A Framework for Expressing Prioritized Constraints Using Infinitesimal Logic

by

Ruchi Agarwal
B.Sc., University of Victoria, 2003

A thesis submitted in partial fulfillment
of the requirements for the degree of

Master of Science
in the Department of Computer Science

Supervisor: Dr. William W. Wadge

Abstract

In this thesis, we propose an extension to the multiple-valued infinitesimal logic framework to provide a simple representation for prioritized constraints. We introduce two unary operators, μ and ω, to infinitesimal logic in order to define preferential constraints and backup constraints, respectively. The new framework naturally allows us to define a hierarchy of priorities among constraints. Also, we present a lazy algorithm for evaluating the multiple-valued prioritized constraint expressions of our representation. Our algorithm, which is similar to the alpha-beta pruning technique for minimax game tree evaluation, is based on a recursive depth-first traversal of the parse tree for the expression and works by evaluating operands within an increasingly narrower range of interest. Our implementation of this representation for querying a movies database demonstrates the expressive power and flexibility of our framework.

Contents

Abstract ii

Contents iii

List of Figures v

Acknowledgments vi

1 Introduction 1

1.1 Problem and Motivation 1

1.2 Priority Assertion Framework 2

1.3 Thesis Outline 3

2 Literature Survey 4

2.1 Related Work 4

2.2 Our Approach 7

3 Priority Assertion 8

3.1 Infinitesimal Logic 8

3.2 The Operators μ and ω 11

3.2.1 Definition and Theorems 11

3.2.1.1 Multiplication 11

3.2.1.2 Division 18

3.2.1.3 Duality 19

3.3 Asserting Priorities using PCEs 21

4 The Lazy Evaluation of a PCE **26**

4.1 Lazy Evaluation 27

4.1.1 Evaluation of an *and* Node 29

4.1.2 Evaluation of an *or* Node 30

4.1.3 Evaluation of a *not* Node 31

4.1.4 Evaluation of μ and ω Nodes 31

4.2 Correctness of the Lazy Evaluation Algorithm 32

4.3 Comparison of Alpha-Beta Pruning and the Lazy Evaluation of PCEs 34

4.3.1 Alpha-Beta Pruning 35

4.3.2 Comparison 36

5 Prioritized Queries and the Movies Database Application **39**

5.1 Database 39

5.2 Front-End Application 40

5.3 User Interface 41

5.3.1 Connecting to the Database 42

5.3.2 Specifying a Query 42

5.3.3 Viewing the Results 43

5.3.4 Connection and Query Information 44

5.3.5 Example Queries 45

6 Conclusion **47**

Bibliography **48**

A Infinitesimal Logic Expression Evaluator: Prolog Source Code **50**

B Java Source Code **57**

List of Figures

3.1 Example apartment data. 23

3.2 Example apartment data with evaluations. 24

3.3 Example data for flights to Toronto. 24

3.4 Example data for flights to Toronto with evaluations. 25

4.1 Evaluation of an *and* node. 29

4.2 Evaluation of an *or* node. 30

4.3 Evaluation of a *not* node. 31

5.1 Initial View of the Movies Database application. 42

5.2 View of the Movies Database application after a Query. 44

5.3 View of Hitchcock Query. 45

5.4 View of Awards Query. 46

Acknowledgments

First, I would like to thank Dr. William Wadge for his flexible supervision, knowledgeable suggestions, and valuable guidance throughout my studies at the University of Victoria. His ideas on the use of infinitesimal logic for priority assertion formed the foundation for my research. It was truly a pleasure to learn from him.

I am also grateful to my committee members, Dr. Bruce Kapron and Dr. Alex Thomo for careful reviewing of this thesis and their valuable comments and suggestions.

In addition, I am thankful to Dr. Gio Wiederhold of Stanford University for his movies data site where we obtained the test data for our movies database application. Also, thanks to Dr. Alex Thomo for suggesting the use of this movies data and his helpful input on database issues.

Finally, I would like to express my sincerest gratitude to my family. In particular, I would like to thank my parents, brother, sister-in-law and niece for providing such a caring, loving environment and my husband for his constant support, patience and encouragement.

Chapter 1

Introduction

Search technology has tremendously advanced over the last few years. However, querying techniques still lack the flexibility required to accurately express user priorities. For the most part, queries are intended to accept elements that satisfy a given crisp criteria and reject those that do not. We propose a framework based on infinitesimal logic for specifying prioritized soft constraints. In this chapter, we present the motivations for this representation, give a brief description of our approach, and give an outline for the rest of this thesis.

1.1 Problem and Motivation

Classical query techniques only allow users to specify hard constraints, ie. requirements, that every solution must satisfy. For example, performing a SQL query on a relational database results in the tuples that satisfy all of the query constraints. This rigid framework gives rise to two common problematic scenarios: a large set of undifferentiated solutions or an empty solution set.

Human thinking, however, is normally not so restrictive. In cases where there are many solutions, we would like to differentiate among them by specifying some preferential constraints. On the other hand, if there are no solutions satisfying all

of our requirements, we are often willing to make compromises and settle for some backup options rather than having no solution at all. Most importantly, the use of infinitesimal logic as the foundation of our representation allows us to capture the notion of an option being infinitely more preferable than another option.

For example, suppose that we would like to take a flight to Toronto, ON and that we would like a window seat. In most cases, it is common sense that the destination is far more important than the seat choice. In fact, we can say that it is infinitely more important. There is no way to specify this criteria using hard constraints. Using conventional techniques, we would need to formulate multiple queries to capture these preferences. To express queries like the flight search example presented here, we need a framework that more accurately models human thinking. We can achieve this by allowing users to specify soft constraints, ie. constraints that may be violated, and allowing them to specify priorities on the constraints such that some constraints are infinitely more (or less) important than others. We shall see later that the flight query above can be easily expressed using our framework as follows

$$(destination = Toronto, ON) \quad and \quad \mu(seat = window)$$

1.2 Priority Assertion Framework

In this thesis, we present a representation allowing two different types of soft constraints: preferential constraints and backup constraints. By preferential constraints, we mean constraints that we would like to satisfy, if possible, in addition to the required constraints. By backup constraints, we mean constraints that we would like to satisfy in case we are not able to satisfy the required constraints. The foundation for our approach is infinitesimal logic, introduced by Rondogiannis and Wadge [1]. Infinitesimal logic is an infinite-valued logic based on an uncountable linearly ordered set of truth values. The values are obtained by successive multiplication of the classical truth values, $True$ and $False$, by an infinitesimal value, ϵ. T_k denotes the value

$\epsilon^k T$ and F_k denotes the value $\epsilon^k F$. The classical $True$ and $False$ values are represented by T_0 and F_0, respectively. Any intermediate truth value, T_k, is infinitely truer than T_{k+1} and F_k is infinitely falser than F_{k+1}. The value 0 represents *undefined*. Intuitively, multiplication by ϵ mitigates the truth value to which it is applied by an infinite amount.

We introduce two unary operators, μ and ω, to classical Boolean logic in order to define preferential constraints and backup constraints, respectively. Increasing degrees of μ and ω specify a hierarchy on constraints. In general, μ^k mitigates false values by a degree k and ω^k mitigates true values by degree k.

We call expressions involving infinitesimal truth values and the operators μ and ω prioritized constraint expressions (PCEs). A lazy evaluation algorithm, with a linear time complexity with respect to the number of operators, is used to evaluate the PCEs. The algorithm is similar to the alpha-beta pruning technique.

Although we present the framework in the context of database queries, our representation is quite general and could be used for any application requiring priority assertion.

1.3 Thesis Outline

The rest of this thesis is organized as follows: In chapter 2, we discuss related work in the area of priority assertion. Chapter 3 shows how we use infinitesimal logic extended by the unary operators, μ and ω, to assert priorities. In chapter 4, we present our lazy evaluation algorithm for the prioritized constraint expressions of our representation and compare our approach with the alpha-beta pruning method for minimax game trees. Chapter 5 describes our implementation of a movies database program to demonstrate the use of our framework for SQL queries. Finally, chapter 6 concludes the thesis.

Chapter 2

Literature Survey

During the last three decades, the concept of priority assertion has received attention from a diverse community of researchers in various fields of mathematics, science and engineering. In this chapter, we discuss a number of the relevant frameworks that have emerged from this research and give reasons for our chosen approach.

2.1 Related Work

In [2], Chomicki presents a qualitative model to address the issue of user preferences in the context of database queries. Preferences are defined using preference relations between tuples and a winnow operator is used to select from its argument relation the most preferred tuples according to a given preference relation. Unlike our representation which can be applied to any context requiring priority assertion, this approach is restricted to the relational database domain since the definition of the preference relations relies on tuple information.

A constraint hierarchy scheme, which is implemented in the graphical constraint based application ThingLab, is presented in [3] by Borning, Freeman-Benson and Wilson. Given a constraint hierarchy H with n non-required levels, H_0 is a vector of required constraints, H_1 is a vector of the constraints in H at the strongest non-required

level, and so forth through the weakest constraints H_n. A solution to a constraint hierarchy is a set of valuations, such that all valuations in the set satisfy all of the required constraints. No valuation in the set is "better" than any other valuation in the set but the valuations can be compared using a number of different comparators. The approach in [3] differs from our approach since it uses a generative algorithm to generate solutions based on constraints whereas our approach provides an evaluation algorithm to evaluate each option according to the constraints. Although the algorithms of [3] have a lower runtime than our approach, they are more complex and, therefore, more difficult to implement than our lazy evaluation algorithm. Also, because of the linearly ordered infinitesimal logic framework of our approach, hierarchies may be easily expressed and comparing solutions is more straightforward. In addition, our approach provides the ability to distinguish between preferential constraints and backup constraints, whereas [3] only allows the specification of a hierarchy among preferential constraints. There are some metric-better comparators described in [3], however, that our approach can not handle.

Assuming that a constraint hierarchy is specified in a similar manner as the constraint hierarchy scheme of [3], Hosobe proposes a method for solving nonlinear constraints with hierarchical preferences in [4]. The proposed method finds locally optimal solutions by combining the hierarchical QR decomposition with the Gauss-Newton method. As with the approach presented in [3], this approach fails to address the expression of backup constraints. In addition, like the comparator scheme of [3], the computations involved with finding the solution are far more complex and time-consuming than our simple lazy evaluation algorithm.

In addition to the qualitative approaches discussed here, there are several quantitative models for asserting priorities. One of the most prevalent examples of a quantitative representation is fuzzy logic. Fuzzy logic, introduced in the 1960s by Dr. Lotfi A. Zadeh, extends classical Boolean logic to handle the concept of partial truth. The basis of fuzzy logic is fuzzy sets. Unlike conventional set theory where an object

either does or does not belong to a set, fuzzy set theory allows partial membership. That is, objects can belong to a set to a certain degree between 0% and 100% [5]. Although, fuzzy logic facilitates the expression of a hierarchy of priorities, it lacks the ability to express preference by an infinite degree, which often arises in common sense reasoning and decision-making problems. In fact, this is a problem with any quantitative approach. For example, consider a flight constraint problem where we need to fly to Toronto. If possible, we would prefer an aisle seat in business class with caviar. Using a quantitative approach, it is possible that a flight to Hong Kong that has an aisle seat in business class with caviar is calculated to have higher priority than a flight to Toronto without any of the other preferences. It is obvious here, however, that reaching Toronto is infinitely more important than the other preferences. Using fuzzy logic, or any other quantitative framework, there is no way to guarantee that flights to Toronto are given higher priority than flights to any other destination. In general, there is no way to specify preference by an infinite degree using a quantitative approach.

Another quantitative approach is presented in [6] where Agrawal and Wimmers, specify preferences using scoring functions. A priority level is represented by a real value from [0, 1] where a score of 1 indicates the highest level of user preference and 0 indicates the lowest level of user preference. Then, a numeric score is calculated using preference functions and associated with every tuple of the query solution. A tuple with a higher score is preferred to a tuple with a lower score. As with fuzzy logic, the drawback of this model is that it is quantitative, and, therefore, is strictly less general than other qualitative approaches such as [2] and our approach. Specifically, it is impossible to numerically express that one constraint is infinitely more (or less) important than another constraint.

