A FRAMEWORK FOR REASONING PRECISELY WITH VAGUE CONCEPTS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

By
Nita Goyal
May 1994

Abstract

Many knowledge-based systems need to represent vague concepts such as “old” and “tall”. The practical approach of representing vague concepts as precise intervals over numbers (e.g., “old” as the interval [70,110]) is well-accepted in Artificial Intelligence. However, there have been no systematic procedures, only *ad hoc* methods, to delimit the boundaries of intervals representing the vague predicates. A key observation is that the vague concepts and their interval boundaries are constrained by the underlying domain knowledge. Therefore, any systematic approach to assigning interval boundaries must take the domain knowledge into account. In this dissertation, we introduce a framework to represent the domain knowledge and use it to reason about the interval boundaries via a query language. This framework is comprised of a *constraint language* to represent logical constraints on the vague concepts, as well as numerical constraints on the interval boundaries; a *query language* to request information about the interval boundaries; and an *algorithm* to answer the queries. The algorithm preprocesses the constraints by extracting the numerical information from the logical constraints and then combines them with the given numerical constraints. We have implemented the framework and applied it to two domains to illustrate its usefulness.

Acknowledgements

I would like to thank Prof. Yoav Shoham for his continued advice and support throughout the course of my dissertation. I would also like to thank Prof. Nils Nilsson and Dr. Pat Hayes for their valuable input both on the technical content as well as on the writing of the dissertation.

I will forever be grateful to Surajit Chaudhuri for his constant encouragement and support through the tough periods of the Ph.D.. In addition to being a caring friend, he helped me learn how to do research, how to write, and devoted innumerable hours in technical discussions. Without his support this dissertation would not have materialized.

My officemates Becky Thomas and Moshe Tennenholtz provided hours of interesting conversations on every topic under the sun in addition to the technical discussions. I am grateful for their friendship.

I am also grateful to Pandu Nayak and Alon Levy for many discussions and for their valuable feedback. I would like to thank everyone in the nobotics group for providing an enjoyable and challenging work environment and Dr. Garry Gold for his help in formulating the rules for the medical domain.

I would like to thank all my friends who made my life at Stanford fun and in whose absence the dissertation would have been over sooner. I am particularly indebted to Ramana Venkata, Surajit Chaudhuri, Anoop Goyal and Ashish Gupta for being my family here.

Last but not the least, the encouragement and support of my parents and their belief in me has made everything possible. I owe whatever I am to them.

Contents

List of Figures

Chapter 1

Introduction

To behave intelligently in the real world, an entity must possess knowledge about that world and must be able to use this knowledge effectively. The basic knowledge about the world that is possessed by every school child and the methods for making obvious inferences from this knowledge is called commonsense. Capturing this commonsense to make a computer program behave intelligently has been recognized as a fundamental problem of the field of Artificial Intelligence [McCarthy, 1959]. Commonsense knowledge and commonsense reasoning are involved in most types of intelligent activities, such as using natural language, planning, learning, high-level vision, and expert-level reasoning [Davis, 1990, page 1].

There are many aspects to human commonsense knowledge that make its representation and reasoning with it difficult in a rigid computer program. Human reasoning is flexible enough to deal with partial, uncertain or imprecise information; in fact, most of our everyday knowledge falls in this category. As quoted in [Davis, 1990, page 18], "You know that the water in the tea-kettle will come to a boil soon after you turn the knob on the stove, but you do not know how soon it will boil, how hot the flame is, or how much liquid is in the kettle. This is incompleteness. Moreover, you cannot be entirely sure that the water will boil at all; the stove may be broken, or someone may come and turn it off, or the kettle may be empty. This is uncertainty." But at least we can identify precisely whether the water is boiling or not, or we could if we had a thermometer. However, we often have to deal with concepts that are not precise.

The fact that concepts are often *vague* is an important and ubiquitous aspect of commonsense knowledge that goes beyond uncertainty and incompleteness. To quote Davis, "Many categories of common sense have no well-marked boundary lines; there are clear examples and clear nonexamples, but in between lies an uncertain region that we cannot categorize, even in principle." For example, there is no minimum precise body temperature that a doctor considers high and there is no maximum number of hairs that a person might have and still be considered bald. The concepts high_temperature and bald are vague. To represent and reason with commonsense knowledge, it is of paramount importance to be able to deal with the vagueness of concepts.

Since such vague concepts have no precise definition, representing knowledge about them and reasoning with this knowledge poses a problem. "From a theoretical point of view, this vagueness is extremely difficult to deal with, and no really satisfactory solutions have been proposed. The difficulties are made vivid by an ancient paradox called the *Sorites* (meaning heap). If you have a heap of sand, and you take away one single grain of sand, you will obviously still have a heap of sand. But therefore, by induction, you can take away all the sand, and still have a heap, which is absurd. Like the other terms above, heap is vague; there is no specific minimum number of grains that a heap can contain." [Davis, 1990, pages 19-20]. What then do we do in the face of this difficulty?

In Section 1.1, we discuss the nonstandard logics that attempt to capture vagueness and in Section 1.2 we discuss the practical approach taken by most AI systems to deal with vague concepts. In this dissertation, we will be concerned with the practical aspect of representation and reasoning with vague concepts[1].

[1]The issues of uncertainty and incompleteness of information rather than its vagueness is addressed by the work on probabilistic and possibilistic logics. Henceforth, we will not be concerned with these issues and a reader interested in them should refer to [Shafer and Pearl, 1990].

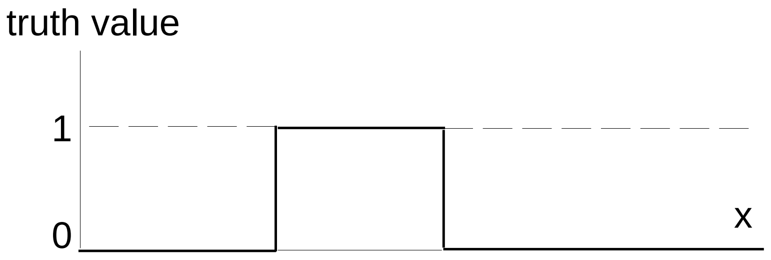


Figure 1.1: A Binary-valued Truth Function

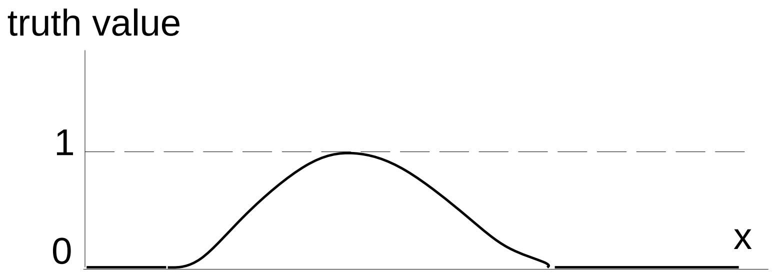


Figure 1.2: A Fuzzy Truth Function

1.1 Nonstandard Logics addressing Vagueness

Previously proposed approaches for representing and reasoning with vague concepts capture the imprecision of the symbols by using specialized representational apparatus. The most prominent approach is fuzzy logic [Zadeh, 1983; Zadeh, 1988]; another approach is that of *vague predicates* [Parikh, 1983].

In fuzzy logic, the truth value of a proposition can range over the unit interval [0, 1] as opposed to being either true or false (*i.e.,* in the set $\{0, 1\}$) as in classical logic. Figure 1.1 illustrates a binary-valued truth function and Figure 1.2 a fuzzy function. Another difference is that classical logic allows only two quantifiers: "all" or "some", whereas the propositions in fuzzy logic can also be quantified using fuzzy quantifiers such as *most*, *mostly*, *usually*, *frequently*, *etc.* For example, *(most) birds can fly*, or *(usually) a cup of coffee costs about fifty cents*. Inferencing is carried out through methods such as the intersection/product syllogism [Zadeh, 1988].

The notion of *vague predicates* is defined in [Parikh, 1983] by first defining a vague real number and then an arithmetic over the vague real numbers. The definition of a vague predicate and the accompanying semantics and inference mechanism is meant to get over the observational paradox such as that demonstrated by *Sorites* where the transitivity of the indistinguishability relation leads to absurd conclusions.

These nonstandard logics gets around many of the conceptual problems associated with vagueness but they modify the standard Tarskian semantics of binary-valued logics and the resulting framework is considerably more complex. For instance, the automated reasoning processes can no longer be used in a straightforward manner. Even though conceptually the logic might capture the intuition of vagueness to some extent, there still remains the problem of defining the right fuzzy value function for predicates. In spite of the existence of these logics that get around the conceptual problem of representation, many practical systems prefer to use the simpler representation of binary-valued concepts. These systems prefer to deal with the ubiquitous vague concepts using this simpler representation even though it may not be as intuitively appealing as the nonstandard logics. In the next section, we will discuss the approach taken by these systems to address the issue of vague concepts within the simpler binary-valued framework.

1.2 Practical Approach addressing Vagueness

It is commonly accepted in Artificial Intelligence that, though inadequate theoretically, in practice it is often adequate to assume that a vague concept *is* precise and that there is indeed a well-defined boundary. This practical approach is illustrated by an example from [Davis, 1990, pages 19-20]: “Suppose that “bald” did refer to some specific number of hairs on the head, only we do not know which number. We know that a man with twenty thousand hairs on his head is not bald, and that a man with three hairs on his head is bald, but somewhere in between we are doubtful.” This precise representation of the vague concept bald is still useful for reasoning.

In contrast to the nonstandard logical approaches that lead to complex representation and reasoning, many system builders who encounter the vagueness problem adopt the approach of representing a vague concept as a precise one. For instance, GUARDIAN [Hayes-Roth *et al.*, 1989] and PROTEGE-II [Shahar *et al.*, 1992] are AI systems for medical diagnosis that represent vague concepts such as high temperature or low blood-pressure as precise. Some other AI systems that take this approach are MCM [DÁmbrosio *et al.*, 1987] which is a system for chemical manufacturing process control, expert systems for preventive control in power plant (GTES, ESCARTA, SMOP) [Jiang *et al.*, 1991], and expert system to predict thunderstorms and severe weather (TIPS) [Passner and Lee, 1991].

In this dissertation, we propose to adopt this practical approach of interpreting a vague concept as a precise one. Since a large number of vague concepts are abstractions over numbers, we will, in particular, consider the vague concepts that are defined over real-valued measure spaces. Therefore, the vague concepts will be interpreted as precise intervals over the real number range. For example in a medical system, we can assume that the concept high temperature is the real interval (99,106], normal temperature is the interval (97,99] and low temperature is the interval (94,97]. The main attractions of this predicate-as-interval approach are its conceptual simplicity and the ability to use standard reasoning mechanisms under this approach. In other words, high, normal and low temperatures can be interpreted as assertions in the classical sense using Tarskian semantics and standard reasoning techniques can be applied to them.

Some may object to this interpretation of vague predicates as intervals on the grounds that it is unintuitive since it causes an abrupt change in the truth value at the interval boundary. We argue that, despite the apparent conceptual difficulty, the predicate-as-interval approach can be justified in a wide variety of circumstances.

1.2.1 Justifying Predicate-as-Interval

Sometimes information to determine precise intervals of a vague concept is simply *given*. For example, the exact boiling point of water is known and can be used to divide the temperature range into three classes: below boiling point, at boiling point and above boiling point. Use of such "landmark" values for reasoning about physical phenomena is used widely in qualitative physics in the form of quantity spaces [Weld and de Kleer, 1990].

Intervals are commonly used as a representational primitive in temporal reasoning.

The duration of an event or the time interval in which an instantaneous event may have occurred are often represented as temporal intervals even though the event duration or the actual time of an event might be vague. The relationship between the intervals as well as the time-points is then used to constrain the intervals. For example, "John was away yesterday" refers to an event that occurred over a period of time. "We found the letter yesterday" refers to an instantaneous event of finding the letter that is known to have happened sometime during a time interval. [Allen, 1985] is the seminal AI paper on the interval calculus for temporal relations. Further studies of the logical and computational properties of the interval calculus include [Vilain *et al.*, 1990; Allen and Hayes, 1985; Ladkin, 1987].

There are many situations where it is *necessary* to define vague concepts as precise intervals. Grading of students in a course requires the grader to assign letter grades based on numerical scores. Driving and voting ages, income ranges for taxation, legal blood alcohol level for drivers are but a few everyday examples where crisp cutoffs are used rather than the vague notions of mature person, a rich or poor person, or a sober person. This is necessary for pragmatic reasons irrespective of whether the crisp boundaries are intuitive or not.

Even if it is not necessary to define predicates as intervals, in many situations it is *sufficient* to do so and does not present any pragmatic difficulty. Take for instance the classification of light wavelengths as colors. Like others before him, [Parikh, 1983] points out that as wavelength is a continuous quantity, colors change gradually with red merging into orange and then into yellow as an example. Parikh goes on to propose a notion of *vague predicates*, but we point out that in most cases of interest this theoretical difficulty does not translate to a pragmatic one. When in the real world it is *important* to distinguish colors, the colors are selected so that the wavelengths are sufficiently far apart. For instance, the colors at the traffic lights can be distinguished easily. The colors green, yellow and red correspond to wavelength intervals [500 mμ, 570 mμ], [570 mμ, 590 mμ] and [610 mμ, 700 mμ], respectively [Encyclopedia Britanica, 1986]; in fact, the wavelengths at stoplights fall into narrower intervals. These colors can be distinguished easily, and it is unimportant where exactly the boundaries are placed between them as long as they satisfy the conditions which necessitated distinguishing the colors. People would be in trouble if the colors used for traffic lights were instead venetian red (590.2 mμ), cadmium red (604.8 mμ) and English vermilion (608.1 mμ).

In this dissertation, we do not suggest that the vagueness of predicates is always a non-issue. For example, the crisp color-wavelength connection discussed above may be inappropriate to reason about the artistic value of color composition. We have only argued that there are many real world situations where it is necessary to model predicates as intervals and others where it is sufficient. The ubiquity of this practical approach and the fact that implementors of many AI systems have tended to adopt it because of its pragmatic advantage establishes that this approach warrants a careful, systematic study.

1.3 Predicate-as-Interval: A Systematic Study

We have established that the approach of interpreting a vague predicate as an interval over numbers is practical and useful. The immediate question that arises in order to establish this interpretation is – how to determine the interval-boundaries (henceforth, also referred to as *thresholds*) that correspond to the vague predicate? The rest of the dissertation is concerned with answering this question by providing a systematic method for determining the thresholds.

Despite the pervasiveness of the vagueness problem, and the pervasiveness of the practical approach of representing vague concepts as intervals, there has been no effort in AI to provide a systematic way to compute thresholds in this practical approach. Most systems that have adopted this approach have used *ad hoc* methods to delimit the interval boundaries. The thresholds are established by experts based on their experience. As these systems get large these thresholds should remain compatible with each other and with the world that is being modeled. Enforcing this compatibility can get particularly difficult when there are multiple experts. This involves a long-drawn process of iterative changes to the thresholds until a satisfactory assignment is achieved.

We introduce, instead, a systematic framework for representing and reasoning with vague concepts as intervals that has the advantages of (1) improving our understanding of the issues involved in the practical approach, and (2) replacing the *ad hoc* approach used by system designers to delimit the interval-boundaries. This framework can then be used by a system designer to define the vague concepts precisely and to avoid the pitfalls associated with the *ad hoc* method.

The framework is based on the key observation that vague concepts and their interval-boundaries are constrained by the underlying *domain knowledge* that must be used to reason about the thresholds. The definition of a concept depends on the context in which it will be used and on its relationship to other concepts. For instance, the fact that the heart rate of a patient is related to the blood pressure must be utilized while defining low/high heart rate and low/high blood-pressure. We extend Davis' baldness example to understand the role that the domain knowledge plays.

Example 1.1: "Anyone with 3 or fewer hairs is bald and anyone with 20000 or more hairs is not bald"
"All old people are bald" (note that "old" itself is a vague concept that we will assume has a well-defined boundary)
"Anyone who is 60 years or younger is not old whereas anyone over 100 is old"
"All presidents of companies are old"
"Tom's age is 80 years, he has 500 hairs and is the president of a company"
"Jim's age is 85 years and he has 800 hairs"
"Sam's age is 45 and he has 650 hairs"

Is Tom bald? Logical reasoning tells us that since Tom is president of a company, he is old and therefore bald. Note that here we used only the logical relations between the concepts *president*, old and bald, where old and bald are vague concepts but *president* is not.

Is Jim bald? We can reason that since Tom is old, the oldness threshold[2] can be at most 80. Since Jim's age is 85 which is over the oldness threshold, he must be old and therefore bald. Note that here we needed numerical reasoning with Tom and

[2]By oldness threshold we mean that age such that everyone of higher age is old whereas everyone of lower age is not old. The baldness threshold is defined analogously.

Jim's ages and oldness threshold, as well as logical reasoning that since Jim is old he must be bald.

We can ask if the baldness threshold is necessarily more than 800? Since Jim is bald and has 800 hairs, the baldness threshold must be at least 800. Therefore, the answer to the query is *yes* and hence anyone with less than 800 hairs is bald. Here we needed numerical reasoning about Jim's hairs and the baldness threshold.

Is Sam bald? Since anyone with less than 800 hairs is bald, and Sam has only 650 hairs, he must be bald. Here we needed numerical reasoning with number of hairs on Sam's head and the baldness threshold. ▮

As illustrated by this example, we need to *represent* both logical relations between symbolic concepts and numerical relations on thresholds. Also, logical as well as numerical *reasoning* is required to answer the interesting queries. Hence, the proposed framework facilitates this representation and supports queries about the thresholds.

1.3.1 Framework

The framework is comprised of three main parts – a constraint language to express domain knowledge, a query language to query the domain knowledge and an algorithm to answer the queries.

- The first part of the framework is a *constraint language* that captures the domain knowledge. The language enables the expression of logical constraints on the vague concepts as well as numerical constraints on the thresholds of these concepts. Explicit representation of the thresholds is important to represent the numerical constraints and as we shall see, to ask queries.

- The second part of the framework is a *query language* that extracts relevant information about the thresholds implied by the domain knowledge. In particular, the queries enable us to delimit the thresholds based on the information provided in the domain knowledge[3]. This is exactly what a system designer

[3]Note that it is not necessary to assign specific values to the thresholds to answer any queries, although this assignment is made much easier in our framework.

needs to define intervals for a vague concept that are consistent with the domain knowledge. For example, the answer to the query "what is the minimum permissible value for the baldness threshold?" provides the designer with useful information to define the interval for bald.

- The third part of the framework is an *algorithm* to answer the queries in the query language using the domain knowledge expressed in the constraint language.

1.4 Outline of the Dissertation

In this chapter, we discussed the importance of vague concepts in representing commonsense knowledge. The nonstandard logics such as fuzzy logics try to conceptually capture the intuition behind vagueness but at the cost of modifying the standard Tarskian semantics and making the inference mechanisms complex. We claim that the practical approach of interpreting a vague predicate as a precise interval is simple and useful, as evidenced by its widespread use by system builders, and deserves a thorough study. We introduce a framework to systematically establish the definition of a vague predicate as an interval. This framework utilizes the underlying domain knowledge to determine the thresholds of the intervals. The three main components of this framework are a *constraint language* to represent the domain knowledge, a *query language* to query the constraints about thresholds, and an *algorithm* to compute the answers to the queries.

In Chapter 2 we discuss the constraint language in which the domain knowledge is expressed as also the query language in which the queries can be asked on the constraints.

In Chapter 3 we discuss the algorithm for answering the queries.

In Chapter 4 we discuss this algorithm further together with some useful heuristics and some sample applications of the algorithm.

In Chapter 5 we extend the constraint language to be more general, along with the algorithms for answering queries on the extended language.

Chapter 6 concludes the dissertation and mentions some open questions.

Chapter 2

Constraint and Query Languages

The first component of the framework to reason precisely with vague concepts is a constraint language in which to represent the domain knowledge. The second component is a query language in which queries can be asked about vague concepts and their thresholds. We discuss these languages in this chapter.

2.1 Constraint Language

To express the domain knowledge, the constraint language must have an explicit representation of thresholds. Also, as illustrated by the example of bald people in Chapter 1, the language must be able to express numerical constraints as well as logical constraints. We present such a language here, chosen for its familiarity as well as to strike a tradeoff between expressivity and efficiency of answering queries. We denote the constraint language described here by $\mathcal{CL}$.

The predicates that denote the vague concepts in the logical language are distinguished from the other predicates. We refer to these predicates, which must all be unary, as interval-predicates and to all other predicates as *noninterval-predicates*. Henceforth, interval-predicates appear in the sans serif font and the noninterval-predicates in *italics*. The set of interval-predicates is denoted by $\mathcal{IP}$, and the set of noninterval-predicates by $\mathcal{NIP}$. With every predicate $\mathsf{P} \in \mathcal{IP}$ we associate two *threshold terms* P^- and P^+, called the *lower* and *upper thresholds* of P, respectively.

The set of all threshold terms is denoted by $\mathcal{T}$, *i.e.,* $\mathcal{T} = \{\mathsf{P}^-,\ \mathsf{P}^+ \mid \mathsf{P} \in \mathcal{IP}\}$.

The interval-predicates are interpreted in a special way to reflect our intuition about the vague predicates: P is interpreted as the interval $[\mathsf{P}^-,\ \mathsf{P}^+]$ over $\Re$, the set of real numbers[1]. We refer to this interpretation as the *predicate-as-interval assumption.* This assumption can also be stated as

$$\mathsf{P}(x) \Leftrightarrow \mathsf{P}^- \leq x \leq \mathsf{P}^+$$

1. **Numerical Constraints:** The language of numerical constraints is that of linear arithmetic inequalities where the threshold terms in $\mathcal{T}$ are the variables of the inequalities. A numerical constraint must be reducible to the form

   $$(a_1x_1 + \ldots + a_nx_n)\ relop\ b$$

   where $a_1, \ldots, a_n, b \in \Re$, and $x_1, \ldots, x_n \in \mathcal{T}$,
   and $relop \in \{\leq, \geq, <, >, =\}$.

   We denote the numerical constraint language by $\mathcal{NL}$ and the set of numerical constraints by NC. Due to the predicate-as-interval assumption, NC always includes the following set of inequalities

   $$\{\mathsf{P}^- \leq \mathsf{P}^+ \mid \mathsf{P} \in \mathcal{IP}\}$$

2. **Logical Constraints:** In this chapter, we consider a simple language for logical constraints that we denote by $\mathcal{LL}$. This language simplifies the discussion and understanding of the ensuing algorithm in Chapters 3 and 4. An extended form of this language together with the algorithms are discussed in Chapter 5. A user of this framework can choose a language of desired expressivity depending upon the application. In general, the more restricted the language, the more efficient will be the query-answering algorithm.

[1]To be exact, the interval associated with the predicate could be open or closed at either end. Even though it does not affect the discussion here, these special cases are discussed in Appendix A for the sake of completeness.

The logical constraints in the language $\mathcal{LL}$ are definite Horn clauses [Lloyd, 1987], but with certain restrictions. A Horn clause is of the form

$$\forall\overline{x}[Q(\overline{y}) \leftarrow P_1(\overline{x_1}) \wedge \ldots \wedge P_n(\overline{x_n})]$$

where the single positive literal $Q(\overline{y})$ is called the *head* of the clause, $P_1(\overline{x_1}) \wedge \ldots \wedge P_n(\overline{x_n})$ is a conjunction of positive literals and is called the *body* of the clause, and $\leftarrow$ is the logical implication. $\overline{x}$ is the tuple of all variables occurring in the clause and they are all universally quantified. A definite Horn clause is a Horn clause that has a non-empty *head* (though the *body* can be empty)[2].

The first restriction in the language $\mathcal{LL}$ is that the rules do not have any function symbols[3]. The second restriction is that we do not allow recursion among the clauses. The predicates of these logical constraints come from the set of interval-predicates $\mathcal{IP}$ as well as from noninterval-predicates $\mathcal{NIP}$. The set of logical constraints is denoted by LC.

The semantics of the language is that of classical first-order logic.

The logical constraint language $\mathcal{LL}$ is quite general since it is capable of expressing if-then rules that occur quite commonly in AI applications. The linear inequalities of the numerical constraint language $\mathcal{NL}$ allow the expression of the intuitive numerical ranges in which a threshold is known to lie and relations between these thresholds. Let us consider again the constraints in Example 1.1 and see how they will be represented in the constraint language $\mathcal{CL}$. We extend the example to include another constraint that all rich VPs become presidents of companies.

Example 2.1: The set of interval-predicates $\mathcal{IP}$ is

$$\mathcal{IP} = \{\textsf{bald},\ \textsf{old},\ \textsf{rich}\}$$

[2]Definite Horn clauses are also called *rules*.

[3]Rules without function symbols are also called Datalog sentences in the deductive database literature [Ullman, 1988].

where $\mathsf{bald}(x)$ means that x number of hairs lies in the category bald, $\mathsf{old}(x)$ means that x years lies in the category old, and $\mathsf{rich}(x)$ means that x amount of money lies in the category rich.
The set of noninterval-predicates $\mathcal{NIP}$ is

$$\mathcal{NIP} = \{age,\ hairs,\ pres,\ money,\ was\_VP\}$$

where $age(x,\ y)$ means that the person x is y years old, $hairs(x,\ y)$ means that the person x has y number of hairs, $pres(x)$ means that the person x is a president of a company, $money(x,\ y)$ means that the person x has y amount of money, and $was\_VP(x)$ means that the person x was a VP of a company.
The set of numerical constraints NC is

$$\begin{aligned} NC = \{&\mathsf{bald}^- = 0,\ 3 \le \mathsf{bald}^+ \le 20000,\ \mathsf{old}^+ = \infty, \\ &60 \le \mathsf{old}^- \le 100,\ 0.1 \le \mathsf{rich}^- \le 1,\ \mathsf{rich}^+ = \infty\ \} \\ &\cup\ \ \{\mathsf{bald}^- \le \mathsf{bald}^+,\ \mathsf{old}^- \le \mathsf{old}^+,\ \mathsf{rich}^- \le \mathsf{rich}^+\} \end{aligned}$$

The unit for bald is number of hairs, for old is age in years, and for rich is money in millions of dollars.
The set of logical constraints LC is

$$\begin{aligned} LC = \{&pres(x) \leftarrow was\_VP(x) \wedge money(x,y) \wedge \mathsf{rich}(y) \\ &\mathsf{bald}(z) \leftarrow \mathsf{old}(y)\ \wedge\ age(x,y)\ \wedge\ hairs(x,z) \\ &\mathsf{old}(y)\ \ \leftarrow pres(x)\ \wedge\ age(x,y) \\ &age(\mathtt{Tom},80),\ hairs(\mathtt{Tom},500),\ was\_VP(\mathtt{Tom}),\ money(\mathtt{Tom},6) \\ &age(\mathtt{Jim},85),\ hairs(\mathtt{Jim},800) \\ &age(\mathtt{Sam},45),\ hairs(\mathtt{Sam},650)\} \end{aligned}$$

Here, the first three constraints are rules whereas the rest of the constraints are ground literals. ∎

2.1.1 Comparison to Other Languages

The constraint language $\mathcal{CL}$ described in the last section has both a numerical as well as a logical component. There are other languages that combine quantitative and qualitative constraints, but differ from $\mathcal{CL}$ in various ways.

Williams' qualitative algebra [Williams, 1988] expresses operations on reals and signs of reals, but is not concerned with logical constraints of the kind that we have in $\mathcal{LL}$. The operations on reals form the quantitative constraints and the operations on the signs of reals (+, 0, $\Leftrightarrow$) form the qualitative constraints. There is a sign algebra that combines the two kinds of expressions.

Similarly, [Meiri, 1991] and [Kautz and Ladkin, 1991] present frameworks for expressing and processing both quantitative and qualitative temporal constraints. The quantitative constraints are constraints on the distance between time points and the qualitative constraints are interval relations between time intervals (13 possible relations between intervals are defined in [Allen, 1985]). Their language limits the constraints, whether numerical or logical, to be binary whereas our language does not. On the other hand, their language can express disjunctive relations between intervals which our language does not. It is possible in our language to express the disjunctive relations by allowing disjunctions of linear arithmetic inequalities in the numerical constraints. The algorithm discussed in Chapter 3 will still apply though some of the optimizations to speed it up will not be applicable.

Most closely related to our language is a language for *constraint logic programming* $CLP(\Re)$, in the style of Lassez et al. [Jaffar and Lassez, 1987]. $CLP(\Re)$ considers general Horn theories, as opposed to our limited Datalog theories. However, $CLP(\Re)$ does not allow numerical constraints in the head of a clause. In our language the interval-predicates can occur in the head which, if represented in $CLP(\Re)$, would correspond to numerical constraints occurring in the head. For example, if P is an interval-predicate, then the predicate-as-interval assumption says that

$$\mathsf{P}(x) \Leftrightarrow \mathsf{P}^- \leq x \leq \mathsf{P}^+$$

One way implication can be represented in $CLP(\Re)$ as

$$\mathsf{P}(x) \leftarrow \mathsf{P}^- \leq x \leq \mathsf{P}^+$$

but the implication in the other direction cannot be expressed since that would mean that the arithmetic constraint $\mathsf{P}^- \leq x \leq \mathsf{P}^+$ occurs in the head of the clause which is not allowed in $CLP(\Re)$. Hence, we cannot represent the constraints in $\mathcal{LL}$ in the language of $CLP(\Re)$.

2.2 Query Language

Given the domain knowledge in the constraint language, a user would like to extract information about thresholds. Since the constraints are often not sufficient to pinpoint an exact value for the threshold, the ability to query where the thresholds can and cannot lie becomes important. Querying is a useful tool for a system designer to find the threshold values allowed by the constraints during the design stage of a knowledge-based system. Therefore, the query language, denoted by $\mathcal{QL}$, is geared to support queries that will aid in the threshold determination process.
The kind of queries supported in the language $\mathcal{QL}$ are described below. Here

$$\begin{array}{rcl}
\mathsf{P}_1^{th},\ldots,\mathsf{P}_n^{th} & \in & \mathcal{T} \quad \text{where } \mathsf{P}^{th} \text{ is either } \mathsf{P}^- \text{ or } \mathsf{P}^+ \\
\mathsf{a}_1,\ldots,\mathsf{a}_n & \in & \Re \\
op_1,\ldots,op_n & \in & \{\wedge,\vee\} \\
rel_1,\ldots,rel_n & \in & \{\leq,\geq,<,>,=,\neq\} \text{ and} \\
i & \in & \{1,\ldots,n\}
\end{array}$$

The formal language of the query is in prefix notation and is written here in `typewriter` font. For logical expressions with *ops* $\wedge$ and $\vee$, assume the usual preference of parentheses around $\wedge$ followed by $\vee$ (*i.e.,* $p \vee q \wedge r$ should be read as $(p \vee (q \wedge r))$).

1. Is it *necessarily* the case that $(\mathsf{P}_1^{th}\ rel_1\ \mathsf{a}_1)\ op_1 \ldots op_n\ (\mathsf{P}_n^{th}\ rel_n\ \mathsf{a}_n)$?
   (`necessarily` $(op_1\ (rel_1\ \mathsf{P}_1^{th}\ \mathsf{a}_1)\ (op_2\ (rel_2\ \mathsf{P}_2^{th}\ \mathsf{a}_2)\ (op_3 \ldots))))$

2. Is it *possibly* the case that $(\mathsf{P}_1^{th}\ rel_1\ \mathsf{a}_1)\ op_1 \ldots op_n\ (\mathsf{P}_n^{th}\ rel_n\ \mathsf{a}_n)$?
   (`possibly` $(op_1\ (rel_1\ \mathsf{P}_1^{th}\ \mathsf{a}_1)\ (op_2\ (rel_2\ \mathsf{P}_2^{th}\ \mathsf{a}_2)\ (op_3 \ldots))))$

3. What is the *minimum* value that P_i^{th} can take?
   (`minimum` P_i^{th})

4. What is the *maximum* value that P_i^{th} can take?
   (`maximum` P_i^{th})

Many queries can be composed from the primitive queries defined above. For example, the query "$\mathsf{P(a)}$?" is expressible in $\mathcal{QL}$ by casting it as "Is it necessarily the case that $(\mathsf{P}^- \leq \mathsf{a}) \wedge (\mathsf{P}^+ \geq \mathsf{a})$?", or formally as

$$\texttt{(necessarily (and } (\leq\ \mathsf{P}^-\ \mathtt{a})\ (\geq\ \mathsf{P}^+\ \mathtt{a})))$$

If the answer is *yes* then $\mathsf{P(a)}$ is *true*, otherwise it is unknown. If the answer to "Is it possibly the case that $(\mathsf{P}^- \leq \mathsf{a}) \wedge (\mathsf{P}^+ \geq \mathsf{a})$?" is *no* then $\mathsf{P(a)}$ is *false*, otherwise it is unknown.

The semantics of the query (`necessarily` q) is that whether q is logically implied by $(LC \cup NC)$[4]; alternatively, whether q is true in every model where $LC \cup NC$ is true. This is also represented by the standard symbol for logical entailment $(LC \cup NC) \models q$. The semantics of the query (`possibly` q) is whether q and $LC \cup NC$ can be true together; alternatively, whether there exists a model of $LC \cup NC$ where q is true.

The semantics of the query (`minimum` P^-) is that the answer to the query is a real number $\mathtt{a}$ such that $(LC \cup NC) \models (\mathsf{P}^- \geq \mathtt{a})$ and there does not exist any other real number $\mathtt{b}$ such that $(\mathtt{b} < \mathtt{a})$ and $(LC \cup NC) \models (\mathsf{P}^- \geq \mathtt{b})$. Analogously, the query (`maximum` P^-) means that the answer to the query is the real number $\mathtt{a}$ such that $(LC \cup NC) \models (\mathsf{P}^- \leq \mathtt{a})$ and there does not exist any other real number $\mathtt{b}$ such that $(\mathtt{b} > \mathtt{a})$ and $(LC \cup NC) \models (\mathsf{P}^- \leq \mathtt{b})$.

These queries are useful for a system designer to define the vague predicates in a system precisely. For instance, to set a cutoff between bald and $\mathsf{not\ bald}$, one can initially ask for the minimum and maximum values allowed for bald^+. In this case,

[4]Throughout the dissertation, we use *set* of sentences and *conjunction* of sentences interchangeably. A set of sentences appearing in a logical expression should be taken to mean the conjunction of sentences in that set.

the first query is

$$(\texttt{minimum}\ \ \mathsf{bald}^+)$$

which returns the answer 800, and the second query is

$$(\texttt{maximum}\ \ \mathsf{bald}^+)$$

which returns the answer 20000.

