Hierarchical Constraint Logic Programming (Ph.D. Dissertation)

Molly Ann Wilson

Technical Report 93-05-01

Dept. of Computer Science and Engineering
University of Washington

May 1993

University of Washington

Abstract

Hierarchical Constraint Logic Programming

by Molly Ann Wilson

Chairperson of the Supervisory Committee: Professor Alan H. Borning
Department of Computer Science and Engineering

A constraint describes a relation to be maintained; it states what the relationship is as opposed to how to maintain it. In many applications, such as interactive graphics, planning, document formatting, and decision support, one needs to express *preferences* as well as strict requirements. Such constraints are sometimes called *soft* constraints; the required ones are called *hard* constraints. We allow an arbitrary number of levels of preference, each successive level being more weakly preferred than the previous one. A collection of constraints at various levels of preference is known as a *constraint hierarchy*. Constraint Logic Programming (CLP) is a general scheme for extending logic programming to include constraints. It is parameterized by $\mathcal{D}$, the domain of the constraints. However, CLP($\mathcal{D}$) languages, as well as most other constraint systems, only allow the programmer to specify constraints that must hold. If we wish to make full use of the constraint paradigm, we need ways to represent these defaults and preferences declaratively, as constraints, rather than encoding them in the procedural parts of the language. We describe a scheme for extending CLP($\mathcal{D}$) to include both required and preferential constraints. We present a theory of constraint hierarchies, and an extension, Hierarchical Constraint Logic Programming, of the CLP scheme to include constraint hierarchies. We give an operational, model theoretic and fixed-point semantics for the HCLP scheme. Finally, we describe two interpreters we have written for instances of the HCLP scheme, give example programs, and discuss related work.

TABLE OF CONTENTS

Page

List of Figures iv

Chapter 1: Introduction 1
1.1 What is a Constraint 1
1.2 What is a Constraint Hierarchy 2
1.3 What is Logic Programming 3
1.4 Why Add Constraints to Logic Programming 4
1.5 Overview of the Dissertation 4

Chapter 2: Constraints and Constraint Hierarchies 6
2.1 Definitions 6
2.1.1 Error Functions 7
2.1.2 Combining Functions 7
2.1.3 Solutions to Constraint Hierarchies 8
2.2 A Brief Example 9
2.3 Comparators 9
2.4 Examples 11
2.5 Remarks on the Comparators 14
2.5.1 Errors for Inequalities 15
2.5.2 Existence of Solutions 16
2.5.3 Disorderly Aspects of Comparators 17

Chapter 3: Extensions to the Constraint Hierarchy Theory 19
3.1 Read-only Variables in Constraint Hierarchies 19
3.1.1 Blocked Hierarchies 23
3.1.2 Illustrative Examples of Using Read-only Annotations 26
3.1.3 Practical Examples of Using Read-only Annotations 27
3.1.4 Circularities 28
3.2 Write-only Annotations 29
3.3 Inter-Hierarchy Comparison 30

3.3.1 Extended Definitions 31

Chapter 4: Logic Programming, Constraint Logic Programming, and Hierarchical Constraint Logic Programming 33
4.1 Logic Programming 33
4.1.1 Definitions and Syntax 33
4.1.2 Semantics 35
4.1.3 Prolog — an Example 37
4.2 Constraint Logic Programming 38
4.2.1 CLP($\mathcal{R}$)— an Example 39
4.3 Putting It All Together: Hierarchical Constraint Logic Programming 40
4.3.1 Adding the Constraint Hierarchy to Logic Programming 40
4.3.2 HCLP 41
4.3.3 HCLP($\mathcal{R}$) 41

Chapter 5: Semantics 44
5.1 Operational Semantics of HCLP 44
5.2 A Model Theory for HCLP 47
5.2.1 Review of CLP Model Theory 47
5.2.2 An Extended Model 48
5.2.3 Comparators as Preference Relations 51
5.2.4 Mapping the Extended Model to a Standard Model 51
5.2.5 A Model for Inter-Hierarchy Comparison 52
5.3 A Fixed-Point Semantics 53
5.3.1 A Fixed-Point Semantics for Inter-Hierarchy Comparison 54
5.4 Relations between the Operational, Model-theoretic, and Fixed-Point Semantics of HCLP 55

Chapter 6: Implementation 63
6.1 Constraint Satisfaction Algorithms 63
6.2 Algorithms for Linear Equality and Inequality Constraints 64
6.2.1 A Recursive Definition 65
6.2.2 An Example 66

6.2.3 The Algorithm Itself . . . 67
6.3 Other Algorithms . . . 70
6.4 Implementation of HCLP($\mathcal{R}$) . . . 71
6.4.1 A Simple Interpreter for HCLP($\mathcal{R}, \mathcal{LPB}$) . . . 71
6.4.2 A DeltaStar-Based Interpreter for HCLP($\mathcal{R}, \star$) . . . 72

Chapter 7: Applications . . . 78
7.1 Interactive Graphics Examples . . . 79
7.2 Planning and Scheduling . . . 83
7.3 Document Formatting . . . 85
7.4 Which Comparator to Use? . . . 86
7.5 Inter-Hierarchy Comparison in HCLP($\mathcal{R}$) . . . 87

Chapter 8: Related Work . . . 91
8.1 Constraint Logic Programming Languages . . . 91
8.2 Other Constraint Languages . . . 94
8.3 Applications . . . 94
8.4 Reasoning . . . 95

Chapter 9: Conclusion . . . 98
9.1 Contributions . . . 98
9.2 Future Research . . . 99

Bibliography . . . 101

Appendix A: An Algorithm for Interpreting HCLP Programs . . . 108

LIST OF FIGURES

Number		*Page*
6-1	Basic algorithm	68
6-2	Incremental algorithm—Adding a constraint at level 2	69
6-3	Disjoint Subproblems	69
6-4	Local Comparator Algorithm	70
7-1	Moving an endpoint of a horizontal line	79
7-2	Moving an endpoint of an anchored horizontal line	80

ACKNOWLEDGMENTS

I am deeply indebted to my advisor, Alan Borning, for his continued support, and for his belief in my ability to be both a good researcher and a good parent. Bjorn Freeman-Benson has collaborated with us on constraint hierarchies and constraint satisfaction algorithms throughout the project, and Amy Martindale and Michael Maher worked with us on the original version of HCLP. Joxan Jaffar and Pascal Van Hentenryck gave valuable suggestions and advice, particularly with the formal semantics aspects of HCLP. Anonymous referees for the article "Hierarchical Constraint Logic Programming" (to appear in the *Journal of Logic Programming*) provided particularly useful and detailed recommendations and suggestions for improving both the content and readability of this work. Michael Sannella, Steve Hanks, and Dan Weld made many useful comments on drafts of this dissertation. Thanks to all for their help. I am especially grateful to my family for their patience and understanding — my husband Andy whose response to my wanting to get a Ph.D. was to propose marriage, and Nathaniel and Calla Rose who, both prenatally and postnatally, endured the stresses of qualifying exams, interviews, and deadlines. I never could have done it without their help.

This research was supported in part by the National Science Foundation under Grant No. CCR-9107395 and by a fellowship from Apple Computer.

Chapter 1

Introduction

Many problems in computer science involve keeping track of relations. These may be relations among graphical objects on a display, arithmetic relations among numbers, information about which nodes in a graph are adjacent to each other, or requirements about job scheduling, to name just a few. These relationships can quickly grow complex as the number of objects in a system increases. Frequently, these problems can be solved more easily when these relations can be represented declaratively: when there is a language that can capture the underlying constraints on the objects without depending on *how* the actual relations are to be solved. The notion of a constraint language arises from this desire for abstraction. Constraints provide an elegant means of stating relationships and of declaratively characterizing properties among objects that can be maintained by an underlying system.

1.1 What is a Constraint

A constraint describes a relation that should be satisfied. Examples of constraints include:

- a constraint that a resistor in a circuit simulation obey Ohm's Law
- a constraint that two views of the same data remain consistent (for example, bar graph and pie chart views)
- a default constraint that parts of an object being edited remain fixed, unless there is some stronger constraint that forces them to change.

A constraint describes a relation to be maintained; it states what the relationship is as opposed to how to maintain it. There are many advantages to stating such relations declaratively: it frees the user from having to write procedures, and a single constraint succinctly represents many operations because in general constraints are multi-directional. As an example, the constraint $A + B = C$ allows the value of any one of A, B, or C to be determined from the other two. It also describes a predicate that will be true whenever $A + B = C$ and false otherwise. In addition, constraints are modular. The constraint $B = D$, taken in conjunction with the previous one, sets up dependencies among A, B, C, and D, but a user can simply state the two local constraints and let the underlying system be responsible for ensuring that both are satisfied.

Of course, we must be careful to avoid putting so much power into the constraints that there is no solver, or no adequately efficient solver, to maintain the relations specified by the constraints. This issue will be discussed further in Chapter 6 Section 6.1.

1.2 What is a Constraint Hierarchy

Many applications of constraints either need, or would benefit from, support for default and preferential constraints, as well as required ones. Such constraints are sometimes called *soft* constraints; the required ones are called *hard* constraints. The required constraints must hold. The system should try to satisfy the preferential or soft constraints if possible, but no error condition arises if it can't. We allow an arbitrary number of levels of preference, each successive level being more weakly preferred than the previous one. A collection of constraints at various levels of preference is known as a *constraint hierarchy*. A more formal definition of a constraint hierarchy is given in Chapter 2 Section 2.1.

Thus, in the $A + B = C$ example, we could also include weak constraints that A and B remain unchanged, and a weaker constraint that C remain the same. Given this hierarchy, if we edit A, the system will change C rather than B to re-satisfy the constraints.

Another example of the usefulness of soft constraints arises in an interactive graphics application where we want to state a default that objects in the picture remain stationary during editing, unless there is some constraint or user edit that forces them to move. In a scheduling application, to cite another example, some constraints might be requirements, while others would be only preferences (such as not scheduling a meeting too early in the morning). In a screen layout application, it might be required that two windows be aligned vertically, but only preferred

that they be at the top of the screen.

Some of the preferences may be stronger than others. For example, it might be strongly preferred that the meeting last an hour, but only weakly preferred that it be at 9:00 am. Also, if a preferential constraint cannot be satisfied, we may still wish to satisfy it as well as possible, using some error metric, rather than simply ignoring it if it can't be satisfied completely. In the interactive graphics application mentioned above, if an object must move during editing, we probably wish it to move as little as possible, rather than arbitrarily hurtling off the screen.

In much of the previous applications-oriented work in this area (e.g., [Borning 81, Gosling 83]), soft constraints were encoded procedurally in the constraint satisfier. If we wish to make full use of the constraint paradigm, it is important to represent these defaults and preferences declaratively, as constraints, rather than encoding them in the procedural parts of the language. Introducing the notion of the constraint hierarchy has enabled us to enlarge the domain of problems that are conducive to being solved via the constraint paradigm.

1.3 What is Logic Programming

Logic programming is a programming paradigm based on a subset of first order logic. Logic programs are declarative rather than procedural in that they state the logical relationships necessary to solve the problem at hand. The underlying system is responsible for making the logical inferences that allow computation to occur. Users can pose queries to the system which then uses the rules supplied by the programmer to answer the queries. In general, logic programs can be used either to prove an assertion, or to determine which set of variable bindings make a query true. In the former case, the variables in the original query are bound and can therefore be considered as input. In the latter case they are unbound and can be viewed as output variables.

The declarative property of logic programming languages not only makes them appealing to programmers, it also means that these languages have a clean and well-defined semantics. The meaning of logic programming languages is derived from the semantics of logic itself, and is therefore well founded.

1.4 Why Add Constraints to Logic Programming

Constraints seem to be a natural extension to the logic programming framework, as they share many of the properties discussed above. They are declarative; they are multidirectional; what constitutes a solution to a set of constraints is well defined for many domains. In fact Jaffar and Lassez [Jaffar & Lassez 87] showed that pure Prolog can be viewed as an instance of a more general Constraint Logic Programming (CLP) scheme. This paradigm extends the notion of logic programming to include constraints, and overcomes the limitations of some logic programming languages while retaining the clean semantics that have characterized languages in this family.

The next step seemed to be to add the notion of a *constraint hierarchy*, as well as simple hard constraints to the logic programming framework. This can be viewed in two complementary ways. Firstly, constraint hierarchies in and of themselves are not a programming language. Logic programming, because of its similarity to constraints, seemed a good choice for a wiring language, i.e. a language with the power of recursion and conditionals that could be used to incorporate the hard and soft constraints. Secondly, CLP had extended and generalized the logic programming scheme. Why not similarly extend the CLP scheme to include levels of constraints? Thus two related notions led to the development of Hierarchical Constraint Logic Programming (HCLP), a family of languages that combine hard and soft constraints within the logic programming framework.

1.5 Overview of the Dissertation

This dissertation goes on to discuss HCLP, its components, its semantics, an implementation of an HCLP language, and various applications. The HCLP family of languages was introduced in [Borning et al. 89]. Reference [Borning et al. 92] discusses constraint hierarchy theory and applications. A general algorithm for solving constraint hierarchies was given in [Freeman-Benson et al. 92]. Reference [Wilson & Borning 89] explores the nonmonotonic properties of HCLP. Portions of this thesis appear in [Wilson & Borning 93].

- Chapter 2 gives formal definitions for constraints, constraint hierarchies, and solutions to constraint hierarchies. It includes examples of using constraint hierarchies and discusses relations that hold among various comparators, i.e. methods of comparing solutions to constraint hierarchies.

- Chapter 3 discusses some extensions to the constraint hierarchy theory including read-only and write-only annotations, as well as inter-hierarchy comparison. Reading this chapter is not necessary for understanding the remainder of the dissertation, with the exception of Section 7.5.

- Chapter 4 gives a brief introduction to Logic Programming, and includes a small program written in Prolog. Then Constraint Logic Programming is introduced, and a brief CLP($\mathcal{R}$) program is given. Finally, we discuss Hierarchical Constraint Logic Programming, and extend the CLP($\mathcal{R}$) program to include hierarchies.

- Chapter 5 presents the operational and declarative semantics of Hierarchical Constraint Logic Programming and gives results that demonstrate the equivalence of the logical, fixed-point, and operational semantics of the HCLP languages.

- Chapter 6 discusses two implementations of HCLP languages. The DeltaStar algorithm for solving constraint hierarchies is introduced, and its incorporation into the current HCLP($\mathcal{R}$) interpreter is explained.

- Chapter 7 presents a number of examples of HCLP programs. It also discusses the pros and cons of using inter-hierarchy comparison in HCLP($\mathcal{R}$) programs.

- Chapter 8 discusses related work. Specifically, we relate HCLP to general constraint logic programming languages, other constraint languages, some applications, and research in artificial intelligence.

- Chapter 9 summarizes the contributions of the work in HCLP and discusses some directions for further research.

Chapter 2

Constraints and Constraint Hierarchies

In this chapter, formal definitions for constraints, constraint hierarchies, and solutions to constraint hierarchies are presented. Then we give some examples of using constraint hierarchies. Finally, we discuss relations that hold among various comparators, i.e. methods of comparing solutions to constraint hierarchies.

2.1 Definitions

A constraint is a relation over some domain $\mathcal{D}$ (e.g. integers, booleans, finite domains). The domain $\mathcal{D}$ determines the constraint predicate symbols $\Pi_{\mathcal{D}}$ of the language, which must include $=$. A constraint is thus an expression of the form $p(t_1, \ldots, t_n)$ where p is an n-ary symbol in $\Pi_{\mathcal{D}}$ and each t_i is a term. A *labeled constraint* is a constraint labeled with a strength, written lc, where l is a strength and c is a constraint. The strengths are totally ordered.

A *constraint hierarchy* is a finite set of labeled constraints. Given a constraint hierarchy H, H_0 is a vector of the required constraints in H, in some arbitrary order, with their labels removed. H_1 is a vector of the constraints in H at the strongest non-required level, and so forth through the weakest constraints H_n, where n is the number of non-required levels in the hierarchy. We also define $H_k = \emptyset$ for $k > n$.

A valuation for a set of constraints is a function that maps the free variables in the constraints

to elements in the domain $\mathcal{D}$ over which the constraints are defined. A *solution* to a constraint hierarchy is a set of valuations for the free variables in the hierarchy. We require any valuation in the solution set to satisfy at least the required constraints. In addition, the solution set contains those valuations that satisfy the non-required constraints at least as well as any other valuation that also satisfies the required constraints. In other words, there is no valuation satisfying the required constraints that is "better" than any valuation in the solution. There are a number of reasonable methods for comparing valuations to determine which is better. We call such methods *comparators*. In the following sections we give formal definitions for the solution to a constraint hierarchy and for various comparators.

2.1.1 Error Functions

In order to compare valuations, we will need some measure of how well a particular valuation satisfies a given constraint. The error function $e(c\theta)$ indicates how nearly constraint c is satisfied for a valuation θ. This function returns a non-negative real number and must have the property that $e(c\theta) = 0$ if and only if $c\theta$ holds. ($c\theta$ denotes the result of applying the valuation θ to c.) For any domain $\mathcal{D}$, we can use the trivial error function that returns 0 if the constraint is satisfied and 1 if it is not. A comparator that uses this error function is a *predicate* comparator. For a domain that is a metric space, in place of the trivial error function, we can define an error function by using the domain's metric. For example, the error for $X = Y$ would be the distance between X and Y. Such a comparator is a *metric* comparator. Because the definition of a specific comparator depends on the error function used, metric comparators are domain dependent.

The error function $\mathbf{E}(C\theta)$ maps e over a vector of constraints $C = [c_1, \ldots, c_k]$:

$$\mathbf{E}(C\theta) \quad = \quad [e(c_1\theta), \ldots, e(c_k\theta)]$$

An *error sequence* is a vector $[\mathbf{E}(H_1\theta), \mathbf{E}(H_2\theta), \ldots, \mathbf{E}(H_n\theta)]$.

Finally, the error v_i for the i^{th} constraint can be weighted by a weight w_i. Each weight is a positive real number.

2.1.2 Combining Functions

Some of the comparators that we are interested in will first combine the errors at a given level in the hierarchy before comparing valuations. We now introduce the notion of a *combining function*, g, that is applied to real-valued vectors and that returns some value that can be compared using

the associated relations $<>_g$ and $<_g$. For example, g may sum a vector of numbers, or select the maximum of a vector of numbers. We require $<_g$ to be irreflexive, antisymmetric, and transitive. We require $<>_g$ to be reflexive and symmetric. (We use the notation $<>_g$ rather than $=$ because, for some of the comparators, the relation is not transitive. The symbol $<>_g$ indicates that two valuations cannot be ordered using $<_g$. For some comparators, this will be because they are equal; for others, because they are incomparable.)

The combining function $\mathbf{G}$ is a generalization of g that is applied to error sequences and that returns a sequence of values that can be compared using $<>_g$ and $<_g$. Such a sequence is a *combined error sequence*. Let $R = [\mathbf{E}(H_1\theta), \ldots, \mathbf{E}(H_n\theta)]$. Then

$$\mathbf{G}(R) \quad = \quad [g(\mathbf{E}(H_1\theta)), \ldots, g(\mathbf{E}(H_n\theta))]$$

A lexicographic ordering $<\mathbf{g}$ can be defined on combined error sequences $u_1, \ldots, u_n$ and $w_1, \ldots, w_n$ in the standard way:

$$\begin{aligned} u_1, \ldots, u_n \quad <\mathbf{g} \quad & w_1, \ldots, w_n \text{ if} \\ & \exists k \in 1 \ldots n \text{ such that} \\ & \forall i \in 1 \ldots k-1 \; u_i \; <>_g \; v_i \; \wedge \\ & u_k \; <_g \; v_k \end{aligned}$$

2.1.3 Solutions to Constraint Hierarchies

Finally, we can define the solution set S to a constraint hierarchy H, by using the comparator defined by the combining function g, its associated function $\mathbf{G}$, and the lexicographic ordering defined by $<\mathbf{g}$.

$$\begin{aligned} S_0 \quad &= \quad \{\theta \mid \forall c \in H_0 \; e(c\theta) = 0\} \\ S \quad &= \quad \{\theta \mid \theta \in S_0 \wedge \forall \sigma \in S_0 \\ & \qquad \neg(\mathbf{G}([\mathbf{E}(H_1\sigma), \ldots, \mathbf{E}(H_n\sigma)]) <\mathbf{g}\; \mathbf{G}([\mathbf{E}(H_1\theta), \ldots, \mathbf{E}(H_n\theta)]))\} \end{aligned}$$

S_0 is the set of solutions to the required constraints (ignoring the soft constraints). The desired set S is all valuations in S_0 for which no better valuations in S_0 exist, where better is determined using the lexicographic ordering defined by $<\mathbf{g}$.

2.2 A Brief Example

Before we give definitions for various comparators, a brief example will help to solidify the notion of a solution to a constraint hierarchy.

Let us consider the following simple constraint hierarchy over the domain of the reals:

$$\begin{aligned} \textit{required} \quad & X > 0 \\ \textit{strong} \quad & X < 10 \\ \textit{weak} \quad & X = 4 \end{aligned}$$

The set S_0 consists of all valuations that map X to a positive real number. The solution set S consists of the single valuation that maps X to 4. Let us call this valuation θ. Consider the valuation σ that maps X to 5. Then $e((X < 10)\theta)$ is 0. $e((X < 10)\sigma)$ is also 0. $\mathbf{E}([(X < 10)\theta])$ is [0]. (There is only one constraint at the *strong* level.) $\mathbf{E}([(X < 10)\sigma])$ is also [0]. $e((X = 4)\theta)$ is 0. $e((X = 4)\sigma)$ is 1. $\mathbf{E}([(X = 4)\theta])$ is [0]. $\mathbf{E}([(X = 4)\sigma])$ is [1]. The combined error sequence $\mathbf{G}(\mathbf{E}([(X < 10)\theta]), \mathbf{E}([(X = 4)\theta]))$ evaluates to $[[0],[0]]$. (Again, there is only one constraint at each level in the hierarchy, so the combining function has no effect.) The combined error sequence $\mathbf{G}(\mathbf{E}([(X < 10)\sigma]), \mathbf{E}([(X = 4)\sigma]))$ evaluates to $[[0],[1]]$. Since $[[0],[0]] <_{\mathbf{g}} [[0][1]]$, σ is not in S. Moreover, there is no valuation in S_0 that is less than $[[0][0]]$ in the lexicographic order defined by any $<_{\mathbf{g}}$ where $<_g$ and $<>_g$ have the properties defined above. So θ is in S.

2.3 Comparators

We now define a number of comparators, each of which gives rise to a different way of defining the set of solutions to a constraint hierarchy. We can classify types of comparators (as opposed to defining a specific comparator) as either *global*, *local*, or *regional*. Since the error sequences for the constraints at levels $H_1, \ldots, H_n$ are being compared using a lexicographic ordering, if a solution θ is better than a solution σ, there is some level k in the hierarchy such that for $1 \leq i < k$, $g(\mathbf{E}(H_i\theta)) <>_g g(\mathbf{E}(H_i\sigma))$, and at level k, $g(\mathbf{E}(H_k\theta)) <_g g(\mathbf{E}(H_k\sigma))$.

For a local comparator, each constraint is considered individually. Solution θ must do exactly as well as σ for each constraint in levels $1 \ldots k-1$, and at level k, θ must do at least as well as σ for all constraints, and strictly better for at least one. For a global comparator, the errors for all constraints at a given level are aggregated using g. For a regional comparator, each constraint at a given level is considered individually (as with a local comparator). However, unlike a local

comparator, two solutions that are incomparable at strong levels may still be compared at weaker levels and one discarded, so that a regional comparator will, in general, discriminate more than a local one.

We now define a number of useful classes of comparators, by defining the combining function g and the relations $<>_g$ and $<_g$ for each. Each of these classes defines some number of actual comparators by specifying the error function.

Weighted-sum-better, *worst-case-better*, and *least-squares-better* are global comparators, in which the constraint errors at a given level are combined by taking the weighted sum, the weighted maximum, and weighted sum of the squares respectively. *Locally-better* and *regionally-better* are local and regional comparators, respectively.

For weighted-sum-better, $g(\mathbf{v}) = \sum_{i=1}^{|\mathbf{v}|} w_i v_i$, $<_g$ is defined as for the reals, and $<>_g$ is equivalent to $=$ for the reals.

For worst-case-better, $g(\mathbf{v}) = \max\{w_i v_i \mid 1 \leq i \leq |\mathbf{v}|\}$, $<_g$ is defined as for the reals, and $<>_g$ is equivalent to $=$ for the reals.

For least-squares-better, $g(\mathbf{v}) = \sum_{i=1}^{|\mathbf{v}|} w_i v_i^2$, $<_g$ is defined as for the reals, and $<>_g$ is equivalent to $=$ for the reals.

For locally-better, $g(\mathbf{v}) = \mathbf{v}$ and $<>_g$ and $<_g$ are defined as follows:

$$\mathbf{v} <_g \mathbf{u} \equiv \forall i \ v_i \leq u_i \wedge \exists j \text{ such that } v_j < u_j$$
$$\mathbf{v} <>_g \mathbf{u} \equiv \forall i \ v_i = u_i$$

For regionally-better, $g(\mathbf{v}) = \mathbf{v}$ and $<>_g$ and $<_g$ are defined as follows:

$$\mathbf{v} <_g \mathbf{u} \equiv \forall i \ v_i \leq u_i \wedge \exists j \text{ such that } v_j < u_j$$
$$\mathbf{v} <>_g \mathbf{u} \equiv \neg((\mathbf{v} <_g \mathbf{u}) \vee (\mathbf{u} <_g \mathbf{v}))$$

Orthogonal to the choice of a global, local, or regional combining function, we can choose an appropriate error function for the constraints. *Locally-predicate-better* (LPB) is locally-better using the trivial error function that returns 0 if the constraint is satisfied and 1 if it is not. *Locally-metric-better* (LMB) is locally-better using a domain metric in computing the constraint errors. *Weighted-sum-predicate-better*, *weighted-sum-metric-better*, and so forth, are all defined analogously.

Unsatisfied-count-better is a special case of *weighted-sum-predicate-better*, using weights of 1 on each constraint; it counts the number of unsatisfied constraints in making its comparisons. The predicate versions of the other two global comparators aren't particularly useful: *worst-case-predicate-better* has an all-or-nothing behavior which doesn't filter out solutions as well as

one might like; and *least-squares-predicate-better* always gives the same results as weighted-sum-predicate-better (since $1^2 = 1$). Hereafter, we will use the more concise worst-case-better (WCB) and least-squares-better (LSB) to refer to the metric versions of these two global comparators.

We will use the terminology $better(\theta, \sigma, H)$ to signify that $\mathbf{G}([\mathbf{E}(H_1\theta), \ldots, \mathbf{E}(H_n\theta)]) <_{\mathbf{G}} \mathbf{G}([\mathbf{E}(H_1\sigma), \ldots, \mathbf{E}(H_n\sigma)])$.

All of the comparators are irreflexive:

$$\forall\theta \forall H \ \neg better(\theta, \theta, H)$$

Most of the comparators are transitive:

$$\forall\theta, \sigma, \tau \, \forall H \ better(\theta, \sigma, H) \wedge better(\sigma, \tau, H) \rightarrow better(\theta, \tau, H)$$

The regional comparators are not transitive, as the corresponding $<>_g$ relation is not transitive. However, for any comparator where $<>_g$ is transitive, then the corresponding better will also be transitive.

While there are many plausible candidates for comparators, we insist that *better* respect the hierarchy—if there is some valuation in S_0 that completely satisfies all the constraints through level k, then all valuations in S must satisfy all the constraints through level k:

$$\begin{aligned} &\text{if } \exists\theta \in S_0 \wedge \exists k > 0 \text{ such that} \\ &\quad \forall i \in 1 \ldots k \ \forall p \in H_i \ \ p\theta \text{ holds} \\ &\quad \text{then } \forall\sigma \in S \ \forall i \in 1 \ldots k \ \forall p \in H_i \ \ p\sigma \text{ holds} \end{aligned}$$

2.4 Examples

As a simple example to illustrate some of the differences among the comparators, consider a constraint-based spreadsheet, or a graphical calculator such as was described in [Borning 81]. Suppose there is a "sum" constraint relating real-valued variables `A`, `B`, and `C`. Previously, the values for these variables were `A=2`, `B=3`, and `C=5`. The user has just edited `C` to be `7`. The following constraint hierarchy expresses the desired semantics:

$$\begin{aligned} \textit{required} \quad & C = A + B \\ \textit{strong} \quad & C = 7 \\ \textit{weak} \quad & A = 2 \\ \textit{weak} \quad & B = 3 \end{aligned}$$

The *required* $C = A + B$ constraint represents the sum constraint. The *strong* $C = 7$ constraint represents the user's edit. (Making this constraint a strong preference rather than a requirement allows the system to refuse to accept the edit if it conflicts with some required constraint; if instead we wished to be notified of a failure in this case we would make the edit also *required*.) The two constraints *weak* $A = 2$ and *weak* $B = 3$ express a desire that the rest of the system be changed as little as possible in accommodating the edit to C. Without them, $A = 1000000$, $B = -999993$, and $C = 7$ would be a perfectly valid result.