2.2 Our Approach

In our approach, the underlying basis of infinitesimal logic provides a general and flexible representation since it is purely logic based. It overcomes the expressive limitations imposed by classical two-valued and three-valued logics and serves as a better technique for addressing preference and uncertainty issues. Moreover, infinitesimal logic naturally allows us to define a hierarchy of priorities among constraints such that elements that are at a higher truth level are infinitely more preferable to elements at a lower truth level.

The addition of operators μ and ω allow an effective way to assert and distinguish between preferential and backup constraints. The representation is highly expressive while notationally simple.

Our lazy evaluation algorithm for the evaluation of prioritized constraint expressions is straightforward to implement. Also, it is general enough so that it may be adapted for evaluating any multiple-valued logic expressions.

Chapter 3

Priority Assertion

In order for any database to return accurate results for a query, the user must be able to accurately express what they are searching for. This includes being able to assert priorities among the query constraints. In this chapter, we present a notationally simple framework that allows users to formulate query expressions involving prioritized constraints. We introduce two unary operators, μ and ω, to infinitesimal logic in order to define preferential constraints and backup constraints, respectively. In section 3.1, we present the basic ideas behind infinitesimal logic. Then, in section 3.2, we present the operators, μ and ω and in section 3.3, we show how this framework allows us to assert priorities.

3.1 Infinitesimal Logic

Classical logic is based on two truth values, *True* and *False*. In order to obtain an accurate model for priority assertion, it is necessary to consider a logic which allows additional truth values of differing strengths.

In [1], Rondogiannis and Wadge present a infinite-valued logic called infinitesimal logic which allows them to define, in a logical way, the meaning of negation-as-failure and to clearly distinguish it from classical negation. In this domain, infinitesimal logic

is used to obtain a purely declarative semantics for logic programs with negation-as-failure in their clause bodies. It is argued that infinitesimal logic can also be used in contexts requiring priority assertion.

Infinitesimal logic, as presented in [1], has an uncountable linearly ordered set of truth values between the classical values $False$ and $True$, with a $Zero$ element in the middle. The values are ordered as follows.

$$F_0 < F_1 < \cdots < F_\alpha < \cdots < 0 < \cdots < T_\alpha < \cdots < T_1 < T_0 \tag{3.1}$$

The infinite set of truth values is obtained by successive multiplication of the classical truth values by an infinitesimal, ϵ[1]. Intuitively, the $False$ and $True$ values of classical logic are represented here by F_0 and T_0, respectively, and 0 captures the notion of $undefined$. An intermediate truth value T_k represents the value $\epsilon^k T$ and, similarly, F_k represents the value $\epsilon^k F$. The new values allow us to express different levels of truthfulness and falsity. The meaning of ϵ is translated to the logic domain to denote that $\epsilon^k T$ is infinitely truer than $\epsilon^{k+1} T$ and, similarly, $\epsilon^k F$ is infinitely falser than $\epsilon^{k+1} F$. That is, ϵ mitigates (by an infinite amount) the infinitesimal truth value to which it is applied.

Note that in [1], α runs through the countable ordinals, not just the natural numbers. This is necessary for their work in obtaining a minimum model semantics for logic programs with negation. For our purposes, however, we will only use natural numbers for the α values.

We define the logical operator connectives *and* and *or* in the context of infinitesimal logic as

$$A \text{ and } B = minimum(A, B) \tag{3.2}$$

where $minimum(A, B)$ is the least true of the infinitesimal truth values A and B and

$$A \text{ or } B = maximum(A, B) \tag{3.3}$$

[1]If we use only finite indices, then any fractional value can be used in the place of ϵ and the anomalies noted with the quantitative approaches in section 2.1 do not arise.

where $maximum(A, B)$ is the most true of the infinitesimal truth values A and B. We define the operator *not* as

$$not\ T_n = F_n \quad , \quad not\ F_n = T_n \quad , \quad not\ 0 = 0 \tag{3.4}$$

As in classical Boolean logic, the *and* and *or* operators satisfy the idempotent, commutative, associative and distributive laws.

Idempotent:

$$\begin{aligned} A\ and\ A &= A \\ A\ or\ A &= A \end{aligned} \tag{3.5}$$

Commutative:

$$\begin{aligned} A\ and\ B &= B\ and\ A \\ A\ or\ B &= B\ or\ A \end{aligned} \tag{3.6}$$

Associative:

$$\begin{aligned} (A\ and\ B)\ and\ C &= A\ and\ (B\ and\ C) \\ (A\ or\ B)\ or\ C &= A\ or\ (B\ or\ C) \end{aligned} \tag{3.7}$$

Distributive:

$$\begin{aligned} A\ and\ (B\ or\ C) &= (A\ and\ B)\ or\ (A\ and\ C) \\ A\ or\ (B\ and\ C) &= (A\ or\ B)\ and\ (A\ or\ C) \end{aligned} \tag{3.8}$$

The *not* operator has highest precedence, followed by the *and* operator. The *or* operator has lowest precedence. In an expression involving operators from more than one level in the precedence hierarchy, operators with higher precedence are executed

first. In an expression involving operators at the same level in the precedence hierarchy, left-to-right associativity is assumed. Besides these implicit precedence rules, parenthesis may be used to explicitly indicate the priority of execution in an expression.

3.2 The Operators μ and ω

In [1], it is suggested that infinitesimal logic could be used in contexts requiring priority assertion. In our approach, we achieve this by introducing two unary operators, μ and ω, to infinitesimal logic in order to define preferential constraints and backup constraints, respectively. First, we give definitions for the operators μ and ω in terms of the infinitesimal logic framework. We then show some properties of the operators and discuss how they may be used to assert priorities.

3.2.1 Definition and Theorems

First, let us formally define μ and ω as follows.

Definition 3.2.1. *For any infinitesimal truth value, A,*

$$\mu A = (A \; or \; \epsilon A)$$

Definition 3.2.2. *For any infinitesimal truth value, A,*

$$\omega A = (A \; and \; \epsilon A)$$

The operators, μ and ω have the same precedence as the *not* operator.

3.2.1.1 Multiplication

We show that multiplication by μ promotes false values by one degree while leaving true values unchanged.

Theorem 3.2.3.

$$\mu F_n = F_{n+1}$$

Proof.

$$\begin{aligned}\mu F_n &= F_n \; or \; \epsilon F_n \\ &= F_n \; or \; F_{n+1} \\ &= F_{n+1}\end{aligned}$$

□

Theorem 3.2.4.

$$\mu T_n = T_n$$

Proof.

$$\begin{aligned}\mu T_n &= T_n \; or \; \epsilon T_n \\ &= T_n \; or \; T_{n+1} \\ &= T_n\end{aligned}$$

□

Similarly, we show that multiplication by ω promotes false values by one degree while leaving true values unchanged.

Theorem 3.2.5.

$$\omega T_n = T_{n+1}$$

Proof.

$$\begin{aligned}\omega T_n &= T_n \; and \; \epsilon T_n \\ &= T_n \; and \; T_{n+1} \\ &= T_{n+1}\end{aligned}$$

□

Theorem 3.2.6.

$$\omega F_n = F_n$$

Proof.

$$\begin{aligned}\omega F_n &= F_n \; and \; \epsilon F_n \\ &= F_n \; and \; F_{n+1} \\ &= F_n\end{aligned}$$

□

Now, let us prove that multiplication by the operators μ and ω is cumulative.

Theorem 3.2.7. *For any infinitesimal truth value, A,*

$$\mu^k A = (A \; or \; \epsilon^k A) \quad for \;\; k \geq 1$$

Proof. Let A be an infinitesimal truth value.

Base Case: When k = 1, $\mu^1 A = \mu A = (A \; or \; \epsilon A)$ from the definition.

Inductive Hypothesis: Assume that $\mu^i A = (A \; or \; \epsilon^i A)$ is true for some $1 \leq i \leq k$.

Inductive Step:

$$\begin{aligned}\mu^{i+1} A &= \mu \; \mu^i (A \; or \; \epsilon A) \\ &= \mu (A \; or \; \epsilon^i A)\end{aligned}$$

Now there are two cases.

Case 1: If $A = T_x$.

$$\begin{aligned}\mu^{i+1} T_x &= \mu (T_x \; or \; \epsilon^i T_x) \\ &= \mu (T_x \; or \; T_{x+i}) \\ &= \mu T_x \\ &= T_x \\ &= A\end{aligned}$$

Case 2: If $A = F_x$.

$$\begin{aligned}\mu^{i+1}F_x &= \mu(F_x \; or \; \epsilon^i F_x) \\ &= \mu(F_x \; or \; F_{x+i}) \\ &= \mu F_{x+i} \\ &= F_{x+i+1} \\ &= \epsilon^{i+1} F_x \\ &= \epsilon^{i+1} A\end{aligned}$$

So, combining the two cases gives us

$$\mu^{i+1}A = (A \; or \; \epsilon^{i+1}A)$$

. *Conclusion:* Therefore, by the principal of mathematical induction, $\mu^k A = (A \; or \; \epsilon^k A)$ for $k \geq 1$. □

Theorem 3.2.8.

$$\omega^k A = (A \; and \; \epsilon^k A) \quad for \;\; k \geq 1$$

Proof. *Base Case:* When k = 1, $\omega^1 A = \mu A = (A \; and \; \epsilon A)$ from the definition. *Inductive Hypothesis:* Assume that $\omega^i A = (A \; and \; \epsilon^i A)$ is true for some $1 \leq i \leq k$. *Inductive Step:*

$$\begin{aligned}\omega^{i+1}A &= \omega \; \omega^i (A \; and \; \epsilon A) \\ &= \omega(A \; and \; \epsilon^i A)\end{aligned}$$

Now there are two cases.

Case 1: If $A = F_x$.

$$\begin{aligned}\omega^{i+1}F_x &= \omega(F_x \; and \; \epsilon^i F_x) \\ &= \omega(F_x \; and \; F_{x+i}) \\ &= \omega F_x \\ &= F_x \\ &= A\end{aligned}$$

Case 2: If $A = T_x$.

$$\begin{aligned}\omega^{i+1}T_x &= \omega(T_x \; and \; \epsilon^i T_x) \\ &= \omega(T_x \; and \; T_{x+i}) \\ &= \omega T_{x+i} \\ &= T_{x+i+1} \\ &= \epsilon^{i+1} T_x \\ &= \epsilon^{i+1} A\end{aligned}$$

So, combining the two cases gives us

$$\omega^{i+1}A = (A \; and \; \epsilon^{i+1}A)$$

. *Conclusion:* Therefore, by the principal of mathematical induction, $\omega^k A = (A \; and \; \epsilon^k A)$ for $k \geq 1$. □

Now we can show that μ^k promotes false values by k degrees, while leaving true values unchanged.

Theorem 3.2.9.

$$\mu^k F_n = F_{n+k}$$

Proof.

$$\begin{aligned}\mu^k F_n &= F_n \text{ or } \epsilon^k F_n \\ &= F_n \text{ or } F_{n+k} \\ &= F_{n+k}\end{aligned}$$

□

Theorem 3.2.10.

$$\mu^k T_n = T_n$$

Proof.

$$\begin{aligned}\mu^k T_n &= T_n \text{ or } \epsilon^k T_n \\ &= T_n \text{ or } T_{n+k} \\ &= T_n\end{aligned}$$

□

On the other hand, the following shows that ω^k promotes true values by k degrees while leaving false values unchanged.

Theorem 3.2.11.

$$\omega^k T_n = T_{n+k}$$

Proof.

$$\begin{aligned}\omega^k T_n &= T_n \text{ and } \epsilon^k T_n \\ &= T_n \text{ and } T_{n+k} \\ &= T_{n+k}\end{aligned}$$

□

Theorem 3.2.12.

$$\omega^k F_n = F_n$$

Proof.

$$\begin{aligned}\omega^k F_n &= F_n \text{ and } \epsilon^k F_n \\ &= F_n \text{ and } F_{n+k} \\ &= F_n\end{aligned}$$

□

Informally, we can say that μ has the effect of mitigating failure and ω has the effect of mitigating success. For completeness, note that $\mu^k 0 = \omega^k 0 = \epsilon^k 0 = 0$.

Now, we show that multiplication by μ or ω preserves the ordering on infinitesimal truth values.