These values are then useful for picking a candidate value in the allowed range $[min, max]$. So, for instance, one can pick 900 as the value for the threshold bald^+ since it lies in the allowed range $[800, 20000]$. Anything outside this range is already known to be inconsistent with the given constraints.

Once a candidate value for the threshold is picked one can check whether this value is consistent with the constraints. In this example, checking that

$$(\texttt{necessarily}\ \ (=\ \ \mathsf{bald}^+\ 900))$$

returns the answer *yes* which will ensure that 900 is a valid assignment. We consider an extension to Example 2.1 to illustrate why it is important to check the candidate value for consistency even when it lies in the $[min, max]$ range.

Example 2.2: In addition to the constraints in Example 2.1, we have the following ground facts about Bob which must be added to the set of logical constraints LC:

$$\{age(\mathsf{Bob}, 70),\quad hairs(\mathsf{Bob}, 1000)\}$$

The minimum and maximum values of old^- give us the allowed range of $[60, 80]$. Therefore, it is uncertain whether Bob is old or not. Similarly, it is uncertain whether Bob is bald or not. As before, let us choose the value of bald^+ to be 900. But now this is consistent or inconsistent depending upon what is the value chosen for old^-. For instance, if $(\mathsf{old}^- = 65)$, then we can conclude that Bob is old and therefore bald. But according to the cutoff $(\mathsf{bald}^+ = 900)$, he should not be bald since he has 1000 hairs. Therefore, $(\mathsf{old}^- = 65)$ is inconsistent with $(\mathsf{bald}^+ = 900)$. On the other hand, if the value chosen for old^- is more than 70, then there is no inconsistency. ∎

This example illustrates that the values assigned to thresholds might *individually* be consistent with the constraints, but when considered *simultaneously*, they could be inconsistent. A query such as

$$\texttt{(necessarily (and (}= \ \mathsf{bald}^+ \ 900) \ (= \ \mathsf{old}^- \ 65)))$$

would detect such inconsistencies.

Another application of the querying mechanism is at the time a value of a threshold is desired in a certain range. Then one can check if a value in that range is possible and then pick a value in that range, followed by checking as before that this is consistent with other assignments. For example, if we desire a value of bald^+ in the range $[500, 5000]$, then we can ask the following query

$$\texttt{(possibly (and (}\geq \ \mathsf{bald}^+ \ 500) \ (\leq \ \mathsf{bald}^+ \ 5000)))$$

If the answer is *yes* then we can pick a value in the range and check for consistency as before. If the answer is *no*, then no value in the range is valid.

Hence, this query language is quite general and is useful to a system designer in assigning values to the thresholds that are consistent with the given constraints.

Chapter 3

Answering Queries: Preprocessing Constraints

In Chapter 2 we described the first two parts of the framework for reasoning precisely with vague concepts: first, the constraint language in which the knowledge about the domain is expressed and, second, the query language in which the queries about the interval-predicates and their thresholds can be asked. The third part of the framework is the algorithm responsible for answering the queries on constraints. The main considerations for the algorithm are its soundness, completeness and efficiency. Soundness means that any answer to a given query must be correct. Completeness means that if an answer to a query is derivable from the constraints, then that answer must be derived, rather than returning "unknown". Usually, there is a tradeoff between the soundness and completeness of an algorithm and its efficiency. In this chapter, we will describe an algorithm that is sound and complete in Section 3.1 and establish these properties formally in Section 3.2. We will also discuss the complexity issues in Section 3.3.

3.1 Algorithm to Answer Queries

The main feature of the *query-answering* algorithm described here is that it preprocesses the constraints so that answering the query is efficient at runtime. Since the

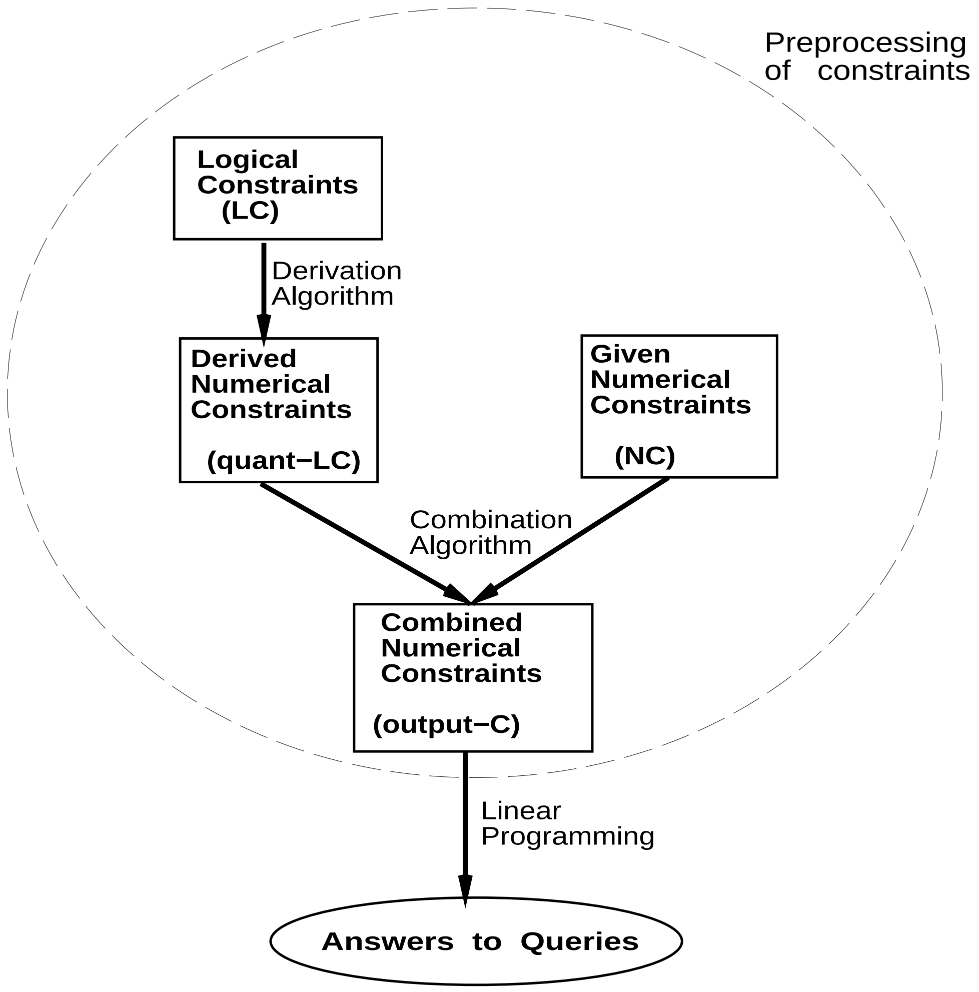


Figure 3.1: Overview of the Query-Answering Algorithm

queries require information about the thresholds of interval-predicates only, the preprocessing stage extracts the information pertaining to these thresholds while eliminating the information about the noninterval-predicates. The preprocessing is a two-step procedure: first, using the predicate-as-interval assumption, the procedure extracts the numerical information about the interval-predicates from the logical constraints LC. This derived information is in the form of disjunctions of linear arithmetic constraints. Next, the procedure combines these disjunctive constraints with the given numerical constraints NC.

Figure 3.1 is an overview of the Query-Answering Algorithm. The dotted lines mark the preprocessing of constraints. At the end of preprocessing, the constraints are in the form of disjunctions of linear arithmetic inequalities and the queries are answered using efficient linear programming techniques. We will discuss this in Chapter 4. The preprocessing stage is the subject of this chapter. The first step in preprocessing, where numerical information is derived from logical constraints LC, is called the *derivation algorithm* and is described in Section 3.1.1. The second step in preprocessing where the derived numerical information from the first step is combined with NC, is called *combination algorithm* and is described in Section 3.1.2. The formal results and complexity issues about the derivation algorithm are in Section 3.2 and Section 3.3, respectively.

3.1.1 Derivation Algorithm

The *derivation* algorithm eliminates the occurrence of noninterval-predicates from the set of given logical constraints LC while retaining all the information about the thresholds of interval-predicates. Figure 3.2 gives an overview of the derivation algorithm. The algorithm first computes the logical implication of LC that has no occurrence of any noninterval-predicate. This procedure is called *eliminate_NIP*. Then the clauses in this implication are converted to numerical constraints using the predicate-as-interval assumption for the interval-predicates. The procedure that achieves this is called *convert_to_numerical*. A description of the derivation algorithm is in Algorithm 3.1.

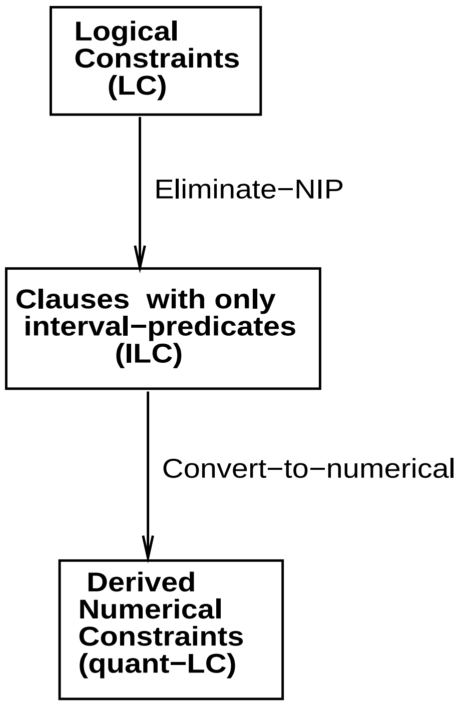

Logical Constraints (LC)
Eliminate−NIP
Clauses with only interval−predicates (ILC)
Convert−to−numerical
Derived Numerical Constraints (quant−LC)

Figure 3.2: Derivation Algorithm

ALGORITHM 3.1 (Derivation)

Input: Set of logical constraints LC.
Set of interval-predicates $\mathcal{IP}$.
Set of noninterval-predicates $\mathcal{NIP}$.

Output: Set of arithmetic constraints $quant\_LC$ derived from LC by eliminating predicates from $\mathcal{NIP}$ and converting to numerical constraints using predicate-as-interval assumption on predicates from $\mathcal{IP}$.

Method:

```
$ILC \leftarrow Eliminate\_NIP(LC, \mathcal{IP}, \mathcal{NIP})$.
                                    % $ILC$ is the set of clauses derived from $LC$
                                    % such that it has only interval-predicates
$quant\_LC \leftarrow Convert\_to\_numerical(ILC)$.
**Return**($quant\_LC$)                                                        ∎
```

Eliminating Noninterval-Predicates

The first step of the derivation algorithm is Algorithm 3.2 that eliminates the noninterval-predicates from the logical constraints while retaining all the information about the interval-predicates.

ALGORITHM 3.2 (Eliminate_NIP)

```
Input: Set of logical constraints $LC$.
       Set of interval-predicates $\mathcal{IP}$.
       Set of noninterval-predicates $\mathcal{NIP}$.
Output: $ILC$ the set of clauses derived from $LC$
            such that it has only interval-predicates
Method:
    Initialize $ILC \leftarrow \emptyset$.
    For every clause $c \in LC$ such that $head(c) \in \mathcal{IP}$ do
        $ILC_c \leftarrow Expand(c, LC, \mathcal{IP}, \mathcal{NIP})$.
        $ILC \leftarrow ILC \cup ILC_c$.
    Endfor
    Return($ILC$)                                              ▮
```

Starting with all those clauses in LC that have interval-predicates at the *head*, we *expand* their bodies using other clauses in LC until all noninterval-predicates are eliminated from the body. Intuitively, the procedure *expand* does the job of extracting from the noninterval-predicates all the information that they contribute to the thresholds and then eliminates them. A description of the *expand* procedure is in Algorithm 3.3.

ALGORITHM 3.3 (Expand)

```
Input: Clause $c$ from the set of logical constraints $LC$
            such that its head has an interval-predicate.
       The set of logical constraints $LC$.
       The set of interval-predicates $\mathcal{IP}$.
```

```
          The set of interval-predicates $\mathcal{NIP}$.
**Output**: The set of clauses obtained by expanding $c$ using clauses in $LC$
               such that only noninterval-predicates are expanded.
**Method**:
    **If** $body(c)$ *is empty* **or**
      for every literal $l \in body(c)$ it is case that $l \in \mathcal{IP}$
    **Then return**($\{c\}$)
    **Else** Initialize $S \leftarrow \emptyset$.
        **For** every literal $l \in body(c)$ such that $l \in \mathcal{NIP}$ **do**
          **For** every clause $r \in LC$ such that $unifiable(l, head(r))$ **do**
                                                  % *notation for unification from*
                                    % *[Genesereth and Nilsson, 1987, Section4.2]*
            $\theta \leftarrow mgu(l, head(r))$.
            $new\_c \leftarrow [head(c) \leftarrow (body(c) \Leftrightarrow \{l\}) \cup body(r)]\theta$.
            $S \leftarrow S \cup Expand(new\_c, LC, \mathcal{IP}, \mathcal{NIP})$.
            **Endfor**
        **Endif**
    **Return**($S$)                                                              ∎
```

Expand is very similar to SLD resolution [Lloyd, 1987] but with two differences: (1) only noninterval-predicates are expanded (2) all possible expansions are computed. Since only the noninterval-predicates are expanded, we are finally left with only interval-predicates. Computing all possible expansions ensures that we are not losing any information in this process. The expansion of a clause can also be represented by a derivation tree.

The derivation tree is a tree where the head of the rule being expanded is at the root of the tree and each literal in the body of the rule is a child of the root. Then, each further expansion of a literal leads to the subsequent expanded literals being added as children. The leaf nodes form the literals in the expansion of the root.

An application of the algorithm on Example 2.1 is illuminating:

Example 3.1:

Restating the set of logical constraints LC

$$\begin{aligned} LC = \{&pres(x) \leftarrow was\_VP(x) \wedge money(x,y) \wedge \mathsf{rich}(y)\\ &\mathsf{bald}(z) \leftarrow \mathsf{old}(y) \wedge age(x,y) \wedge hairs(x,z)\\ &\mathsf{old}(y) \leftarrow pres(x) \wedge age(x,y)\\ &age(\mathtt{Tom},80),\ hairs(\mathtt{Tom},500),\ was\_VP(\mathtt{Tom}),\ money(\mathtt{Tom},6)\\ &age(\mathtt{Jim},85),\ hairs(\mathtt{Jim},800)\\ &age(\mathtt{Sam},45),\ hairs(\mathtt{Sam},650)\} \end{aligned}$$

The first step in the procedure is to locate clauses with interval-predicates at the head. Here there are two such clauses: one with **bald** at the head, and the other with **old** at the head.

For the clause with **bald** at the head, we do not need to expand the interval-predicate **old** in the body but only the two noninterval-predicates *age* and *hairs*. For each of these, we have three choices for unification: one each by instantiating x with `Tom`, `Jim` and `Sam` respectively. Therefore, we get three expanded clauses in ILC:

$$\begin{aligned} \mathsf{bald}(500) &\leftarrow \mathsf{old}(80)\\ \mathsf{bald}(800) &\leftarrow \mathsf{old}(85)\\ \mathsf{bald}(650) &\leftarrow \mathsf{old}(45) \end{aligned}$$

For the clause in LC with **old** at the head, we can expand the noninterval-predicate *pres* with another rule from LC to obtain

$$\mathsf{old}(y) \leftarrow was\_VP(x) \wedge money(x,z) \wedge \mathsf{rich}(z) \wedge age(x,y)$$

On expanding further, we find that we can only instantiate x with `Tom`. Therefore, we get one more expanded clause in ILC:

$$\mathsf{old}(80) \leftarrow \mathsf{rich}(6)$$

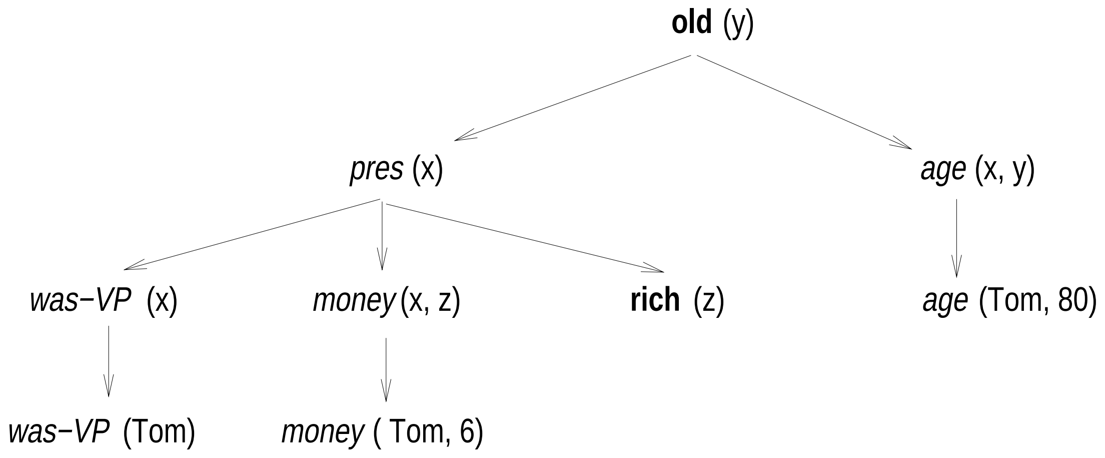


Figure 3.3: Derivation Tree for Expansion of old

The derivation tree corresponding to the expansion for old is in Figure 3.3. Note that there will be three derivation trees for bald, each corresponding to the three possible expansions of bald. ∎

The *eliminate_NIP* procedure described above is top-down in that we start with a rule and expand its body-literals until we get to the ground facts. Instead, we could compute ILC in a bottom-up manner, starting from the ground facts and going up to the rules. This would require using an iterative *expand* algorithm rather than the recursive one that we have now. This is analogous to backward chaining rather than forward chaining in a rule-based system. The bottom-up method is discussed next.

Bottom-up Elimination of Noninterval Predicates The intuition behind this method is to expand the predicates in such a fashion so that a predicate being expanded depends only on the predicates that have already been expanded. There are two advantages: firstly, it avoids any duplication of work in expanding a predicate and secondly, the information required for one expansion is already available (therefore, one computation need not be deferred for the result of another). To get the dependency relation between the predicates, we construct the dependency graph.

The dependency graph $G = (V, E)$ corresponding to the set of logical constraints LC is meant to represent the dependencies between the predicates occurring in LC.

The set of nodes V of the graph is the set of all predicates, interval as well as noninterval predicates; *i.e.,* $V = \mathcal{IP} \cup \mathcal{NIP}$. There is a directed edge (p, q) in the graph if and only if there is a rule in LC where predicate q occurs in the head of the rule and predicate p occurs in its body. Since there is no recursion among the rules, the graph will always be acyclic and hence we can perform a topological sort on the graph. Then, a predicate depends only on the predicates that are lower in the topological order. This allows us to expand the predicates in the topological order and satisfy our intuition behind the bottom-up method. The algorithm for elimination of noninterval-predicates is described in Algorithm 3.4 and illustrated with Example 3.2.

ALGORITHM 3.4 (Eliminate_NIP_bottomup)

```
Input: Set of logical constraints $LC$.
       Set of interval-predicates $\mathcal{IP}$.
       Set of noninterval-predicates $\mathcal{NIP}$.
Output: $ILC$ the set of clauses derived from $LC$
          such that it has only interval-predicates
Method:
    Construct dependency graph $G = (V, E)$ from $LC$.
    Do a topological sort of $G$ [Cormen et al., 1986, Section 23.4].
    Expand nodes of $G$ in topological order.
    Expansions of nodes that are interval-predicates form the desired set $ILC$. ∎
```

Example 3.2: The dependency graph corresponding to LC in Example 3.1 is depicted in Figure 3.4.
Expanding in the topological order, we have the following expansions:

expansion(*was_VP*) = { *was_VP*(`Tom`) }
expansion(*money*) = { *money*(`Tom, 6`) }
expansion(rich) = { }
expansion(*pres*) = { *pres*(`Tom`) $\leftarrow$ rich(6) }
expansion(*age*) = { *age*(`Tom, 80`), *age*(`Jim, 85`), *age*(`Sam, 45`) }
expansion(old) = { old(80) $\leftarrow$ rich(6) }

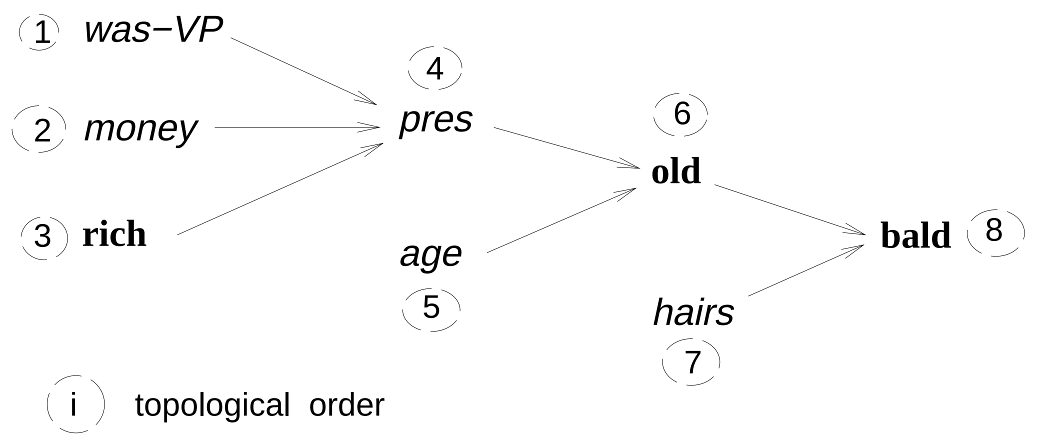


Figure 3.4: Dependency Graph

$$
\begin{aligned}
\text{expansion}(\textit{hairs}) &= \{\ \textit{hairs}(\texttt{Tom, 500}), \textit{hairs}(\texttt{Jim, 800}), \textit{hairs}(\texttt{Sam, 650})\ \}\\
\text{expansion}(\mathsf{bald}) &= \{\ \mathsf{bald}(500) \leftarrow \mathsf{old}(80), \mathsf{bald}(800) \leftarrow \mathsf{old}(85),\\
&\qquad \mathsf{bald}(650) \leftarrow \mathsf{old}(45)\ \}
\end{aligned}
$$

Since rich, old and bald are the interval-predicates, ILC is the union of their expansions, *i.e.,*

$$
\begin{aligned}
ILC = \{\quad & \mathsf{old}(80) & \leftarrow\quad & \mathsf{rich}(6)\\
& \mathsf{bald}(500) & \leftarrow\quad & \mathsf{old}(80)\\
& \mathsf{bald}(800) & \leftarrow\quad & \mathsf{old}(85)\\
& \mathsf{bald}(650) & \leftarrow\quad & \mathsf{old}(45)\ \}
\end{aligned}
$$

∎

Converting to Numerical Constraints

The first part of the derivation algorithm preprocesses the set of logical constraints LC to eliminate the noninterval-predicates such that we are left with the set ILC that has clauses with only interval-predicates. In the second step, we convert these clauses into numerical constraints using the predicate-as-interval assumption. The algorithm that achieves this is called *Convert_to_numerical* and is described as Algorithm 3.5.

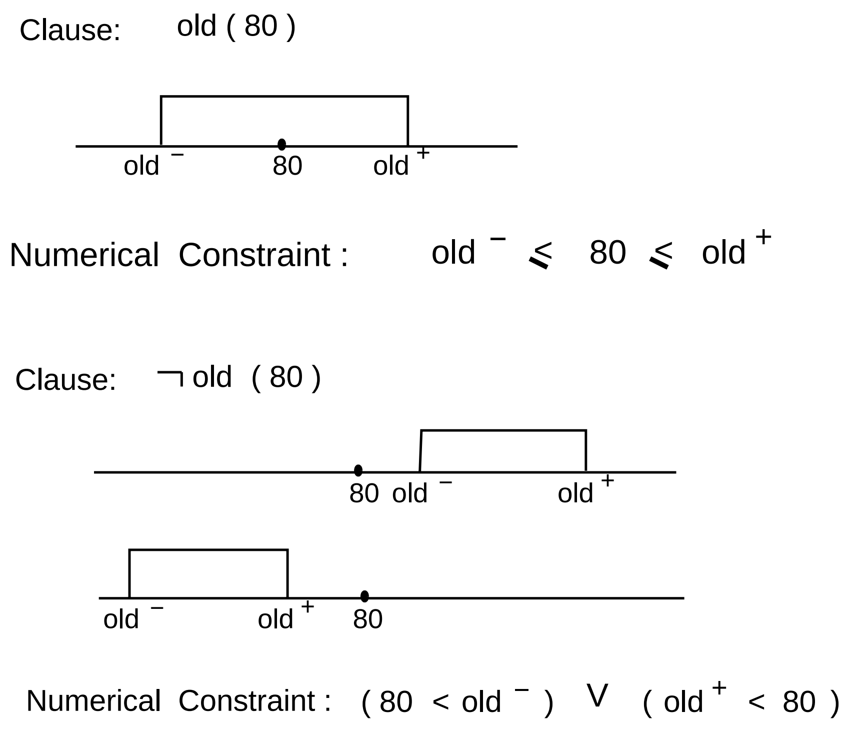
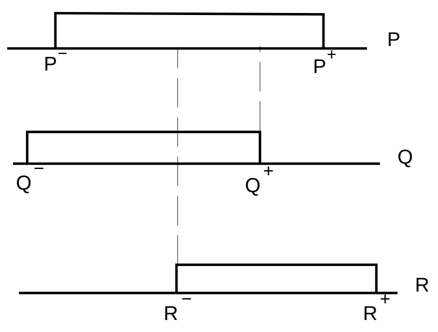

Clause: old (80)
old −
80
old +
Numerical Constraint : old − ≤ 80 ≤ old +
Clause: ¬ old (80)
80 old −
old +
old −
old +
80
Numerical Constraint : (80 < old −) ∨ (old + < 80)

Figure 3.5: Converting clause with constant to numerical constraint

The intuition behind this algorithm is that due to the predicate-as-interval assumption, we can interpret an interval-predicate as an interval over the real number line. Therefore, an atomic clause such as old(80) means that 80 must lie within the interval corresponding to old which will be $[\mathsf{old}^-, \mathsf{old}^+]$. Similarly, a clause such as $\neg$old(80) means that 80 must *not* lie within the interval corresponding to old. This idea is illustrated in Figure 3.5. If we consider a clause with a variable, such as $\mathsf{P}(x) \leftarrow \mathsf{Q}(x), \mathsf{R}(x)$, we can reason about the three intervals corresponding to the interval-predicates P, Q, R and deduce relationships between the corresponding thresholds, as shown in Figure 3.6.

We observe that since the interval-predicates are unary, we can fragment any clause in *ILC* into subclauses that have only a constant or only one variable. This is used to decompose every clause in *ILC* into subclauses that fall into one of the six

Clause : P (x) <-- Q (x) , R (x)



numerical constraints:

$P^- \leq \max(Q^-, R^-)$

$P^+ \geq \min(Q^+, R^+)$

which is equivalent to:

$(P^- \leq Q^-) \vee (P^- \leq R^-)$

$(P^+ \geq Q^+) \vee (P^+ \geq R^+)$

Figure 3.6: Converting clause with variable to numerical constraint

categories discussed in Algorithm 3.5.

ALGORITHM 3.5 (Convert_to_numerical)

Input: Set of clauses *ILC* that has only interval-predicates.
Output: Set of arithmetic constraints *quant_LC* obtained
by converting *ILC* using the predicate-as-interval assumption.
Method:

```
Initialize $quant\_LC \leftarrow \emptyset$.
For every clause $lc \in ILC$ do
    Initialize $nc \leftarrow emptyset$.
                        % nc is the numerical constraint obtained by converting lc
    $lc\_subclauses \leftarrow Make\_Subclauses(lc)$.    % Make_subclauses breaks lc
                     % into subclauses with only a constant or only one variable each
    For every subclause $subcl \in lc\_subclauses$ do
        Case subcl of:                                          % a is a constant

          "P(a)":                        $subcl' \leftarrow (P^- \leq a \leq P^+)$
```

```
            "$\leftarrow \mathsf{P}(\mathsf{a})$":        $subcl' \leftarrow (\mathsf{a} < \mathsf{P}^-) \vee (\mathsf{a} > \mathsf{P}^+)$
            "$\mathsf{P}(x)$":          $subcl' \leftarrow (\mathsf{P}^- = \Leftrightarrow\infty) \wedge (\mathsf{P}^+ = +\infty)$
            "$\leftarrow \mathsf{P}(x)$":        $subcl' \leftarrow \mathsf{P}^- > \mathsf{P}^+$
            "$\leftarrow \mathsf{P}_1(x), \ldots, \mathsf{P}_n(x)$":
                $subcl' \leftarrow \vee_{i=1}^{n} \vee_{j=1}^{n} (\mathsf{P}_i^- > \mathsf{P}_j^+)$
            "$\mathsf{P}(x) \leftarrow \mathsf{Q}_1(x), \ldots, \mathsf{Q}_n(x)$":
                  $subcl' \leftarrow (\vee_{i=1}^{n} \mathsf{P}^- \leq \mathsf{Q}_i^-) \wedge (\vee_{i=1}^{n} \mathsf{P}^+ \geq \mathsf{Q}_i^+)$.
          $nc \leftarrow nc \vee subcl'$
      Endfor
      $quant\_LC \leftarrow quant\_LC \cup nc$.
  Endfor
  Return($quant\_LC$)                                                          ∎
```

This algorithm works by fragmenting each clause in *ILC* into subclauses such that each subclause has at most one variable and no two subclauses have the same variable. Note that this is always possible because all interval-predicates are unary. For example, the clause $\mathsf{P}(\mathsf{a}) \leftarrow \mathsf{Q}(x) \wedge \mathsf{R}(x) \wedge \mathsf{S}(\mathsf{b})$ is a disjunction of three subclauses: "$\mathsf{P}(\mathsf{a})$", "$\leftarrow \mathsf{Q}(x) \wedge \mathsf{R}(x)$" and "$\leftarrow \mathsf{S}(\mathsf{b})$". In general, each subclause thus obtained will be one of the six basic types described in Algorithm 3.5. Each type of subclause is converted to a numerical constraint by using the predicate-as-interval assumption, and by interpreting the connectives $\neg, \vee, \wedge$ as complement, union and intersection of intervals, respectively. Therefore, for any clause in *ILC*, we can always decompose it into a disjunction of subclauses, each of which can be converted to disjunctions of linear arithmetic inequalities. Any clause in *ILC* is thus converted to a set of disjunctions of linear arithmetic inequalities.

We apply this algorithm to Example 3.2 to get the *quant_LC* shown in Example 3.3.

Example 3.3: In this example, each clause in *ILC* gets fragmented into subclauses of the first two types: "$\mathsf{P}(\mathsf{a})$" and "$\leftarrow \mathsf{P}(\mathsf{a})$".

$\mathsf{bald}(500) \leftarrow \mathsf{old}(80)$ has two subclauses and is equivalent to their disjunction

$$\text{“}\mathsf{bald}(500)\text{”} \vee \text{“}\leftarrow \mathsf{old}(80)\text{”}$$

Converting each subclause using the *convert_to_numerical* algorithm, we get the numerical constraint

$$(\mathsf{bald}^- \leq 500 \leq \mathsf{bald}^+) \vee \ (80 < \mathsf{old}^-) \ \vee \ (\mathsf{old}^+ < 80)$$

which is equivalent to the set of numerical constraints

$$\begin{aligned}\{&(\mathsf{bald}^- \leq 500) \vee \ (80 < \mathsf{old}^-) \ \vee \ (\mathsf{old}^+ < 80),\\ &(500 \leq \mathsf{bald}^+) \vee \ (80 < \mathsf{old}^-) \ \vee \ (\mathsf{old}^+ < 80)\}\end{aligned}$$

Therefore, the set of numerical constraints *quant_LC* derived from *LC* is

$$\begin{aligned} quant\_LC = \{&(\mathsf{bald}^- \leq 500) \vee (80 < \mathsf{old}^-) \ \vee \ (\mathsf{old}^+ < 80)\\ &(500 \leq \mathsf{bald}^+) \vee (80 < \mathsf{old}^-) \ \vee \ (\mathsf{old}^+ < 80)\\ &(\mathsf{old}^- \leq 80) \vee (6 < \mathsf{rich}^-) \ \vee \ (\mathsf{rich}^+ < 6)\\ &(80 \leq \mathsf{old}^+) \vee \ (6 < \mathsf{rich}^-) \ \vee \ (\mathsf{rich}^+ < 6)\\ &(\mathsf{bald}^- \leq 800) \vee (85 < \mathsf{old}^-) \ \vee \ (\mathsf{old}^+ < 85)\\ &(800 \leq \mathsf{bald}^+) \vee \ (85 < \mathsf{old}^-) \ \vee \ (\mathsf{old}^+ < 85)\\ &(\mathsf{bald}^- \leq 650) \vee (45 < \mathsf{old}^-) \ \vee \ (\mathsf{old}^+ < 45)\\ &(650 \leq \mathsf{bald}^+) \vee \ (45 < \mathsf{old}^-) \ \vee \ (\mathsf{old}^+ < 45) \ \}\end{aligned}$$

∎

3.1.2 Combining with Numerical Constraints

The first part in the preprocessing of constraints is the derivation of numerical constraints from the logical constraints. The second part is combining these derived constraints *quant_LC* with the given set of numerical constraints *NC* to be able to answer the queries. In principle, we could just take the union of the two sets of constraints to get the combined set that we call *output_C*. Then answering a query would require computing the answer over *output_C*. We first discuss what kind of procedures are required to answer a query over *output_C* and hence justify the need to do something more to combine the two sets *quant_LC* and *NC* than just taking their union. We then describe the *combination* algorithm and illustrate it through an example.

