refutation (with answer $\{x/a\}$) and a proof (with answer $\{x/b\}$).

6.4 Basic Results

In this section we present the soundness and completeness results for Negation as Partial Failure. These theorems establish the correspondence between the procedural

and the fixpoint semantics. For the purpose of these theorems we will denote the ordering in the knowledge lattice by $\succeq$.

Theorem 6.20 (Soundness) *Let P be a normalized program, G a normalized goal, and θ a substitution for the variables of G.*

1. *If G has a proof with answer θ, then $(\Phi_P \uparrow \omega)(G\theta) \succeq$ `true`.*
2. *If G has a refutation with answer θ, then $(\Phi_P \uparrow \omega)(G\theta) \succeq$ `false`.*

Proof: (By induction on k)

Basis: ($k = 0$) Suppose G has a proof of rank 0 with answer θ. Then $G =$ `true` and $\theta = \varepsilon$. Now, I_0(`true`) = `true`, and since Φ_P is monotonic, $(\Phi_P \uparrow \omega)(G\theta) \succeq$ `true`. Similarly, if G has a refutation of rank 0 with answer θ, then $(\Phi_P \uparrow \omega)(G\theta) \succeq$ `false`.

Induction: Assume the result holds for proofs and refutations of rank k, and suppose that G has a proof of rank $k+1$ with answer θ. There are five cases to consider:

1. *G is an atom:* Then $P \cup \{G\}$ has an SLDPF-tree with at least one leafnode F, which has a proof of rank k with answer θ', such that $\theta = (\sigma_1 \cdots \sigma_n \theta')_G$, where $\sigma_1, \cdots, \sigma_n$ are the mgu's associated with the edges along the path from G to F in the tree. Now, the result is proved by performing a secondary induction on the length n of this path.

   The result is vacuously true if $n = 0$, since in this case the only atoms with SLDPF-trees are `true` and `false` which do not have proofs of positive rank.

   Now, suppose the result holds for SLDPF-trees with success branches of depth $n \geq 0$, and consider one of depth $n+1$. In this case, P must have a clause

$G' \leftarrow G_1$ such that $\sigma_1 = mgu(G, G')$ and $G_1\sigma_1$ has an SLDPF-tree with a success branch of depth n. Then it is easy to see that $G_1\sigma_1$ has a proof of rank k with answer θ_1 such that $\theta = (\sigma_1\theta_1)_G$. By the secondary induction hypothesis, $(\Phi_P \uparrow \omega)(G_1\sigma_1\theta_1) \succeq \texttt{true}$. Then,

$$\begin{aligned}
(\Phi_P \uparrow \omega)(G\theta) &= (\Phi_P \uparrow \omega)(G\sigma_1\theta_1) && \text{since } G\theta = G\sigma_1\theta_1 \\
&= (\Phi_P \uparrow \omega)(G'\sigma_1\theta_1) && \text{since } \sigma_1 = mgu(G, G') \\
&\succeq (\Phi_P \uparrow \omega)(G_1\sigma_1\theta_1) && \text{by Lemma 6.16} \\
&\succeq \texttt{true} && \text{by the secondary ind. hyp.}
\end{aligned}$$

2. *G is $\neg G'$:* Then G' must have a refutation of rank k with answer θ. By the inductive hypothesis, $(\Phi_P \uparrow \omega)\ (G'\theta) \succeq \texttt{false}$. Hence,

$$\begin{aligned}
(\Phi_P \uparrow \omega)\ (G\theta) &= (\Phi_P \uparrow \omega)\ (\neg G'\theta) \\
&= \neg(\Phi_P \uparrow \omega)\ (G'\theta) \\
&\succeq \texttt{true}.
\end{aligned}$$

3. *G is $G_1 \odot G_2$:* Then G_i has a proof of rank k_i with answer θ_i, such that $\theta = (\theta'_1 \odot \theta'_2)_G$, where θ'_i is a variant of θ_i w.r.t. G_i. Furthermore, $k = max(k_1, k_2)$. By the inductive hypothesis, $(\Phi_P \uparrow \omega)\ (G_i\theta_i) \succeq \texttt{true}$ for $i = 1, 2$. Hence,

$$\begin{aligned}
&(\Phi_P \uparrow \omega)\ ([G_1 \otimes G_2\ ]\theta) = (\Phi_P \uparrow \omega)\ (G_1\theta \otimes G_2\theta\ ) \\
&\quad\succeq (\Phi_P \uparrow \omega)\ (G_1\theta) \otimes (\Phi_P \uparrow \omega)\ (G_2\theta), && \text{by Lemma 6.12} \\
&\quad= (\Phi_P \uparrow \omega)\ (G_1(\theta'_1 \odot \theta'_2)) \otimes (\Phi_P \uparrow \omega)\ (G_2(\theta'_1 \odot \theta'_2)) \\
&\quad\succeq (\Phi_P \uparrow \omega)\ (G_1\theta'_1) \otimes (\Phi_P \uparrow \omega)\ (G_2\theta'_2), && \text{by Corollary 6.14} \\
&\quad= (\Phi_P \uparrow \omega)\ (G_1\theta_1) \otimes (\Phi_P \uparrow \omega)\ (G_2\theta_2), && \text{by Lemma 6.11} \\
&\quad\succeq \texttt{true}, && \text{by the inductive hypothesis and properties of } \otimes .
\end{aligned}$$

4. *G is $G_1 \wedge G_2$:* This case, for proofs, is identical to the case for $G_1 \otimes G_2$.

5. *G is $G_1 \vee G_2$:* Then one of G_1 and G_2 (say G_1) has a proof of rank k with answer θ. Hence,

$$\begin{aligned}(\Phi_P \uparrow \omega)\,([G_1 \vee G_2\,]\theta) &= (\Phi_P \uparrow \omega)\,(G_1\theta \vee G_2\theta) \\ &\succeq (\Phi_P \uparrow \omega)\,(G_1\theta) \vee (\Phi_P \uparrow \omega)\,(G_2\theta), && \text{by Lemma 6.12} \\ &\succeq \texttt{true}, && \text{by the inductive hypothesis and properties of } \vee .\end{aligned}$$

Finally, the induction part is proved in a similar fashion for the case when G has a refutation of rank $k+1$. ∎

The key to the proof of the Completeness Theorem is the following Lifting Lemma. It generalizes the Lifting Lemma which is used in establishing the completeness of SLD-resolution (see [37]).

Lemma 6.21 (Lifting Lemma) *Let P be a normalized program, G a normalized goal, and θ a substitution without generic constants for the variables of G. Suppose that $G\theta$ has a proof (respectively, refutation) with answer η. Then G has a proof (respectively, refutation) with answer σ, such that $G\theta\eta = G\sigma\gamma$, for some substitution γ.*

Proof: (By induction on k)

Basis: ($k = 0$)

Suppose that $G\theta$ has a proof of rank 0 with answer η. Then $G\theta = \texttt{true}$, and hence, $G = \texttt{true}$ and $\eta = \varepsilon$. But $G = \texttt{true}$ has a proof of rank 0 with answer ε. Clearly,

$\theta\eta = \theta\varepsilon = \theta = \varepsilon\theta$. Now, take $\sigma = \varepsilon$ and let $\gamma = \theta$. By a similar argument, if $G\theta$ has a refutation of rank 0, then G has a refutation of rank 0 with answer σ, such that $\sigma = \varepsilon$ and $\gamma = \theta$.

Induction: Assume the result holds for proofs and refutations of rank k. We present the argument for proofs of rank $k+1$; the argument for refutations is similar. Suppose that $G\theta$ has a proof of rank $k+1$ with answer η. We have to consider the following cases:

1. *G is an atom A*: Then $P \cup \{A\theta\}$ has an SLDPF-tree with at least one success branch. Furthermore, the corresponding leafnode F has a proof of rank k with answer ρ, such that $\eta = (\sigma_1 \cdots \sigma_n \rho)_{A\theta}$, where $\sigma_1, \cdots, \sigma_n$ are the mgu's associated with the edges along the path in the tree from $A\theta$ to F. The result is proved by a secondary induction on the length n of this path similar to the proof of the Lifting Lemma for SLD-resolution.

   If $n = 0$, then the result holds vacuously, since in that case, $A\theta = \texttt{true}$ and it only has a proof of rank $k = 0$.

   For the induction step, assume that $n > 0$. Then there exists a clause $B \leftarrow G_1 \in P$, such that $\sigma_1 = mgu(A\theta, B)$. We consider two cases. If $vars(B) \subseteq vars(G_1)$, then it must be the case that $G_1 = \texttt{true}$ and thus $n = 1$. Now, G_1 has a proof of rank 0 with answer ε, and $\eta = (\sigma_1)_{A\theta}$. Since $A\theta$ and B are unifiable via σ_1, by the unique renaming assumption, A and B are unifiable via $\theta\sigma_1$. Let $\tau = mgu(A, B)$. Then $\theta\sigma_1 = \tau\gamma$ for some substitution γ. But, $G_1\tau = \texttt{true}$ has a proof of rank 0 with answer ε. Hence, A has a proof of rank 1 with answer σ, where σ is the restriction of τ to $vars(A)$. Furthermore, we have $A\theta\eta = A\theta\sigma_1 = A\tau\gamma = A\sigma\gamma$.

On the other hand, if $vars(B) \subseteq vars(G_1)$, then n must be greater than 1. In this case, $\sigma_1 = mgu(A\theta, B)$ and $G_1\sigma_1$ has an SLDPF-tree with a success branch of length n whose associated mgu's are $\sigma_2, \cdots, \sigma_{n+1}$, and whose leafnode F has a proof of rank k with answer β. Moreover, $\eta = (\sigma_1 \cdots \sigma_{n+1}\beta)_{A\theta}$, and $G_1\sigma_1$ has a proof of rank $k+1$ with answer $\rho = (\sigma_2 \cdots \sigma_{n+1}\beta)_{G_1\sigma_1}$. Now, since $A\theta$ and B are unifiable via σ_1, by the unique renaming assumption, A and B are unifiable via $\theta\sigma_1$. Let $\tau = mgu(A, B)$ and let δ be a substitution such that $\theta\sigma_1 = \tau\delta$. Using the unique renaming assumption again, we have $G_1\sigma_1 = G_1\theta\sigma_1 = G_1\tau\delta$. Thus, $G_1\tau\delta$ has a proof of rank $k+1$ with answer ρ. Now, recall that the SLDPF-tree with the root node $G_1\tau\delta = G_1\sigma_1$ has a success branch of length n, as described above. Hence, by the secondary inductive hypothesis, $G_1\tau$ has a proof of rank $k+1$ with answer α, such that $G_1\tau\delta\rho = G_1\tau\alpha\gamma$ for some substitution γ.

Now, let $\alpha = (\alpha_1 \cdots \alpha_n\phi)_{G_1\tau}$, where the α_i's are the mgu's on the success branch of the SLDPF-tree of $G_1\tau$ whose leafnode has a proof of rank k with answer ϕ. Then, by the definitions of SLDPF-tree and proof, A has a proof of rank $k+1$ with answer $\sigma = (\tau\alpha_1 \cdots \alpha_n\phi)_A$. Furthermore, we have:

$$\begin{aligned}
A\sigma &= A\tau\alpha_1 \cdots \alpha_n\phi \\
&= B\tau\alpha_1 \cdots \alpha_n\phi \\
&= B\tau(\alpha_1 \cdots \alpha_n\phi)_{B\tau} \\
&= B\tau(\alpha_1 \cdots \alpha_n\phi)_{G_1\tau} \quad (\text{since } vars(B) \subseteq vars(G_1)) \\
&= B\tau\alpha \\
&= A\tau\alpha.
\end{aligned}$$

Hence, $\sigma = (\tau\alpha)_A$. Now,

$$\begin{aligned}
A\theta\eta &= A\theta\sigma_1\cdots\sigma_{n+1}\beta \\
&= A\tau\delta\sigma_2\cdots\sigma_{n+1}\beta \\
&= B\tau\delta(\sigma_2\cdots\sigma_{n+1}\beta)_{B\tau\delta} \\
&= B\tau\delta(\sigma_2\cdots\sigma_{n+1}\beta)_{G_1\tau\delta} \\
&= B\tau\delta(\sigma_2\cdots\sigma_{n+1}\beta)_{G_1\sigma_1} \\
&= B\tau\delta\rho \\
&= B\tau(\delta\rho)_{B\tau} \\
&= B\tau(\delta\rho)_{G_1\tau} \\
&= B\tau(\alpha\gamma)_{G_1\tau} \\
&= B\tau(\alpha\gamma)_{B\tau} \\
&= B\tau\alpha\gamma \\
&= A\tau\alpha\gamma \\
&= A\sigma\gamma.
\end{aligned}$$

Thus, we have shown that A has a proof of rank $k+1$ with answer σ such that $A\theta\eta = A\sigma\gamma$.

2. *G is $\neg G'$:* Then $G'\theta$ has a refutation of rank k with answer η. By the inductive hypothesis, G' has a refutation of rank k with answer σ, such that $G'\theta\eta = G'\sigma\gamma$, for some substitution γ. Hence, $G = \neg G'$ has a proof of rank $k+1$ with answer σ and $G\theta\eta = G\sigma\gamma$.

3. *G is* $G_1 \otimes G_2$ *:* Then $G_1\theta$ has a proof of rank k_1 with answer η_1 and $G_2\theta$ has a proof of rank k_2 with answer η_2, $k = max(k_1, k_2)$, and $\eta = (\eta_1' \odot \eta_2')_{G\theta}$, where η_i' is a variant of η_i w.r.t. $G_i\theta$ $(i = 1, 2)$. Now, by the inductive hypothesis, for $i = 1, 2$, G_i has a proof of rank k_i with answer σ_i, such that $G_i\theta\eta_i = G_i\sigma_i\gamma_i$, for some substitution γ_i. Let σ_i' be variants of the answers σ_i w.r.t. G_i, $i = 1, 2$, such that $vars(G_i\sigma_i)$ is disjoint from $vars(G)$ and from $dom(\theta\eta_i')$, and so that $dom(\sigma_i') \subseteq vars(G_i)$. So $vrange(\sigma_i')$ is also disjoint from $vars(G)$ and from $dom(\theta\eta_i')$. Thus all the conditions of Lemma 5.20 are satisfied. It follows that $\sigma_1' \odot \sigma_2'$ exists and, furthermore, there is a substitution γ such that $\theta(\eta_1' \odot \eta_2') = (\sigma_1' \odot \sigma_2')\gamma$. Let $\sigma = (\sigma_1' \odot \sigma_2')_G$. Then G has a proof of rank $k+1$ with answer σ such that $G\theta\eta = G\sigma\gamma$.