We now list the solutions for a number of the comparators.

Locally-predicate-better yields two solutions:

$A = 2,\ B = 5,\ C = 7$

$A = 4,\ B = 3,\ C = 7$

In the first solution, the $A = 2$ constraint is satisfied but not $B = 3$; in the second, $B = 3$ is satisfied but not $A = 2$.

Locally-metric-better yields an infinite number of solutions:

$A = x,\ B = 7 - x,\ C = 7 \quad \text{for all } x \in [2 \ldots 4]$

None of the solutions in the set is better than any other in the set. For example, the solution $A = 2.9$, $B = 4.1$, $C = 7$ doesn't satisfy the constraint on A as well as $A = 2$, $B = 5$, $C = 7$, but does better for the constraint on B. However, outlying solutions such as $A = 1000000$, $B = -999993$, and $C = 7$ are ruled out.

Weighted-sum-predicate-better yields the same two solutions as locally-predicate-better if the weights on the two *weak* constraints are equal; otherwise it picks one solution or the other depending on which weight is larger.

Weighted-sum-metric-better yields the same infinite set of solutions as locally-metric-better if the weights on the two *weak* constraints are equal; otherwise it picks either $A = 2$, $B = 5$, $C = 7$, or $A = 4$, $B = 3$, $C = 7$ respectively, depending on whether the weight on the constraint on A or on B is larger.

Least-squares-better yields a single solution, which is $A = 3$, $B = 4$, $C = 7$ when the weights on the *weak* constraints are equal. (This is also the solution for worst-case-metric better with equal weights.)

For this example, the regional comparators yield the same solutions as their local counterparts.

Now consider a simple scheduling problem in which the regionally-metric comparator does give a different answer than locally-metric-better. In this example, we have three people who wish

to get together for a meeting. It is crucial that two of those people, Bjorn and Michael, attend the meeting, so their wishes will have higher priority than the third person, Alan, who wants to sit in on the meeting, but whose attendance is not absolutely necessary. Bjorn and Michael agree that their requested meeting times are preferential, rather than required constraints, as they know that it is vital that this meeting take place. The meeting will last about an hour. Bjorn would like to be finished by noon. Michael would prefer to meet after work — at 5:00 p.m. Alan decides that 3:00 p.m. is the best time for him. The following constraint hierarchy reflects these preferences:

strong	Bjorn:	meet before or at `11:00 a.m.`
strong	Michael:	meet at or after `5:00 p.m.`
weak	Alan:	meet at `3:00 p.m.`

The weight on each of these constraints is 1. Note that there are no required constraints. If we made Bjorn's and Michael's requests required, then there would be no solutions. Alan's request is weak because it is not crucial that he attend the meeting.

Here is a list of solutions for various comparators.

Locally-predicate-better yields two answers — either meet at or before 11:00 a.m. as Bjorn prefers, or meet at or after 5:00 p.m., as Michael prefers.

As with the previous example, locally-metric-better yields an infinite number of solutions: any hour between 11:00 a.m. and 5:00 p.m. inclusive. As in the case of the locally-predicate solutions, Alan's weak desires have no bearing on the final meeting time.

Regionally-predicate-better has the same solutions as locally-predicate-better. (Note that if Alan wished to meet before 11:00 a.m., say at 9:00 a.m., then regionally-predicate-better would select 9:00 a.m. as the solution, whereas the solutions given by the locally-predicate-better comparator would favor 9:00 a.m. over all other morning times, but not over any of the evening times. Similarly, if Alan wished to meet at a time that was compatible with Michael, say at 5:30 p.m., then regionally-predicate-better would yield 5:30 as the meeting time.)

Regionally-metric-better yields the single solution of having the meeting at 3:00 p.m. In this example, the regional comparator was able to further discriminate based on preferences weaker in the hierarchy. This highlights the difference between the regional and local comparators.

Weighted-sum-predicate-better gives the same solutions as locally-predicate-better, before or at 11:00 a.m. and at or after 5:00 p.m. Weighted-sum-metric-better gives the same 3:00 p.m. solution as does regionally-metric-better. Least-squares-better and worst-case-better both yield

a meeting time of 2:00 p.m. For both of these latter cases, the weak constraint has no effect on the solution.

2.5 Remarks on the Comparators

The definitions of the global comparators include weights on the constraints. For the local comparators, adding weights would be futile, since the result would be the same with or without the weights.

One might argue that allowing an arbitrary number of constraint strengths is unnecessary: since soft constraints can have weights on them, one could make do with only two levels (required and one preferential level), and use appropriate weights to achieve the desired effects. There are three reasons we believe such an argument is not valid: two conceptual, and the other pragmatic. To illustrate the first reason, consider moving a line with a mouse in an interactive graphics application. The line has a strong constraint that it be horizontal, and another strong constraint that one endpoint follow the mouse. There is also a weaker constraint that the line be attached to some fixed point in the diagram. The user's expectations in this case are likely that the line will remain exactly horizontal and will precisely follow the mouse (letting the weaker attachment constraint be unsatisfied), rather than keeping the line nearly horizontal, or quite close to the mouse, but letting the weaker constraint have a bit of influence on the result. Second, since adding weights to constraints is futile for the local comparators, we would need to give up these comparators and use only global ones. Third, solutions to constraint hierarchies in which one level completely dominates the next can often be found much more efficiently than solutions to systems with only one preferential level and weights on the constraints—see Section 6.1.

Most of the concepts in constraint hierarchies derive from concepts in subfields of operations research such as linear programming [Murty 83], multiobjective linear programming [Murty 83], goal programming [Ignizio 85], and generalized goal programming [Ignizio 83]. The domain of the constraints in operations research is usually the real numbers, or sometimes the integers (for integer programming problems). The notion of constraint hierarchies is preceded by the approach to multiobjective problems of placing the objective functions in a priority order. The concept of a *locally-better* solution is derived from the concept of a *vector minimum* (or *pareto optimal solution*, or *nondominated solution*) to a multiobjective linear programming problem. Similarly, the concepts of *weighted-sum-better* and *worst-case-better* solutions are both derived

from analogous concepts in multiobjective linear programming problems and generalized goal programming.

There are a number of relations that hold between local and global comparators.

Proposition 1 *For a given error function e,*

$$\forall\theta\forall\sigma\forall H \; \textit{locally-better}(\theta,\sigma,H) \rightarrow \textit{weighted-sum-better}(\theta,\sigma,H)$$

Proof: Suppose *locally-better*(θ,σ,H) holds. Then there is some level $k > 0$ in H such that the error after applying θ to each of the constraints through levels $k-1$ is equal to that after applying σ. It then follows that the sum of the weighted errors after applying θ to the constraints through levels $k-1$ is equal to that after applying σ. Furthermore, at level k the error after applying θ is strictly less for at least one constraint and less than or equal for all the rest. This implies that the weighted sum of the errors after applying θ to the constraints at level k is strictly less than that after applying σ. Therefore *weighted-sum-better*(θ,σ,H) also holds. ∎

Corollary 1 *For a given constraint hierarchy, let S_{LB} denote the set of solutions found using the* locally-better *comparator, and S_{WSB} that for* weighted-sum-better. *Then $S_{\text{WSB}} \subseteq S_{\text{LB}}$.*

Proposition 2 *For a given error function e,*

$$\forall\theta\forall\sigma\forall H \; \textit{locally-better}(\theta,\sigma,H) \rightarrow \textit{least-squares-better}(\theta,\sigma,H)$$

The proof is similar to that for Proposition 1.

Corollary 2 *Let S_{LSQ} denote the set S of solutions found using the* least-squares-better *comparator. Then $S_{\text{LSQ}} \subseteq S_{\text{LB}}$.*

Propositions 1 and 2 concern particular instances of the *globally-better* schema. However, *locally-better* does not imply *globally-better* for an arbitrary combining function g. In particular, *locally-better* does not imply *worst-case-better*.

2.5.1 Errors for Inequalities

A problem arises in connection with metric predicates and strict inequalities. For example, what should be the error function for the constraint $X > Y$, where X and Y are reals? If X is greater than Y, then the error must be 0. If X isn't greater than Y, we'd like the error to be smaller the closer X is to Y. Thus, an obvious error function is $e(X > Y) = 0$ if $X > Y$, otherwise $Y - X$.

This isn't correct, however, since it gives an error of 0 if X and Y are equal. However, if the error when X and Y are equal is some positive number d, then we get a smaller error when Y is equal to $X + d/2$ than when Y is equal to X, thus violating our desire that the error become smaller as X gets closer to Y.

To solve this problem, we introduce an infinitesimal number ϵ [Robinson 66], which is greater than 0 and less than any positive standard real number. Using ϵ we can then define

$$e(X > Y) = \begin{cases} Y - X & \text{if } X < Y \\ \epsilon & \text{if } X = Y \\ 0 & \text{if } X > Y \end{cases}$$

$$e(X \neq Y) = \begin{cases} 0 & \text{if } X \neq Y \\ \epsilon & \text{if } X = Y \end{cases}$$

$$e(X < Y) = \begin{cases} 0 & \text{if } X < Y \\ \epsilon & \text{if } X = Y \\ X - Y & \text{if } X > Y \end{cases}$$

Note that ϵ is only being added to the range of the error function, not to the domain $\mathcal{D}$. If we did try to change the domain itself to be the hyperreal numbers, we would end up with the same problem as before.[1]

2.5.2 Existence of Solutions

If the set of solutions S_0 for the required constraints is non-empty, intuitively one might expect that the set of solutions S for the hierarchy would be non-empty as well. However, this is not always the case. Consider the hierarchy *required* $N > 0$, *strong* $N = 0$ for the domain of the real numbers, using a metric comparator. Then S_0 consists of all valuations mapping N to a positive number, but S is empty, since for any valuation $\{N \mapsto d\} \in S_0$, we can find another valuation, for example $\{N \mapsto d/2\}$, that better satisfies the soft constraint $N = 0$.

However, the following proposition, especially relevant for floating point numbers, does hold:

Proposition 3 *If S_0 is non-empty and finite, and if a transitive comparator is used, then S is non-empty.*

[1] What would be the error for the constraint $0 > \epsilon/2$? According to the definition, the error would be $\epsilon/2$. But this is less than the error for $0 > 0$, even though the $0 > 0$ constraint is more nearly satisfied.

Proof: Suppose to the contrary that S is empty. Pick a valuation θ_1 from S_0. Since $\theta_1 \notin S$, there must be some $\theta_2 \in S_0$ such that $better(\theta_2, \theta_1, H)$. Similarly, since $\theta_2 \notin S$, there is an $\theta_3 \in S_0$ such that $better(\theta_3, \theta_2, H)$, and so forth for an infinite chain $\theta_4, \theta_5, \ldots$. Since *better* is transitive, it follows by induction that $\forall i, j > 0\ [i > j \rightarrow better(\theta_i, \theta_j, H)]$. The irreflexivity property of *better* requires that $\forall i > 0\ \neg better(\theta_i, \theta_i, H)$. Thus all the θ_i are distinct, and so there are an infinite number of them. But, by hypothesis S_0 is finite, a contradiction. ∎

For most (if not all) practical applications of constraint hierarchies, H will be finite. For example, when constraint hierarchies are embedded in programming languages, then if the program terminates, the resulting set of constraints will be finite.[2] The next proposition tells us that in many cases of practical importance, if the required constraints can be satisfied, then solutions to the hierarchy exist.

Proposition 4 *If S_0 is non-empty, and if a predicate comparator is used, then S is non-empty.*

Proof: Suppose to the contrary that S is empty. Using the same argument as before, we show that there must be an infinite number of distinct valuations $\theta_i \in S_0$. However, if the comparator is predicate, one valuation cannot be better than another if both valuations satisfy exactly the same subset of constraints in H. Therefore each of the θ_i must satisfy a different subset of the constraints in H. However, this is a contradiction, since H is finite. ∎

2.5.3 Disorderly Aspects of Comparators

There are times when we would like to consider the effect of adding a constraint to an existing hierarchy, thereby refining the set of valuations that solves the hierarchy. This can occur, for example, if we are trying to solve constraints concurrently; the system would like to begin solving the hierarchy before all of the constraints are known.

Unfortunately, the *orderliness* property defined below does not hold for comparators that respect the hierarchy. Intuitively, this means that adding new constraints to a hierarchy may result in a solution that is not a refinement of the previous solution, but rather a different solution altogether.

Definition. Let H and J be constraint hierarchies. Let $\mathcal{C}$ be a comparator. Then $\mathcal{C}$ is *orderly* if $S_{\{H \cup J\}}(\mathcal{C}) \subseteq S_{\{H\}}(\mathcal{C})$. A comparator that is not orderly is *disorderly*.

[2] Unless we had a language that allowed an infinite set of constraints to be specified by a single statement.

Note that adding only required constraints to an existing set of required constraints will either narrow or leave the same the set of valuations that satisfy those constraints. Thus disorderliness arises only in the case of non-required constraints. This orderliness property is similar to the "stability of rejection" property discussed in [Saraswat 89].

Proposition 5 *Let $\mathcal{D}$ be a nontrivial domain. Then any comparator that respects the hierarchy is disorderly.*

> **Proof:** Let H = {`weak X = a`}, and let J = {`strong X = b`}, where `a` and `b` are two distinct elements in $\mathcal{D}$. Let $\mathcal{C}$ be a comparator that respects the hierarchy. Then $S_{\{H\}}(\mathcal{C})$ consists of the valuation that maps `X` to `a`, and $S_{\{H \cup J\}}(\mathcal{C})$ consists of the valuation that maps `X` to `b`. $S_{\{H \cup J\}}(\mathcal{C}) \not\subseteq S_{\{H\}}(\mathcal{C})$ since `a` and `b` are distinct. Therefore $\mathcal{C}$ is not orderly. ∎

Thus, if we have an incremental solver, adding a new constraint could in general require us to retract a previous solution.

Chapter 3

Extensions to the Constraint Hierarchy Theory

In this chapter, various extensions to the constraint hierarchy theory are explored. The first two extensions involve adding read-only and write-only annotations to the variables in the constraints. The third extension allows for comparing solutions across two or more different hierarchies. Read-only and write-only annotations are not incorporated in the current HCLP theory or implementation, although we believe that they could be in the future. Inter-hierarchy comparison is supported by the current theory (see Chapter 5 Section 5.2.5), but not by the current implementation. Some of the advantages and disadvantages of incorporating it into the implementation are discussed in Chapter 7 Section 7.5.

3.1 Read-only Variables in Constraint Hierarchies

We can roughly classify constraint-based languages and systems as using one of two approaches: the *refinement* model or the *perturbation* model. In both cases constraints restrict the values that variables may take on. In the refinement model, variables are initially unconstrained; constraints are added as the computation unfolds, progressively refining the permissible values of the variables. This approach is more or less universally adopted in the logic programming community, for example, in the CLP language scheme [Cohen 90, Jaffar & Lassez 87] and instances of it (e.g. [Colmerauer 90, Dincbas et al. 88, Heintze et al. 91, Van Hentenryck 89, Jaffar & Michaylov 87]),

and the cc languages [Saraswat et al. 91, Saraswat 89].

In contrast, in the perturbation model, at the beginning of an execution cycle variables have specific values associated with them that satisfy the constraints. The value of one or more variables is perturbed (usually by some outside influence, such as an edit request from the user), and the task of the computer is to adjust the values of the variables so that the constraints are again satisfied. The perturbation model has been often used in constraint-based applications such as the interactive graphics systems Sketchpad [Sutherland 63], ThingLab I [Borning 81], Magritte [Gosling 83], and Juno [Nelson 85], and user interface construction systems such as Garnet [Myers et al. 90a, Myers et al. 90b]. We can also view the ubiquitous spreadsheet as using the perturbation model: formulas are constraints relating the permissible values in cells. Before a user action, cells have values that satisfy the constraints (formulas). The user edits the value in a cell, or edits a formula, and the system must change the values of other cells as needed so that the constraints are again satisfied.

In the perturbation model, there will generally be many ways to update the current state so that the constraints are again satisfied. As a trivial example, suppose we have a constraint $A+B=C$, and edit the value of B. Should we change A, C, both A and C, undo the change to B, or what? These systems often employ *read-only annotations* to help limit this choice. A common special case is to use *one-way constraints*, that is, constraints in which all but one of the variables are declared to be read-only. For the $A+B=C$ constraint, if A and B are declared to be read-only, it is clear what to do when B is edited (at least if there are no circularities in the constraint graph). Spreadsheets are examples of systems using one-way constraints; Garnet is another. However, even in perturbation systems that use multi-way constraints, read-only annotations are often employed. In Sketchpad, for example, the constraint that related a "master" picture and an instance of it was read-only on the master. In ThingLab I read-only annotations were used to force a spreadsheet-like constraint on a column of numbers and their sum to be used in one direction only.

Constraint hierarchies provide an alternative method for specifying this choice, without giving up the generality of multi-way constraints. In a sum constraint similar to the one in Chapter 2 Section 2.4, for example, we could have *medium* constraints that the values for A and B remain the same, while having a *weak* constraint that the value for C remain the same. Then updates to A or B would have the effect of forcing C to change, and in this way we could achieve directionality without using read-only annotations. However, even in a multi-way constraint

system with hierarchies, read-only annotations can still be useful. One use is in constraints that reference an external input device or other outside source of information. If we have a constraint that a point follow the mouse, the constraint should be read-only on the mouse position (unless, of course, the mouse is equipped with a small computer-controlled motor). Another use is in constraints describing a change over time, where the constraint relates an old and a new state. Here, we may wish to make the old state read-only, so that the present can't alter the past.

Intuitively, when choosing the best solutions to a constraint hierarchy, constraints should not be allowed to affect the choice of values for their read-only variables, i.e., information can flow out of the read-only variables, but not into them. (Alternatively we can say that constraints are only allowed to affect the choice of values for their unannotated variables.) However, we still want the constraints to be satisfied if possible (respecting their strengths). In particular, required constraints must be satisfied, even if they contain read-only annotations.

We now give an informal outline of the definition for solutions to constraint hierarchies with read-only annotations. One way of preventing a constraint from affecting the choice of values for a variable is to replace that occurrence of the variable by a constant. Thus, we begin the definition of the set of solutions to a constraint hierarchy H by forming a set Q of constraint hierarchies, where each element of Q is a constraint hierarchy with arbitrary domain elements substituted for the read-only variables. (Note that the same variable v may have read-only occurrences and normal occurrences. Only the read-only occurrences are replaced when forming elements of Q.) Intuitively, we guess a valuation for v, and then form a hierarchy using that guess. After making all possible guesses, we weed out solutions arising from incorrect ones. (Note that this is purely a specification of the meaning of read-only annotations, not a reasonable algorithm for actually solving such constraint hierarchies!)

Here is an example. ($X?$ signifies that this occurrence of X is read-only.)

Original H	*$q \in Q$ formed by replacing $Y?$ with $d \in \mathcal{D}$*			
	$Y? \mapsto 9.83$	$Y? \mapsto 3$	$Y? \mapsto -6.2$	$\cdots$
required $X = Y?$	$X = \mathbf{9.83}$	$X = \mathbf{3}$	$X = \mathbf{-6.2}$	
strong $X = 4$	$X = 4$	$X = 4$	$X = 4$	$\cdots$
weak $Y = 3$	$Y = 3$	$Y = 3$	$Y = 3$	

Next we solve the constraint hierarchies in Q, discarding any valuations that map the remaining unannotated occurrences of a variable to something different from what was substituted for its read-only occurrences. (In other words, we discard all valuations in which we guessed incor-

rectly.) This ensures that the permissible values for a variable won't be affected by read-only occurrences of that variable, but that they will be consistent with the read-only occurrences. Continuing the example:

	$q \in Q$ *formed by replacing* $Y?$ *with* $d \in \mathcal{D}$		
replacement ρ	$Y? \mapsto 9.83$	$Y? \mapsto 3$	$\cdots$
hierarchy q	*required* $X = \mathbf{9.83}$ *strong* $X = 4$ *weak* $Y = 3$	*required* $X = \mathbf{3}$ *strong* $X = 4$ *weak* $Y = 3$	$\cdots$
valuation θ	$\{Y \mapsto 3, X \mapsto 9.83\}$	$\{Y \mapsto 3, X \mapsto 3\}$	$\cdots$
consistency	$Y\theta \neq Y?\rho$	$Y\theta = Y?\rho$	$\cdots$
outcome	Discard	Keep	$\cdots$

The valuation $\{Y \mapsto 3, X \mapsto 3\}$ is the only consistent solution, and thus is the solution to the original hierarchy.

We now give a formal definition of the meaning of read-only annotations. In the definition, we will introduce new variables w_i, which we will want to omit in the final solution. We therefore define an operator *omitting*.

Definition. Let θ be a valuation. Let the domain of θ be the variables $v_1, \ldots, v_n$. Then

$$\theta \text{ omitting } w_1, \ldots, w_m$$

is the valuation σ such that the domain of σ is $\{v_1, \ldots, v_n\} - \{w_1, \ldots, w_m\}$, and such that $\sigma v = \theta v$ for all v in the domain of σ. Similarly, if Θ is a set of valuations,

$$\Theta \text{ omitting } w_1, \ldots, w_m \quad = \quad \{\theta \text{ omitting } w_1, \ldots, w_m \mid \theta \in \Theta\}$$

Definition. Let X be a variable in a constraint. We define a read-only annotation on X as $X?$.

Definition Let H be a constraint hierarchy containing read-only annotations, and let $\mathcal{D}$ be the domain of the constraints. Let $v_1, \ldots, v_m$ be the variables in H that have one or more read-only occurrences. Let $w_1, \ldots, w_m$ be new variables not occurring in H, and let J be the hierarchy that results from substituting w_i for each read-only occurrence of the corresponding variable v_i. (The w_i are no longer annotated as read-only in J; also, occurrences of the variables v_i that

aren't annotated as read-only are unaffected.) Define Q as the set of all hierarchies $J\rho$, where each ρ is formed by substituting arbitrary domain elements for the w_i:

$$Q \quad = \quad \{J\rho \mid d_1 \in \mathcal{D}, \ldots, d_m \in \mathcal{D},\ \rho = \{w_1 \mapsto d_1, \ldots, w_m \mapsto d_m\}\}$$

Let $solutions(J\rho)$ be the set of solutions to $J\rho$. (Here we are using the definition of "solutions" given in the basic theory Section 2.1, since J has no variables with read-only annotations.) Let the set of *consistent solutions* to $J\rho$ be defined as:

$$\begin{aligned} consistent(J\rho) \quad = \quad & \{\theta \mid \theta \in solutions(J\rho) \ \wedge \\ & \qquad w_1\rho = v_1\theta \ \wedge \ \ldots \ \wedge \ w_m\rho = v_m\theta\} \end{aligned}$$

In English, to be a consistent solution, if ρ maps w_i to some domain element d_i, then θ must map the corresponding v_i to the same domain element d_i (i.e., we guessed correctly).

The desired set of solutions to H is the set of all consistent solutions, omitting the mappings for the newly introduced variables w_i:

$$solutions(H) \quad = \quad \left(\bigcup_{J\rho \in Q} consistent(J\rho) \right) \text{ omitting } w_1, \ldots w_m$$

Proposition 6 *For a constraint hierarchy H containing only required constraints, let H' be the same hierarchy, but with the read-only annotations removed. Then $solutions(H) = solutions(H')$.*

Proof:

$solutions(H) \supseteq solutions(H')$

Let $v_1, \ldots, v_m$, $w_1, \ldots, w_m$, and J be defined as above. Let θ be a solution for H'. Define $\rho = \{w_i \mapsto v_i\theta, \ldots, w_m \mapsto v_m\theta\}$. (In other words, if θ maps v_i to d_i, then ρ maps the corresponding w_i to d_i.) Then clearly $\theta \in solutions(J\rho)$ and θ is consistent. So $\theta \in solutions(H)$.

$solutions(H) \subseteq solutions(H')$

Now assume θ is a solution for H. By definition, θ is a consistent solution to $J\rho$ for some ρ. As H consists only of required constraints and as θ is consistent with ρ, θ also satisfies all of the constraints in H'. ∎

3.1.1 Blocked Hierarchies

Even with this definition, it is possible for a constraint to restrict the values that its read-only annotated variables can take on. For example, consider the following constraint hierarchy for the domain $\mathcal{R}$:

$required \quad V > 0$

$required\ V? = 1$

The $V > 0$ constraint contains the only unannotated occurrence of V, and thus only $V > 0$ is allowed to affect the choice of values for V, and not $V? = 1$. However, the solutions to the first constraint by itself, $V > 0$, includes $V \mapsto 0.3$, $V \mapsto 1.728$, and so forth, in addition to $V \mapsto 1$, while $solutions(H) = \{V \mapsto 1\}$. Thus, the choice of values for V is being affected by the $V? = 1$ constraint. We therefore impose an additional check, $blocked(H)$, that tests for this situation.

The $blocked(H)$ predicate is true if any constraint in H limits the permissible values for one of its read-only annotated variables. In such a case, additional constraints can be added to the hierarchy so that the set of solutions can be found without any constraints limiting the permissible values for the read-only annotated variables.

The definition of $blocked(H)$ is based on the following observation: if there is a domain element d such that there are no solutions when d replaces all occurrences of a variable (both annotated and unannotated), but there are solutions when d replaces only the unannotated occurrences, then the annotated (read-only) occurrences are eliminating d from $solutions(H)$. Thus, if such a d exists, the annotated occurrences are restricting the values that the variable can take on, and $blocked(H)$ is true.

Definition.

$$
\begin{aligned}
blocked(H) \quad \equiv \quad & \exists d \in \mathcal{D}\ \exists i \in [1 \ldots m] \text{ such that} \\
& solutions(J\rho\theta\sigma) = \emptyset \ \wedge \ solutions(J\theta\sigma) \neq \emptyset \\
& \qquad \text{where } \rho = \{w_i \mapsto d\}\,,\ \theta = \{v_i \mapsto d\}\,, \text{ and} \\
& \qquad \sigma = \ \{w_1 \mapsto v_1, \ldots, w_{i-1} \mapsto v_{i-1}, \\
& \qquad\qquad w_{i+1} \mapsto v_{i+1}, \ldots, w_m \mapsto v_m\}
\end{aligned}
$$

If there are no read-only annotations on the variables in H, then clearly $blocked(H)$ is false.

The following proposition shows the relationship between solutions to hierarchies without read-only annotations and the solutions when a constraint with read-only annotations is included in the hierarchy. In the first case, the resultant hierarchy is not blocked, and in the second case, it is.

Proposition 7 *Let H be a constraint hierarchy without read-only annotations. Let c be a constraint and let $v_1, \ldots, v_n$ be the variables in c with read-only annotations. Let V be the variables in $H \cup \{c\}$.*

$$\neg blocked(H \cup \{c\}) \rightarrow$$
$$solutions(H \cup \{c\}) \ omitting \ V - \{v_1, \ldots v_n\} =$$
$$solutions(H) \ omitting \ V - \{v_1, \ldots v_n\}$$

$$blocked(H \cup \{c\}) \rightarrow$$
$$solutions(H \cup \{c\}) \ omitting \ V - \{v_1, \ldots v_n\} \subset$$
$$solutions(H) \ omitting \ V - \{v_1, \ldots v_n\}$$

Note that $solutions(H)$ omitting $V - \{v_1, \ldots v_n\}$ contains valuations only for the variables with read-only annotations.

Within the logic programming community, read-only annotations were originally introduced in Concurrent Prolog [Shapiro 86] for an entirely different purpose than ours, namely for the control of communication and synchronization among networks of processes. In our work so far, having a blocked solution is an unusual and undesirable state, which would arise only if a design or other error had been made in specifying the constraints. In contrast, in concurrent logic programming, blocking caused by read-only annotations is ubiquitous and essential for controlling program execution.