There are four kinds of queries that are allowed in the query language, as discussed in Chapter 2. Of these, the first two queries that ask whether a certain statement is *necessarily* or *possibly* true can be answered by adding the statement to the constraint set *output_C* and checking whether the resulting set is consistent. The existing linear programming techniques to check consistency of linear inequalities are useful. These are usually applied to constraints without disjunctions. The third and fourth queries that ask for *minimum* and *maximum* values of thresholds can be answered by applying linear optimization techniques to *output_C*. Again, these techniques are usually applied to constraints without disjunctions. Hence, we need to consider the disjunctive normal form (DNF) of *output_C* so that we can apply these techniques to each disjunct (that will have only conjunctions of linear inequalities) separately and then combine the result. Consider Example 3.4 where these queries are asked on the sets *quant_LC* from Example 3.3 and *NC* from Example 2.1:

Example 3.4:
Restating the numerical constraints from Example 2.1:

$$\begin{aligned} NC = \{ & \mathsf{bald}^- = 0,\ 3 \le \mathsf{bald}^+ \le 20000,\ \mathsf{old}^+ = \infty, \\ & 60 \le \mathsf{old}^- \le 100,\ 0.1 \le \mathsf{rich}^- \le 1,\ \mathsf{rich}^+ = \infty\ \} \\ & \cup\ \ \{\mathsf{bald}^- \le \mathsf{bald}^+,\ \mathsf{old}^- \le \mathsf{old}^+,\ \mathsf{rich}^- \le \mathsf{rich}^+\} \end{aligned}$$

The combined set of constraints *output_C* is

$$output\_C \ = \ quant\_LC \ \cup \ NC$$

Since *quant_LC* has disjunctive constraints, so will *output_C*. In the DNF form, *output_C* has 3^8 possible disjunctions.

Consider the query (`necessarily` ($\mathsf{bald}^+ \le 1000$)). We need to add the negation of the statement, *i.e.,* ($\mathsf{bald}^+ > 1000$), to each of the 3^8 possible disjuncts of *output_C*. If the negation is consistent with any of the disjuncts, then the original statement is not necessarily true, and hence the answer to the query will be *no*. If the negation is inconsistent with every disjunct then the original statement is necessarily true and hence the answer to the query is *yes*. In this example, ($\mathsf{bald}^+ > 1000$) is consistent with all disjuncts; hence, the answer to the query is *no*.

For the query (`possibly` ($\mathsf{bald}^+ \leq 1000$)), if the statement is consistent with at least one disjunct then the answer to the query is *yes*, otherwise it is *no*. Here, it is consistent with every disjunct in fact; hence, the answer to the query is *yes*.

For the query (`minimum` bald^+), we compute the minimum value of bald^+ in every disjunct and the least among all the minimums is the answer to the query. Analogously for finding the maximum. Here, the minimum is 800 and the maximum is 20000. ∎

We observe from the example that we need to consider the DNF of *output_C* for computing the answer, which might be large in many cases. Hence, it is important to use heuristics to prune the size of this set and reduce the number of disjuncts that need to be checked. Because of these efficiency concerns, the process of combining *quant_LC* and *NC* is more complex than simply taking their union. We first prune the size of *quant_LC* using the set *NC* before taking their union, and then further detect redundant disjuncts in *output_C*. The queries are answered using the existing linear programming techniques on this set. We describe the overall combination procedure in Algorithm 3.6. The details of the heuristics for pruning and speedup are discussed in Chapter 4 as also the application of existing linear programming and inequality reasoning methods to answer queries on a single disjunct.

ALGORITHM 3.6 (Combination)

```
Input: Set of derived disjunctive numerical constraints quant_LC.
       Set of given numerical constraints NC.
Output: Set of combined numerical constraints output_C.
Method:
    Reduce(quant_LC, NC).      % Uses NC to reduce the size of quant_LC
    Cover(quant_LC).           % Remove redundant disjuncts from quant_LC
                               % using the cover method
    Construct output_C by generating the DNF of quant_LC
              and adding NC to each disjunct.                          ∎
```

3.2 Formal results on conservation of numerical information

In the last section we discussed the algorithm to preprocess the constraints before answering the queries. In the first stage of preprocessing, we derive the numerical information from the logical constraints using the *derivation* algorithm. In this section, we establish formally that no numerical information is lost in the derivation process and hence the query answers are sound and complete.

We first describe the notation used to develop the framework in Section 3.2.1. Then we establish the soundness and completeness results in Section 3.2.2. The complexity discussion is deferred to Section 3.3.

3.2.1 Notation

Let LC denote the set of logical constraints in the logical constraint language $\mathcal{LL}$, let NC denote the set of numerical constraints in the language $\mathcal{NL}$ and let $\mathcal{T}$ denote the set of threshold terms in the language. Also, let $\mathcal{IP}$ be the set of interval-predicates, $\mathcal{NIP}$ be the set of noninterval-predicates, $\Re$ be the set of real numbers and $\mathcal{N}$ be the set of natural numbers.

We will use the font $\mathsf{P}, \mathsf{Q}, \ldots$ for the predicates in $\mathcal{IP}$. There are two *threshold terms* associated with each interval-predicate P: P^- and P^+ denoting the lower and upper thresholds, respectively; *i.e.,*

$$\mathcal{T} = \{\mathsf{P}^-, \mathsf{P}^+ \mid \mathsf{P} \in \mathcal{IP}\}$$

The set LC of logical constraints is partitioned into two sets:

1. GF is the set of atomic ground literals in LC.
2. RS is the set of rules in LC that are not atomic ground literals. Note that the constraints of the form $P(x)$ that are atomic but not ground are in RS since they can be looked upon as a rule $P(x) \leftarrow \emptyset$ with $P(x)$ as the head of the rule with an empty body.

Also note that $RS \cup GF = LC$, and $RS \cap GF = \emptyset$.

For any rule $r \in RS$, we define the following functions that can apply to the rule:

$preds(r)$	is the set of predicates occurring in r
$head(r)$	is the literal occurring in the head of r
$body(r)$	is the subclause that is the body of r
$head\_pred(r)$	is the predicate that occurs in the head of r
$body\_preds(r)$	is the set of predicates in the body of r

We also define some functions on any predicate p where $p \in \mathcal{IP} \cup \mathcal{NIP}$.

- $GF(p)$ is the set of all ground literals with predicate p, *i.e.,*

$$GF(p) = \{l \mid l \in GF,\ head\_pred(l) = p\}$$

- $GF\_consts(p)$ is the set of all constant tuples that appear in the ground literals with predicate p, *i.e.,*

$$GF\_consts(p) = \{\bar{\mathsf{a}} \mid p(\bar{\mathsf{a}}) \in GF(p)\}$$

  where $\bar{\mathsf{a}}$ is a tuple of constants.

- $Rules(p)$, also denoted by R_p, is the set of all rules that have the predicate p at the head, *i.e.,*

$$R_p = \{r \mid r \in RS,\ head\_pred(r) = p\}$$

The algorithm builds the dependency graph corresponding to LC and then computes the expansions of predicates using the graph. Next, we describe the notation corresponding to the dependency graph and these expansions.

Dependency Graph

The dependency graph G for the constraint set LC has the set of nodes V and the set of edges E, *i.e.,* $G = (V, E)$ where $V = \mathcal{IP} \cup \mathcal{NIP}$. For every rule $r \in RS$, we define a set of edges corresponding to the rule, called $rule\_edges(r)$

$$rule\_edges(r) = \{(u,v) \mid u \in body\_preds(r),\ v = head\_pred(r)\}$$

The set E of edges in the graph is defined in terms of these rule-edges.

$$E = \{rule\_edges(r) \mid r \in RS\}$$

We also define the following functions on G and on its nodes u, v:

$$\begin{aligned} in(v) &= \{\ u \mid (u,v) \in E\ \} \\ out(v) &= \{\ u \mid (v,u) \in E\ \} \\ sources(G) &= \{\ v \mid v \in V,\ in(v) = \emptyset\ \} \\ sinks(G) &= \{\ v \mid v \in V,\ out(v) = \emptyset\ \} \end{aligned}$$

Once the graph is constructed, the algorithm performs a topological sort on the graph. We define the function $topo : V \rightarrow \mathcal{N}$, where $\mathcal{N}$ is the set of natural numbers, that specifies a topological order number for each node (and hence predicate) in the graph. $topo(v)$ for any $v \in V$ is the topological order assigned to node v on performing the topological sort on G. Note that even though the topological sort of G is not unique, nevertheless, our arguments work for any particular sorting. The following Lemma 3.1 establishes the relationship between the topological order number of the predicates that occur in a single rule of LC.

Lemma 3.1: *For every rule $r \in RS$, if u is any predicate in the body and v is the predicate in the head of the rule, i.e., $u \in body\_preds(r)$ and $v = head\_pred(r)$, then $topo(v) > topo(u)$.*
In other words, if $(u,v) \in rule\_edges(r)$, then $topo(v) > topo(u)$.

Proof: By definition of $rule\_edges$, we have that $(u,v) \in rule\_edges(r)$. Since there is an edge (u,v) in the graph G, therefore u should occur before v in the topological order (by definition of topological order of a graph). Hence, $topo(v) > topo(u)$. ∎

Expansions

For any constraint $r \in LC$, we want to define the expansion of r as produced by Algorithm 3.3, *i.e.,* $expn(r)$ must be the set of descendants of r in the derivation tree (Section 3.1.1) of r in LC, where the predicates from $\mathcal{IP}$ are never expanded. We define the expansions more precisely below:

1. If $r \in GF$, then $expn(r) = \{r\}$

2. If $r \in RS$, let r be of the form

$$m \leftarrow l_1 \wedge \ldots \wedge l_k$$

where $m, l_1, \ldots, l_k$ are literals in the body of r. We define an *expand_step* of r as expanding any one literal in the body of r only once by using a rule in LC. We say that

$$r' = expand\_step(r, l_i, r_{i1})$$

where l_i is a literal in the body of r and l_i has predicate p such that $p \notin \mathcal{IP}$, r_{i1} is a rule in LC that has predicate p in the head and r' is obtained from r by expanding l_i using r_{i1}. In other words, rule $r_{i1} \in R_{l_i}$ and is of the form

$$l'_i \leftarrow d_1 \wedge \ldots \wedge d_j$$

(Note: Since $r_{i1} \in R_{l_i}$, therefore l'_i has predicate p also.) On unifying l and l'_i we get the most general unifier $mgu(l_i, l'_i) = \theta$, and the rule r' obtained by expanding r is

$$[m \leftarrow l_1 \wedge \ldots \wedge l_{i-1} \wedge d_1 \wedge \ldots \wedge d_j \wedge l_{i+1} \wedge \ldots \wedge l_k]\theta$$

Then we can define $expn(r)$ in terms of *expand_step* as

$$expn(r) = \{r' \mid r' = expand\_step(r, l, r_l), \text{ for every } l \in body(r) \text{ and } pred(l) \notin \mathcal{IP},$$

$$\text{and for every } \; r_l \in R_l\}$$

(Note: If $body(r) = \emptyset$, then $expn(r) = \{r\}$)

We define the set of expansions of r that have only interval-predicates in the body as

$$IP\_expn(r) = \{r' \mid r' \in expn(r), \text{ and for every } \; P \in body\_preds(r'),\; P \in \mathcal{IP}\;\}$$

Similarly, we define the set of expansions of r that have only either interval-predicates or source nodes from the graph as

$$source\_expn(r) = \{r' \mid r' \in expn(r), \text{ and for all } P \in body\_preds(r), \\ \text{either } P \in \mathcal{IP} \text{ or } P \in sources(G)\}$$

Also note that $IP\_expn(r) \subseteq source\_expn(r)$
Similarly, we can define the source expansions for a predicate $p \in V$,

$$source\_expn(p) = \{r' \mid r' \in source\_expn(r), \text{ where } r \in LC \text{ and } head\_pred(r) = p\}$$

We note that for all noninterval-predicates that are source nodes of the graph, the source expansion is the set of ground literals, *i.e.,*

$$\forall\, p \in sources(G) \;\text{ where } p \in \mathcal{NIP}\;,\; source\_expn(p) = GF(p)$$

Similarly, we define IP_expn for a predicate $p \in V$,

$$IP\_expn(p) = \{r' \mid r' \in IP\_expn(r), \text{ where } \; r \in LC \text{ and } head\_pred(r) = p\}$$

Again note that for all noninterval-predicates that are source nodes of the graph, the IP_expn is the set of ground literals, *i.e.,*

$$\forall\, p \in sources(G) \;\text{ where } p \in \mathcal{NIP}\;,\; IP\_expn(p) = GF(p)$$

The set of clauses ILC obtained by eliminating noninterval-predicates can now be defined in terms of the *IP_expn* of all the nodes of the graph. The procedure *Eliminate_NIP* in Algorithm 3.2 as well as procedure *Eliminate_NIP_bottomup* in Algorithm 3.4 compute exactly the set ILC defined below

$$ILC \;=\; \cup_{v \in \mathcal{IP}}\, IP\_expn(v)$$

3.2.2 Soundness and Completeness

We establish formally that no numerical information is lost in the conversion performed by algorithm *Derivation*. We begin by defining the models of LC that are faithful to the predicate-as-interval assumption; we call these the *standard models*. Specifically, in all standard models $M = (D, \mu)$ over a domain D, the interpretation function μ will have to map interval-predicates to intervals over the reals. In the following, $\Re$ denotes the set of real numbers.

Definition 1: [*Standard Model*] Given a set of logical constraints LC, the set of interval-predicates $\mathcal{IP}$, and the set of threshold constants $\mathcal{T}$, a *standard model of LC w.r.t.* $\mathcal{IP}$ is a model $M = (D, \mu)$ such that $D = \Re$, $M \models LC$, and for every $\mathsf{P} \in \mathcal{IP}$ there exist $\mathsf{P}^-, \mathsf{P}^+ \in \mathcal{T}$ and it is the case that $\mu(\mathsf{P}^-), \mu(\mathsf{P}^+) \in \Re$ and $\mu(\mathsf{P}) = \{x \mid \mu(\mathsf{P}^-) \leq x \leq \mu(\mathsf{P}^+),\ x \in \Re\}$. (Here, the arithmetic operators $<, >, \leq, \geq, =$ have the usual interpretation over $\Re$.) ▮

Definition 2: [*Numerical Submodel*] Given LC, $\mathcal{IP}$ and $\mathcal{T}$ as above, a *numerical submodel of LC w.r.t.* $\mathcal{IP}$ is a model $M = (\Re, \mu)$ such that there is some standard model $M' = (D, \mu')$ of LC w.r.t. $\mathcal{IP}$, and μ is the restriction of μ' to terms in $\mathcal{T}$. ▮

The following theorem establishes that the algorithm *Derivation* is sound and complete w.r.t. the numerical information. Note that this result holds irrespective of what query language is used as long as it is querying the thresholds.

Theorem 3.2: *(*Derivation:Soundness and Completeness*) The class of numerical submodels of LC w.r.t.* $\mathcal{IP}$ *is identical to the class of models over* $\Re$ *of quant_LC.*

Proof: We first prove that the class of numerical submodels of LC w.r.t. $\mathcal{IP}$ is identical to the class of standard models of ILC (Theorem 3.3) and then prove that the class of standard models of ILC is identical to the class of models of *quant_LC* (Theorem 3.5). ∎

(For the following, note that the models over $\Re$ of *quant_LC* are the same as the numerical submodels of *quant_LC* because it has only thresholds of interval-predicates. Also, the standard models of ILC are the same as the numerical submodels of ILC (w.r.t. the same $\mathcal{IP}$) because ILC has no noninterval-predicates.)

Theorem 3.3: *(*Eliminate_NIP: Soundness and Completeness*) The class of numerical submodels of LC w.r.t. $\mathcal{IP}$ is identical to the class of standard models of ILC w.r.t. $\mathcal{IP}$.*

Proof: *Soundness:* Every numerical submodel of LC w.r.t. $\mathcal{IP}$ can be extended to a standard model of LC w.r.t $\mathcal{IP}$ (if LC is consistent. But, since LC has only definite Horn clauses, it is always consistent). $LC \Rightarrow ILC$ (from the definition of ILC). Therefore, every model of LC is also a model of ILC. Hence, every standard model of LC w.r.t. $\mathcal{IP}$ is also a standard model of ILC w.r.t. $\mathcal{IP}$ (note that it is the same $\mathcal{IP}$) , and the numerical submodels of LC w.r.t. $\mathcal{IP}$ are also numerical submodels of ILC w.r.t. $\mathcal{IP}$. Since ILC has only predicates from $\mathcal{IP}$, its numerical submodels are the same as its standard models. Hence, we have the soundness: Every numerical submodel of LC w.r.t. $\mathcal{IP}$ is a standard model of ILC w.r.t. $\mathcal{IP}$.
Completeness:
Consider an arbitrary standard model $M' = (\Re, \mu')$ of ILC. We will show that M' can be extended to a model $M = (D, \mu)$ of LC w.r.t. $\mathcal{IP}$, such that its numerical submodel is exactly M'.
$M = (D, \mu)$ must satisfy the following conditions to be a standard model of LC w.r.t. $\mathcal{IP}$:

1. $D \supseteq \Re$
2. $\forall \mathsf{P} \in \mathcal{IP},\ \mu(\mathsf{P}) = \mu'(\mathsf{P}),\ \mu(\mathsf{P}^-) = \mu'(\mathsf{P}^-)$ and $\mu(\mathsf{P}^+) = \mu'(\mathsf{P}^+)$
3. M must interpret $\mathcal{NIP}$ such that LC is satisfied.

Condition (1) is satisfied by constructing $D = \Re \cup \mathcal{C}$ where $\mathcal{C}$ is the set of all constants that appear in LC, and tuples formed from these constants. The maximum length of a tuple must be the maximum arity of any predicate in $\mathcal{NIP}$.
Next, we construct $\mu(\mathsf{P})$ for $\mu(\mathsf{P}) \in \mathcal{IP}$ by assigning $\mu(\mathsf{P}) = \mu'(\mathsf{P})$.
We construct M to satisfy condition (3) (*i.e.,* $M \models LC$) by considering the dependency graph $G = (V, E)$ of LC (here we assume that the nodes $v_1, \ldots, v_n \in V$ are in topological order):

1. For every node $v \in V$, if $\bar{\mathsf{a}} \in GF\_consts(v)$ then $\bar{\mathsf{a}} \in \mu(v)$.

   For every node $v \in sources(G) \cap \mathcal{IP}$, it is the case that $\mu(v) = \mu'(v)$ because $GF(v) \subseteq ILC$ and $M' \models ILC$.

   If $v(\overline{x}) \in LC$ for predicate $v \in \mathcal{NIP}$, then $\mu(v) = D$.

2. Consider $v_1, \ldots, v_n$ in topological order in G. For each v_i, consider the rules in LC that have v_i at the head, *i.e.,* $R_{v_i} = \{r \mid r \in LC,\ head\_pred(r) = v_i\}$. For each rule r in R_{v_i}, build the model for v_i by looking at the model for predicates in the body. Since we proceed in topological order, $topo(u) < topo(v_i)$ for $u \in body\_preds(r)$, (from Lemma 3.1) and therefore $\mu(u)$ would be defined before using it to define $\mu(v_i)$.
   Let r be the rule: $v(\overline{x}) \leftarrow u_1(\overline{x_1}) \wedge \ldots \wedge u_n(\overline{x_n})$.
   Then, $\bar{\mathsf{a}} \in \mu(v)$ whenever there are consistent substitutions $\theta, \theta_1, \ldots, \theta_n$ such that $\overline{x}\theta = \bar{\mathsf{a}}$, $\overline{x_1}\theta_1 = \overline{\mathsf{b_1}}, \ldots, \overline{x_n}\theta_n = \overline{\mathsf{b_n}}$; and $\overline{\mathsf{b_1}} \in \mu(u_1), \ldots, \overline{\mathsf{b_n}} \in \mu(u_n)$.

We first prove that M is a standard model of LC. By construction of M, we can see that $M \models LC$. Since all interval-predicates are interpreted as intervals, M is also a standard model of LC. Thus condition (3) is satisfied.

We will next show that the numerical submodel of M is exactly M',
i.e., $\forall \mathsf{P} \in \mathcal{IP}$, $\mu(\mathsf{P}) = \mu'(\mathsf{P})$. We prove the above by induction on topological order of nodes in $\mathcal{IP}$ (proof of condition (2)):

1. (*Base step*) $topo(\mathsf{P}) = 1$ means that P must be a source node. For $\mathsf{P} \in sources(G)$, $\mu(\mathsf{P}) = \mu'(\mathsf{P})$ by step (1) of model construction.

2. (*Inductive step*) We assume that $\mu(\mathsf{P}) = \mu'(\mathsf{P})$ is true when $topo(\mathsf{P}) < i$. We show that $\mu(\mathsf{P}) = \mu'(\mathsf{P})$ for $topo(\mathsf{P}) = i$. Also assume that $\mathsf{P} \notin sources(G)$ because that step has already been considered.

   Note that in the model construction process, we never delete an element of μ' but might only add to it to construct $\mu(\mathsf{P})$. To prove the theorem by contradiction, assume that $\mathtt{a} \in \mu(\mathsf{P})$ but $\mathtt{a} \notin \mu'(\mathsf{P})$. ($\mathsf{P}$ is unary; therefore, $\mathtt{a}$ is a single constant rather than a tuple.)

   We apply Lemma 3.4. Since P is not a source, therefore, $\mathtt{a} \notin GF\_consts(\mathsf{P})$. Hence we have that, for some $ir \in IP\_expn(\mathsf{P})$ where ir is the rule

$$\mathsf{P}(x) \leftarrow \mathsf{Q}_1(x_1) \wedge \ldots \wedge \mathsf{Q}_n(x_n)$$

   there are consistent substitutions $\theta, \theta_1, \ldots, \theta_n$ such that
   $x\theta = \mathtt{a}$, $x_1\theta_1 = \mathtt{b}_1$,…,$x_n\theta_n = \mathtt{b}_\mathtt{n}$; and $\mathtt{b}_1 \in \mu(\mathsf{Q}_1)$, …, $\mathtt{b}_\mathtt{n} \in \mu(\mathsf{Q}_n)$.

   Since $topo(\mathsf{Q}_1), \ldots, topo(\mathsf{Q}_n) < i$ (Lemma 3.1), hence, by the inductive statement we have that

$$\mu(\mathsf{Q}_1) = \mu'(\mathsf{Q}_1), \ldots, \mu(\mathsf{Q}_n) = \mu'(\mathsf{Q}_n)$$

   Therefore, we conclude that $\mathtt{b}_1 \in \mu'(\mathsf{Q}_1), \ldots, \mathtt{b}_\mathtt{n} \in \mu'(\mathsf{Q}_n)$.
   But the rule $ir \in ILC$; therefore $M' \models ir$.
   Since $\mathtt{b}_1 \in \mu'(\mathsf{Q}_1), \ldots, \mathtt{b}_\mathtt{n} \in \mu'(\mathsf{Q}_n)$ for the consistent substitutions $\theta_1, \ldots, \theta_n$ in ir, we must have that for the consistent substitution $x\theta = \mathtt{a}$, $\mathtt{a} \in \mu'(\mathsf{P})$. But this contradicts our earlier assumption that $\mathtt{a} \notin \mu'(\mathsf{P})$. Therefore, the original assumption was incorrect and the inductive step must hold.

∎

Lemma 3.4: *If constant* $\bar{\mathtt{a}} \in \mu(v)$, *then either* $\bar{\mathtt{a}} \in GF\_consts(v)$ *or for some rule* $ir \in IP\_expn(v)$ *of the form*

$$v(\overline{x}) \leftarrow \mathsf{Q}_1(x_1) \wedge \ldots \wedge \mathsf{Q}_n(x_n)$$

there exist consistent substitutions $\tau, \tau_1, \ldots, \tau_n$ *such that*

$$\overline{x}\tau = \overline{\mathtt{a}}, x_1\tau_1 = \mathtt{b_1}, \ldots, x_n\tau_n = \mathtt{b_n}, \textit{and}$$

$$\mathtt{b_1} \in \mu(\mathsf{Q}_1), \ldots, \mathtt{b_n} \in \mu(\mathsf{Q}_n)$$

Proof: (By Induction on *topo*(v))

1. (*Base*) If $v \in sources(G)$, then by construction of μ, we know that if $\overline{\mathtt{a}} \in GF\_consts(v)$, then $\overline{\mathtt{a}} \in \mu(v)$.
   If $v \in \mathcal{IP}$, and $v(\overline{\mathtt{a}}) \in LC$, then we have the identity substitution for $v(\overline{\mathtt{a}}) \in IP\_expn(v)$; on the other hand, if $v(\overline{x}) \in LC$, then we have the substitution $\{\overline{\mathtt{a}} \mid \overline{x}\}$ for $v(\overline{x}) \in IP\_expn(v)$.
   If $v \in \mathcal{NIP}$ and $v(\overline{x}) \in LC$, then $\mu(v) = D$ and therefore we have the substitution $\{\overline{\mathtt{a}} \mid \overline{x}\}$ for $v(\overline{x}) \in IP\_expn(v)$.

2. (*Inductive step*) Let the lemma be true if $topo(v) < k$. We prove it for $topo(v) = k$ (where $v \notin sources(G)$).

   Let $\overline{\mathtt{a}} \in \mu(v)$. Then by the model construction process, there is a rule $r \in LC$ of the form
   $$v(\overline{x}) \leftarrow u_1(\overline{x_1}) \wedge \ldots \wedge u_n(\overline{x_n})$$
   with consistent substitutions $\theta, \theta_1, \ldots, \theta_n$ such that $\overline{x}\theta = \overline{\mathtt{a}}$, $\overline{x_1}\theta_1 = \overline{\mathtt{b_1}}$, ..., $\overline{x_n}\theta_n = \overline{\mathtt{b_n}}$; and $\overline{\mathtt{b_1}} \in \mu(u_1)$, ..., $\overline{\mathtt{b_n}} \in \mu(u_n)$.

   Since $topo(u_1), \ldots, topo(u_n) < k$, therefore on applying the inductive step to u_i for all $i = 1 \ldots n$,
   since $\overline{\mathtt{b_i}} \in \mu(u_i)$, either $\overline{\mathtt{b_i}} \in GF\_consts(u_i)$ or for some rule $ur_i \in IP\_expn(u_i)$ of the form
   $$u_i(\overline{x_i}) \leftarrow \mathsf{P}_{i1}(y_{i1}) \wedge \mathsf{P}_{i2}(y_{i2}) \wedge \ldots$$
   there exist consistent substitutions $\theta_i, \varphi_{i1}, \varphi_{i2}, \ldots$ such that, $\overline{x_i}\theta_i = \overline{\mathtt{b_i}}, y_{i1}\varphi_{i1} = \mathtt{c_{i1}}, y_{i2}\varphi_{i2} = \mathtt{c_{i2}}, \ldots$ where $\mathtt{c_{i1}} \in \mu(\mathsf{P}_{i1}), \mathtt{c_{i2}} \in \mu(\mathsf{P}_{i2}), \ldots$.

   Let $\overline{\mathtt{b_i}} \in GF\_consts(u_i)$ for $u_{j+1}, \ldots, u_n$ only. Then, in rule r, we expand $u_{j+1}, \ldots, u_n$ using the ground facts and expand $u_1, \ldots, u_j$ using the ur_i's. Let

vr be the rule obtained by applying substitution $\sigma = \theta_{j+1} \ldots \theta_n$ to r (this will eliminate the ground facts) and then expanding the rest of the body literals using $ur_1, \ldots, ur_j$. The rule vr will be of the form

$$v(\overline{x}\sigma) \leftarrow \ldots \wedge \mathsf{P}_{i1}(y_{i1}\sigma) \wedge \mathsf{P}_{i2}(y_{i2}\sigma) \wedge \ldots \text{ for all } i = 1 \ldots j$$

This rule vr has only interval-predicates in the body; therefore this rule must be in the IP_expn of v, *i.e.,* $vr \in IP\_expn(v)$. Also, we can apply the substitutions $\theta, \ldots, \varphi_{i1}, \varphi_{i2}, \ldots$ for all $i = 1 \ldots j$ to vr (note that these substitutions are consistent), such that $\overline{x}\theta = \overline{\mathsf{a}}$, $y_{i1}\varphi_{i1} = \mathtt{c_{i1}}$, $y_{i2}\varphi_{i2} = \mathtt{c_{i2}}, \ldots$ and
$\mathtt{c_{i1}} \in \mu(\mathsf{P}_{i1}), \mathtt{c_{i2}} \in \mu(\mathsf{P}_{i2}), \ldots$.

But this is exactly what we wanted to prove.

∎

Theorem 3.5: *(*Convert_to_Numerical: Soundness and Completeness*) The set of standard models of ILC given $\mathcal{IP}$ is identical to the set of models of quant_LC.*

Proof: The clauses in ILC must be one of the following types. We will show that for each of them, the conversion in *quant_LC* preserves the sets of models. In the following discussion, $\mathsf{P}, \mathsf{Q}, \ldots$ are interval-predicates, $\mathsf{a}, \mathsf{b}, \ldots$ are constants and $x, y, \ldots$ are variables. Also, $M = (\Re, \mu)$ is a standard model of ILC (*i.e.,* $\forall \mathsf{P} \in \mathcal{IP}, \mu(\mathsf{P}) = \{x \mid \mu(\mathsf{P}^-) \leq x \leq \mu(\mathsf{P}^+), x \in \Re\}$) and $M' = (\Re, \mu')$ is the corresponding model of *quant_LC* which we will show is equal to M. For each $C \in ILC$, its translation in *quant_LC* is denoted by *quant_C*.

1. C is $\mathsf{P}(\mathsf{a})$.
   quant_C is $\mathsf{P}^- \leq \mathsf{a} \leq \mathsf{P}^+$.
   C says that $(\mathsf{a} \in \mu(P))$ which means that $(\mu(\mathsf{P}^-) \leq \mathsf{a} \leq \mu(\mathsf{P}^+))$ whereas
   quant_C says $(\mu'(\mathsf{P}^-) \leq \mathsf{a} \leq \mu'(\mathsf{P}^+))$.
   Therefore $\mu = \mu'$.

2. C is $\neg\mathsf{P}(\mathsf{a})$ or $\leftarrow \mathsf{P}(\mathsf{a})$.
   quant_C is $(\mathsf{a} < \mathsf{P}^-) \vee (\mathsf{a} > \mathsf{P}^+)$.

C says that $(\mathsf{a} \notin \mu(P))$ which means that $(\mathsf{a} < \mu(\mathsf{P}^-)) \vee (\mathsf{a} > \mu(\mathsf{P}^+))$, whereas $quant\_C$ says $(\mathsf{a} < \mu'(\mathsf{P}^-)) \vee (\mathsf{a} > \mu'(\mathsf{P}^+))$.
Therefore, $\mu = \mu'$.

3. C is $\mathsf{P}(x)$.
   $quant\_C$ is $(\mathsf{P}^- = \Leftrightarrow\infty) \wedge (\mathsf{P}^+ = +\infty)$
   C says that $\mu(P) = \Re$ which means that $(\mu(\mathsf{P}^-) = \Leftrightarrow\infty) \wedge (\mu(\mathsf{P}^+) = +\infty)$ whereas
   $quant\_C$ says that $(\mu'(\mathsf{P}^-) = \Leftrightarrow\infty) \wedge (\mu'(\mathsf{P}^+) = +\infty)$.
   Therefore $\mu = \mu'$.

4. C is $\neg\mathsf{P}(x)$ or $\leftarrow \mathsf{P}(x)$.
   $quant\_C$ is $\mathsf{P}^- > \mathsf{P}^+$.
   C says that $\mu(P) = \emptyset$ which means that $\mu(\mathsf{P}^-) > \mu(\mathsf{P}^+)$.
   $quant\_C$ says that $\mu'(\mathsf{P}^-) > \mu'(\mathsf{P}^+)$.
   Therefore $\mu = \mu'$.

5. C is $\leftarrow \mathsf{P}_1(x), \ldots, \mathsf{P}_n(x)$.
   $quant\_C$ is $\vee_{i=1}^{n} \vee_{j=1}^{n} (\mathsf{P}_i^- > \mathsf{P}_j^+)$.
   C says that $\neg\exists x(\mathsf{P}_1(x) \wedge \ldots \wedge \mathsf{P}_n(x))$ which means that $\mu(\mathsf{P}_1) \cap \ldots \cap \mu(\mathsf{P}_n) = \emptyset$, *i.e.*,
   $max(\mu(\mathsf{P_1}^-), \ldots, \mu(\mathsf{P_n}^-)) > min(\mu(\mathsf{P_1}^+), \ldots, \mu(\mathsf{P_n}^+))$ which is the same as $\vee_{i=1}^{n} \vee_{j=1}^{n} (\mu(\mathsf{P}_i^-) > \mu(\mathsf{P}_j^+))$.
   $quant\_C$ says that $\vee_{i=1}^{n} \vee_{j=1}^{n} (\mu'(\mathsf{P}_i^-) > \mu'(\mathsf{P}_j^+))$.
   Therefore $\mu = \mu'$.

6. C is $\mathsf{P}(x) \leftarrow \mathsf{Q}_1(x), \ldots, \mathsf{Q}_n(x)$.
   $quant\_C$ is $(\vee_{i=1}^{n} \mathsf{P}^- \leq \mathsf{Q}_i^-) \wedge (\vee_{i=1}^{n} \mathsf{P}^+ \geq \mathsf{Q}_i^+)$.
   C says that $(\mu(\mathsf{Q}_1) \cap \ldots \cap \mu(\mathsf{Q}_n)) \subseteq \mu(\mathsf{P})$ which means that
   $(\mu(\mathsf{P}^-) \leq max(\mu(\mathsf{Q_1}^-), \ldots, \mu(\mathsf{Q_n}^-))) \wedge (\mu(\mathsf{P}^+) \geq min(\mu(\mathsf{Q_1}^+), \ldots, \mu(\mathsf{Q_n}^+)))$
   i.e., $(\vee_{i=1}^{n} \mu(\mathsf{P}^-) \leq \mu(\mathsf{Q}_i^-)) \wedge (\vee_{i=1}^{n} \mu(\mathsf{P}^+) \geq \mu(\mathsf{Q}_i^+))$.
   $quant\_C$ says that $(\vee_{i=1}^{n} \mu'(\mathsf{P}^-) \leq \mu'(\mathsf{Q}_i^-)) \wedge (\vee_{i=1}^{n} \mu'(\mathsf{P}^+) \geq \mu'(\mathsf{Q}_i^+))$. Therefore $\mu = \mu'$.

All other forms of statements in ILC are combinations of the above basic 6 types (lets call them $B1, \ldots, B6$). The proof is as follows:
Each clause C in ILC can be partitioned into *subclauses* using these rules:

1. A literal with a constant forms a *subclause*, called a *const_subclause*. If the *const_subclause* came from $head(C)$ it is called *head_const_subclause*, else it is called *body_const_subclause*.