4. *G is* $G_1 \wedge G_2$ *:* This case is similar to the case for $G_1 \otimes G_2$.

5. *G is* $G_1 \vee G_2$ *:* Then $G\theta = G_1\theta \vee G_2\theta$ has a proof of rank $k+1$ with answer η and hence, $G_1\theta$ or $G_2\theta$ (say $G_1\theta$) has proof of rank k with answer η. By the inductive hypothesis, G_1 has a proof of rank k with answer σ such that $G_1\theta\eta = G_1\sigma\alpha$, for some substitution α. By the unique renaming assumption, σ and α can be chosen to satisfy the condition that $vrange(\sigma) \cap Y = \emptyset$ and $dom(\alpha) \cap Y = \emptyset$, where $Y = vars(G_2) - vars(G_1)$. By Lemma 5.21, there is a substitution γ such that $G_i\theta\eta = G_i\sigma\gamma$, for $i = 1, 2$. Thus, by the definition of proof, G has a proof of rank $k+1$ with answer σ. ∎

Theorem 6.22 (Completeness) *Let P be a normalized program and G a normalized goal. Suppose θ is a substitution for the variables of G that has no generic*

constants. Then,

1. *If* $(\Phi_P \uparrow \omega)\ (G\theta) \succeq \texttt{true}$, *then* G *has a proof with answer* σ, *such that* $G\theta = G\sigma\gamma$, *for some substitution* γ;

2. *If* $(\Phi_P \uparrow \omega)\ (G\theta) \succeq \texttt{false}$, *then* G *has a refutation with answer* σ, *such that* $G\theta = G\sigma\gamma$, *for some substitution* γ.

Proof: We prove that the conclusions of both parts of the theorem hold by induction on $n < \omega$, where $(\Phi_P \uparrow n)(G\theta) \succeq \texttt{true}$ or $(\Phi_P \uparrow n)(G\theta) \succeq \texttt{false}$.

Basis: $(n = 0)$

Suppose that $(\Phi_P \uparrow 0)(G\theta) = I_0(G\theta) \succeq \texttt{true}$. We prove the result by a secondary induction on the structure of G:

1. *G is an atom:* Then for any ground substitution δ, $I_0(G\theta\delta) = \texttt{true}$, and hence, $G\theta\delta = G = \texttt{true}$. So G has a proof of rank 0 with answer $\sigma = \varepsilon$. Clearly, $\theta = \varepsilon\theta$.

2. *G is $\neg G'$:* Then $I_0(G'\theta) \succeq \texttt{false}$, and so by the secondary inductive hypothesis G' has a refutation with answer σ, such that $G\theta = G\sigma\gamma$, for some substitution γ. Hence, $G = \neg G'$ has a proof with answer σ.

3. *G is $G_1 \otimes G_2$:* Since $I_0(G_1 \otimes G_2)\theta \succeq \texttt{true}$, we have $I_0(G_1\theta\delta \otimes G_2\theta\delta) \succeq \texttt{true}$ for every ground substitution δ. Therefore, for $i = 1, 2$, we have $I_0(G_i\theta\delta) \succeq \texttt{true}$ for every ground δ, and hence $I_0(G_i\theta) \succeq \texttt{true}$ for $i = 1, 2$. Then, by the secondary induction hypothesis, for $i = 1, 2$, G_i has a proof with answer σ_i such that $G_i\theta = G_i\sigma_i\gamma_i$, for some substitution γ_i. Let σ_i' be a variant of σ_i w.r.t. G_i, such that, for $i = 1, 2$, $vars(G_i\sigma_i')$ and hence also $vrange(\sigma_i')$ are disjoint from

$vars(G)$ and from $dom(\theta)$, and $dom(\sigma_i) \subseteq vars(G_i)$. Hence, by Lemma 5.20, $\sigma_1' \odot \sigma_2'$ exists, and there is a substitution γ such that $\theta = (\sigma_1' \odot \sigma_2')\gamma$. Let $\sigma = (\sigma_1' \odot \sigma_2')_G$. It follows that G has a proof with answer σ such that $G\theta = G\sigma\gamma$.

4. *G is $G_1 \vee G_2$:* Since $I_0(G_1 \vee G_2)\theta \succeq \texttt{true}$, we have $I_0(G_1\theta\delta \vee G_2\theta\delta) \succeq \texttt{true}$ for every ground substitution δ. Therefore, for each ground δ, $I_0(G_i\theta\delta) \succeq \texttt{true}$ for at least one $i = 1, 2$. Now, let δ be a gc-substitution for $G\theta$. Assume, without loss of generality, that $I_0(G_1\theta\delta) \succeq \texttt{true}$. Then $I_0(G_1\theta) \succeq \texttt{true}$ by Lemma 6.9. Hence, by the secondary induction hypothesis, G_1 has a proof with answer σ such that $G_1\theta = G_1\sigma\alpha$ for some substitution α. By the unique renaming assumption, σ and α can be chosen to satisfy $vrange(\sigma) \cap Y = \emptyset$ and $dom(\alpha) \cap Y = \emptyset$, where $Y = vars(G_2) - vars(G_1)$. By Lemma 5.21, there is a substitution γ such that $G\theta = G\sigma\gamma$ and, by the definition of proof, σ is a proof of G.

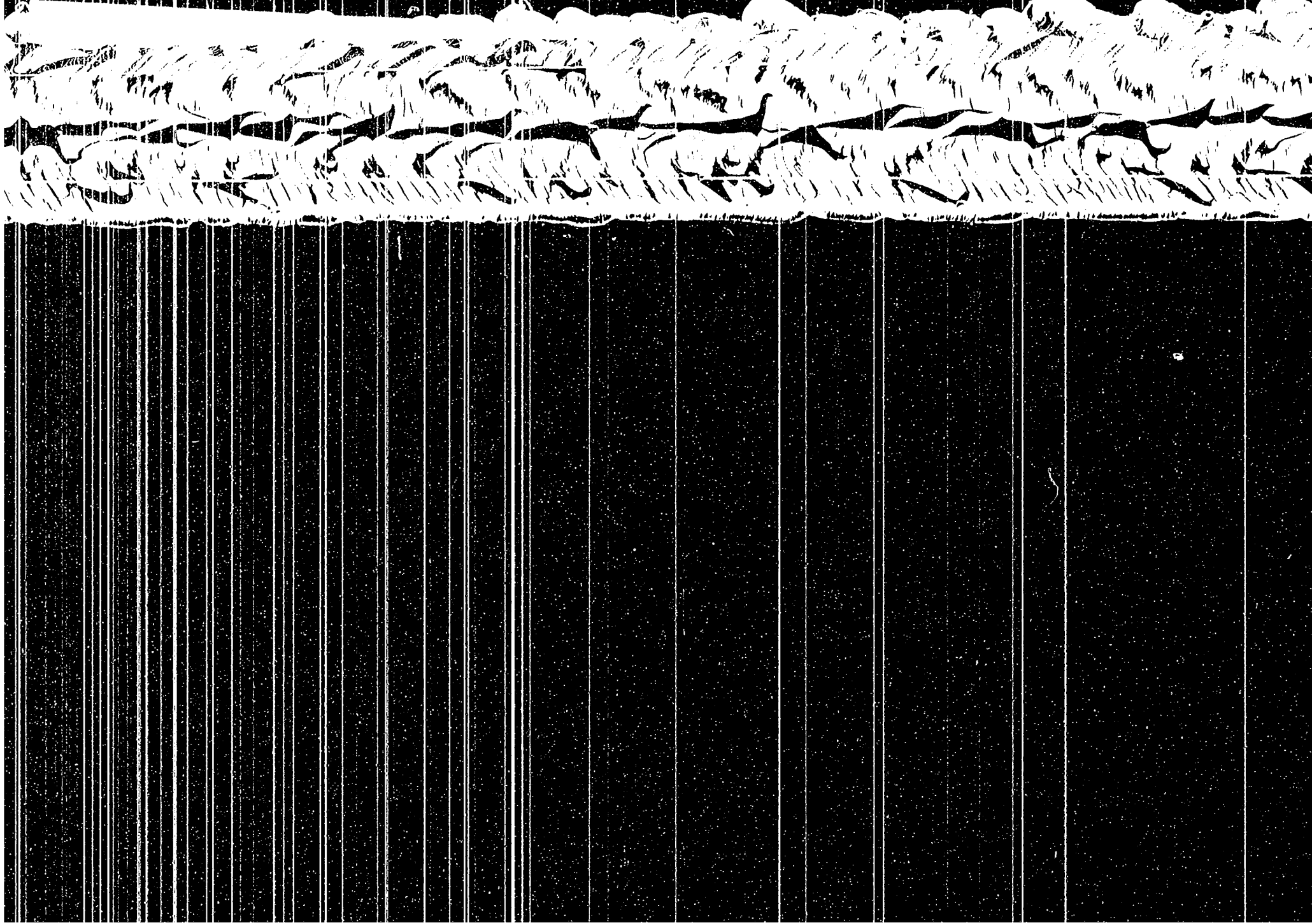
5. *G is $G_1 \wedge G_2$:* This case is similar to the case for $G = G_1 \otimes G_2$.

The case where $I_0(G\theta) \succeq \texttt{false}$ is proved in a similar manner.

Induction: Assume the result holds for the nth iteration of Φ_P, and that $(\Phi_P \uparrow n+1)(G\theta) \succeq \texttt{true}$. The result is, again, proved by a secondary induction on the structure of G.

1. *G is an atom A:* By assumption we have

$$(\Phi_P \uparrow n+1)(A\theta) = [\Phi_P(\Phi_P \uparrow n)](A\theta) \succeq \texttt{true}.$$



Hence, for every ground substitution δ, $[\Phi_P(\Phi_P \uparrow n)\,](A\theta\delta) \succeq \texttt{true}$, which implies:

$$\sum\{(\Phi_P \uparrow n)\,(F\eta) \mid A' \leftarrow F \in P \text{ and } \eta = mgu(A\theta\delta, A')\} \succeq \texttt{true}.$$

Let δ be a gc-substitution for $A\theta$. It follows that P must contain a clause $A' \leftarrow F$ such that $\eta = mgu(A\theta\delta, A')$ and $(\Phi_P \uparrow n)\,(F\eta) \succeq \texttt{true}$. Neither $A\theta$ nor A' contains a generic constant, and, by the unique renaming assumption, $vars(A\theta) \cap (vars(A') \cup vars(F)) = \emptyset$. Thus, by Lemma 6.3, $A\theta$ and A' are unifiable and, if $\mu = mgu(A\theta, A')$, then $\mu_{A\theta}$ is injective and variable-pure, and $\delta\eta = \mu\tau$, where τ is a gc-substitution for $A'\mu$. Thus, since $dom(\delta) \cap vars(F) = \emptyset$ (because $dom(\delta) = vars(G)$), we get $F\mu\tau = F\delta\eta = F\eta$. So $(\Phi_P \uparrow n)(F\mu\tau) \succeq \texttt{true}$, and hence $(\Phi_P \uparrow n)(F\mu) \succeq \texttt{true}$ by Lemma 6.9. By the induction hypothesis, $F\mu$ has a proof with answer ε', where ε' is injective and variable-pure. Hence $A\theta$ has a proof with answer $(\mu\varepsilon')_{A\theta}$; note that $(\mu\varepsilon')_{A\theta}$ is injective and variable-pure since $\mu_{A\theta}$ and ε' both are. By the Lifting Lemma we conclude that A has a proof with answer σ such that $A\theta\mu\varepsilon' = A\sigma\rho$ for some substitution ρ. Since $(\mu\varepsilon')_{A\theta}$ is injective and variable-pure, there is a μ' such that $A\theta\mu\varepsilon'\mu' = A\theta$. Let $\gamma = \rho\mu'$. Then $A\theta = A\sigma\gamma$ as required.

2. *G is $\neg G'$:* Then $(\Phi_P \uparrow n+1)\,(\neg G'\theta) = \neg(\Phi_P \uparrow n+1)\,(G'\theta) \succeq \texttt{true}$. So, we have $(\Phi_P \uparrow n+1)\,(G'\theta) \succeq \texttt{false}$. Now, by the secondary inductive hypothesis, G' has a refutation with answer σ such that $G\theta = G\sigma\gamma$, for some substitution γ. Hence, $G = \neg G'$ has a proof with answer σ.

3. *G is $G_1 \otimes G_2$ or $G_1 \wedge G_2$ or $G_1 \vee G_2$:* Each of these three cases are proved in the same manner as the basis ($n = 0$), where I_0 is replaced with $(\Phi_P \uparrow n+1)$.

The result for the case where $(\Phi_P \uparrow n+1)\ (G\theta) \succeq \texttt{false}$ is proved in a similar manner. Finally, we observe that, since the induction establishes the result for all $n < \omega$, it also holds for ω. ∎

6.5 Incorporating Closed World Assumption

In this section we will present a modified version of Negation as Partial Failure which incorporates a version of the Closed World (CW) Assumption. The CW Assumption is essentially an inference rule stating that if a ground atom A is not a logical consequence of a program, then infer $\neg A$. Thus, the CW Assumption provides a mechanism for deducing negative information. This inference rule, introduced by Reiter [43], is often a natural rule in the context of database applications.