There were problems with the original formulation of read-only annotations in Concurrent Prolog (see [Saraswat 85] for a discussion), and a number of alternatives have been proposed. For example, Maher [Maher 87] describes ALPS, a class of languages that incorporates constraints into a flat committed-choice logic language. There are some similarities between the ALPS work and the use of read-only annotations in constraint hierarchies. In particular, the definition of blocked relates to the notions of satisfaction and validation given by Maher for the ALPS commit law. In this context, satisfaction and validation are concerned with the head, H, and the guard, G, of the rule in question. This head-guard pair is satisfied by a particular atom, A, if the current set of constraints (or global constraints) is consistent with the constraint $A = H$ and with the constraints in the guard G. The head-guard pair is validated if $A = H$ and G follow from the global constraints, or in other words, the rule can not be invalidated in the future by a further restriction of the values of global variables. According to this law, a rule can be committed to if

it is the only rule satisfied by the existing set of constraints, or if the rule is validated by those constraints.

Proposition 7 highlights the relationship with ALPS. When the hierarchy $H \cup \{c\}$ is not blocked, then H is in a sense validating the constraint c, or at least validating the read-only component of that constraint. When $H \cup \{c\}$ is blocked, then c is satisfied, or rather, it is *satisfiable*. Some new constraint, say c', can be added to H to form the new hierarchy H' so that the hierarchy $H' \cup \{c\}$ is no longer blocked and H' thus validates c.

3.1.2 Illustrative Examples of Using Read-only Annotations

Consider the hierarchy H for the domain $\mathcal{R}$:

$$\begin{array}{ll} \textit{required} & C * 1.8 = F? - 32.0 \\ \textit{strong} & C = 0.0 \\ \textit{weak} & F = 212.0 \end{array}$$

Without the read-only annotation on F, the solution to this hierarchy would be $\{\{C \mapsto 0.0, F \mapsto 32.0\}\}$. With it, the solution is $\{\{\mathrm{C} \mapsto 100.0, \mathrm{F} \mapsto 212.0\}\}$. (Note that the *strong* $C = 0.0$ constraint is not satisfied because there is no consistent solution that satisfies it.) To find the solution while accommodating the read-only annotation, the hierarchy J is formed by replacing $F?$ by a newly introduced variable W:

$$\begin{array}{ll} \textit{required} & C * 1.8 = W - 32.0 \\ \textit{strong} & C = 0.0 \\ \textit{weak} & F = 212.0 \end{array}$$

Q is the set of all hierarchies resulting from substituting an arbitrary real number for W. For example, the hierarchy resulting from the substitution $\rho = \{W \mapsto 14.0\}$ is:

$$\begin{array}{ll} \textit{required} & C * 1.8 = 14.0 - 32.0 \\ \textit{strong} & C = 0.0 \\ \textit{weak} & F = 212.0 \end{array}$$

which has the singleton set of solutions $\{\theta = \{C \mapsto$ -10.0$, F \mapsto 212.0\}\}$, but is not consistent because $W\rho \neq F\theta$ $(14.0 \neq 212.0)$.

The only hierarchy in Q with a consistent solution results from $\rho = \{W \mapsto 212.0\}$:

$$\begin{array}{ll} \textit{required} & C * 1.8 = 212.0 - 32.0 \\ \textit{strong} & C = 0.0 \\ \textit{weak} & F = 212.0 \end{array}$$

Now consider the motivating example in Section 3.1.1 for which *blocked* is true:

$$\begin{array}{ll} \textit{required} & V > 0 \\ \textit{required} & V? = 1 \end{array}$$

To illustrate the definition of *blocked*, form the new hierarchy J by replacing $V?$ with W:

$$\begin{array}{ll} \textit{required} & V > 0 \\ \textit{required} & W = 1 \end{array}$$

There exists a $d \in \mathcal{R}$, for example $d = 6$, such that, for the substitutions $\rho = \{W \mapsto 6\}$, $\theta = \{V \mapsto 6\}$, and $\sigma = \{\}$, $J\rho\theta\sigma$ has no solutions, but $J\theta\sigma$ does have a solution:

$J\rho\theta\sigma$	$J\theta\sigma$
$\textit{required} \quad 6 > 0$	$\textit{required} \quad 6 > 0$
$\textit{required} \quad 6 = 1$	$\textit{required}\ W = 1$
no solutions	$\{\{W \mapsto 1\}\}$

Hence *blocked* is true for this hierarchy. However, if we added the additional constraint $\textit{required}\ V = 1$ to the original hierarchy, then *blocked* would become false.

3.1.3 Practical Examples of Using Read-only Annotations

A trivial but useful example is a spreadsheet-like constraint that $A? + B? + C? = \mathit{Sum}$. The read-only annotations prevent the user from editing *Sum* and having the change propagate back to A, B, or C, but still allow the user to edit A, B, or C.

As noted in the introduction, an important use of read-only annotations is in constraints that reference an external input device or other outside source of information. For example, if we have a constraint that a point P follow the mouse, the constraint should be read-only on the mouse position:

$$P \quad = \quad \mathit{mouse.position}?$$

As another example, suppose we have a simple scrollbar displayed on the screen. When the "thumb" is dragged up and down, we want the top and bottom of the scrollbar to remain fixed. However, we want to be able to reposition the scrollbar as a whole, so simply anchoring the top and bottom isn't the correct solution.[1] To handle this problem cleanly, we define a constraint relating the position of the thumb, the top, the bottom, and a number *percent*, in which the the top and bottom are annotated as read-only:

$$percent \quad = \quad \frac{thumb - bottom?}{top? - bottom?}$$

The read-only annotations on *top* and *bottom* are specific to this constraint, so the whole scrollbar can be repositioned by some other "move" constraint.

3.1.4 Circularities

While the sets of solutions to many hierarchies are intuitively clear, this clarity often vanishes when the hierarchy contains cycles. We present two such examples here. These are pathological cases that would not arise in realistic applications—but nevertheless the theory should and does specify how they are to be handled.

The following two hierarchies both contain a cycle through variables annotated as read-only. In the first hierarchy, none of the constraints in the cycle is more restrictive than the others and so, intuitively, information can flow properly and still yield a solution.

$$\begin{aligned} &required \;\; X? = Y + 1 \\ &required \;\; X = Y? + 1 \end{aligned}$$

For this hierarchy, *blocked* is false and the set of solutions is the infinite set $\{\{X \mapsto d + 1, Y \mapsto d\} \mid d \in \mathcal{R}\}$.

In the second hierarchy, however, the *required* $X? = Y + 1$ constraint is more restrictive than the *required* $X \geq Y?$ one. Thus the "unequal" information flow results in *blocked* being true.

$$\begin{aligned} &required \;\; X? = Y + 1 \\ &required \;\; Y = 20 \\ &required \;\; X \geq Y? \end{aligned}$$

[1] We could almost achieve the desired result by putting strong (but not required) anchors on the top and bottom of the mouse. However, if other constraints on the output value from the slider became too strong, then the top or bottom would move; we would prefer a more robust object.

For this hierarchy, the set of solutions is $\{\{X \mapsto 21, Y \mapsto 20\}\}$; however, *blocked* is true.

3.2 Write-only Annotations

In addition to read-only annotations, it is also convenient if *write-only annotations* are available. Intuitively, if a variable is annotated as write-only in a constraint, we only want information to be able to flow from the constraint into that variable, and not back. We could define the effect of write-only annotations from first principles, in a manner analogous to the definition for read-only annotations. However, it is simpler to define write-only annotations in terms of read-only annotations.

Definition. Let X be a variable in a constraint. We define a write-only annotation on X as $X!$.

Definition. Let H be a constraint hierarchy containing write-only annotations (it may contain read-only annotations as well), and let $\mathcal{D}$ be the domain of the constraints. Let $v_1, \ldots, v_m$ be the variables in H that have one or more write-only occurrences. Let $w_1, \ldots, w_m$ be new variables not occurring in H, and let J be the hierarchy that results from substituting w_i for each write-only occurrence of the corresponding variable v_i. Let J' be the hierarchy formed by augmenting J with the additional required constraints $v_i = w_i?$ for $1 \leq i \leq m$. The desired set of solutions to H is the the set of solutions to J', with the mappings for the w_i omitted:

$$solutions(H) \quad = \quad solutions(J') \text{ omitting } w_1, \ldots w_m$$

The definition of the set $solutions(J')$ used above is, of course, that given in Section 3.1.

For example, let H be:

required $X! = Y$
strong $X = 4$
weak $Y = 3$

Intuitively, even though the constraint $X = 4$ is stronger than the constraint $Y = 3$, information will only be allowed to flow from Y to X in the $X! = Y$ constraint, since X is annotated as write-only. Tracing through the definition, the hierarchy J' is formed by replacing $X!$ by a newly introduced variable W, and adding the required constraint $X = W?$.

required $W = Y$
required $X = W$?
strong $X = 4$
weak $Y = 3$

The set of solutions to J' is $\{\{W \mapsto 3, X \mapsto 3, Y \mapsto 3\}\}$. The desired set of solutions to H is the same, but with the mapping for W omitted: $\{\{X \mapsto 3, Y \mapsto 3\}\}$.

3.3 Inter-Hierarchy Comparison

In some applications, it is useful to compare not just solutions to a given constraint hierarchy, but also solutions arising from several different hierarchies, for example, hierarchies arising from different rule choices in a logic program. To return to the scheduling example given in Chapter 2 Section 2.4, suppose that instead of only being able to meet before noon, Bjorn decides that he can also meet at 6:00 p.m. Then instead of a single constraint hierarchy, we would now represent this problem using the following *two* hierarchies:

strong	Bjorn:	meet before or at `11:00 a.m.`
strong	Michael:	meet at or after `5:00 p.m.`
weak	Alan:	meet at `3:00 p.m.`

strong	Bjorn:	meet before or at `6:00 p.m.`
strong	Michael:	meet at or after `5:00 p.m.`
weak	Alan:	meet at `3:00 p.m.`

As mentioned previously, 11:00 a.m. is one of many locally-predicate-better solutions to the first hierarchy. 6:00 p.m. is the only solution to the second hierarchy. It seems evident to a person trying to solve this problem that 6:00 p.m. is the "best" solution since it satisfies the schedules of the two people who strongly prefer to attend the meeting. One way to achieve this answer using the constraint hierarchy theory is to allow a comparison between the solutions arising from the first hierarchy and those arising from the second with respect to how well a solution satisfies its *own* hierarchy. (Clearly we wouldn't want to compare 11:00 a.m. and 6:00 p.m. using just one of the hierarchies. 11:00 a.m. isn't even a solution to the second hierarchy!)

In [Wilson & Borning 89] an earlier version of the constraint hierarchy theory was extended to allow for such inter-hierarchy comparisons. In what follows, the definitions from Chapter 2 Section 2.1 are similarly extended.

3.3.1 Extended Definitions

A solution to a *set* of constraint hierarchies Δ will consist of a valuation for all the free variables in Δ. We wish to generalize the previous definitions so that the set S contains all solutions to Δ, rather than just to a single hierarchy. Where Δ consists of a single hierarchy, the following definitions are equivalent to those given above.

$$
\begin{aligned}
S_{0_\Delta} &= \{\theta_H \mid H \in \Delta \wedge \forall c \in H_0\ e(c\theta_H) = 0\} \\
S_\Delta &= \{\theta_H \mid \theta_H \in S_{0_\Delta} \wedge \forall \sigma_J \in S_{0_\Delta} \\
&\qquad \neg(\mathbf{G}([\mathbf{E}(J_1\sigma_J), \ldots, \mathbf{E}(J_n\sigma_J)]) <_{\mathbf{g}} \mathbf{G}([\mathbf{E}(H_1\theta_H), \ldots, \mathbf{E}(H_n\theta_H)])) \\
&\qquad \text{where } n \text{ is the max of the number of levels in } H \text{ and } J\}
\end{aligned}
$$

We first define the set S_{0_Δ} of valuations that satisfy all the required constraints in some hierarchy in Δ. Each valuation θ in S_{0_Δ} is annotated by the hierarchy H that it satisfies. Using S_{0_Δ}, we define the set S_Δ as before, only now we are comparing across different hierarchies. Thus we eliminate potential valuations that are worse than some other from any hierarchy in Δ.

Consider the example given in Section 3.3. Let us refer to the first hierarchy as J and the second as H. Let σ be the valuation that maps the meeting time to 11:00 a.m. and let θ be the valuation that maps the meeting time to 6:00 p.m. The combined error sequence for the valuation σ_J using the weighted-sum-better comparator is $[[6], [4]]$. The combined error sequence for the valuation θ_H, also using weighted-sum-better, is $[[0], [3]]$. According to the extended definition given above, it is apparent that θ_H is better than σ_J because is satisfies the constraints in its own hierarchy better than σ satisfies the constraints in the hierarchy H. Note that θ is not a weighted-sum-better solution to hierarchy J, nor is σ a solution (using *any* comparator) to hierarchy H. We achieve the desired solution of meeting at 6:00 p.m. only by using inter-hierarchy comparison.

Extending the definition in this way gives rise to some nonmonotonic properties. These will be discussed in Chapter 7 Section 7.5.

We should point out that inter-hierarchy comparison only makes sense with respect to the global comparators, in which the errors at each level in the hierarchy are conglomerated, and it is therefore reasonable to compare those errors arising from completely different sets of constraints.

For the local and regional comparators, on the other hand, it is not clear what it would mean to order vectors of errors from different constraints via the $<_g$ relation. For this reason, inter-hierarchy comparison is only defined for global comparators.

As before, we insist that all comparators respect the set of hierarchies—the analogue to respecting the hierarchy. A comparator respects the set of hierarchies iff: if there is some valuation in S_0 that completely satisfies all the constraints through level k in its respective hierarchy, then all valuations in S must satisfy all the constraints in their respective hierarchies through level k.

$$
\begin{aligned}
&\text{if } \exists \theta_H \in S_0 \wedge \exists k > 0 \text{ such that} \\
&\qquad \forall i \in 1 \ldots k \; \forall p \in H_i \;\; p\theta_H \text{ holds} \\
&\qquad \text{then } \forall \sigma_J \in S \; \forall i \in 1 \ldots k \; \forall p \in J_i \;\; p\sigma_J \text{ holds}
\end{aligned}
$$

Chapter 4

Logic Programming, Constraint Logic Programming, and Hierarchical Constraint Logic Programming

4.1 Logic Programming

Logic programming is a programming paradigm based on a subset of first order logic. Its appeal stems from its simplicity, its declarative properties, and its well-defined semantics.

4.1.1 Definitions and Syntax

A *term* is defined in the manner of [Lloyd 84] as

- a variable,
- a constant, or
- if f is an n-ary function, and $t_1, \ldots, t_n$ are terms, then $f(t_1, \ldots, t_n)$ is also a term.

If p is an n-ary predicate, and $t_1, \ldots, t_n$ are terms, then $p(t_1, \ldots, t_n)$ is called an *atom*. A *clause* is defined as a formula of the form

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

where each L_i is a literal, that is either an atom or a negation of an atom, and where $x_1, \ldots x_n$ are the variables occurring in $L_1 \vee \ldots \vee L_m$. In other words, a clause is a disjunction of literals where all the variables are universally quantified.

In logic programming, a special notation is used to represent clausal formulas. A clause of the form

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

is written as

$$A_1, \ldots, A_j \leftarrow B_1, \ldots, B_k$$

The literals on the left hand side of the $\leftarrow$ are all positive and the comma denotes disjunction. The literals on the right hand side are negative and the comma denotes conjunction.

There is a special type of clause that forms the basis of logic programming that is called a *definite clause*, or sometimes a *program clause*. A definite clause consists of exactly one positive literal and is of the form

$$A \leftarrow B_1, \ldots, B_k$$

This clause, or *rule*, has a single positive literal on the left hand side called the *head* of the rule, and zero or more negative literals on the right hand side which make up the *body* of the rule. In the case where the right hand side is empty, the rule is called a *unit clause*, or sometimes a *fact* and is written

$$A \leftarrow$$

or just simply,

$$A.$$

In the case where the left hand side is empty, the clause is called a *goal*, or sometimes a *query*, and is written

$$\leftarrow B_1, \ldots, B_k$$

The empty clause is a special case and is written $\Box$.

Most logic programming languages are restricted to that subset of first order logic consisting of *Horn clauses*, i.e. rules, facts, and goals.

4.1.2 Semantics

In order to give meaning to logic programs, we need to define their behavior over some idealized interpreter. This is referred to as the procedural or operational semantics. To determine whether this ideal interpreter is doing its job correctly, we need to compare the procedural semantics to a more declarative semantics that defines the meaning of the underlying logical formulas. There are several equivalent ways of specifying the declarative semantics for logic programming. These include the logical consequence (or model theoretic or minimal model) semantics and the fixed point (or set-theoretic) semantics.

Operational Semantics

Let us consider again a typical rule in a logic programming language:

$$A \leftarrow B_1, \ldots, B_k$$

In the logical framework, we can think of this rule as meaning that A is true if $B_1, \ldots, B_k$ are all true. But we can also think of this in a more typical programming language fashion as a procedure. The head of the rule can be viewed as the procedure name, while the right-hand-side of the rule can be viewed as the procedure body. A query is a sequence of (possibly non-deterministic) procedure calls. Computation proceeds by calling each procedure in the query by first matching the call to a procedure of the same name in the program (i.e. the left-hand-side of a rule), and then computing the body of the procedure (i.e. the right-hand-side of the same rule), which itself may consist of more procedure calls. Computation halts when there are no more calls to be made.

To see why this procedural view works, remember that

$$A \leftarrow B_1, \ldots, B_k$$

is in essence an alternate way of writing

$$(A \vee \neg B_1 \vee \ldots \vee \neg B_k)$$

The deduction rule *resolution* states that for formulas, f, g, and h, if $f \vee h$ and $g \vee \neg h$, then $f \vee g$. Using resolution, therefore, we can take a set of goals

$$A_1, \ldots, A_i, \ldots, A_n$$

and the rule

$$A \leftarrow B_1, \ldots, B_k$$

and assuming that $A = A_i$, form the *resolvent*

$$(A_1, \ldots, A_{i-1}, A_{i+1}, \ldots A_n, B_1, \ldots, B_k)$$

Resolution is a refutation procedure that works by showing that something is false; it is only complete if it refutes a given formula by deriving the empty formula. Therefore, in the case of logic programming, if computation ends by deriving the empty clause, then the initial goal is true.

There are many different types of resolution. A form called SLD resolution, or SL resolution with definite clauses, is both sound and complete [Lloyd 84]. SL stands for selected literal and is a special case of linear resolution. In linear resolution the clause that is selected for the next resolution step must be the resolvent from the previous step. SL resolution further restricts the choice of literals on which to resolve, by selecting one that was among those most recently introduced [Maier & Warren 88]. A computation rule selects which atom to resolve next. A search rule determines which clause in the program will be selected next in the refutation step. An SLD interpreter using the breadth first search rule is both sound and complete, independent of the computation rule employed [Lloyd 84].

Declarative Semantics

There are two major results equating the operational and declarative semantics of logic programming. The first has to do with the equivalence of the operational semantics and the logical consequence semantics. In [Emden & Kowalksi 76] it is noted that the completeness of resolution means that derivability (in the operational model) coincides with logical implication (in the model theory). Hill [Hill 74] also gives this result: a query will succeed (i.e. have a derivation ending with the empty goal) if and only if the query is a logical consequence of the program.

The second major result equates the operational semantics to the fixed-point semantics. In [Emden & Kowalksi 76] the notion of a fixed-point is extended from the denotation of procedure definitions to the denotation of predicate logic programs. A transformation (or functional operator) T is associated with a program, and the meaning of the program is taken to be the least-fixed-point of T. It turns out that the fixed-point and model-theoretic semantics also coincide; we now have three ways of thinking about the meaning of logic programs.

4.1.3 Prolog — an Example

Prolog is the prototypical logic programming language. The syntax of Prolog is similar to the syntax of declarative clauses described above, except that we use the symbol `:-` rather than $\leftarrow$, and `?-` is used to denote a query. Prolog uses SLD resolution, but the search rule is based on depth-first search, rather than breadth-first. This amounts to an ordering rule in Prolog, that is the order of the rules in the program determines which derivations will occur first. It is possible that a query has a successful derivation, but that the ordering of the rules prevents the interpreter from finding it, and an infinite derivation will occur. For this reason, most standard Prolog implementations are not complete. Prolog sacrifices semantic purity for efficiency in several other ways as well. The occur check, which determines whether the system is attempting to unify a term with another term containing it, for example, is omitted for efficiency reasons. Cut is an operator that is used to avoid excessive backtracking, but which also further erodes the semantic equivalence of the operational and logical semantics.

Below is a sample Prolog program that determines ancestors:

```
mother(molly,hannah).
father(molly,jake)
mother(nate,molly)
mother(calla,molly)

parent(X,Y):- mother(X,Y).
parent(X,Y):- father(X,Y).

ancestor(X,Y):- parent(X,Y).
ancestor(X,Y):- parent(X,Z), ancestor(Z,Y).
```

The query `?- ancestor(calla,Y)` will produce three answers unifying `Y` with Molly, Hannah, and Jake respectively. These three answers represent three separate, successful derivations, although in practice they will be produced via backtracking to the last choice point and selecting a different rule in the resolution process.

Logic programming languages, as exemplified by Prolog, offer many advantages. They are declarative and have a clear semantics. They can express problems and solutions concisely owing

to the ability to have variables in answers and to the fact that they are multidirectional; the same program can be used with different input and output modes to achieve different programming goals. Furthermore, logic programming lends itself to certain types of programming styles such as generate and test.

On the other hand, logic programming languages can be inefficient because of the tremendous amount of searching for successful derivations that can occur. They are also restricted in their domain. For example programs that use numbers cannot be run multidirectionally. It was these drawbacks that led to the formulation of constraint logic programming [Jaffar & Lassez 87].

4.2 Constraint Logic Programming

The domain of Prolog is restricted to the Herbrand Universe, i.e. all those terms that can be constructed using the constant and function symbols of the program. Logic programming with constraints, on the other hand, extends the domain of discourse to include more powerful constraints. Constraint Logic Programming [Jaffar & Lassez 87] is a general scheme for such extensions, and is parameterized by $\mathcal{D}$, the domain of the constraints. The language that arises from a fixed set of constraints over $\mathcal{D}$ can be denoted by CLP($\mathcal{D}$). In place of unification (which can be viewed as testing the satisfiability of equations over the Herbrand universe), constraints are accumulated and tested for satisfiability over $\mathcal{D}$, using techniques appropriate to the domain. Several such languages have now been implemented, including CLP($\mathcal{R}$) [Jaffar & Michaylov 87, Jaffar et al. 92], Prolog III [Colmerauer 90], CHIP [Dincbas et al. 88, Van Hentenryck 89], CAL [Satoh & Aiba 90], CLP(Σ^*) [Walinsky 89], and Echidna [Sidebottoms & Havens 91].

The formal semantics of such languages differ mainly in the choice of underlying domain and constraints, as was shown formally in [Jaffar & Lassez 87]. It was also shown that for every language that can be obtained from the CLP scheme for solution-compact and satisfaction-complete domains $\mathcal{D}$, numerous desirable properties of the declarative and operational semantics hold—properties that had been considered characteristic of logic programming. In particular, CLP languages have coincident logical, fixed-point, and operational semantics.

Formally, a constraint is a relation over some domain $\mathcal{D}$. The domain $\mathcal{D}$ determines the constraint predicate symbols $\Pi_{\mathcal{D}}$ of the language, which must include $=$. A constraint is thus an expression of the form $p(t_1, \ldots, t_n)$ where p is an n-ary symbol in $\Pi_{\mathcal{D}}$ and each t_i is a term.

In CLP, rules are of the form

$$p(\mathbf{t}) \ :- q_1(\mathbf{t}), \ldots, q_m(\mathbf{t}), c_1(\mathbf{t}), \ldots, c_n(\mathbf{t}).$$

where where $p, q_1, \ldots, q_m$ are predicate symbols, $\mathbf{t}$ denotes a list of terms, and $c_1, \ldots, c_n$ are constraints. Operationally, we can think of executing the Prolog part of the program in the usual way, accumulating constraints on logic variables as we go, and either verifying that the constraints are solvable or else backtracking if they are not. The program can terminate with substitutions being found for all variables in the input, or with some constrained variables still unbound, in which case the output would include the remaining constraints on these variables.

4.2.1 CLP($\mathcal{R}$)— an Example

CLP($\mathcal{R}$) is a CLP language defined over the domain of the real numbers. The CLP($\mathcal{R}$) rules for computing mortgage interest [Heintze et al. 91] illustrate of the power of the language, since they can be used in a variety of ways (to compute the monthly payment given the other information, to find the symbolic relation between the principal and monthly payment, and so forth).

```
mortgage(Principal,Months,Interest,Balance,MonthlyPayment) :-
         Months > 0,
         Months <= 1,
         Balance + MonthlyPayment = Principal * (1 + Interest).

mortgage(Principal,Months,Interest,Balance,MonthlyPayment) :-
         Months > 1,
         mortgage(Principal * (1 + Interest) - MonthlyPayment,
         Months - 1, Interest, Balance, MonthlyPayment).
```

Compare the efficiency of the two goals,

```
?- mortgage(100000,T,0.01,0,MP),
   T >= 240.
```

```
?- T >= 240,
   mortgage(100000,T,0.01,0,MP).
```

In both cases, we are asking the system to compute the relation between the life of the mortgage and the monthly payments given a selling price of $100,000 and a 1% monthly interest rate. In addition, the life of the loan is constrained to be greater than or equal to 20 years. In the first query, this constraint follows the mortgage predicate, whereas in the second query it comes first. The first query is extremely inefficient, as the system will set up a constraint relating time to `1`, then `2`, then `3`, and so on until `T = 239` and failing each time as it discovers that the constraint `T >= 240` is unsatisfiable. In the second query, the information in the constraint can be carried around symbolically, without needing to ground the time variable `T`. As this example demonstrates, constraints can detect early failure, and as a result prune large areas of the search tree, thereby increasing the efficiency of the interpreter.

This example also shows how the use of constraints extends the multidirectionality of logic programming to other domains. The mortgage predicate can be called with its arguments instantiated to values (i.e. as input) or with variable arguments (i.e. as output), or with any combination of the two.

Now consider the goal `?- mortgage(P,360,0.01,0,MP).` In this query, we are asking for the relation between the principal and the monthly payments given a 30 year loan and a 1% monthly interest rate. This query succeeds with the constraint `P = 97.2183*MP`. This is called an *answer constraint* and demonstrates the ability of constraint logic programming languages to represent an infinite number of solutions quite concisely in the form of constraints.

4.3 Putting It All Together: Hierarchical Constraint Logic Programming

4.3.1 Adding the Constraint Hierarchy to Logic Programming

As discussed in Chapter 1 many applications of constraints either need, or would benefit from, support for default and preferential constraints, as well as required ones. ThingLab [Borning 81], as well as the other applications, used a constraint package built on top of an existing language. However, there are many benefits to having constraint hierarchies completely integrated with

a programming language. For example, in an integrated language we will be assured that the constraints are considered, and there is no need to call the constraint satisfier explicitly. (In a package, the programmer might simply ignore the constraints.) An integrated system allows more opportunities for optimizing the implementation. Finally, in the case of logic programming, there is an elegant theory available (the CLP scheme).

Thus, just as CLP is an extension of logic programming, the CLP scheme can be extended to include both hard and soft constraints. The Hierarchical Constraint Logic Programming scheme HCLP is just such an extension and is parameterized both by the domain $\mathcal{D}$ of the constraints and by the comparator $\mathcal{C}$, which is used to select among alternate ways of satisfying the soft constraints.

4.3.2 HCLP

An HCLP rule (or clause) takes the form

$$p(\mathbf{t}) \ :- \ q_1(\mathbf{t}), \ldots, q_m(\mathbf{t}), l_1 c_1(\mathbf{t}), \ldots, l_n c_n(\mathbf{t}).$$

where $\mathbf{t}$ is a list of terms, $p(\mathbf{t}), q_1(\mathbf{t}), \ldots, q_m(\mathbf{t})$ are atoms and $l_1 c_1(\mathbf{t}), \ldots, l_n c_n(\mathbf{t})$ are labeled constraints. An HCLP program is a collection of rules. A goal, or query, is a multiset of atoms. Operationally, goals are executed as in CLP, temporarily ignoring the non-required constraints, except to accumulate them. After a goal has been successfully reduced, the answer may still not be unique. In this case, the accumulated hierarchy of non-required constraints is then solved, using a method appropriate for the domain and the comparator $\mathcal{C}$, thus further refining the valuations in the solution. Additional valuations may be produced by backtracking.