2. The largest part of C which has the same variable constitutes a *subclause* called a *var_subclause*. A *var_subclause* can be further distinguished on the basis of whether it includes $head(C)$ or not. If it does, then it is called the *headed_var_subclause*, otherwise it is called the *body_var_subclause*. Since all predicates are unary, the head will be present only in one subclause.

A *head_const_subclause* can be translated into *quant_C* using $B1$ and *body_const_subclause* can be translated using $B2$. A *body_var_subclause* is translated into *quant_C* using $B4$ if it has only one literal, and using $B5$ if it has more than one. A *headed_var_subclause* is translated using $B3$ if it does not have any body. If it has a body then $B6$ is used. The translations for the subclauses are combined by disjunction.

It is easy to see that C is a disjunction of its subclauses (Convert C into a disjunction where head will be the only positive literal. Each subclause is a set of literals with the same variables, or a literal with a constant; hence, C is a disjunction of the subclauses). Since each subclause has one of the 6 forms discussed earlier, hence the translation of each subclause preserves the set of models. Therefore, the disjunction of the subclause-translations will preserve the set of models of C (this set is the union of models for each subclause).

We have shown that for each clause $C \in ILC$, the set of standard models for C is the same as the set of models for the corresponding sentence $quant\_C \in quant\_LC$. Therefore, the set of standard models for ILC will be the intersection of standard models for each clause in ILC and this will be same as the intersection of models for each sentence in $quant\_LC$. Thus, the set of standard models of ILC given $\mathcal{IP}$ is identical to the set of models of $quant\_LC$. ∎

3.3 Complexity

We first discuss the complexity of the problem of answering a query given a set of constraints in the constraint language. Then we will discuss the complexity of the query-answering algorithm that was described in Section 3.1. One kind of query expressible in the query language $\mathcal{QL}$ is $\mathsf{Q(a)}$ (as discussed in Section 2.2). We will show the lower bound for answering this query; hence, this will also be the lower bound for answering the full spectrum of queries allowed in our language.

The complexity of the problem of answering a query is based on a result in deductive databases. Therefore, we first define the problem of expression complexity for nonrecursive Datalog and use this problem to derive our result in Theorem 3.6.

Definition 3: [*Nonrecursive Expression Complexity*] Given a set Σ of Horn rules without functions or recursion, *i.e.,* nonrecursive sentences of the form

$$\forall \overline{x}[R(\overline{y}) \leftarrow P_1(\overline{x_1}) \wedge \ldots \wedge P_n(\overline{x_n})]$$

where the body might also be empty, and a query q of the form

$$R(\overline{\mathsf{c}})$$

where $\overline{\mathsf{c}}$ is a tuple of constants only, determine whether $\Sigma \models q$. ▮

Theorem 3.6: (Interval Query) *Given a set LC of sentences in the logical constraint language $\mathcal{LL}$, a set NC of sentences in the numerical constraint language $\mathcal{NL}$, and a query of the form* (`necessarily` q) *in the query language $\mathcal{QL}$, the problem of answering the query, i.e., determining whether $(LC \cup NC) \models q$ is PSPACE-hard.*

Proof: The nonrecursive expression complexity problem defined in definition 3 is NP-complete if there is only one rule (but multiple ground literals) in the set Σ of Horn clauses [Chandra and Merlin, 1977], and is PSPACE-complete if there are multiple rules [Vardi, 1993].

An instance of the nonrecursive expression complexity problem can be reduced to an instance of the interval query problem in polynomial time by adding the rule

$Q(\mathtt{a}) \leftarrow R(\overline{\mathtt{c}})$ where Q is a new unary interval-predicate, to the set Σ of datalog rules. Then, $LC = \Sigma \cup \{Q(\mathtt{a}) \leftarrow R(\overline{\mathtt{c}})\}$ and $NC = \emptyset$. The query in the interval query problem is (`necessarily` $Q(\mathtt{a})$) or in terms of primitive statements in $\mathcal{QL}$:
(`necessarily` (`and` ($\mathsf{Q}^{-} \leq \mathtt{a}$) ($\mathtt{a} \leq \mathsf{Q}^{+}$))).

A solution to the interval query problem can now be used to construct a solution for the nonrecursive expression complexity problem because $R(\overline{\mathtt{c}})$ is true if and only if $Q(\mathtt{a})$ is. Thus, the interval query problem has the nonrecursive expression complexity problem as a lower bound. Since the nonrecursive expression complexity problem is PSPACE-complete, therefore, the interval query problem is PSPACE-hard. ▮

Since the problem of answering a query in $\mathcal{QL}$, directly from the given constraints in $\mathcal{CL}$ is intractable, we can only expect an exponential time algorithm at best. In particular, our query-answering algorithm (figure 3.1) of first preprocessing the constraints and then answering the queries cannot be better than exponential time. We next carry out a simple analysis for the complexity of the derivation algorithm 3.1 that preprocesses the constraints.

The Derivation Algorithm 3.1 has two procedures: the *Eliminate_NIP* Algorithm 3.2 that takes the set of constraints LC as input and produces another set ILC as output, and the *Convert_to_numerical* Algorithm 3.5 that takes ILC as input and produces *quant_LC* as output. The second procedure is linear in the size of ILC since it requires identifying which of the six possible cases applies and then converting the disjunct accordingly in a constant time step. Thus, *Eliminate_NIP* is the procedure that dominates the complexity of *Derivation* and we will analyze it to determine the complexity of the *Derivation* algorithm.

Theorem 3.7: *(*Derivation Complexity: Lower Bound*) The lower bound for the problem of deriving the set quant_LC from LC is in EXPSPACE.*

Proof: We prove that the problem of deriving *quant_LC* from LC has a lower bound in EXPSPACE by showing an instance where the size of LC is $\mathcal{O}(n)$ and the size of ILC is $\mathcal{O}(2^n)$. Then *quant_LC* will also have a size of $\mathcal{O}(2^n)$. Consider,

$LC = \{\ P_1(x) \leftarrow \mathsf{A}_1(x)$

$P_1(x) \leftarrow \mathsf{B_1}(x)$

$\vdots$

$P_n(x) \leftarrow \mathsf{A_n}(x)$

$P_n(x) \leftarrow \mathsf{B_n}(x)$

$\mathsf{Q}(x) \leftarrow P_1(x) \wedge \ldots \wedge P_n(x)$ }

where Q, A, B are interval-predicates and $P_1, \ldots, P_n$ are noninterval-predicates. Expansion of the predicate Q involves expanding each of $P_1, \ldots, P_n$. For expansion of each P_i, we have two rules to choose from, either using the one with $\mathsf{A_i}$ or the one with $\mathsf{A_i}$, giving us a total of 2^n choices for expanding Q. Therefore, Q has 2^n possible distinct expansions and hence the size of *ILC* is $\mathcal{O}(2^n)$. ∎

Theorem 3.8: *(*Derivation Complexity: Upper Bound*) Derivation algorithm 3.1 that takes the logical constraints LC as input and produces the numerical constraints quant_LC as output, is in DOUBLYEXPSPACE.*

Proof: We carry out a simple analysis of the *Eliminate_NIP* algorithm 3.2 to show that it requires doubly exponential space in the worst case.

Let there be m rules in LC with interval-predicates at the head, and n rules with noninterval-predicates at the head. Let the body of any rule in LC have at most b literals from $\mathcal{NIP}$ and at most t literals from $\mathcal{IP}$. Let a noninterval-predicate occurs at most k times at the head (*i.e.,* there are at most k rules with the same noninterval-predicate at the head); and let there be at most p distinct noninterval-predicates that occur in the head of any rule. Also, let there be l ground atoms.

Then the upper bound on the number of literals in $LC = (b + t) * (m + n) + l$.

We carry out the analysis by using the derivation tree of LC. Remember that the set of expansions generated by the algorithm can be represented by the derivation trees corresponding to them (section 3.2.1) where the leaves of a tree represent the literals in an expansion. We first estimate the size of a single derivation tree to get an estimate on the size of an expansion and then count the total number of derivation trees to estimate the number of possible expansions.

Let us now estimate the size of a derivation tree representing an expansion. Since, only the noninterval-predicates are expanded in any rule, a node in the derivation tree cannot have more than b children. Therefore the maximum branching factor of the tree is b. Going down a path of the tree from the root to a leaf, we cannot encounter any interval-predicates (except the root and the leaf itself) because an interval-predicate is never expanded. Also, since the rules are nonrecursive, no predicate can be repeated on this path. Therefore, the maximum length of a path is the number of distinct noninterval-predicates p. Thus, the size of a derivation tree, or of a single expansion is $\mathcal{O}(b^p)$.

Let us now count the maximum number of derivation trees possible. At each node of the derivation tree, there is a choice of k rules from which to expand the node. Since there are at most b^p nodes, the total number of derivation trees possible is $\mathcal{O}(k^{b^p})$.

Since in the worst case the space requirement of the algorithm is doubly exponential, the *derivation* algorithm is in DOUBLYEXPSPACE. ▮

Theorems 3.7 shows that the lower bound for the complexity of the derivation algorithm is exponential space and Theorem 3.8 shows that the upper bound is doubly exponential space. Despite the worst-case complexity being severe, the query-answering algorithm has been found to be quite acceptable for the following reasons:

1. We can identify the syntactic restrictions on the constraint language $\mathcal{CL}$ that will avoid the exponential blowup. Thus, expressivity can be traded off for efficiency depending upon the application. The sources of exponential blowup in the size of ILC (and therefore, of $quant\_LC$) are

   (a) Repeated occurrence of any $\mathcal{NIP}$ predicate in the body of a clause that is further expanded into another constraint.

   (b) Multiple possible expansions of any $\mathcal{NIP}$ predicate. For example, the predicate $\mathsf{P_i}$ in the proof for theorem 3.7 could be expanded in two ways using either predicate $\mathsf{A_i}$ or $\mathsf{B_i}$.

   If we allow any noninterval-predicate to occur at most once in the body of a clause, and if we allow a noninterval-predicate to occur in the head of at most one clause, then the size of ILC will be polynomial in the size of LC. This can be easily proved by induction on the number of rules.

2. We observe that the algorithm is exponential only in the size of the non-ground rules in the logical constraints, not in the size of the ground literals. The number of non-ground constraints has been found to be small compared to the number of ground literals for many applications.

3. The *derivation* algorithm is part of the preprocessing stage and needs to be invoked only once for all the queries. Hence, the cost is amortized over all the queries. Subsequent to the preprocessing stage, the runtime complexity of answering each query is exponential time. Indeed, we cannot expect better since Theorem 3.9 shows that the problem itself is intractable.

4. Since ILC has only *unary* interval-predicates, we might actually have a tight upper bound of EXPSPACE on the complexity of the *derivation* algorithm. We present the following conjectures without proof:

   (a) The size of each clause in ILC that is generated from LC is polynomial in the size of LC.

   (b) The number of clauses in ILC is at most singly exponential in the size of LC.

We have discussed the complexity the *derivation* algorithm which is the preprocessing stage of the query-answering algorithm. We next discuss the complexity of answering a given query using the derived numerical constraints $quant\_LC$ and the given numerical constraints NC. A basic operation in answering any query in the query language $\mathcal{QL}$ is to check the set $quant\_LC$ for consistency. Therefore, we consider the complexity of the problem of checking the set $quant\_LC$ for consistency in Theorem 3.9

Theorem 3.9 : *(*Combination Complexity*) The problem of checking whether $quant\_LC$ is consistent is NP-complete.*

Proof: The set *quant_LC* has linear arithmetic inequalities in the conjunctive normal form (CNF). Checking for consistency of *quant_LC* is the same as checking whether it is satisfiable. We can show that this problem is in NP. Given an assignment of values to the variables in *quant_LC*, substituting the values in the expression for *quant_LC* and checking whether they satisfy the expression or not can be done in polynomial time. Thus, the problem is in NP.

We show that the problem is NP-hard by reducing the 3SAT problem to the problem of checking whether arithmetic constraints in the CNF are satisfiable. We give the reduction of the 3SAT problem to the arithmetic constraint checking problem below. Since 3SAT is NP-complete [Garey and Johnson, 1979], our problem will be NP-hard too.

Consider an instance of 3SAT. Let $C = \{c_1, \ldots, c_n\}$ be the n clauses, each with exactly 3 literals. Let $x_1, \ldots, x_m$ be the Boolean variables that occur in these clauses. For each variable x_i, substitute an arithmetic constraint with real variable y_i. If the variable occurs as a positive literal, then substitute $y_i \leq 10$[1]; if variable x_i occurs as a negative literal, then substitute $y_i > 10$. The resulting set of clauses is C' and this is an instance of the arithmetic constraint-checking problem. For a solution of this problem, construct a solution to 3SAT as follows. Whenever the value of y_i is less than or equal to 10, assign the value *true* to x_i; and whenever y_i is strictly greater than 10, make x_i *false*. Then, C is satisfiable if and only if C' is.

Since the problem of checking whether *quant_LC* is consistent is in NP and is also NP-hard, it must be NP-complete. ∎

From the complexity discussion we observe that the problem is intractable, but in practice there could be many conditions which make it tractable or solvable in reasonable time. Hence, the use of heuristics is justified to speed up the algorithm. In Chapter 4, we will discuss the domain-independent heuristics that improve the performance of the algorithm significantly. We also discuss the experimentation of the algorithm on two domains to show that the algorithm is pragmatic and that the heuristics are useful.

[1]Any real number can be used instead of 10.

Chapter 4

Answering Queries: Experimentation with Heuristics

The algorithm for answering queries was discussed in Chapter 3. The algorithm was intractable since the problem itself is intractable. Hence, it is important to use heuristics to speed up the running time of the algorithm. This importance was also pointed out in the last chapter while discussing the *combination* algorithm briefly. In this chapter, we first discuss the *combination* algorithm in detail in Section 4.1 together with the necessary linear programming and inequality reasoning procedures. Here, we will also describe the domain-independent heuristics that are useful during the combination of the constraints. In Section 4.2 we describe two domains — a medical domain and a weather domain — to which the query-answering algorithm was applied and discuss the results of the experiments.

4.1 Combining Constraints and Answering Queries using Heuristics

We reproduce the combination algorithm in Chapter 3 here as algorithm 4.1. The first step of the algorithm is to reduce the size of the derived numerical constraint set *quant_LC* using the given numerical constraint set NC. We discuss this pruning heuristic in Section 4.1.1. At the next stage, we convert *quant_LC* to the disjunctive normal form (DNF). The DNF can be pruned further by using a *cover* heuristic to remove redundant disjuncts from *quant_LC*. We discuss this step in Section 4.1.2. The next step is to combine the two sets *quant_LC* and *NC* by adding the set *NC* to each disjunct in the DNF of *quant_LC* to obtain the combined set *output_C*. After the constraints are combined, the final step is to compute the answer to each query over a single disjunct of *output_C*. We discuss the existing methods available in Section 4.1.3.

ALGORITHM 4.1 (Combination)

```
Input: Set of derived disjunctive numerical constraints quant_LC.
       Set of given numerical constraints NC.
Output: Set of combined numerical constraints output_C.
Method:
    Reduce(quant_LC, NC).          % Uses NC to reduce the size of quant_LC
    Cover(quant_LC).               % Convert quant_LC to DNF and
                     % remove redundant disjuncts using the cover method
    Construct output_C by adding NC to each disjunct of quant_LC.        ▮
```

4.1.1 Pruning derived constraint set

The basic idea behind the pruning procedure *Reduce* is that the constraints in *NC* can be used to determine the truth or falsity of some arithmetic inequalities in *quant_LC* which is in the conjunctive normal form (CNF). This helps in removing those inequalities whose truth value is already known. The *reduce* procedure, described as Algorithm 4.2, first computes the lower and upper bounds of all the thresholds in *NC*. For this computation of bounds, any of the existing methods discussed in Section 4.1.3 can be used. These lower and upper bounds are then used to check the truth and falsity of some inequalities.

If an inequality in a disjunctive statement of *quant_LC* is determined to be true, then that complete disjunctive statement can be deleted from *quant_LC* since it is already satisfied. On the other hand, if an inequality is determined to be false, then that disjunct can be deleted.

ALGORITHM 4.2 (Reduce)

```
Input: Set of derived disjunctive numerical constraints $quant\_LC$ in CNF.
       Set of given numerical constraints $NC$.
Output: The reduced constraint set $quant\_LC$.
Method:
    $Bounds \leftarrow Find\_bounds(NC)$.
                        % For all thresholds, finds upper and lower bounds in NC
    Initialize $reduced\_quant\_LC \leftarrow \emptyset$.
    For every $qlc \in quant\_LC$ do
        $qlc' \leftarrow \emptyset$.
        For every disjunct $d \in qlc$ do
            If $True(d, Bounds)$ then
                                % d is already satisfied by bounds from NC
                goto $addconstr$
            Else if not $False(d, Bounds)$ then
                                % d is not inconsistent with bounds from NC
                    $qlc' \leftarrow qlc' \vee d$
        Endfor
        $addconstr$: If $qlc' \neq \emptyset$ then $reduced\_quant\_LC \leftarrow reduced\_quant\_LC \cup qlc'$
    Endfor
    Return($reduced\_quant\_LC$).  ∎
```

We illustrate this algorithm in Example 4.1 by working it out on Example 2.1.

Example 4.1: The set NC from example 2.1 is

$$\begin{aligned} NC = \{&\mathsf{bald}^- = 0,\ 3 \le \mathsf{bald}^+ \le 20000,\ \mathsf{old}^+ = \infty, \\ &60 \le \mathsf{old}^- \le 100,\ 0.1 \le \mathsf{rich}^- \le 1,\ \mathsf{rich}^+ = \infty\ \} \\ &\cup\ \{\mathsf{bald}^- \le \mathsf{bald}^+,\ \mathsf{old}^- \le \mathsf{old}^+,\ \mathsf{rich}^- \le \mathsf{rich}^+\} \end{aligned}$$

The lower and upper bounds for all thresholds are as follows:

$$\begin{aligned} Bounds = \{ & \mathsf{bald}^- \in [0,0], & \mathsf{bald}^+ \in [3, 20000] \\ & \mathsf{old}^- \in [60, 100], & \mathsf{old}^+ \in [\infty, \infty] \\ & \mathsf{rich}^- \in [0.1, 1], & \mathsf{rich}^+ \in [\infty, \infty] \ \} \end{aligned}$$

The set *quant_LC* from example 3.3 is:

$$\begin{aligned} quant\_LC = \{ & (\mathsf{bald}^- \leq 500) \vee (80 < \mathsf{old}^-) \ \vee \ (\mathsf{old}^+ < 80), \\ & (500 \leq \mathsf{bald}^+) \vee (80 < \mathsf{old}^-) \ \vee \ (\mathsf{old}^+ < 80), \\ & (\mathsf{old}^- \leq 80) \vee (6 < \mathsf{rich}^-) \ \vee \ (\mathsf{rich}^+ < 6), \\ & (80 \leq \mathsf{old}^+) \vee \ (6 < \mathsf{rich}^-) \ \vee \ (\mathsf{rich}^+ < 6), \\ & (\mathsf{bald}^- \leq 800) \vee (85 < \mathsf{old}^-) \ \vee \ (\mathsf{old}^+ < 85), \\ & (800 \leq \mathsf{bald}^+) \vee \ (85 < \mathsf{old}^-) \ \vee \ (\mathsf{old}^+ < 85), \\ & (\mathsf{bald}^- \leq 650) \vee (45 < \mathsf{old}^-) \ \vee \ (\mathsf{old}^+ < 45), \\ & (650 \leq \mathsf{bald}^+) \vee \ (45 < \mathsf{old}^-) \ \vee \ (\mathsf{old}^+ < 45) \ \} \end{aligned}$$

Applying the *reduce* procedure to each of the 8 constraints in *quant_LC*: in the first constraint, the disjunct ($\mathsf{bald}^- \leq 500$) is definitely true because $\mathsf{bald}^- \in [0, 0]$; therefore the first constraint is already satisfied. In the second constraint, we cannot say anything about the first two disjuncts for sure, but the third disjunct ($\mathsf{old}^+ < 80$) is definitely false since $\mathsf{old}^+ \in [\infty, \infty]$; therefore, the second constraint is reduced to $(500 \leq \mathsf{bald}^+) \vee (80 < \mathsf{old}^-)$. Applying similar reasoning to the remaining six constraints, we get the following reduced set *quant_LC*:

$$\begin{aligned} quant\_LC = \{ & (500 \leq \mathsf{bald}^+) \vee (80 < \mathsf{old}^-), \\ & (\mathsf{old}^- \leq 80), \\ & (800 \leq \mathsf{bald}^+) \vee \ (85 < \mathsf{old}^-), \\ & (650 \leq \mathsf{bald}^+) \vee \ (45 < \mathsf{old}^-) \ \} \end{aligned}$$

Observe the significant reduction in the size of *quant_LC*; initially it would have had $3^8 = 6561$ disjuncts in the DNF in the worst case. Now, it will have only $2*1*2*2 = 8$ disjuncts in the worst case.

In fact, we can do better. The *reduce* procedure reduces some of the constraints in *quant_LC* such that they are not disjunctive any more. We can add these constraints to NC and reapply the *reduce* procedure to the new *quant_LC*. This can be applied iteratively until all constraints in *quant_LC* are disjunctive. For instance, the second constraint here in *quant_LC* is non-disjunctive. On adding it to NC, we get new bounds for old^-, *i.e.,* $\mathsf{old}^- \in [60, 80]$ and this reduces the set *quant_LC* further:

$$
\begin{aligned}
quant\_LC = \{&(500 \leq \mathsf{bald}^+),\\
&(800 \leq \mathsf{bald}^+)\ \}
\end{aligned}
$$

This new set has no disjunctions so we can add all the constraints here to the set NC. This will give us the new bounds

$$
\begin{aligned}
Bounds = \{\ &\mathsf{bald}^- \in [0, 0], & \mathsf{bald}^+ \in [800, 20000]\\
&\mathsf{old}^- \in [60, 80], & \mathsf{old}^+ \in [\infty, \infty]\\
&\mathsf{rich}^- \in [0.1, 1], & \mathsf{rich}^+ \in [\infty, \infty]\ \}
\end{aligned}
$$

∎

Though we cannot realistically expect all the disjunctions to disappear for all cases, we see that *reduce* is an extremely effective domain-independent heuristic. The effectiveness of this heuristic depends on how tight the bounds of thresholds in NC are. Intuitively, the bounds from NC indicate the amount of uncertainty attached to the value of each threshold. The tighter the bounds specified in NC, the more effective is this heuristic, since more disjuncts can be determined to be definitely true or false.

Regarding the complexity of the *reduce* heuristic, it is only as complex as the procedure to find the bounds of thresholds in NC. As we will see in Section 4.1.3, this procedure is tractable and its complexity depends on the size of NC and the kind of constraints that it has. The rest of the *reduce* procedure is linear in the size of *quant_LC*. Hence, this heuristic is also very efficient. In fact, the performance of the *derivation* algorithm can be improved by applying this heuristic as each constraint of *quant_LC* is generated; then we do not need to store all the constraints of *quant_LC* but can prune its size while it is being generated.

The efficiency and effectiveness of this heuristic is also one reason why we do not want to allow the constraints in NC to be more expressive unless it is important for the application (see comparison to [Meiri, 1991; Kautz and Ladkin, 1991] in Section 2.1.1). Theoretically, the complexity of the *derivation* algorithm is unaffected even if we allow NC to have disjunctions of arithmetic inequalities, since the expressive power of NC will still be as much as that of *quant_LC*. But, then we cannot apply the *reduce* heuristic because the space of NC becomes non-convex and the problem of finding lower and upper bounds becomes as hard as that for *quant_LC*.

4.1.2 Removing redundant disjuncts

In the last section we discussed how the set *quant_LC* was pruned using the constraints in NC. To answer the queries, we need to first convert *quant_LC* into DNF and then combine it with NC to be able to compute the answer over each disjunct of the combined set *output_C*. Many of the disjuncts in the DNF of *quant_LC* are in fact redundant; in this section we discuss how to detect the redundant disjuncts. Not having to compute the answers to the queries over these redundant disjuncts leads to large savings. The domain-independent heuristic that detects the redundant disjuncts is called *cover* since it checks which disjuncts are already covered by others. Example 4.2 illustrates why some disjuncts are redundant:

Example 4.2: Let S be a set with three clauses in CNF

$$(a \vee b \vee c) \wedge (a \vee d \vee e) \wedge (b \vee d)$$

The DNF of the set S is

$$\begin{aligned} (a \wedge a \wedge b) \quad &\vee \quad (a \wedge a \wedge d) \vee \\ (a \wedge d \wedge b) \quad &\vee \quad (a \wedge d \wedge d) \vee \end{aligned}$$

$$\begin{array}{lcl}
(a \wedge e \wedge b) & \vee & (a \wedge e \wedge d) \vee \\
(b \wedge a \wedge b) & \vee & (b \wedge a \wedge d) \vee \\
(b \wedge d \wedge b) & \vee & (b \wedge d \wedge d) \vee \\
(b \wedge e \wedge b) & \vee & (b \wedge e \wedge d) \vee \\
(c \wedge a \wedge b) & \vee & (c \wedge a \wedge d) \vee \\
(c \wedge d \wedge b) & \vee & (c \wedge d \wedge d) \vee \\
(c \wedge e \wedge b) & \vee & (c \wedge e \wedge d)
\end{array}$$

which when simplified has only five disjuncts since all the other disjuncts are redundant:

$$(a \wedge b) \vee (a \wedge d) \vee (b \wedge d) \vee (b \wedge e) \vee (c \wedge d)$$

The reason that some disjuncts in the DNF are redundant is that in the CNF, some conjuncts have a common disjunct. ∎

In *quant_LC* too there are many constraints that have common inequalities. Hence we expect many disjuncts in the DNF to be redundant; detecting and ignoring these disjuncts should be useful. We can show that checking whether a particular disjunct in the DNF of *quant_LC* is redundant or not is equivalent to solving the well-known *set-covering* problem [Cormen *et al.*, 1986, pages 974-978].

Definition 4: [*Set Covering Problem*] An instance $(X, \mathcal{F})$ of the set-covering problem consists of a finite set X and a family $\mathcal{F}$ of subsets of X, such that every element of X belongs to at least one subset in $\mathcal{F}$:

$$X = \bigcup_{S \in \mathcal{F}} S$$

We say that a subset $S \in \mathcal{F}$ *covers* its elements. The problem is to find a minimum-size subset $\mathcal{C} \subseteq \mathcal{F}$ whose members cover all of X:

$$X = \bigcup_{S \in \mathcal{C}} S$$

We say that any $\mathcal{C}$ satisfying this condition *covers* X. ∎

Let us number each constraint in the CNF of *quant_LC* in order. For example 4.2, this numbering will be:

$$\begin{aligned} (a \vee b \vee c) & \quad \cdots \quad (1) \\ (a \vee d \vee e) & \quad \cdots \quad (2) \\ (b \vee d) & \quad \cdots \quad (3) \end{aligned}$$

For each literal in *quant_LC*, compute the set of constraints in which it occurs. For the above example, it is:

$$\begin{aligned} S(a) &= \{1,2\} \\ S(b) &= \{1,3\} \\ S(c) &= \{1\} \\ S(d) &= \{2,3\} \\ S(e) &= \{2\} \end{aligned}$$

Then the finite set X is the set of constraints of *quant_LC*, *i.e.,*

$$X = \{(a \vee b \vee c) \wedge (a \vee d \vee e) \wedge (b \vee d)\}$$

in this case, or referring to the constraints by their ordering number, $X = \{1, 2, 3\}$. The family $\mathcal{F}$ of subsets of X is then

$$\mathcal{F} = S(a) \cup S(b) \cup S(c) \cup S(d) \cup S(e)$$

The problem then is to find *all* minimum-size subsets $\mathcal{C} \subseteq \mathcal{F}$ that cover X. The DNF of *quant_LC* is the disjunction of all such minimum covers. In this example, we have five minimum covers: $\{S(a), S(b)\}$, $\{S(a), S(d)\}$, $\{S(b), S(d)\}$, $\{S(b), S(e)\}$ and $\{S(c), S(d)\}$.

If we are given any disjunct from the DNF of *quant_LC*, such as $(a \wedge e \wedge b)$, then we can check whether it is redundant or not by checking whether it is a minimum cover or not. It is redundant if and only if it is not a minimum cover.

Unfortunately, the *set-covering* problem is known to be NP-hard. Hence, any algorithm for checking for redundancy of disjuncts in $quant\_LC$ will take at least exponential time. Instead of using an exponential heuristic to speed up an exponential algorithm, we use a greedy approximation algorithm to the set-covering problem [Cormen *et al.*, 1986]. The greedy set-cover algorithm runs in time polynomial in $|X|$ and $|\mathcal{F}|$ and has a logarithmic ratio bound of $(ln|X|+1)$ which is the maximum ratio of the size of an approximate answer to the optimal answer.

We describe the *cover* heuristic using the greedy algorithm in Algorithm 4.3 and use Example 4.3 to illustrate how it works.

ALGORITHM 4.3 (Cover)

Input: Set $quant\_LC$ in CNF with n disjunctive constraints and k distinct literals.
Output: Set $quant\_LC$ in DNF after removing redundant disjuncts.
Method:

```
    Number the constraints in $quant\_LC$ from $1, \ldots, n$.
    Initialize $X = \{1, \ldots, n\}$.
    Construct $\mathcal{F}$ as a family of $k$ subsets of $X$ where each subset corresponds
                to the set of constraints in which each literal occurs in $quant\_LC$.
    Let $literal\_mapping$ be the mapping of each literal
        to its corresponding subset in $\mathcal{F}$.
    Generate each disjunct $dc$ in DNF of $quant\_LC$ in order.
                                  % Note that dc will be a set of literals. It is also a
                             % subset of $\mathcal{F}$ and covers X through the $literal\_mapping$.
    For each such disjunct $dc$, check whether it is redundant using
        the procedure Redundant?$(X, \mathcal{F}, dc, literal\_mapping)$.
    Build the DNF of $quant\_LC$ using only non-redundant disjuncts.                ∎
```

ALGORITHM 4.4 (Redundant?)

```
Input: Finite set $X$.
       Family $\mathcal{F}$ of subsets of $X$.
       $literal\_mapping$ of literals to a subset in $\mathcal{F}$.
       Set $dc$ which is a set of literals. It is also a subset of $\mathcal{F}$
           and covers $X$ through the $literal\_mapping$.
Output: $yes$ or $no$.            % Corresponds to whether dc is redundant or not.
% The algorithm checks whether there is a proper subset of dc that also covers X.
Method:
    Initialize $U \leftarrow X$.
    Initialize $\mathcal{C} \leftarrow \emptyset$.
    while $U \neq \emptyset$
            do select an $S \in dc$ that maximizes $|S \cap U|$
               $U \leftarrow U \Leftrightarrow S$
               $\mathcal{C} \leftarrow \mathcal{C} \cup \{S\}$
    If $dc \subseteq \mathcal{C}$ then return($no$)
       else return($yes$)                                              ∎
```

Example 4.3: The constraints of *quant_LC* were numbered in example 4.2. The set X is

$$X = \{1, 2, 3\}$$

The *literal_mapping* for each literal of *quant_LC* is

$$\begin{aligned} S(a) &= \{1,2\} \\ S(b) &= \{1,3\} \\ S(c) &= \{1\} \\ S(d) &= \{2,3\} \\ S(e) &= \{2\} \end{aligned}$$

The family of subsets $\mathcal{F}$ is the set of all these sets. For each of the 18 disjunct in DNF of *quant_LC*, we check whether it is redundant or not. Consider dc to be $(a \wedge e \wedge b)$

for instance. The subset of $\mathcal{F}$ that corresponds to dc is then the set $S(a) \cup S(e) \cup S(b)$, *i.e.,*

$$dc = \{\{1,2\},\{2\},\{1,3\}\}$$

On applying procedure *redundant?*, we could get either $\mathcal{C} = S(a) \cup S(b)$ or $\mathcal{C} = S(e) \cup S(b)$. Since dc is not a proper subset of either of these minimal covers, we find that $(a \wedge e \wedge b)$ is redundant and hence can be deleted. ∎

Thus, performing this fast *cover* check on each disjunct enables us to delete a number of redundant ones. Despite these heuristics, the size of a DNF might be large, in which case we might not want to store the DNF but generate each disjunct every time we want to compute the answer to a query. There is the usual time vs. space tradeoff and depending upon the application — size of *quant_LC* in CNF, available storage and the desired response time for each query, the user can decide to either precompute the DNF and store it, or to generate it every time. In the former case, we might actually want to use an exact (though exponential) algorithm to generate the minimal DNF; whereas, in the latter case it is better to use a fast and approximate method to eliminate many but not all redundant disjuncts.

An interesting observation is that for many queries it is not necessary to check all the disjuncts in the pruned DNF form. For instance, a query whether a constraint is *possibly* true or not, has to find any one disjunct over which the constraint is satisfied to return a true answer; and for a query whether a constraint is *necessarily* true or not, one has to find any one disjunct where the constraint is violated to return a false answer. Furthermore, even for queries where all disjuncts have to be checked, an approximate answer can be obtained by computing only on a few disjuncts. For instance, a query to find the minimum value of a threshold can return the minimum over only a few disjuncts. This approximate answer is still useful since it supplies a lower bound on the threshold, even though not the tightest lower bound. Thus, this procedure gives a useful approximate answer any time that an answer is required, and the approximation gets closer to the optimal as the allowed time for computing the answer to the query increases. It is also possible to use heuristics depending upon the structure of *quant_LC*, to check the disjuncts in an order such that the likelihood of encountering a complete or optimal answer earlier goes up.

4.1.3 Answering for each disjunct

We have discussed previously how the set *quant_LC* is pruned *a priori* to eliminate redundant disjuncts and how the disjuncts of *output_C* are generated. Each disjunct thus generated is a set of linear arithmetic constraints. We now discuss how any query is answered on such a single disjunct, say D, that is a conjunction of linear arithmetic inequalities[1]. We first discuss the queries for checking a constraint for implication and consistency (queries 1 and 2 in Section 2.2) and then the queries for maximum and minimum values of thresholds (queries 3 and 4 in Section 2.2).