We incorporate the CW Assumption into our logic by modifying the fixpoint semantics of Section 6. Atoms which do not unify with the head of any clause will now be interpreted as `false` rather than $\bot$. The closed world procedural semantics is the obtained by modifying the procedural semantics of Section 6 to compensate for the CW Assumption in the new fixpoint semantics. This is accomplished by implicitly adding to the program a clause of the form $A \leftarrow \texttt{false}$ for each atom A which does not unify with the head of any clause in the original program. In this way, a subgoal which does not unify with the head of any clause in the original program will have a refutation with respect to the extended program. We will establish soundness and completeness results for the new notions of cw-proof and cw-refutation that match almost exactly the corresponding results of Section 6. This method of modeling the cw-fixpoint semantics procedurally is however not entirely satisfactory since, as we

shall see, it effectively involves computation over an infinite program. It is an interesting open problem whether or not there exists another cw-procedural semantics that has both the desirable computational and completeness properties of the procedural semantics of Section 6.

We begin by describing in detail the changes that are needed in our fixpoint semantics to incorporate the CW Assumption. The primary difference with our original fixpoint semantics is in the definition of the initial interpretation.

Definition 6.23 The *initial interpretation* I_0^{cw} of a program P is defined as follows. For any atom $A \in B_P$:

$$I_0^{cw}(A) = \begin{cases} \texttt{true} & \text{if } A = \texttt{true}; \\ \texttt{false} & \text{if } A \neq \texttt{true} \text{ and } A \text{ does not unify with the head of} \\ & \text{any clause in } P; \\ \bot & \text{otherwise.} \end{cases}$$

Now, with each program P, we associate a new semantic operator denoted by Φ_P^{cw} which is defined as follows.

Definition 6.24 Let P be a program and let $A \in B_P$. The *closed world semantic operator* Φ_P^{cw} is a function mapping interpretations to interpretations, defined as follows:

$$\Phi_P^{cw}(I)(A) = \begin{cases} \texttt{true} & \text{if } A = \texttt{true}; \\ \texttt{false} & \text{if } A \neq \texttt{true} \text{ and } A \text{ does} \\ & \text{not unify with the} \\ & \text{head of any clause in } P; \\ \sum\{I(G\sigma) \mid A' \leftarrow G \in P,\ A = A'\sigma\} & \text{otherwise.} \end{cases}$$

Definition 6.25 The *upward iteration* of Φ_P^{cw} is defined as follows:

$$\Phi_P^{cw} \uparrow \alpha = \begin{cases} I_0^{cw} & \text{if } \alpha = 0; \\ \Phi_P^{cw}(\Phi_P^{cw} \uparrow (\alpha - 1)) & \text{if } \alpha \text{ is a successor ordinal;} \\ \sum\{\Phi_P^{cw} \uparrow \beta \mid \beta < \alpha\} & \text{if } \alpha \text{ is a limit ordinal.} \end{cases}$$

We shall see below that cw-fixpoint semantics can be reduced to the fixpoint semantics of Section 6. For this purpose we extend the notions of semantic operator and upward iteration (as defined originally for normalized programs) to apply to infinite normalized programs. It is easy to verify that such an extension causes no semantic difficulty.

We now define a special class of programs which provide the technical tool for specifying the relationship between our original semantics and the closed world semantics of normalized programs.

Definition 6.26 Let P be a normalized program. Then the *normalized extension* of P, denoted P^+, is a possibly infinite program defined as follows.

1. For every clause $C \in P$, $C \in P^+$.

2. For every atom $A \notin \{\texttt{true},\texttt{false}\}$ which does not unify with the head of any clause in P, the clause $A \leftarrow \texttt{false} \in P^+$.

The following lemma is an immediate consequence of the definitions.

Lemma 6.27 *Let P be a (normalized) program and let P^+ be its normalized extension. Then for every interpretation I and for every $A \in B_P$ such that A unifies with the head of some clause in P,*

$$\Phi_P(I)(A) = \Phi_{P^+}(I)(A) = \Phi_P^{cw}(I)(A).$$

The next lemma establishes the relationship between the fixpoint semantics for extended normalized programs and the cw-fixpoint semantics for normalized programs.

Lemma 6.28 *Let P be a normalized program and G a normalized goal. Then:*

$$(\Phi_P^{cw} \uparrow \omega)(G) = (\Phi_{P^+} \uparrow \omega)(G).$$

Proof: We will use induction to show that for every $n < \omega$ and every formula G,

$$(\Phi_P^{cw} \uparrow n)(G) = (\Phi_{P^+} \uparrow n')(G),$$

where $n' \in \{n, n+1\}$. By Lemma 6.5, it suffices to show this equality holds for every $A \in B_P$. Let $n = 0$. If $A \in \{\texttt{true}, \texttt{false}\}$, then the equality follows immediately with $n' = 0$. If $A \notin \{\texttt{true}, \texttt{false}\}$ and A does not unify with the head of any clause in P, then on the one hand $(\Phi_P^{cw} \uparrow 0)(A) = \texttt{false}$, and on the other hand $A \leftarrow \texttt{false} \in P^+$ and so

$$\begin{aligned}
(\Phi_{P^+} \uparrow 1)(A) &= \Phi_{P^+}(\Phi_{P^+} \uparrow 0)(A) \\
&= \sum\{I_0(G\gamma) \mid A' \leftarrow G \in P^+, A = A'\gamma\} \\
&= \sum\{\texttt{false}\} \quad (\text{since } G = \texttt{false}) \\
&= \texttt{false}.
\end{aligned}$$

If A unifies with the head of some clause in P, then, by the definition of Φ_P^{cw} and Φ_{P^+},

$$(\Phi_P^{cw} \uparrow 0)(A) = \bot = (\Phi_{P^+} \uparrow 0)(A).$$

Now suppose that the result holds for all integers smaller than n and for all formulas G. To show that the result holds for n, by Lemma 6.5, it suffices to argue

that it holds for all $A \in B_P$. The cases when $A \in \{\texttt{true}, \texttt{false}\}$ follow immediately from the definitions. If $A \notin \{\texttt{true}, \texttt{false}\}$ and A does not unify with the head of any clause in P, then we have $(\Phi_P^{cw} \uparrow n)(A) = \texttt{false}$. On the other hand,

$$\begin{aligned}
(\Phi_{P+} \uparrow n+1)(A) &= \Phi_{P+}(\Phi_{P+} \uparrow n)(A) \\
&= \Phi_{P+}(\Phi_P^{cw} \uparrow n'')(A) \qquad (n'' \in \{n, n-1\}) \\
&= \sum\{(\Phi_P^{cw} \uparrow n'')(G\sigma) \mid A' \leftarrow G \in P^+, A = A'\sigma\} \\
&= \sum\{\texttt{false}\} \\
&= \texttt{false}.
\end{aligned}$$

Finally, suppose that A unifies with the head of some clause in P. Then

$$\begin{aligned}
(\Phi_P^{cw} \uparrow n)(A) &= \Phi_P^{cw}(\Phi_P^{cw} \uparrow n-1)(A) \\
&= \Phi_P^{cw}(\Phi_{P+} \uparrow n'')(A) \qquad (n'' \in \{n, n-1\}) \\
&= \Phi_{P+}(\Phi_{P+} \uparrow n'')(A) \qquad \text{(by Lemma 6.27)} \\
&= (\Phi_{P+} \uparrow n')(A) \qquad (n' \in \{n, n+1\}). \blacksquare
\end{aligned}$$

Definition 6.29 Let P be a normalized program and G a normalized goal.

1. G has a *closed world proof (cw-proof) of rank 0 with answer* θ with respect to P if $G = \texttt{true}$ and θ is the identity substitution ε. G has a *closed world refutation (cw-refutation) of rank 0 with answer* θ with respect to P if

   (a) $G = \texttt{false}$ and $\theta = \varepsilon$, or

   (b) $G\theta$ is an atom which does not unify with the head of any clause in P.

2. The definitions of *cw-proofs* and *cw-refutations of rank* $k+1$ are identical to those in parts 2 and 3 of Definition 6.18.

The most significant departure from the earlier notions of proof and refutation is in the way that a cw-refutation of rank 0 is defined. We now have a refutation, not only when a subgoal `false` is reached, but also when we reach a subgoal which has substitution instance which does not unify with the head of any clause in the program; the answers in this case are all substitutions, possibly infinitely many, that result in nonunification. Here the resemblance to the traditional notion of Negation as Failure should be clear.

The following lemma specifies the relationship between the procedural semantics for extended normalized programs and the closed world procedural semantics for normalized programs. The proof is by straightforward induction and we omit it here.

Lemma 6.30 *Let P be a normalized program and G a normalized goal. Then G has a cw-proof (respectively, a cw-refutation) with answer σ, with respect to P, if and only if G has a proof (respectively, a refutation) with answer σ, with respect to P^+.*

Now we have all we need in order to prove the Soundness and Completeness theorems for the Negation as Partial Failure with the Closed World Assumption.

Theorem 6.31 (Closed World Soundness) *Let P be a normalized program, G a normalized goal, and θ a substitution for the variables of G.*

1. *If G has a cw-proof with answer θ with respect to P, then $(\Phi_P^{cw} \uparrow \omega)(G\theta) \succeq$* `true`.

2. *If G has a cw-refutation with answer θ with respect to P, then $(\Phi_P^{cw} \uparrow \omega)(G\theta) \succeq$* `false`.

Proof: Suppose that G has a cw-proof (respectively, a cw-refutation) with answer θ with respect to P. Then, by Lemma 6.30, G has a proof (respectively, a refutation) with answer θ, with respect to P^+. By the Soundness Theorem of Section 6 (now applied to possibly infinite programs), we conclude that $(\Phi_{P+} \uparrow \omega)(G\theta) \succeq$ `true` (respectively, `false`). Hence, by Lemma 6.28, we have $(\Phi_P^{cw} \uparrow \omega)(G\theta) \succeq$ `true` (respectively, `false`). ∎

Theorem 6.32 (Closed World Completeness) *Let P be a normalized program, G a normalized goal, and θ a substitution for the variables of G.*

1. *If $(\Phi_P^{cw} \uparrow \omega)(G\theta) \succeq$ `true`, then G has a cw-proof with answer σ with respect to P such that $G\theta = G\sigma\gamma$, for some substitution γ.*

2. *If $(\Phi_P^{cw} \uparrow \omega)(G\theta) \succeq$ `false`, then G has a cw-refutation with answer σ with respect to P such that $G\theta = G\sigma\gamma$, for some substitution γ.*

Proof: Suppose that $(\Phi_P^{cw} \uparrow \omega)(G\theta) \succeq$ `true` (respectively, `false`). Then, by Lemma 6.28, we have $(\Phi_{P+} \uparrow \omega)(G\theta) \succeq$ `true` (respectively, `false`). Hence, by the Completeness Theorem of Section 6 (now applied to possibly infinite normalized programs) G has a proof (respectively, a refutation) with answer σ with respect to P^+, such that $G\theta = G\sigma\gamma$, for some substitution γ. Now, by Lemma 6.30, we conclude that G has a cw-proof (respectively, a cw-refutation) of rank k with answer σ with respect to P, such that $G\theta = G\sigma\gamma$, for some substitution γ. ∎

CHAPTER 7. GENERALIZED SEMANTICS FOR KNOWLEDGE-BASED PROGRAMS

7.1 Logic Programming Semantics and Distributive Bilattices

We will now extend the logic programming semantics presented in the previous sections for the bilattice $\mathcal{FOUR}$ to arbitrary bilattices. We first present a generalized version of the fixpoint semantics presented earlier for knowledge-based multi-valued logics based on bilattices, and then we give the corresponding generalized procedural semantics for logic programs followed by the soundness and completeness results showing the correspondence between the fixpoint and the procedural semantics.

Finally, we present an alternate procedural semantics based on the join irreducible elements of a large class of bilattices which satisfy certain finiteness conditions (specifically, those that have the descending chain property in the knowledge ordering). This allows us to restrict our attention to a relatively small subset of special truth values. The revised procedural semantics will eliminate the need for extraneous searches during the query evaluation process, which as we shall explain below, is a problem of the original generalized semantics. As it turns out, the SLDPF-resolution which provided the operational semantics for logic programs based on $\mathcal{FOUR}$, is a special case of the generalized join-irreducible procedural semantics. The main results of this chapter will establish the connection between the two procedural semantics.

The syntax of programs is essentially identical to that presented in the previous chapter except now we extend the notion of an atom to include any element of the underlying bilattice except $\bot$ and $\top$.

7.1.1 Fixpoint Semantics

The generalized fixpoint semantics for logic programs over arbitrary bilattices are defined in a similar manner as in the special case of the bilattice $\mathcal{FOUR}$. For completeness, we repeat some of the definitions given previously along with the new generalized definitions of the semantic and fixpoint operators.

Definition 7.1 Let $\mathcal{B}$ be a bilattice.

1. An *interpretation* for a program P is a mapping $I : B_P \rightarrow \mathcal{B}$.

2. We extend the interpretation I to ground formulas as follows:

$$\begin{aligned} I(\neg A) &= \neg I(A); \\ I(A_1 \oplus A_2) &= I(A_1) \oplus I(A_2); \\ I(A_1 \otimes A_2) &= I(A_1) \otimes I(A_2); \\ I(A_1 \wedge A_2) &= I(A_1) \wedge I(A_2); \\ I(A_1 \vee A_2) &= I(A_1) \vee I(A_2). \end{aligned}$$

3. We further extend the interpretation I to non-ground formulas. For a non-ground formula G:

$$I(G) = \prod\{I(G\sigma) \mid \sigma \text{ is a ground substitution for the variables of } G\}.$$

Definition 7.2 The *initial interpretation* I_0 of a program P is defined as follows. For any atom $A \in B_P$:

$$I_0(A) = \begin{cases} A & \text{if } A \in \mathcal{B} - \{\bot, \top\} \\ \bot & \text{otherwise.} \end{cases}$$

Note that for any atomic formula G and any $c \in \mathcal{B} - \{\bot, \top\}$, if $I_0(G) \geq_k c$, then $G = c$.

Now we can associate a semantic operator with each program.