Theorem 3.2.13. *Given infinitesimal truth values, V_1 and V_2, if $V_1 < V_2$, then*

1. $\mu^k V_1 < \mu^k V_2$
2. $\omega^k V_1 < \omega^k V_2$

We prove 1. Statement 2 can be proved similarly.

Proof of 1. There are 5 cases.

Case 1: If $V_1 = F_x$ and $V_2 = F_y$, where $x < y$.
$\mu^k V_1 = \mu^k F_x = F_{x+k}$
$\mu^k V_2 = \mu^k F_y = F_{y+k}$
Since $x < y$, we know that $F_{x+k} < F_{y+k}$. Therefore, $\mu^k V_1 < \mu^k V_2$.

Case 2: If $V_1 = F_x$ and $V_2 = T_y$.
$\mu^k V_1 = \mu^k F_x = F_{x+k}$
$\mu^k V_2 = \mu^k T_y = T_y$
We know that $F_{x+k} < T_y$. Therefore, $\mu^k V_1 < \mu^k V_2$.

Case 3: If $V_1 = F_x$ and $V_2 = 0$.

$\mu^k V_1 = \mu^k F_x = F_{x+k}$

$\mu^k V_2 = \mu^k 0 = 0$

We know that $F_{x+k} < 0$. Therefore, $\mu^k V_1 < \mu^k V_2$.

Case 4: If $V_1 = 0$ and $V_2 = T_y$.

$\mu^k V_1 = \mu^k 0 = 0$

$\mu^k V_2 = \mu^k T_y = T_y$

We know that $0 < T_y$. Therefore, $\mu^k V_1 < \mu^k V_2$.

Case 5: If $V_1 = T_x$ and $V_2 = T_y$, where $x > y$.

$\mu^k V_1 = \mu^k T_x = T_x$

$\mu^k V_2 = \mu^k T_y = T_y$

Since $x > y$, we know that $T_x < T_y$. Therefore, $\mu^k V_1 < \mu^k V_2$.

This proves that, if $V_1 < V_2$, then $\mu^k V_1 < \mu^k V_2$. □

3.2.1.2 Division

Formally, division by μ and division by ω can be defined as follows.

Definition 3.2.14.

$$T_n/\mu = T_n$$

$$F_n/\mu = \begin{cases} F_{n-1} & \text{for } n > 1 \\ \\ F_0 & \text{for } n \leq 1 \end{cases}$$

Definition 3.2.15.

$$F_n/\omega = F_n$$

$$T_n/\omega = \begin{cases} T_{n-1} & \text{for } n > 1 \\ \\ T_0 & \text{for } n \leq 1 \end{cases}$$

Similarly to the proof for multiplication, it can be proved that division by μ^k and division by ω^k is cumulative.

Theorem 3.2.16.

$$T_n/\mu^k = T_n$$

$$F_n/\mu^k = \begin{cases} F_{n-k} & \text{for } n > k \\ \\ F_0 & \text{for } n \leq k \end{cases}$$

Theorem 3.2.17.

$$F_n/\omega^k = F_n$$

$$T_n/\omega^k = \begin{cases} T_{n-k} & \text{for } n > k \\ \\ T_0 & \text{for } n \leq k \end{cases}$$

3.2.1.3 Duality

Now, we show that μ and ω are dual operators of each other.

Theorem 3.2.18. *For any infinitesimal truth value, A,*

$$\mu^k \omega^k A = \omega^k \mu^k A = \epsilon^k A$$

First, we prove the following lemma.

Lemma 3.2.19. *For any infinitesimal truth value, A,*

$$\mu^k \omega^k A = \epsilon^k A$$

Proof. Let A be an infinitesimal truth value. There are 2 cases.

Case 1: If $A = T_x$.

$$\begin{aligned} \mu^k \omega^k A &= \mu^k \omega^k T_x \\ &= \mu^k T_{x+k} \\ &= T_{x+k} \\ &= \epsilon^k T_x \\ &= \epsilon^k A \end{aligned}$$

Case 2: If $A = F_x$.

$$\begin{aligned} \mu^k \omega^k A &= \mu^k \omega^k F_x \\ &= \mu^k F_x \\ &= F_{x+k} \\ &= \epsilon^k F_x \\ &= \epsilon^k A \end{aligned} \tag{3.9}$$

□

Similarly, we prove the following lemma.

Lemma 3.2.20. *For any infinitesimal truth value, A,*

$$\omega^k \mu^k A = \epsilon^k A$$

Proof. Let A be an infinitesimal truth value. There are 2 cases.

Case 1: If $A = T_x$.

$$\begin{aligned}\omega^k \mu^k A &= \omega^k \mu^k T_x \\ &= \omega^k T_x \\ &= T_{x+k} \\ &= \epsilon^k T_x \\ &= \epsilon^k A\end{aligned}$$

Case 2: If $A = F_x$.

$$\begin{aligned}\omega^k \mu^k A &= \omega^k \mu^k F_x \\ &= \omega^k F_{x+k} \\ &= F_{x+k} \\ &= \epsilon^k F_x \\ &= \epsilon^k A\end{aligned} \tag{3.10}$$

□

Again, for completeness, it is not hard to prove that $\omega^k \mu^k 0 = \mu^k \omega^k 0 = \epsilon 0 = 0$. From this fact, combined with lemmas 3.2.19 and 3.2.20, we can conclude theorem 3.2.18.

We will call expressions involving infinitesimal logic and the operators μ and ω prioritized constraint expressions (PCEs).

3.3 Asserting Priorities using PCEs

From the above definitions, we can say that μ has the effect of mitigating failure and ω has the effect of mitigating success.

For example, let A, B, and C represent propositions. Then the expression

$$A \textit{ and } \mu B \textit{ and } \mu^2 C$$

means that A is required and B and C are preferred (but not required). The degree of μ specifies the degree of preference, where a higher degree of μ corresponds to a lower degree of preference. If A, B, and C are true then the expression evaluates to T. However, suppose A and B are true, but C is false. Then we get a result of F_2, a very modest failure. Similarly, we get a modest failure, F_1, if B is false and complete failure, F_0, if A is false.

On the other hand, the dual operator ω specifies backup options. For example, the expression

$$A \; or \; \omega B \; or \; \omega^2 C$$

means that we would like A but, if that is not possible, we could settle for B or C. The degree of ω specifies the order for the backup options. It can be shown that if A is true, we get total success, T_0. Otherwise, if A is false, but B is true, we get partial success, T_1. If A and B are false, but C is true, we get modest success, T_2.

Example 3.3.1.

Now, we reconsider the flight search example presented in section 1.1 and formulate a PCE for the problem. The search criteria is that the destination is Toronto, ON and that we would like a window seat. Therefore, the destination is a required constraint and the seat preference is a preferential constraint. This can be expressed as follows.

$$(destination = Toronto, ON) \quad and \quad \mu(seat = window)$$

Example 3.3.2.

Let us consider another problem. For example, suppose that we are given the data shown in figure 3.1 about apartments for rent and suppose that we have the following constraints.

1. We require a 2 bedroom apartment.

2. We would prefer 2 bathrooms.

Apt. ID	Num Bedrooms	Num Bathrooms	Rent Price
204	3	2	890
205	1	1	550
206	2	2	650
207	2	1	600
208	1	1	500
209	2	2	750
210	3	1	750

Figure 3.1: Example apartment data.

3. The price should be less than \$700 per month, but if there are no apartments available at that cost, then we can afford a price of at most \$800 per month.

In this case, the required constraints are that the apartment should have 2 bedrooms and the price should be lower than \$700. Having 2 bathrooms is a preferential constraint and a price lower than \$800 is a backup constraint. Let us define the propositions A, B, C and D as follows.

A: The apartment has 2 bedrooms.

B: The apartment has 2 bathrooms.

C: The price is less than \$700 per month.

D: The price is less than \$800 per month.

Then the search criteria can be represented by the expression $A \textit{ and } \mu B \textit{ and } (C \textit{ or } \omega D)$. Figure 3.2 shows the data table sorted in descending order by the resulting truth values for evaluating the apartments using this query string. We can clearly see from this evaluation that apartment 206 is our best option, followed by apartments 209 and 207. The other apartments fail to satisfy our required criteria and our backup options.

Example 3.3.3.

Apt. ID	Num Bedrooms	Num Bathrooms	Rent Price	Truth Value
206	2	2	650	T_0
209	2	2	750	T_1
207	2	1	600	F_1
204	3	2	890	F_0
205	1	1	550	F_0
208	1	1	500	F_0
210	3	1	750	F_0

Figure 3.2: Example apartment data with evaluations.

Option	Depart	Return	Num Stopovers
1	May 10	May 17	1
2	May 10	May 15	0
3	May 10	May 17	0
4	May 11	May 15	1
5	May 10	May 16	0
6	May 9	May 17	0
7	May 12	May 16	1

Figure 3.3: Example data for flights to Toronto.

Now, consider the data shown in figure 3.3 and suppose we have the following flight search criteria.

1. We require 2 tickets for a round-trip flight from Victoria, BC to Toronto, ON leaving on May 10 and returning on May 15.

2. If there are no seats available for return on May 15, we would be willing to return on May 16, or even May 17.

3. We would prefer to get a direct flight if possible.

Let variables *numpassengers*, *origin*, *destination*, *numstopovers*, *depart* and *return* represent the number of passengers, the origin and destination of travel, the number of stop-overs, and the departure and return date, respectively. Let us define the propositions A, B, C, D and E as follows.

A : *numpassengers* = 2 *and origin* = Victoria, BC *and destination* = Toronto, ON *and depart* = May 10

B : *return* = May 15

C : *return* = May 16

D : *return* = May 17

E : *numstopovers* = 0

Then our flight search can be expressed as A *and* $(B$ *or* ωC *or* $\omega^2 D)$ *and* μE, giving us the evaluation table shown in figure 3.4 sorted in descending order by the

Option	Depart	Return	Num Stopovers	Truth Value
2	May 10	May 15	0	T_0
5	May 10	May 16	0	T_1
3	May 10	May 17	0	T_2
1	May 10	May 17	1	F_1
4	May 11	May 15	1	F_0
6	May 9	May 17	0	F_0
7	May 12	May 16	1	F_0

Figure 3.4: Example data for flights to Toronto with evaluations.

infinitesimal truth values calculated for the options. From this table, we see that option 2 is our best choice followed by options 5, 3, and 1.

In this way, the operators μ and ω combined with the infinitesimal logic framework allow us to naturally define a hierarchy of priorities among constraints in a simple and elegant fashion. In the next chapter, we show how our lazy algorithm for evaluation of the PCEs allows us to easily associate an infinitesimal truth value with each record to clearly see a ranking of our options.

Chapter 4

The Lazy Evaluation of a PCE

The basic principle of lazy evaluation is to evaluate only the parts of the expression that you need in order to determine its final value [7]. For example, consider the lazy evaluation of Boolean expressions. Given a Boolean expression of the form

$$B_1 \; and \; B_2 \; and \; \cdots \; and \; B_n$$

we know that it evaluates to $False$ if at least one B_i evaluates to $False$, and $True$ otherwise. Therefore, we can immediately return $False$ after the first B_i that evaluates to $False$ without evaluating the remainder of the expression [8]. Otherwise, we return $True$ if all B_is evaluate to $True$. Similarly, given a Boolean expression of the form

$$B_1 \; or \; B_2 \; or \; \cdots \; or \; B_n$$

we know that it evaluates to $True$ if at least one B_i evaluates to $True$, and $False$ otherwise. Therefore, we can immediately return $True$ after the first B_i that evaluates to $True$, or return $False$ if all B_is evaluate to $False$ [8].

The lazy evaluation of a prioritized constraint expression (PCE), on the other hand, is not as straightforward as the lazy evaluation of a Boolean expression since there can be an arbitrary number of truth values. In section 4.1, we present a recursive algorithm for the lazy evaluation of PCEs based on a depth-first traversal of the parse

tree for the expression. In section 4.3, we compare our algorithm with the alpha-beta pruning technique used for minimax game tree evaluation. Appendix A contains the Prolog source code for an evaluator for PCEs. This evaluator implements standard evaluation and also the lazy evaluation technique presented in this chapter.