Queries for checking implication and consistency

We discuss how queries of the form (`necessarily` q) or (`possibly` q) can be reduced to checking for consistency of a set of linear inequalities. We discuss it for atomic queries as well as for queries that have conjunctions and disjunctions. Note that an atomic query is of the form (*thresh relop const*) where *thresh* is a threshold symbol, *relop* is one of the relational operators among $\{\leq, \geq, <, >, =, \neq\}$, and *const* is a numerical constant. The negation of an atomic query will also be a simple linear inequality of the form (*thresh inverse-relop const*) where *inverse-relop* is the inverse of the relational operator in q. The inverse of $\{\leq, \geq, <, >, =, \neq\}$ are in order $\{>, <, \geq, \leq, \neq, =\}$ and are themselves all relational operators.

A query of the form (`necessarily` q) means that we need to test whether $output\_C \models q$ holds. We perform this test by checking for each disjunct D of *output_C* whether $(D \wedge \neg q)$ is consistent or not. If it is consistent then $D \not\models q$ and we return a *no* answer for that disjunct; on the other hand, if it is inconsistent then $D \models q$ holds and therefore we return a *yes* answer for that disjunct. If the answer to the query is *yes* in every disjunct of *output_C* then the answer is *yes* for *output_C*; if the answer is *no* for any one disjunct then the answer is *no* for *output_C*. (This follows from the fact that $(a \vee b) \Rightarrow q \equiv (a \Rightarrow q) \wedge (b \Rightarrow q)$ and therefore the query q has to be true

[1]We use the terms "set of inequalities" and "conjunction of inequalities" interchangeably.

over every disjunct to be true over the whole disjunctive statement.) When the query itself has disjunctions or conjunctions then we use the procedure of algorithm 4.5 to break down the query into the simple procedure of checking a set of inequalities for consistency.

ALGORITHM 4.5 (Necessarily-Query)

```
Input: Set of inequalities $output\_C$ in DNF.
       Query (necessarily $q$).
Output: $yes$ or $no$ answer to the query
            according to whether $output\_C \models q$ or not.
Method:
1. If $q$ is atomic of the form ($thresh\ relop\ const$) then
   For every disjunct $D$ of $output\_C$
   do check whether $consistent(D \wedge \neg q)$.
      If $consistent$ then return($no$) and exit the algorithm.
   Endfor
   return($yes$).
      % because $(D \wedge \neg q)$ is inconsistent for every disjunct, therefore $output\_C \models q$
2. If $q$ is of the form $q_1 \wedge q_2$ where $q_1, q_2$ are atomic then
   For every disjunct $D$ of $output\_C$
   do check whether $consistent(D \wedge \neg q_1)$.
      If $consistent$ then return($no$) and exit the algorithm.
      Else check whether $consistent(D \wedge \neg q_2)$.
         If $consistent$ then return($no$) and exit the algorithm.
   Endfor
   return($yes$).
      % the logic follows from the fact that $D \Rightarrow (q_1 \wedge q_2) \equiv (D \Rightarrow q_1) \wedge (D \Rightarrow q_2)$
3. If $q$ is of the form $q_1 \vee q_2$ where $q_1, q_2$ are atomic then
   For every disjunct $D$ of $output\_C$
   do check whether $consistent(D \wedge \neg q_1 \wedge \neg q_2)$.
      If $consistent$ then return($no$) and exit the algorithm.
   Endfor
   return(yes).
                    % the logic follows from the fact that $\neg(q_1 \vee q_2) \equiv (\neg q_1 \wedge \neg q_2)$
4. If $q$ has conjunctions as well as disjunctions then
   Convert $q$ to CNF, i.e., of the form $(q_3 \vee q_4) \wedge (q_5 \vee q_6)$
   where $q_3, q_4, q_5, q_6$ are atomic
   Use step 2 by substituting for $q_1, q_2$ as
       $q_1 \leftarrow (q_3 \vee q_4)$
       $q_1 \leftarrow (q_5 \vee q_6)$
   and use step 3 for checking consistency of a disjunct $D$ with
       $\neg(q_3 \vee q_4)$ and $\neg(q_5 \vee q_6)$                                   ∎
```

A query of the form (`possibly` q) means that we need to test whether $(output\_C \wedge q)$ holds. We perform this test by checking for each disjunct D of *output_C* whether $(D \wedge q)$ is consistent or not. If it is consistent then we return a *yes* answer for that disjunct, else we return a *no* answer for that disjunct. If the answer to the query is *no* in every disjunct of *output_C* then the answer is *no* for *output_C*; if the answer is *yes* for any one disjunct then the answer is *yes* for *output_C*. (This follows from the fact that $(a \vee b) \wedge q \equiv (a \wedge q) \vee (b \wedge q)$ and therefore the query q has to be true over some disjunct to be true over the whole disjunctive statement.) When the query itself has disjunctions or conjunctions then we use the procedure of algorithm 4.6 to break down the query into the simple procedure of checking a set of inequalities for consistency.

ALGORITHM 4.6 (Possibly-Query)

```
Input: Set of inequalities output_C in DNF.
       Query (possibly $q$).
Output: yes or no answer to the query according to
        whether $output\_C \wedge q$ is consistent or not.
Method:
1. If $q$ is atomic, then
   For every disjunct $D$ of output_C
   do check whether $consistent(D \wedge q)$.
```

```
        **If** *consistent* **then return**(*yes*) and exit the algorithm.
    **Endfor**
    **return**(*no*).
    % *Because* $(D \wedge q)$ *is inconsistent for every disjunct, therefore* $output\_C \wedge q$ *is too*
2. **If** $q$ is of the form $q_1 \wedge q_2$ where $q_1, q_2$ are atomic **then**
    **For every** disjunct $D$ of $output\_C$
    **do** check whether $consistent(D \wedge q_1)$.
        **If** *consistent* **then**
            check whether $consistent(D \wedge q_2)$.
            **If** *consistent* **then return**(*yes*) and exit the algorithm.
    **Endfor**
    **return**(*no*).
3. **If** $q$ is of the form $q_1 \vee q_2$ where $q_1, q_2$ are atomic **then**
    **For every** disjunct $D$ of $output\_C$
    **do** check whether $consistent(D \wedge q_1)$.
        **If** *consistent* **then return**(*yes*) and exit the algorithm.
        **Else** check whether $consistent(D \wedge q_2)$.
                **If** *consistent* **then return**(*yes*) and exit the algorithm.
    **Endfor**
    **return**(*no*).
            % *the logic follows from the fact that* $D \wedge (q_1 \vee q_2) \equiv (D \wedge q_1) \vee (D \wedge q_2)$
4. **If** $q$ has conjunctions as well as disjunctions **then**
    Convert it to DNF, *i.e.,* of the form $(q_3 \wedge q_4) \vee (q_5 \wedge q_6)$
    where $q_3, q_4, q_5, q_6$ are atomic
    Use step 3 by substituting for $q_1, q_2$ as
        $q_1 \leftarrow (q_3 \wedge q_4)$
        $q_1 \leftarrow (q_5 \wedge q_6)$
    and use step 2 for checking consistency of a disjunct $D$ with
        $(q_3 \vee q_4)$ and $(q_5 \vee q_6)$                                      ∎
```

The above algorithms illustrate how a compound query can be reduced to operations on atomic queries, where an atomic query is one on a single linear inequality without any disjunctions. The basic operation that is required for these procedures is checking a set of inequalities for consistency. A consistency check is part of any algorithm that optimizes a linear function over a set of linear inequalities. We will discuss these procedures next as part of the discussion for computing extreme threshold values.

Queries for extreme threshold values

The queries of the form (`minimum P`$^-$) or (`maximum P`$^-$) where `P`$^-$ is any threshold require the computation of lower and upper bounds of thresholds over the set *output_C* of arithmetic constraints in DNF. We can achieve this by computing the lower or the upper bound for each disjunct D of *output_C* and then computing the global minimum or maximum. The basic operation here is to compute the minimum and maximum values of a threshold over a set of linear inequalities (without disjunction).

Computation of the extreme values of thresholds as well as checking the consistency of a set of inequalities can be done using one of the many existing methods[2]. The appropriate method depends upon the kind of constraints present in NC because the constraints in *quant_LC* either have inequalities with only one variable or inequalities with simple order relations (*i.e.,* of the form $x \leq y$, $x \geq y$, $x = y$, $x < y$ or $x > y$).

If NC has only simple order relations or bounded differences (these are inequalities of the form $x \Leftrightarrow y \leq \mathsf{a}$) then we can use an efficient $\mathcal{O}(n^3)$ procedure (where n is the number of variables) from [Davis, 1987], [Meiri, 1991] or Sacks' *bounder* [Sacks, 1987]. For more general linear constraints, we have to use a linear programming method that is still tractable $\mathcal{O}(n^{3.5}L)$ (L is size of input) [Karmarkar, 1984; Khachiyan, 1979; Megiddo, 1983]. In practice, the simplex algorithm would be preferable because despite its theoretical worst case running time being exponential, it is efficient for most practical problems. [Schrijver, 1986] has a good overview of the different linear programming methods. Lassez's work on canonical form of generalized linear constraints [Lassez and McAloon, 1992] has potential applications, though the

[2]The thresholds will be treated as the variables that have to be solved for, in the following discussion.

advantage of a canonical form would be apparent only if all the sets of inequalities are stored and a number of queries are asked on the canonical form of this set.

We use an algorithm for inequalities with one variable or simple order relations based on the constraint propagation technique in [Davis, 1987]. This algorithm builds a graph where there is a node in the graph corresponding to each threshold and there is an edge (x, y) corresponding to the inequality $x \leq y$. The algorithm keeps track of whether the inequality was strict or not and also of the lower and upper bounds of each threshold. The bounds are propagated through the graph, to get the tightest lower and upper bounds.

4.2 Experimentation

In this section, we discuss the application of the query-answering algorithm to two domains — a medical domain and a weather domain. For each of these, we describe the constraints that were the input to the algorithm and discuss the computation of answers to various queries. The experiments demonstrate that despite the severe worst case complexity, the algorithm is quite efficient in practice. They also illustrate the usefulness of heuristics in speeding up the runtime of the algorithm.

The implementation of the program is done in Common Lisp using Lucid Lisp on a DEC 3100 workstation. The program does not generate the clauses in a batch and then convert them, but rather, each clause that is generated by expansion is immediately converted to a numerical constraint and the *reduce* heuristic is applied to check whether that constraint should be pruned or not. This saves space since the pruned constraints need not be stored at any time. The *reduce* heuristic is applied iteratively until all the constraints left in $quant\_LC$ are disjunctive. Also, the disjuncts in the DNF of $quant\_LC$ are generated at runtime for each query; therefore, the *cover* heuristic is applied at runtime for each query. The greedy set-covering method is used for *cover*. In both our applications, the numerical constraints NC have inequalities with one variable or order relations. Therefore, we use an efficient $\mathcal{O}(n^3)$ procedure for getting lower and upper bounds of thresholds.

4.2.1 Medical Domain

This application captures some of the rules governing the cardiovascular system. The constraints and data were obtained in consultation with the members of the Guardian project at Stanford. The rules relate five parameters in the human cardiovascular system: cardiac output (CO) in liters/minute, stroke volume (SV), heart rate (HR) in beats/minute, pulmonary capillary wedge pressure (PCWP) in mmHg and systemic vascular resistance (SVR) in dynes sec/cm5. Of these, all except the stroke volume (SV) can be directly measured. The ground data is therefore available for the other four parameters. There are ranges defined over these four parameters that are used by doctors (and hence by any intelligent medical reasoning system), such as "normal-CO", "low-HR" *etc.* The ranges of these parameters are also related to other conditions that are not defined over any parameters.

The inputs to the algorithm are described in Appendix B. The inputs include the set of interval-predicates $\mathcal{IP}$ which are the ranges over the four measurable parameters, the set of noninterval-predicates $\mathcal{NIP}$ which are the conditions not defined over any parameters, the set of logical constraints LC and the set of numerical constraints NC. Some features of this domain are:

Number of interval-predicates, $\mid \mathcal{IP} \mid$	$= 17$
Number of noninterval-predicates, $\mid \mathcal{NIP} \mid$	$= 12$
Number of logical rules, $\lvert RS \rvert$	$= 24$
Number of ground literals, $\lvert GF \rvert$	$= 32$
Number of numerical constraints, $\lvert NC \rvert$	$= 48$
Number of rules with interval-pred at head, m	$= 12$
Number of rules with noninterval-pred at head, n	$= 12$
Maximum no. of $\mathcal{NIP}$ literals in any rule-body, b	$= 3$
Maximum no. of $\mathcal{IP}$ literals in any rule-body, t	$= 2$
Maximum no. of literals in any rule-body	$= 4$
Maximum repetitions of $\mathcal{NIP}$ pred in rule-head, k	$= 3$
Number of distinct $\mathcal{NIP}$ preds at head, p	$= 8$

Maximum arity of a predicate	= 2

The worst case upper bound on the size of *quant_LC* according to the analysis in Theorem 3.8 is $\mathcal{O}(k^{b^p})$ which is $3^{3^8} = 3^{6561}$ in this case. Below we display the times taken for various operations in spite of this severe upper bound and the sizes of the generated sets.

Time for derivation of *quant_LC* from *LC*	= 1.00 second
No. of constraints in CNF of *quant_LC*	= 416
No. of constraints pruned by *reduce*	= 407
No. of constraints added by *reduce* to *NC*	= 4
No. of disjunctive constraints in *quant_LC*	= 9
No. of disjuncts in DNF of *output_C*	= 2592
No. of disjuncts found to be redundant by *cover*	= 2408

Note the large number of constraints that are pruned by the *reduce* and *cover* heuristics. Since each constraint in *quant_LC* has 2 or 3 disjuncts, if *reduce* heuristic were not applied then the size of *quant_LC* would have been 3^{416} in the worst case instead of 2592. If the *cover* heuristic were not applied, then computing the answer to each query would require computing it over 2592 disjuncts. By applying *cover*, 2408 of these disjuncts are found to be redundant and hence the answer to any query needs to be computed only over 184 disjuncts. We next display the times taken in seconds by some queries in Table 4.1. The first column displays the time taken in seconds to answer a query if the *cover* heuristic is not applied. The third column displays the time if the heuristic is applied. Note the significant time saving here. The fifth column displays the answer returned for the query. The second and the fourth columns display the number of disjuncts that had to be checked before the disjunct with the right answer was encountered. (The total number of disjuncts checked, of course, depends on the query and the answer.) For query 1 with the *cover* heuristic for instance, 10 disjuncts returned an answer which was not the global minimum but the 11th disjunct returned the global minimum; all the 184 disjuncts had to be

Query No.	Query
1	(`minimum` low_CO$^-$)
2	(`maximum` low_CO$^-$)
3	(`possibly` ($\leq$ low_CO$^-$ 2.1)
4	(`necessarily` ($\leq$ low_CO$^-$ 2.9)

	W/o Cover		Cover		Answer
	Time (in secs)	disj (max 2592)	Time (in secs)	disj (max 184)	
1	39.12	161	1.47	11	2.0
2	29.14	1945	4.21	121	2.8
3	1.86	161	0.11	11	yes
4	31.16	2592	2.08	184	yes

Table 4.1: Queries in Medical Domain

checked before the global minimum was determined though. This figure indicates that if we were seeking an approximate answer to the query by checking only a few disjuncts, then how likely we are to get the correct answer.

4.2.2 Weather Domain

This application captures some of the rules governing the weather. The constraints were obtained from books on weather and data was obtained from the weather logs of the Palo Alto weather station. The rules relate four parameters that are directly measured: mean-temperature (in Fahrenheit), precipitation (in 1/100 inch), relative-humidity and smog-index . There are ranges defined over these four parameters such as "hot", "low-smog *etc.* The ranges of these parameters are also related to other parameters that are not measured numerically; such as "cloudy-sky", "low-perceived-humidity" and so on.

The inputs to the algorithm are described in Appendix C. The inputs include the set of interval-predicates $\mathcal{IP}$, the set of noninterval-predicates $\mathcal{NIP}$, the set of logical constraints LC and the set of numerical constraints NC. Some features of this domain are:

Number of interval-predicates, $\mid \mathcal{IP} \mid$	$= 16$
Number of noninterval-predicates, $\mid \mathcal{NIP} \mid$	$= 13$
Number of logical rules, $\|RS\|$	$= 37$
Number of ground literals, $\|GF\|$	$= 77$
Number of numerical constraints, $\|NC\|$	$= 46$
Number of rules with interval-pred at head, m	$= 15$
Number of rules with noninterval-pred at head, n	$= 19$
Maximum no. of $\mathcal{NIP}$ literals in any rule-body, b	$= 3$
Maximum no. of $\mathcal{IP}$ literals in any rule-body, t	$= 2$
Maximum no. of literals in any rule-body	$= 5$
Maximum repetitions of $\mathcal{NIP}$ pred in rule-head, k	$= 6$
Number of distinct $\mathcal{NIP}$ preds at head, p	$= 5$
Maximum arity of a predicate	$= 2$

The worst case upper bound on the size of *quant_LC* according to the analysis in Theorem 3.8 is $\mathcal{O}(k^{b^p})$ which is $6^{3^5} = 6^{243} \approx 10^{189}$ in this case. Below we display the times taken for various operations in spite of this severe upper bound and the sizes of the generated sets.

Time for derivation of *quant_LC* from *LC*	$= 1.15$ seconds
No. of constraints in CNF of *quant_LC*	$= 360$
No. of constraints pruned by *reduce*	$= 360$
No. of constraints added by *reduce* to *NC*	$= 9$
No. of disjunctive constraints in *quant_LC*	$= 0$
No. of disjuncts in DNF of *output_C*	$= 1$

Note the large number of constraints that are pruned by the *reduce* and *cover* heuristics in this application too. The interesting point to note here is that *reduce* was effective in doing away with all disjunctive constraints. This was so because the bounds provided by *NC* were tight enough to enable the determination of the truth values of all disjuncts in CNF of $quant\_LC$. There were 9 constraints generated from $quant\_LC$ that had no disjunctions and hence were added to NC. Therefore, $output\_C$ has only conjunctions of linear inequalities and computing the answers to queries is really efficient. (Hence, we do not display the times.)

Chapter 5

Extensions to the Constraint Language

In this chapter, we extend the constraint language $\mathcal{CL}$ in several ways and discuss the ramifications of these changes on the query-answering algorithm. We describe the algorithms for the extended languages and formally establish the scope of these procedures.

We consider the extension of the language for logical constraints $\mathcal{LL}$ to the case where goal clauses (*i.e.,* Horn clauses with empty head) are allowed in Section 5.1. In Section 5.2 we consider the case where the logical constraint language is extended to allow certain types of arithmetic inequalities. Next, in Section 5.3 we allow the rules in the logical constraint language to be recursive. In Section 5.4, we relax the restriction of Horn clauses and allow $\mathcal{LL}$ to have clauses with more than one positive literal. We impose a condition called the *single-noninterval-in-head* on these clauses. In Section 5.5, we do not change the logical constraint language but instead relax the restriction that interval-predicates be unary as long as the interpretation of the n-ary predicate is a rectangular box in $\Re^n$. In Section 5.6, the constraint language remains the same and the interval-predicates are unary, but we consider the case where the interval-predicates are interpreted as a finite union of disjoint intervals instead of the single interval interpretation we had earlier.

A point to remember in this chapter is that the algorithms are meant to indicate how the extensions might be handled to still obtain sound and complete derivations; we do not try to give the most efficient algorithms and we do not discuss how these algorithms can be optimized to make them more efficient. It is also possible to trade either soundness or completeness for efficiency and we indicate that in some places.

5.1 Goal Clause

We extend the logical constraint language $\mathcal{LL}$ to allow goal clauses in the language, while still keeping the other restrictions of no functions or recursion in the language. A goal clause is a clause that has no positive literals; alternatively, it is a rule with an empty head:

$$\forall \overline{x}[\leftarrow P_1(\overline{x_1}) \wedge \ldots \wedge P_n(\overline{x_n})]$$

We will denote this augmented logical constraint language by $\mathcal{LL}^{goal}$ and the constraint language that allows constraints from $\mathcal{LL}^{goal}$ and from $\mathcal{NL}$ by $\mathcal{CL}^{goal}$. All the sets of constraints and algorithms that are changed by this language extension will be indicated by a *goal* superscript.

The algorithm for query-answering requires some change to account for goal clauses. Algorithm 5.1 is the updated *Eliminate_NIP* algorithm and Algorithm 5.2 is the updated *Expand* algorithm. All the other procedures remain unchanged.

ALGORITHM 5.1 (Eliminate_NIP^{goal})

```
Input: Set of logical constraints $LC^{goal}$.
       Set of interval-predicates $\mathcal{IP}$.
       Set of noninterval-predicates $\mathcal{NIP}$.
Output: $ILC$ the set of clauses derived from $LC$
             such that it has only interval-predicates
Method:
    Initialize $ILC \leftarrow \emptyset$.
    For every clause $c \in LC^{goal}$ such that $head(c) \in \mathcal{IP}$
             or $head(c)$ is empty do
        $ILC_c \leftarrow Expand(c, LC^{goal}, \mathcal{IP}, \mathcal{NIP})$.
        $ILC \leftarrow ILC \cup ILC_c$.
    **Endfor**
    **Return**($ILC$) ∎
```

The only difference between the $Eliminate\_NIP^{goal}$ algorithm and the original *Eliminate_NIP* algorithm is that now we need to expand the goal clauses too. The expansion algorithm is also slightly different to account for the goal clauses.

ALGORITHM 5.2 (Expand^{goal})

Input: Clause c from the set of logical constraints LC^{goal} such that its head is either empty or has an interval-predicate.
The set of logical constraints LC^{goal}.
The set of interval-predicates $\mathcal{IP}$.
The set of interval-predicates $\mathcal{NIP}$).

Output: The set of clauses obtained by expanding c using clauses in LC^{goal} such that only noninterval-predicates are expanded.

Method:

```
    If $c$ is empty then $LC^{goal}$ is inconsistent.            % c is empty if
    If body(c) is empty or                  % both head(c) and body(c) are empty
      for every literal $l \in body(c)$ it is case that $l \in \mathcal{IP}$
    Then return($\{c\}$)
    Else
        Initialize $S \leftarrow \emptyset$.
        For every literal $l \in body(c)$ such that $l \in \mathcal{NIP}$ do
          For every clause $r \in LC$ such that $unifiable(l, head(r))$ do
                                                   % notation for unification from
            $\theta \leftarrow mgu(l, head(r))$.       % [Genesereth and Nilsson, 1987, Section4.2]
            $new\_c \leftarrow [head(c) \leftarrow (body(c) \Leftrightarrow \{l\}) \cup body(r)]\theta$.
            $S \leftarrow S \cup Expand(new\_c, LC, \mathcal{IP}, \mathcal{NIP})$.
            Endfor
        Endif
    Return($S$) ∎
```

The only difference between the $Expand^{goal}$ algorithm and the original *Expand* algorithm is the check for the consistency of LC^{goal}. The set LC could never be inconsistent since it had only definite Horn rules, but due to the presence of goal clauses, LC^{goal} can be inconsistent. Consider the goal clause

$$\forall \overline{x}[\leftarrow P_1(\overline{x_1}) \wedge \ldots \wedge P_n(\overline{x_n})]$$

where $P_1, \ldots, P_n \in \mathcal{NIP}$. It is inconsistent if the literals $P_1(\overline{x_1}), \ldots, P_n(\overline{x_n})$ can all be simultaneously satisfied. The $Expand^{goal}$ checks for such an inconsistency by expanding all the noninterval-predicates of a goal clause; if this leads to an empty body, it means that the body of the clause is satisfiable and hence the set of clauses LC^{goal} is inconsistent[1].

In Theorem 5.1 we show that the $Derivation^{goal}$ algorithm is sound and complete. Wherever necessary, we add the superscript *goal* to terms to indicate that their definition should change to account for goal clauses. We do not give the new definitions since it is obvious in most cases. Note that the dependency graph should have an *empty* node and that a rule-edge corresponding to a goal clause will point to the empty node. Also, since there are no edges going out of the empty node, it will be a sink in the graph and can have the highest topological order assigned to it.

Theorem 5.1: *The class of numerical submodels of LC^{goal} w.r.t. $\mathcal{IP}$ is identical to the class of models over $\Re$ of $quant\_LC^{goal}$.*

Proof: Theorem 3.5 is true for goal clauses since its proof did not require a clause to be a definite clause. So we extend the proof of Theorem 3.3 to account for goal clauses.

The construction of model M from M' is carried out in exactly the same way. The interpretations of all predicates are unaffected by the goal clause when the model is being constructed since a goal clause can always have the highest *topo* value in some

[1]If a goal clause with interval-predicates causes an inconsistency, it will not be detected at this stage but rather at a later stage when the numerical constraints are combined and inequality reasoners or linear programming techniques are used.

topological order and hence is encountered last. Also, since the head of this clause is the empty node, there is no interpretation constructed for this node.

To prove that M is still a model of LC^{goal} we show that M must satisfy C using the fact that M' is a model of ILC:
If the goal clause C has only interval-predicates then it occurs unchanged in ILC. So it will be satisfied by M' and hence by M.
If C has only noninterval-predicates then C will not be satisfied by M only if there is an instantiation which makes all the literals in C true. But this is not possible because it means that LC^{goal} is inconsistent and would have been detected by the algorithm.
If C had interval-predicates as well as noninterval-predicates then C will not be satisfied by M only if there is an instantiation which makes all the literals in C true. But such an instantiation is not possible because it would make all the noninterval literals true and the remaining goal clause with only interval-predicates would occur in ILC. Then there will be an instantiation of this clause which makes it invalid and therefore ILC cannot have any model, in particular the model M'.

If C is of the form $\leftarrow \mathsf{P}(x) \wedge A(x)$ where $\mathsf{P} \in \mathcal{IP}$ and $A \in \mathcal{NIP}$ then there is an instantiation a for x that makes $\mathsf{P}(x)$ and $A(x)$ true. The expansion $\leftarrow \mathsf{P}(\mathsf{a})$ that occurs in ILC will be invalid since $\mathsf{P}(\mathsf{a})$ is true. Therefore ILC will have no model.

Therefore if M' is a model of ILC then M is a model of LC^{goal}. To show that its numerical submodel is exactly M', *i.e.,* $\forall \mathsf{P} \in \mathcal{IP}, \mu(\mathsf{P}) = \mu'(\mathsf{P})$, we note that no $\mu(\mathsf{P})$ obtained through the procedure for constructing M from M' is affected by any goal clause (because it is at the highest topological order and is encountered last). Therefore, the original proof holds even for LC^{goal}. ∎

5.2 Logical Constraints with Arithmetic Inequalities

We can extend the logical constraint language $\mathcal{LL}$ to include linear arithmetic inequalities in the clauses. We denote this extended language by $\mathcal{LL}^{lineq}$ and the extended constraint language by $\mathcal{CL}^{lineq}$. The numerical constraint language $\mathcal{NL}$ remains unchanged.

The syntactic restrictions on the arithmetic inequalities in $\mathcal{LL}^{lineq}$ are as follows:

1. Any linear inequality or equality with one variable is allowed in the *body* of any rule. The inequalities that fall in this category are $(x\ relop\ \mathtt{a})$ where x is a real variable, $\mathtt{a}$ is a real number constant and *relop* is a relational operator from the set $\{\leq, \geq, <, >, =, \neq\}$.

2. Linear inequalities with one variable that are allowed in the *head* of any rule are of the form $(x\ \ relop\ \ \mathtt{a})$ where x is a real variable, $\mathtt{a}$ is a real number constant and *relop* is a relational operator from the set $\{\leq, \geq, <, >, =\}$.

3. Linear equalities and inequalities that are order relations between two variables are allowed in the *body* of any rule. These inequalities can be of the form $(x\ \ relop\ \ y)$ where x and y are real variables and $relop \in \{\leq, \geq, <, >, =, \neq\}$.

The algorithm for query-answering is modified by treating the arithmetic inequalities as interval-predicates and not expanding them during the *Eliminate_NIP* stage of the *derivation* algorithm. The *convert_to_numerical* algorithm is modified to convert clauses with arithmetic inequalities into numerical constraints. When we do not have any inequalities, each clause to be converted can be easily decomposed into subclauses with at most one variable because all the predicates are unary. When we have inequalities with two variables, that is not the case. We now decompose the clauses into subclauses such that each subclause has the least number of variables possible. We describe the conversions for the additional cases in algorithm 5.3. As before, a superscript *lineq* indicates that the corresponding set or language are allowed to have linear inequalities with the restrictions described above.

ALGORITHM 5.3 (Convert_to_numerical^{lineq})

Input: Set of clauses ILC^{lineq} that has only interval-predicates
and linear inequalities.

Output: Set of arithmetic constraints $quant\_LC^{lineq}$ obtained by converting ILC^{lineq} using the predicate-as-interval assumption.

Method:

```
Initialize $quant\_LC^{lineq} \leftarrow \emptyset$.
For every clause $lc \in ILC^{lineq}$ do
    Initialize $nc \leftarrow emptyset$.
                    % nc is the numerical constraint obtained by converting lc
    $lc \leftarrow Remove\_singlevar\_ineq(lc)$
                    % removes inequalities with one variable by replacing them
                                % with new interval-predicates
    $lc\_subclauses \leftarrow Make\_Subclauses^{lineq}(lc)$.
      % $Make\_subclauses^{lineq}$ breaks lc into subclauses with fewest variables
    For every subclause $subcl \in lc\_subclauses$ that has an inequality do
                  % cases where subclause has no inequality remain unchanged
        $subcl' \leftarrow Convert\_subclause(subcl)$
        $nc \leftarrow nc \vee subcl'$
    Endfor
    $quant\_LC^{lineq} \leftarrow quant\_LC^{lineq} \cup nc$.
Endfor
Return($quant\_LC^{lineq}$)
```
∎

Initially, all the inequalities with one variable of the form $(x \;\; relop \;\; \mathtt{a})$ are removed by replacing them with a new interval-predicate $\mathsf{P}(x)$. The thresholds of this predicate are set according to what the *relop* was. If *relop* is $\neq$, then $x \neq \mathtt{a}$ is equivalent to $(x < \mathtt{a}) \vee (x > \mathtt{a})$ and the clause is equivalent to two clauses, each with one disjunct. The procedure for removing the single-variable inequalities is described in Algorithm 5.4.

ALGORITHM 5.4 (Remove_singlevar_ineq)

Input: Clause lc with single-variable inequalities.
Output: Clause(s) rlc equivalent to lc but with no single-variable inequalities.
Method:
1. If lc has $x \neq \mathtt{a}$ then replace lc by two clauses lc_1 and lc_2

```
    which differ from $lc$ only in this inequality.
    $lc_1$ has $(x < \mathsf{a})$ instead of $x \neq \mathsf{a}$ and
    $lc_2$ has $(x > \mathsf{a})$ instead of $x \neq \mathsf{a}$.
    Apply $Remove\_singlevar\_ineq$ to $lc_1$ and $lc_2$.
    The union of clauses obtained is $rlc$.
2. Replace inequality $(x = \mathsf{a})$ by $\mathsf{P}(x)$, where $\mathsf{P}$ is a new interval-predicate,
        to obtain $rlc$.
   Add constraints $(\mathsf{P}^- \leq \mathsf{P}^+)$, $(\mathsf{P}^- = \mathsf{a})$ and $(\mathsf{P}^+ = \mathsf{a})$ to $quant\_LC^{lineq}$.
3. Replace inequality $(x \leq \mathsf{a})$ or $(x < \mathsf{a})$ by $\mathsf{P}(x)$
        where $\mathsf{P}$ is a new interval-predicate to obtain $rlc$.
   Add constraints $(\mathsf{P}^- \leq \mathsf{P}^+)$, $(\mathsf{P}^- = \Leftrightarrow\infty)$ and $(\mathsf{P}^+ = \mathsf{a})$ to $quant\_LC^{lineq}$.
                      % refer to Appendix A for the difference in boundary value
                                                      % for the strict inequalities
4. Replace inequality $(x \geq \mathsf{a})$ or $(x > \mathsf{a})$ by $\mathsf{P}(x)$
        where $\mathsf{P}$ is a new interval-predicate to obtain $rlc$.
   Add constraints $(\mathsf{P}^- \leq \mathsf{P}^+)$, $(\mathsf{P}^- = \mathsf{a})$ and $(\mathsf{P}^+ = +\infty)$ to $quant\_LC^{lineq}$.    ▮
```

Even after removing the single-variable inequalities from ILC, it can still have two-variable inequalities. We can remove two-variable equalities that are of the form $x = y$ by substituting y for x everywhere in the clause. The only literals in any clause of ILC that have more than one variable are the two-variable inequalities. Therefore, to make subclauses out of each clause, the procedure $Make\_Subclauses^{lineq}$ first separates out all the inequalities that have connected variables. This can be done by representing the inequalities in the clause through a graph where the nodes are the variables and there is an edge (x, y) whenever there is an inequality $x \leq y$ or $x < y$. Each connected component of this graph gives the variables in each subclause. This procedure is described in Algorithm 5.5.

ALGORITHM 5.5 (Make_Subclauses^{lineq})

Input: Clause lc with interval-predicates and two-variable inequalities.

Output: Set $lc\_subclauses$ of subclauses of lc.

```
Method:
1. If $x_1, \ldots, x_n$ are the variables in inequalities,
      construct a graph $G$ with $n$ nodes $x_1, \ldots, x_n$ and
      an edge $(x, y)$ corresponding to each inequality $x \leq y$ or $x < y$.
                                                  % $x \geq y$ is same as $y \leq x$
2. Find connected components of graph $G$.
3. For each connected component $CC$ do
        Get all the literals in $lc$ with the same variables as in $CC$.
        Compute the transitive closure of the inequalities in $CC$.
                                              % If there are $n$ variables in $CC$
                    % there are at most $n^2$ inequalities in the transitive closure
        Add the subclause formed by the literals and
        the inequalities (transitive closure) to $lc\_subclauses$.                ∎
```

Each subclause is next converted to numerical constraints by the procedure *Convert_subclause*. We describe this conversion on a case by case basis in Algorithm 5.6.