Definition 7.3 Let $\mathcal{B}$ be a distributive bilattice. Let P be a program and let $A \in B_P$. The *semantic operator* Φ_P is a function mapping interpretations to interpretations, defined as follows:

$$\Phi_P(I)(A) = \begin{cases} A & \text{if } A \in \mathcal{B} - \{\bot, \top\} \\ \sum\{I(G\sigma) \mid A' \leftarrow G \in P \text{ and } A = A'\sigma\} & \text{otherwise.} \end{cases}$$

Pointwise partial orderings are also defined on interpretations in the following manner:

1. $I_1 \leq_k I_2$ if $I_1(A) \leq_k I_2(A)$, for every ground atom $A \in B_P$.

2. $I_1 \leq_t I_2$ if $I_1(A) \leq_t I_2(A)$, for every ground atom $A \in B_P$.

Using this pointwise ordering, the space of interpretations itself becomes a distributive bilattice. Furthermore, the Φ_P operator is monotonic with respect to the knowledge ordering [18]:

$$I_1 \leq_k I_2 \implies \Phi_P(I_1) \leq_k \Phi_P(I_2),$$

and hence, by the Knaster-Tarski theorem [53], Φ_P has a least fixpoint. It is this least fixpoint which provides the denotational meaning of the program P. In order

to approximate the least fixpoint of the operator Φ_P, we use the following notion of upward iteration.

Definition 7.4 The *upward iteration* of Φ_P is defined as follows:

$$\Phi_P \uparrow \alpha = \begin{cases} I_0 & \text{if } \alpha = 0 \\ \Phi_P(\Phi_P \uparrow (\alpha - 1)) & \text{if } \alpha \text{ is a successor ordinal} \\ \sum\{\Phi_P \uparrow \beta \mid \beta < \alpha\} & \text{if } \alpha \text{ is a limit ordinal} \end{cases}$$

The smallest ordinal at which this sequence gives the least fixpoint of Φ_P is called the *closure ordinal*. In $\mathcal{FOUR}$ and in fact in any bilattice which satisfies the infinitary distributivity conditions, Φ_P is continuous and its closure ordinal is at most ω [18].

It can be easily verified that the results proved for the fixpoint semantics based on $\mathcal{FOUR}$ in Chapter 6, also hold for the generalized case. We shall use these results in the sequel.

7.1.2 Generalized Procedural Semantics

In this section we present a generalized procedural semantics for logic programs based on arbitrary bilattices. As in the case of $\mathcal{FOUR}$, we use a resolution-based procedural semantics which will allow us to start with any formula as a goal and within a uniform framework derive both negative and positive information about that goal representing evidence for or against its truth. In the context of the bilattice $\mathcal{FOUR}$ this means that if the derivation from a goal G leads to success, then G is at least `true`, and if it leads to failure, then G is at least `false`. More generally, if a derivation from G leads to a constant b where b is a truth value in the underlying bilattice, we say that G has at least the value b.

This generalized procedural model is an extension of the well-known operational semantics in SLDNF-resolution. We call our generalized procedural semantics *SLDK-resolution* (K stands for Knowledge-based). In our derivation trees, each branch of a subtree with the root node A, where A is an atom, corresponds to a clause whose head unifies with A. Each such clause is seen as contributing to the information the system contains about the truth or falsity of A. SLDK-resolution extends the treatment of $\neg$ (under SLDNF-resolution) to the operators $\wedge$, $\vee$, $\ominus$, and $\otimes$. More precisely, if during the derivation a subgoal is reached which is a formula containing one of these operators, then an attempt is made to establish appropriate derivations for the two operands based on the way they act on the elements of the bilattice. S-unifiers ensure that the answers obtained from the derivation trees of each operand will not contradict each other once they are finally applied to the formula itself. Furthermore, as in the case of the case of SLDPF-resolution which was based on $\mathcal{FOUR}$, we interpret free variables in the body of a clause as being quantified by $\sum$, which is its own dual under negation. Thus our procedural semantics remains sound even in the presence of nonground negative subgoals. We now present the details formally in the following definitions.

Definition 7.5 An *SLDK-tree for* $P \cup \{A\}$, where P is a normalized program and A is an atom, is a (possibly infinite) tree satisfying the following conditions:

1. The root of the tree is A.

2. Let G be a nonleaf node. Then G is an atom and for each clause $G' \leftarrow G'' \in P$, if G and G' are unifiable, then the node has a child $G''\gamma$, where $\gamma = mgu(G, G')$. We say that γ is the *substitution associated with the edge* between G and $G''\gamma$.

3. Let G be a leaf node. Then either G is an atom which does not unify with the head of any clause in P (in particular, G may be any $b \in \mathcal{B} - \{\bot, \top\}$), or G is a complex formula.

Definition 7.6 Let $\mathcal{B}$ be a distributive bilattice and $b \in \mathcal{B}$. Let P be a normalized program and G a normalized goal. Then

1. G has a *b-derivation of rank 0 with answer* θ if $G = c$, where $c \in \mathcal{B} - \{\bot, \top\}$, $b \preceq c$, and θ is the identity substitution ε.

2. G has a *b-derivation of rank* $k+1$ *with answer* θ, if:

   (a) G is an atom A, and $P \cup \{A\}$ has an SLDK-tree with at least one leaf node G' which has a b-derivation of rank k with answer θ' such that $\theta = (\sigma_1 \cdots \sigma_n \theta')_G$, where $\sigma_1, \cdots, \sigma_n$ are the substitutions associated with edges along the path from G to G'.

   (b) G is $\neg G'$, and G' has a $\neg b$-derivation of rank k, with answer θ.

   (c) G is $G_1 \square G_2$, where $\square \in \{\otimes, \vee, \wedge\}$, and G_1 has a c-derivation of rank k_1 and G_2 has a d-derivation of rank k_2 with answers θ_1 and θ_2, respectively, $k = max(k_1, k_2)$, $\theta = (\theta_1' \odot \theta_2')_G$, where θ_i' is a variant of θ_i w.r.t. G_i $(i = 1, 2)$, and $b = c \square d$.

Definition 7.7 Let $\mathcal{B}$ be a distributive bilattice and $b \in \mathcal{B}$. Let P be a normalized program and G a normalized goal. Then G has a *b-proof with answer* θ, if there exists an $n \geq 1$ and $b_1, \cdots, b_n \in \mathcal{B}$, such that

1. G has a b_i-derivation of rank k_i with answer θ_i $(i = 1, 2, \cdots, n)$,

2. $b = b_1 \oplus b_2 \oplus \cdots \oplus b_n$, and

3. $\theta = (\odot\{\theta_i \mid 1 \leq i \leq n\})_G$.

Note that, in the above definition, we distiguish between the notions of a b-derivation and b-proof. This is done in order to emphasize the fact that for a given goal there may be many independent c-derivations (for any $c \in \mathcal{B}$), and all of these c-derivations contribute to the knowledge the system has about the goal. Thus, the information provided by various c-derivations is collected by taking the knowledge join of the c's.

7.2 Generalized Soundness and Completeness Results

We will now show that the generalized semantics presented in the previous section is sound and complete. The proofs are similar to those for the four-valued semantics of Chapter 6.

Lemma 7.8 *Let $\mathcal{B}$ be a distributive bilattice and $b \in \mathcal{B}$. Let P be a normalized program, G a normalized goal, and θ a substitution for the variables of G. If G has a b-derivation of rank k with answer θ, for some $k \geq 0$, then $(\Phi_P \uparrow \omega)(G\theta) \succeq b$.*

Proof: (By induction on k)

Basis: ($k = 0$) Suppose G has a b-derivation of rank 0 with answer θ. Then $G = c$, where $b \preceq c \in \mathcal{B}$, and $\theta = \varepsilon$. Now, $I_0(G\theta) = I_0(c) = c \succeq b$. Since Φ_P is monotonic, $(\Phi_P \uparrow \omega)(G\theta) = (\Phi_P \uparrow \omega)(c) \succeq b$.

Induction: Assume the result holds for b-derivations of rank k, and suppose first that G has a b-derivation of rank $k+1$ with answer θ. We have the following cases to consider:

1. *G is an atom:* Then $P \cup \{G\}$ has an SLDK-tree with at least one leafnode F which has a b-derivation of rank k with answer θ', such that $\theta = (\sigma_1 \cdots \sigma_n \theta')_G$, where $\sigma_1, \cdots, \sigma_n$ are the mgu's associated with the edges along the path from G to F in the tree. Now, the result is proved by performing a secondary induction on the length n of this path.

   The result is vacuously true if $n = 0$, since in this case atoms with SLDK-trees do not have b-derivations of positive rank (the SLDK-tree of an atom G is of height 0 whenever $G \in \mathcal{B}$ or if it does not unify with the head of any clause in the program, and in either case, any b-derivation of G is of rank 0).

   Now, suppose the result holds for SLDK-trees with b-derivation branches of depth $n \geq 0$, and consider one of depth $n+1$. In this case, P must have a clause $C' \leftarrow C_1$ such that $\sigma_1 = mgu(G, C')$ and $G_1\sigma_1$ has an SLDK-tree with a b-derivation branch of depth n. Then, it is easy to see that $G_1\sigma_1$ has a b-derivation of rank $k+1$ with answer θ_1 such that $\theta = (\sigma_1\theta_1)_G$. By the secondary induction hypothesis, $(\Phi_P \uparrow \omega)(G_1\sigma_1\theta_1) \succeq b$. Then,

$$\begin{aligned}
(\Phi_P \uparrow \omega)(G\theta) &= (\Phi_P \uparrow \omega)(G\sigma_1\theta_1) && \text{since } G\theta = G\sigma_1\theta_1 \\
&= (\Phi_P \uparrow \omega)(G'\sigma_1\theta_1) && \text{since } \sigma_1 = mgu(G, G') \\
&\succeq (\Phi_P \uparrow \omega)(G_1\sigma_1\theta_1) && \text{by Lemma 6.16} \\
&\succeq b && \text{by the secondary ind. hyp.}
\end{aligned}$$

2. *G is* $\neg G'$*:* Then G' must have a $\neg b$-proof of rank k with answer θ. By the inductive hypothesis, $(\Phi_P \uparrow \omega)\,(G'\theta) \succeq \neg b$. Hence,

$$\begin{aligned}(\Phi_P \uparrow \omega)\,(G\theta) &= (\Phi_P \uparrow \omega)\,(\neg G'\theta) \\ &= \neg(\Phi_P \uparrow \omega)\,(G'\theta) \\ &\succeq b.\end{aligned}$$

3. *G is* $G_1 \square G_2$*, where* $\square \in \{\otimes, \vee, \wedge\}$ *:* G_1 has a c-derivation of rank k_1 and G_2 has a d-derivation of rank k_2 with answers θ_1 and θ_2, respectively, such that $\theta = (\theta_1' \odot \theta_2')_G$, $k = max(k_1, k_2)$, and $b = c \square d$, where θ_i' are variants of θ_i w.r.t. G $(i = 1, 2)$. Now

$$\begin{aligned}(\Phi_P \uparrow \omega)\,([G_1 \square G_2]\theta) &= (\Phi_P \uparrow \omega)\,(G_1\theta \square G_2\theta) \\ &\succeq (\Phi_P \uparrow \omega)\,(G_1\theta) \square (\Phi_P \uparrow \omega)\,(G_2\theta), && \text{by Lemma 6.12} \\ &= (\Phi_P \uparrow \omega)\,(G_1(\theta_1' \odot \theta_2')) \square (\Phi_P \uparrow \omega)\,(G_2(\theta_1' \odot \theta_2')) \\ &\succeq (\Phi_P \uparrow \omega)\,(G_1\theta_1') \square (\Phi_P \uparrow \omega)\,(G_2\theta_2'), && \text{by Corollary 6.14} \\ &\succeq (\Phi_P \uparrow \omega)\,(G_1\theta_1) \square (\Phi_P \uparrow \omega)\,(G_2\theta_2), && \text{by Lemma 6.11} \\ &\succeq c \square d, && \text{by inductive hypothesis and properties of } \square \\ &= b. \ \blacksquare\end{aligned}$$

Theorem 7.9 (Soundness) *Let* $\mathcal{B}$ *be a distributive bilattice and* $b \in \mathcal{B}$*. Let* P *be a normalized program,* G *a normalized goal, and* θ *a substitution for the variables of* G*. If* G *has a* b*-proof with answer* θ*, then* $(\Phi_P \uparrow \omega)(G\theta) \succeq b$*.*

Proof: Suppose that G has b_i-derivation of rank k_i with answer θ_i $(1 \leq i \leq n)$, such that $b = b_1 \oplus \cdots \oplus b_n$, and $\theta = (\odot\{\theta_1, \cdots, \theta_n\})_G$. Then by Lemma 7.8 we have

$(\Phi_P \uparrow \omega)\ (G\theta_i) \succeq b_i$, for $1 \leq i \leq n$. But, for each i, $G\theta$ is a substitution instance of $G\theta_i$ and hence, by Lemma 6.13, $(\Phi_P \uparrow \omega)\ (G\theta) \succeq (\Phi_P \uparrow \omega)(G\theta_i)$ for $1 \leq i \leq n$. Thus,

$$\begin{aligned}(\Phi_P \uparrow \omega)\ (G\theta) &\succeq \Sigma(\Phi_P \uparrow \omega)\ (G\theta_i) \\ &\succeq b_1 \oplus \cdots \oplus b_n \\ &= b. \ \blacksquare\end{aligned}$$

Next we present the completeness results for the generalized semantics. As in the four-valued case, we will use the following Lifting Lemma in the proof of the Completeness Theorem.

Lemma 7.10 (Lifting Lemma) *Let $\mathcal{B}$ be a distributive bilattice and $b \in \mathcal{B}$. Suppose that P is a normalized program, G a normalized goal, and θ a substitution without generic constants for the variables of G. Also, suppose $G\theta$ has a b-derivation of rank $k \geq 0$ with answer η, with respect to a program P. Then G has a b-derivation of rank k with answer σ, such that $G\theta\eta = G\sigma\gamma$, for some substitution γ.*

Proof: (By induction on k)

Basis: ($k = 0$) Suppose that $G\theta$ has a b-derivation of rank 0 with answer η. Then $G\theta = c$ where $c \in \mathcal{B} - \{\bot, \top\}$, and $b \preceq c$. Hence, $G = c$ and $\eta = \varepsilon$. But $G = c$ has a b-derivation of rank 0 with answer ε. Clearly, $\theta\eta = \theta\varepsilon = \theta = \varepsilon\theta$. Now, take $\sigma = \varepsilon$ and $\gamma = \theta$.