An HCLP program can include a list of symbolic names for the strength labels, which in an implementation are then mapped to the non-negative integers. This is accomplished by the use of a "levels" predicate. If the label on a constraint is omitted, the label defaults to *required*. Regarding the comparator to be used, if it is significant, we will refer to the program as e.g. an HCLP($\mathcal{D}, \mathcal{LPB}$) one; but if any of various comparators might be appropriate, we will refer to the code simply as an HCLP($\mathcal{D}$) program.

4.3.3 HCLP($\mathcal{R}$)

HCLP($\mathcal{R}$) is an HCLP language defined for the domain of the real numbers. As such, all CLP($\mathcal{R}$) programs can be viewed as instances of HCLP($\mathcal{R}$) programs without any non-required constraints.

Conversely, we can begin with CLP($\mathcal{R}$) programs and add non-required constraints to produce an HCLP($\mathcal{R}$) program. We could, for example, take the mortgage program written in CLP($\mathcal{R}$) and add preferential constraints. The following goal uses the standard CLP($\mathcal{R}$) rule to find a symbolic constraint relating the Principal and the MonthlyPayment for a conventional fixed-rate 30 year mortgage at 1% interest per month, and then gives preferences regarding the maximum monthly payment and the minimum amount borrowed. For the given goal, the two preferences can be satisfied simultaneously.

```
?- mortgage(Principal,360,0.01,0,MonthlyPayment),
   strong Principal >= 100000, strong MonthlyPayment <= 1500.
```

When the monthly payment falls between \$1,500 and \$1,028.61, then both of the *strong* constraints can be satisfied. However if the query changes to

```
?- mortgage(Principal,360,0.01,0,MonthlyPayment),
   strong Principal >= 100000, strong MonthlyPayment <= 1000.
```

then the *strong* constraints can not be satisfied at the same time, i.e. given the constraints on the interest rate and the life of the loan, a buyer could not purchase a house for \$100,000 or more and keep the monthly payment below \$1,000. In this case, the single solution found by weighted-sum-metric-better would yield a monthly payment of \$1,028.61 for a loan of \$100,000. (No other solution has as small a combined error, since a given change in the principal results in a much smaller change to the monthly payment.) Worst-case-better and least-squares-better give solutions that are very close (within a dollar) to this one.

As another simple example, consider the $A + B = C$ problem discussed in Chapter 2 Section 2.4, where we want updates to C to produce reasonable behavior in A and B. We could code an adder in HCLP($\mathcal{R}$) by first defining the four levels in the hierarchy using the `levels` predicate, and then defining a rule for the adder as follows:

```
levels([required,strong,medium,weak]).

adder(OldA,OldB,OldC,NewA,NewB,NewC):-
       required OldA + OldB = OldC,
       required NewA + NewB = NewC,
       medium OldA = NewA,
       medium OldB = NewB,
       weak OldC = NewC.
```

To update the value of C we can invoke the goal,

```
?- strong NewC = 7, adder(2,3,5,NewA,NewB,NewC).
```

Of course, the solution depends on the actual choice of comparator.

Chapter 5

Semantics

This chapter presents the operational and declarative semantics of hierarchical constraint logic programming. Just as CLP preserves the strong relationship between the procedural and declarative semantics that is one of the important properties of logic programming, HCLP programs have logical and fixed-point theories that coincide with the operational models for the same programs. These three ways of viewing HCLP programs are presented below, followed by the results that demonstrate their equivalence.

5.1 Operational Semantics of HCLP

As presented in Chapter 4 Section 4.3.2, an HCLP rule (or clause) takes the form

$$p(\mathbf{t}) \ :- q_1(\mathbf{t}), \ldots, q_m(\mathbf{t}), l_1 c_1(\mathbf{t}), \ldots, l_n c_n(\mathbf{t}).$$

where $\mathbf{t}$ is a list of terms, $p(\mathbf{t}), q_1(\mathbf{t}), \ldots, q_m(\mathbf{t})$ are atoms and $l_1 c_1(\mathbf{t}), \ldots, l_n c_n(\mathbf{t})$ are labeled constraints. (In actuality, the atoms and constraints may include different lists of terms, but for simplicity we use $\mathbf{t}$, which is a list of all terms contained in the predicates and constraints of the rule.) An HCLP program is a collection of rules. A goal, or query, is a multiset of atoms. Whereas in practice, a goal may also contain constraints, without loss of generality, we will view goals as consisting only of atoms. (Any goal consisting of constraints can be renamed as a new predicate, and then this predicate can become the new goal.)

We present the notion of a derivation for a query Q to capture the operational behavior of an HCLP program. We assume in what follows that selected rules undergo a variable transformation

to ensure that they do not clash with existing variables. For each step in the derivation, an atom from the goal list is matched against the head of a rule in the program P, that atom is removed from the list of goals, and the atoms on the right hand side of the rule are added to the new goal list. (A *computation rule* determines which atom will be selected next. A *fair* computation rule is one in which each atom that appears in the derivation is chosen at some step. We assume that a fair computation rule is used.) The constraints are added to the constraint hierarchy (that consists of all the labeled constraints accumulated up to this point in the derivation). In addition, required equality constraints are created between the arguments in the selected atom and the arguments in the head of the selected rule. These constraints are treated no differently than any other constraints and are merely accumulated and added to the hierarchy. If there is no solution to the required constraints in the hierarchy, then the derivation is said to have failed. If there is some element in the derivation sequence such that all of the goals in the goal list have been reduced, and if there is a solution to the resulting constraint hierarchy, then the derivation is said to have succeeded. The final constraint hierarchy is the hierarchy associated with this empty goal list. A solution to this final hierarchy is then a solution to the original query.

More formally, a derivation for a program P and a query Q with selection rule R is a (possibly infinite) sequence of tuples $G_0, G_1, \ldots$ Each tuple G_i consists of a goal list and a constraint hierarchy. We define

$$G_0 \quad = \quad \langle Q, H^0 = \emptyset \rangle$$

Note that $H^0, H^1, \ldots$ are the hierarchies for $G_0, G_1, \ldots$, in contrast to $H_0, H_1, \ldots, H_n$ which are the sets of constraints in the hierarchy H at levels $0, 1, \ldots, n$ respectively.

Let G_i be a of tuple of the form $\langle \{p_1(\mathbf{x}_1), \ldots, p_n(\mathbf{x}_n)\}, H^i \rangle$ where $S_0(H^i) \neq \emptyset$. If there is a rule

$$p_j(\mathbf{t}) \ :- \ q_1(\mathbf{t}), \ldots, q_m(\mathbf{t}), l_1 c_1(\mathbf{t}), \ldots, l_k c_k(\mathbf{t}).$$

in P, and if R selects the atom $p_j(\mathbf{x}_j)$ at step i, then,

$$\begin{aligned} G_{i+1} \quad = \quad & \langle \{p_1(\mathbf{x}_1), \ldots, p_n(\mathbf{x}_n)\} - \{p_j(\mathbf{x}_j)\} \cup \{q_1(\mathbf{t}), \ldots, q_m(\mathbf{t})\} \ , \\ & H^i \cup \{l_1 c_1(\mathbf{t}), \ldots, l_k c_k(\mathbf{t})\} \cup \{\mathbf{t} = \mathbf{x}_j\} \rangle \end{aligned}$$

In the above equation, $\{p_1(\mathbf{x}_1), \ldots, p_n(\mathbf{x}_n)\} - \{p_j(\mathbf{x}_j)\}$ are the remaining unreduced goals from G_i, $\{q_1(\mathbf{x}_1), \ldots, q_m(\mathbf{x}_m)\}$ are the new goals from the rule, H^i is the previous hierarchy, and

$\{l_1c_1(\mathbf{t}), \ldots, l_kc_k(\mathbf{t})\}$ are the new constraints from the rule. $\{\mathbf{t} = \mathbf{x}_j\}$ are the required constraints that result from equating each argument in $\mathbf{t}$ with its corresponding argument in $\mathbf{x}_j$. For this derivation to be successful, it must be the case that $S_0(H^{i+1}) \neq \emptyset$. We emphasize that this derivation step is relative to the rule

$$p_j(\mathbf{t}) \ :- \ q_1(\mathbf{t}), \ldots, q_m(\mathbf{t}), l_1c_1(\mathbf{t}), \ldots, l_kc_k(\mathbf{t}).$$

i.e. if some other rule with head p_j was used at this step, then another derivation would result.

A derivation is successful if there is some tuple $G_f = \langle \emptyset, H^f \rangle$ in the derivation sequence, and if the hierarchy H^f has a solution. H^f is known as the *final constraint hierarchy*. A valuation s is a *computed solution* for the query Q iff Q has a successful derivation with final constraint hierarchy H^f and s is a solution for H^f. A derivation is *finitely failed* if there is no rule in P whose head has the same predicate symbol as the atom selected at a given step, or if the set of required constraints at some step in the derivation has no solutions, or if the final constraint hierarchy has no solutions. (See Chapter 2 Section 2.5.2 for cases where there are no solutions to constraint hierarchies even when there is a solution for the required constraints.) A query is finitely failed if every derivation for that query is finitely failed. Let FF_P denote the finite failure set with respect to a program P.

$$FF_P = \{Q \mid \ Q \text{ is finitely failed } \}$$

The finite failure set is used to characterize the "no" answers given by an HCLP interpreter.

If a goal succeeds, an interpreter will return an *answer*. An answer consists of a set of constraints (without strength annotations) on the variables in the initial goal. Additional answers may be produced by backtracking. Each answer represents one or more valuations in the solution to the constraint hierarchy. For example, the answer $X = 2$ represents the single valuation that maps X to 2, while the answer $Y > 5$ represents an infinite set of valuations, with each member of the set mapping Y onto a different number greater than 5. We make this distinction between answers and valuations since, on the one hand, we obviously prefer that an algorithm return $Y > 5$ rather than an infinite number of valuations. On the other hand, it is easier to define the comparators in terms of valuations rather than answers.

5.2 A Model Theory for HCLP

In [Shoham 88], the notion of preferred models is introduced as a way to represent the meaning of certain nonmonotonic logics. Some subset of the models of a set of formulas can be selected as the "preferred" models, thereby defining a particular nonmonotonic logic. A preference relation $\sqsubset$ is used to partially order the models. $M_1 \sqsubset M_2$ denotes that the interpretation M_1 is preferred over the interpretation M_2. A preferred model for a sentence A is an interpretation M such that $M \models A$ and there is no other interpretation M' such that $M' \models A$ and $M' \sqsubset M$. There are many possible methods of ordering models, and various logics can be characterized by defining different preference criteria.

There has been other work, specifically in the area of logic programming with negation, that deals with the notion of a *canonical model* for a particular logic program. There have been various methods used for defining what a canonical model should be (see [Apt et al. 88, Gelfond & Lifschitz 88, Przymusinski 88]), but the intention is always that the canonical model represent exactly those queries that have "yes" answers in the program. A canonical model for a program P is defined in several of these approaches by selecting some variant of P, P', and using a minimal model for P'. While we might also wish to adopt the concept of a canonical model to represent the meaning of an HCLP program, the idea of ordering models via a preference relation fits more closely with the notion of comparators than does the variation of the canonical model approach.

In this section, we first give a very short review of CLP theory and then discuss some of the aspects of HCLP that require us to use the notion of extended models. We then use these extended models with a preference relation to define the preferred models of HCLP programs. Finally we show how this framework can be altered to give a formal semantics for HCLP programs with inter-hierarchy comparison.

5.2.1 Review of CLP Model Theory

In [Jaffar & Lassez 87] a model is defined for CLP programs. First, the *base* of a program is defined as:

$$
\begin{aligned}
P_{\text{base}} \quad = \quad & \{p(x_1, x_2, \ldots, x_n)\theta \mid \\
& \quad p \text{ is a predicate in } \Pi_{\mathcal{D}} \text{ and} \\
& \quad \theta \text{ is a valuation for the variables } x_1, \ldots, x_n\}
\end{aligned}
$$

Then a *model* of a program P is defined as a subset I of P_{base} such that for every rule in P

$$A \leftarrow B_1, B_2, \ldots, B_m, C$$

and for every valuation θ that satisfies the constraints in C,

$$\{B_1\theta, B_2\theta, \ldots, B_n\theta\} \subseteq I \text{ implies } A\theta \in I$$

5.2.2 An Extended Model

A model for an HCLP program must contend with the non-required constraints. This can be quite complicated, as any reading of the program that doesn't in some way take error into account will not capture the intended meaning of the constraint hierarchy. In fact, unlike CLP, we cannot determine whether a particular valuation satisfies a non-required constraint unless it is viewed in the context of the entire hierarchy. It is the *disorderly* property of constraint hierarchies (see Chapter 2 Section 2.5.3) that gives rise to this phenomenon. In essence, this property states that the solution to a constraint hierarchy, H, may be disjoint with the solution to the hierarchy $H \cup \{lc\}$ where l is a label and c is a constraint. This means that we cannot look at error in isolation—the meaning depends on how rules are combined. To handle this, we define an *extended model* for P which consists of tuples of predicates and error sequences. If we consider the predicates in the extended model without the error sequences, then we simply have a model for P minus all of the non-required constraints, i.e. a CLP program. Intuitively we want to start out with a model for the underlying CLP program and then use the comparators to define a preference relation that utilizes the error sequences.

Proceeding as described above yields a model theory for HCLP with inter-hierarchy comparison. In order to first give a model theory for intra-hierarchy, or single hierarchy comparison, we need to complicate the notion of an extended model so that we can isolate all tuples in the extended model arising from the same derivation. It is not sufficient to look at a valuation in isolation, as its being in the solution set depends on how well it satisfies the hierarchy *in comparison* to other valuations that also satisfy the required constraints and that arise from the same derivation. To clarify this point, consider the following HCLP($\mathcal{R}, \mathcal{LMB}$) (locally-metric-better) program. (The numbers on the left are not part of the program; they will be used later to refer to particular rules.)

1 `squid(X):- mollusc(X), weak X` $\geq$ `10.`

2 `mollusc(X):- X = 11.`

3 `mollusc(X):- X` $\leq$ `3.`

The query `?- squid(X)` has two answers, one that maps X to 11 and one that maps X to 3. An extended model would include the tuples $\langle \text{mollusc}(11), [] \rangle$, $\langle \text{squid}(11), [[0]] \rangle$, and $\langle \text{squid}(2), [[8]] \rangle$, among others. (Note that [] is the empty error sequence.) If we compare the tuples $\langle \text{squid}(11), [[0]] \rangle$ and $\langle \text{squid}(3), [[7]] \rangle$, then we would wrongly eliminate squid(3) from the solution set as [[0]]<[[7]]. On the other hand, if we look at the tuple $\langle \text{squid}(2), [[8]] \rangle$ by itself, we will not recognize that there is another valuation, namely that which maps X to 3, whose error is less than the error for the valuation that maps X to 2. In order to avoid false comparisons, while also ensuring that the right valuations are compared, the extended model is made up of a set of sets, rather than a single set. Each set corresponds to a particular constraint hierarchy and each valuation in a set can be compared with every other valuation in the same set. Numbering the rules and subscripting the subsets of the extended model are record-keeping devices used to differentiate the different subsets.

While this appears to diminish the declarative nature of the model theory, it is a necessary extension. Intra-hierarchy comparison based as it is on a single derivation is in some sense inherently operational. Yet we find it useful to present a model theory for several reasons. First, it is helpful to be able to make comparisons with the more standard CLP model theory. It turns out that HCLP programs without non-required constraints yield extended models whose similarity to the models for the equivalent CLP programs are evident (which is as it should be!). Second, one of the main motivations for using single hierarchy comparisons is efficiency. The extended models for HCLP programs with inter-hierarchy comparison are declarative in nature, and with the exception of the error sequences are identical to models for the equivalent CLP programs. Third, the model theory enables us to consider the comparators as preference relations. This is a quite useful view and it allows us to see HCLP in relation to nonmonotonic logic. The constraint hierarchy in conjunction with logic programming allows us to prune the set of preferred valuations.

Let a *numbered program* be a program such that every rule has a unique number.

Let the *extended base* of a program P be defined as

$$
\begin{aligned}
P_{\text{ext-base}} \quad = \quad & \{\langle p(x_1, \ldots, x_n)\theta, R\rangle \mid \\
& \quad p \text{ is a predicate in } \Pi_{\mathcal{D}} \text{ and} \\
& \quad \theta \text{ is a valuation on the variables } x_1, \ldots, x_n \text{ and} \\
& \quad R \text{ is an error sequence}\}
\end{aligned}
$$

(Error sequences are defined formally in Chapter 2 Section 2.1.)

Let the result of *interleaving* error sequences $R_1, R_2, \ldots, R_m$, each of length n, be a new sequence of length mn, denoted by $R_1 \oplus R_2 \oplus \cdots \oplus R_m$. If $R_1 = [r_{11}, \ldots, r_{1n}]$, $R_2 = [r_{21}, \ldots, r_{2n}]$, $\ldots$, and $R_m = [r_{m1}, \ldots, r_{mn}]$, then

$$
R_1 \oplus R_2 \oplus \cdots \oplus R_m \quad = \quad [r_{11}, r_{21}, \ldots r_{m1}, \ \ \ldots, \ \ r_{1n}, r_{2n}, \ldots r_{mn}]
$$

Let $\wp(B)$ denote the power set of the set B. Let an *extended model* for a program P be a subset I of $\wp(P_{\text{ext-base}})$ such that for every rule in the numbered program P

$$
(i) \qquad\qquad A \leftarrow B_1, B_2, \ldots, B_m, H
$$

and for every valuation $\theta \in S(H_0)$,

$$
\langle B_1\theta, R_1\rangle \in I_1, \ \langle B_2\theta, R_2\rangle \in I_2, \ \ldots, \ \langle B_m\theta, R_m\rangle \in I_m \quad \text{for } I_1, I_2, \ldots, I_m \in I
$$

implies

$$
\langle A\theta, R_1 \oplus R_2 \oplus \cdots \oplus R_m \oplus [\mathbf{E}(H_1\theta), \mathbf{E}(H_2\theta), \ldots, \mathbf{E}(H_n\theta)]\rangle \in I_{i,1,2,\ldots,m}
$$

Let the *minimal extended model* for a program P, denoted MM_P, be an extended model for P such that there is no other extended model M'_P for P such that $M'_P \subset MM_P$.

For the program fragment given above, the extended minimal model consists of 4 subsets. The singleton subset I_2 consists of the tuple $\langle \text{mollusc}(11), [\,]\rangle$. I_3 is infinite and contains all tuples of the form $\langle \text{mollusc}(X), [\,]\rangle$, for all $X \leq 3$. The singleton subset $I_{1,2}$ consists of the tuple $\langle \text{squid}(11), [[0]]\rangle$. $I_{1,3}$ is also infinite and contains all tuples of the form $\langle \text{squid}(X), [[10 - X]]\rangle$, for all $X \leq 3$. For example, $\langle \text{squid}(3, [[7]]\rangle$, $\langle \text{squid}(0), [[10]]\rangle$, and $\langle \text{squid}(-1.3), [[11.3]]\rangle$ are members of $I_{1,3}$, among others.

5.2.3 Comparators as Preference Relations

Intuitively, the minimal extended model contains the smallest set of subsets of tuples that satisfy the required constraints, without taking the non-required constraints into consideration. It is through applying the comparators that the intended meaning of the hierarchy is achieved, but using the comparators to eliminate less desirable valuations means, in effect, that the subsets of tuples are getting smaller, i.e. some valuations that satisfy the required constraints will no longer be in the solution set. In other words we can no longer refer to this "better" solution set as an extended model, according to the definition given above. Therefore, we will define preference relations over subsets of $\wp(P_{\text{ext-base}})$ (extended interpretations), rather than over extended models. Let g be a comparator, and let I and I' be extended interpretations for a program P. Let S and S' be members of I and I' respectively such that S and S' have identical subscripts. Then $I' \sqsubset_g I$ if

1. $S' \subset S$, and
2. if $\exists \langle p(x_1, \ldots, x_n)\sigma, R_\sigma \rangle \in S, \notin S'$ then $\exists \langle p(x_1, \ldots, x_n)\theta, R_\theta \rangle \in S'$ and $\mathbf{G}(R_\theta) <_{\mathbf{g}} \mathbf{G}(R_\sigma)$

5.2.4 Mapping the Extended Model to a Standard Model

Our goal is to define a model for an HCLP program P using the comparator g. We still need to define a set that represents the answers to a query. First we define the pruning operator that simply removes the error sequences from an extended interpretation and collapses the subsets into a single set. Let I be an extended interpretation. Then

$$\begin{aligned} \text{prune}(I) \quad = \quad & \{p(x_1, \ldots, x_n)\theta \mid \exists S \in I \ \wedge \\ & \langle p(x_1, \ldots, x_n)\theta, R_\theta \rangle \in S\} \end{aligned}$$

Now we say that prune(M) is a preferred model for a program P using the comparator g if

1. MM_P is an extended minimal model for P using g and $M \sqsubset_g MM_P$, and
2. there is no other extended interpretation M' such that $M' \sqsubset_g M$

If a program contains no non-required constraints, then there is an equivalent CLP program that can be produced by simply omitting the *required* label from each constraint. In this case the extended minimal model I will consist of sets of tuples whose second elements are empty error

sequences. Therefore, none of these empty sequences will dominate any other sequence in the same set and no ground atoms will be eliminated in the preferred model M. For programs with required constraints only, M consists simply of all the first elements in the tuples in the sets in I.

5.2.5 A Model for Inter-Hierarchy Comparison

With only a small change, the extended model theory can be altered to give a semantics for inter-hierarchy comparison. (Inter-hierarchy comparison in HCLP programs is discussed in Chapter 7 Section 7.5.) Rather than dividing the extended model I into sets, the extended model for inter-hierarchy comparison consists of a single subset of $P_{\text{ext-base}}$.

Let an *extended model* for a program P using inter-hierarchy comparison be a subset I of $P_{\text{ext-base}}$ such that for every rule $A \leftarrow B_1, B_2, \ldots, B_m, H$ in P, and for every valuation $\theta \in S(H_0)$,

$$\langle B_1\theta, R_1\rangle, \langle B_2\theta, R_2\rangle, \ldots, \langle B_m\theta, R_m\rangle \in I$$

implies

$$\langle A\theta, R_1 \oplus R_2 \oplus \cdots \oplus R_m \oplus [\mathbf{E}(H_1\theta), \mathbf{E}(H_2\theta), \ldots, \mathbf{E}(H_n\theta)]\rangle \in I$$

where $\oplus$ is the interleave operator defined in Section 5.2.2.

A minimal extended model is defined as above.

The preference relation on extended interpretations is also a bit simpler than the one used for single hierarchy comparison. Let g be a comparator, and let I and I' be extended interpretations for a program P using inter-hierarchy comparison. Then $I' \sqsubset_g I$ if

1. $I' \subset I$, and
2. if $\exists\langle p(x_1, \ldots, x_n)\sigma, R_\sigma\rangle \in I, \notin I'$ then $\exists\langle p(x_1, \ldots, x_n)\theta, R_\theta\rangle \in I'$ and $\mathbf{G}(R_\theta) <_{\mathbf{g}} \mathbf{G}(R_\sigma)$

Finally, we need to redefine the prune operator for inter-hierarchy comparison.

$$\begin{aligned} \text{prune}(I) \quad = \quad & \{p(x_1, \ldots, x_n)\theta \mid \\ & \langle p(x_1, \ldots, x_n)\theta, R_\theta\rangle \in I\} \end{aligned}$$

Then, as defined for intra-hierarchy comparison, prune(M) is a preferred model for a program P with comparator g using inter-hierarchy comparison if

1. MM_P is an extended minimal model for P using g and $M \sqsubset_g MM_P$, and
2. there is no other extended interpretation M' such that $M' \sqsubset_g M$.

5.3 A Fixed-Point Semantics

To provide a fixed-point semantics for HCLP (without inter-hierarchy comparison), a function T_P is defined that maps sets of sets of tuples of the form $\langle A\theta, R\rangle$ into sets of sets of tuples that can be formed via the application of a single rule in the program P. A single set represents derivations that can later be compared because they are constructed from the same constraint hierarchy.

More formally:

$$T_P : \wp(\wp(P_{\text{ext-base}})) \rightarrow \wp(\wp(P_{\text{ext-base}}))$$

For $I \subseteq \wp(P_{\text{ext-base}})$

$$\begin{aligned}
T_P(I) \quad = \quad & \{F \mid \\
& A \leftarrow B_1, B_2, \ldots, B_m, H \text{ is a rule in } P, \text{ and} \\
& F = \{\langle A\theta, R\rangle \mid \\
& \qquad \langle B_1\theta, R_1\rangle \in I_1, \\
& \qquad \langle B_2\theta, R_2\rangle \in I_2, \\
& \qquad \vdots \\
& \qquad \langle B_m\theta, R_m\rangle \in I_m, \\
& \qquad \text{for } I_1, I_2, \ldots, I_m \in I, \text{ and} \\
& \qquad \theta \in S(H_0), \text{ and} \\
& \qquad R = R_1 \oplus R_2 \oplus \cdots \oplus R_m \oplus [\mathbf{E}(H_1\theta), \mathbf{E}(H_2\theta), \ldots, \mathbf{E}(H_n\theta)]\} \\
& \}
\end{aligned}$$

Let

$$\begin{aligned}
T_P \uparrow \omega \quad &= \quad \bigcup_{i=1}^{\infty} T_P^i(\emptyset) \\
T_P \downarrow \omega \quad &= \quad \bigcap_{i=1}^{\infty} T_P^i(\wp(P_{\text{ext-base}}))
\end{aligned}$$

While $T_P \uparrow \omega$ is a fixed-point, the valuations contained in its sets still need to be compared. The S_{best} operator is essentially a filter that eliminates those valuations whose combined error vectors are larger than some other valuation in the same subset. S_{best} computes the *preferred solutions* of the set I. Let

$$\begin{aligned} S_{\text{best}}(I) \quad = \quad & \{A\theta \mid \\ & \exists I' \in I \text{ and} \\ & \exists \langle A\theta, R_\theta \rangle \in I' \text{ and} \\ & \neg\exists \langle A\sigma, R_\sigma \rangle \in I' \text{ such that} \\ & \mathbf{G}(R_\sigma) <_{\mathbf{g}} \mathbf{G}(R_\theta)\} \end{aligned}$$

If a program contains no non-required constraints, then $T_P \uparrow \omega$ will consist of sets of tuples whose second elements are empty error sequences. Therefore, none of these empty sequences will dominate any other sequence in the same set and no ground atoms will be eliminated in $S_{\text{best}}(T_P \uparrow \omega)$. For programs with required constraints only, $S_{\text{best}}(T_P \uparrow \omega)$ consists simply of all the first elements in the tuples in the sets in I.

5.3.1 A Fixed-Point Semantics for Inter-Hierarchy Comparison

We can also alter the definition of S_{best} only slightly to achieve a fixed-point characterization for inter-hierarchy comparison, in much the same way as for the model theory. I now consists of a single subset of $\wp(P_{\text{ext-base}})$. Then we redefine the mapping function T_P as

$$T_P : \wp(P_{\text{ext-base}}) \rightarrow \wp(P_{\text{ext-base}})$$

For $I \subseteq P_{\text{ext-base}}$

$$\begin{aligned} T_P(I) \quad = \quad & \{\{\langle A\theta, R \rangle \mid \\ & A \leftarrow B_1, B_2, \ldots, B_m, H \text{ is a rule in } P, \text{ and} \\ & \langle B_1\theta, R_1 \rangle \in I, \\ & \langle B_2\theta, R_2 \rangle \in I, \\ & \vdots \\ & \langle B_m\theta, R_m \rangle \in I, \\ & \theta \in S(H_0), \text{ and} \end{aligned}$$

$$R = R_1 \oplus R_2 \oplus \cdots \oplus R_m \oplus [\mathbf{E}(H_1\theta), \mathbf{E}(H_2\theta), \ldots, \mathbf{E}(H_n\theta)]\}$$

$\}$

Similarly, we redefine S_{best} as

$$\begin{aligned} S_{\text{best}}(I) \quad = \quad & \{A\theta \mid \\ & \langle A\theta, R_\theta \rangle \in I \text{ and} \\ & \neg\exists \langle A\sigma, R_\sigma \rangle \in I \text{ such that} \\ & \mathbf{G}(R_\sigma) <_{\mathbf{g}} \mathbf{G}(R_\theta)\} \end{aligned}$$

In this definition, the mapping S_{best} is not monotonic, as is also discussed in [Wilson & Borning 89].