ALGORITHM 5.6 (Convert_subclause)

```
Input: Subclause $subcl$ that has the transitive closure
                      of connected two-variable inequalities.
Output: Numerical constraint(s) $subcl'$ using
              predicate-as-interval assumption on $subcl$.
Method:
1. $subcl$ is of the form $\leftarrow \mathsf{P_1}(x_1) \wedge \ldots \wedge \mathsf{P_n}(x_n) \wedge Ineq(x_1, \ldots, x_n)$
   For each inequality $x_i \leq x_j$ where $i, j \in \{1, \ldots, n\}$,
        the conversion $conv(x_i \leq x_j)$ is the expression $\mathsf{P_j}^+ < \mathsf{P_i}^-$
        and for strict inequality $conv(x_i < x_j)$ it is $\mathsf{P_j}^+ \leq \mathsf{P_i}^-$.
   $subcl'$ is the disjunction of $conv(ineq)$ for each $ineq \in Ineq(x_1, \ldots, x_n)$.
   If there are multiple literals in $subcl$ with the same variable, for instance
      $\mathsf{Q_1}(x_1) \wedge \ldots \wedge \mathsf{Q_m}(x_1)$ instead of just $\mathsf{P_1}(x_1)$ then
      in the expression for $conv(ineq)$
      replace $\mathsf{P_1}^-$ by $max\{\mathsf{Q_1}^-, \ldots, \mathsf{Q_m}^-\}$, and
```

```
    replace $\mathsf{P_1}^+$ by $min\{\mathsf{Q_1}^+, \ldots, \mathsf{Q_m}^+\}$
2. subcl is of the form
   $\mathsf{Q}(x) \leftarrow \mathsf{P}(x) \wedge \mathsf{P_1}(x_1) \wedge \ldots \wedge \mathsf{P_n}(x_n) \wedge$
           $Ineq(x_1, \ldots, x_n, y_1, \ldots, y_k, z_1, \ldots, z_m) \wedge$
           $\mathsf{R_1}(y_1) \wedge \ldots \wedge \mathsf{R_k}(y_k) \wedge (x \leq y_1) \wedge \ldots \wedge (x \leq y_k) \wedge$
           $\mathsf{S_1}(z_1) \wedge \ldots \wedge \mathsf{S_m}(z_m) \wedge (x \geq z_1) \wedge \ldots \wedge (x \geq y_k)$
   where instead of $\leq$ or $\geq$, we could have $<$ or $>$ respectively.
   For each inequality ineq in the first set of inequalities $Ineq(\ldots)$,
       expression $conv(ineq)$ is similar to that in step 1 of the algorithm.
       $conv1$ is the disjunction of all such expressions $conv(ineq)$.
       $conv2$ is conjunction of inequalities:
           $\mathsf{Q}^- \leq \mathsf{P}^-$
           $\mathsf{Q}^+ \geq min\{\mathsf{P}^+, \mathsf{R_1}^+, \ldots, \mathsf{R_k}^+\}$
   $conv3$ is conjunction of inequalities:
         $\mathsf{Q}^- \leq max\{\mathsf{P}^-, \mathsf{S_1}^-, \ldots, \mathsf{S_m}^-\}$
         $\mathsf{Q}^+ \geq \mathsf{P}^+$
   $subcl'$ is $(conv1 \vee conv2 \vee conv3)$
   If there are multiple literals with variable $x$ in the body then
     consider their intersection rather than just the interval for P.
   If there is no literal in the body with variable $x$ then
     consider the interval $(-\infty, +\infty)$ instead of $[\mathsf{P}^-, \mathsf{P}^+]$.   ∎
```

We have discussed the changes in the algorithm due to the extended language. We claim that the algorithm is still sound and complete. We do not give the proof but it can be constructed along the lines of the proof for the basic language $\mathcal{CL}$. The intuition behind the proof is that the equivalence of models for LC^{lineq} and ILC^{lineq} is not affected because the arithmetic inequalities are not expanded at this stage. The equivalence of the models for ILC^{lineq} and $quant\_LC^{lineq}$ can be shown by proving for each case that the models for a subclause are the same as the models for the converted inequalities.

We would like to point out that the syntactic restrictions on the arithmetic inequalities become important only in the second stage of the query-answering algorithm when ILC is converted to numerical constraints. In the first stage of eliminating noninterval-predicates, the form of the arithmetic inequalities is unimportant since they are not expanded. The arithmetic inequalities in this stage are only instantiated as values of some variables get fixed due to unification of other noninterval-predicates. Hence, we could in fact have a more general language $\mathcal{LL}^{arith}$ of logical constraints that can have arithmetic inequalities that are not restricted to be linear or by the number of variables they have. The only restriction on this language will be that after the $\mathcal{NIP}$ predicates are eliminated using the $Eliminate\_NIP^{lineq}$ algorithm, the arithmetic inequalities in the output set ILC must satisfy the syntactic restrictions specified for language $\mathcal{LL}^{lineq}$. Thus, we have a procedural definition for this more general language $\mathcal{LL}^{arith}$ rather than a syntactic one as for $\mathcal{LL}^{lineq}$.

We also define a more restricted language than $\mathcal{LL}^{lineq}$ that allows only inequalities with one variable but not those with two variables. We denote this language by $\mathcal{LL}^{v1}$.

Since the effect of linear inequalities is apparent only in the second stage of the algorithm, whereas the effect of goal clauses is apparent only in the first stage of the algorithm, both the extensions to the language can be made independent of each other. Thus, we could have a logical constraint language $\mathcal{LL}^{goal}_{arith}$ that has both goal clauses and arithmetic constraints.

5.3 Recursion

We extend the logical constraint language to allow recursive rules. We denote this extended language by $\mathcal{LL}^{rec}$. In Section 5.3.1, we discuss how the query-answering algorithm needs to be extended to account for the recursive rules. In Section 5.3.2, we show that the extended algorithm is sound and complete and in Section 5.3.3 we discuss the complexity of the problem.

5.3.1 Algorithm

Consider Algorithm 3.4 for eliminating noninterval-predicates in a bottom-up manner using the dependency graph. We could do a topological sort on the graph because the rules were nonrecursive and hence the graph was acyclic. When the rules are recursive, we need to modify this step as discussed below. Again, we use the superscript r to denote the modified procedures and sets due to the recursive rules. The new algorithm to eliminate noninterval-predicates is described in Algorithm 5.7.

We build the dependency graph from the set LC^{rec} of logical constraints as before. If there is a cycle in the graph, then the rules in LC^{rec} are recursive. In this case, we find the strongly-connected components (SCC) of the graph G^{rec} using the algorithm in [Cormen *et al.*, 1986, Section 23.5]. It takes $\mathcal{O}(V + E)$-time to compute the strongly connected components, and the same time to compute the *component graph* G^{SCC} obtained by shrinking each strongly connected component to a single vertex (this graph is guaranteed to be a DAG). We also refer to such a vertex, obtained by collapsing a strongly connected component into a single node, as a *supernode*. Then we do a topological sort on the component graph and assign an ordering number to each node (all the predicates in a supernode get the same ordering number). We can then expand the predicates in this order, starting from source nodes as usual using Algorithm 3.3. The only different case is when a supernode is encountered and we discuss that below.

Each supernode represents a strongly connected component (SCC) and hence the set of rules associated with it. Note that every predicate occurring in the SCC has expansions only in terms of the ground facts or in terms of the interval-predicates occurring before it in the topological order. Also note that we can ignore an SCC that has only noninterval-predicates if any one of the following holds

1. The SCC is disconnected from the rest of the graph.
2. The SCC has only incoming edges.
3. The SCC has only outgoing edges and the predicates have no ground facts.

Let SCC be the set of nodes in the strongly connected component and let R^{SCC}

be the set of constraints in LC with nodes from SCC at the head, *i.e.,*

$$R^{SCC} = \{r \mid r \in LC^{rec}, head\_pred(r) \in SCC\}$$

Let $topo(SCC) = k$, that is, the topological order for all nodes in SCC be k. We define R^{SCC}_{IP} as the set obtained from R^{SCC} by expanding out all noninterval-predicates not in SCC. That is, for any rule $r \in R^{SCC}$, if $v \in body\_preds(r) \cap \mathcal{NIP}$ and $topo(v) < k$, then replace v by one of its expansions from $IP\_expn(v)$. We get as many different rules by replacing v as there are expansions in $IP\_expn(v)$. When all such noninterval-predicates have been expanded out, we obtain set R^{SCC}_{IP}. Note that this has only noninterval-predicates from SCC. The rest are all interval-predicates.

ALGORITHM 5.7 (Eliminate_NIP^{rec})

```
Input: Set of logical constraints $LC^{rec}$.
       Set of interval-predicates $\mathcal{IP}$.
       Set of noninterval-predicates $\mathcal{NIP}$.
Output: $ILC^{rec}$ the set of clauses derived from $LC^{rec}$
        such that it has only interval-predicates
Method:
    Construct dependency graph $G^{rec} = (V, E)$ from $LC^{rec}$.
    Find the strongly connected components (SCC)
        of $G^{rec}$ [Cormen et al., 1986, Section 23.5]
    Construct the component graph $G^{SCC}$ of $G^{rec}$.
    Do a topological sort of $G^{SCC}$ [Cormen et al., 1986, Section 23.4].
    Expand every node $v$ of $G^{SCC}$ in topological order
            If $v$ is not a supernode then use Expand (Algorithm 3.3).
            If $v$ is a supernode then
                Compute set $R^{SCC}_{IP}$ for $v$.
                Expand the predicates in $v$ using
                        $Constraint\_Strata(v, R^{SCC}_{IP}, \mathcal{IP}, \mathcal{NIP})$.
    Expansions of nodes in $\mathcal{IP}$ form the desired set $ILC^{rec}$.
```
∎

The procedure *Constraint_Strata* (Algorithm 5.8) computes the expansions of the rules in a controlled manner to ensure that the expansion procedure terminates. For this we define a series of sets-of-constraints $S_0, \ldots, S_{n+1}$ where

$$\begin{aligned} S_0 &= \{r \mid r \in R_{IP}^{SCC}, head\_pred(r) \in \mathcal{NIP}\} \\ S_{n+1} &= \emptyset \end{aligned}$$

The set S_i is computed from the sets $S_0, \ldots, S_{i-1}$. We will later prove in Section 5.3.2 that n must be finite, *i.e.,* the series of sets will terminate.

ALGORITHM 5.8 (Constraint_Strata)

```
Input: Supernode $v$ (which is a set of predicates).
       Set $R_{IP}^{SCC}$ of rules for $v$ such that
           the only noninterval-predicates are from the supernode.
       Set of interval-predicates $\mathcal{IP}$ .
       Set of noninterval-predicates $\mathcal{NIP}$ .
Output: Set of expansions for every predicate in $v$ such that there are no
        noninterval-predicates in the expansion (i.e., IP_expn of every predicate).
Method:
    $S_0 \leftarrow \{r \mid r \in R_{IP}^{SCC}, head\_pred(r) \in \mathcal{NIP}\}$
    Compute series of sets $S_1, \ldots, S_{n+1}$ until $S_{n+1} = \emptyset$
            using the procedure outlined below.
    $IR^{SCC} \leftarrow I(S_0) \cup \ldots \cup I(S_n)$ where
    $I(S_i) = \{r \mid r \in S_i, body\_preds(r) \subseteq \mathcal{IP}\}$
    Return($IP\_expn(v)$) where
        $IP\_expn(v) = \{r \mid r \in IR^{SCC}, head\_pred(r) = v\}$
```
∎

PROCEDURE: Computing set S_i from $S_0, \ldots, S_{i-1}$

Initialize $S_i \leftarrow \emptyset$.

A rule r is added to S_i if:

1. $r = expand\_step(r_1, l, r_2)$ where $r_1, r_2 \in S_0, \ldots, S_{i-1}$, at least one of r_1 or r_2 is in S_{i-1}, $l \in body(r_1)$ and $pred(l) \in \mathcal{NIP}$. Intuitively, this means that

r is generated by using two rules from previous sets, at least one rule from the immediately preceding set, and no interval-predicates must be expanded to obtain r.

2. $redundant\_rule?(r, S) = false$ where $S = S_0 \cup \ldots \cup S_{i-1} \cup S_i$
r is checked not just for the same rule existing elsewhere, but also those with variable renaming and having the same "pattern" as defined by the function $redundant\_rule?$. The test is actually whether r is "contained" in any other rule, where the result for containment comes from deductive databases (Theorem 14.1 from [Ullman, 1989]).

3. The number of $\mathcal{NIP}$ -predicates in r has not increased from r_1 or r_2.
If $N(r)$ = number of occurrences of $\mathcal{NIP}$ -predicates in r, then $N(r) \leq max(N(r_1), N(r_2))$.

Keep adding rules to S_i as long as new rules can be generated. If $S_i = \emptyset$ after this, then terminate the procedure $(n + 1 = i)$, else compute the next set S_{i+1}. ■

The function $redundant\_rule?(r, S)$ where r is a rule and S is a set of rules, is used while computing the series of sets to test whether r is redundant in S. We describe this function below:

FUNCTION $redundant\_rule?(r, S) : true/false$
Check if there is a containment mapping from any rule in S to r.
If there is, then r is redundant.
There is a containment mapping from rule r' to rule r if every constant and predicate is mapped to itself and every variable is mapped to a constant or a variable, and this mapping is consistent. If there is such a mapping that maps every literal of r' to a literal in r, then there is a containment mapping.
The definition of containment mapping from [Ullman, 1989] is as follows.

Definition 5: [*Containment Mapping*] Let r_1 and r_2 be rules
$r_1 : I \leftarrow J_1 \wedge \ldots \wedge J_l$
$r_2 : H \leftarrow G_1 \wedge \ldots \wedge G_k$
A symbol mapping h is said to be a *containment mapping* if h turns r_2 into r_1; that

is, $h(H) = I$, and for each $i = 1, 2, \ldots, k$ there is some j such that $h(G_i) = J_j$. Note that there is no requirement that each J_j be the target of some G_i, so $h(r_2)$ could look like r_1 with some body literals missing. ▮

Containment mapping between two rules is related to their logical entailment through Theorem 14.1 from [Ullman, 1989].

Theorem 5.2: *Let r_1 and r_2 be as defined in Definition 5. Then $r_1 \Rightarrow r_2$ if and only if there is a containment mapping from r_2 to r_1.*

5.3.2 Formal Results

In this section, we first prove that the algorithm discussed in the previous section terminates. The termination proof is in Section 5.3.2. Then we show that the algorithm is sound and complete in Section 5.3.2.

Termination proof

We first prove that the set ILC^{rec} as defined earlier is finite in Lemma 5.3 and then prove that the expansion of any noninterval-predicate in a strongly connected component must also be finite in Lemma 5.4. Then we will prove that the procedure to generate ILC^{rec} terminates in Theorem 5.5.

Lemma 5.3: *Size of ILC^{rec} obtained from a strongly connected component SCC is finite.*

Proof: The number of predicates and constants is bounded (because there are no functions). Therefore, any unbounded expansion must have repeated predicates with variables. For the rest of the argument, note that all predicates are unary, since ILC^{rec} has only $\mathcal{IP}$ -predicates.

Let x be the variable occurring in the head. Then the unbounded expansion must have predicates with other variables. Separate the variables to form subclauses that have at most one variable. A literal with a constant forms a subclause on its own. Each subclause with a variable can be of size $1, \ldots, p$ where p is the size of $\mathcal{IP}$. All subclauses with variables other than x must be one of a finite numbers possible. For example, two subclauses of size one, $\mathsf{P}(y)$ and $\mathsf{P}(z)$ are identical if y is mapped to z. Similarly for subclauses of larger size. Therefore, size of ILC^{rec} is finite.

In fact, we can compute the upper bound for size of ILC^{rec}.
Let $p = |\mathcal{IP}|$, and $c = |\mathcal{C}|$ where $\mathcal{C}$ is the set of all constants and tuples formed from them,.
$|ILC^{rec}| \leq 2pc + \sum_{i=1}^{p+1}[C(p,i) + pC(p, i \Leftrightarrow 1)]$
where $C(m,n)$ = number of ways of choosing n things out of m, and $C(n, n+1) = 0$.
($2pc$ = number of subclauses with constants, literals can be positive or negative.
$C(p,i)$ = number of subclauses of length i, with a variable, where all literals are negative.
$pC(p, i \Leftrightarrow 1)$ = number of subclauses of length i, with a variable, where exactly one literal is positive.) ∎

Lemma 5.4: *The number of IP_expansions of a predicate $N \in \mathcal{NIP} \cap SCC$ are finite.*

Proof: The proof is similar to that in Lemma 5.3. Separate the subclauses in the body only, except that the head can have more than one variable, so all these variables are "special" (but finite). For all the non-special variables, the argument from previous lemma applies.

Let $arity(N) = k$. We can compute the number of possible "patterns" of the head, where a "pattern" is when a different constant, or a variable is used at a particular position in the $k\Leftrightarrow$tuple for N. Thus, $N(x,y)$ has patterns –
$N(x,y), N(\mathsf{a},y), N(x,\mathsf{a}), N(\mathsf{a},\mathsf{a})$ if a is the only constant.

The maximum number of possible patterns for head $N(x_1, \ldots, x_k)$ =
$c^k + kc^{k-1} + C(k,2)\cdot 2\cdot c^{k-2} + C(k,3)\cdot 3!\cdot c^{k-3} + \ldots + k!$

$$= \sum_{i=0}^{k}[C(k,i)\cdot i!\cdot c^{k-i}]$$

where i is the number of variables present in the pattern.

We can now compute the number of possible distinct *IP_expansions* for a particular pattern $N(\overline{x})$.

The number of possible subclauses that do not have any variables or constants from the head $\overline{x}$ are

$$p(c \Leftrightarrow \overline{c}) + \sum_{i=1}^{p} C(p, i)$$

where $\overline{c}$ = number of constants in $\overline{x}$.

Number of subclauses with constants from head = $p\overline{x}$.

Number of subclauses with variables from head, *i.e.,* $(k \Leftrightarrow \overline{c})$ variables =

$$(k \Leftrightarrow \overline{c}) \cdot \sum_{i=1}^{p} C(p, i)$$

$\blacksquare$

Theorem 5.5: *[*Termination*] The procedure to compute the series of sets* $S_0, \ldots, S_{n+1}$ *terminates.*

Proof: From the computation procedure described in the last section, the number of noninterval-predicates in a generated rule has an upper bound (because of the third condition). Therefore, for the procedure to be non-terminating, some interval-predicate must be repeated infinite number of times. Since the number of variables in noninterval-predicates must be finite, applying the argument of the previous lemma we can see that there are only finitely many repetitions of interval-predicates possible. (In Lemma 5.4, assume that $\overline{c}$ is the number of constants in $\mathcal{NIP}$ -predicates and $k \Leftrightarrow \overline{c}$ is the number of variables. Rest of the argument is the same. Since, we check for the redundant literals through containment mapping, we generate only distinct combinations of interval-predicates.) $\blacksquare$

Soundness and Completeness

We show in Theorem 5.6 that Theorem 3.3 holds even when the rules are recursive; *i.e.,* , the procedure that eliminates the noninterval-predicates preserves the numerical information. In Theorem 5.7 we show that the conversion of logical constraints to numerical constraints also preserves the numerical information.

Theorem 5.6: [Eliminate_NIP^{rec}: Soundness and Completeness] *The class of numerical submodels of* LC^{rec} *w.r.t.* $\mathcal{IP}$ *is identical to the class of standard models of* ILC^{rec} *w.r.t.* $\mathcal{IP}$.

Proof: Soundness proof is unchanged.
Completeness:
Assume that we are given the model $M' = (\Re, \mu\prime)$ for ILC^{rec}. We build M the same way as before starting with sources. When we reach a SCC, we build the model for $\mathcal{NIP}$ -predicates first. Using the rules in $IP\_expn(v)$ for $v \in \mathcal{NIP} \cap SCC$, we build $\mu(v)$ (*i.e.,* whenever the body of an IP_expansion is satisfied by a substitution, then the head must be too). Thus all IP_expansions are satisfied. Then, we do the same for $\mathcal{IP}$ -predicates in SCC. Since, the *IP_expansions* for $\mathcal{IP}$ -predicates in SCC are present in ILC^{rec}, $\mu = \mu'$ for these predicates. Therefore, all we need to prove is that $M \models R^{SCC}$.

Consider $r \in R^{SCC}$ such that $head\_pred(r) = N$ and $N \in \mathcal{NIP}$. If $body(r)$ has only $\mathcal{IP}$ -predicates, then $r \in IP\_expn(N)$ and hence $M \models r$. If $body(r)$ has only $\mathcal{IP}$ -predicates and $\mathcal{NIP}$ -predicates not from SCC, then we can expand the $\mathcal{NIP}$ -predicate via an *IP_expansion* and $M \models$ the resulting *IP_expn*. Then by reasoning of Theorem 3.2 we can show that $M \models r$ (since no recursion is involved here).

Therefore, consider the case where $body(r)$ has at least one $A \in SCC \cap$ $\mathcal{NIP}$. Then r is of the form
$r: \ N(\overline{y}) \leftarrow A(\overline{x}) \wedge \alpha$
Let $M \models body(r)$. We will show that $M \models head(r)$. *i.e.,* if there is a substitution τ for r such that $(body(r))\tau \in \mu(body(r))$ then we will show that $(head(r))\tau \in \mu(head(r))$.

Since $(body(r))\tau \in \mu(body(r))$, therefore $A(\overline{x}\tau) \in \mu(A)$. This is possible if either $A(\overline{x}\tau)$ is a ground fact or there is an IP_expansion of A (in say set S_i) that satisfies $\overline{x}\tau$ (because those are the only two ways that μ was constructed).

Let $r_1: A(\overline{x}) \leftarrow body(r_1)$
exists in S_i such that the substitution τ satisfies $(body(r_1))\tau \in \mu(body(r_1))$ where $body(r_1)$ has only $\mathcal{IP}$ -predicates.

There must exist another rule r_2 in another set S_j where $j > i$ and this rule is obtained by expanding r using r_1. Note that r_2 has one less $\mathcal{NIP}$ -predicate than r, so it will not be eliminated using that restriction. Using the substitution τ for this rule, we see that $body(r_2)\tau \in \mu(body(r_2))$ (because it has literals from bodies of r and r_1 and τ satisfies both). If $body(r_2)$ has no other $\mathcal{NIP}$ -predicates, then r_2 is an IP_expansion; therefore, $M \models r_2$. Since τ satisfies the body, it must satisfy the head too, *i.e.,* $head(r_2)\tau \in \mu(head(r_2))$. But this is what we wanted to prove.

If $body(r_2)$ has other $\mathcal{NIP}$ -predicates, then the same argument can be applied by expanding those $\mathcal{NIP}$ -predicates using IP_expansions until we eliminate all $\mathcal{NIP}$ -predicates from the body. ∎

Theorem 5.7: *[*Convert_to_Numerical^{rec}*:* Soundness and Completeness*] The set of standard models of ILC^{rec} given $\mathcal{IP}$ is identical to the set of models of quant_LC^{rec}.*

Proof: Let a clause lc be converted where lc has many subclauses referred to by $lc_1, lc_2, \ldots$, *i.e.,* $lc \equiv lc_1 \vee lc_2 \vee \ldots$, and where the same interval-predicate P may occur in more than one subclause (because of recursion). Each subclause is converted while preserving the models that satisfy it. The models for a clause are the union of models for its subclauses. The only difference that recursion makes is that some of these models for subclauses interpret the same predicate. Since we take the union of these models, the interpretation for this predicate P only gets expanded (*i.e.,* more possibilities are added). Therefore, the same conversion formulae as in Algorithm 3.5 will preserve the models. ∎

5.3.3 Complexity

We discuss the complexity of answering a query given recursive logical constraints using a result in data dependency theory in deductive databases. An embedded full dependency is defined in Definition 6, and the embedded full dependency implication problem is defined in Definition 7. The complexity result is given in Theorem 5.8.

Definition 6: [Embedded Full Dependency] An embedded full dependency is defined as a sentence of the form

$$\forall x_1 \ldots x_m [P(\overline{y_1}) \wedge \ldots \wedge P(\overline{y_n}) \Rightarrow P(\overline{z})]$$

where $\overline{z}$ can have only the universally quantified variables $x_1 \ldots x_m$ that should also occur in $y_1 \ldots y_m$. ∎

Definition 7: [Embedded Implicational Problem] Given a set , of embedded full dependencies and another embedded full dependency φ, the problem of verifying whether , $\models \varphi$ is defined as an embedded implicational problem. ∎

Theorem 5.8: *[*Recursive Interval Query*] Given a set Σ of Horn rules without functions, i.e., sentences of the form*

$$\forall \overline{x} [Q(\overline{y}) \leftarrow P_1(\overline{x_1}) \wedge \ldots \wedge P_n(\overline{x_n})]$$

where the body might also be empty, and a query q of the form allowed by the query language, the problem of determining whether $\Sigma \models q$ is EXPTIME-hard.

Proof: The embedded full dependency implicational problem is known to be EXPTIME-complete [Chandra *et al.*, 1981]. We reduce the embedded full dependency implicational problem to the recursive interval query problem in polynomial time to show that the recursive interval query problem must be EXPTIME-hard.
Consider an instance of the embedded implicational problem where φ is of the form

$$\forall x_1 \ldots x_m [P(\overline{y_1}) \wedge \ldots \wedge P(\overline{y_n}) \Rightarrow P(\overline{z})]$$

For each variable in φ, introduce new constants, say, $\mathtt{a_1}, \ldots, \mathtt{a_n}$ for the universally quantified variables and $\mathtt{b_i}$'s for the existentially quantified ones. Let the new form of φ by substituting the constants be

$$P(\overline{c_1}) \wedge \ldots \wedge P(\overline{c_n}) \Rightarrow P(\overline{d})$$

We reduce it to an instance of the recursive interval query problem by adding a new rule $Q(\mathsf{a}) \leftarrow P(\overline{d})$ where Q is a new unary interval-predicate and a is a new constant. Also add the ground literals $P(\overline{c_1})$, ..., $P(\overline{c_n})$ to , and now ask the query $Q(\mathsf{a})$. Thus, the augmented , forms the set Σ of Horn clauses without functions for the recursive interval query problem, and $P(\overline{d})$ forms the query q, and we need to verify whether $\Sigma \models q$. It is easy to see that $\Sigma \models q$ if and only if , $\models \varphi$. ∎

From the above theorem, we see that the problem of answering a query is intractable. The embedded implicational problem has a severe restriction that it can have only one predicate. Since we have more than one predicate in our problem, our problem could be even harder. Therefore, any algorithm is bound to take at least exponential time in the worst case.

The logical constraint language $\mathcal{LL}^{rec}$ that has recursive rules can be combined with the language that has goal clauses. The combined algorithm will still satisfy the requirements of soundness and completeness. Therefore, we could have a language $\mathcal{LL}^{goal}_{rec}$ together with the appropriate algorithms.

We cannot combine $\mathcal{LL}^{rec}$ with $\mathcal{LL}^{lineq}$ though, because the termination proof depends upon the interval-predicates being unary. The order relations between two variables will not satisfy this condition and hence the algorithm for recursive case may not terminate if we have such arithmetic constraints. We can allow arithmetic constraints with one variable though since it satisfies the unary-predicate condition. Thus, we can have a language, say $\mathcal{LL}^{v1}_{rec}$ that has recursive rules as well as linear inequalities with single variable, and obtain the appropriate sound and complete algorithms by combining the two algorithms.

Also, since the languages for goal clauses and arithmetic constraints are compatible, we can combine them with the language with recursive rules to get the language $\mathcal{LL}^{grv_1}$ that has goal clauses, linear inequalities with single variable, and recursive rules.

5.4 Clauses with Multiple Positive Literals

We extend the language $\mathcal{LL}$ to the case where the clauses are not required to be Horn but are allowed to have more than one positive literal. We also allow the ground literals to be negative. There is an additional restriction on the clauses referred to as the *single-noninterval-in-head* condition that is defined procedurally in the next section. We denote this extended language for logical constraints by $\mathcal{LL}^{neg}$. As usual, the superscript *neg* is used to denote the changed procedures and sets.

The clauses with multiple positive literals are represented by placing all negative literals to the right of the arrow, and all positive ones to the left. Thus $p, q \leftarrow r, s$ means that $p \vee q \vee \neg r \vee \neg s$.

We assume the classical interpretation of negation. Thus, only those facts are positive that are stated to be so and only those facts are negative that are stated to be so (or which are logically derivable as positive or negative). We interpret the interval-predicates to mean that the interval is positive and the rest is negative. Thus, for the interval-predicate $\mathsf{P}(x)$, the interpretation $\mu(\mathsf{P}) = [\mathsf{P}^-, \mathsf{P}^+]$ and the interpretation $\mu(\neg\mathsf{P}) = (\Leftrightarrow\infty, \mathsf{P}^-) \cup (\mathsf{P}^+, \Leftrightarrow\infty)$.

In Section 5.4.1, we discuss the new algorithm to eliminate the noninterval-predicates from the set of logical constraints LC^{neg} and derive the set ILC^{neg}. In Section 5.4.2 we show that the new algorithm is sound and complete. In Section 5.4.3 we discuss the extended algorithm to convert the constraints in ILC^{neg} to numerical constraints in the set $quant\_LC^{neg}$.

5.4.1 Eliminating Noninterval Predicates

The procedure for eliminating noninterval predicates *Eliminate_NIP* (Algorithm 3.2) is affected by the presence of multiple positive literals only if there is a clause that has multiple-positive literals with noninterval-predicates. If all the multiple positive literals occurring in every clause are interval-predicates, then the noninterval-predicates in the body can be expanded as before without affecting the expansion procedure in any way.

We consider the case where there are clauses with multiple positive literals with noninterval-predicates. Here, the "expansion" of a noninterval-predicate involves resolving two clauses on a noninterval-predicate. For example, if the clauses are $p, q \leftarrow r, s$ and $t, u \leftarrow p, v$ where $p \in \mathcal{NIP}$, then the resolvent is $q, t, u \leftarrow r, s, v$. The elimination algorithm therefore carries out all possible resolutions on noninterval-predicates such that the noninterval-predicates are eliminated.

Let there be k noninterval-predicates $A_1, \ldots, A_k$. We construct a series of sets of constraints $S_k, \ldots, S_0$ starting with $S_k = LC^{neg}$ where S_i has only interval-predicates and the predicates $A_1, \ldots, A_i$. Then, S_0 has only interval-predicates and hence $ILC^{neg} = S_0$.

We place the restriction on LC^{neg}, that if a noninterval-predicate occurs in the head in a rule, then it does not occur anywhere else in the rule. Also, this condition must be satisfied for every set S_i. We refer to this condition as the *single-noninterval-in-head* condition. We describe the procedure $Eliminate\_NIP^{neg}$ in Algorithm 5.9 and the procedure $Eliminate\_A_i$ that constructs S_{i-1} from S_i by eliminating A_i in Algorithm 5.10.

ALGORITHM 5.9 (Eliminate_NIP^{neg})

```
Input: Set of logical constraints $LC^{neg}$
          that satisfy the single-noninterval-in-head condition.
       Set of interval-predicates $\mathcal{IP}$.
       Set of noninterval-predicates $\mathcal{NIP} = \{A_1, \ldots, A_k\}$.
Output: $ILC^{neg}$ the set of clauses derived from $LC^{neg}$
          such that it has only interval-predicates
Method:
    Initialize $S_k = LC^{neg}$.
    For $i$ from $k$ down to 1 do
        $S_{i-1} \leftarrow Eliminate\_A_i(S_i, A_i)$
        where $S_{i-1}$ also satisfies the single-noninterval-in-head condition.
    Endfor
    Return($S_0$)
```
∎

ALGORITHM 5.10 (Eliminate_A_i)

```
Input: Set $S_i$ of logical constraints from $\mathcal{LL}^{neg}$ such that
            the only noninterval-predicates it has are $A_1, \ldots, A_i$.
       Predicate $A_i \in \mathcal{NIP}$.
Output: Set $S_{i-1}$ of logical constraints from $\mathcal{LL}^{neg}$ such that
            the only noninterval-predicates it has are $A_1, \ldots, A_{i-1}$.
Method:
    Initialize $SA \leftarrow S_i$.
    Initialize $SB \leftarrow$ all constraints in $SA$ that do not have $A_i$.
    Repeat
        For every clause $c_1 \in SA$ such that $A_i \in head\_preds(c_1)$ do
            For every clause $c_2 \in SA$ such that $A_i \in body\_preds(c_2)$ do
                Resolve $c_1, c_2$ on $A_i$, if possible, to obtain $c$.
                If $c$ does not satisfy the single-noninterval-in-head condition then
                    signal violation and exit.
                If $c$ is not subsumed by any other clause in $SB$ then add $c$ to $SB$;
            Endfor
        Endfor
        $SA \leftarrow SB$.
        $SB \leftarrow$ all constraints in $SA$ that do not have $A_i$.
    until $SA = SB$.
    Return($SA$)                                                            ∎
```

We observe that the algorithm requires that in any S_i computed by the procedure *Eliminate_A_i*, if the predicate A_i occurs in the head then A_i does not occur in the body of that same clause. This restriction on all the sets $S_k, \ldots, S_1$ is the *single-noninterval-in-head* condition.

5.4.2 Soundness and Completeness

In this section we show that the algorithm **Eliminate_NIP**^{neg} that eliminates noninterval-predicates from LC^{neg} to obtain ILC^{neg} is sound and complete in the sense that the numerical information is preserved. Theorem 5.9 establishes this result.

Theorem 5.9: *The set of standard models of ILC^{neg} is identical to the set of numerical submodels of LC^{neg}.*

Proof: Soundness: Follows from $LC^{neg} \Rightarrow ILC^{neg}$ since the only operation for derivation is resolution.

Completeness: Given a model $M' = (\Re, \mu')$ for ILC^{neg}, we construct a model $M = (D, \mu)$ for LC^{neg}. Once we construct the model, we show that it is indeed a model by proving that $M \models LC^{neg}$ and then show that its numerical submodel is identical to M'.

Constructing the model:

As before, $D = \Re \cup \mathcal{C}$ and for all $\mathsf{P} \in \mathcal{IP}$, $\mu(\mathsf{P}) = \mu'(\mathsf{P})$. Build the interpretation of $\mathcal{NIP}$-predicates in the order $A_1, \ldots, A_k$. Since $S_0 = ILC^{neg}$, we already know that $M \models S_0$. We initialize $\mu(A_i)$ for $i = 1, \ldots, k$ to the constant tuples appearing in positive ground-facts of A_i, also called $pos\_GF(A_i)$. We also define a set not_A_i of constant tuples appearing in the negative ground-facts $neg\_GF(A_i)$ of A_i.