4.1 Lazy Evaluation

Our algorithm is based on the observation that, given a conjunction of propositions A and B, we are only interested in the exact value of B if B is less true (or more false) than A. Conversely, given a disjunction of A and B, we are only interested in the exact value of B if B is more true (or less false) than A.

To determine which parts of a PCE do not need to be evaluated, we must consider the parse tree for the PCE. The leaves of the parse tree correspond to the terminal operands and the internal nodes represent the operators. The *and* and *or* nodes have exactly two children since they are binary operators. The operators nodes for *not*, μ and ω have exactly one child since they are unary operators.

We associate two parameters, *min* and *max*, with each operator node to denote the lower and upper bound, respectively, of the range of truth values for the subexpression rooted at that node that can affect the final value of the main expression. We evaluate the expression by a recursive depth-first traversal of the parse tree. Following is the pseudocode for the recursive algorithm.

```
EVAL(node, min, max)
 1  if TYPE_LEAF(node.type)
 2     then return VAL(node)
 3  elseif TYPE_AND(node.type)
 4     then V_left = EVAL(node.left, min, max)
 5          if V_left ≤ min
 6             then return min
 7             else  V_right = EVAL(node.right, min, V_left)
 8                   return AND(V_left, V_right)
 9  elseif TYPE_OR(node.type)
10     then V_left = EVAL(node.left, min, max)
11          if V_left ≥ max
12             then return max
13             else  V_right = EVAL(node.right, V_left, max)
14                   return OR(V_left, V_right)
15  elseif TYPE_NOT(node.type)
16     then return NOT(EVAL(node.child, NOT(max), NOT(min)))
17  elseif TYPE_M(node.type)
18     then deg = GETDEG(node)
19          return μ^deg(EVAL(node.child, min/μ^deg, max/μ^deg))
20  elseif TYPE_W(node.type)
21     then deg = GETDEG(node)
22          return ω^deg(EVAL(node.child, min/ω^deg, max/ω^deg))
```

We initialize the current node to be the root node of the expression. When we begin evaluation, the range of possible values for the expression includes all infinitesimal truth values so we initialize the current node's min and max to F_0 and T_0, respectively. Then we evaluate the tree recursively. If the current node is a leaf node, then we simply return its value. Otherwise, we evaluate the node according to its type.

Note that each node is visited at most once during the evaluation, so we have a time complexity of $O(n)$ in the worst case, where n is the number of nodes in the parse tree for the expression.

4.1.1 Evaluation of an *and* Node

At an *and* node, first we recursively evaluate the left child of the current node. Then if the left child's value, V_{left}, is less or equally true than the current node's *min* value, we simply return *min* without evaluating the right child. This is because, since this is an *and* node, the right child cannot make the value of the subexpression rooted at the current node more true than V_{left}. Therefore, the value of the current node is outside the range of values that can affect the final value of the expression.

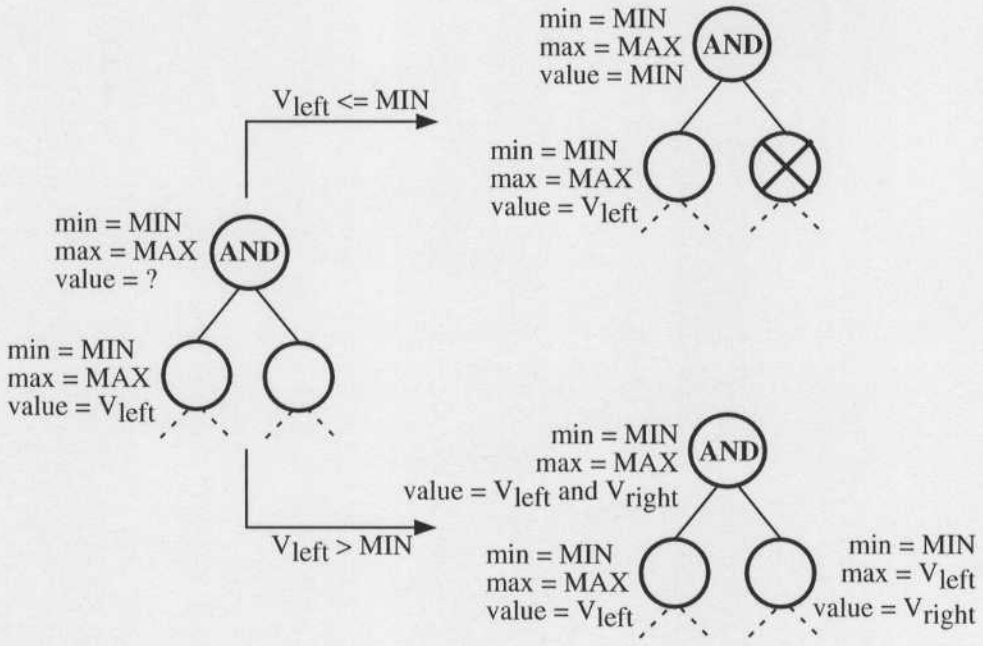

Figure 4.1: Evaluation of an *and* node.

On the other hand, if V_{left} is more true than the current node's *min*, we must recursively evaluate the right child. The *min* value of the right child is set to the

same as that of the current node. However, we must set the *max* value of the right child to V_{left} since values that are more true than that will not affect the current node's value. This is because, being an *and* node, the current node's value cannot exceed the value of the left child. After obtaining the right child's value, V_{right}, we return V_{left} *and* V_{right} as the value of the current node.

4.1.2 Evaluation of an *or* Node

Similarly, at an *or* node, if the left child's value, V_{left}, is more or equally true than the current node's *max* value, we simply return *max* without evaluating the right child.

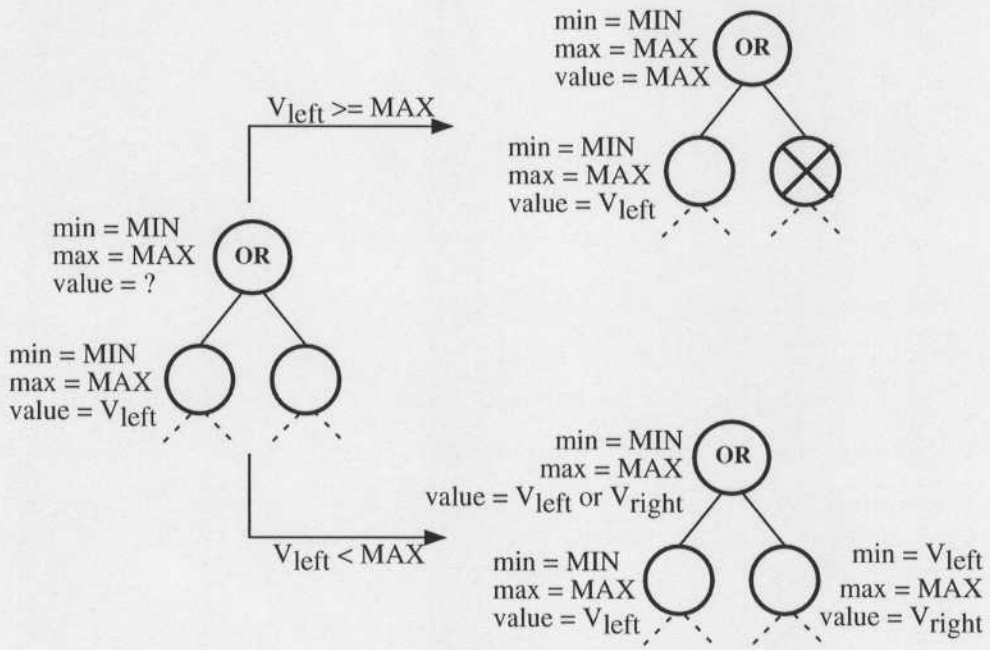

Figure 4.2: Evaluation of an *or* node.

However, if V_{left} is less true than the current node's *max*, we must recursively evaluate the right child. The *max* value of the right child is set to the same as that of the current node. However, we must set the *min* value of the right child to V_{left} since, being an *or* node, the current node's value cannot be less than the value of the

left child. After obtaining the right child's value, V_{right}, we return V_{left} *or* V_{right} as the value of the current node.

4.1.3 Evaluation of a *not* Node

At a *not* node, we must evaluate its child node with modified *min* and *max* values. Suppose we denote the *min* and *max* values of the current node by $curr_{min}$ and $curr_{max}$. Then we recursively evaluate the child node by setting the *min* to *not* $curr_{max}$ and the *max* to *not* $curr_{min}$, respectively. After obtaining the child node's value, V_{child}, we return *not* V_{child} as the value of the current node.

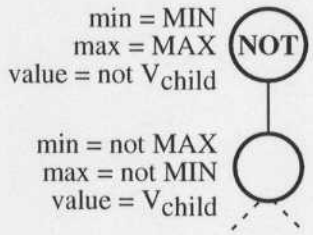

Figure 4.3: Evaluation of a *not* node.

4.1.4 Evaluation of μ and ω Nodes

As with the *not* node, at a μ or ω node, we must evaluate the child node with modified *min* and *max* values. Recall that a μ and ω operator simply increment the degree of the operand's truth value. For calculating the new *min* and *max* values for evaluating the child, we need to "undo" the affect of the current node's operator. That is, instead of multiplying by ϵ^k, we divide the *min* and *max* values by ϵ^k.

Then to evaluate the child of a μ^k node, we execute EVAL(*node.child*, min/μ^k, max/μ^k). After obtaining the child node's value, V_{child}, we return μ^k V_{child} as the value of the current node. Similarly, to evaluate the child of a ω^k node, we execute EVAL(*node.child*, min/ω^k, max/ω^k) and return ω^k V_{child} as the value of the current node.

4.2 Correctness of the Lazy Evaluation Algorithm

In this section, we prove that pruning the evaluation tree during the execution of our lazy evaluation algorithm produces the same value that would be calculated without any pruning. First, we note that pruning, if any, is only done at *and* and *or* nodes when we determine that the value of the node's right child will be outside the range of values that can affect the final value of the expression. Let us call this range the *range of interest*. The range of interest is defined by the node's *min* and *max* values, which is passed down recursively from the parent node to the child node(s). Therefore, in order to prove correctness of the algorithm, we must prove that each node has the correct *min* and *max* values, in turn defining the corect range of interest. We prove this using induction on the depth, n, of the parse tree for the expression.

Proof. *Base Case:* At depth 0, we have just the root node. At the root node, $min = F_0$ and $max = T_0$, specifying that the range of interest is the whole range of infinitesimal truth values. And we know that the final value of the expression must lie within this range. Therefore, at depth 0, we have the correct *min* and *max* values.

Induction Hypothesis: Assume that, for some depth $0 \leq k \leq n$, the *min* and *max* values for each node define the correct range of interest.

Induction Step: From the induction hypothesis, we can assume that at depth k, all nodes have the correct *min* and *max* values. We must show that all nodes at depth $k+1$ have the correct values also. Note that the nodes at depth $k+1$ must be children of the nodes at depth k. So we must show that for any node at depth k, the *min* and *max* values that it passes on to it's child node(s) are correct.

We consider different cases depending on the types of the nodes at depth k.

and node: At an *and* node, recall that for evaluating the left child, we simply pass the same *min* and *max* values. Note that the value of an *and* node is simply the minimum of the values of its two child nodes. Therefore, we know that values for the children must be at least within the range for the *and* node in order to affect the final value of the expression, because if they were outside that range, then their minimum (which is the the value of the *and* node) may be ouside that range. Therefore, we know that the left child of an *and* node is assigned the correct range of interest. Now we consider the right child. Given that the value evaluated for the left child is V_1, we know that the value of the *and* node must be at most V_1. Therefore, we know that the lower bound on the range of interest for the right child must be at most V_1. So the *min* and *max* values that are passed to the children of an *and* node are correct.

or node: Similarly, at an *or* node, for evaluating the left child, we simply pass the same *min* and *max* values. Note that the value of an *or* node is simply the maximum of the values of its two children nodes. Therefore, we know that values for the children must be at least within the range for the *or* node in order to affect the final value of the expression, because if they were outside that range, then their maximum (which is the the value of the *or* node) may be outside that range. Therefore, we know that the left child of an *or* node is assigned the correct range of interest. Now we consider the right child. Given that the value evaluated for the left child is V_1, we know that the value of the *or* node must be at least V_1. Therefore, we know that the upper bound on the range of interest for the right child must be at most V_1. So the *min* and *max* values that are passed to the children of an *or* node are correct.

not node: Recall that we do not do any pruning at a *not* node. Therefore, if the value of the *not* node is V, the value at the child node must have been *not* V. Given that the range of interest at the *not* node is defined by lower bound *min* and upper bound *max*, we know that $min \leq V$ and $V \leq max$. So, we know that

not min $\geq$ *not V* and *not V* $\geq$ *not max*. That is, *not max* $\leq$ *not V* $\leq$ *not min*. Therefore, the range of interest passed down to the child of a *not* node is defined by lower bound *not max* and upper bound *not min*.