5.4 Relations between the Operational, Model-theoretic, and Fixed-Point Semantics of HCLP

The following two propositions give an equivalence for the computed solutions of a correct HCLP interpreter and both the preferred model of a program, and the preferred solutions of the fixed-point of the T operator.

Proposition 8 *v is a computed solution for a query Q and program P iff Qv is in the preferred model for P*

Proof: Without loss of generality, we will assume that an initial query consists of a single predicate. (Note that a multiset of atoms can always be replaced by a single atom by the introduction of a new predicate and a new rule whose body is the original query.) Let $Q = \{q_0(\mathbf{x})\}$. We also assume that programs are numbered. We first prove the following lemma equating derivations of length i with sets in MM_P which are subscripted by sequences of length i.

Lemma 1 *A query Q has a successful derivation of length i with final constraint hierarchy H^f iff there is a set $I_s \in MM_P$, with $|s| = i$, and*

$$\begin{aligned} I_s \quad = \quad & \{\langle q_0(\mathbf{x})\theta, R_{q_0} \rangle \mid \\ & \theta \in S(H_0^f) \text{ and} \\ & R_{q_0} = [\mathbf{E}(H_1^f\theta), \ldots, \mathbf{E}(H_n^f\theta)]\} \end{aligned}$$

Proof: We show the first direction ($\Rightarrow$) by induction on i.

Base Case: Let $i = 1$. Then $G_0 = \langle \{q_0(\mathbf{x})\}, \emptyset \rangle$ and $G_1 = G_f = \langle \emptyset, H^f \rangle$. So there must be a rule in P of the form

$$(a) \qquad q_0(\mathbf{t}) \ :- \ s_1 c_1(\mathbf{t}), \ldots, s_k c_k(\mathbf{t}).$$

and $H^f = \{s_1 c_1(\mathbf{t}), \ldots, s_k c_k(\mathbf{t})\} \cup \{\mathbf{t} = \mathbf{x}\}$. By the definition of MM_P, there is a set I_a in MM_P such that

$$\begin{aligned} I_a \quad = \quad & \{\langle q_0(\mathbf{x})\theta, R_{q_0} \rangle \mid \\ & \theta \in S(J_0) \ \text{ and} \\ & R_{q_0} = [\mathbf{E}(J_1\theta), \ldots, \mathbf{E}(J_n\theta)] \\ & \text{where } J = \{s_1 c_1(\mathbf{t}), \ldots, s_k c_k(\mathbf{t})\}\} \end{aligned}$$

So $J = H^f - \{\mathbf{x} = \mathbf{t}\}$, $J_1 = H_1, \ldots, J_n = H_n$, and $\theta \in S(H_0^f)$. We know that the latter holds as $\theta \in S(J_0)$, and if one of the constraints in $\{\mathbf{x} = \mathbf{t}\}$ violated a constraint in H_0, then the derivation would not succeed. (Note that if the constraint $\mathbf{x} = \mathbf{t}$ implies further restrictions on the set $S(H_0^f)$ because one or both of $\mathbf{x}$ and $\mathbf{t}$ is a vector of terms as well as variables, then the claim will still hold because a valuation is a mapping for only the free variables in a constraint.)

Induction Step: Assume that the claim holds for derivations of length i. Consider a successful derivation for $\{q_0(\mathbf{x})\}$ of length $i+1$ with final constraint hierarchy H^f. $G_0 = \langle \{q_0(\mathbf{x})\}, \emptyset \rangle$. So there is a rule in P

$$(b) \qquad q_0(\mathbf{t}) \ :- \ p_1(\mathbf{t}), \ldots, p_m(\mathbf{t}), s_1 c_1(\mathbf{t}), \ldots, s_k c_k(\mathbf{t}).$$

$G_1 = \langle \{p_1(\mathbf{t}), \ldots, p_m(\mathbf{t})\}, J \cup \{\mathbf{x} = \mathbf{t}\} \rangle$, where $J = \{s_1 c_1(\mathbf{t}), \ldots, s_k c_k(\mathbf{t})\}$. We know that the derivation is successful, so $\{p_1(\mathbf{t})\}, \ldots, \{p_m(\mathbf{t})\}$ must all have successful derivations whose lengths sum to i with final constraint hierarchies $H^{f_1}, \ldots, H^{f_m}$. So by the induction hypothesis, there are sets $I_{p_1}, \ldots, I_{p_m}$ in MM_P such that for all $j \in 1 \ldots m$

$$\begin{aligned} I_{p_j} \quad = \quad & \{\langle p_j(\mathbf{t})\theta, R_{p_j} \rangle \mid \\ & \theta \in S(H_0^{f_j}) \ \text{ and} \\ & R_{p_j} = [\mathbf{E}(H_1^{f_j}\theta), \ldots, \mathbf{E}(H_n^{f_j}\theta)]\} \end{aligned}$$

So, again by the definition of MM_P, there exists a set $I_{b,p_1,\ldots,p_m}$ in MM_P such that

$$
\begin{aligned}
I_{b,p_1,\ldots,p_m} \quad = \quad & \{\langle q_0(\mathbf{x})\theta, R_{q_0}\rangle \mid \\
& \theta \in S(J_0) \text{ and} \\
& R_{q_o} = R_{p_1} \oplus \ldots \oplus R_{p_m} \oplus [\mathbf{E}(J_1\theta), \ldots, \mathbf{E}(J_n\theta)]\}
\end{aligned}
$$

We know that $H^f = J \cup \{\mathbf{x} = \mathbf{t}\} \cup \bigcup_{j=1}^{m} H^{f_j}$. Let $F = H^f - \{\mathbf{x} = \mathbf{t}\}$. Then $R_{q_0} = [\mathbf{E}(F_1\theta), \ldots, \mathbf{E}(F_n\theta)]$. $H_1^f = F_1, \ldots, H_n^f = F_n$, and $\theta \in S(F_0)$ because for $j \in 1 \ldots m$, θ is in $S(H_0^{f_j})$, θ is in $S(J_0)$, and θ satisfies $\{\mathbf{x} = \mathbf{t}\}$ for the same reasons discussed in the base case.

This completes the proof of the forward direction. We now prove the other direction ($\Leftarrow$) by induction on the length of s.

Base Case: Let $|s| = 1$. Assume there is a set I_s in MM_P with $|s| = 1$, and

$$
\begin{aligned}
I_s \quad = \quad & \{\langle q_0(\mathbf{x})\theta, R_{q_0}\rangle \mid \\
& \theta \in S(J_0) \text{ and} \\
& R_{q_0} = [\mathbf{E}(J_1\theta), \ldots, \mathbf{E}(J_n\theta)]\}
\end{aligned}
$$

By definition of MM_P, there must be a rule in P of the form

$$
(s) \qquad q_0(\mathbf{t}) \ :- \ s_1 c_1(\mathbf{t}), \ldots, s_k c_k(\mathbf{t}).
$$

So there is a successful derivation for $\{q_0(\mathbf{x})\}$ of length 1, namely $G_0 = \langle\{q_0(\mathbf{x})\}, \emptyset\rangle$, $G_1 = G_f = \langle\emptyset, H^f\rangle$, where $H^f = \{s_1 c_1(\mathbf{t}), \ldots, s_k c_k(\mathbf{t})\} \cup \{\mathbf{x} = \mathbf{t}\}$. By arguments similar to those used in the proof of the other direction, we can see that $H_1^f = J_1, \ldots, H_n^f = J_n$, and θ satisfies the constraint $\mathbf{x} = \mathbf{t}$.

Induction Step: Assume the induction hypothesis holds for all $s \leq i$. Consider a set $I_{a,s}$ in MM_P such that $|a| = 1$, $|s| = i$, and

$$
\begin{aligned}
I_{a,s} \quad = \quad & \{\langle q_0(\mathbf{x})\theta, R_{q_0}\rangle \mid \\
& \theta \in S(J_0) \text{ and} \\
& R_{q_0} = [\mathbf{E}(J_1\theta), \ldots, \mathbf{E}(J_n\theta)]\}
\end{aligned}
$$

Again, by definition of MM_P, there is a rule

$$
(a) \qquad q_0(\mathbf{t}) \ :- \ p_1(\mathbf{t}), \ldots, p_m(\mathbf{t}), s_1 c_1(\mathbf{t}), \ldots, s_k c_k(\mathbf{t}).
$$

in P, $J = \{s_1 c_1(\mathbf{t}), \ldots, s_k c_k(\mathbf{t})\}$, and for $j \in 1 \ldots m$, there are sets I_{p_j} in MM_P such that

$$
\begin{aligned}
I_{p_j} \quad = \quad & \{\langle p_j(\mathbf{t})\theta, R_{p_j}\rangle \mid \\
& \theta \in S(H_0^{f_j}) \text{ and} \\
& R_{p_j} = [\mathbf{E}(H_1^{f_j}\theta), \ldots, \mathbf{E}(H_n^{f_j}\theta)]\}
\end{aligned}
$$

$R_{q_o} = R_{p_1} \oplus \ldots \oplus R_{p_m} \oplus [\mathbf{E}(J_1\theta), \ldots, \mathbf{E}(J_n\theta)]$. Now we know that $\sum_{j=1}^{m} |p_j| = |s| = i$. So for $j \in 1 \ldots m$, $|p_j| \leq i$, and the induction hypothesis holds. It follows that for $j \in 1 \ldots m$ there are derivations of $\{p_j(\mathbf{t})\}$ with final constraint hierarchies H^{f_j}. We can therefore construct a derivation for $\{q_0(\mathbf{x})\}$ by first reducing $q_0(\mathbf{x})$ using rule (a), and then following the derivations for each of the $\{p_j(\mathbf{t})\}$. The final hierarchy in this derivation is

$$H^f = J \cup \{\mathbf{x} = \mathbf{t}\} \cup \bigcup_{j=1}^{m} H^{f_j}$$

Again, we can see that if $R_{q_0} = [E(F_1\theta), \ldots, E(H_n\theta)]$, then $H_1^f = F_1, \ldots, H_n^f = F_n$, and θ satisfies the constraint $\mathbf{x} = \mathbf{t}$. This completes the proof of the lemma. ∎

Returning to the proof of the proposition, let v be a computed solution for $Q = \{q_0(\mathbf{x})\}$ so that $\{q_0(\mathbf{x})\}$ has a successful derivation with final constraint hierarchy H^f and $v \in S(H^f)$. Based on the lemma, we know that there is a set I_s in MM_P such that

$$\begin{aligned} I_s \quad = \quad & \{\langle q_0(\mathbf{x})\theta, R_{q_0}\rangle \mid \\ & \theta \in S(H_0^f) \text{ and} \\ & R_{q_0} = [\mathbf{E}(H_1^f\theta), \ldots, \mathbf{E}(H_n^f\theta)]\} \end{aligned}$$

Since I_s contains all tuples of the form $\langle q_0(\mathbf{x})\theta, R_{q_0}\rangle$ for exactly those valuations $\theta \in S(H_0^f)$, and since $v \in S(H^f)$, $q_0(\mathbf{x})v$ must be in the preferred model for P.

Similarly, if $q_0(\mathbf{x})v$ is in the preferred model for P, then by the lemma, there is a successful derivation for $\{q_0(\mathbf{x})\}$ with final constraint hierarchy H^f. As $\langle q_0(\mathbf{x})v, R_{q_0}\rangle$ is in I_s which in turn is in MM_P, it follows that $v \in S(H_0^f)$. As $q_0(\mathbf{x})v$ is in the preferred model, there can be no other valuation $w \in S(H_0^f)$ such that the combined error sequence for w precedes the combined error sequence for v in the lexicographic ordering defined by the comparator. Therefore $v \in S(H^f)$. ∎

Proposition 9 *Qv is in the preferred model for a program P iff $Qv \in S_{\text{best}}(T_P \uparrow \omega)$*

Proof: We will do this proof in two parts. First we show that $MM_P = T_P \uparrow \omega$. Then we show that the preferred model is equivalent to $S_{\text{best}}(MM_P)$.

We show that the claim $\forall i > 0$, $T_P^i(\emptyset) \subseteq MM_P$ holds by induction on i.

Base Case: Let $i = 1$. Then

$$
\begin{aligned}
T_P^1(I) \quad = \quad & \{F \mid \\
& A \leftarrow H \text{ is a rule in } P, \text{ and} \\
& F = \{\langle A\theta, R\rangle \mid \\
& \qquad \theta \in S(H_0), \text{ and} \\
& \qquad R = [\mathbf{E}(H_1\theta), \ldots, \mathbf{E}(H_n\theta)]\} \\
& \}
\end{aligned}
$$

Let (a) be the number of the rule $A \leftarrow H$ in P. Then by definition of MM_P, there is a set I_a in MM_P such that

$$
\begin{aligned}
I_a \quad = \quad & \{\langle A\theta, R\rangle \mid \\
& \qquad \theta \in S(H_0) \text{ and} \\
& \qquad R = [\mathbf{E}(H_1\theta), \ldots, \mathbf{E}(H_n\theta)]\}
\end{aligned}
$$

Induction Step: Assume that for all values less than or equal to i, $T_P^i(\emptyset) \subseteq MM_P$. Consider $T_P^{i+1}(\emptyset) = T_P(T_P^i(\emptyset))$. By definition

$$
\begin{aligned}
T_P(T_P^i(\emptyset)) \quad = \quad & \{F \mid \\
& A \leftarrow B_1, B_2, \ldots, B_m, H \text{ is a rule in } P, \text{ and} \\
& F = \{\langle A\theta, R\rangle \mid \\
& \qquad \langle B_1\theta, R_1\rangle \in I_1, \\
& \qquad \langle B_2\theta, R_2\rangle \in I_2, \\
& \qquad \vdots \\
& \qquad \langle B_m\theta, R_m\rangle \in I_m, \\
& \qquad \text{for } I_1, I_2, \ldots, I_m \in T_P^i(\emptyset), \text{ and} \\
& \qquad \theta \in S(H_0), \text{ and} \\
& \qquad R = R_1 \oplus R_2 \oplus \cdots \oplus R_m \oplus [\mathbf{E}(H_1\theta), \mathbf{E}(H_2\theta), \ldots, \mathbf{E}(H_n\theta)]\} \\
& \}
\end{aligned}
$$

By the induction hypothesis, since $I_1, I_2, \ldots, I_m \in T_P^i(\emptyset)$, $I_1, \ldots, I_m$ are also in MM_P. By definition of MM_P, there is a set $I_{a,1,\ldots,m}$ in MM_P such that

$$
\begin{aligned}
I_{a,1,\ldots,m} \quad = \quad & \{\langle A\theta, R\rangle \mid \\
& \qquad \theta \in S(H_0) \text{ and} \\
& \qquad R = R_1 \oplus \ldots \oplus R_m \oplus [\mathbf{E}(H_1\theta), \ldots, \mathbf{E}(H_n\theta)]\}
\end{aligned}
$$

To complete the proof that $MM_P = T_P \uparrow \omega$, we now show that if I_s is in MM_P, then there is a value $i = |s|$ such that $I_s \in T_P^i(\emptyset)$. We will proceed by induction on $|s|$.

Base Case: Let $|s| = 1$. Therefore there is a rule

$$(s) \qquad A \leftarrow H.$$

in P, and

$$\begin{aligned} I_s \quad = \quad & \{\langle A\theta, R\rangle \mid \\ & \theta \in S(H_0) \text{ and} \\ & R = [\mathbf{E}(H_1\theta), \ldots, \mathbf{E}(H_n\theta)]\} \end{aligned}$$

Clearly $I_s \in T_P^1(\emptyset)$.

Induction Step: Assume for all sets I_s in MM_P where $|s| = i$ that $I_s \in T_P^i(\emptyset)$. Consider a set $I_{a,s}$ in MM_P such that $|s| = i$, $|a| = 1$,

$$(a) \qquad A \leftarrow B_1, \ldots, B_m, H.$$

is a rule in P, and

$$\begin{aligned} I_{a,s} \quad = \quad & \{\langle A\theta, R\rangle \mid \\ & \theta \in S(H_0) \text{ and} \\ & R = [\mathbf{E}(H_1\theta), \ldots, \mathbf{E}(H_n\theta)]\} \end{aligned}$$

We know from the induction hypothesis that $I_1, \ldots, I_m \in T_P^i(\emptyset)$. From the definition of T_P it is easy to see that $I_{a,s} \in T_P^{i+1}(\emptyset)$ holds. This completes the proof that $MM_P = T_P \uparrow \omega$.

Let prune(M) be a preferred model for a program P. We now show that prune(M) = $S_{\text{best}}(MM_P)$.

Assume for the sake of contradiction that there is a valuation $\theta \in$ prune(M), but $\theta \notin S_{\text{best}}(MM_P)$. We know that there is some set $I \in MM_P$ such that $\langle A\theta, R_\theta\rangle \in I$. As $\langle A\theta, R_\theta\rangle \in M$, there can be no tuple $\langle A\sigma, R_\sigma\rangle \in I$ such that $\mathbf{G}(R_\sigma) <_{\mathbf{g}} \mathbf{G}(R_\theta)$. Otherwise, we could create an M' containing a set I' such that $\langle A\sigma, R_\sigma\rangle \in I'$, and $\langle A\theta, R_\theta\rangle \notin I'$ and $M' \sqsubset_g M$. But then by definition of S_{best}, $A\theta \in S_{\text{best}}(MM_P)$, leading to a contradiction.

Now assume for the sake of contradiction that $A\theta \in S_{\text{best}}(MM_P)$, but $A\theta \notin$ prune(M). By definition of S_{best}, we know that there can be no tuple $\langle A\sigma, R_\sigma\rangle$ in the same set in MM_P as $\langle A\theta, R_\theta\rangle$ such that $\mathbf{G}(R_\sigma) <_{\mathbf{g}} \mathbf{G}(R_\theta)$. Consider prune($M$). As $M \sqsubset_g MM_P$ (by definition), there is a set $I \in MM_P$, and there is a set $I' \in M$ such that $I' \subseteq I$, and $\langle A\theta, R_\theta\rangle \notin I', \in I$.

But this implies that there is a tuple $\langle A\sigma, R_\sigma \rangle \in I'$ and $\mathbf{G}(R_\sigma) < \mathbf{gG}(R_\theta)$. This leads to a contradiction. This completes the proof that the preferred model is equivalent to $S_{\text{best}}(MM_P)$.

Given that we have established that $MM_P = T_P \uparrow \omega$, we can see that Qv is in the preferred model for a program P iff $Qv \in S_{\text{best}}(T_P \uparrow \omega)$. ∎

The following proposition characterizes the "no" answers to queries in a HCLP program.

Proposition 10 $Q \in FF_P$ *iff* $\forall \theta$, $Q\theta \notin S_{\text{best}}(T_P \downarrow \omega)$

Proof: Assume that there is a query $Q = \{q_0(\mathbf{x})\} \in FF_P$. There are three cases in which a derivation can be finitely failed. The first case occurs when there is no rule in the program P whose head has the same predicate symbol as the atom selected at the given step. We can see that in this case such an atom, let's call it $r(\mathbf{t})$, would not occur in $T_P^1(\wp(P_{\text{ext-base}}))$, or more precisely, the tuple $\langle r(\mathbf{t})\theta, R \rangle$ for any valuation θ, would not appear in any set in $T_P^1(\wp(P_{\text{ext-base}}))$. Therefore the head of any rule containing $r(\mathbf{t})$ on the right hand side would not occur in $T_P^2(\wp(P_{\text{ext-base}}))$, and so on. For some $i > 0$, $\langle q_0(\mathbf{x})\theta, R \rangle$ for any valuation θ would not occur in $T_P^i(\wp(P_{\text{ext-base}}))$.

The second case in which a derivation can finitely fail is when the set of constraints at some step in the derivation has no solutions. It is easy to see from the definition of T, that when the rule corresponding to that step is used to construct the set F, that the requirement that $\theta \in S(H_0)$ would not hold, and F would be empty.

The third case occurs when there is a solution for all of the required constraints encountered in the derivation, but the resulting final constraint hierarchy H^f has no solutions. In this event, there would be a set F in $(T_P \downarrow \omega)$ corresponding to the derivation, but the set $S_{\text{best}}(T_P \downarrow \omega)$ would not contain any valuations relating to that set, because for any given valuation θ, there would always be another valuation σ such that $\mathbf{G}(R_\sigma) < \mathbf{gG}(R_\theta)$.

Now assume that there is a query $Q = \{q_0(\mathbf{x})\} \notin FF_P$. For a query *not* to be in the finite failure set, it must be the case that either the query has a successful derivation, or it has an infinite derivation. In the former case, $q_0(\mathbf{x}) \in S_{\text{best}}(T_P \uparrow \omega)$. We now show by induction on i that $\forall i$, $T_P^i(\emptyset) \subseteq T_P \downarrow \omega$.

Base Case: Consider the case when $i = 1$. As the empty set is a subset of any set, it is clear that $T_P(\emptyset) \subseteq T_P \downarrow \omega$.

Induction Step: Assume that $T_P^i(\emptyset) \subseteq T_P \downarrow \omega$ holds for all values less than or equal to i. Consider $T_P(T_P^i(\emptyset))$. From the induction hypothesis, we know that $T_P^i(\emptyset) \subseteq T_P \downarrow \omega$.

As $T_P \downarrow \omega$ consists of the intersection of all sets of the form $T_P^j(\wp(P_{\text{ext-base}}))$ for $j \in 1 \ldots \infty$, we know that $T_P^i(\emptyset) \subseteq T_P^j(\wp(P_{\text{ext-base}}))$ for some arbitrary j. Therefore $T_P^{i+1}(\emptyset)) \subseteq T_P^{j+1}(\wp(P_{\text{ext-base}}))$. As j was selected arbitrarily, we can see that $T_P^{i+1}(\emptyset) \subseteq T_P \downarrow \omega$.

In the case of an infinite derivation, note that any rule which is selected at some step in the derivation must have all of the predicates on the right hand side of the rule defined. (We say that a predicate is defined in P if there is at least one rule in P whose head consists of that predicate.) Because we are assuming a fair computation rule, any atom that appears in the derivation must be selected at some step. If some predicate in the body of a rule was not defined, then the fair computation rule ensures that the derivation would be finitely failed. Consider a query $\{q_0(\mathbf{x})\}$ with an infinite derivation. There must be a rule selected by the derivation of the form

$$q_0(\mathbf{t}) \ :- \ p_1(\mathbf{t}), \ldots, p_m(\mathbf{t}), s_1 c_1(\mathbf{t}), \ldots, s_k c_k(\mathbf{t}).$$

It is clear that there will be a set in $T_P^1(\wp(P_{\text{ext-base}}))$ with tuples of the form $\langle q_0(\mathbf{x})\theta, R\rangle$ where θ satisfies the required constraints on the right side of the rule. From the argument given above, the predicates $p_1, \ldots, p_m$ are defined in the program and so tuples corresponding to these predicates will be in sets in $T_P^1(\wp(P_{\text{ext-base}}))$. So tuples of the form $\langle q_0(\mathbf{x})\theta, R\rangle$ will be in a set in $T_P^2(\wp(P_{\text{ext-base}}))$. But the predicates on the right hand side of the rules selected for $p_1, \ldots, p_m$ are also defined, using the same argument. So tuples corresponding to these predicates will be in sets in $T_P^2(\wp(P_{\text{ext-base}}))$, and tuples of the form $\langle q_0(\mathbf{x})\theta, R\rangle$ will be in a set in $T_P^3(\wp(P_{\text{ext-base}}))$, and so on. Therefore for queries $Q \notin FF_P$ because they have infinite derivations, it is the case that $\exists\theta$ such that $Q\theta \in S_{\text{best}}(T_P \downarrow \omega)$. ∎

Chapter 6

Implementation

In this chapter, we discuss two implementations of HCLP($\mathcal{R}$). We introduce the DeltaStar algorithm for solving constraint hierarchies. Then we discuss its incorporation into the current HCLP($\mathcal{R}$) interpreter.

6.1 Constraint Satisfaction Algorithms

Searching for an efficient constraint satisfaction algorithm that works for all domains, comparators, and kinds of constraints would be a futile endeavor. Rather, we need to look for algorithms specialized by one or more attributes. In [Freeman-Benson et al. 90] a number of algorithms for solving constraint hierarchies are outlined, each of which makes a trade-off between generality and efficiency. Much of the constraint hierarchy research so far has used the *locally-predicate-better* comparator over arbitrary domains. When there are no circularities in the constraint graph, there is an efficient incremental algorithm for this comparator. For arbitrary linear constraints, there is also an efficient algorithm based on linear programming techniques, and this is the algorithm used in the current implementation of HCLP($\mathcal{R}$). For more details on the incremental acyclic algorithm, the reader is referred to [Freeman-Benson & Maloney 89, Freeman-Benson et al. 89, Freeman-Benson et al. 90, Maloney 91, Sannella et al. 93]; [Freeman-Benson et al. 90] and [Maloney 91] include proofs of correctness and complexity results.

6.2 Algorithms for Linear Equality and Inequality Constraints

One disadvantage of local propagation algorithms is that they usually cannot reliably handle cycles in the constraint graph. In some cases these algorithms will find an acyclic solution to a cyclic graph, but this behavior is not guaranteed; the algorithms often halt with a "cyclic constraint graph" error message instead. Further, if the constraints are truly simultaneous, then local propagation algorithms simply cannot find a solution. Therefore, we designed another set of algorithms that can solve constraint hierarchies consisting of arbitrary collections of linear equality and inequality constraints using the *weighted-sum-metric-better*, *worst-case-better*, and *locally-metric-better* comparators. These algorithms are instances of our general DeltaStar [Freeman-Benson & Wilson 90, Freeman-Benson et al. 92] framework and are collectively referred to as the Orange algorithms.

The DeltaStar framework is an algorithm for incrementally solving a constraint hierarchy, based on an alternate, but provably equivalent, description of the constraint hierarchy. DeltaStar is actually a family of algorithms, parameterized by an underlying "flat" constraint solver (i.e., one that solves a collection of required constraints). It is an algorithm for incrementally solving constraint hierarchies, but not for solving the constraints themselves. Rather, DeltaStar is built above a flat incremental constraint solver that provides the actual constraint solving techniques (numeric, symbolic, iterative, local-propagation, etc.). Thus it is adaptable to many different constraint solving algorithms.

One of the motivations for the DeltaStar framework was to replace the solver in our original HCLP($\mathcal{R}$, $\mathcal{LPB}$) interpreter (see Section 6.4.1) with an incremental version. In HCLP, there are two situations in which an incremental algorithm can save computation. First, if backtracking occurs in the normal course of executing an HCLP program that contains multiple definitions of a predicate, the constraints arising from the old definition of the predicate must be retracted, and ones from the new definition added, but other constraints are unaffected. An incremental algorithm would allow solutions to be modified incrementally in this situation. Second, an incremental algorithm is important for efficiency in interactive graphics applications where answers must be produced before a goal is completely reduced—we cannot wait until the complete history of the input events is known before computing display information. Instead, at appropriate points in program execution, we need to solve the currently generated constraint hierarchy. And

again, we'd prefer not to start from scratch each time we do.

6.2.1 A Recursive Definition

It was useful to alter our original constraint hierarchy definitions in order to construct an efficient implementation because the original definitions compared all valuations across all levels in the hierarchy simultaneously. What was wanted was a way to define the set of solutions recursively, level by level, so that a solver would not have to know about hierarchies explicitly, but instead could treat the constraints at each level in the same manner.