Induction: Assume the result holds for b-derivations of rank k. We will prove the induction for b-derivations of rank $k+1$. Suppose that $G\theta$ has a b-derivation of rank $k+1$ with answer η. We have to consider the following cases.

1. *G is an atom A*: Then $P \cup \{A\theta\}$ has an SLDK-tree with at least one leafnode F which has a b-derivation of rank k with answer ρ, such that $\eta = (\sigma_1 \cdots \sigma_n \rho)_{A\theta}$, where $\sigma_1, \cdots, \sigma_n$ are the mgu's associated with the edges along the path in the tree from $A\theta$ to F. The result is proved by a secondary induction on the length n of this path. The argument is essentially the same as that in the proof of Lifting Lemma in the four-valued case (see Lemma 6.21).

2. *G is* $\neg G'$*:* Then $G'\theta$ has a $\neg b$-derivation of rank k with answer η. By the inductive hypothesis, G' has a $\neg b$-derivation of rank k with answer σ, such that $G'\theta\eta = G'\sigma\gamma$, for some substitution γ. Hence, $G = \neg G'$ has a b-derivation of rank $k+1$ with answer σ, and $G\theta\eta = G\sigma\gamma$.

3. *G is* $G_1 \Box G_2$*, where* $\Box \in \{\otimes, \vee, \wedge\}$ *:* Then $G_1\theta \Box G_2\theta$ has a b-derivation of rank $k+1$. Hence, $G_1\theta$ has a c-derivation of rank k_1 with answer η_1, and $G_2\theta$ has a d-derivation of rank k_2 with answer η_2, for some $c, d \in \mathcal{B}$, such that $b = c \Box d$ and $k = max(k_1, k_2)$, and $\eta = (\eta_1' \odot \eta_2')_{G\theta}$, where η_i' are variants η_i w.r.t. G_i $(i = 1, 2)$. By the inductive hypothesis, G_1 has a c-derivation of rank k_1 with answer σ_1, such that $G_1\theta\eta_1 = G_1\sigma_1\gamma_1$, for some substitution γ_1, and G_2 has a d-derivation of rank k_2 with answer σ_2, such that $G_2\theta\eta_2 = G_2\sigma_2\gamma_2$, for some substitution γ_2. Let σ_i' be variants of answers σ_i w.r.t. G_i, $i = 1, 2$, such that $vrange(\sigma_i)$ is disjoint from $vars(G)$ and from $dom(\theta\eta_i')$, and $dom(\sigma_i') \subseteq vars(G_i)$. Hence, all the conditions of Lemma 5.20 are satisfied. It follows that, $\sigma_1' \odot \sigma_2'$ exists, and furthermore, there is a substitution γ such that $\theta(\eta_1' \odot \eta_2') = (\sigma_1' \odot \sigma_2')\gamma$. Let $\sigma = (\sigma_1' \odot \sigma_2')_G$. Then G has a b-derivation of

rank $k+1$ with answer σ such that $G\theta\eta = G\sigma\gamma$ ■.

Lemma 7.11 *Let $\mathcal{B}$ be a distributive bilattice and $b \in \mathcal{B}$. Let P be a normalized program and G a normalized goal. Suppose θ is a substitution for the variables of G without generic constants. If $(\Phi_P \uparrow \omega)\ (G\theta) \succeq b$, then, for some $k \geq 0$, G has a b-derivation of rank k with answer σ, such that $G\theta = G\sigma\gamma$, for some substitution γ.*

Proof: We prove the result by induction on $n < \omega$, where, for $b \neq \bot$, $(\Phi_P \uparrow n)(G\theta) \succeq b$. For $b = \bot$, the result is trivial.

Basis: $(n = 0)$ First suppose that $(\Phi_P \uparrow 0)(G\theta) = I_0(G\theta) \succeq b$. We prove the result by a secondary induction on the structure of G:

1. *G is an atom:* Then since $I_0(G\theta) \succeq b$, for any ground substitution δ, $I_0(G\theta\delta) \succeq b$. Hence, it must be the case that $G\theta\delta = G = b$. Then G has a b-derivation of rank 0 with answer $\sigma = \varepsilon$. Clearly, $\theta = \varepsilon\theta$.

2. *G is $\neg G'$:* Then $I_0(G'\theta) \succeq \neg b$. So by the secondary inductive hypothesis G' has a $\neg b$-derivation of rank $k \geq 0$, with answer σ, such that $G'\theta = G'\sigma\gamma$, for some substitution γ. Hence, $G = \neg G'$ has a b-derivation of rank $k+1$ with answer σ and $G\theta = G\sigma\gamma$.

3. *G is $G_1 \square G_2$, where $\square \in \{\otimes, \vee, \wedge\}$:* Since $I_0(G_1 \square G_2)\theta \succeq b$, for every ground substitution δ, we have that $I_0(G_1\theta\delta \square G_2\theta\delta) \succeq b$. Therefore, $I_0(G_1\theta\delta) \succeq b_1$, and $I_0(G_2\theta\delta) \succeq b_2$, for some $b_1, b_2 \in \mathcal{B}$, such that $b_1 \square b_2 = b$. Then, by the secondary induction hypothesis, for $i = 1, 2$, G_i has a b_i-derivation with answer σ_i such that $G_i\theta = G_i\sigma_i\gamma_i$, for some substitution γ_i. Let σ_i' be a variant of σ_i

w.r.t G_i $(i = 1, 2)$, such that $vars(G_i\sigma_i')$ and hence also $vrange(\sigma_i')$ are disjoint from $vars(G)$ and from $dom(\theta)$. Hence, by Lemma 5.20, $\sigma_1' \odot \sigma_2'$ exists, and there is a substitution γ such that $\theta = (\sigma_1' \odot \sigma_2')\gamma$. Let $\sigma = (\sigma_1' \odot \sigma_2')_G$. It follows that G has a b-derivation with answer σ such that $G\theta = G\sigma\gamma$.

Induction: Assume that the implication holds for the nth iteration of Φ_P. Now, suppose that $(\Phi_P \uparrow n+1)\ (G\theta) \succeq b$. The result is proved by a secondary induction on the structure of G:

1. *G is an atom A:* By assumption we have:

$$(\Phi_P \uparrow n+1)\ (A\theta) = [\Phi_P(\Phi_P \uparrow n)\ ]\ (A\theta) \succeq b.$$

Hence, for every ground substitution δ, $[\Phi_P(\Phi_P \uparrow n)\ ]\ \ (A\theta\delta) \succeq b$. Then we have:

$$\sum\{(\Phi_P \uparrow n)\ (F\eta) \mid A' \leftarrow F \in P \text{ and } \eta = mgu(A\theta\delta, A')\} \succeq b.$$

Let δ be a gc-substitution for $A\theta$. It follows that P must contain a clause $A' \leftarrow F$ such that $\eta = mgu(A\theta\delta, A')$ and $(\Phi_P \uparrow n)\ (F\eta) \succeq b$. Neither $A\theta$ nor A' contains a generic constant, and, by the unique renaming assumption, $vars(A\theta) \cap (vars(A') \cup vars(F)) = \emptyset$. Thus, by Lemma 6.3, $A\theta$ and A' are unifiable, and, if $\mu = mgu(A\theta, A')$, then $\mu_{A\theta}$ is injective and variable-pure, and $\delta\eta = \mu\tau$, where τ is a gc-substitution for $A'\mu$. Thus, since $dom(\delta) \cap vars(F) = \emptyset$ (because $dom(\delta) = vars(G)$), we get $F\mu\tau = F\delta\eta = F\eta$. So $(\Phi_P \uparrow n)(F\mu\tau) \succeq b$, and hence $(\Phi_P \uparrow n)(F\mu) \succeq b$ by Lemma 6.9. By the induction hypothesis, $F\mu$ has a b-derivation with answer ε', where ε' is injective and variable-pure. Hence $A\theta$ has a b-derivation with answer $(\mu\varepsilon')_{A\theta}$ which is injective and variable-pure

since $\mu_{A\theta}$ and ε' both are. By the Lifting Lemma we conclude that A has a b-derivation with answer σ such that $A\theta\mu\varepsilon' = A\sigma\rho$ for some substitution ρ. Since $(\mu\varepsilon')_{A\theta}$ is injective and variable-pure, there is a μ' such that $A\theta\mu\varepsilon'\mu' = A\theta$. Let $\gamma = \rho\mu'$. Then $A\theta = A\sigma\gamma$ as required.

2. *G is $\neg G'$:* Then $(\Phi_P \uparrow n+1)\ (\neg G'\theta) = \neg(\Phi_P \uparrow n+1)\ (G'\theta) \succeq b$. So, we have $(\Phi_P \uparrow n+1)\ (G'\theta) \succeq \neg b$. Now, by the secondary inductive hypothesis, G' has a $\neg b$-derivation of rank k, for some $k \geq 0$, with answer σ, such that $G'\theta = G'\sigma\gamma$, for some substitution γ. Hence, $G = \neg G'$ has a b-derivation of rank $k+1$ with answer σ, and $G\theta = G\sigma\gamma$.

3. *G is $G_1 \square G_2$, where $\square \in \{\otimes, \wedge, \vee\}$:* The derivation of this case is similar to that of the basis case (where $n = 0$), except we replace I_0 with $(\Phi_P \uparrow n+1)$.

Finally, we observe that, since the induction establishes the result for all $n < \omega$, it also holds for ω. ∎

Theorem 7.12 (Completeness) *Let $\mathcal{B}$ be a distributive bilattice and $b \in \mathcal{B}$. Let P be a normalized program and G a normalized goal. Suppose θ is a substitution for the variables of G without generic constants. If $(\Phi_P \uparrow \omega)\ (G\theta) \succeq b$, then G has a b-proof with answer σ, such that $G\theta = G\sigma\gamma$, for some substitution γ.*

Proof: The proof is immediate from Lemma 7.11 and the definition of b-proof. ∎

7.3 Logic Programming with Join Irreducible Elements

Although the definition of a b-proof provides a sound and complete procedural semantics for logic programming over arbitrary distributive bilattices, it has a drawback. For a given truth value b, the search for a b-derivation of a complex goal G may entail searches for c-derivations of the subformulas of G for a large number of truth values c that are only remotely related to b; moreover, this *complexity* ramifies as we pass down the parse tree of G. It turns out that for finite distributive bilattices (and, more generally, bilattices with the *descending chain property*), we can essentially restrict our attention to b-derivations where b ranges over the relatively small subset of k-join-irreducible truth-values. Moreover, in the search for a b-derivation for G, we need only look for b-derivations of the subformulas of G.

We now present a *join-irreducible* operational semantics as an alternative to the standard one presented above. As we are concerned with the join-irreducible elements only in the k-ordering, throughout this section we will drop the subscript k and denote the class of these bilattice elements by $JIR(\mathcal{B})$.

Definition 7.13 Let $\mathcal{B}$ be a distributive bilattice, G a normalized formula, and P a normalized program over $\mathcal{B}$. Suppose that $b \in JIR(\mathcal{B})$.

1. G has a *b-JIR-proof of rank 0* with answer θ, if $G = c$, where $c \in \mathcal{B} - \{\bot, \top\}$, $b \preceq c$, and $\theta = \varepsilon$.

2. G has a *b-JIR-proof of rank $k+1$* with answer θ, if

   (a) G is an atom A, and $P \cup \{A\}$ has an SLDK-tree with at least one leafnode G', such that G' has a b-JIR-proof of rank k with answer θ', such that

$\theta = (\sigma_1 \cdots \sigma_n \theta')_G$, where $\sigma_1, \cdots, \sigma_n$ are the mgu's associated with the edges along the path form G to G'; or

(b) G is $\neg G'$, and G' has a $\neg b$-*JIR*-proof of rank k with answer θ; or

(c) $b \in JIR^+(\mathcal{B})$ and

i. $G = G_1 \vee G_2$, and at least one of G_1 or G_2 has a b-*JIR*-proof of rank k with answer θ; or

ii. $G = G_1 \wedge G_2$ or $G = G_1 \otimes G_2$, and G_i has a b-*JIR*-proof of rank k_i with answer θ_i $(i = 1, 2)$, such that $k = max(k_1, k_2)$, and $\theta = (\theta_1' \odot \theta_2')_G$, where θ_i' are variants of θ_i w.r.t. G_i; or

(d) $b \in JIR^-(\mathcal{B})$ and

i. $G = G_1 \vee G_2$ or $G = G_1 \otimes G_2$, and G_i has a b-*JIR*-proof of rank k_i with answer θ_i $(i = 1, 2)$, such that $k = max(k_1, k_2)$, and $\theta = (\theta_1' \odot \theta_2')_G$, where θ_i' are variants of θ_i w.r.t. G_i; or

ii. $G = G_1 \wedge G_2$, and at least one of G_1 or G_2 has a b-*JIR*-proof of rank k with answer θ.

Definition 7.14 Let $\mathcal{B}$ be a distributive bilattice with the Descending Chain Property in the knowledge ordering (DCP_k). Let $a \in \mathcal{B}$, and suppose that $a = b_1 \oplus \cdots \oplus b_n$, where $b_1 \oplus \cdots \oplus b_n$ is an irredundant decomposition of a as a join of join-irreducible elements of $\mathcal{B}$ in the knowledge ordering. Let G be a normalized formula and P a normalized program over $\mathcal{B}$. Then G has an a-*JIR*-proof with answer θ, if G has a b_i-*JIR*-proof with answer θ_i $(1 \leq i \leq n)$, such that $\theta = (\odot\{\theta_1, \cdots, \theta_n\})_G$.

In the rest of this section we will show that for distributive bilattices with the

descending chain property in the knowledge-ordering, the join-irreducible semantics and the generalized semantics presented earlier are equivalent.