We build $\mu(A_i)$ for $i = 1, \ldots, k$, by having the model satisfy S_i, given that it already satisfies $S_0, \ldots, S_{i-1}$. Since $S_k = LC^{neg}$, this process will ensure that $M \models LC^{neg}$. Those constraints in S_i that do not have A_i are already satisfied because they also appear in S_{i-1}.

Now, consider the constraints in S_i that have A_i in the head. These constraints have only interval-predicates and the noninterval-predicates $A_1, \ldots, A_{i-1}$, all of which already have the interpretation μ. Now we construct the interpretation $\mu(A_i)$ for A_i. Consider a constraint $r \in S_i$ that has $A_i(\overline{x})$ in the head; because of the *single-noninterval-in-head* condition, A_i does not appear anywhere else in r. If there is a substitution τ such that $\tau(body(r)) \in \mu(body(r))$ (*i.e.,* the substitution satisfies the body), then consider $\tau(head(r))$. If for any predicate $u(\overline{y}) \in head\_preds(r)$ it is the case that $\overline{y}\tau \in \mu(u)$, then do nothing (because this means that the head is already satisfied); else, add $\overline{x}\tau$ to $\mu(A_i)$ and thus have A_i satisfy the head.

We can show that $\overline{x}\tau \notin not\_A_i$ and hence this will not lead to any contradictions . If $\overline{x}\tau \in not\_A_i$ then $\neg A_i(\overline{x}\tau)$ would have been resolved with r to give a rule r_1 in S_{i-1}

such that r_1 is of the form $\tau(head(r) \setminus A_i(\overline{x})) \leftarrow \tau(body(r))$. Since all clauses in S_{i-1} are satisfied, r_1 is too and hence if $\tau(body(r))$ is satisfied then $\tau(head(r))$ without $A_i(\overline{x})$ will also be satisfied. But this contradicts the fact that no other predicate in $head(r)$ is satisfied by the substitution τ. Hence, $\overline{x}\tau \notin not\_A_i$.

Once this process has been repeated for all such τ's and all such r's that have A_i in the head, we have $\mu(A_i)$. We can show that $\mu(A_i)$ satisfies those constraints in S_i that have A_i in the body (Lemma 5.10). Thus, all constraints in S_i have been satisfied.

Also, note that during this process, we did not make any changes to the interpretations of $\mathcal{IP}$ -predicates. Hence, for all $\mathsf{P} \in \mathcal{IP}$, $\mu(\mathsf{P}) = \mu'(\mathsf{P})$, which means that the numerical submodel of M is identical to M'. ∎

Lemma 5.10: *The interpretation μ of the predicate $\mathrm{A_i} \in \mathcal{NIP}$ satisfies those constraints in S_i that have $\mathrm{A_i}$ in the body.*

Proof: *(By contradiction)* Let constraint $c_2 \in S_i$, where $A_i(\overline{x})$ occurs in $body(c_2)$, violates this. This means that $body(c_2)$ is satisfied but $head(c_2)$ is not for any substitution τ of c_2. Let c_1 be another constraint in S_i such that it has A_i in the head, and this was the constraint that added $\overline{x}\tau$ to $\mu(A_i)$. Let c_1 and c_2 were resolved to obtain c that was in S_{i-1} and hence μ satisfies c. c_1 can either be a positive ground fact or a clause.

If c_1 is a positive ground fact, then it must be $A_i(\overline{x}\tau)$ since c_1 was the one that added that constant to $\mu(A_i)$. This means that c is of the form $\tau(head(c_2)) \leftarrow \tau(body(c_2) \setminus A_i(\overline{x}))$. That is, if $body(c_2)$ is satisfied by τ then $\tau(head(c_2))$ must be too. But this contradicts our assumption that $head(c_2)$ is not satisfied by any substitution.

If c_1 is not a ground fact but a rule with a body, then it must have $A_i(\overline{x})$ in the head. Also, since c_1 was the one that added the constant $\overline{x}\tau$ to $\mu(A_i)$, it must be the case that τ did not satisfy any other predicates in $head(c_1)$ though it satisfies $body(c_1)$. Then, on resolving c_1 and c_2, we obtain c that must be of the form $(head(c_1) \setminus A_i(\overline{x})), head(c_2) \leftarrow body(c_1), (body(c_2) \setminus A_i(\overline{x}))$, and it is the case that: τ satisfies $body(c)$ but τ does not satisfy $head(c)$. But, this is a contradiction since $c \in S_{i-1}$.

If c_2 has more than one occurrence of A_i in the body, then this argument can be extended by resolving c_2 repeatedly with c_1 and the successive resolvents, until a constraint c without any occurrence of A_i is obtained (and which must hence be in S_{i-1}). ∎

5.4.3 Converting to Numerical Constraints

Once the expanded clauses have only interval-predicates they can be converted to numerical constraints. The expressions for exact conversions are complex. We first define some terms to express the converted expressions. We have also worked out weaker and stronger converted constraints which have simpler expressions.

For intervals $\mu(\mathsf{P_1}), \ldots, \mu(\mathsf{P_n})$ of interval-predicates, we define the following terms:

Definition 8: *(Intersection)* The intersection of intervals $\mu(\mathsf{P_1}), \ldots, \mu(\mathsf{P_n})$ is denoted by $inter(\mathsf{P_1}, \ldots, \mathsf{P_n})$ and is the same as the interval

$$inter(\mathsf{P_1}, \ldots, \mathsf{P_n}) \stackrel{\text{def}}{=} [max(\mathsf{P_1}^-, \ldots, \mathsf{P_n}^-), min(\mathsf{P_1}^+, \ldots, \mathsf{P_n}^+)]$$

∎

Definition 9: (*Empty*) If an interval $A = [\mathsf{A}^-, \mathsf{A}^+]$ is empty, then

$$empty(A) \stackrel{\text{def}}{=} (\mathsf{A}^+ < \mathsf{A}^-)$$

If an interval $A = [\mathsf{A}^-, \mathsf{A}^+]$ is not empty, then

$$not\_empty(A) \stackrel{\text{def}}{=} (\mathsf{A}^- \leq \mathsf{A}^+)$$

∎

Definition 10: *(Cover)* An interval $A = [\mathsf{A}^-, \mathsf{A}^+]$ is said to cover interval $B = [\mathsf{B}^-, \mathsf{B}^+]$ if and only if every point in interval B is also in interval A. That is,

$$covers(A, B) \stackrel{\text{def}}{=} (\mathsf{A}^- \leq \mathsf{B}^-) \wedge (\mathsf{B}^+ \leq \mathsf{A}^+)$$

∎

Definition 11: (*Union*) The union of intervals $\mu(\mathsf{P_1}), \ldots, \mu(\mathsf{P_n})$ for the n interval-predicates is denoted by $union(\mathsf{P_1}, \ldots, \mathsf{P_n})$. An interval $A = [\mathsf{A}^-, \mathsf{A}^+]$ is said to cover a union of intervals $union(\mathsf{P_1}, \ldots, \mathsf{P_n})$, if and only if every point in the union is in A. That is,

$$(\mathsf{A}^- \leq min(\mathsf{P_1}^-, \ldots, \mathsf{P_n}^-)) \wedge (max(\mathsf{P_1}^+, \ldots, \mathsf{P_n}^+) \leq \mathsf{A}^+)$$

A is a *minimal cover* if

$$(\mathsf{A}^- = min(\mathsf{P_1}^-, \ldots, \mathsf{P_n}^-)) \wedge (max(\mathsf{P_1}^+, \ldots, \mathsf{P_n}^+) = \mathsf{A}^+)$$

A is referred to as $min\_cover(\mathsf{P_1}, \ldots, \mathsf{P_n})$. ∎

Definition 12: *(Minmax Conversions)* We give the conversions of some expressions with min and max functions into disjunctions and conjunctions of linear inequalities: The conversion of $max(x_1, \ldots, x_n) \leq min(y_1, \ldots, y_m)$ is

$$\bigwedge_{i=1}^{n} \bigwedge_{j=1}^{m} (x_i \leq y_m)$$

The conversion of $min(x_1, \ldots, x_n) \leq min(y_1, \ldots, y_m)$ is

$$\bigwedge_{j=1}^{m} [\bigvee_{i=1}^{n} (x_i \leq y_j)]$$

The conversion of $max(x_1, \ldots, x_n) \leq max(y_1, \ldots, y_m)$ is

$$\bigwedge_{i=1}^{n} [\bigvee_{j=1}^{m} (x_i \leq y_j)]$$

The conversion of $min(x_1, \ldots, x_n) \leq max(y_1, \ldots, y_m)$ is

$$\bigvee_{i=1}^{n} \bigvee_{j=1}^{m} (x_i \leq y_j)$$

For expressions with $<$ instead of $\leq$, we replace the $\leq$ in the converted expressions by $<$. ∎

Definition 13: *(Overlap)* We say that the intervals $\mathsf{P}_1, \ldots, \mathsf{P}_n$ overlap, or that $overlap(\mathsf{P}_1, \ldots, \mathsf{P}_n)$ is true, when $union(\mathsf{P}_1, \ldots, \mathsf{P}_n) = min\_cover(\mathsf{P}_1, \ldots, \mathsf{P}_n)$. Intuitively, the intervals overlap when they are connected and their union itself forms an interval.

Two intervals $\mathsf{P}_1, \mathsf{P}_2$ overlap if they intersect. That is,

$$overlap(\mathsf{P}_1, \mathsf{P}_2) \overset{\text{def}}{=} (inter(\mathsf{P}_1, \mathsf{P}_2) \neq \emptyset)$$

The overlap of n intervals, $\mathsf{P}_1, \ldots, \mathsf{P}_n$, is defined inductively.

$$\begin{aligned} overlap(\mathsf{P}_1, \ldots, \mathsf{P}_n) \overset{\text{def}}{=} \bigvee\nolimits_{i_1,\ldots,i_n} \; & [overlap(\mathsf{P}_{\mathsf{i}_1}, \ldots, \mathsf{P}_{\mathsf{i}_{\mathsf{n-1}}}) \\ & \wedge \{inter(\mathsf{P}_{\mathsf{i}_\mathsf{n}}, min\_cover(\mathsf{P}_{\mathsf{i}_1}, \ldots, \mathsf{P}_{\mathsf{i}_{\mathsf{n-1}}})) \neq \emptyset\}] \end{aligned}$$

where $i_1, \ldots, i_n \in \{1, \ldots, n\}$ and $i_1 \neq \ldots \neq i_{n-1} \neq i_n$, and all combinations of $i_1, \ldots, i_n$ are considered. ▮

Remark 5.11: The expression for overlapping intervals can be converted to linear inequalities using the linear inequalities for intersection of intervals. That is, $inter(A, B) = [max(\mathsf{A}^-, \mathsf{B}^-), min(\mathsf{A}^+, \mathsf{B}^+)]$ and $inter(A, B) \neq \emptyset$ means that

$$max(\mathsf{A}^-, \mathsf{B}^-) \leq min(\mathsf{A}^+, \mathsf{B}^+)$$

The linear inequalities for *min_cover* can also be substituted, as well as conversions for any *minmax* expressions. ▮

Definition 14: *(Union Cover)* We say that the union of intervals $union(\mathsf{P}_1, \ldots, \mathsf{P}_n)$ covers the interval $Q = [\mathsf{Q}^-, \mathsf{Q}^+]$ if and only if every point in Q is also in the union. Then the union is called a *union cover* of Q. ▮

Remark 5.12: If a union of intervals $union(\mathsf{P}_1, \ldots, \mathsf{P}_n)$ covers an interval Q, then Q must be covered by some overlapping subset of intervals from $\mathsf{P}_1, \ldots, \mathsf{P}_n$. That is, there must be a subset of intervals, say $\mathsf{P}_1, \ldots, \mathsf{P}_k$ where $1 \leq k \leq n$, such that $overlap(\mathsf{P}_1, \ldots, \mathsf{P}_k)$ is true and $min\_cover(\mathsf{P}_1, \ldots, \mathsf{P}_k)$ is a cover for Q. The linear inequality expression will consider all such subsets and check if they overlap and form a cover, *i.e.,*

$$\begin{aligned} union\_cover(\mathsf{P}_1, \ldots, \mathsf{P}_n, Q) \overset{\text{def}}{=} \bigvee_{i_1,\ldots,i_k} & [overlap(\mathsf{P}_{i_1}, \ldots, \mathsf{P}_{i_k}) \\ & \wedge \{A = min\_cover(\mathsf{P}_{i_1}, \ldots, \mathsf{P}_{i_k})\} \\ & \wedge covers(A, Q)] \end{aligned}$$

where $i_1, \ldots, i_k \in \{1, \ldots, n\}$ and $i_1 \neq \ldots \neq i_k$. ∎

We now discuss the conversions of clauses that have only $\mathcal{IP}$ -predicates. In addition to giving the exact conversions, we also specify conversions that are either sound or complete but not both. The expressions for these conversions are simpler than that for exact conversions.

The sound conversions that are not complete mean that these converted constraints must be satisfied; if a point violates these constraints, then it definitely violates the given constraints, but if it satisfies these constraints, then it does not necessarily satisfy the original constraints. In terms of the answers to queries that check whether a condition is satisfied or not, it means that a "yes" answer may be incorrect but a "no" answer is correct. For queries that ask for minimum and maximum values of thresholds, the answers returned may not be the tightest possible.

The complete conversions that are not sound mean that if these converted constraints are satisfied, then the given constraints will definitely be satisfied; if a point satisfies the converted constraints then it satisfies the given constraints, but if it violates the converted constraints it does not necessarily violate the given constraints. In terms of the answers to queries that check whether a condition is satisfied or not, it means that a "no" answer may be incorrect but a "yes" answer is correct. For queries that ask for minimum and maximum values of thresholds, the answers returned may be tighter than the correct answer.

The following is the extended algorithm $Convert\_to\_numerical^{neg}$ for converting the constraints in ILC^{neg} to numerical constraints in $quant\_LC^{neg}$ using the predicate-as-interval assumption.

ALGORITHM Convert_to_numerical^{neg}

1. $\mathsf{P}(\mathsf{a})$ or $\mathsf{P}(\mathsf{a}) \leftarrow$

   Exact Conversion:

   $\mathsf{a} \in \mu(\mathsf{P})$, *i.e.,*

   $\mathsf{P}^- \leq \mathsf{a} \leq \mathsf{P}^+$

2. $\neg\mathsf{P}(\mathsf{a})$ or $\leftarrow \mathsf{P}(\mathsf{a})$

   Exact Conversion:

   $\mathsf{a} \notin \mu(\mathsf{P})$, *i.e.,*

   $(\mathsf{a} < \mathsf{P}^-) \vee (\mathsf{a} > \mathsf{P}^+)$

3. $\mathsf{P}(x)$ or $\mathsf{P}(x) \leftarrow$

   Exact Conversion:

   $\mu(\mathsf{P}) = \Re$, *i.e.,*

   $(\mathsf{P}^- = -\infty) \wedge (\mathsf{P}^+ = +\infty)$

4. $\neg\mathsf{P}(x)$ or $\leftarrow \mathsf{P}(x)$

   Exact Conversion:

   $\mu(\mathsf{P}) = \emptyset$, *i.e.,*

   $\mathsf{P}^- > \mathsf{P}^+$

5. $\leftarrow \mathsf{P}_1(x), \ldots, \mathsf{P}_n(x)$

   Exact Conversion:

   $\mu(\mathsf{P}_1) \cap \ldots \cap \mu(\mathsf{P}_n) = \emptyset$, which is the same as

   $inter(\mathsf{P}_1, \ldots, \mathsf{P}_n) = \emptyset$,

   and using the expressions for intersection and empty set, we have

   $max(\mathsf{P_1}^-, \ldots, \mathsf{P_n}^-) > min(\mathsf{P_1}^+, \ldots, \mathsf{P_n}^+)$, *i.e.,*

   $\bigvee_{i=1}^{n} \bigvee_{j=1}^{n} (\mathsf{P}_i^- > \mathsf{P}_j^+)$

6. $\mathsf{P}(x) \leftarrow \mathsf{Q}_1(x), \ldots, \mathsf{Q}_n(x)$

   Exact Conversion:

   $(\mu(\mathsf{Q}_1) \cap \ldots \cap \mu(\mathsf{Q}_n)) \subseteq \mu(\mathsf{P})$, *i.e.,*

   $\mu(\mathsf{P})$ covers $inter(\mathsf{Q}_1, \ldots, \mathsf{Q}_n)$, and using expressions for intersection, cover and

minmax conversions, we have
$(\mathsf{P}^- \leq max(\mathsf{Q_1}^-, \ldots, \mathsf{Q_n}^-)) \wedge (\mathsf{P}^+ \geq min(\mathsf{Q_1}^+, \ldots, \mathsf{Q_n}^+))$, *i.e.,*
$(\bigvee_{i=1}^{n} \mathsf{P}^- \leq \mathsf{Q}_i^-) \wedge (\bigvee_{i=1}^{n} \mathsf{P}^+ \geq \mathsf{Q}_i^+)$

7. $\mathsf{P}_1(x), \ldots, \mathsf{P}_n(x) \leftarrow$
   Exact Conversion:
   $\mu(\mathsf{P}_1) \cup \ldots \cup \mu(\mathsf{P}_n) = \Re$
   This means that *overlap*$(\mathsf{P}_1, \ldots, \mathsf{P}_n)$ is true and that *min_cover*$(\mathsf{P}_1, \ldots, \mathsf{P}_n) = \Re$. We can also say that *union*$(\mathsf{P}_1, \ldots, \mathsf{P}_n)$ covers $\Re$.
   Therefore, the conversion into linear inequalities is

$$overlap(\mathsf{P}_1, \ldots, \mathsf{P}_n) \wedge \{min\_cover(\mathsf{P}_1, \ldots, \mathsf{P}_n) = \Re\}$$

   We can also specify a *sound conversion* :

$$[min(\mathsf{P_1}^-, \ldots, \mathsf{P_n}^-), max(\mathsf{P_1}^+, \ldots, \mathsf{P_n}^+)] = \Re$$

   that is

$$[\bigvee_{i=1}^{n} (\mathsf{P_i}^- = \Leftrightarrow\infty)] \wedge [\bigvee_{i=1}^{n} (\mathsf{P_i}^+ = +\infty)]$$

   We can specify a *complete conversion* :

$$\bigvee_{i=1}^{n} (\mu(\mathsf{P}) = \Re)$$

   which is the same as

$$\bigvee_{i=1}^{n} [(\mathsf{P_i}^- = \Leftrightarrow\infty) \wedge (\mathsf{P_i}^+ = +\infty)]$$

8. $\mathsf{Q}_1(x), \ldots, \mathsf{Q}_m(x) \leftarrow \mathsf{P}_1(x), \ldots, \mathsf{P}_n(x)$
   Exact Conversion:
   $[\mu(\mathsf{P_1}) \cap \ldots \cap \mu(\mathsf{P_n})] \subseteq [\mu(\mathsf{Q_1}) \cup \ldots \cup \mu(\mathsf{Q_m})]$
   which means that the union of Q's covers the intersection of P's, *i.e., union*$(\mathsf{Q}_1, \ldots, \mathsf{Q}_m)$ is a union cover for *inter*$(\mathsf{P}_1, \ldots, \mathsf{P}_n)$.

The linear inequality expression will be

$$union\_cover(\mathsf{Q}_1, \ldots, \mathsf{Q}_m, inter(\mathsf{P}_1, \ldots, \mathsf{P}_n))$$

We can specify sound or complete conversions by specifying which subsets of $\mathsf{Q}_1, \ldots, \mathsf{Q}_m$ must cover the intersection of $\mathsf{P}_1, \ldots, \mathsf{P}_n$.

∎

The extended language $\mathcal{LL}^{neg}$ can be combined with the language with goal clauses $\mathcal{LL}^{goal}$ as long as the extended language also satisfies the *single-noninterval-in-head* condition. We denote such a combined language by $\mathcal{LL}^{neg}_{goal}$. The respective algorithms can also be combined to provide a combined sound and complete algorithm.

This language can also be allowed to have arithmetic inequalities with single-variable. Such a language will be denoted by $\mathcal{LL}^{neg}_{v1}$ and a language where all three are combined as $\mathcal{LL}^{gnv_1}$. Arithmetic inequalities with two variables will require the conversion algorithm to include cases where there are multiple positive literals as well as arithmetic inequalities with two variables.

The *single-noninterval-in-head* condition precludes recursion, therefore, we cannot combine this language with $\mathcal{LL}^{rec}$.

5.5 n-ary interval predicates

We consider the case where the interval-predicates are not restricted to be unary, but are allowed to be n-ary. We make the assumption that the n dimensions of the predicate are all independent of each other. Under this assumption, the constraints with n-ary predicates boils down to the unary case. We discuss how this reduction can be made in this section.

Let us first consider a binary interval-predicate $\mathsf{P}(x, y)$ which is interpreted as a rectangular box over $\Re^2$. We say that $\mathsf{P}(x, y)$ is true for all points inside the box and false for points outside. The four vertices of the rectangle are expressed in terms of four thresholds $\mathsf{P_x}^-, \mathsf{P_x}^+, \mathsf{P_y}^-, \mathsf{P_y}^+$. The interpretation of P is given by the following expression

$$\mu(\mathsf{P}) = \{\langle x, y\rangle \mid \mathsf{P_x}^- \leq x \leq \mathsf{P_x}^+, \mathsf{P_y}^- \leq y \leq \mathsf{P_y}^+ \text{where } x, y \in \Re\}$$

Note that for a binary predicate we need four thresholds to denote the interpretation, just as for a unary predicate we needed two. In general, we need $2n$ thresholds for an n-ary predicate.

The binary predicate can be expressed as a conjunction of two unary interval-predicates. Let these unary predicates are $\mathsf{P_x}$ and $\mathsf{P_y}$. Then, $\mathsf{P}(x, y) \Leftrightarrow \mathsf{P_x}(x) \wedge \mathsf{P_y}(y)$, where

$$\begin{aligned} \mu(\mathsf{P_x}) &= \{x \mid \mathsf{P_x}^- \leq x \leq \mathsf{P_x}^+, \text{where } x \in \Re\} \\ \mu(\mathsf{P_y}) &= \{y \mid \mathsf{P_y}^- \leq y \leq \mathsf{P_y}^+, \text{where } y \in \Re\} \end{aligned}$$

where the thresholds for $\mathsf{P_x}$ and $\mathsf{P_y}$ are the same as the thresholds for $\mathsf{P}(x, y)$.

Therefore, we can substitute $\mathsf{P_x}(x) \wedge \mathsf{P_y}(y)$ wherever we have $\mathsf{P}(x, y)$. If $\mathsf{P}(x, y)$ occurs in the body of the clause, then a substitution will give one new clause. If $\mathsf{P}(x, y)$ occurs in the head of the clause, then substitution will give rise to 2 clauses, one each with $\mathsf{P_x}$ and $\mathsf{P_y}$ at the head.

Let us extend this to the case of an n-ary interval-predicate $\mathsf{P}(x_1, \ldots, x_n)$ which is interpreted as a rectangular box over $\Re^n$. The vertices of the rectangle are expressed in terms of the thresholds $\mathsf{P_1}^-, \mathsf{P_1}^+, \ldots, \mathsf{P_n}^-, \mathsf{P_n}^+$. The interpretation of the predicate P is

$$\begin{aligned} \mu(\mathsf{P}) = \{\langle x_1, \ldots, x_n\rangle \quad \mid \quad & \mathsf{P_1}^- \leq x_1 \leq \mathsf{P_1}^+, \ldots, \mathsf{P_n}^- \leq x_n \leq \mathsf{P_n}^+, \\ & \text{where } x_1, \ldots, x_n \in \Re\} \end{aligned}$$

The n-ary predicate can be expressed as a conjunction of n unary interval-predicates. Let these unary predicates are $\mathsf{P_1}, \ldots, \mathsf{P_n}$. Then,

$$\mathsf{P}(x_1, \ldots, x_n) \Leftrightarrow \mathsf{P_1}(x_1) \wedge \ldots \wedge \mathsf{P_n}(x_n)$$

where the interpretations of the unary predicates are as follows:

$$\begin{aligned}\mu(\mathsf{P_1}) &= \{x_1 \mid \mathsf{P_1}^- \le x_1 \le \mathsf{P_1}^+, \text{where } x_1 \in \Re\}\\ &\vdots\\ \mu(\mathsf{P_n}) &= \{x_n \mid \mathsf{P_n}^- \le x_n \le \mathsf{P_n}^+, \text{where } x_n \in \Re\}\end{aligned}$$

where the thresholds for $\mathsf{P_1}, \ldots, \mathsf{P_n}$ are the same as the thresholds for $\mathsf{P}(x_1, \ldots, x_n)$.

Therefore, we can substitute $\mathsf{P_1}(x_1) \wedge \ldots \wedge \mathsf{P_n}(x_n)$ wherever we have $\mathsf{P}(x_1, \ldots, x_n)$. If $\mathsf{P}(x_1, \ldots, x_n)$ occurs in the body of the clause, then a substitution will give one new clause. For instance, a clause of the form

$$\alpha \leftarrow \beta \ \wedge \ \mathsf{P}(x_1, \ldots, x_n)$$

where α and β are subclauses, can be converted to the clause

$$\alpha \leftarrow \beta,\ \mathsf{P_1}(x_1) \wedge \ldots \wedge \mathsf{P_n}(x_n)$$

If $\mathsf{P}(x_1, \ldots, x_n)$ occurs in the head of the clause, then substitution will give rise to n clauses, one each with $\mathsf{P_1}, \ldots, \mathsf{P_n}$ at the head. For instance, a clause of the form

$$\alpha \vee \mathsf{P}(x_1, \ldots, x_n) \leftarrow \beta$$

can be converted to n clauses

$$\begin{aligned}\alpha \vee \mathsf{P_1}(x_1) &\leftarrow \beta\\ &\vdots\\ \alpha \vee \mathsf{P_n}(x_n) &\leftarrow \beta\end{aligned}$$

Once all the clauses have only unary interval-predicates, the original algorithm for unary predicates can be applied.

5.6 Interpreting a unary predicate as finite union of intervals

We can extend the interpretation of a unary predicate from a single interval to a finite union of disjoint intervals. This means that an interval-predicate P can be interpreted as the union of intervals $I_1 \cup \ldots \cup I_n$ for some finite n. We note that this interpretation is closed with respect to $\neg, \wedge, \vee$ operators on intervals-predicates.

If $\mu(\mathsf{P}(x))$ and $\mu(\mathsf{Q}(x))$ are finite union of intervals, then

$$\begin{aligned}
\mu(\neg\mathsf{P}(x)) &= \Re \setminus \mu(\mathsf{P}(x)) \\
\mu(\mathsf{P}(x) \wedge \mathsf{Q}(x)) &= \mu(\mathsf{P}(x)) \cap \mu(\mathsf{Q}(x)) \\
\mu(\mathsf{P}(x) \vee \mathsf{Q}(x)) &= \mu(\mathsf{P}(x)) \cup \mu(\mathsf{Q}(x))
\end{aligned}$$

All these interpretations are themselves finite union of intervals.

Let us denote $\mu(\mathsf{P})$ by the $2n$ thresholds of the n intervals, *i.e.,* by
$\{\mathsf{P_1}^-, \mathsf{P_1}^+, \ldots, \mathsf{P_n}^-, \mathsf{P_n}^+\}$ such that
$\mu(\mathsf{P}) = [\mathsf{P_1}^-, \mathsf{P_1}^+] \cup \ldots \cup [\mathsf{P_n}^-, \mathsf{P_n}^+]$

If a finite upper bound n on the number of disjoint intervals in $\mu(\mathsf{P})$ is known, then we can substitute $\mathsf{P}(x)$ by a disjunction of n unary predicates, each of which is interpreted as a single interval. Thus, $\mathsf{P}(x) \Leftrightarrow \mathsf{P_1}(x) \vee \ldots \vee \mathsf{P_n}(x)$, where for all $i = 1, \ldots, n$, $\mu(\mathsf{P_i}) = [\mathsf{P_i}^-, \mathsf{P_i}^+]$. We have the additional constraint on these thresholds that

$$\Leftrightarrow\!\infty \leq \mathsf{P_1}^- \leq \mathsf{P_1}^+ \leq \ldots \leq \mathsf{P_n}^- \leq \mathsf{P_n}^+ \leq +\infty$$

If $\mathsf{P}(x)$ occurs in the body of a clause, then substitution will give rise to n new clauses. A clause of the form

$$\alpha \leftarrow \mathsf{P}(x),\ \beta$$

is converted to the following form on substitution

$$\begin{aligned}
\alpha &\leftarrow \mathsf{P_1}(x),\ \beta \\
&\vdots
\end{aligned}$$

$$\alpha \;\leftarrow\; \mathsf{P}_\mathsf{n}(x),\; \beta$$

If $\mathsf{P}(x)$ occurs in the head of a clause, then substitution will give rise to a new clause. A clause of the form

$$\alpha,\; \mathsf{P}(x) \leftarrow \beta$$

is converted to the following form on substitution

$$\alpha,\; \mathsf{P}_1(x), \ldots, \mathsf{P}_\mathsf{n}(x) \leftarrow \beta$$

Thus the conversion will be the conversions for predicates with single intervals, together with the additional constraint relating the thresholds of P. Some of the converted expressions get simplified due to this additional constraint. For example, to check if any subset of $\mathsf{P}_1, \ldots, \mathsf{P}_\mathsf{n}$ overlap, we know that P_1 cannot overlap with P_3 unless it also overlaps with P_2. Therefore, we need to consider only those subsets that are contiguous. In fact, we can specify some constraints on the negation, conjunction and disjunction of such predicates.

1. $\mathsf{P}(x)$
   $\mu(\mathsf{P}) = [\mathsf{P}_1{}^-, \mathsf{P}_1{}^+] \cup \ldots \cup [\mathsf{P}_\mathsf{n}{}^-, \mathsf{P}_\mathsf{n}{}^+]$

2. $\neg\mathsf{P}(x) \Leftrightarrow \mathsf{S}(x)$

$$\begin{aligned} \mu(\mathsf{S}) &= [\mathsf{S}_1{}^-, \mathsf{S}_1{}^+] \cup \ldots \cup [\mathsf{S}_\mathsf{n}{}^-, \mathsf{S}_\mathsf{n}{}^+] \\ &= \Re \setminus \mu(\mathsf{P}) \\ &= (\Leftrightarrow\infty, +\infty) \setminus \{(\mathsf{P}_1{}^-, \mathsf{P}_1{}^+) \cup \ldots \cup (\mathsf{P}_\mathsf{m}{}^-, \mathsf{P}_\mathsf{m}{}^+)\} \end{aligned}$$

   where the following constraints hold on thresholds:

$$\begin{aligned} &(\mathsf{S}_1{}^- = \Leftrightarrow\infty) &\vee\quad &(\mathsf{P}_1{}^- = \Leftrightarrow\infty) \\ &(\mathsf{S}_1{}^- \neq \Leftrightarrow\infty) &\vee\quad &(\mathsf{P}_1{}^- \neq \Leftrightarrow\infty) \\ &(\mathsf{S}_\mathsf{n}{}^+ = +\infty) &\vee\quad &(\mathsf{P}_\mathsf{m}{}^+ = +\infty) \\ &(\mathsf{S}_\mathsf{n}{}^+ \neq +\infty) &\vee\quad &(\mathsf{P}_\mathsf{m}{}^+ \neq +\infty) \end{aligned}$$

$$\begin{aligned}
(m \Leftrightarrow 1) &\leq n \leq (m+1) \\
(\mathsf{S_1}^- = \Leftrightarrow\infty) &\vee \{(\mathsf{S_1}^- = \mathsf{P_1}^+) \wedge (\mathsf{S_2}^- = \mathsf{P_2}^+) \wedge \ldots \wedge (\mathsf{S_{min(n,m)}}^- = \mathsf{P_{min(n,m)}}^+)\} \\
(\mathsf{S_1}^- \neq \Leftrightarrow\infty) &\vee \{(\mathsf{S_1}^+ = \mathsf{P_1}^-) \wedge (\mathsf{S_2}^+ = \mathsf{P_2}^-) \wedge \ldots \wedge (\mathsf{S_{min(n,m)}}^+ = \mathsf{P_{min(n,m)}}^-)\} \\
(\mathsf{S_n}^- = +\infty) &\vee \{(\mathsf{S_n}^+ = \mathsf{P_m}^-) \wedge (\mathsf{S_{n-1}}^+ = \mathsf{P_{m-1}}^-) \wedge \ldots\} \\
(\mathsf{S_n}^- \neq +\infty) &\vee \{(\mathsf{S_n}^- = \mathsf{P_m}^+) \wedge (\mathsf{S_{n-1}}^- = \mathsf{P_{m-1}}^+) \wedge \ldots\}
\end{aligned}$$

3. $\mathsf{P}(x) \wedge \mathsf{Q}(x) \Leftrightarrow \mathsf{S}(x)$

$$\begin{aligned}
\mu(\mathsf{S}) &= [\mathsf{S_1}^-, \mathsf{S_1}^+] \cup \ldots \cup [\mathsf{S_n}^-, \mathsf{S_n}^+] \\
&= \mu(\mathsf{P}) \cap \mu(\mathsf{Q}) \\
&= \{(\mathsf{P_1}^-, \mathsf{P_1}^+) \cup \ldots \cup (\mathsf{P_m}^-, \mathsf{P_m}^+)\} \cap \{(\mathsf{Q_1}^-, \mathsf{Q_1}^+) \cup \ldots \cup (\mathsf{Q_k}^-, \mathsf{Q_k}^+)\}
\end{aligned}$$

where the following constraints hold on thresholds, for all $i = 1, \ldots, n$:

$$\begin{aligned}
0 &\leq n \leq max(m,k) \\
\mathsf{S_i}^- &\geq min(\mathsf{P_i}^-, \mathsf{Q_i}^-) \\
\mathsf{S_i}^- &\in \{\mathsf{P_j}^-, \mathsf{Q_l}^- \mid 1 \leq j \leq m, 1 \leq l \leq k\} \\
\mathsf{S_i}^+ &\in \{\mathsf{P_j}^+, \mathsf{Q_l}^+ \mid 1 \leq j \leq m, 1 \leq l \leq k\} \\
\mathsf{S_1}^- &\geq max(\mathsf{P_1}^-, \mathsf{Q_1}^-) \\
\mathsf{S_n}^+ &\leq min(\mathsf{P_m}^+, \mathsf{Q_k}^+)
\end{aligned}$$

4. $\mathsf{P}(x) \vee \mathsf{Q}(x) \Leftrightarrow \mathsf{S}(x)$

$$\begin{aligned}
\mu(\mathsf{S}) &= [\mathsf{S_1}^-, \mathsf{S_1}^+] \cup \ldots \cup [\mathsf{S_n}^-, \mathsf{S_n}^+] \\
&= \mu(\mathsf{P}) \cup \mu(\mathsf{Q}) \\
&= \{(\mathsf{P_1}^-, \mathsf{P_1}^+) \cup \ldots \cup (\mathsf{P_m}^-, \mathsf{P_m}^+)\} \cup \{(\mathsf{Q_1}^-, \mathsf{Q_1}^+) \cup \ldots \cup (\mathsf{Q_k}^-, \mathsf{Q_k}^+)\}
\end{aligned}$$

where the following constraints hold on thresholds, for all $i = 1, \ldots, n$:

$$1 \leq n \leq m + k; \text{if } m = k = 0 \text{ then } n = 0$$

$$\begin{aligned}
\mathsf{S_i}^- &\leq max(\mathsf{P_i}^-, \mathsf{Q_i}^-)\\
\mathsf{S_i}^- &\in \{\mathsf{P_j}^-, \mathsf{Q_l}^- \mid 1 \leq j \leq m, 1 \leq l \leq k\}\\
\mathsf{S_i}^+ &\in \{\mathsf{P_j}^+, \mathsf{Q_l}^+ \mid 1 \leq j \leq m, 1 \leq l \leq k\}\\
\mathsf{S_1}^- &= min(\mathsf{P_1}^-, \mathsf{Q_1}^-)\\
\mathsf{S_n}^+ &= max(\mathsf{P_m}^+, \mathsf{Q_k}^+)
\end{aligned}$$

Except for the constraints for $\neg\mathsf{P}(x)$, all other constraints are incomplete (but sound), *i.e.,* they do not completely characterize $\mathsf{S}(x)$.