μ node: Suppose the value of the μ^k node is V, and the range of interest is defined by lower bound *min* and upper bound *max*. This means that *min* $\leq$V$\leq$ *max*. Therefore, by theorem 3.2.13, we know that $\mu^k min \leq \mu^k V \leq \mu^k max$. So the range of interest passed down to the child of a μ^k node is defined by lower bound $\mu^k min$ and upper bound $\mu^k max$.

ω node: Suppose the value of the ω^k node is V, and the range of interest is defined by lower bound *min* and upper bound *max*. This means that *min* $\leq$V$\leq$ *max*. Therefore, by theorem 3.2.13, we know that $\omega^k min \leq \omega^k V \leq \omega^k max$. So the range of interest passed down to the child of a ω^k node is defined by lower bound $\omega^k min$ and upper bound $\omega^k max$.

Therefore, in each case, the value that is passed to the child node(s) is correct. That is, the *min* and *max* values of nodes at depth $k+1$ all define the correct range of interest.

Conclusion: Therefore, by the principal of mathematical induction, the *min* and *max* values for each node of the parse tree define the correct range of interest for that node. □

4.3 Comparison of Alpha-Beta Pruning and the Lazy Evaluation of PCEs

The algorithm presented in this paper for the lazy evaluation of PCEs is analogous to alpha-beta pruning used for minimax game trees. In this section, we briefly describe

the alpha beta pruning algorithm. Then we compare it with our algorithm for the lazy evaluation of PCEs.

4.3.1 Alpha-Beta Pruning

Minimax game trees are used to determine the best playing strategy in a two-player game, where one player strives to maximize the score and the other player strives to minimize the score. A node of a game tree represents a position or state of the game and a value is associated with each node. The children of a node represent the possible moves a player may make from the current position. The players take turns in making moves so we have alternating levels corresponding to each player's turn to make a move.

Alpha-beta pruning is a recursive algorithm used to reduce the number of nodes evaluated, while still producing the best strategy. Following is the pseudocode [9] for alpha-beta pruning.

```
ABPRUNING(node, alpha, beta)
 1  if TYPE_LEAF(node.type)
 2     then return VAL(node)
 3  elseif TYPE_MAXIMIZING(node.type)
       then for each child of node
 4            do alpha = MAXIMUM(alpha, ABPRUNING(node.child, alpha, beta))
 5               if beta ≤ alpha
 6                  then return beta
 7          return alpha
 8  elseif TYPE_MINIMIZING(node.type)
       then for each child of node
 9            do alpha = MINIMUM(beta, ABPRUNING(node.child, alpha, beta))
10               if beta ≤ alpha
11                  then return alpha
12          return beta
```

Two values. *alpha* and *beta*, are maintained to represent the minimum score that the maximizing player is assured of and the maximum score that the minimizing player is assured of, respectively. Initially, *alpha* is set to $-infinity$ and *beta* is set to $infinity$. As recursion progresses, the range represented by *alpha* and *beta* becomes smaller. Whenever *beta* is not greater than *alpha*, the current position cannot be the result of best play by both players. Therefore, the subtree associated with the current node need not be explored any further.

4.3.2 Comparison

Our lazy evaluation algorithm, particularly the evaluation of leaf, *and* and *or* nodes, works in a similar way to alpha-beta pruning. We have two parameters, *min* and *max* to define the range of truth values that can affect the final value of the expression. As the recursion progresses, the range becomes smaller. Whenever the lower bound,

min, exceeds the upper bound, *max*, we know that the current node's value is not within the range that can affect the final value of the expression. Therefore, the subexpression rooted at the current node need not be evaluated.

Now, we rewrite the part of the algorithm for the lazy evaluation of PCEs that handles the leaf, *and* and *or* nodes so we can more clearly see the resemblance to the alpha-beta pruning algorithm.

```
EVAL(node, min, max)
 1  if TYPE_LEAF(node.type)
 2     then return VAL(node)
 3  elseif TYPE_AND(node.type)
 4     then max = MAXIMUM(min, EVAL(node.left, min, max))
 5          if max ≤ min
 6             then return max
 7          return AND(max, EVAL(node.right, min, max))
 8  elseif TYPE_AND(node.type)
 9     then max = MAXIMUM(min, EVAL(node.left, min, max))
10          if max ≤ min
11             then return max
12          return AND(max, EVAL(node.right, min, max))
            ⋮
```

Notice the unnumbered lines in the pseudocode for ABPRUNING. They are the parts of the algorithm that set up the loop for handling all of the children of the current node. There is no corresponding line in our pseudocode for EVAL since each node has at most 2 children, so no loop is needed.

Now, we note that both algorithms are recursive and take three parameters: the current node, followed by two values, *min* and *max* for EVAL, and *alpha* and *beta* for ABPRUNING. Also, note that *min* and *alpha* are both initialized to the minimum

possible value and *max* and *beta* are both initialized to the maximum possible value in their corresponding ranges of values.

Next, observe lines 3-7 of both algorithms. Considering that *min* corresponds to *alpha* and *max* corresponds to *beta*, we see that the part of the algorithm that handles a maximizing node ABPRUNING is extremely similar to the part of the algorithm that handles an *and* node in EVAL. There are just two differences. On line 4, we see that MAXIMUM(*alpha*, ABPRUNING(*child*, *alpha*, *beta*)) has direct correspondence to MAXIMUM(*min*, EVAL(*child*, *min*, *max*)). In ABPRUNING, this value is assigned to *alpha* which corresponds to *min* in EVAL. However, we see that this value is assigned to *max* in EVAL. This is because, in a minimax tree for alpha-beta pruning, we have alternating levels corresponding to each player trying to either minimize or maximize the score, depending on the level. Whereas, in a parse tree for lazy evaluation, there is no concept of alternating levels. Also, on line 8, ABPRUNING returns a value whereas EVAL returns the result of performing an *and*. This is because the internal nodes in the parse tree for EVAL are associated with operators, whereas game tree nodes are not.

Similarly, we can see from lines 8-12 of both algorithms that a minimizing node in ABPRUNING closely corresponds to an *or* node in EVAL.

Chapter 5

Prioritized Queries and the Movies Database Application

In this chapter, we discuss our implementation of our movies database application which allows users to specify prioritized queries using the framework presented in this thesis. In section 5.1, we briefly describe the database. Section 5.2 discusses the Java implementation of the front-end for the database and section 5.3 describes how to use the application to run prioritized queries on the movies database.

5.1 Database

The movies test data that we use was obtained from Gio Wiederhold's movies database [10]. Our data is stored in a MySQL database, *movies*, which consists of 4 tables:

1. ***main***: This is the main movies table and contains over 11 000 records. The field names are *filmid*, *title*, *year*, *director*, *producers*, *studios*, *prc*, *cat*, *awards*, *lc*, and *notes*.

2. ***actors***: This table contains information about actors and consists of 6816 records. The field names are *stagenm*, *dow*, *birthlnm*, *birthfnm*, *gender*, *dob*,

dod, *type*, *origin*, *photo* and *notes*.

3. ***people***: This table contains information about movie people other than actors, mainly directors and producers. There are about 3309 records. The field names are *idname*, *Pcode*, *Did*, *dow*, *lastnm*, *firstnm*, *dob*, *dod*, *backgrd* and *notes*

4. ***casts***: This table documents who acted in what role in which movie and contains 47 959 records. The field names are *filmid*, *title*, *actor*, *roletype*, *role*, *awards* and *notes*.

For more detailed information about the tables and data, please refer to the database documentation site [10].

5.2 Front-End Application

The front-end for our movies database prioritized query application is implemented as a Java Swing application. The driver class for the application is `MoviesDatabase.java`. This class also implements the GUI for the application.

The `ConditionGroup.java` class is used to store the condition information provided by the user. `TableMap.java` and `TableSorter.java` handle the events and updates on the query result table. Infinitesimal truth values are represented by objects of the `InfinitesimalTruthValue.java` class. `ParseTreeNode.java` parses the query string provided by the user in the form of a PCE and returns a parse tree for the expression. This class also implements the lazy evaluation algorithm presented in section 4.1 which is used to calculate the truth value for each row in the query result table. An exception that is an instance of `SyntaxError.java` is thrown if there are any parsing errors.

The application communicates with the MySQL database through the `JDBCAdapter.java` class using JDBC. The classes `Condition.java`, `ConditionValue.java` and `RowValue.java`

are used to store and retrieve condition information from the database. Initially, only the *SELECT* and *FROM* part of the query are executed to obtain a result set. Then, a parse tree is obtained for the expression in the *WHERE* clause. By performing the lazy evaluation algorithm on this parse tree, an infinitesimal truth value is computed for each row in the result set. This step has time complexity $O(nm)$ where n is the number of operators in the PCE and m is the number of rows in the result set. It takes approximately 5 to 8 seconds, depending on the complexity of the PCE, to return the results using our application[1]. The query result table is presented to the user with all of the results sorted in descending order by their infinitesimal truth value.

Appendix B contains the Java source code for `ParseTreeNode.java` and `InfinitesimalTruthValue.java`, as they are the classes that implement the main ideas behind our framework. The implementation of the other classes are fairly straightforward and, therefore, not included in the appendix.

5.3 User Interface

As shown in figure 5.1, the interface for the movies database application is divided into 4 main sections: *Connection*, *Query*, *QueryResult* and *Information*.

The main shortcoming of our current implementation is that the conditions and the PCE must be specified separately, as described in section 5.3.2. Ideally, the user should be able to incorporate the μ and ω operators into a single PCE which could be run as an SQL query. However, this would require the implementation of an extension to SQL to handle the parsing and evaluation of our priority operators. Since our application is still a research prototype and its current function is to test and demonstrate the expressiveness and flexibility of our framework, the task of implementing an extension

[1] A regular SELECT query in SQL on this movies database of 11 000 records, without using the hierarchical constraints of our application, takes around 1 second to execute on the same machine that we tested our movies database application. The increased query time is result of executing the evaluation algorithm for each row in the result set.

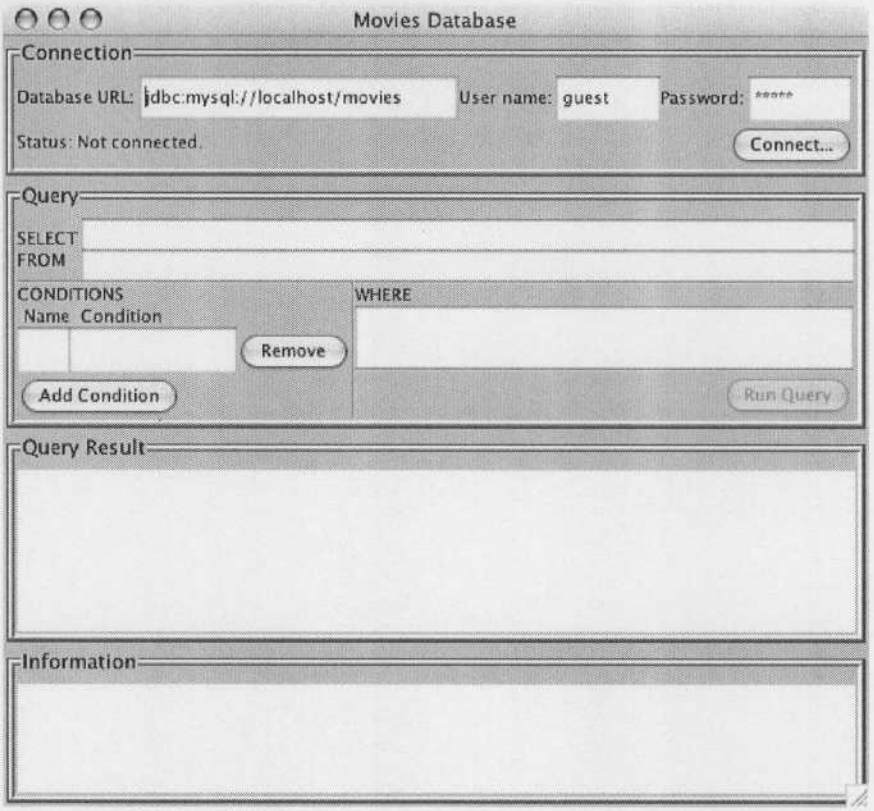

Figure 5.1: Initial View of the Movies Database application.

to SQL is left as future work.