Lemma 7.15 *Let $\mathcal{B}$ be a distributive bilattice, P be a normalized program, and G a normalized goal over $\mathcal{B}$. Suppose that $c \in JIR(\mathcal{B})$. If G has a c-JIR-proof of rank $k \geq 0$ with answer θ, then G has a b-JIR-proof with answer θ for every $b \preceq c$, where $b \in JIR(\mathcal{B})$.*

Proof: Let b be an arbitrary element of $JIR(\mathcal{B})$, such that $b \preceq c$. The result is proved by induction on k.

Basis: $(k = 0)$ If G has a c-*JIR*-proof of rank 0 with answer θ, then $G = d$, for some $d \in \mathcal{B} - \{\bot, \top\}$, such that $c \preceq d$, and $\theta = \varepsilon$. Since, $b \preceq c \preceq d$, G has a b-*JIR*-proof of rank 0 with answer $\theta = \varepsilon$.

Induction: Suppose the result holds for c-*JIR*-proof of rank k and assume that G has a c-*JIR*-proof of rank $k+1$ with answer θ. We need to consider the following cases.

1. *G is an atom:* Since G has a c-*JIR*-proof of rank $k+1$ with answer θ, $P \cup \{G\}$ has an SLDK-tree with at least one leafnode G' which has a c-*JIR*-proof of rank k with answer θ'. Moreover, $\theta = (\sigma_1 \cdots \sigma_n \theta')_G$, where $\sigma_1, \cdots, \sigma_n$ are the substitutions associated with the edges along the path from G to G'. By the inductive hypothesis, G' has a b-*JIR*-proof of rank k with answer θ'. By definition, G has a b-*JIR*-proof of rank $k+1$ with answer θ.

2. *G is $\neg G'$:* Then G' has a $\neg c$-*JIR*-proof of rank k with answer θ. Note that, since $b \preceq c$, we also have $\neg b \preceq \neg c$. By the inductive hypothesis, G' has a

$\neg b$-*JIR*-proof of rank k with answer θ. Thus, by the definition of JIR-proof, G has a b-*JIR*-proof of rank $k+1$ with answer θ.

3. *G is $G_1 \wedge G_2$:* There are two subcases.

   (a) $c \in JIR^{+}(\mathcal{B})$: Then, for $i = 1,2$, G_i has a c-*JIR*-proof of rank k_i with answer θ_i, such that $k = max(k_1, k_2)$ and $\theta = (\theta_1' \odot \theta_2')_G$, where θ_i' is a variant of θ_i w.r.t. G_i $(i = 1,2)$. By the inductive hypothesis, G_i has a b-*JIR*-proof of rank k_i with answer θ_i. Now, since $b \preceq c$, $c \in JIR^{+}(\mathcal{B})$, and $b \in JIR(\mathcal{B})$, by Lemma 4.19, we conclude that $b \in JIR^{+}(\mathcal{B})$. Then, by the definition of *JIR*-proof, $G = G_1 \wedge G_2$ also has a b-*JIR*-proof of rank $k+1$ with answer θ.

   (b) $c \in JIR^{-}(\mathcal{B})$: Then G_1 or G_2 (say G_1) has a c-*JIR*-proof of rank k with answer θ. Hence, by the inductive hypothesis, G_1 has a b-*JIR*-proof of rank k with answer θ. Then, by the definition of *JIR*-proof, $G = G_1 \wedge G_2$ has a b-*JIR*-proof of rank $k+1$ with answer θ.

4. *G is $G_1 \vee G_2$:* This case is dual to the case when $G = G_1 \wedge G_2$.

5. *G is $G_1 \otimes G_2$:* Then, for $i = 1,2$, G_i has a c-*JIR*-proof of rank k_i with answer θ_i, such that $k = max(k_1, k_2)$ and $\theta = (\theta_1' \odot \theta_2')_G$, where θ_i' is a variant of θ_i w.r.t. G_i $(i = 1,2)$. By the inductive hypothesis, G_i has a b-*JIR*-proof of rank k_i with answer θ_i. Now, by the definition of *JIR*-proof, $G = G_1 \otimes G_2$ has a b-*JIR*-proof of rank $k+1$ with answer θ. ∎

Lemma 7.16 *Let $\mathcal{B}$ be a distributive bilattice with the DCP_k. Let P be a normalized program over $\mathcal{B}$ and G a normalized goal. Suppose that $c \in \mathcal{B}$. For every $b \in JIR(\mathcal{B})$, where $b \preceq c$, if G has a c-JIR-proof with answer θ, then G has a b-JIR-proof with answer σ, such that $G\theta = G\sigma\gamma$, for some substitution γ.*

Proof: Suppose that G has a *c-JIR*-proof with answer θ. Since $\mathcal{B}$ satisfies the DCP_k, we can write $c = c_1 \ominus \cdots \ominus c_n$, such that $c_i \in JIR(\mathcal{B})$, for $i = 1, 2, \cdots, n$. Furthermore, G has a c_i-*JIR*-proof of rank k_i with answer θ_i, such that $\theta = (\odot\{\theta_1, \cdots, \theta_n\})_G$.

Now, $b \preceq c = c_1 \ominus \cdots \ominus c_n$. Hence, by Lemma 4.18, for some j $(1 \leq j \leq n)$, $b \preceq c_j$. Then, by Lemma 7.15, G has a *b-JIR*-proof with answer θ_j. Furthermore, note that $\odot\{\theta_1, \cdots, \theta_n\} = \theta_j\alpha$, where $\alpha = mgsu(\{\theta_1, \cdots, \theta_n\})$.

Putting things together, we have

$$\begin{aligned} G\theta &= G(\odot\{\theta_1, \cdots, \theta_n\}) \\ &= G\theta_j\alpha \\ &= G\sigma\gamma, \end{aligned}$$

where $\sigma = \theta_j$ and $\gamma = \alpha$. ∎

Lemma 7.17 *Let $\mathcal{B}$ be a distributive bilattice with the DCP_k. Let P be a normalized program and G a normalized goal over $\mathcal{B}$. Suppose that $c \in \mathcal{B}$. If G has a c-derivation of rank k, for some $k \geq 0$, with answer θ, then G has a c-JIR-proof with answer θ', such that $G\theta = G\theta'\gamma$, for some substitution γ.*

Proof: (By induction on k)

Basis ($k = 0$): If G has a c-derivation of rank 0 with answer θ, then $G = b$, for some $b \in \mathcal{B} - \{\bot, \top\}$, such that $c \preceq b$, and $\theta = \varepsilon$. Since $\mathcal{B}$ has the DCP_k, we can write $c = c_1 \oplus \cdots \oplus c_n$, where $c_1 \oplus \cdots \oplus c_n$ is an irredundant decomposition of c as join of join-irreducibles, i.e., $c_i \in JIR(\mathcal{B})$. Then $c_i \preceq b$ for $1 \leq i \leq n$. Now, by the definition of *JIR*-proof, G has a c_i-*JIR*-proof of rank 0 with answer $\theta = \varepsilon$. Hence, G has a c-*JIR*-proof with answer $\theta' = \odot\{\varepsilon, \cdots, \varepsilon\} = \varepsilon$.

Induction: Assume the result holds for c-derivations of rank k and consider the case where G has a c-derivation of rank $k+1$ with answer θ. We need to consider the following cases.

1. *G is an atom*: Then $P \cup \{G\}$ has an SLDK-tree with at least one leafnode G' which has a c-derivation of rank k with answer δ. Let $\sigma_1, \cdots, \sigma_m$ be the substitutions associated with edges along the path from G to G'. Then $\theta = (\sigma_1 \cdots \sigma_m \delta)_G$. By the inductive hypothesis, G' has a c-*JIR*-proof with answer δ', such that $G'\delta = G'\delta'\tau$, for some substitution τ. Since $\mathcal{B}$ has the DCP_k, c has an irredundant decomposition $b_1 \oplus \cdots \oplus b_n$, where $b_i \in JIR(\mathcal{B})$. Furthermore, for each i, G' has a b_i-*JIR*-proof with answer δ'_i, such that $\delta' = (\odot\{\delta'_1, \cdots, \delta'_n\})_{G'}$. By the definition of *JIR*-proof, G has a b_i-*JIR*-proof with answer α_i $(1 \leq i \leq n)$, where $\alpha_i = (\sigma_1 \cdots \sigma_m \delta'_i)_G$. Then G has a c-*JIR*-proof with answer θ', where $\theta' = (\odot\{\alpha_1, \cdots, \alpha_n\})_G$.

   Finally, we must show that $G\theta = G\theta'\gamma$, for some substitution γ. Let $\sigma = \sigma_1 \cdots \sigma_m$ (hence, $\alpha_i = (\sigma\delta'_i)_G$). Note that, by the unique renaming assumption, $\sigma\delta'_i$ and α_i are idempotent, and hence, by Lemma 5.12 [1], there exists a

[1] In the rest of this chapter we often use generalized versions (i.e., extended to sets of substitutions) of the properties of $\odot$ proved in Chapter 5.

substitution β such that

$$G[\odot\{\sigma\delta'_1,\cdots,\sigma\delta'_m\}] = G \odot \{\alpha_1,\cdots,\alpha_n\}\beta.$$

Furthermore, by Lemma 5.10, there exists a substitution ρ such that

$$\odot\{\sigma\delta'_1,\cdots,\sigma\delta'_n\}\rho = \sigma(\odot\{\delta'_1,\cdots,\delta'_n\}).$$

Then we have:

$$\begin{aligned} G\theta'\beta\rho\tau &= G[\odot\{\alpha_1,\cdots,\alpha_n\}]\beta\rho\tau \\ &= G \odot \{\sigma\delta'_1,\cdots,\sigma\delta'_n\}\rho\tau \\ &= G\sigma \odot \{\delta'_1,\cdots,\delta'_n\}\rho\tau \\ &= G\sigma\delta'\tau \qquad (\text{since } vars(G\sigma) \subseteq vars(G')) \\ &= G\sigma\delta. \end{aligned}$$

Now, let $\gamma = \beta\tau$.

2. G is $G_1 \wedge G_2$: Then, for $j = 1,2$, G_j has a c_j-derivation of rank k_j with answer θ_j, such that $k = max(k_1,k_2)$, $c = c_1 \wedge c_2$, and $\theta = (\eta_1 \odot \eta_2)_G$, where η_j are variants of θ_j w.r.t. G_j. By the inductive hypothesis, G_j has a c_j-*JIR*-proof with answer θ'_j, such that $G_j\theta_j = G_j\theta'_j\gamma_j$, for some substitution γ_j $(j = 1,2)$. Since, $\mathcal{B}$ has the DCP_k, we can write $c = b_1 \ominus \cdots \ominus b_n$, where $b_i \in JIR(\mathcal{B})$, for $1 \leq i \leq n$. Hence, $b_i \preceq c_1 \wedge c_2$. Now, for each b_i we must consider the following cases.

   (a) $b_i \in JIR^+(\mathcal{B})$:

   In this case by Lemma 4.16(2), $b_i \preceq c_j$, for $j = 1,2$, and by Lemma 7.16,

G_j has a b_i-*JIR*-proof with answer θ'_j. Let θ''_j be an idempotent variant of θ'_j $(j = 1, 2)$ w.r.t. G_j such that the conditions of Lemma 5.20 are satisfied. Hence, by the definition of *JIR*-proof, $G = G_1 \wedge G_2$ has a b_i-*JIR*-proof with answer $\phi_i = (\theta''_1 \odot \theta''_2)_G$ such that $G\theta = G(\eta_1 \odot \eta_2) = G(\theta''_1 \odot \theta''_2)\gamma' = G\phi_i\gamma'$, for some substitution γ'.

(b) $b_i \in JIR^-(\mathcal{B})$:

In this case by Lemma 4.16, $b_i \preceq c_j$, for $j = 1$ or $j = 2$ (assume $b_i \preceq c_1$), and by Lemma 7.16, G_1 has a b_i-*JIR*-proof with answer θ'_1. Hence, by the definition of *JIR*-proof, $G = G_1 \wedge G_2$ has a b_i-*JIR*-proof with answer $\phi_i = \theta'_1$.

Hence, G has a c-*JIR*-proof with answer $\theta' = (\odot\{\phi_1, \cdots, \phi_n\})_G$. Now, by Lemma 5.12, there exists a substitution ρ such that

$$G \odot \{\phi_1, \cdots, \phi_n\}\rho = G \odot \{\theta''_1 \odot \theta''_2, \theta'_1, \theta'_2\},$$

and, by applications of Lemma 5.12,

$$G \odot \{\theta''_1 \odot \theta''_2, \theta'_1, \theta'_2\}\beta = G(\theta''_1 \odot \theta''_2),$$

for some substitution β. Hence,

$$\begin{aligned} G\theta &= G(\eta_1 \odot \eta_2) \\ &= G(\theta''_1 \odot \theta''_2)\gamma' \\ &= G\theta'\rho\beta\gamma'. \end{aligned}$$

Now let $\gamma = \rho\beta\gamma'$.

3. *G is* $G_1 \otimes G_2$: Then, for $j = 1, 2$, G_j has a c_j-derivation of rank k_j with answer θ_j, such that $k = max(k_1, k_2)$, $c \preceq c_1 \otimes c_2$, and $\theta = (\eta_1 \odot \eta_2)_G$, where η_j is a variant of θ_j w.r.t. G_j. Since, $\mathcal{B}$ has the DCP_k, we can write $c = b_1 \oplus \cdots \oplus b_n$, where $b_i \in JIR(\mathcal{B})$, for $1 \leq i \leq n$. Hence, $b_i \preceq c_1 \otimes c_2$. Also, by the inductive hypothesis, G_j has a c_j-*JIR*-proof with answer θ'_j, such that $G_j\theta_j = G_j\theta'_j\gamma_j$, for some substitution γ_j $(j = 1, 2)$. Now, by Lemma 4.16, for each b_i, we have $b_i \preceq c_j$, for $j = 1, 2$, and by Lemma 7.16, G_j has a b_i-*JIR*-proof with answer θ'_j. Let θ''_j be a variant of θ'_j w.r.t. G_j, such that $vrange(\theta''_j)$ is disjoint from $vars(G)$ and from $dom(\eta_j)$. Hence, by Lemma 5.20, $\theta''_1 \odot \theta''_2$ exists and furthermore, there is a substitution γ' such that $\eta_1 \odot \eta_2 = (\theta''_1 \odot \theta''_2)\gamma'$. Let $\theta' = (\theta''_1 \odot \theta''_2)_G$. Then G has a b_i-*JIR*-proof with answer θ' such that $G\theta = G\theta'\gamma'$. Hence, by an argument similar to the previous case, G has a c-*JIR*-proof with answer θ' such that $G\theta = G\theta'\gamma$, for some substitution γ.