Chapter 6

Conclusion

We started with the problem of representing and reasoning with vague concepts in knowledge-based systems, particularly while using the currently prevalent representational mechanisms such as those which have Tarskian semantics. We argued that in practice, it is sufficient to represent a vague concept as if it were precise, if we could carefully define what the corresponding precise concept would be. Our contribution in the dissertation is to provide a systematic framework for the hitherto *ad hoc* process of defining the vague concepts as if they were precise.

In this framework, the vague concepts are interpreted as intervals over numbers, and the concept is defined precisely if the interval-boundaries (or thresholds) are known. Since the underlying domain knowledge has a bearing upon the meaning of vague concepts, the framework is based upon capturing this knowledge in a constraint language. Then querying this knowledge is useful in determining the threshold values; hence, a query language is provided as part of the framework. Thus, the framework has three main parts: a *constraint language* in which to express the domain knowledge, a *query language* in which to ask queries about the thresholds to extract information from the domain knowledge, and an *algorithm* to compute the answers to the queries.

In the dissertation, we discussed a constraint language that is used widely in rule-based systems and provided an algorithm to compute answers to the queries. Some important features of this algorithm are:

- The algorithm preprocesses the constraints so that the runtime cost of answering the queries is reduced.

- The preprocessing algorithm is independent of the choice of the query language.
- The algorithm combines the logical and numerical constraints in a novel way using the predicate-as-interval assumption without losing any numerical information in this process.

This algorithm was experimented with on two domains to illustrate that the heuristics are necessary to speed up the algorithm and that despite the severe worst-case complexity, the algorithm is efficient in practice.

6.1 Open Questions

The dissertation introduces a basic framework in which to systematically look at the problem of representing vague concepts as precise. It then demonstrates how this framework can be applied in a useful manner. There are many open questions that need to be addressed to make this framework even more useful:

- Choosing a constraint language and query language depending upon the application. Extensive experimentation will help in determining what choice of languages is useful, keeping in view that, in general, the more expressive the language gets, the more inefficient the query-answering algorithm may become. We explored the different choices in extending the logical constraint language in Chapter 5. Similarly, exploring other extensions to the logical constraint language as well as extensions to the numerical constraint language such as nonlinear constraints, or extensions to the query languages will be illuminating. In some application, a different interpretation for the interval-predicate might be more appropriate, for instance, interpreting it as a union of disjoint intervals rather than a single interval.
- Developing incremental algorithms for answering queries. In the algorithm described in the dissertation, the preprocessing algorithm is independent of the numerical constraints. Hence, any additions to numerical constraints does not

affect the performance of the algorithm. Addition of ground facts requires the repetition of only the last stage of expanding rules without having to rebuild the dependency graph. But the addition of a new rule requires redoing most of the work. It will be useful to have an incremental algorithm that does not require too much recomputation when new rules are added to the knowledge-base.

- The current framework provides the valid ranges for choosing threshold values; any point in this valid region will be a consistent choice. It will be interesting to combine this method of delineating valid regions for thresholds from declarative information, with the clustering techniques that cluster large amounts of ground data to provide numerical ranges for concepts [Kerber, 1992].

Appendix A

Function Definitions

For each interval-predicate, we also specify whether the interval associated with it must be open or closed at each end; *i.e.,* the interpretation of P could be $[\mathsf{P}^-, \mathsf{P}^+]$, $(\mathsf{P}^-, \mathsf{P}^+]$, $[\mathsf{P}^-, \mathsf{P}^+)$, or $(\mathsf{P}^-, \mathsf{P}^+)$. We define terms $\mathsf{P}^-.bd$ and $\mathsf{P}^+.bd$ for the thresholds also called the *boundary_indicators*, which can have values *open* or *closed* accordingly.

We also introduce new relations $\leq_{th}, \geq_{th}, <_{th}, >_{th}$ for relations between thresholds which could be the end of a closed or an open interval. The definitions of these new relations are in Algorithms A.2, A.1, A.4, A.3. Effectively, these new relations allow us to specify constraints on the thresholds without having to bother about the *boundary_indicators*.

ALGORITHM A.1 ($\geq_{th}$)

```
Input: Expressions $e1$ and $e2$.                    % $e1, e2$ can be thresholds
Output: Value of relation $e1 \geq_{th} e2$.
Method:
    return($e2 \leq_{th} e1$)                                            ∎
```

ALGORITHM A.2 ($\leq_{th}$)

```
Input: Expressions $e1$ and $e2$.                    % $e1, e2$ can be thresholds
Output: Value of relation $e1 \leq_{th} e2$.
                    % $\mathsf{P}^-, \mathsf{P}^+, \mathsf{Q}^-, \mathsf{Q}^+$ are thresholds , a, b are constants
```

Method:

```
case $e1 \leq_{th} e2$ of
    1: $\mathsf{a} \leq_{th} \mathsf{b}$                                  % neither is a threshold
       return($e1 \leq e2$)
    2: $\mathsf{P}^- \leq_{th} \mathsf{a}$
       if $\mathsf{P}^-.bd = closed$ then return($e1 \leq e2$)
       else return($e1 < e2$)
    3: $\mathsf{a} \leq_{th} \mathsf{P}^-$
       return($e1 \leq e2$)
    4: $\mathsf{P}^+ \leq_{th} \mathsf{a}$
       return($e1 \leq e2$)
    5: $\mathsf{a} \leq_{th} \mathsf{P}^+$
       if $\mathsf{P}^+.bd = closed$ then return($e1 \leq e2$)
       else return($e1 < e2$)
    6: $\mathsf{P}^- \leq_{th} \mathsf{Q}^-$
       if $\mathsf{P}^-.bd = closed$ then return($e1 \leq e2$)
       else if $\mathsf{Q}^-.bd = open$ then return($e1 \leq e2$)
            else return($e1 < e2$)
    7: $\mathsf{P}^+ \leq_{th} \mathsf{Q}^+$
       if $\mathsf{Q}^+.bd = closed$ then return($e1 \leq e2$)
       else if $\mathsf{P}^+.bd = open$ then return($e1 \leq e2$)
            else return($e1 < e2$)
    8: $\mathsf{P}^- \leq_{th} \mathsf{Q}^+$
       if $\mathsf{P}^-.bd = open$ then return($e1 < e2$)
       else if $\mathsf{Q}^+.bd = closed$ then return($e1 \leq e2$)
            else return($e1 < e2$)
    9: $\mathsf{P}^+ \leq_{th} \mathsf{Q}^-$
       return($e1 \leq e2$)
end
```
∎

ALGORITHM A.3 ($>_{th}$)

```
Input: Expressions $e1$ and $e2$.                    % $e1, e2$ can be thresholds
Output: Value of relation $e1 >_{th} e2$.
Method:
    return($e2 <_{th} e1$)                                                    ∎
```

ALGORITHM A.4 ($<_{th}$)

```
Input: Expressions $e1$ and $e2$.                    % $e1, e2$ can be thresholds
Output: Value of relation $e1 <_{th} e2$.
Method:
    case $e1 <_{th} e2$ of
        1: $a \leq_{th} b$                            % neither is a threshold
           return($e1 < e2$)
        2: $P^- <_{th} a$
           return($e1 < e2$)
        3: $a <_{th} P^-$
           if $P^-.bd = closed$ then return($e1 < e2$)
           else return($e1 \leq e2$)
        4: $P^+ <_{th} a$
           if $P^+.bd = closed$ then return($e1 < e2$)
           else return($e1 \leq e2$)
        5: $a <_{th} P^+$
           return($e1 < e2$)
        6: $P^- <_{th} Q^-$
           if $P^-.bd = open$ then return($e1 < e2$)
           else if $Q^-.bd = closed$ then return($e1 < e2$)
                else return($e1 \leq e2$)
        7: $P^+ <_{th} Q^+$
           if $Q^+.bd = open$ then return($e1 < e2$)
           else if $P^+.bd = closed$ then return($e1 < e2$)
                else return($e1 \leq e2$)
    8: $\mathsf{P}^{-} <_{th} \mathsf{Q}^{+}$
       **return**($e1 < e2$)
    9: $\mathsf{P}^{+} <_{th} \mathsf{Q}^{-}$
       **if** $\mathsf{P}^{+}.bd = open$ **then return**($e1 < e2$)
       **else if** $\mathsf{Q}^{-}.bd = closed$ **then return**($e1 < e2$)
            **else return**($e1 \leq e2$)
end                                                        ∎
```

Conversion of clauses with inequalities If an inequality has only one variable, say x, and is of the form $x = \mathsf{a}$ then simply substitute a for x everywhere in the clause. If it is an inequality *i.e.,* $\leq, \geq, <, >$ then we introduce a new interval-predicate P with boundary-indicators $\mathsf{P}^{-}.bd$ and $\mathsf{P}^{+}.bd$. We add the inequality $\mathsf{P}^{-} \leq_{th} \mathsf{P}^{+}$ to NC and replace the inequality in the clause by the $\mathsf{P}(x)$. Then, according to the form of the inequality, we also add the following numerical constraints to *quant_LC*:

- $x \leq \mathsf{a}$: $\mathsf{P}^{-} = \Leftrightarrow\infty$ and $\mathsf{P}^{+} = \mathsf{a}$ where $\mathsf{P}^{-}.bd = open$ and $\mathsf{P}^{+}.bd = closed$.
- $x \geq \mathsf{a}$: $\mathsf{P}^{-} = \mathsf{a}$ and $\mathsf{P}^{+} = +\infty$ where $\mathsf{P}^{-}.bd = closed$ and $\mathsf{P}^{+}.bd = open$.
- $x < \mathsf{a}$: $\mathsf{P}^{-} = \Leftrightarrow\infty$ and $\mathsf{P}^{+} = \mathsf{a}$ where $\mathsf{P}^{-}.bd = open$ and $\mathsf{P}^{+}.bd = open$.
- $x > \mathsf{a}$: $\mathsf{P}^{-} = \mathsf{a}$ and $\mathsf{P}^{+} = +\infty$ where $\mathsf{P}^{-}.bd = open$ and $\mathsf{P}^{+}.bd = open$.

Appendix B

Medical Domain Application

The set of interval-predicates $\mathcal{IP}$ is:

```
(verylow-CO low-CO normal-CO high-CO veryhigh-CO
verylow-HR low-HR normal-HR high-HR veryhigh-HR criticallyhigh-HR
low-PCWP normal-PCWP high-PCWP
low-SVR normal-SVR high-SVR)
```

The set of noninterval-predicates $\mathcal{NIP}$ is:

```
(low-SV normal-SV high-SV CO HR PCWP SVR bradycardia
sinus-tachycardia extreme-tach vasoconstriction vasodilation)
```

The set of numerical constraints NC is:

```
((= verylow-CO+ low-CO-) (= verylow-CO- 0) (<= 2.0 low-CO-)
(<= low-CO- 2.8)
(= low-CO+ normal-CO-) (<= 3.0 normal-CO-) (<= normal-CO- 4.2)
(= normal-CO+ high-CO-) (<= 5 high-CO-) (<= high-CO- 6.5)
(= high-CO+ veryhigh-CO-) (<= 7.5 veryhigh-CO-) (<= veryhigh-CO- 8.5)
(= veryhigh-CO+ 50)
(= verylow-HR- 0) (= verylow-HR+ low-HR-) (<= 50 low-HR-)
(<= low-HR- 60)
(= low-HR+ normal-HR-) ((<= 60 normal-HR-) (<= normal-HR- 70)
(= normal-HR+ high-HR-) (<= 90 high-HR-) (<= high-HR- 110)
(= high-HR+ veryhigh-HR-) (<= 105 veryhigh-HR-) (<= veryhigh-HR- 120)
(= veryhigh-HR+ criticallyhigh-HR-) (<= 150 criticallyhigh-HR-)
(<= criticallyhigh-HR- 175) (= criticallyhigh-HR+ 500)
(= low-PCWP- 0)(= low-PCWP+ normal-PCWP-) (<= 8 normal-PCWP-)
(<= normal-PCWP- 10)
(= normal-PCWP+ high-PCWP-) (<= 12 high-PCWP-) (<= high-PCWP- 15)
(= high-PCWP+ 100) (= low-SVR- 0)
(= low-SVR+ normal-SVR-) (<= 900 normal-SVR-) (<= normal-SVR- 1200)
(= normal-SVR+ high-SVR-) (<= 1400 high-SVR-) (<= high-SVR- 1700)
(= high-SVR+ 5000))
```

Note that all the numerical constraints here are either inequalities with only one variable or are order constraints. The set of logical constraints LC is given below. The patient-data at the end are the ground literals and the other constraints form the set of rules. A rule written as

```
((normal-CO y) (normal-SV x) (normal-HR u) (CO x y) (HR x u))
```

represents a rule of the form

$$\mathsf{normal\_CO}(y) \leftarrow normal\_SV(x) \wedge \mathsf{normal\_HR}(u) \wedge CO(x,y) \wedge HR(x,u)$$

The set of logical constraints LC is

```
(((normal-CO y) (normal-SV x) (normal-HR u) (CO x y) (HR x u))
((low-CO y) (low-SV x)(normal-HR u) (CO x y) (HR x u))
((low-CO y) (normal-SV x) (low-HR u) (CO x y) (HR x u))
((verylow-CO y) (normal-SV x)(bradycardia x)(CO x y))
((verylow-CO y) (low-SV x) (low-HR u) (CO x y) (HR x u))
((verylow-CO y) (low-SV x) (bradycardia x) (CO x y))
((high-CO y) (normal-SV x) (high-HR u) (CO x y) (HR x u))
((high-CO y) (high-SV x) (normal-HR u) (CO x y) (HR x u))
((high-CO y) (high-SV x) (high-HR u) (CO x y) (HR x u))
((veryhigh-CO y) (normal-SV x) (sinus-tachycardia x) (CO x y))
((veryhigh-CO y) (high-SV x) (sinus-tachycardia x) (CO x y))
((low-CO y) (low-SV x) (extreme-tach x) (CO x y))
((normal-SV x) (normal-PCWP z) (normal-SVR u) (PCWP x z) (SVR x u))
((low-SV x) (low-PCWP z) (normal-SVR u) (PCWP x z) (SVR x u))
((low-SV x) (normal-PCWP z) (vasoconstriction x) (PCWP x z))
((low-SV x) (low-PCWP z) (vasoconstriction x) (PCWP x z))
((high-SV x) (high-PCWP z) (normal-SVR u) (PCWP x z) (SVR x u))
((high-SV x) (normal-PCWP z) (vasodilation x) (PCWP x z))
((high-SV x) (high-PCWP z) (vasodilation x) (PCWP x z))
((bradycardia x) (verylow-HR y) (HR x y))
((sinus-tachycardia x) (veryhigh-HR y) (HR x y))
((extreme-tach x) (criticallyhigh-HR y) (HR x y))
((vasoconstriction x) (high-SVR y) (SVR x y))
((vasodilation x) (low-SVR y) (SVR x y))
((CO patient-1 6.0)) ((HR patient-1 75))
((PCWP patient-1 17)) ((SVR patient-1 1300))
((CO patient-2 3.4)) ((HR patient-2 103))
((PCWP patient-2 7.7)) ((SVR patient-2 1800))
((CO patient-3 2.6)) ((HR patient-3 180))
((PCWP patient-3 7)) ((SVR patient-3 1600))
((CO patient-4 3.9)) ((HR patient-4 80))
((PCWP patient-4 11)) ((SVR patient-4 1300))
((CO patient-5 2.2)) ((HR patient-5 48))
((PCWP patient-5 8.5)) ((SVR patient-5 1600))
((CO patient-6 6)) ((HR patient-6 118))
((PCWP patient-6 16)) ((SVR patient-6 1250))
((CO patient-7 8)) ((HR patient-7 138))
((PCWP patient-7 17)) ((SVR patient-7 1000))
((CO patient-8 9)) ((HR patient-8 155))
((PCWP patient-8 14)) ((SVR patient-8 800)))
```

Appendix C

Weather Domain Application

The set of interval-predicates $\mathcal{IP}$ is:

```
(freezing cold medium-temp hot searing
no-precip low-precip medium-precip heavy-precip
low-humidity medium-humidity high-humidity
low-smog moderate-smog unhealthy-smog very-unhealthy-smog)
```

The set of noninterval-predicates $\mathcal{NIP}$ is:

```
(low-perceived-humidity medium-perceived-humidity
high-perceived-humidity
relative-humidity mean-temperature precipitation smog-index
partly-cloudy-sky cloudy-sky clear-sky picnic-day)
```

The set of numerical constraints NC is:

```
(((= freezing- -100)) ((= freezing+ cold-))
((>= cold- 29)) ((<= cold- 43)) ((= cold+ medium-temp-))
((>= medium-temp- 43)) ((<= medium-temp- 60))
((= medium-temp+ hot- )) ((>= hot- 61)) ((<= hot- 76))
((= hot+ searing-)) ((>= searing- 76))
((<= searing- 90)) ((= searing+ 200))
((= no-precip- 0)) ((= no-precip+ 0)) ((= low-precip- 3))
((= low-precip+ medium-precip-)) ((>= medium-precip- 3))
((<= medium-precip- 15)) ((= medium-precip+ heavy-precip-))
((>= heavy-precip- 15)) ((<= heavy-precip- 100))
((= heavy-precip+ 200)) ((= low-humidity- 0))
((= low-humidity+ medium-humidity-)) ((>= medium-humidity- 20))
((<= medium-humidity- 78)) ((= medium-humidity+ high-humidity-))
((>= high-humidity- 78)) ((<= high-humidity- 99))
((= high-humidity+ 100)) ((= low-smog- 0))
((= low-smog+ moderate-smog-)) ((>= moderate-smog- 40))
((<= moderate-smog- 65)) ((= moderate-smog+ unhealthy-smog-))
((>= unhealthy-smog- 85)) ((<= unhealthy-smog- 120))
((= unhealthy-smog+ very-unhealthy-smog-))
((>= very-unhealthy-smog- 175))
((<= very-unhealthy-smog- 225)) ((= very-unhealthy-smog+ 300)))
```

Note that all the numerical constraints here are either inequalities with only one variable or are order constraints. The set of logical constraints LC is given below. The weather-data at the end are the ground literals and the other constraints form the set of rules. The syntax of the rules is the same as for the medical domain in Appendix B.

The set of logical constraints LC is

```
(((low-perceived-humidity x) (relative-humidity x xh) (low-humidity xh)
                              (mean-temperature x xt) (freezing xt))
((low-perceived-humidity x) (relative-humidity x xh) (low-humidity xh)
                             (mean-temperature x xt) (cold xt))
((low-perceived-humidity x) (relative-humidity x xh) (low-humidity xh)
                             (mean-temperature x xt) (medium-temp xt))
((low-perceived-humidity x) (relative-humidity x xh) (low-humidity xh)
                             (mean-temperature x xt) (hot xt))
((low-perceived-humidity x) (relative-humidity x xh) (medium-humidity xh)
                             (mean-temperature x xt) (freezing xt))
((low-perceived-humidity x)(relative-humidity x xh) (medium-humidity xh)
                            (mean-temperature x xt) (cold xt))
```

```
((medium-perceived-humidity x)(relative-humidity x xh) (low-humidity xh)
                              (mean-temperature x xt) (searing xt))
((medium-perceived-humidity x) (relative-humidity x xh)
                               (medium-humidity xh) (mean-temperature x xt)
                               (medium-temp xt))
((medium-perceived-humidity x)(relative-humidity x xh)
                              (medium-humidity xh) (mean-temperature x xt)
                              (hot-temp xt))
((medium-perceived-humidity x)(relative-humidity x xh)
                              (high-humidity xh) (mean-temperature x xt)
                              (freezing xt))
((medium-perceived-humidity x)(relative-humidity x xh)
                              (high-humidity xh) (mean-temperature x xt)
                              (cold xt))
((high-perceived-humidity x)(relative-humidity x xh) (medium-humidity xh)
                            (mean-temperature x xt) (searing xt))
((high-perceived-humidity x)(relative-humidity x xh) (high-humidity xh)
                            (mean-temperature x xt) (medium-temp xt))
((high-perceived-humidity x)(relative-humidity x xh) (high-humidity xh)
                            (mean-temperature x xt) (hot xt))
((high-perceived-humidity x)(relative-humidity x xh) (high-humidity xh)
                            (mean-temperature x xt) (searing xt))
((high-humidity xh)(relative-humidity x xh)
                   (precipitation x xr) (low-precip xr))
((high-humidity xh)(relative-humidity x xh)
                   (precipitation x xr) (medium-precip xr))
((high-humidity xh)(relative-humidity x xh)
                   (precipitation x xr) (heavy-precip xr))
((partly-cloudy-sky x) (precipitation x xr) (low-precip xr))
((cloudy-sky x) (precipitation x xr) (medium-precip xr))
((cloudy-sky x) (precipitation x xr) (heavy-precip xr))
```

```
((no-precip xr) (precipitation x xr) (clear-sky x))
((no-precip xr) (precipitation x xr) (mean-temperature x xt) (hot xt))
((no-precip xr) (precipitation x xr) (mean-temperature x xt) (searing xt))
((low-smog xs) (smog-index x xs) (precipitation x xr) (medium-precip xr)
                (mean-temperature x xt) (freezing xt))
((low-smog xs)(smog-index x xs) (precipitation x xr) (medium-precip xr)
               (mean-temperature x xt) (cold xt))
((low-smog xs) (smog-index x xs) (precipitation x xr)
                (heavy-precip xr) (mean-temperature x xt) (freezing xt))
((low-smog xs) (smog-index x xs) (precipitation x xr) (heavy-precip xr)
                (mean-temperature x xt) (cold xt))
((low-smog xs) (smog-index x xs) (precipitation x xr) (heavy-precip xr)
                (mean-temperature x xt) (medium-temp xt))
((moderate-smog xs) (smog-index x xs) (precipitation x xr) (no-precip xr)
                     (mean-temperature x xt) (searing xt))
((medium-temp xt) (mean-temperature x xt) (picnic-day x))
((no-precip xr) (precipitation x xr) (picnic-day x))
((low-perceived-humidity x) (picnic-day x))
((low-smog xs) (smog-index x xs) (picnic-day x))
((mean-temperature 5jan92 52))
((mean-temperature 30jan92 50))
((mean-temperature 10feb92 54.5))
((mean-temperature 10mar92 54))
((mean-temperature 20apr92 66))
((mean-temperature 21may92 62))
((mean-temperature 25jun92 67))
((mean-temperature 11jul92 75))
((mean-temperature 10aug92 71))
((mean-temperature 30oct92 61))
((mean-temperature 25dec92 40.5))
((mean-temperature 15nov92 53))
```

```
((mean-temperature 5sep92 65))
((mean-temperature 20feb93 48.5))
((precipitation 5jan92 61))
((precipitation 30jan92 0))
((precipitation 10feb92 31))
((precipitation 10mar92 0))
((precipitation 20apr92 0))
((precipitation 21may92 0))
((precipitation 25jun92 0))
((precipitation 11jul92 0))
((precipitation 10aug92 0))
((precipitation 30oct92 90))
((precipitation 25dec92 0))
((precipitation 15nov92 0))
((precipitation 5sep92 0))
((precipitation 20feb93 66))
((relative-humidity 5jan92 100))
((relative-humidity 30jan92 93))
((relative-humidity 10feb92 100))
((relative-humidity 10mar92 94))
((relative-humidity 20apr92 88))
((relative-humidity 21may92 77))
((relative-humidity 25jun92 90))
((relative-humidity 11jul92 68))
((relative-humidity 10aug92 80))
((relative-humidity 30oct92 100))
((relative-humidity 25dec92 91))
((relative-humidity 15nov92 93))
((relative-humidity 5sep92 85))
((relative-humidity 20feb93 100))
((smog-index 5jan92 33))
```

```
((smog-index 30jan92 53))
((smog-index 10feb92 19))
((smog-index 10mar92 23))
((smog-index 20apr92 37))
((smog-index 21may92 34))
((smog-index 25jun92 17))
((smog-index 11jul92 25))
((smog-index 10aug92 42))
((smog-index 30oct92 23))
((smog-index 25dec92 30))
((smog-index 15nov92 35))
((smog-index 5sep92 43))
((smog-index 20feb93 25))
((cloudy-sky 5jan92))
((partly-cloudy-sky 30jan92))
((cloudy-sky 10feb92))
((clear-sky 10mar92))
((clear-sky 20apr92))
((clear-sky 21may92))
((clear-sky 25jun92))
((partly-cloudy-sky 11jul92))
((clear-sky 10aug92))
((cloudy-sky 30oct92))
((clear-sky 25dec92))
((clear-sky 15nov92))
((clear-sky 5sep92))
((cloudy-sky 20feb93))
((picnic-day 20apr92))
((picnic-day 21may92))
((picnic-day 25jun92))
((picnic-day 11jul92))
```

```
((picnic-day 10aug92))
((picnic-day 5sep92)))
```

Bibliography

[Allen and Hayes, 1985] Allen, James F. and Hayes, Pat 1985. A common-sense theory of time. In *Proceedings of the International Joint Conference on Artificial Intelligence*. The International Joint Conferences on Artificial Intelligence, Inc., Morgan Kaufmann Publishers, Inc. 528–531.

[Allen, 1985] Allen, James F. 1985. Maintaining knowledge about temporal intervals. In Brachman, Ronald J. and Levesque, Hector J., editors 1985, *Readings in Knowledge Representation*. Morgan Kaufmann Publishers, Inc. chapter 30, 509–522.

[Chandra and Merlin, 1977] Chandra, A.K. and Merlin, P.M. 1977. Optimal implementation of conjunctive queries in relational databases. In *Proceedings of Symposium on Theory of Computing, New York*. 77–90.

[Chandra *et al.*, 1981] Chandra, A.K.; Lewsis, H.; and Makowsky, 1981. Embedded implicational dependencies and their inference problem. In *Proceedings of Symposium on Theory of Computing, Milwaukee*. 342–354.

[Cormen *et al.*, 1986] Cormen, Thomas H.; Leiserson, Charles E.; and Rivest, Ronald L. 1986. *Introduction to Algorithms*. The MIT Press, McGraw-Hill Book Company.

[DÁmbrosio *et al.*, 1987] DÁmbrosio, Bruce; Fehling, Michael R.; Forrest, Stephanie; Raulefs, Peter; and Wilber, B. Michael 1987. Real-time process management for materials composition in chemical manufacturing. *IEEE Expert* (Summer):80–93.

[Davis, 1987] Davis, Ernest 1987. Constraint propagation with interval labels. *Artificial Intelligence* 32(3):281–331.

[Davis, 1990] Davis, Ernest 1990. *Representations of Commonsense Knowledge.* Morgan Kaufmann Publishers, Inc.

[Encyclopedia Britanica, 1986] Colour. *Encyclopedia Britanica* 16:659.

[Garey and Johnson, 1979] Garey, Michael R. and Johnson, David S. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W.H. Freeman and Co.

[Genesereth and Nilsson, 1987] Genesereth, Michael R. and Nilsson, Nils J. 1987. *Logical Foundations of Artificial Intelligence.* Morgan Kaufmann Publishers, Inc.

[Hayes-Roth *et al.*, 1989] Hayes-Roth, Barbara; Washington, Rich; Hewett, R.; Hewett, M.; and Seiver, A. 1989. Intelligent monitoring and control. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence.* The International Joint Conferences on Artificial Intelligence, Inc., Morgan Kaufmann Publishers, Inc. 43–249.

[Jaffar and Lassez, 1987] Jaffar, Joxan and Lassez, Jean-Louis 1987. Constraint logic programming. *POPL* 111 – 119.

[Jiang *et al.*, 1991] Jiang, Dareng; Han, Chia Yung; and Wee, William G. 1991. Prototype expert system for preventive control in power plant. *SPIE, Applications of Artificial Intelligence* 1468:16–25.

[Karmarkar, 1984] Karmarkar, N. 1984. A new polynomial-time algorithm for linear programming. *Combinatorica* 4:373–395.

[Kautz and Ladkin, 1991] Kautz, Henry A. and Ladkin, Peter B. 1991. Integrating metric and qualitative temporal reasoning. In *Proceedings of the Ninth National Conference on Artificial Intelligence.* American Association for Artificial Intelligence, AAAI Press, The MIT Press. 241–246.

[Kerber, 1992] Kerber, Randy 1992. Chimerge: Discretization of numeic attributes. In *Proceedings of the Tenth National Conference on Artificial Intelligence.* American Association for Artificial Intelligence, AAAI Press, The MIT Press.

[Khachiyan, 1979] Khachiyan, L.G. 1979. A polynomial algorithm in linear programming. *Soviet Mathematics Doklady* 20(1):191–194.

[Ladkin, 1987] Ladkin, Peter B. 1987. Models of axioms for time intervals. In *Proceedings of the National Conference on Artificial Intelligence*. American Association for Artificial Intelligence, AAAI Press, The MIT Press. 234–239.

[Lassez and McAloon, 1992] Lassez, Jean-Louis and McAloon, Ken 1992. A canonical form for generalized linear constraints. *Journal of Symbolic Computation*.

[Lloyd, 1987] Lloyd, John Wylie 1987. *Foundations of Logic Programming*. Springer-Verlag, second edition.

[McCarthy, 1959] McCarthy, John 1959. Programs with common sense. In *Proc. Symposium on Mechanisation of Thought Processes 1*, London.

[Megiddo, 1983] Megiddo, N. 1983. Towards a genuinely polynomial algorithm for linear programming. *SIAM Journal on Computing* 12(?):347–353.

[Meiri, 1991] Meiri, Itay 1991. Combining qualitative and quantitative constraints in temporal reasoning. In *Proceedings of the Ninth National Conference on Artificial Intelligence*. American Association for Artificial Intelligence, AAAI Press, The MIT Press. 260–267.

[Parikh, 1983] Parikh, Rohit 1983. The problem of vague predicates. In Cohen, and Wartofsky, , editors 1983, *Language, Logic, and Method*. Reidel Publishers. 241–261.

[Passner and Lee, 1991] Passner, jeffrey E. and Lee, Robert R. 1991. Use of an expert system to predict thunderstorms and severe weather. *SPIE, Applications of Artificial Intelligence* 1468:2–10.

[Sacks, 1987] Sacks, Elisha 1987. Hierarchical reasoning about inequalities. In *Proceedings of the Sixth National Conference on Artificial Intelligence*. American Association for Artificial Intelligence, Morgan Kaufmann Publishers, Inc. 649–654.

[Schrijver, 1986] Schrijver, Alexander 1986. *Theory of Linear and Integer Programming.* John Wiley and Sons.

[Shafer and Pearl, 1990] Shafer, Glenn and Pearl, Judea, editors 1990. *Readings in Uncertain Reasoning.* Morgan Kaufmann Publishers, Inc.

[Shahar *et al.*, 1992] Shahar, Yuval; Tu, Samson W.; and Musen, Mark A. 1992. Knowledge acquisition for temporal-abstraction mechanisms. *Knowledge Acquisition* 4:217–236.

[Ullman, 1988] Ullman, Jeffrey D. 1988. *Principles of Database and Knowledge-base Systems*, volume I. Computer Science Press.

[Ullman, 1989] Ullman, Jeffrey D. 1989. *Principles of Database and Knowledge-base Systems*, volume II: The New Technologies. Computer Science Press.

[Vardi, 1993] Vardi, Moshe Y. 1993. personal communication.

[Vilain *et al.*, 1990] Vilain, M.; Kautz, Henry A.; and van Beek, Peter 1990. Constraint propagation algorithms for temporal reasoning: A revised report. In Weld, Daniel S. and Kleer, Johande, editors 1990, *Readings in Qualitative Reasoning about Physical Systems.* Morgan Kaufmann Publishers, Inc. chapter 4.3, 373–381.

[Weld and de Kleer, 1990] Weld, Daniel S. and Kleer, Johande 1990. *Readings in Qualitative Reasoning about Physical Systems.* Morgan Kaufmann Publishers, Inc.

[Williams, 1988] Williams, Brian C. 1988. Minima: A symbolic approach to qualitative algebraic reasoning. In *Proceedings of the Seventh National Conference on Artificial Intelligence.* American Association for Artificial Intelligence, Morgan Kaufmann Publishers, Inc.

[Zadeh, 1983] Zadeh, Lotfi A. 1983. Commonsense and fuzzy logic. In Cercone, N. and McCalla, G., editors 1983, *The Knowledge Frontier: Essays in the Representation of Knowledge.* New York: Springer-Verlag. chapter 5, 103–136.

[Zadeh, 1988] Zadeh, Lotfi A. 1988. Fuzzy logic. *Computer* 21(4):83–93.