5.3.1 Connecting to the Database

The *Connection* section allows users to specify the URL of the database that they wish to connect to and the username and password to access the database. Clicking on the *Connect* button attempts to establish a connection with the specified database.

5.3.2 Specifying a Query

Users may specifiy the search criteria in the *Query* section. The *SELECT* and *FROM* fields accept any expressions that would be valid in normal MySQL *SELECT*

and *FROM* clauses[2]. That is, the *SELECT* field specifies the column(s) to select from the table(s) specified in the *FROM* field. All constraints must be defined in the *CONDITIONS* area by specifying the name of the constraint in the *Name* field and the corresponding condition in the adjacent *Condition* field. A condition may be specified as any expression that would be valid in a normal MySQL *WHERE* clause. A user may add conditions by clicking on the *AddCondition* button. A condition may be removed my clicking on the adjacent *Remove* button. Then, in the *WHERE* field, a user may specify a PCE using the condition names specified in *CONDITIONS*. The `&` and `|` symbols represent the *and* and *or* operators, respectively. The *not*, μ and ω operators are represented by `!`, `m` and `w`, respectively. The `m` and `w` symbols may be followed by a positive number to specify the associated degree. Clicking on the *RunQuery* button executes the query and returns the results in the *Query Result* section.

5.3.3 Viewing the Results

The query result table consists of all the records specified by the *SELECT* and *FROM* fields. The last column shows the infinitesimal truth values calculated for each row according to the PCE specified in the *WHERE* field using the conditions specified in *CONDITIONS*. The results are sorted in descending order of the infinitesimal truth values, where `e` represents ϵ. Clicking on a column header in the query result table sorts the table by that column in descending order and pressing the `Shift` key while clicking on the column header sorts the table by that column in ascending order. Figure 5.2 shows a view of the application after running a query.

[2]For detailed documentation on MySQL syntax, please refer to [11].

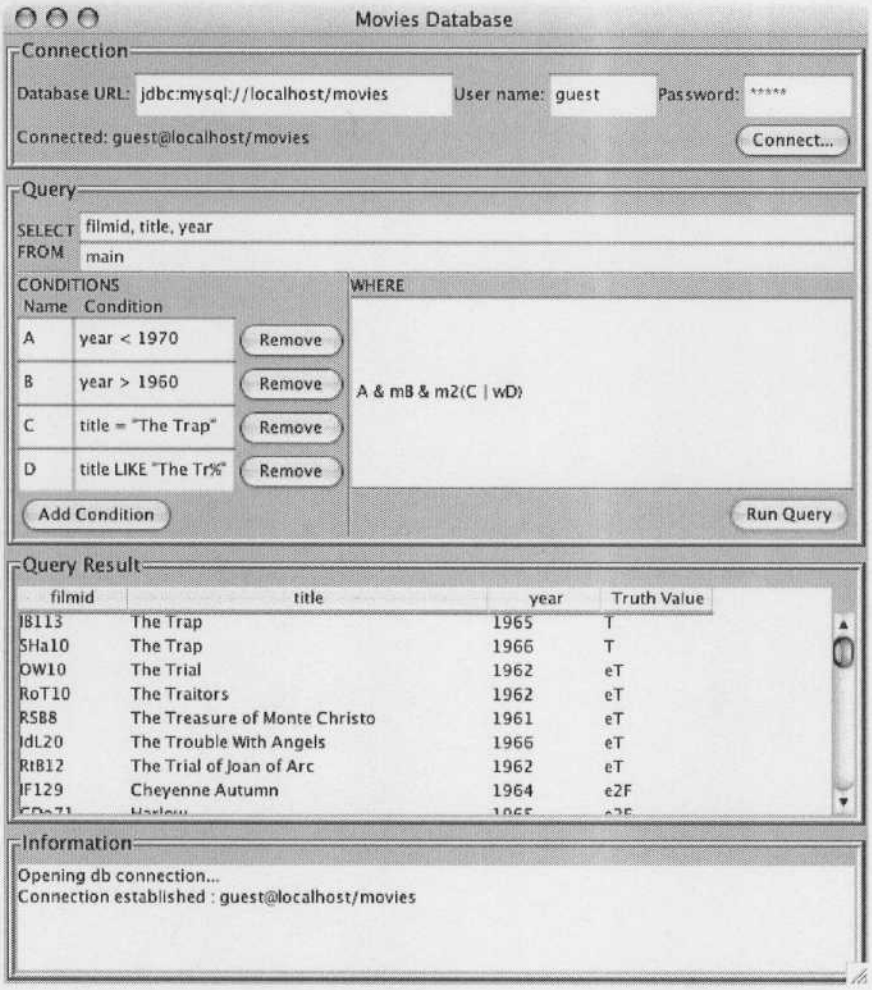

Figure 5.2: View of the Movies Database application after a Query.

5.3.4 Connection and Query Information

The *Information* section provides the user with event alerts and error information regarding the connection and queries.

5.3.5 Example Queries

Example 5.3.1. *Figure 5.3 shows the query results for movies directed by Hitchcock, preferably after 1960 and if possible, containing "Bang" in the title.*

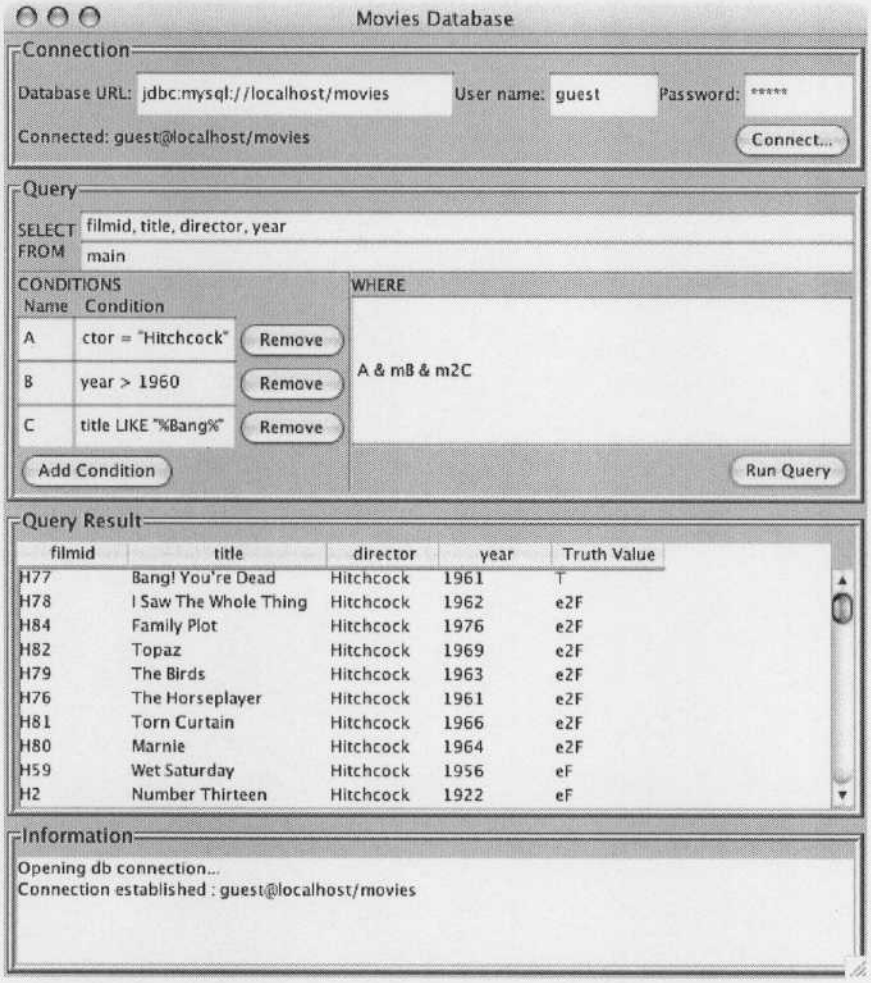

Figure 5.3: View of Hitchcock Query.

Example 5.3.2. *Figure 5.4 shows the query results for movies that received an Academy Award for Best Picture and Best Director, or if that is not possible, just for Best Picture, preferably after 1990.*

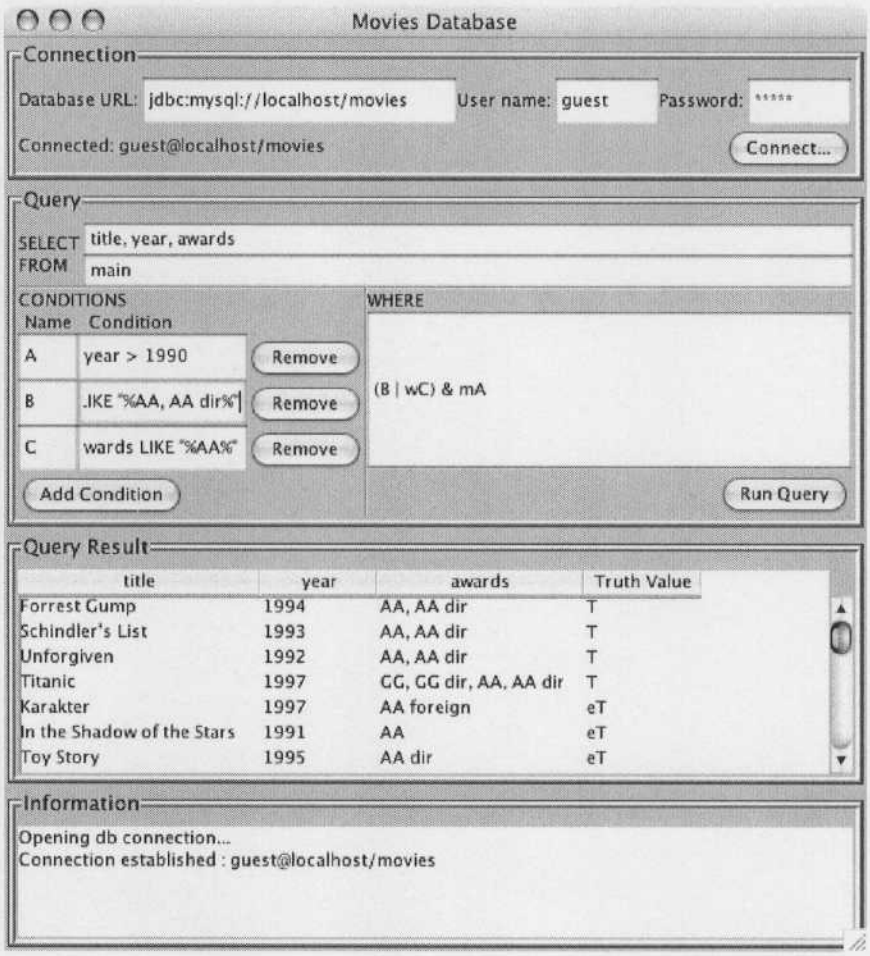

Figure 5.4: View of Awards Query.

Chapter 6

Conclusion

In order to truly take advantage of advances in search technology, we need to provide the user with increased control and expressive power for specifying search criteria. This includes being able to assert priorities among query constraints.

In this thesis, we have presented a framework for expressing a hierarchy of priorities among both preferential and backup constraints. Infinitesimal logic provides for a flexible logic-based foundation for our representation and a simple and elegant way to express a hierarchy among the priorities. The prioritized constraint expressions of our framework can be evaluated using a recursive lazy evaluation algorithm.