4. *G is* $G_1 \vee G_2$: This case is the dual of the case for $G = G_1 \wedge G_2$. ■

Lemma 7.18 *Let $\mathcal{B}$ be a distributive bilattice with the DCP_k. Let P be a normalized program over $\mathcal{B}$ and G a normalized goal. Suppose that $c \in JIR(\mathcal{B})$. If G has a c-JIR-proof with answer θ, then G has a c-derivation with answer θ.*

Proof: The result is proved by induction on the rank k of the c-*JIR*-proof for G.

Basis: $(k = 0)$ Suppose that G has a c-*JIR*-proof of rank 0 with answer θ. Then $G = b$, for some $b \in \mathcal{B} - \{\bot, \top\}$, such that $c \preceq b$, and $\theta = \varepsilon$. Now, the result follows immediately from the definition of c-derivation.

Induction: Assume the result holds for *c-JIR*-proof of rank k and consider the case when G has a *c-JIR*-proof of rank $k+1$ with answer θ. We must consider the following cases.

1. *G is an atom:* Since G has a c-*JIR*-proof of rank $k+1$ with answer θ, $P \cup \{G\}$ has an SLDK-tree with at least one leafnode G' which has a *c-JIR*-proof of rank k with answer θ'. Moreover, $\theta = (\sigma_1 \cdots \sigma_n \theta')_G$, where $\sigma_1, \cdots, \sigma_n$ are the substitutions associated with the edges along the path from G to G'. By the inductive hypothesis, G' has a c-derivation with answer θ'. Clearly, by the definition of c-derivation, G has a c-derivation with answer θ.

2. *G is $\neg G'$:* Then G' has a *b-JIR*-proof of rank k with answer θ, for some $b \in \mathcal{B}$, such that $\neg b = c$. By the inductive hypothesis, G' has a b-derivation with answer θ. Clearly, by the definition, G has a c-derivation with answer θ.

3. *G is $G_1 \wedge G_2$:* There are two subcases.

   (a) $c \in JIR^+(\mathcal{B})$: Then, for $i = 1,2$, G_i has a *c-JIR*-proof of rank k_i with answer θ_i, such that $k = max(k_1, k_2)$, and $\theta = (\theta_1' \odot \theta_2')_G$, where θ_i' is a variant of θ_i w.r.t. G_i. By the inductive hypothesis, G_i has a c-derivation with answer θ_i. Since $c = c \wedge c$, $G = G_1 \wedge G_2$ has a c-derivation with answer $(\theta_1' \odot \theta_2')_G = \theta$.

   (b) $c \in JIR^-(\mathcal{B})$: Then G_1 or G_2 (say G_1) has a *c-JIR*-proof of rank k with answer θ. Hence, by the inductive hypothesis, G_1 has a c-derivation with answer θ. On the other hand, G_2 has a $\perp$-derivation with answer ε. Now, since $c \in JIR^-(\mathcal{B})$, $c = c \wedge \perp$. Hence, $G = G_1 \wedge G_2$ has a c-derivation with answer θ.

4. *G is* $G_1 \vee G_2$ *:* This case is the dual of the case when $G = G_1 \wedge G_2$.

5. *G is* $G_1 \otimes G_2$ *:* The proof of this case is similar to part 3(a). ∎

We are now in a position to state our main result in this section, showing the correspondence between the two bilattice-based procedural semantics.

Theorem 7.19 *Let* $\mathcal{B}$ *be a distributive bilattice which has the* DCP_k. *Let* P *be a normalized program over* $\mathcal{B}$ *and* G *a normalized goal. Suppose that* $c \in \mathcal{B}$.

1. *If G has a c-proof with answer* θ, *then G has a c-JIR-proof with answer* θ', *such that* $G\theta = G\theta'\gamma$, *for some substitution* γ; *and*

2. *If G has a c-JIR-proof with answer* θ', *then G has a c-proof with answer* θ'.

Proof:

1. Suppose that G has a c-proof with answer θ. Then G has a c_i-derivation of rank k_i with answer θ_i, for $1 \leq i \leq n$, where $c = c_1 \oplus \cdots \oplus c_n$, and $\theta = (\odot\{\theta_1, \cdots, \theta_n\})_G$. Then, by Lemma 7.17, for each i, G has a c_i-*JIR*-proof with answer σ_i, such that $G\theta_i = G\sigma_i\gamma_i$, for some substitution γ_i.

   By the definition of *JIR*-proof, for each i and for every j $(1 \leq j \leq m_i)$, G has a c_i^j-proof with answer σ_i^j, such that $c_i = c_i^1 \oplus c_i^2 \oplus \cdots \oplus c_i^{m_i}$, where $c_i^j \in JIR(\mathcal{B})$, and $\sigma_i = (\odot\{\sigma_i^j \mid 1 \leq j \leq m_i\})_G$. Clearly,

$$c = c_1^1 \oplus \cdots \oplus c_1^{m_1} \oplus \cdots \oplus c_n^1 \oplus \cdots \oplus c_n^{m_n}.$$

   Now, in order to obtain an irredundant representation of c we may need to throw out some of the c_i^j. However, it can be easily verified that for any set

S of unifiable idempotent substitutions and any $S' \subseteq S$, $\odot S' \leq \odot S$. Hence, by Lemma 5.12 and by the definition of *JIR*-proof, G has a *c-JIR*-proof with answer θ' such that, for some substitution β,

$$G\theta'\beta = G(\odot\{\sigma_1^1, \cdots, \sigma_1^{m_1}, \cdots, \sigma_n^1, \cdots, \sigma_n^{m_n}\}).$$

Now, to complete the proof, we must show that for some substitution γ, $G\theta = G\theta'\gamma$.

First we observe that by selecting the answers σ_i to be appropriate variants w.r.t. G_i, we can guarantee that $vrange(\sigma_i)$ is disjoint from $vars(G_i)$ and from $dom(\theta_i)$. Hence, by Lemma 5.20,

$$\odot\{\theta_1, \cdots, \theta_n\} = (\odot\{\sigma_1, \cdots, \sigma_n\})\gamma',$$

for some substitution γ'. Since by the unique renaming assumption we can guarantee that σ_i $(i = 1, \cdots, n)$ are idempotent, by Lemma 5.17 (associativity of $\odot$), for some substitution ρ,

$$\odot\{\sigma_1, \cdots, \sigma_n\} = \odot\{\sigma_1^1, \cdots, \sigma_1^{m_1}, \cdots, \sigma_n^1, \cdots, \sigma_n^{m_n}\}\rho.$$

Hence,

$$\begin{aligned} G\theta &= G \odot \{\theta_1, \cdots, \theta_n\} \\ &= G(\odot\{\sigma_1, \cdots, \sigma_n\})\gamma' \\ &= G \odot \{\sigma_1, \cdots, \sigma_n\} = \odot\{\sigma_1^1, \cdots, \sigma_1^{m_1}, \cdots, \sigma_n^1, \cdots, \sigma_n^{m_n}\}\rho\gamma' \\ &= G\theta'\beta\rho\gamma'. \end{aligned}$$

Now let $\gamma = \beta\rho\gamma'$.

2. Suppose that G has a c-*JIR*-proof with answer θ'. Then G has a c_i-*JIR*-proof with answer θ_i, such that $\theta' = (\Im\{\theta_1, \cdots, \theta_n\})_G$, and $c = c_1 \ominus \cdots \ominus c_n$, where $c_i \in JIR(\mathcal{B})$. Now, by Lemma 7.18, G has a c_i-derivation of rank k_i with answer θ_i. By the definition of c-proof, G has a c-proof with answer θ'. ∎

Theorem 7.19 implies a completeness theorem for the join-irreducible procedural semantics as a corollary of the Completeness Theorem for the generalized procedural semantics presented earlier (Theorem 7.12).

Theorem 7.20 (Join-Irreducible Completeness) *Let $\mathcal{B}$ be a distributive bilattice with DCP_k, and $b \in \mathcal{B}$. Let P be a normalized program and G a normalized goal. Suppose θ is a substitution for the variables of G. If $(\Phi_P \uparrow \omega)\ (G\theta) \succeq b$, then G has a b-JIR-proof with answer σ, such that $G\theta = G\sigma\gamma$, for some substitution γ.*

Proof: Suppose that $(\Phi_P \uparrow \omega)(G\theta) \succeq b$, then by Theorem 7.12 G has a b-proof with answer σ' such that $G\theta = G\sigma'\gamma'$, for some substitution γ'. Now, by Theorem 7.19, G has a b-*JIR*-proof with answer σ such that $G\sigma' = G\sigma\gamma''$, for some substitution γ''. Now:

$$\begin{aligned} G\theta &= G\sigma'\gamma' \\ &= G\sigma\gamma''\gamma' \\ &= G\sigma\gamma, \end{aligned}$$

where $\gamma = \gamma''\gamma'$. ∎

CHAPTER 8. CONCLUSIONS

Many domain problems in Artificial Intelligence require representation of knowledge which is uncertain, approximate, incomplete, or even contradictory. Realization of this fact has been the motivation behind a great deal of literature concerned with the formalization of revisable reasoning by relying on non-classical logics. First-order predicate calculus, itself, has been found not to be completely satisfactory because of its all or nothing nature; statements can be only be true or false. Naturally, this problem is extended to traditional logic programming languages, since they use the machinery of first-order logic. For example, the only natural way to deal with uncertainty, in Prolog, is to predefine a fixed set of terms and explicitly pattern match for them through the unification process.

Yet, logic programming provides a natural and declarative framework for knowledge representation as well as inference mechanism in many AI applications. Thus, it is desirable to construct logic programming systems that can adequately deal with the type of commonsense reasoning involved in many AI problem domains. One way that this issue has manifested itself has been in a variety of approaches to provide appropriate semantics for negation in logic programs. Another, has been the use of non-classical logics such has three-valued logics, intuitionistic logics, probabilistic logics, and modal logics as the underlying framework for logic programming. In this

work, we have presented a generalized framework for knowledge-based logic programming which can model, as special cases, many useful logics, such as those mentioned above. This generalized framework for both the declarative and the procedural semantics of logic programs, is made possible by relying on algebraic properties of bilattices which combine the truth and knowledge components on a space of truth values. Similar frameworks have been studied by Ginsberg [24] (in the context of *truth maintenance systems* and by Fitting [18] (in the context of logic programming).

The present work sets forth a natural procedural semantics for knowledge-based logic programs which takes full advantage of the bilattice properties, particularly, for the class of distributive bilattices whose knowledge ordering has the descending chain property. Furthermore, the procedural semantics has been specified according to a parallel computation model for evaluation of queries by making full use of the natural properties of the bilattice operators and incorporating the notion of substitution unification. It is hoped that the generalized framework presented her can serve as the basis for implementing a flexible logic programming language which can be found useful in a variety of AI applications without suffering from the aforementioned shortcomings of standard logic programming languages. In the following we will summarize some of the most important results in this work and discuss some ideas for future extensions and further research.

The main contribution of this work to the theory of bilattices is the analysis of the extension of the lattice theoretic results on join-irreducible elements to the realm of bilattices. Birkhoff's representation theorem for distributive bilattices (see Theorem 2.23) shows the importance of join-irreducible elements as a characteristic set of elements in a lattice. It turns out that join-irreducible elements play a very

similar role in distributive bilattices. Since we are mainly interested in using bilattice elements in the context of semantics for knowledge-based logic programs, we focus our attention on k-join-irreducible elements. In particular, Lemma 4.14 provides a characterization of the k-join-irreducible elements of a bilattice in terms of the join-irreducible elements of the underlying lattice structures. The most important result of Chapter 4 is Lemma 4.17 which provides the basis for the subsequent development of the generalized knowledge-based procedural semantics. It must be noted that it is possible to provide further characterizations of the join-irreducible elements in a distributive bilattice based on the notions of filters and ideals of lattices. Evidence of this is provided by the fact that the set of negative k-join-irreducible elements in a distributive bilattice $\mathcal{B}$, $JIR_k^-(\mathcal{B})$, is, in fact, the principle ideal generated by the element `false` (minus the bottom element), and the set $JIR_k^+(\mathcal{B})$ is the principle ideal generated by the element `true`. Based on these ideas, it seems possible to specify a representation theorem similar to Birkhoff's theorem for distributive bilattices. Considering the principle *truth filters* and *truth ideals* in such bilattices can shed further light on the relationships between the knowledge and truth components of the join irreducible truth values.

While both Ginsberg [24] and Fitting [18, 19] consider various types of bilattices and the relationships among various categories of bilattice elements, it seems that it is the join-irreducible characterization of distributive bilattices provides the best results in the development of more natural semantics for logic programming languages.

In order to provide a parallel computation model for our procedural semantics we have used the notion of substitution unification. Substitution unification ensures the consistency of bindings obtained for shared variables during independent compu-

tations of subgoals. Similar ideas have been studied in the literature, such as the notion of *reconciliation* [26] and that of *parallel composition* [38]. But, these approaches invariably use expression-based systems of equations and identify the desired substitutions by the solutions to these systems of equations. In a manner of speaking, this treatment hides some of the interesting algebraic properties of substitution unifiers.

We fully develop these properties in an equation or expression independent manner. However, for the purpose of logic programming semantics, all of these approaches are equivalent. In particular, in Lemma 5.11 we showed that the substitution unification of two idempotent substitutions is in fact the least upper bound of the two substitutions (according to the usual ordering of substitutions). This result is also obtained by Palamidessi in [38], indicating that the notions of parallel composition and substitution unification are equivalent when restricting our attention to the lattice of idempotent substitutions.