For global comparators, this desired definition is easy to formulate: S_R, the solution set, is defined using S_i, the set of valuations that best satisfy the constraints through level i in the hierarchy. At each level, valuations in S_{i-1} are compared based on how well they satisfy constraints at level i. Only those valuations that are no worse than any other make it through to the next level. The final solution S_R is simply the valuations that remain after passing through the entire constraint hierarchy. Formally, let g be a combining function that, when applied to real-valued vectors, returns some value that can be compared via $<>_g$ and $<_g$. Then,

$$
\begin{array}{rcl}
\underline{S_i} & \underline{\text{is}} & \underline{\text{a set of solutions:}} \\
S_0 & = & \{\theta \mid \forall c \in H_0 \; e(c\theta) = 0\} \\
S_i & = & \{\theta \mid \theta \in S_{i-1} \wedge \forall \sigma \in S_{i-1} \\
 & & \quad \neg(g(\mathbf{E}(H_i\sigma)) < g(\mathbf{E}(H_i\theta)))\} \\
S_R & = & S_n \\
 & & \quad \text{(where } n \text{ is the number of} \\
 & & \quad \text{non-required levels in } H\text{)}
\end{array}
$$

However, for local comparators, the desired definition is not as obvious. The difficulties are formally described in [Wilson 91, Freeman-Benson 91], but informally the problem is that global comparators use a total order over the valuations but local comparators use a partial order. Thus we use the following formulation when dealing with local comparators: T_R is defined using T_i, the set of sets of valuations that best satisfy the constraints through level i. As with S_i, valuations in T_{i-1} are compared at each level based on how well they satisfy constraints at level i. The difference between S_R and T_R has to do with what happens to a valuation that graduates to the next level. In the case of S_R, all such valuations are placed in the same set. In the case of T_R, the valuations are pruned and partitioned using the function $\mathcal{P}$: valuations that are equal

in the partial order are put into the same set, valuations that are incomparable are put into different sets, and valuations that are dominated are discarded. The final solution set T_R is simply the union of the sets of solutions remaining at the final level. (This final union is done for convenience, so that the result is a set of solutions rather than a set of sets.)

$$\begin{aligned}
\mathcal{P}(\Theta, H_i) \quad = \quad & \{Q_1, \ldots, Q_m\} \\
& \quad \forall \theta, \sigma \in \Theta,\ \left(\theta, \sigma \in Q_j \ \Leftrightarrow\ \vec{\theta} <>_g \vec{\sigma}\right) \\
& \wedge\ \forall \theta \in \Theta,\ \left(\theta \in Q_j \ \Leftrightarrow\ \nexists \sigma \in \Theta,\ \vec{\sigma} <_g \vec{\theta}\right) \\
& \text{where} \\
& \qquad \vec{\theta} = g(\mathbf{E}(H_i\theta)) \\
& \qquad \vec{\sigma} = g(\mathbf{E}(H_i\sigma))
\end{aligned}$$

$$\begin{aligned}
\underline{T_i} \quad \underline{\text{is}} \quad & \underline{\text{a set of sets of solutions:}} \\
T_0 \quad = \quad & \{S_0\} \\
T_i \quad = \quad & \bigcup_{\Theta \in T_{i-1}} \mathcal{P}(\Theta, H_i) \\
T_R \quad = \quad & \bigcup_{V \in T_n} V
\end{aligned}$$

6.2.2 An Example

To clarify the distinctions between S_R and T_R, consider the following constraint hierarchy. The S_R example uses the *unsatisfied-count-better* comparator, a special case of *weighted-sum-better* that uses the trivial error function and unit weights, and the T_R example uses the *locally-predicate-better* comparator.

symbolic strength	*numeric strength*	*constraints*
required	0	X $\geq$ 0
strong	1	X $\leq$ 4, X $\geq$ 10
medium	2	X = 12

$$\begin{aligned}
S_0 &= \{X \mapsto [0 \ldots \infty)\} \\
S_1 &= \{X \mapsto [0 \ldots 4, 10 \ldots \infty)\} \\
S_2 &= \{X \mapsto 12\} \\
S_R &= \{X \mapsto 12\}
\end{aligned}
\qquad\qquad
\begin{aligned}
T_0 &= \left\{\{X \mapsto [0 \ldots \infty)\}\right\} \\
T_1 &= \left\{\{X \mapsto [0 \ldots 4]\}, \{X \mapsto [10 \ldots \infty)\}\right\} \\
T_2 &= \left\{\{X \mapsto [0 \ldots 4]\}, \{X \mapsto 12\}\right\} \\
T_R &= \{X \mapsto [0 \ldots 4],\ \ X \mapsto 12\}
\end{aligned}$$

There is only one set of valuations in each of S_0, S_1, S_2, and S_R. There is one set is T_0, however there are two sets in T_1: one that satisfies "$X \leq 4$" but not "$X \geq 10$", and the other that satisfies "$X \geq 10$" but not "$X \leq 4$". (In the partial order of valuations, these two sets are incomparable yet they dominate all other sets.) There are also two sets at level T_2. Furthermore, the *medium* strength "$X = 12$" constraint only applies to $[10 \ldots \infty)$, thus it only refines the second set—the first set is untouched.

6.2.3 The Algorithm Itself

For brevity, we describe the algorithms in the following sections in terms of adding constraints to a constraint hierarchy. However, all the algorithms work equally well for removing constraints. A complete listing of the code for the DeltaStar algorithm is available in [Freeman-Benson & Wilson 90].

The Flat Solver Interface

The DeltaStar algorithm is built above a flat constraint solver that provides the actual constraint solving techniques and comparison methods. The key routine provided by the flat solver is `filter`:

> `filter( S : Solution, C : Set of Constraints ) -> Solution`
> Return the subset of the existing solution that minimizes the error of the set of constraints. The implementation of this routine effectively defines the *error metric* and the *comparator* supported by the flat solver.

In addition, the flat solver should provide other entries for efficiently determining if a new constraint is compatible with a current solution (i.e., if the error in satisfying it is 0), and for quickly adding a constraint to a current solution, given a guarantee that the constraint is compatible.

A Basic Algorithm

The S_R construction can be converted directly into the algorithm in figure 6-1. This basic algorithm iteratively filters the set of all possible valuations, `all`, using the constraints at each level in the hierarchy. However, for incremental use, this basic algorithm has some obvious defects:

```
all ← Flat.all_solutions;
n ← number of levels in H;
for i ← 0 ... n do
  all ← Flat.filter( all, H_i );
return all;
```

Figure 6-1: Basic algorithm

Overeager If H_m is the only level in the hierarchy that is modified (either by adding or removing constraints), then the solution sets of levels $0, \ldots, m-1$ will not change. Recomputing these solution sets wastes CPU cycles.

Redundant If a redundant constraint is added to the hierarchy, it will not change any solution sets, thus again this algorithm does unnecessary work.

Disjoint If the constraint graph formed by hierarchy H forms disjoint subgraphs, then modifications to one subgraph cannot affect other subgraphs. In other words, if H_a and H_b have no variables in common, then the solution of their union is the union of their solutions, i.e., $S_{a \cup b} = S_a \cup S_b$. And, if only H_a is modified, then only S_a will be affected, thus the basic algorithm does unnecessary work by also computing S_b.

A Better Algorithm

The first two inefficiencies can be easily solved using an incremental algorithm that retains the partial solution sets $S_0, \ldots, S_n$ between invocations and thus can avoid recomputing the sets that do not change. Figure 6-2 includes the code and an illustration of adding a new constraint at level 2.

A Disjoint Subgraph Algorithm

Because the run-time cost of flat constraint solvers is polynomial or exponential in the size of the problem, it can be more efficient to solve many small problems than one big problem. Fortunately, the constraints in many graphical applications can be divided into disjoint sets. For example, the x and y dimensions are completely independent in the graphical layout system of

```
global array S[n];
l ← index of modified level;
for i ← l ... n do
  t ← Flat.filter( S[i − 1], H_i );
  if t = S[i] then return S[n];
  S[i] ← t;
return S[n];
```

Figure 6-2: Incremental algorithm—Adding a constraint at level 2

Figure 6-3: Disjoint Subproblems

[Bill & Lundell 90] and, by solving for each dimension independently, they more than halved the response time. Similarly, when using a constraint hierarchy, two subproblems may be disjoint through level $i-1$ but connected at level i. Figure 6-3 illustrates how each level is divided into a number of subproblems, and how the incremental algorithm recomputes as few of them as possible. Recall that the top is the strongest (H_0) and the bottom is the weakest (H_n). Our preliminary performance measurements indicate that using disjoint subgraphs can improve the run-time by more than one order of magnitude.

Using Local Comparators

Because they compare solutions constraint-by-constraint rather than computing a global aggregate, local comparators are potentially more efficient than global ones. The DeltaStar algo-

```
global array T[n];
l ← index of modified level;
for i ← l ... n do
  s ← ∅;
  for each set t in T[i − 1] do
    s ← s ∪ { F | F = Flat.filter( t, H_{i,t})};
  T[i] ← s;
return union of all t in T[n];
```

Figure 6-4: Local Comparator Algorithm

rithm for local comparators uses the T_R construction from Section 6.2.1. Figure 6-4 illustrates that there is a set of sets of solutions for each level, and how each set can generate multiple sets at the next level.

Typically, graphical applications can only display one solution at a time (consider displaying sixteen different solutions to the placement of a dialog box). Based on this observation, two modifications can be made to the local comparator algorithm: one, compute just one solution by discarding all but one of the sets at each level T_i (illustrated graphically using grey regions in Figure 6-4); or two, use backtracking or lazy lists to produce solutions on demand. The discarding-all-but-one technique has the disadvantage that the remaining solutions can never be examined, but the powerful advantage of using less memory.

6.3 Other Algorithms

Although not designed for solving constraint hierarchies, many other constraint solving techniques are available, including augmented term rewriting [Leler 87], relaxation [Borning 81, Konopasek & Jayaraman 84, Sutherland 63], and searching for a solution over a finite domain. Augmented term rewriting is an equation rewriting technique borrowed from functional programming languages, with added support for objects and multi-directional constraints. Relaxation is an iterative numerical technique, in which the value of each real-valued variable is repeatedly adjusted to minimize the error in satisfying its constraints. Relaxation will converge on a solution close to a *least-squares-better* solution, unless it gets trapped in a local but suboptimal minimum. Mackworth [Mackworth 77], Van Hentenryck [Van Hentenryck 89], and others describe efficient algorithms for solving sets of constraints on variables ranging over finite domains.

6.4 Implementation of HCLP($\mathcal{R}$)

To test our ideas, and to allow us to experiment with HCLP programs, we first implemented a simple interpreter for HCLP($\mathcal{R}, \mathcal{LPB}$), i.e., for the domain of the real numbers, using the locally-predicate-better comparator, in CLP($\mathcal{R}$). We implemented a second interpreter in COMMON LISP, again for the domain of the real numbers, but which supports several different metric comparators rather than the single $\mathcal{LPB}$ comparator.

6.4.1 A Simple Interpreter for HCLP($\mathcal{R}, \mathcal{LPB}$)

Our first interpreter is written in CLP($\mathcal{R}$), allowing it to take advantage of the underlying CLP($\mathcal{R}$) constraint solver and backtracking facility. It has two phases. (The code for the interpreter is given in Appendix A.)

The first phase is a meta-interpreter, much like traditional Prolog meta-interpreters [Sterling & Shapiro 86]. It accepts a goal and either satisfies it immediately, or looks up the goal in the rule base, reduces it to subgoals, and recursively solves the subgoals. Required constraints are passed on to the CLP($\mathcal{R}$) solver immediately, while non-required constraints are simply pushed onto a stack. Non-required constraints that are part of the body of some rule are of course only added to the stack if that rule (minus the non-required constraints) succeeds. Upon completion of this phase, variable bindings and required constraints are maintained within the environment, and the stack of non-required constraints is passed as a constraint hierarchy to the second phase.

The second phase performs a recursive search for answers representing locally-predicate-better solutions to the constraint hierarchy produced for the particular derivation found during the first phase. The algorithm uses a recursive rule *Solve*. Each invocation of *Solve* represents a node in an implicit search tree of possible non-required constraints to satisfy next. A number of data structures are maintained by each invocation of *Solve*, including *Answer* (a list of unlabeled constraints that represents the answer computed so far), and *Untried* (a list of labeled constraints that have not yet been dealt with). Let s be the strongest strength of the constraints in *Untried*. For each constraint c in *Untried* with strength s, *Solve* appends c to the current answer, refines *Untried* by removing constraints that either have become unsatisfiable by the assumption that c holds or that are implied by the current answer, and then recursively calls itself with the remaining untried constraints. The base case is reached when the hierarchy is empty.

Each leaf in the implicit search tree represents an answer to the goal. Upon request, the

interpreter will backtrack to find alternate answers. These answers can arise in two ways. First, it is possible that the constraint hierarchy produced by the current choices of rules has more than one answer. Second, it is also possible that a goal can be satisfied in more than one way at the rule level: by using different rules to solve a goal, a new constraint hierarchy may be obtained. All answers to the current hierarchy are given before an attempt is made to resatisfy the goal. There is a unique computation tree associated with every answer, but the answers themselves are not always unique.

Here is a trivial example in HCLP($\mathcal{R}, \mathcal{LPB}$) to illustrate the interpreter's behavior upon backtracking.

```
banana(X) :- artichoke(X), weak X>6.
artichoke(X) :- strong X=1.
artichoke(X) :- required X>0, required X<10, weak X<4.
```

The first answer to `?-banana(A)` is produced by selecting the first of the `artichoke` clauses, yielding the hierarchy *strong* $A = 1$, *weak* $A > 6$. There is a single answer to this hierarchy, namely $A = 1$. Upon backtracking, the second `artichoke` clause is selected, resulting in the hierarchy *required* $A > 0$, *required* $A < 10$, *weak* $A < 4$, *weak* $A > 6$. This hierarchy has two answers. The first is $0 < A < 4$. Upon backtracking the second and final answer $6 < A < 10$ is then found.

As a result of being implemented on top of CLP($\mathcal{R}$), the interpreter is small (2 pages of code) and clean. However, the second phase is not incremental—rather, it recomputes all the LPB answers for each invocation, instead of incrementally updating its answers as constraints are added and deleted due to backtracking, thus making it not particularly efficient. A second deficiency is that it doesn't check for duplicate constraints when pushing non-required constraints onto the stack. However, since it implements only the LPB comparator, rather than one of the global ones, the only consequence is that a given answer could be produced more than once upon backtracking.

6.4.2 A DeltaStar-Based Interpreter for HCLP($\mathcal{R}, \star$)

This first implementation only supported the locally-predicate-better comparator. However, metric comparators are important for such applications as interactive graphics, layout, and scheduling, since if a soft constraint is unsatisfied we may nevertheless wish to satisfy it as well as

possible by minimizing its error. Global comparators, which consider the aggregate error for the constraints at a given level, are useful as well for such applications. There is also a fundamental efficiency problem, as noted above, since the interpreter used a batch algorithm, rather than an incremental one, to produce its answers.

We therefore wrote a second HCLP interpreter, again for the domain of the real numbers, but which supports the weighted-sum-metric-better, worst-case-better, and locally-metric-better comparators. The comparator to be used in a given program is indicated by a declaration at the beginning of an HCLP program. This second interpreter could thus be precisely but verbosely named HCLP($\mathcal{R}$,$\langle WSMB, WCB, LMB\rangle$). (So far we have resisted the name HCLP($\mathcal{R}$,$\langle\downarrow,|,\&\rangle$).)

An unfortunate consequence of the desire to support metric comparators is that we could no longer build so simply on top of CLP($\mathcal{R}$), since we now care not just whether or not a constraint is satisfied, but also about the error in satisfying it—information not conveniently available from CLP($\mathcal{R}$). The second interpreter is thus implemented in Common Lisp, and has to do much more of the work itself (such as keeping track of backtracking information).

The interpreter uses the incremental algorithm, DeltaStar, to find solutions to constraint hierarchies. DeltaStar manages the incremental addition and deletion of constraints at different levels of the hierarchy, given the pluggable flat solver. The remainder of the interpreter maintains the database of clauses, backtracking information, and other details.

The HCLP($\mathcal{R},\star$) interpreter includes some evaluable predicates for performing input and graphical output, so that we can use HCLP($\mathcal{R},\star$) for interactive graphics applications. For example, there are predicates for getting the mouse position and button state, and for drawing lines, circles, placing text, and so forth. The interpreter makes the appropriate calls to Garnet routines to perform the needed actions. (Garnet [Myers et al. 90b] is a user interface toolkit, written in Common Lisp and using X windows.)

Using DeltaStar in HCLP($\mathcal{R},\star$)

In our Common Lisp implementation of HCLP($\mathcal{R},\star$), the flat solver is the Simplex algorithm, with implementations of `filter` that support minimizing the weighted sum of a set of constraints, minimizing the maximum error of a set of constraints, and minimizing the pareto-optimal point of a set of constraints, thus implementing weighted-sum-metric-better, worst-case-better, and locally-metric-better respectively. The class of constraints that can be accommodated are linear equalities and non-strict inequalities.

To support these comparators, DELTASTAR transforms the constraint hierarchy into a series of linear programming problems. In a standard linear programming problem [Murty 83], we wish to minimize (or maximize) a linear expression in k real-valued variables $x_1, \ldots, x_k$, subject to the nonnegativity constraints $x_1 \geq 0, \ldots, x_k \geq 0$, and also to m additional linear equality or inequality constraints on $x_1, \ldots, x_k$. The expression to be minimized or maximized is called the *objective function*. Reference [Murty 83] is a comprehensive discussion of linear programming theory and algorithms; all of the transformation techniques mentioned in the following paragraphs are discussed in this volume. Our implementation uses code for the Simplex algorithm taken from [Press et al. 86] and translated to Lisp.

In general, the variables in the constraint hierarchy are unrestricted in sign, while those in a linear programming problem must be nonnegative. There is a standard technique for handling unrestricted variables in linear programming problems. Each unrestricted variable x_j is replaced by the difference of two nonnegative variables x_j^+ and x_j^-, so that $x_j = x_j^+ - x_j^-$. We then solve the problem involving the x_j^+ and x_j^- variables, and use this solution to find values for the original x_j. (This is not a particularly efficient way of handling this situation, but we used it in this prototype implementation as we wanted to use the Simplex code unaltered.)

We now consider the weighted-sum-metric-better comparator. Initially, we minimize the weighted sum of the errors of the H_1 constraints, subject to the H_0 constraints. Even after the transformation to handle the variables without sign restrictions, this still isn't quite a linear programming problem, since the objective function is a weighted sum of absolute values. However, we adapt another standard technique for converting a problem in which the objective function is the weighted sum of absolute values into a linear programming problem. Let c be a constraint in H_1. If c is an equality constraint $a_1x_1 + \cdots + a_kx_k = b$, then the error in satisfying c is $e = \mid a_1x_1 + \cdots + a_kx_k - b \mid$. We augment the linear programming program with two new variables e^+ and e^- (both of which must obey the usual non-negativity constraints), and add the constraint $a_1x_1 + \cdots + a_kx_k - b = e^+ - e^-$. If the property

$$e^+ = 0 \quad \text{if} \quad e \leq 0$$
$$e^- = 0 \quad \text{if} \quad e \geq 0$$

is satisfied, then clearly $e = e^+ + e^-$. Conveniently, this property is in fact satisfied by the solution produced by the Simplex algorithm. Hence, if the weight for constraint c is w, then its contribution to the objective function (the weighted sum of the errors) is $we^+ + we^-$. If c is an

inequality $a_1x_1 + \cdots + a_kx_k \leq b$, then its contribution to the objective function is simply we^+. (In this case we drop the we^- term from the objective function. If the inequality is satisfied, then e^+ will be 0, and e^- will be 0 or positive. If the inequality is not satisfied, then e^+ will be positive and e^- will be 0.) Finally, if c is an inequality $a_1x_1 + \cdots + a_kx_k \geq b$, then its contribution to the objective function is we^-.

If this initial linear programming problem (minimizing the weighted sum of the errors of the H_1 constraints, subject to the H_0 constraints) has a unique solution, we are done. Otherwise, we add to the linear programming problem a constraint that the weighted sum of the H_1 constraints attain its minimum value (as computed in the previous step), and set up another problem, where the new objective function minimizes the weighted sum of the errors of the H_2 constraints. We continue in this manner for the remaining levels.

For the worst-case-better comparator, we initially minimize the maximum of the weighted errors of the H_1 constraints, subject to the H_0 constraints. As before, this isn't a linear programming problem, but yet another standard technique is available for transforming it into one. See [Murty 83, page 18].

For locally-metric-better, we consider each constraint in level H_1 individually in relation to the solution for H_0. Calling Simplex with a particular H_1 constraint tells us the bounds of the solution with respect to that constraint. When all of the constraints in H_1 have been handled, the various solutions are combined to yield a solution for level H_1. If all of the constraints at that level are satisfied, then this process continues using the constraints at level H_2 in relation to the current solution. If some constraint at level H_1 is not satisfied, then the current solution is the solution to the entire hierarchy.

Filter routines for each of these comparators are defined separately from the logic programming interpreter. A call to `filter` solves a single level in the hierarchy by minimizing the error of a set of constraints (the current solution) with respect to some other set of constraints (the constraints at some level in the constraint hierarchy). The calling routine in the interpreter uses `filter` to solve the entire constraint hierarchy.

Regionally-metric-better is not currently available. However, it could be added to the implementation by changing the routine that calls `filter`. The filter for locally-metric-better could be used as is. Rather than stopping iteration through the hierarchy when some constraint cannot be satisfied, as is now done for locally-metric-better, the routine would continue to call `filter` through all levels of the hierarchy. If, in the future, we wanted to implement least-squares-better,

we would define a filter using some non-linear equation solver. The logic programming interpreter would not need to be revised.

Efficiency Issues

This second interpreter is still a research prototype to test our ideas, rather than being production-quality software. Among its limitations are its restriction to linear equalities and non-strict inequalities, and its efficiency.

Regarding the classes of constraints that can be accommodated, clearly it would be desirable to at least provide local propagation for non-linear constraints, *à la* CLP($\mathcal{R}$). Regarding efficiency, implementing the interpreter in COMMON LISP has made the implementation easier, but slower than writing in a language such as C. In addition, DELTASTAR uses only a narrow interface between the flat constraint solver and the rest of the algorithm. Many optimizations would be possible here, following the excellent example of the CLP($\mathcal{R}$) interpreter [Jaffar et al. 92], such as handling simple constraints within the inference engine, providing different solvers for equalities and inequalities, and implementing an incremental version of the Simplex algorithm more efficiently. Nevertheless, the use of the DELTASTAR algorithm has aided us in rapidly testing different satisfaction algorithms and comparators. For example, only one person-hour was needed to add the weighted-sum-metric-better comparator once the DELTASTAR framework was in place.

The time complexity of the HCLP interpreter is dominated by the cost of the flat constraint solver. In the complexity discussion below, we factor out this cost, and represent it simply as p (where p is a function of the number of variables and the number of constraints). The worst-case time complexity of the flat constraint solver we use (the Simplex algorithm) is exponential in the number of variables; however, this behavior is apparently exhibited only by artificially constructed examples. On real problems Simplex performs remarkably well. There *are* linear programming algorithms whose worst case time is polynomial. Whether such algorithms (Karmarkar's algorithm in particular) are superior in practical use is still a matter of debate [Karloff 91].

The cost of solving a particular HCLP($\mathcal{R}, \star$) query can be broken down into two parts. The first is the cost of finding some solution considering only the required constraints, i.e. the cost of solving the corresponding CLP($\mathcal{R}$) program. (Actually, there is a fair amount of overhead in storing the non-required constraints and storing solutions in the event of backtracking, but this is

also dominated by the cost of calling the flat solver to solve the required constraints.) The second is the cost of solving the constraint hierarchy. While this is a fairly expensive procedure, it is only done once per answer because there is no need to solve the hierarchy until we know that a particular derivation will not fail. Furthermore, because the algorithm in the current interpreter is incremental, not all of the solution is lost upon backtracking.

Many of the optimizations described above will improve the running time of the interpreter with respect to the first type of cost, i.e. that of finding a solution to the required constraints. Using a more efficient flat solver would improve both the cost of finding the set S_0 and of solving the entire hierarchy.

Consider a particular call to filter, `filter(S,C)`, where S and C are sets of constraints. Let v denote the number of variables in S and C. Let c denote the number of constraints in C. Let n be the number of levels used in the HCLP($\mathcal{R}, \star$) program. Let p be the cost of running the linear programming algorithm in the average case for v variables and c constraints. The cost of `filter` (in the average case) for both weighted-sum-metric-better and worst-case-better is $2vp$. The cost of `filter` for locally-metric-better is $2cvp$. It is often the case that `filter` will not be called n times. We already saw how this could be in the case of locally-metric-better, but it will also be true if a particular derivation does not include constraints at level n, or in the case that a unique solution is found before processing the constraints at level n. However, assuming that `filter` is called n times, then the cost of solving the hierarchy is $2nvp$ for weighted-sum-metric-better and worst-case-better, and $2ncvp$ for locally-metric-better.

Chapter 7

Applications

This chapter presents a number of examples of HCLP programs. The programs here are all simple, but illustrate the use of constraint hierarchies for a variety of application areas. In the discussions, the significance of the different possible comparators are emphasized, as well as how one or another might be most appropriate for a given application. All of the sample programs in this chapter are written for the domain of the real numbers. (However, implementations of HCLP languages for other domains are of course possible as well, and would be useful for other applications. For example, the HCLP language CHAL [Satoh & Aiba 91a, Satoh & Aiba 91b] includes support for the domain of the booleans, as well as for polynomial equations over algebraic numbers. See also the discussion of this language in Chapter 8 on related work.)

Regarding the comparator to be used, if it is significant, the program will be referred to as e.g. an HCLP($\mathcal{R}, \mathcal{LPB}$) one; but if any of various comparators might be appropriate, we will refer to the code simply as an HCLP($\mathcal{R}$) program.

An HCLP program can include a list of symbolic names for the strength labels, which in an implementation are then mapped to the non-negative integers. If the label on a constraint is omitted, the label defaults to *required*; weights default to 1. For brevity, we assume that for all the program examples in this paper, the following strengths have been defined: *required*, *strong*, *medium*, *weak*.

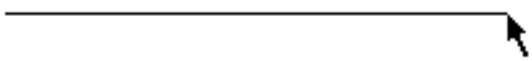

Figure 7-1: Moving an endpoint of a horizontal line

7.1 Interactive Graphics Examples

The original motivation for the definition of constraint hierarchies was to support interactive graphics in a more declarative manner. The following example is illustrative of a wide class of such programs. There is a horizontal line displayed on the screen, and we are moving one endpoint with the mouse (Figure 7-1). There is a *required* constraint that the line be horizontal, a *medium* constraint that one endpoint of the line follow the mouse, and a *weak* constraint that the endpoints of the line remain fixed. This *weak* constraint gives stability to the line as it is moved, so that, for example, it doesn't suddenly triple in length as we move the endpoint by some small distance.

The HCLP($\mathcal{R}$) rule below expresses the desired update behavior. It takes as arguments terms representing the old and new states of the horizontal line, and a third term that is the x-y distance by which one endpoint should be moved. Any or all of the terms may contain variables. However, in typical use in an interactive graphics application, the old state of the line and the displacement would be ground, while the new state of the line would be a variable, whose value would be computed as a result of satisfying the constraints.

```
move_horiz_end2(line_segment(OldX1,OldY1,OldX2,OldY2),
                line_segment(NewX1,NewY1,NewX2,NewY2),
                delta(DX,DY)) :-
    required OldY1 = OldY2, required NewY1 = NewY2,
    medium OldX2 + DX = NewX2, medium OldY2 + DY = NewY2,
    weak OldX1 = NewX1, weak OldY1 = NewY1,
    weak OldX2 = NewX2, weak OldY2 = NewY2.
```

Suppose now we anchor the other end of the horizontal line, so that this other end becomes difficult to move (Figure 7-2). We'll use a *strong* rather than a *required* constraint, so that the anchor could be moved if needed by using an even stronger mouse constraint.

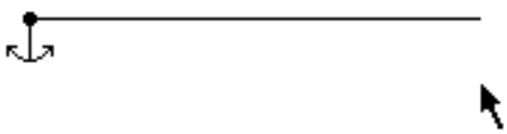

Figure 7-2: Moving an endpoint of an anchored horizontal line

```
move_horiz_end2_anchor_end1(line_segment(OldX1,OldY1,OldX2,OldY2),
                            line_segment(NewX1,NewY1,NewX2,NewY2),
                            Displacement) :-
   move_horiz_end2(line_segment(OldX1,OldY1,OldX2,OldY2),
                   line_segment(NewX1,NewY1,NewX2,NewY2),
                   Displacement),
   strong OldX1 = NewX1, strong OldY1 = NewY1.
```

Since in this version the anchor constraints are stronger than the mouse constraints, now the line will stretch in the x direction, following the mouse, but its y position will remain constant. In other words, the mouse constraint on the new x value of *end2* will be satisfied, but the mouse constraint on the new y value will be overridden by the stronger constraint that it be the same as the old y value. This is the same behavior as was exhibited by the original ThingLab [Borning 81], but now produced as a consequence of declaratively represented hard and soft constraints.