The movies database application implements our priority assertion framework. However, the application is still a research prototype that currently serves as a tool to test and demonstrate the expressive power and flexibility of our framework. Ideally, an extension to SQL should be implemented to handle the parsing and evaluation of the operators μ and ω, allowing the user to include priorities in a single PCE that could be run as a SQL query.

Although presented in the context of database queries, it is clear that the framework presented in this thesis can be used in any context requiring priority assertion.

Bibliography

[1] P. Rondogiannis and W. W. Wadge, "Minimum model semantics for logic programs with negation-as-failure," *ACM Transactions on Computational Logic*, 2005.

[2] J. Chomicki, "Preference formulas in relational queries," *ACM Transactions on Database Systems*, vol. 28, 2003.

[3] A. Borning, B. Freeman-Benson, and M. Wilson, "Constraint hierarchies," *Lisp and Symbolic Computation*, vol. 5, pp. 223–270, 1992.

[4] H. Hosobe, "Hierarchical nonlinear constraint satisfaction," in *ACM Symposium on Applied Computing*, 2004.

[5] B. Kosko, *The Fuzzy Future.* Harmony Books, 1999, p. 7.

[6] R. Agrawal and E. L. Wimmers, "A framework for expressing and combining preferences," in *ACM SIGMOD*, 2000.

[7] (2005, Mar.) Lazy evaluation - Wikpedia, the free encyclopedia. [Online]. Available: http://en.wikipedia.org/wiki/Lazy_evaluation

[8] (2002) Lazy evaluation of boolean expressions. SciFace Software GmbH & Co. KG. [Online]. Available: http://research.mupad.de/doc/25/eng/stdlib/andorl.shtml

[9] (2005, Apr.) Alpha-beta pruning - Wikpedia, the free encyclopedia. [Online]. Available: http://en.wikipedia.org/wiki/Alpha-beta_pruning

[10] G. Wiederhold. Movies database documentation. Stanford University. [Online]. Available: http://www-db.stanford.edu/pub/movies/doc.html

[11] D. Axmark, M. M. Widenius, *et al.* Mysql reference manual. [Online]. Available: http://dev.mysql.com/doc/mysql/en/

[12] L. Zadeh, "Fuzzy logic: issues, contentions and perspectives," in *IEEE International Conference on Acoustics, Speech, and Signal Processing*, 1994.

[13] G. Gerla, *Fuzzy Logic, Mathematical Tools for Approximate Reasoning.* Kluwer Academic, 2001, p. XI.

[14] G. Panti, *Handbook of Defeasible Reasoning and Uncertainty Management Systems.* Kluwer, 1998, vol. 1: Quantified Representation of Uncertainty and Imprecision, ch. 2, Multi-valued logics.

Appendix A

Infinitesimal Logic Expression Evaluator: Prolog Source Code

```
/*-------------------------------------------------
Ruchi Agarwal
University of Victoria

March 2005

PCEE.pl
Evaluates a Prioritized Constraint Expression (PCE).

Implemented using SWI-Prolog.

Notation: t(X) represents a true value "multiplied"
by epsilon to a degree of X.  Similarly, f(X)
represents a false value "multiplied"
by epsilon to a degree of X. 0 represents
indifference.
```

```
You can first assign values to variables using assert.

?- assert(val(a, t(2))).

?- assert(val(b, f(5))).

?- assert(val(c, f(1))).

?- assert(val(d, t(0))).

This sets the value of a to t(2).

Then an ILE can be evaluated by using eval/2.  Eg.

?- eval(a & m (b | c) | w d, V).

This gives the result

V = t(1);

Note that eval/2 is implemented using lazy evaluation.
For standard evaluation, use standard_eval/2.
-------------------------------------------------*/

% -------- Operator definitions-----------------------------------------
:- op(710,fx,'!').
:- op(710,fx,'m').
:- op(710,fx,'w').
:- op(720,yfx,'&').
:- op(730,yfx,'|').
```

```
% ---------Standard Evaluation-----------------------------------------------
standard_eval(E, A) :- atom(E), val(E, A).
standard_eval(! E, B) :- atom(E), val(E, A), inf_not(A, B).
standard_eval(E1 & E2, V) :- standard_eval(E1, V1),
standard_eval(E2, V2),
inf_and(V1, V2, V).
standard_eval(E1 | E2, V) :- standard_eval(E1, V1),
standard_eval(E2, V2),
inf_or(V1, V2, V).
standard_eval(! E1, V) :- standard_eval(E1, V1),
inf_not(V1, V).
standard_eval(m E1, V) :- standard_eval(E1, V1),
inf_m(1, V1, V).
standard_eval(w E1, V) :- standard_eval(E1, V1),
inf_w(1, V1, V).

% ---------Lazy Evaluation-------------------------------------------------

% trivial cases
eval(E, A) :- atom(E), val(E, A).
eval(! E, B) :- atom(E), val(E, A), inf_not(A, B).

% initializer
eval(E1 & E2, V) :- eval(E1 & E2, f(0), t(0), V).
eval(E1 | E2, V) :- eval(E1 | E2, f(0), t(0), V).
eval(! (E), V) :- eval(! (E), f(0), t(0), V).

% base case - at a leaf node (variable node)
% we just get the value of the variable
```

```
eval(X, _, _, V) :- atom(X), val(X, V).

% at an AND node : if the left child's value is not
% more true than the parent's current minimum value,
% then we don't need to evaluate the right child since
% it cannot make the parent any less truer
eval(E1 & _, MIN, MAX, MIN) :- eval(E1, MIN, MAX, V1), lesstrue(V1, MIN).
eval(E1 & _, MIN, MAX, MIN) :- eval(E1, MIN, MAX, V1), equallytrue(V1, MIN).
eval(E1 & E2, MIN, MAX, V) :- eval(E1, MIN, MAX, V1), moretrue(V1, MIN),
                              eval(E2, MIN, V1, V2), inf_and(V1, V2, V).

% at an OR node : if the left child's value is not
% less true than the parent's current maximum value,
% then we don't need to evaluate the right child since
% it cannot make the parent any more truer
eval(E1 | _, MIN, MAX, MAX) :- eval(E1, MIN, MAX, V1), moretrue(V1, MAX).
eval(E1 | _, MIN, MAX, MAX) :- eval(E1, MIN, MAX, V1), equallytrue(V1, MAX).
eval(E1 | E2, MIN, MAX, V) :- eval(E1, MIN, MAX, V1), lesstrue(V1, MAX),
                              eval(E2, V1, MAX, V2), inf_or(V1, V2, V).

% at a NOT node : we evaluate the argument to the
% operator NOT, but we must send down different
% MIN and MAX bounds.  The MIN must be set to NOT of
% the parent's MAX value and the MAX must be set to the
% NOT of the parent's MIN value.
eval(! (E), MIN, MAX, V) :- inf_not(MIN, MIN_NOT), inf_not(MAX, MAX_NOT),
                            eval(E, MAX_NOT, MIN_NOT, V1), inf_not(V1, V).

% at a M node : we evaluate the argument to the
% operator M, but we must divide the MIN and MAX
```

```
% values by M with a degree of D.
eval(m (E), MIN, MAX, V) :- div_m(1, MIN, MIN_M), div_m(1, MAX, MAX_M),
                            eval(E, MIN_M, MAX_M, V1), inf_m(1, V1, V).

% at a W node : we evaluate the argument to the
% operator M, but we must divide the MIN and MAX
% values by W with a degree of D.
eval(w (E), MIN, MAX, V) :- div_w(1, MIN, MIN_M), div_w(1, MAX, MAX_M),
                            eval(E, MIN_M, MAX_M, V1), inf_w(1, V1, V).
%----------Helpers-----------------------------------------------------
% the third infinitesimal truth value is the
% result of performing an AND of
% the first two infinitesimal truth values
inf_and(X, Y, X) :- lesstrue(X, Y).
inf_and(X, Y, X) :- equallytrue(X, Y).
inf_and(X, Y, Y) :- moretrue(X, Y).

% the third infinitesimal truth value is the
% result of performing an OR of
% the first two infinitesimal truth values
inf_or(X, Y, X) :- moretrue(X, Y).
inf_or(X, Y, X) :- equallytrue(X, Y).
inf_or(X, Y, Y) :- lesstrue(X, Y).

% the second infinitesimal truth value is the
% result of performing a NOT of
% the first infinitesimal truth value
inf_not(t(D), f(D)).
inf_not(f(D), t(D)).
inf_not(0, 0).
```

```
% the third infinitesimal truth value is the
% result of performing a M of
% the second argument by degree DEG
inf_m(_, t(D), t(D)).
inf_m(DEG, f(D), f(D2)) :- D2 is D + DEG.

% the third infinitesimal truth value is the
% result of performing a M of
% the second argument by degree DEG
inf_w(_, f(D), f(D)).
inf_w(DEG, t(D), t(D2)) :- D2 is D + DEG.

% the third infinitesimal truth value is the
% result of performing a division by m of
% the second argument by a
% degree of DEG
div_m(_, t(D), t(D)).
div_m(DEG, f(D), f(D2)) :- D > DEG, D2 is D - DEG.
div_m(DEG, f(D), f(0)) :- D =< DEG.

% the third infinitesimal truth value is the
% result of performing a division by m of
% the second argument by a
% degree of DEG
div_w(_, f(D), f(D)).
div_w(DEG, t(D), t(D2)) :- D > DEG, D2 is D - DEG.
div_w(DEG, t(D), t(0)) :- D =< DEG.

% truth values - t(D) is eDt, where D is the
```

```
% degree and f(D) is eDf where D is the degree
t(D) :- number(D), D>=0.
f(D) :- number(D), D>=0.

% helpers to perform comparisons between truth values
moretrue(f(D1), f(D2)) :- D1>D2.
moretrue(t(_), f(_)).
moretrue(t(_), 0).
moretrue(0, f(_)).
moretrue(t(D1), t(D2)) :- D1<D2.

lesstrue(f(D1), f(D2)) :- D1<D2.
lesstrue(f(_), t(_)).
lesstrue(0, t(_)).
lesstrue(f(_), 0).
lesstrue(t(D1), t(D2)) :- D1>D2.

equallytrue(t(D1), t(D2)) :- D1==D2.
equallytrue(f(D1), f(D2)) :- D1==D2.
equallytrue(0, 0).
```