However, our treatment of substitution unification is not restricted to idempotent substitutions and thus we have been able to consider many of its more general properties which are useful in their own right. It would be interesting to further explore the relationships between these two treatments as parts of each make useful contributions to the theory of unification. For example, as pointed out in [38], the fact that the substitution unification is the least upper bound in the lattice of idempotent substitutions, gives us some of the useful properties for free. These include idempotence and associativity (see Lemma 5.18).

In Chapter 6 we considered a special case of our join-irreducible procedural semantics which was subsequently extended to arbitrary distributive bilattices with the descending chain property. The four-valued case deserves special attention since

it illustrates the utility of the proposed semantics in several respects.

The four-valued case represents a minimalistic logic programming system which resolves some of the problems we previously discussed in classical logic programming. In particular, the system is well suited for many AI application domains since it can effectively deal with incomplete or contradictory information. This is especially evident in the context of distributed deductive knowledge bases, where the information is obtained at different sites and independently. Furthermore, due to the emphasis on knowledge rather than truth, as we saw earlier, the semantic problems associated with negation are no longer present. Yet, the four-valued case is strong enough to subsume all of the machinery of first-order and three-valued logic programs.

In the context of this four-valued knowledge-based logic programming system, we also considered the incorporation of the knowledge-base version of Closed World Assumption. The Closed World Soundness and Completeness Theorems showed that, contrary to the classical logic programs, incorporation of this rule into the semantics does not lead to incompleteness.

The Closed World semantics, however, does suffer from a serious shortcoming. In order to obtain negative information about a goal (to obtain a cw-refutation), one must guess the answer. This answer is applied to the goal before the goal is evaluated. It is possible to do better than this by using a concept of *non-unification*, as described below.

As noted above, the incorporation of the CWA into the semantics introduces an asymmetry between the notions of success and failure. This problem can be remedied by allowing the system to return a substitution, as the answer for a refutation, witnessing the fact that the goal does not unify with the head of any clause. We call

such a substitution a *non-unifier*. However, since in general there may be infinitely many most general non-unifiers of two expressions, we would like to use certain type of *substitution schemes* in order to represent these non-unifiers in a finite manner. This points to an area in which the present work is continuing.

It must be noted, however, that one of the main motivations of CWA comes from the area of deductive databases where the programs are primarily function-free. In the absence of function symbols, there are only finitely many non-unifiers that need to be considered. Thus, in the context of a deductive database system based on the four-valued bilattice, the notion of non-unifiers should be sufficient to constructively obtain answers to refutations when evaluating a query.

This situation constitutes a substantial improvement over the traditional logic programming systems which incorporate Negation as Failure. In general, since Negation as Failure cannot be performed on nonground subgoals, bindings can only be obtained by successful calls of positive literals. Negative calls never create bindings; they only succeed or fail. Thus Negation as Failure is purely a test and cannot be used to obtain answers in the query evaluation process.

Another related issue which deserves further study is the possibility of incorporating a generalized version of the CWA in the generalized semantics given in Chapter 7 (discussed further below) for arbitrary distributive bilattices. In this context, for instance, the system could obtain a b-proof for an atom A, if A does not unify with the head of any clause. In this manner, the default truth value will not be `false`, but some other element b in the bilattice. One interesting possibility here is to annotate each predicate symbol with a truth value from the bilattice. Presumably, this annotation will determine what the default value of an atom will be if it does not

unify with the head of any clause (in other words, when no information is available about that atom). The generalization of the CWA in this manner will be an area of future research.

Finally, in Chapter 7 we generalize the ideas presented for the four-valued case. In particular, the operational semantics is generalized to an arbitrary distributive bilattice. We introduce the notion of a *b-proof* for each element of the bilattice except $\top$ and $\bot$. (In the 4-element case **true**-proofs coincide with proofs and a **false**-proofs with refutations.) We prove a soundness and completeness theorem for this procedural semantics, again with respect to the declarative fixpoint semantics (see Theorems 7.8 and 7.12).

The main results of this chapter, however, are the alternate procedural semantics which we obtain by restricting attention to the k-join-irreducible elements of the distributive bilattice. It is the join-irreducible semantics which is, in fact, the generalization of the procedural semantics given for the four-valued case. We show that, for a large class of distributive bilattices, namely those with the descending chain property, the two versions of the procedural semantics are, in fact, equivalent. This result is established in Theorem 7.19. Theorem 7.19, thus, implies a completeness theorem for the join-irreducible procedural semantics as a corollary of the Completeness Theorem for the generalized procedural semantics presented earlier.

The join-irreducible procedural semantics, thus, provides a generalized operational model which can serve as the basis for efficient implementation of knowledge-based logic programming languages. The actual implementation of a logic programming language based on the ideas presented here, however, remains an area in which the present work will hopefully continue.

BIBLIOGRAPHY

[1] K. R. Apt and R. Bol, Logic Programming and negation: a survey, *Journal of Logic Programming*, to appear (1994).

[2] K. R. Apt and M. H. van Emden, Contribution to the theory of logic programming, *JACM*, **29** (1982), pp. 841-862.

[3] N. D. Belnap, Jr. A usefull four-valued logic, in *Modern Uses of Multiple-Valued Logic*, J. Michael Dunn and G. Epstein editors, D. Reidel, Boston (1977), pp. 8-37.

[4] G. Birkhoff, On the combination of subalgebras, *Proc. Cambridge Philosophical Society*, **29** (1933), pp. 441-464.

[5] H. A. Blair and V. S. Subrahmanian, Paraconsistent foundations for logic programming, *Journal of Non-Classical Logic*, **5** (1982), pp. 45-73.

[6] S. Brass and U. W. Lipeck, Specifying closed world assumptions for logic databases, in *Proceedings of the Second Symposium on Mathematical Fundamentals of Database Systems, MFDBS89* (1989).

[7] S. Cerrito, Negation as failure, a linear axiomatization, *Journal of Logic Programming*, **12** (1988), pp. 1-24.

[8] K. L. Clark, Negation as failure, in *Logic and Data Bases*, H. Gallaire and J. Minker editors, Plenum Press, New York (1978), pp. 293-322.

[9] J. S. Conery and D. F. Kibler, Parallel interpretation of logic programs, in *Proceedings of the ACM Conference on Functional Programming Languages and Computer Architecture*, (1981), pp. 163-170.

[10] J. S. Conery and D. F. Kibler, AND parallelism and nondeterminism in logic programs, *New Generation Computing*, **3** (1985), pp. 43-70.

[11] B. A. Davey and H. A. Priestley, *Introduction to Lattices and Order*, Cambridge University Press, Cambridge, Mass. (1990).

[12] J. Dix, Semantics of logic programs, their intuition and formal properties: an overview, In *Logic, Action and Information. Proceedings of the Konstanz Colloquium in Logic and Information*, A. Fuhrmann and H. Rott, editors, DeGruyter (1993).

[13] E. Eder, Properties of substitutions and unification, *Journal of Symbolic Computation*, **1** (1985), pp. 31-46.

[14] M. C. Fitting, A Kripke/Kleene semantics for logic programs, *Journal of Logic Programming*, **4** (1985), pp. 295-312.

[15] M. C. Fitting, Partial models and logic programming, *Theoretical Computer Science*, **48** (1986), pp. 229-255.

[16] M. C. Fitting, Logic programming on a topological bilattice, *Fund. Informatica*, **11** (1988), pp. 209-218.

[17] M. C. Fitting, Negation as refutation, in *Proceedings of the Fourth Annual Symposium on Logic in Computer Science*, R. Parikh editor, IEEE (1978), pp. 63-70.

[18] M. C. Fitting, Bilattices in logic programming, in *The Twentieth International Symposium on Multiple-Valued Logic*, G. Epstein editor, IEEE (1990), pp. 63-70.

[19] M. C. Fitting, Bilattices and semantics of logic programming, *Journal of Logic Programming*, **11** (1991), pp. 91-116.

[20] M. Gelfond, On stratified autoepistemic theories, in *Proceedings of the Sixth National Conference on Artificial Intelligence, AAAI-87*, Morgan Kaufmann, Los Altos, CA (1987), pp. 207-211.

[21] M. Gelfond and V. Lifschitz, The stable model semantics for logic programming, in *Proceedings of the Fifth Logic Programming Symposium, Association for Logic Programming*, MIT Press (1988), pp. 1070-1080.

[22] M. R. Genesereth and N. Nilsson, *Logical Foundations of Artificial Intelligence*, Morgan Kaufamn, Los Altos, CA (1987).

[23] M. L. Ginsberg, Multi-valued logics, in *Proceedings of the Fifth National Conference on Artificial Intelligence, AAAI-86*, Morgan Kaufmann, Los Altos, CA (1986), pp. 243-247.

[24] M. L. Ginsberg, Multi-valued logics: a uniform approach to reasoning in artificial intelligence, *Computational Intelligence*, **4** (1988), pp. 265-316.

[25] J. Herbrand, Investigation of proof theory, in *From Frege to Gödel: A Source Book in Mathematical Logic, 1879-1931*, J. van Heijenoort editor, Harvard University Press (1967), pp. 525-581.

[26] J. M. Jacquet, *Conclog: A Methodological Approach to Concurrent Logic Programming*, Springer-Verlag, Berlin (1991).

[27] S. C. Kleene, *Introduction to Metamathematics*, Van Nostrand Reinhold (1957).

[28] R. A. Kowalski, Predicate logic as a programming language, in *Information Processing 74*, Stockholm, North Holland (1974), pp. 569-774.

[29] K. Kunen, Negation in logic programming, *Journal of Logic Programming*, **4** (1987), pp. 289-308.

[30] K. Kunen, Signed data dependencies in logic programs, *Journal of Logic Programming*, **7** (1987), pp. 231-246.

[31] J. L. Lassez and M. J. Maher, Optimal fixpoints of logic programs, *Theoretical Computer Science*, **39** (1985), pp. 15-25.

[32] J. L. Lassez and M. J. Maher and K. Marriott, Unification revisited, in *Foundations of Deductive Databases and Logic Programming*, J. Minker editor, Morgan Kaufmann, Los Altos, CA (1988), pp. 587-625.

[33] D. McDermott, Non-monotonic logic II: non-monotonic modal theories, *Journal of the ACM*, **1** (1982), pp. 34-57.

[34] R. C. Moore, Semantical considerations on nonmonotonic logic, *Artificial Intelligence*, **1** (1985), pp. 75-94.

[35] R. C. Moore, Autoepistemic logic, in *Non-Standard Logics for Automated Reasoning*, P. Smets et al. editors, Academic Press, London (1988), pp. 105-136.

[36] V. Lifschitz, On the declarative semantics of logic programs with negation, in *Foundations of Deductive Databases and Logic Programming*, J. Minker editor, Morgan Kaufmann, Los Altos, CA (1988), pp. 177-192.

[37] J. W. Lloyd, *Foundations of Logic Programming*, second edition, Springer, Berlin (1987).

[38] C. Palamidessi, Algebraic properties of idempotent substitutions, in *Proccedings of the 17th International Colloquium on Automata, Languages and Programming*, M. S. Paterson editor, Springer-Verlag, Berlin (1990), pp. 386-399.

[39] H. Przymusinska, On the relationship between autoepistemic logic and circumscription for stratified deductive databases, in *Proceedings of the ACM SIGART, International Symposium on Methodologies for Intelligent Systems*, Knoxville, Tenn (1987).

[40] T. C. Przymusinski, Perfect model semantics, in *Proceedings of the Fifth Logic Programming Symposium, Association for Logic Programming*, R. Kowalski and K. Bowen, Editors, MIT press (1988), pp. 1081-1086.

[41] T. C. Przymusinski, On the semantics of stratified deductive databases, in *Foundations of Deductive Databases and Logic Programming*, J. Minker editor, Morgan Kaufmann, Los Altos, CA (1988), pp. 193-216.

[42] T. C. Przymusinski, Three-valued non-monotonic formalisms and logic programming, in *Proceedings of the First International Conference on Knowledge Representation and Reasoning*, Toronto, Canada (1989).

[43] R. Reiter, On closed world data bases, in *Logic and Data Bases*, H. Gallaire and J. Minker editors, Plenum Press, New York (1978), pp. 55-76.

[44] R. Reiter, A logic for default reasoning, *Artificial Intelligence*, 1-2 (1980), pp 81-131.

[45] J. A. Robinson, A machine-oriented logic based on resolution principle, *JACM* **12** (1965), pp. 23-41.

[46] K. Ross and R. W. Topor, Inferring negative information from disjunctive databases, *Technical Report 87-1*, University of Melbourne (1987).

[47] E. Sandewall, A functional approach to nonmonotonic logic, *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, Los Angeles, CA (1985), pp. 100-106.

[48] D. S. Scott, Some ordered sets in computer science, in *Ordered Sets*, I. Rival editor, D. Reidel Publishing Company, Boston, MA (1982), pp. 677-718.

[49] J. C. Shepherdson, Negation in logic programming, in *Foundations of Deductive Databases and Logic Programming*, J. Minker editor, Morgan Kaufmann, Los Altos, CA (1988), pp. 19-88.

[50] J. C. Shepherdson, Logics for negation as failure, in *Logic from Computer Science*, Y. N. Moschovakis editor, Springer-Verlag, Berlin (1990), pp. 521-583.

[51] R. M. Smullyan, *First Order Logic*, Springer-Verlag, Berlin (1968).

[52] A. Takeuchi, *Parallel Logic Programming*, John Wiley and Sons, Inc. (1992).

[53] A. Tarski, A lattice theoretical fixpoint theorem and its applications, *Pacific journal of Mathematics*, **5** (1955), pp. 285-309.

[54] A. Thayse (editor), *From Standard Logic to Logic Programming, Introducing a Logic Based Approach to Artificial Intelligence*, Wiley, Chichester (1988).

[55] A. Van Gelder, Negation as failure using tight derivations for general logic programs, *Journal of Logic Programming*, **6** (1989), pp. 109-133.

[56] M. van Emden and R. A. Kowalski, The semantics of predicate logic as a programming language, *JACM*, **23** (1976), pp. 733-742.

[57] M. van Emden, Quantitative deduction and its fixpoint theory, *Journal of Logic Programming*, **3** (1986), pp. 37-53.