In a similar manner, we can (without any hard thinking required) translate all of the ThingLab examples into HCLP($\mathcal{R}$). For the more complex examples, the HCLP code becomes tediously long. However, we envision such code being written automatically by the interactive graphics application, rather than by a programmer.

If nothing beyond expressing previously implemented interactive graphics examples in HCLP could be achieved, the current research would not be of great interest. However, since the full power of logic programming is available, we can do considerably more. For example, *filters* are a powerful metaphor for the declarative construction of user interfaces. In the filter browser described in [Ege et al. 87], the screen view of some source object is constructed by passing the object through a series of filters to produce the final image. Each filter is represented as a collection of constraints (some of which may be required and some non-required) relating its input and output. Thus the view is updated if the source is changed. Further, since the constraints are bidirectional, we can edit the image to make some change to the source. ThingLab supported

such filter networks for fixed topologies, but it was difficult to make the shape of the network depend on the data. Such dynamically configured constraint networks are needed, for instance, if we want to view a tree, applying a subfilter to each node in the tree to produce its screen image. Such a tree-viewing filter is simple to write in HCLP—we write a recursive `view_tree` rule that sets up a node-viewing filter for each corresponding node in the source and view trees.

```
view_tree(Source,Image) :-
   view_node(Source,Image),
   view_subtrees(Source,Image).

view_subtrees(Source,Image) :-
   leaf(Source), leaf(Image).

view_subtrees(Source,Image) :-
   left(Source,LS), right(Source,RS).
   left(Image,LI), right(Image,RI),
   view_tree(LS,LI), view_tree(RS,RI).

view_node(SourceNode,ImageNode) :- ...
```

As a final graphics example, illustrating the interaction between constraint hierarchies and logic programming, consider the problem of laying out an illustration of a binary tree. Suppose that the tree is represented by terms of the form `node(Value,Left,Right,X,Y)` and `leaf(Value,X,Y)`. `Value` is the value at each node. `Left` and `Right` are the children of the given interior node. Suppose that `X` and `Y` are initially unbound; our task is to bind them to appropriate values for each node. Suppose also that the tree must fit within a window. We will have a required minimum vertical spacing between levels in the tree, and a minimum horizontal spacing between the parent and the left and right children; and also somewhat larger preferred spacings. A recursive `layout` rule will set up the appropriate constraints on the `X` and `Y` variables in each node: hard constraints that enforce the minimum spacing restrictions and that force the entire image of the tree to lie within the window, and soft constraints that try to lay out the nodes using the preferred spacing. The tree will be layed out using the preferred spacing if possible; otherwise it will be squeezed down as needed to fit in the window. (The most appropriate

comparator for this application would be least-squares-better, which would distribute the compression over all the node positions.) And, of course, if the tree cannot be layed out so that the required constraints are satisfied, the goal would fail.

```
layout(node(Value,Left,Right,X,Y),
         Window_left,Window_right,Window_top,Window_bottom) :-
   /* require that the node lie within the window */
   required Window_left <= X, required X <= Window_right,
   required Window_top >= Y, required Y >= Window_bottom,
   /* get the X and Y positions of the left and right children */
   x(Left,LeftX), y(Left,LeftY),
   x(Right,RightX), y(Right,RightY),
   /* set up required constraints using the minimum spacing (5 units) */
   required Y-LeftY >= 5,
   required Y-RightY >= 5,
   required X-LeftX >= 5,
   required RightX-X >= 5,
   /* set up default constraints using the preferred spacing (10 units) */
   medium Y-LeftY = 10,
   medium Y-RightY = 10,
   medium X-LeftX = 10,
   medium RightX-X = 10,
   /* now recursively lay out the positions of the children */
   layout(Left,Window_left,Window_right,Window_top,Window_bottom),
   layout(Right,Window_left,Window_right,Window_top,Window_bottom).

layout(leaf(Value,X,Y),Window_left,Window_right,Window_top,Window_bottom) :-
   /* require that the leaf node lie within the window */
   required Window_left <= X, required X <= Window_right,
   required Window_top >= Y, required Y >= Window_bottom.
```

```
/* access rules to get the X and Y parts of an interior node or a leaf */
x( node(Value,Left,Right,X,Y) , X ).
y( node(Value,Left,Right,X,Y) , Y ).
x( leaf(Value,X,Y) , X).
y( leaf(Value,X,Y) , Y).
```

7.2 Planning and Scheduling

Here is a sample HCLP($\mathcal{R}$) program that determines when a group of people can meet and that will also find a meeting room for them.

```
free(alan,6,8).
free(bjorn,8,9).
free(john,11,12).
free(molly,10,12).
free(conference_room,8,10).
room(conference_room).

find_times([Person|More],Start,End) :-
     find_time_for_one(Person,Start,End),
     find_times(More,Start,End).
find_times([],Start,End).
```

```
find_time_for_one(Person,Start,End) :-
     free(Person,Start_Free,End_Free),
     medium Start_Free ≤ Start,
     medium End_Free ≥ End.

find_room(Room,Start,End) :-
     room(Room),
     free(Room,Start_Free,End_Free),
     strong Start_Free ≤ Start,
     strong End_Free ≥ End.
```

The following query finds a one hour meeting time for Alan, Bjorn, John, and Molly.

```
?- find_times([alan,bjorn,john,molly],S,E),
   find_room(Room,S,E),
   required E - S = 1.
```

The program processes the list of participants, accumulating constraints on the start and end time for each. For each person, *medium* constraints are added that the person be free during the meeting time. Also, we need a meeting room; the program looks for a meeting room, and adds a *strong* constraint that the room be free during the proposed time. (We didn't make it a *required* constraint, since perhaps we can persuade the other users of the room to move their meeting, or there may be some other constraint on everyone's time that takes priority over the room being free, such as a fire drill.) The program will succeed in finding a meeting time regardless of how solutions are chosen, as none of the conflicting constraints are at the required level.

If we are only considering each constraint individually, as with the local and regional comparators, then the program will return as its answer all one-hour intervals between 8:00 and 10:00. (All of these intervals satisfy the required constraint that the meeting last an hour, and the strong preference that the conference room be free. Since we can't satisfy everyone's personal preferences regarding the meeting time, in this case we don't try to distinguish further among the solutions.) For this program, the regional comparators return the same answers as their local counterparts. However, if we add a weaker constraint, for example one that weakly prefers meetings close to lunch time, the regional answers may be further refined and some of these

solutions may be rejected. (For the local comparators, the set of solutions wouldn't be affected by this change.)

Weighted-sum-metric-better also selects all one-hour intervals between 8:00 and 10:00. However, if we were to add another person to the list of attendants for the meeting, say someone who was free from 9:00 to 10:00, then weighted-sum-metric-better would select the hour beginning at 9:00. By minimizing the sum of the errors, this comparator attempts to "make the most people happy."

Weighted-sum-predicate-better yields an 8:00 meeting time as that is the time that satisfies the most people (one, in this case) while still satisfying the stronger meeting time constraints.

Least-squares-better chooses 8:45 as the desired meeting time. This comparator is similar to weighted-sum-metric-better in that the total error is being considered in finding a solution, but because the errors are being squared, outlying constraints (such as Alan's early meeting preference) tend to skew the results.

The answer using worst-case-better is 8:30 as this is the time that produces the smallest single error of any of the times from 8:00 to 10:00. In effect, no one person will be too put out by the results using this comparator.

We can conceive of scenarios where each of these solutions is most desirable. Normally, we might prefer to use a predicate comparator for scheduling meetings, so that we don't find ourselves meeting at strange times that are no good for anyone. Yet in some situations, such as deciding what time of year to meet, it is important to take exact error into account.

7.3 Document Formatting

In this example, we want to lay out a table on a page in the most visually satisfying manner. We achieve this by allowing the white space between rows to be an elastic length. It must be greater than zero (or else the rows would merge together), yet we strongly prefer that it be less than 10 (because too much space between rows is visually unappealing). We do not want this latter constraint to be required, however, since there are some applications that may need this much blank space between lines of the table. We prefer that the table fit on a single page of length 30 (units). There is a *weak* default constraint that the white space be 5, that is if it is possible without violating any of the other constraints. Finally, there is another *weak* constraint specifying the default type size.

```
table(PageLength, TypeSize,NumRow,WhiteSpace):-
     required (WhiteSpace + TypeSize) * NumRow = PageLength,
     required WhiteSpace > 0,
     strong WhiteSpace < 10,
     medium PageLength ≤ 30,
     weak WhiteSpace = 5,
     weak TypeSize = 11.
```

If we use a predicate comparator, then if the *medium* constraint cannot be satisfied and the table takes up more than one page, the *weak* constraint will be satisfied, resulting in WhiteSpace = 5. However, if we use a metric comparator, spacing between the rows will be as small as possible to minimize the error in the PageLength constraint at the *medium* level.

We can avoid this behavior by demoting the *medium* constraint to a *weak* one so that the size of the type, the white space between rows, and the number of pages all interact at the same level in the hierarchy. Weighted-sum-better will characteristically choose the solution that minimizes the error for the majority of the constraints, while worst-case-better finds the middle ground.

As demonstrated by this example, it may not be apparent until some experimentation has taken place what even constitutes a suitable solution. The user may need to experiment with using various comparators (or even combining them for different parts of the problem by breaking a goal into various subgoals and solving them using different comparators), and with different strengths on given constraints, to determine the desired solution.

7.4 Which Comparator to Use?

There has not yet been enough experience to make any conclusive statements about which comparators embedded in an HCLP language are most appropriate for which classes of problems. However, there is considerable work in related areas that sheds some light on the question. (The comparators are all derived from previous formalisms, rather than being ad hoc inventions.)

The global comparators weighted-sum-error-better, worst-case-error-better, and least-squares-error-better are all derived from (and are generalizations of) the standard statistical measures of deviation L_1-*norm*, L_∞-*norm*, and L_2-*norm* respectively. Locally-error-better is derived from the concept of a *vector minimum* (or *pareto-optimal point*, or *nondominated feasible solution*) in multiobjective linear programming problems [Murty 83]. In operations research, the

choice between an L_1-, L_∞-, or L_2-approximation seems often to be made on the class of constraints (for example, are they linear or nonlinear?) and the consequent difficulty of solving the resulting problem. The set of vector-minimum solutions is appealing mathematically—the only solutions that could reasonably be of interest belong to this set—but working with this set of solutions has not been particularly practical [Murty 83].

Our own work on constraint hierarchies originated as a rational reconstruction of the behavior of ThingLab and other constraint-based systems. Our recent work on constraint-based systems for user interface toolkits (ThingLab II [Maloney et al. 89, Maloney 91] and Multi-Garnet [Sannella & Borning 92]) has used the locally-predicate-better comparator. This choice has been based primarily on pragmatic rather than aesthetic or theoretical grounds: the existence of efficient incremental algorithms—DeltaBlue [Freeman-Benson et al. 90] and a derivative algorithm SkyBlue [Sannella 92]—for finding LPB solutions. For user interface applications, we do have extensive experience in the practical use of LPB [Sannella et al. 93]. It also been used by a considerable number of researchers at other institutions as well. LPB has generally proved quite satisfactory. However, for precise layout least-squares-better will often yield more aesthetic results. (The graphical layout system TRIP [Kamada & Kawai 91], for example, uses least-squares-better.)

7.5 Inter-Hierarchy Comparison in HCLP($\mathcal{R}$)

As discussed in Chapter 3 Section 3.3, it is useful in some applications to compare solutions arising from several different hierarchies. Let's return to a simple scheduling problem similar to that given in Section 7.2, but uncomplicated by the choice of a meeting room. That is, we only wish to select a meeting time for two people and we have a room that is available all day.

```
free(nate,8,12).
free(nate,18,21).
free(callie,17,21).
free(conference_room,8,21).
room(conference_room).
```

In this example, Nate is free at two separate times of the day—once before noon and once from early evening on. An HCLP($\mathcal{R}$) program using the weighted-sum-metric-better comparator would produce two answers for the query

```
?-   find_times([nate,callie],S,E),
     find_room(Room,S,E),
     required E - S = 1.
```

The first answer, meeting for an hour sometime between noon and 5:00 p.m., stems from the first rule selection for Nate. The second answer, meeting for an hour sometime between 6:00 p.m. and 9:00 p.m., arises from the second rule choice for Nate. In effect, two hierarchies are constructed here—one using the first and the other using the second free time for Nate. The second answer seems to be the "best" in that it completely satisfies both people's preferences. One way to achieve this answer using the constraint hierarchy theory is to allow a comparison between the solutions arising from the first hierarchy and those arising from the second as discussed in Chapter 3 Section 3.3.1, where the constraint hierarchy theory was extended to allow for such comparisons.

Extending the definition in this way gives rise to some nonmonotonic properties. These were originally discussed in [Wilson & Borning 89]. Using the previous program as an example, consider the answer to the previous query using weighted-sum-metric-better with respect to a program containing only the first rule for Nate, i.e. `free(nate,8,12).` In this case, the answer is meeting for an hour sometime between noon and 5:00 p.m. Now consider the answer to the previous query using the original program and using inter-hierarchy comparison. The answer is an hour between 6:00 p.m. and 9:00 p.m. — an answer that is no longer a superset of the answer to the abbreviated program. This property that has been introduced by the use of inter-hierarchy comparison is called the *nonmonotonicity* property, as the answers to HCLP programs may no longer monotonically increase as new clauses are added to them. A comparator is *monotonic* if the set of solutions to a set of constraint hierarchies Δ is a subset of the set of solutions to the set of constraint hierarchies $\Delta \cup \Gamma$, where Γ is any set of constraint hierarchies.

Definition. Let Δ and Γ be sets of constraint hierarchies. Let θ be a valuation. Let $\mathcal{C}$ be a comparator. Then $\mathcal{C}$ is *monotonic* iff

$$\forall\Delta\forall\Gamma\forall\theta \;\text{ if } \theta_H \in S_\Delta(\mathcal{C}) \text{ then } \theta_H \in S_{\Delta\cup\Gamma}(\mathcal{C}).$$

(Note that this definition does not apply to the local comparators since the set $S_{\Delta\cup\Gamma}$ is not defined for them.) A comparator that is not monotonic is *nonmonotonic*.

Proposition 11 *Let $\mathcal{D}$ be a nontrivial domain (i.e. a domain with more than one element). Then any comparator that respects the set of hierarchies is nonmonotonic.*

> **Proof:** Let Δ = {{`required X = a, weak X = b`}} and let Γ = {{`required X = b`}}, where `a` and `b` are two distinct elements in $\mathcal{D}$. (Both Δ and Γ are singleton sets.) Let $\mathcal{C}$ be a comparator that respects the set of hierarchies. $S_\Delta(\mathcal{C})$ consists of the valuation that maps `X` to `a` and $S_\Gamma(\mathcal{C})$ consists of the valuation that maps `X` to `b`. The valuation in $S_\Gamma(\mathcal{C})$ is better than that in $S_\Delta(\mathcal{C})$ and the valuation that maps `X` to `a` is not in $S_{\Delta\cup\Gamma}(\mathcal{C})$. Therefore $\mathcal{C}$ is nonmonotonic. ∎

There are many other examples of programs where inter-hierarchy comparison yields the most intuitive answers. Aside from the restriction to global comparators discussed in Chapter 3 Section 3.3.1 and the nonmonotonic properties discussed above, there are two other reasons why an HCLP interpreter restricts its comparisons to single hierarchies. The first and most important reason has to do with efficiency. Consider the following program fragment:

```
f(X):- g(X), medium X ≤ 1.
g(1).
g(X):- g(X - 1).
```

There is nothing in the definition of the global comparators that prevents the set of hierarchies Δ from being infinite. In practice, this can occur when rules are recursive, as demonstrated in the program listed above. In general, an interpreter using inter-hierarchy comparison would have to construct all the hierarchies arising from alternate rule choices, collect all the valuations that satisfy the required constraints in those hierarchies, and then compare them to find the solution set. In cases where the set of hierarchies is infinite, such a procedure will not return unless judicious pruning of the search tree allows infinite branches not to be traversed. For programs such as the one described above, in general there is no way to avoid an infinite search for the best solution. (To avoid such a search we would potentially need to solve the halting problem.) If, however, the *medium* constraint in the first rule were altered to *medium* $X > 0$, then all valuations for X that satisfied the predicate `g` would also satisfy all the constraints in their respective hierarchies. We would want an efficient implementation to make use of such information so that answers could be produced one at a time.

The second justification for preferring single hierarchy comparisons is for programs where we want all possible answers to a query. Consider the following program that attempts to characterize mealtimes.

```
free(callie,S,E):- strong S ≥ 18.

mealtime(breakfast,S,E):- S ≥ 6, E ≤ 10, E - S = 0.5
mealtime(lunch,S,E):- S ≥ 12, E ≤ 13, E - S = 1.0
mealtime(dinner,S,E):- S ≥ 17, E ≤ 20, E - S = 1.5

eat(Person,S,E):-
     mealtime(Meal,S,E),
     free(Person,S,E).
```

The first rule states that Callie is free all day, but that she strongly prefers that anything that is planned occur after 6:00 p.m. This may be reasonable for scheduling a get-together, but if we use this in conjunction with planning mealtimes, inter-hierarchy comparison will have Callie skipping breakfast and lunch. Instead, using the more standard intra-hierarchy comparisons, Callie's preference would have no effect on the other mealtimes (using a predicate comparator), but it would move the dinner hour to after 6:00 p.m.

Chapter 8

Related Work

8.1 Constraint Logic Programming Languages

HCLP builds on the CLP scheme [Cohen 90, Jaffar & Lassez 87]. Since HCLP is also a general scheme, it should be possible to implement HCLP languages for any of the domains, such as booleans, finite domains, or trees, supported by existing CLP languages (e.g., [Colmerauer 90, Dincbas et al. 88, Jaffar & Michaylov 87, Jaffar et al. 92, Satoh & Aiba 90, Sidebottoms & Havens 91, Van Hentenryck 89, Walinsky 89]).[1]

A number of CLP languages, for example CHIP [Dincbas et al. 88, Van Hentenryck 89], include a `minimize` operator. If an *a priori* lower bound B on the value of *Var* is known, then a call to `minimize(Var)` could be replaced by a soft constraint *medium* $Var = B$.[2] However, if an *a priori* bound is not known, then this simulation would not work. Reference [Borning et al. 92] describes how the constraint hierarchy theory can be extended to include objective functions. We

[1] Regarding Echidna [Sidebottoms & Havens 91], we should note that some of its constraint solving techniques make use of a hierarchy, but their meaning is quite different than the one we use here. In the case of Echidna, a hierarchy refers to a taxonomy, or a structuring of a discrete domain into subsets with similar properties. This allows the system to use a more efficient arc consistency algorithm.

[2] Actually, the simulation is not quite precise. Consider the CHIP goals `X>=0, X<=10, minimize(X), minimize(0-X)` and `X>=0, X<=10, minimize(0-X), minimize(X)`. These would give $X = 0$ and $X = 10$ respectively. However, the corresponding HCLP goals `required X>=0, required X<=10, medium X=0, medium X=10` and `required X>=0, required X<=10, medium X=10, medium X=0` would both yield the same answers: for example, the two answers $X = 0$ or $X = 10$ for locally-predicate-better, and the single answer $X = 5$ for worst-case-better and least-squares-better.

could similarly extend an HCLP language to include objective functions explicitly, which would handle `minimize` directly (modulo the footnoted comment).

It is also useful to consider the relation between soft constraints and objective functions from the other point of view: of expressing HCLP languages in a CLP language with a `minimize` operator. The latter sort of language would be a very convenient one in which to write an HCLP interpreter. The technique would be similar to that used in our first HCLP interpreter, which was written in CLP($\mathcal{R}$) (Chapter 6 Section 6.4.1). However, rather than just the locally-predicate-better comparator, we could implement other comparators as well in such an interpreter. For example, to implement weighted-sum-better, we would first reduce the goal, satisfying the hard constraints, and accumulating the soft constraints. We would then minimize the value of an expression that was the weighted sum of the errors of the constraints at the strongest non-required level (using the `minimize` operator), then the weighted sum of the errors of the next level, and so forth.

The `cc` family of languages [Saraswat et al. 91, Saraswat 89] generalize the CLP scheme to include such features as concurrency, atomic tell, and blocking ask; up to this point we haven't dealt with these additional issues in the HCLP framework.

Maher and Stuckey [Maher & Stuckey 89] give a definition of constraint hierarchies similar to the one in this paper. In their definition, pre-solutions for hierarchies perform the same function as the set S_0 does in our formulation. Then rather than using the **E** and **G** functions, Maher and Stuckey define a pre-measure that maps pre-solutions and sets of constraints to some scale. The resulting sequences can then be compared via a lexicographic ordering.

Satoh [Satoh 90] proposes a theory for constraint hierarchies using a meta-language to specify an ordering on the interpretations that satisfy the required constraints. The theory is quite general, and can accommodate all of the comparators described in Chapter 2 Section 2.3. However, since it is defined by second-order formulae, it is not in general computable. In subsequent work [Satoh & Aiba 91a, Satoh & Aiba 91b], Satoh and Aiba present an alternative theory that restricts the constraints to a single domain $\mathcal{D}$, so that they can be expressed in a first-order formula. This theory is similar to the one presented here, with the following differences: first, only the locally-predicate-better comparator is supported; second, the semantics of constraint hierarchies (as opposed to the semantics of HCLP) is described model theoretically rather than set theoretically; and third, the class of constraints is generalized from atomic constraints to

disjunctions of conjunctions of atomic constraints, i.e., constraints of the form

$$(c_{11} \wedge c_{12} \wedge \ldots \wedge c_{1n_1}) \vee (c_{21} \wedge c_{22} \wedge \ldots \wedge c_{2n_2}) \vee \ldots \vee (c_{m1} \wedge c_{m2} \wedge \ldots \wedge c_{mn_m})$$

Satoh and Aiba embed such constraints in the CLP language CAL [Satoh & Aiba 90], to yield an HCLP language CHAL [Satoh & Aiba 91a, Satoh & Aiba 91b]. CHAL includes two constraint solvers: an algebraic constraint solver for multi-variate polynomial equations, which uses Buchberger's algorithm to calculate Gröbner bases; and a boolean constraint solver for propositional boolean equations, which uses an extension of Buchberger's algorithm. Satoh and Aiba give examples illustrating each of these domains: a meeting scheduling problem and a gear design problem respectively. In each case both required and soft constraints are used. They also describe an algorithm for finding the locally-predicate-better solutions to a hierarchy, which improves on our algorithm discussed in Chapter 6 Section 6.4.1 by avoiding redundant calls to the solver. It finds solutions essentially by computing maximally consistent subsets of the soft constraints. However, this algorithm is a batch solver, in contrast to the DeltaStar-based incremental algorithm (Chapter 2 Section 6.4.2), and thus must re-compute its answers from scratch after every change to the set of constraints caused by an alternate rule choice. Finally, it should be mentioned that the characterization in [Satoh & Aiba 91a] states the definition of the set of solutions for a given constraint hierarchy in model-theoretic rather than set-theoretic terms, but doesn't deal with the *interactions* between rule choice and constraint hierarchies. We define constraint hierarchies and their solutions using sets, but describe the meaning of HCLP programs using both a model theory and a proof theory.

Ohwada and Mizoguchi [Ohwada & Mizoguchi 90] discuss the use of logic programming for building graphical user interfaces. Default constraints are instrumental in this application, since often only an incomplete specification of an object is given, yet complete information is needed to display a picture. Defaults provide a mechanism whereby information can be assumed in order to specify an object fully, yet it can be overridden, if necessary, as further information becomes known. Rather than a single default level, a hierarchy of default constraints is used to avoid the multiple extension problem. The hierarchy is implemented using the negation-as-failure rule, i.e., if the negation of a constraint is not known to hold, then the constraint can be assumed to be true. A problem with this approach is that it then becomes necessary to list all possible conflicts when a rule is being written in order to avoid inconsistencies. In HCLP, the need for consistency is assumed and there is no need to enumerate specifically those constraints that might conflict with the goal.

8.2 Other Constraint Languages

Outside of logic programming, other programming languages have supported constraints. Steele's Ph.D. dissertation [Steele 80] is one of the first such efforts; an important characteristic of his system is the maintenance of dependency information to support dependency-directed backtracking and to aid in generating explanations. Leler [Leler 87] describes Bertrand, a constraint language based on augmented term rewriting. Freeman-Benson and Borning [Freeman-Benson 90, Freeman-Benson & Borning 92a, Freeman-Benson & Borning 92b, Freeman-Benson 91] combine constraints with object-oriented, imperative programming. Kaleidoscope uses the same constraint hierarchy theory employed in HCLP to reconcile the assignment operation of imperative programming with declarative constraints: in Kaleidoscope, an assignment statement `x` $\leftarrow$ `x+5` is semantically a constraint relating states of x at successive times: $x_{t+1} = x_t + 5$. In addition, all variables have very weak equality constraints between their successive states, so that in the absence of stronger constraints, a variable will retain its value over time.

8.3 Applications

There is a substantial body of related research in the artificial intelligence community. Fox [Fox 87] discusses the problem of constraint-directed reasoning for job-shop scheduling, and allows the relaxation of constraints when conflicts occur, and context-sensitive selection and weighted interpretation of constraints. Descotte and Latombe [Descotte & Latombe 85] make compromises by selective backtracking among inconsistent constraints in a planner for manufacturing. Freuder [Freuder 92] gives a general model for partial constraint satisfaction problems (PCSPs) for variables ranging over finite domains, extending the standard CSP model. In Freuder's model, alternate CSPs are compared with the original problem using a metric on the problem space (as opposed to a metric on the solution space, as in our work). An optimal solution s to the original PCSP would be one in which the distance between the original problem and the new problem (for which s is an exact solution) is minimal. In an earlier CSP extension, Shapiro and Haralick [Shapiro & Haralick 81] define the concepts of exact and inexact matching of two structural descriptions of objects, and show that inexact matching is a special case of the inexact consistent labeling problem.

Decision analysis offers an approach for representing and quantifying the elements that enter into making complex decisions. Often these decisions must be made in the presence of uncer-

tainty, and frequently there exist multiple conflicting objectives that make the problem even more difficult to describe. Reference [von Winterfeldt & Edwards 86] presents methods for structuring decision-making problems, as well as formal models to solve those problems. One of these methods, multiple attribute utility theory (MAUT), provides general techniques for representing the trade-offs involved when there are multiple conflicting objectives. Although the problem domain (i.e. decision analysis with uncertainty) differs from that of HCLP, the procedures for solving problems using MAUT can be quite similar to constraint hierarchy techniques. For example, MAUT procedures look at each potential alternative solution to the decision and evaluate all of the different attributes that enter into the decision separately for each alternative. Then relative weights are assigned to each attribute and these weights and the single-attribute evaluations of alternatives are aggregated to produce an evaluation of the whole. One of the standard aggregation models is the weighted additive model which simply sums the weighted value for each attribute. In HCLP, on the other hand, a problem is formulated by determining which constraints make up the problem and what their relative strengths and weights are. Then a comparator is chosen to aggregate the errors at a given level and compare them. There is a clear relationship between the weighted additive model in MAUT and the weighted-sum-better comparator.

8.4 Reasoning

In non-monotonic reasoning, there are several related problem areas with different emphases. In default reasoning one tries to reason in the absence of complete information, making assumptions about things that are true or false in the absence of knowledge to the contrary. Reference [Ginsberg 87] is a collection of many of the classic papers in the area. Temporal reasoning [Shoham & Baker 92] deals with the difficulty of reaching conclusions about things that change over time and includes the well known frame problem, among others. In knowledge representation, beliefs are sometimes retracted, while the addition of new beliefs may often invalidate information that was previously held to be true. In explanation-based reasoning, or hypothetical reasoning, multiple theories exist to explain an observation, and the accumulation of new facts helps to reduce the number of acceptable explanations, or theories.

These areas are all related in a broad sense in that they involve reasoning in the presence of change: either change through time, change in knowledge, or change in observation. (Reference [del Val & Shoham 92] explores the relationship between temporal reasoning and belief update and

shows that the latter can be expressed in terms of the former.) In the case of default reasoning, new information may involve eliminating false assumptions, just as adding new constraints to a constraint hierarchy may override weaker constraints. Brewka [Brewka 89] describes an approach to representing default information with multiple levels of preference. In this framework, there are many levels of theories, some of which are more preferred than others. A preferred subtheory is obtained by taking a maximally consistent subset of the strongest level, and then adding as many formulas as possible from the next strongest level, and so on, without introducing any inconsistencies.