Appendix B

Java Source Code

```
/*************************************************************
ParseTreeNode.java
Ruchi Agarwal
March 2005
 * This class contains the methods to handle each
 * type of node in the parse tree for the PCE.
 * This class also implements the lazy evaluation
 * algorithm for evaluating a PCE.
*************************************************************/
import java.io.*;
import java.util.*;

/**
 *
 * POS        ::=   int > 0
 * ELEMENT    ::=   var //variable
 *              |   constant //T or F
 *              |   PRE var
```

```
 *              |    PRE constant
 *              |    PRE PAREN
 *              |    PAREN
 * PAREN        |    "(" DISJ ")"
 * PRE         ::=   "!" //unary operators
 *     |    "m"
 *     |    "w"
 *     |    "e"
 *     |    "m" POS
 *     |    "w" POS
 *     |    "w" POS
 * CONJ        ::=   CONJ "&" ELEMENT //conjunction
 *     |    ELEMENT
 * DISJ    ::=   DISJ "|" CONJ //disjunction
 *              |    CONJ

 *
 * Precedence rules from lowest to highest :
 *  1.  |
 *  2.  &
 *  3.  !, m, w, e
 *
 * Operators | and & are left-associative.
 *
 */
public class ParseTreeNode {
  final static int OP_NOT = 0;
 final static int OP_M = 1;
 final static int OP_W = 2;
 final static int OP_E = 3;
```

```
final static int OP_AND = 4;
final static int OP_OR = 5;
final static int TERM_VAR = 6;
final static int TERM_CONST = 7;

ParseTreeNode left;
ParseTreeNode right;
int type;
String contents;
int degree = 0;

//constructor for binary nodes
public ParseTreeNode(int type, ParseTreeNode left, ParseTreeNode right){
 this.type = type;
 this.left = left;
 this.right = right;
 contents = "op";
}

//constructor for unary op nodes - except "not" nodes
public ParseTreeNode(int type, int degree, ParseTreeNode left){
 this.type = type;
 this.degree = degree;
 this.left = left;
 this.right = null;
 contents = "op";
}

//constructor for "not" op nodes
public ParseTreeNode(int type, ParseTreeNode left){
```

```
 this.type = type;
 this.left = left;
 this.right = null;
 contents = "op";
}

//constructor for terminal nodes
public ParseTreeNode(String contents){
 this.left = null;
 this.right = null;
 this.contents = contents;
 if(contents.equals("T") || contents.equals("F")){
  this.type = TERM_CONST;
 }else{
  this.type = TERM_VAR;
 }
}


   static ParseTreeNode disj(StreamTokenizer st) throws SyntaxError {
       ParseTreeNode result = null;
 boolean done = false;

 result = conj(st);
 while (!done){
  try {
   if(st.nextToken() == '|'){ //then there are more conjunctions to be read
    result = new ParseTreeNode(OP_OR, result, conj(st));
   }else{
    done = true;
```

```
   st.pushBack();
  }
 } catch (IOException ioe) {
  throw new SyntaxError("Caught an I/O exception - disj.");
 }
}
return result;
  }

  static ParseTreeNode conj(StreamTokenizer st) throws SyntaxError {
      ParseTreeNode result = null;
boolean done = false;

result = element(st);
while (!done){
 try {
  if(st.nextToken() == '&'){ //then there are more elements to be read
   result = new ParseTreeNode(OP_AND, result, element(st));
  }else{
   done = true;
   st.pushBack();
  }
 } catch (IOException ioe) {
  throw new SyntaxError("Caught an I/O exception - conj.");
 }
}
return result;
  }

  static ParseTreeNode element(StreamTokenizer st) throws SyntaxError {
```

```
try {
 int tok = st.nextToken();
 if(st.ttype == st.TT_WORD){
  if(!st.sval.startsWith("m") &&
      !st.sval.startsWith("w") && !st.sval.startsWith("e")){
   //then it must be a terminal
   return new ParseTreeNode(st.sval);
  }else{
   int degree = 1;
   String pre = st.sval;
   char pretype = pre.charAt(0);
   if(st.nextToken() == '('){
    //must be followed by a parenthesized expression
    st.pushBack();
    if(pre.length() > 1) degree = Integer.parseInt(pre.substring(1));
    switch(pretype){
     case 'm':
      return new ParseTreeNode(OP_M, degree, paren(st));
     case 'w':
      return new ParseTreeNode(OP_W, degree, paren(st));
     case 'e':
      return new ParseTreeNode(OP_E, degree, paren(st));
     default :
      throw new SyntaxError("Expected a m, w, or e");
    }
   }else{
    st.pushBack();
    if(pre.length() > 1){
     int ind = 1;
     //get the index of the last digit of the degree
```

```
while (ind < pre.length()){
 if(Character.isDigit(pre.charAt(ind))){
  ind++;
  continue;
 }else{
  break;
 }
}
//here, ind is the index of the first non-digit character
if(ind > 1){
 degree = Integer.parseInt(pre.substring(1, ind));
}
String cont = pre.substring(ind);
if(cont == ""){
 throw new SyntaxError("Expected a terminal");
}else{
 switch(pretype){
  case 'm':
   return new ParseTreeNode(OP_M, degree, new ParseTreeNode(cont));
  case 'w':
   return new ParseTreeNode(OP_W, degree, new ParseTreeNode(cont));
  case 'e':
   return new ParseTreeNode(OP_E, degree, new ParseTreeNode(cont));
  default :
   throw new SyntaxError("Expected a m, w, or e");
 }
}
}else{
 throw new SyntaxError("Expected a terminal");
}
```

```
   }
  }
 }else{
  if(st.ttype == '!'){
   if(st.nextToken() == '('){
    st.pushBack();
    return new ParseTreeNode(OP_NOT, paren(st));
   }else{
    st.nextToken();
    return new ParseTreeNode(OP_NOT, new ParseTreeNode(st.sval));
   }
  }else{
   st.pushBack();
   return paren(st);
  }
 }
} catch (IOException ioe) {
 throw new SyntaxError("Caught an I/O exception - element.");
}
  }

  static ParseTreeNode paren(StreamTokenizer st) throws SyntaxError {
ParseTreeNode result;
try {
 if(st.nextToken() == '(') {
  result = disj(st);
  if(st.nextToken() != ')') {
   System.out.println("Mismatched Parentheses... found " + st.toString());
   throw new IOException();
  }
```

```
  }else {
   st.pushBack();
   System.out.println("uh oh, no paren found");
   throw new IOException();
  }
 } catch (IOException ioe) {
  throw new SyntaxError("Caught an I/O exception - paren.");
 }
 return result;
   }

static ParseTreeNode parse(String ile) throws Exception{
 return ParseTreeNode.disj(new StreamTokenizer(new StringReader(ile)));
}

public static InfinitesimalTruthValue lazy_evaluate
 (ParseTreeNode node, Hashtable varvals) throws Exception{
 return lazy_evaluate(node, new InfinitesimalTruthValue('F', 0),
   new InfinitesimalTruthValue('T', 0), varvals);
}

public static InfinitesimalTruthValue lazy_evaluate(ParseTreeNode node,
  InfinitesimalTruthValue min, InfinitesimalTruthValue max, Hashtable varvals)
    throws Exception{
 InfinitesimalTruthValue childval, leftval, rightval;
 switch(node.type){
  case OP_M :
   childval = lazy_evaluate(node.left, min.divM(node.degree),
          max.divM(node.degree), varvals);
   return childval.M(node.degree);
```

```
case OP_W :
 childval = lazy_evaluate(node.left, min.divW(node.degree),
        max.divW(node.degree), varvals);
 return childval.W(node.degree);
case OP_E :
 childval = lazy_evaluate(node.left, min.divE(node.degree),
        max.divE(node.degree), varvals);
 return childval.E(node.degree);
case OP_NOT :
 childval = lazy_evaluate(node.left, max.NOT(), min.NOT(), varvals);
 return childval.NOT();
case OP_AND :
 leftval = lazy_evaluate(node.left, min, max, varvals);
 if(leftval.compareTo(min) <= 0) {
  return min;
 }else{
  rightval = lazy_evaluate(node.right, min, leftval, varvals);
  return leftval.AND(rightval);
 }
case OP_OR :
 leftval = lazy_evaluate(node.left, min, max, varvals);
 if(leftval.compareTo(max) >= 0) {
  return max;
 }else{
  rightval = lazy_evaluate(node.right, leftval, max, varvals);
  return leftval.OR(rightval);
 }
case TERM_CONST :
 if(node.contents.equals("T")) return new InfinitesimalTruthValue('T', 0);
 if(node.contents.equals("F")) return new InfinitesimalTruthValue('F', 0);
```

```
    else throw new Exception(node.contents);
   default : //must be a var
    if(varvals == null) throw new Exception("Unrecognized variable.");
    else {
     ConditionValue cv = (ConditionValue)varvals.get(node.contents);
     if(cv == null) throw new Exception("Unrecognized variable.");
     else {
      if(cv.getValue()) return new InfinitesimalTruthValue('T', 0);
      else return new InfinitesimalTruthValue('F', 0);
     }
    }
  }
 }
}

/*************************************************************
InfinitesimalTruthValue.java
Ruchi Agarwal
March 2005
 * This class implements the rules of infinitesimal logic.
*************************************************************/
public class InfinitesimalTruthValue implements Comparable{
 private char boolVal;
 private int degree;

 public InfinitesimalTruthValue(char boolVal, int degree)
  throws UnknownTruthValueException, NegativeTruthDegreeException{
  if(boolVal != 'T' && boolVal != 'F' && boolVal != '0')
   throw new UnknownTruthValueException();
  if(degree < 0)
```

```
  throw new NegativeTruthDegreeException();
 else {
  this.boolVal = boolVal;
  if (this.boolVal == '0'){
   this.degree = 0;
  }else {
   this.degree = degree;
  }
 }
}

public char getBoolVal(){
 return boolVal;
}

public int getDegree(){
 return degree;
}

public InfinitesimalTruthValue AND(InfinitesimalTruthValue itv){
 if(this.compareTo(itv) > 0) return itv;
 else return this;
}

public InfinitesimalTruthValue OR(InfinitesimalTruthValue itv){
 if(this.compareTo(itv) < 0) return itv;
 else return this;
}

public InfinitesimalTruthValue NOT(){
```

```
 try {
  if(this.boolVal == 'T')
   return new InfinitesimalTruthValue('F', this.degree);
  else if(this.boolVal == 'F')
   return new InfinitesimalTruthValue('T', this.degree);
 }catch(Exception e){
  System.out.println("Error performing not.");
 }
 return this;
}

public InfinitesimalTruthValue W(int deg){
 try{
  if(this.boolVal == 'T')
   return new InfinitesimalTruthValue('T', this.degree + deg);
 }catch(Exception e){
  System.out.println("Error performing W.");
 }
 return this;
}

public InfinitesimalTruthValue M(int deg){
 try{
  if(this.boolVal == 'F')
   return new InfinitesimalTruthValue('F', this.degree + deg);
 }catch(Exception e){
  System.out.println("Error performing M.");
 }
 return this;
}
```

```
public InfinitesimalTruthValue E(int deg){
 try {
  return new InfinitesimalTruthValue(this.boolVal, this.degree + deg);
 }catch(Exception e){
  System.out.println("Error performing E.");
  return this;
 }
}

public InfinitesimalTruthValue divW(int deg){
 if(this.boolVal == 'T') {
  int newdeg;
  if(this.degree > deg){
   newdeg = this.degree - deg;
  }else{
   newdeg = 0;
  }
  try{
   return new InfinitesimalTruthValue('T', newdeg);
  }catch(Exception e){
   System.out.println("Error performing divW.");
  }
 }
 return this;
}

public InfinitesimalTruthValue divM(int deg){
 if(this.boolVal == 'F') {
  int newdeg;
```

```
  if(this.degree > deg){
   newdeg = this.degree - deg;
  }else{
   newdeg = 0;
  }
  try{
   return new InfinitesimalTruthValue('F', newdeg);
  }catch(Exception e){
   System.out.println("Error performing divM.");
  }
 }
 return this;
}

public InfinitesimalTruthValue divE(int deg){
 if(this.boolVal == '0') return this;
 int newdeg;
 if(this.degree > deg){
  newdeg = this.degree - deg;
 }else{
  newdeg = 0;
 }
 try{
  return new InfinitesimalTruthValue(this.boolVal, newdeg);
 }catch(Exception e){
  System.out.println("Error performing divE.");
  return this;
 }
}
```

```
public String toString(){
 String b = "0";
 if(boolVal == 'T') b = "T";
 else if(boolVal == 'F') b = "F";
 if(degree == 0) return b;
 else if (degree == 1) return "e" + b;
 else return "e" + degree + b;
}

public int compareTo(Object obj){
 InfinitesimalTruthValue itv = (InfinitesimalTruthValue)obj;
 char thisBool = this.getBoolVal();
 char itvBool = itv.getBoolVal();
 int thisDegree = this.getDegree();
 int itvDegree = itv.getDegree();

 if(thisBool == 'T'){
  if(itvBool == 'T'){
   if(thisDegree < itvDegree) return 1;
   else if(thisDegree > itvDegree) return -1;
   else return 0;
  }else{
   //this truth value must be more true than '0' or any degree of 'F'.
   return 1;
  }
 }else if(thisBool == 'F'){
  if(itvBool == 'F'){
   if(thisDegree > itvDegree) return 1;
   else if(thisDegree < itvDegree) return -1;
   else return 0;
```

```
   }else{
    //this truth value must be less true than '0' or any degree of 'T'.
    return -1;
   }
  }else{
   if(itvBool == 'F') return 1;
   else if(itvBool == 'T') return -1;
   else return 0;
  }
 }
}

class UnknownTruthValueException extends Exception{
 public String toString(){
  return "Incorrect truth type.
   Truth value type must be either T, F, or 0.";
 }
}

class NegativeTruthDegreeException extends Exception{
 public String toString(){
  return "The truth value degree cannot be negative.";
 }
}
```