The problems involved in revising knowledge systems are discussed in [Gärdenfors & Makinson 88]. Formally, revision means adding new information. Contraction of a knowledge system arises when information must be retracted. Revision will often entail contraction as new information may invalidate old beliefs. Rationality postulates are used to ensure that contraction and expansion of the knowledge set is carried out in the best way possible. Intuitively, this means that the most minimal change is made to the theory while still incorporating the new information. This is similar to our own use of comparators and our requirements on the combining functions. (In fact, one of our motivations for defining and using constraint hierarchies arose from our work in interactive graphics and our desire that updates to the screen involve as little change as possible.) Revision can be viewed as adding a constraint to the hierarchy: first it is necessary to "contract," i.e. remove all constraints from the solution that are weaker than the one being added; then it is necessary to "expand," i.e. add all weaker constraints that are consistent with the revised set. Epistemic entrenchment is used to order the sentences in a knowledge set. Those sentences that are the most epistemically entrenched are those that are the most important and should not be removed from the knowledge set before other less entrenched ones. Again, this is similar to our use of levels in the hierarchy. One difference is that the ordering based on epistemic entrenchment is a natural ordering arising from the theory itself, while the ordering of the constraint hierarchy is imposed by the user.

Reiter [Reiter 88] describes integrity constraints that are used to ensure certain properties about the content of a knowledge base. These constraints can be viewed as meta-constraints in that they refer to what the knowledge base should contain, or "know," rather than to properties of the domain. Integrity constraints have been used to prefer one explanation, or hypothesis, over another by considering constraints that are false in the problem domain and false in each of two theories but which are true in the union of the two theories [Seki & Takeuchi 85]. Thus the

theories are mutually incompatible and there exists some "crucial literal" that can be used to discriminate between them because it is supported in one of the two theories, but not in the other. Reference [Sattar & Goebel 91] also discusses the use of crucial literals in hypothetical reasoning. By identifying these crucial literals and querying the user as to the truth of the literal, the number of explanations for a given set of observations can be minimized. Because of the power of the theorem solver used in their approach, integrity constraints are merely facts, corresponding to hard constraints in the constraint hierarchy formalism. Hypotheses can then be interpreted as soft, or default constraints (there is only one level). Once the truth value of a crucial literal is determined, then it becomes a fact and invalidates one (or more) of the hypotheses (defaults).

Despite the similarities discussed above, these approaches differ in their ultimate goal, or intended purpose. Default and temporal reasoning attempt to discover what will be true in a given situation, whereas hypothetical reasoning is concerned with explanations for observed phenomena. Belief revision is concerned with maintaining consistent information in knowledge sets, or databases. Our work in constraint hierarchies, and HCLP in particular, is focussed on computing answers to domain specific problems, and the soft constraints are used to narrow the solution space. Poole [Poole 90] characterizes certain types of reasoning based on who is allowed to choose the assumptions, or hypotheses, and whether the goal is known. Most uses of HCLP are with an unknown goal, and the assumptions are selected by the programmer (who labels the constraints), and the comparator (which selects the "best" answer).

Chapter 9

Conclusion

This dissertation describes the Hierarchical Constraint Logic Programming family of languages. HCLP is based on the marriage of constraint hierarchies and logic programming, two distinct, but compatible schemes. By integrating these two schemes, it is possible to solve problems originally formulated for a more imperative programming environment in an elegant and declarative fashion, as well as to solve new problems that could not be expressed in the CLP framework. Below we list the major contributions of this research, followed by promising areas of future work.

9.1 Contributions

This dissertation standardizes a great deal of the constraint hierarchy formalism by using the notion of combining functions and lexicographic ordering. Earlier work on constraint hierarchies separates the definitions for local and global comparators. The current definitions illustrate that the difference in these types of comparators lies in the way in which the errors are combined, and not in the concept of a solution per se. These current definitions also allow for the introduction of the regional comparators, and pave the way for additional comparators to be defined.

This dissertation also contributes to constraint hierarchy theory by defining the disorderly property that occurs when non-required constraints are added to a hierarchy, and by bringing together various propositions on constraint hierarchies in general. Chapter 3 discusses some of the extensions to constraint hierarchies that show the relationship between hard and soft constraints and work such as concurrent logic programming and nonmonotonic logic.

As discussed in Chapter 1, the work described in this dissertation can be viewed in two, inter-

related ways. Firstly, we can conceive of Hierarchical Constraint Logic Programming as merging the notion of the constraint hierarchy with standard logic programming. Secondly, we can view HCLP as an extension of CLP that allows for non-required constraints. Both of these views capture the contributions of HCLP. Merging hard and soft constraints with logic programming has allowed us to program with the constraint hierarchy in a way that previous work could not. Logic programming provides a powerful programming language that can be used to "wire" the constraints using conditionals, recursion,and other programming constructs. On the other hand, viewing HCLP as an extension of CLP allows us to place HCLP in a promising language family, to incorporate many of the CLP techniques for creating efficient implementations, and to build on the solid semantics of CLP.

One of the main contributions of this work is the development of a declarative semantics for HCLP. Many of the difficulties encountered in defining such a semantics have been discussed in Chapter 5. The semantics of constraint hierarchies alone do not have to contend with the interplay of the logic programming dimension of HCLP. Model theories for logic programming and constraint logic programming do not have to handle the non-obvious ways that non-required constraints can affect solutions. Yet, it is important to capture the declarative meaning of HCLP programs, if only to be able to compare them to other, related programming and reasoning formalisms. The standardized notation introduced in Chapter 2 allows us to see clearly the relation between the solution to a constraint hierarchy and the declarative meaning of an HCLP program, as well as the independence of the meaning from the particular comparator used.

Finally, this work includes the implementation of two HCLP interpreters – one for HCLP($\mathcal{R}$) using the locally predicate better comparator, and one for HCLP($\mathcal{R}$) using locally-metric-better, weighted-sum-metric-better, and worst-case-better. This latter implementation incorporates the DeltaStar algorithm and shows the effectiveness of separating the various components that comprise an HCLP interpreter. We have also presented various examples of application programs for HCLP in the domains of interactive graphics, planning and scheduling, and layout and document formatting.

9.2 Future Research

Future work in HCLP languages generally falls into two categories: improving existing implementations and extending the power of the languages themselves. In the first area there are

several ways to increase both the efficiency and usefulness of the current HCLP($\mathcal{R}$) interpreter; some of which are described in Chapter 6 Section 6.4.2. In addition it would be fruitful to experiment with other solvers, including solvers for non-linear equations, and solvers that support the least-squares-better comparator.

For further work on using HCLP for interactive graphics applications, it would be extremely useful to expand the current graphical interface. This would also allow further experimentation with various comparators, and investigation of their usefulness in solving specific problems.

In the realm of extending the power of HCLP itself, we could incorporate inter-hierarchy comparison into HCLP interpreters. As pointed out in Chapter 7 Section 7.5, while this would extend the usefulness of HCLP, it also has a potentially negative impact on efficiency. Still it presents a fruitful area for future research.

Another promising area of future work involves investigating concurrent forms of HCLP languages, possibly building on the work of the `cc` family of languages [Saraswat 89]. This would possibly entail adding the capacity for read-only and write-only annotations to HCLP programs, as well as finding methods to work around the disorderly property of constraint hierarchies. For example, certain non-required constraints could succeed even if all of the other constraints in the current hierarchy were unknown, if that constraint was logically validated by the known required constraints in the hierarchy.

A final area of future interest involves adding strength labels to predicates, as well as constraints. As with inter-hierarchy comparison, having both required and preferred predicates involves difficulties with both representing and computing a possibly infinite number of solutions to a subproblem. However, previous work in default reasoning within standard logic [Brewka 89] suggests that there is a theoretical basis for adding labels to predicates. In addition, current research on the Kaleidoscope programming language [Freeman-Benson & Borning 92b, Freeman-Benson & Borning 92a] indicates that these "soft" predicates could be quite useful, specifically in the case of user-generated constraints.

Bibliography

[Apt et al. 88] Krzysztof R. Apt, Howard R. Blair, and Adrian Walker. Towards a Theory of Declarative Knowledge. In Jack Minker, editor, *Foundations of Deductive Databases and Logic Programming*. Morgan Kaufmann Publishers, Inc., 1988.

[Bill & Lundell 90] Thomas Bill and Bertil Lundell. Using the Simplex Method for Solving Layout Constraints in Scientific Diagrams, September 1990. Project report for Master's degree in Computer Science and Engineering, School of Computer Science and Engineering, Royal Institute of Technology, Stockholm, Sweden. Work done at Departments of Statistics and Computer Science and Engineering, University of Washington.

[Borning 81] Alan Borning. The Programming Language Aspects of ThingLab, A Constraint-Oriented Simulation Laboratory. *ACM Transactions on Programming Languages and Systems*, 3(4):353–387, October 1981.

[Borning et al. 89] Alan Borning, Michael Maher, Amy Martindale, and Molly Wilson. Constraint Hierarchies and Logic Programming. In *Proceedings of the Sixth International Conference on Logic Programming*, pages 149–164, Lisbon, June 1989.

[Borning et al. 92] Alan Borning, Bjorn Freeman-Benson, and Molly Wilson. Constraint Hierarchies. *Lisp and Symbolic Computation*, 5(3):223–270, September 1992.

[Brewka 89] Gerhard Brewka. Preferred Subtheories: An Extended Logical Framework for Default Reasoning. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 1043–1048, August 1989.

[Cohen 90] Jacques Cohen. Constraint Logic Programming Languages. *Communications of the ACM*, 33(7):52–68, July 1990.

[Colmerauer 90] Alain Colmerauer. An Introduction to Prolog III. *Communications of the ACM*, pages 69–90, July 1990.

[del Val & Shoham 92] Alvaro del Val and Yoav Shoham. Deriving Properties of Belief Update from Theories of Action. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 584–589, July 1992.

[Descotte & Latombe 85] Yannick Descotte and Jean-Claude Latombe. Making Compromises among Antagonist Constraints in a Planner. *Artificial Intelligence*, 27(2):183–217, November 1985.

[Dincbas et al. 88] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Bertheir. The Constraint Logic Programming Language CHIP. In *Proceedings Fifth Generation Computer Systems-88*, 1988.

[Ege et al. 87] Raimund Ege, David Maier, and Alan Borning. The Filter Browser—Defining Interfaces Graphically. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 155–165, Paris, June 1987. Association Française pour la Cybernétique Économique et Technique.

[Emden & Kowalksi 76] M.H. Van Emden and R.A. Kowalksi. The Semantics of Predicate Logic as a Programming Language. *Journal of the ACM*, 23(4):733–742, October 1976.

[Fox 87] Mark S. Fox. *Constraint-Directed Search: A Case Study of Job-Shop Scheduling.* Morgan Kaufmann, Los Altos, California, 1987.

[Freeman-Benson & Borning 92a] Bjorn Freeman-Benson and Alan Borning. Integrating Constraints with an Object-Oriented Language. In *Proceedings of the 1992 European Conference on Object-Oriented Languages*, June 1992.

[Freeman-Benson & Borning 92b] Bjorn Freeman-Benson and Alan Borning. The Design and Implementation of Kaleidoscope'90, A Constraint Imperative Programming Language. In *Proceedings of the IEEE Computer Society International Conference on Computer Languages*, pages 174–180, April 1992.

[Freeman-Benson & Maloney 89] Bjorn Freeman-Benson and John Maloney. The DeltaBlue Algorithm: An Incremental Constraint Hierarchy Solver. In *Proceedings of the Eighth Annual IEEE Phoenix Conference on Computers and Communications*, Scottsdale, Arizona, March 1989. IEEE.

[Freeman-Benson & Wilson 90] Bjorn Freeman-Benson and Molly Wilson. DeltaStar, How I Wonder What You Are: A General Algorithm for Incremental Satisfaction of Constraint Hierarchies. Technical Report 90-05-02, Department of Computer Science and Engineering, University of Washington, May 1990.

[Freeman-Benson 90] Bjorn Freeman-Benson. Kaleidoscope: Mixing Objects, Constraints, and Imperative Programming. In *Proceedings of the 1990 Conference on Object-Oriented Programming Systems, Languages, and Applications, and European Conference on Object-Oriented Programming*, pages 77–88, Ottawa, Canada, October 1990. ACM.

[Freeman-Benson 91] Bjorn N. Freeman-Benson. *Constraint Imperative Programming.* PhD thesis, University of Washington, Department of Computer Science and Engineering, July 1991. Published as Department of Computer Science and Engineering Technical Report 91-07-02.

[Freeman-Benson et al. 89] Bjorn Freeman-Benson, John Maloney, and Alan Borning. The DeltaBlue Algorithm: An Incremental Constraint Hierarchy Solver. Technical Report 89-08-06, Department of Computer Science and Engineering, University of Washington, August 1989.

[Freeman-Benson et al. 90] Bjorn Freeman-Benson, John Maloney, and Alan Borning. An Incremental Constraint Solver. *Communications of the ACM*, 33(1):54–63, January 1990.

[Freeman-Benson et al. 92] Bjorn Freeman-Benson, Molly Wilson, and Alan Borning. DeltaStar: A General Algorithm for Incremental Satisfaction of Constraint Hierarchies. In *Proceedings of the Eleventh Annual IEEE Phoenix Conference on Computers and Communications*, Scottsdale, Arizona, March 1992. IEEE.

[Freuder 92] Eugene C. Freuder. Partial constraint satisfaction. *Artificial Intelligence*, 58(1–3):21 –70, December 1992.

[Gärdenfors & Makinson 88] Peter Gärdenfors and David Makinson. Revisions of Knowledge Systems Using Epistemic Entrenchment. In *Proceedings of the Second Conference on Theoretical Aspects of Reasoning About Knowledge*, pages 83–96, March 1988.

[Gelfond & Lifschitz 88] Michael Gelfond and Vladimir Lifschitz. The Stable Model Semantics for Logic Programming. In *Proceedings of the Fifth International Conference on Logic Programming*, pages 1070–1080, Seattle, August 1988.

[Ginsberg 87] Matthew L. Ginsberg, editor. *Readings in Nonmonotonic Reasoning.* Morgan Kaufmann, Los Altos, California, 1987.

[Gosling 83] James A. Gosling. *Algebraic Constraints.* PhD thesis, Carnegie-Mellon University, May 1983. Published as CMU Computer Science Department tech report CMU-CS-83-132.

[Heintze et al. 91] Nevin Heintze, Joxan Jaffar, Spiro Michaylov, Peter Stuckey, and Roland Yap. The CLP($\mathcal{R}$) Programmer's Manual Version 1.1. Technical report, IBM T.J. Watson Research Center, November 1991.

[Hill 74] R. Hill. LUSH-Resolution and its Completeness, 1974. DCL Memo 78.

[Ignizio 83] James P. Ignizio. Generalized Goal Programming. *Computers and Operations Research*, 10(4):277–290, 1983.

[Ignizio 85] James P. Ignizio. *Introduction to Linear Goal Programming.* Sage Publications, Beverly Hills, 1985. Sage University Paper Series on Qualitative Applications in the Social Sciences, 07-056.

[Jaffar & Lassez 87] Joxan Jaffar and Jean-Louis Lassez. Constraint Logic Programming. In *Proceedings of the Fourteenth ACM Principles of Programming Languages Conference*, Munich, January 1987.

[Jaffar & Michaylov 87] Joxan Jaffar and Spiro Michaylov. Methodology and Implementation of a CLP System. In *Proceedings of the Fourth International Conference on Logic Programming*, pages 196–218, Melbourne, May 1987.

[Jaffar et al. 92] Joxan Jaffar, Spiro Michaylov, Peter Stuckey, and Roland Yap. The CLP($\mathcal{R}$) Language and System. *ACM Transactions on Programming Languages and Systems*, 14(3):339–395, July 1992.

[Kamada & Kawai 91] Tomihisa Kamada and Satoru Kawai. A General Framework for Visualizing Abstract Objects and Relations. *ACM Transactions on Graphics*, 10(1):1–39, January 1991.

[Karloff 91] Howard Karloff. *Linear Programming.* Birkäuser, 1991.

[Konopasek & Jayaraman 84] M. Konopasek and S. Jayaraman. *The TK!Solver Book.* Osborne/McGraw-Hill, Berkeley, CA, 1984.

[Leler 87] William Leler. *Constraint Programming Languages.* Addison-Wesley, 1987.

[Lloyd 84] John W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 1984.

[Mackworth 77] Alan K. Mackworth. Consistency in Networks of Relations. *Artificial Intelligence*, 8(1):99–118, 1977.

[Maher & Stuckey 89] Michael J. Maher and Peter J. Stuckey. Expanding Query Power in Constraint Logic Programming. In *Proceedings of the North American Conference on Logic Programming*, Cleveland, October 1989.

[Maher 87] Michael J. Maher. Logic Semantics for a Class of Committed-choice Programs. In *Proceedings of the Fourth International Conference on Logic Programming*, pages 858–876, Melbourne, May 1987.

[Maier & Warren 88] David Maier and David S. Warren. *Computing With Logic.* The Benjamin/Cummings Publishing Company, Inc., 1988.

[Maloney 91] John Maloney. *Using Constraints for User Interface Construction.* PhD thesis, Department of Computer Science and Engineering, University of Washington, August 1991. Published as Department of Computer Science and Engineering Technical Report 91-08-12.

[Maloney et al. 89] John Maloney, Alan Borning, and Bjorn Freeman-Benson. Constraint Technology for User-Interface Construction in ThingLab II. In *Proceedings of the 1989 ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, New Orleans, October 1989. ACM.

[Murty 83] Katta G. Murty. *Linear Programming.* Wiley, 1983.

[Myers et al. 90a] Brad A. Myers, Dario Guise, Roger B. Dannenberg, Brad Vander Zanden, David Kosbie, Philippe Marchal, Ed Pervin, Andrew Mickish, and John A. Kolojejchick. The Garnet Toolkit Reference Manuals: Support for Highly-Interactive Graphical User Interfaces in Lisp. Technical Report CMU-CS-90-117, Computer Science Dept, Carnegie Mellon University, March 1990.

[Myers et al. 90b] Brad A. Myers, Dario Guise, Roger B. Dannenberg, Brad Vander Zanden, David Kosbie, Philippe Marchal, and Ed Pervin. Comprehensive Support for Graphical, Highly-Interactive User Interfaces: The Garnet User Interface Development Environment. *IEEE Computer*, 23(11):71–85, November 1990.

[Nelson 85] Greg Nelson. Juno, A Constraint-Based Graphics System. In *SIGGRAPH '85 Conference Proceedings*, pages 235–243, San Francisco, July 1985. ACM.

[Ohwada & Mizoguchi 90] Hayato Ohwada and Fumio Mizoguchi. A Constraint Logic Programming Approach for Maintaining Consistency in User-Interface Design. In *Proceedings of the 1990 North American Conference on Logic Programming*, pages 139–153. MIT Press, October 1990.

[Poole 90] David Poole. Hypo-deductive Reasoning for Abduction, Default Reasoning, and Design. In *Proceedings of the AAAI Spring Symposium on Automated Abduction*, 1990.

[Press et al. 86] William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. *Numerical Recipes: The Art of Scientific Computing.* Cambridge University Press, 1986.

[Przymusinski 88] Teodor C. Przymusinski. On the Declarative Semantics of Deductive Databases and Logic Programs. In Jack Minker, editor, *Foundations of Deductive Databases and Logic Programming.* Morgan Kaufmann Publishers, Inc., 1988.

[Reiter 88] Raymond Reiter. On Integrity Constraints. In *Proceedings of the Second Conference on Theoretical Aspects of Reasoning About Knowledge*, pages 97–112, March 1988.

[Robinson 66] A. Robinson. *Non-Standard Analysis.* North-Holland Publishing Company, Amsterdam, 1966.

[Sannella & Borning 92] Michael Sannella and Alan Borning. Multi-Garnet: Integrating Multi-Way Constraints with Garnet. Technical Report 92-07-01, Department of Computer Science and Engineering, University of Washington, September 1992.

[Sannella 92] Michael Sannella. The SkyBlue Constraint Solver. Technical Report 92-07-02, Department of Computer Science and Engineering, University of Washington, December 1992.

[Sannella et al. 93] Michael Sannella, John Maloney, Bjorn Freeman-Benson, and Alan Borning. Multi-way versus One-way Constraints in User Interfaces: Experience with the DeltaBlue Algorithm. *Software—Practice and Experience*, 1993. In Press.

[Saraswat 85] Vijay A. Saraswat. Problems with Concurrent Prolog. Technical Report CS-86-100, Carnegie-Mellon University, May 1985. Revised January 1986.

[Saraswat 89] Vijay Anand Saraswat. *Concurrent Constraint Programming Languages.* PhD thesis, Carnegie-Mellon University, Computer Science Department, January 1989.

[Saraswat et al. 91] Vijay A. Saraswat, Martin Rinard, and Prakash Panangaden. Semantic Foundations of Concurrent Constraint Programming. In *Proceedings of the Eighteenth Annual Principles of Programming Languages Symposium.* ACM, 1991.

[Satoh & Aiba 90] Ken Satoh and Akira Aiba. CAL: A Theoretical Background of Constraint Logic Programming and its Applications (Revised). Technical Report TR-537, Institute for New Generation Computer Technology, Tokyo, February 1990.

[Satoh & Aiba 91a] Ken Satoh and Akira Aiba. Computing Soft Constraints by Hierarchical Constraint Logic Programming. Technical Report TR-610, Institute for New Generation Computer Technology, Tokyo, January 1991.

[Satoh & Aiba 91b] Ken Satoh and Akira Aiba. The Hierarchical Constraint Logic Language CHAL. Technical Report TR-592, Institute for New Generation Computer Technology, Tokyo, September 1991.

[Satoh 90] Ken Satoh. Formalizing Soft Constraints by Interpretation Ordering. In *Proceedings of the European Conference on Artificial Intelligence*, 1990.

[Sattar & Goebel 91] Abdul Sattar and Randy Goebel. Using Crucial Literals to Select Better Theories. *Computational Intelligence*, 7:11–22, 1991.

[Seki & Takeuchi 85] H. Seki and A. Takeuchi. An Algorithm for Finding a Query Which Discriminates Competing Hypotheses. Technical Report 143, Institute for New Generation Computer Technology, 1985.

[Shapiro & Haralick 81] Linda Shapiro and Robert Haralick. Structural Descriptions and Inexact Matching. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-3(5):504–519, September 1981.

[Shapiro 86] Ehud Shapiro. Concurrent Prolog: A Progress Report. *IEEE Computer*, 19(8):44–58, August 1986.

[Shoham & Baker 92] Yoav Shoham and Andrew Baker. Nonmonotonic Temporal Reasoning, 1992. To appear in Handbook of Artificial Intelligence and Logic Programming, D. Gabbay, Ed.

[Shoham 88] Yoav Shoham. *Reasoning About Change: Time and Causation from the Standpoint of Artificial Intelligence.* The MIT Press, 1988.

[Sidebottoms & Havens 91] Greg Sidebottoms and William S. Havens. Hierarchical Arc Consistency Applied to Numeric Processing in Constraint Logic Programming. Technical Report 91-06, Centre for Systems Science and School of Computing Science, Simon Fraser University, August 1991.

[Steele 80] Guy L. Steele. *The Definition and Implementation of a Computer Programming Language Based on Constraints.* PhD thesis, MIT, August 1980. Published as MIT-AI TR 595, August 1980.

[Sterling & Shapiro 86] Leon Sterling and Ehud Shapiro. *The Art of Prolog.* MIT Press, 1986.

[Sutherland 63] Ivan Sutherland. Sketchpad: A Man-Machine Graphical Communication System. In *Proceedings of the Spring Joint Computer Conference.* IFIPS, 1963.

[Van Hentenryck 89] Pascal Van Hentenryck. *Constraint Satisfaction in Logic Programming.* MIT Press, Cambridge, MA, 1989.

[von Winterfeldt & Edwards 86] Detlof von Winterfeldt and Ward Edwards. *Decision Analysis and Behavioral Research.* Cambridge University Press, 1986.

[Walinsky 89] Clifford Walinsky. CLP(Σ^*): Constraint Logic Programming with Regular Sets. In *Proceedings of the Sixth International Conference on Logic Programming*, pages 181–196, Lisbon, June 1989.

[Wilson & Borning 89] Molly Wilson and Alan Borning. Extending Hierarchical Constraint Logic Programming: Nonmonotonicity and Inter-Hierarchy Comparison. In *Proceedings of the North American Conference on Logic Programming*, Cleveland, October 1989.

[Wilson & Borning 93] Molly Wilson and Alan Borning. Hierarchical constraint logic programming, 1993. To appear in the *Journal of Logic Programming*.

[Wilson 91] Molly Wilson. The Semantics of Hierarchical Constraint Logic Programming. Technical Report 91-02-04, Department of Computer Science and Engineering, University of Washington, February 1991.

Appendix A

An Algorithm for Interpreting HCLP Programs

```
PROCEDURE Interpret(Goal,Program);
VAR Succeeded : Boolean;
    Answer : Set of Constraints;
    Untried, Satisfied, Unsatisfied : Hierarchy;
    C1 : Constraint;

BEGIN

  (* Make a call to CLP and get an answer to the predicates and required constraints using the
standard CLP machinery. Also return the hierarchy of non-required constraints in the variable
Untried. The repeat loop causes the CLP engine to backtrack, producing alternate answers. *)

LOOP
    CLP(Goal,Program,Succeeded,Answer,Untried);
    IF NOT Succeeded THEN
        Output "no more answers";
        RETURN;
    END; (* IF *)
```

```
        FOR each constraint C1 in Untried DO
            IF Answer implies C1 is false THEN
                add C1 to Unsatisfied;
                remove C1 from Untried;
            ELSIF Answer implies C1 THEN
                add C1 to Satisfied;
                remove C1 from Untried;
            ELSE
            (* Answer says nothing about C1. C1 remains in Untried *)
            END; (* IF *)
        END; (* FOR *)

    (* Make the call to the constraint hierarchy solver. If Answer is empty, then the solver will output
    the empty set as an answer and then return. *)

        Solve(Answer,Satisfied,Untried,Unsatisfied);
    END; (* LOOP *)
    END Interpret.



    PROCEDURE Solve(CurrentAnswer : Set of Constraints;
                            Satisfied, Untried, Unsatisfied : Hierarchy);
    VAR S : Set of Constraints;
        C1, C2 : Constraint;
        NewAnswer : Set of Constraints;
        NewSatisfied, NewUntried, NewUnsatisfied : Hierarchy;
```

```
BEGIN
(* If all the constraints have been tried, then we have an answer *)
IF Untried is empty THEN
    Output CurrentAnswer;
    RETURN;
END;

(* Explore all answers starting with the strongest constraints *)

S:= set of all constraints at the strongest level in Untried;

FOR each constraint C1 in S DO

(* Note that C1 is consistent with the current answer since it is not in Unsatisfied. Call constraint
solver to combine the current answer and C1. We assume that the constraint solver is powerful
enough either to return a new set of constraints or to fail if there are no substitutions that satisfy
the constraints in the CurrentAnswer and C1. *)

    NewSatisfied:= Satisfied;
    NewUntried:= Untried;
    NewUnsatisfied:= Unsatisfied;
    NewAnswer:= ConstraintSolver(CurrentAnswer,C1);

(* Now update Satisfied and Unsatisfied lists. Note that C1 will be removed from the list of Untried
constraints and put in the list of Satisfied constraints. Any constraint that is not satisfied by the
new answer will be one at a level equal to or weaker than C1, because of the way S is chosen. So
we still have a valid answer. *)

    FOR each constraint C2 in Untried DO
        IF NewAnswer implies C2 is false THEN
            add C2 to NewUnsatisfied;
            remove C2 from NewUntried;
        ELSIF NewAnswer implies C2 THEN
            add C2 to NewSatisfied;
            remove C2 from NewUntried;
```

```
        ELSE
        (* NewAnswer says nothing about C2. C2 remains in NewUntried *)
        END;
    END; (* FOR *)


(* Make recursive call. This call to Solve creates a new node in the implicit search tree. Each
C2 in Untried can be seen as labeling an edge to a child of the current node. *)

    Solve(NewAnswer,NewSatisfied,NewUntried,NewUnsatisfied);
    (* Re-initialize hierarchies *)
END; (* FOR *)
END Solve.
```