# Extending Hierarchical Constraint Logic Programming: Nonmonotonicity and Inter-Hierarchy Comparison

Molly Wilson and Alan Borning

Technical Report 89-05-04
Department of Computer Science and Engineering
University of Washington
July 1989

## Abstract

Hierarchical Constraint Logic Programming languages extend Constraint Logic Programming to include constraint hierarchies. These languages provide both required constraints and default constraints of various strengths. In the original definition of HCLP, alternate solutions to a given constraint hierarchy were compared, and only the "best" solutions were returned. However, there was no attempt to compare solutions arising from different choices of rules in the logic program. In many practical applications of HCLP, to rule out unintuitive solutions we do need to make such *inter-hierarchy* comparisons. Such comparisons introduce nonmonotonic behavior in HCLP programs. We define two related nonmonotonicity properties of HCLP languages, and compare these properties with those of standard nonmonotonic logics. The nonmonotonicity properties create novel implementation problems, which we discuss, while at the same time extending the usefulness of HCLP languages.

Authors' addresses:

Department of Computer Science and Engineering, FR-35
University of Washington
Seattle, Washington 98195

internet: molly@cs.washington.edu
borning@cs.washington.edu

This is a preprint of a paper that will appear in the *Proceedings of the 1989 North American Conference on Logic Programming*, to be held October 1989 in Cleveland, Ohio.

# 1 Introduction

Constraints have proven useful for a variety of applications, including geometric layout, physical simulations, and user interface design. Many such applications require some notion of defaults and preferences—constraints that should hold in the absence of conflicting information or those that we would prefer to be satisfied whenever possible. Consider, for example, the problem of laying out a table in a document. We would like the table to fit on a single page while still leaving adequate white space between rows. This can be represented as the interaction of two constraints: a required constraint that the height of the blank space between lines be greater than zero, and a preferred constraint that the entire table fit on one page. In this way we can make full use of the constraint paradigm, by representing these defaults and preferences declaratively, as constraints, rather than encoding them in the procedural parts of the language.

A theory of constraint hierarchies was developed in [1] which allows a user to specify not only constraints that must hold, but also weaker constraints at an arbitrary number of strengths. This constraint hierarchy scheme is parameterized by a comparator $\mathcal{C}$ that allows us to compare different possible solutions to a single hierarchy and select the best ones. Recently this constraint hierarchy paradigm was integrated with the Constraint Logic Programming scheme [6] to produce Hierarchical Constraint Logic Programming [2]. Like CLP, HCLP is parameterized by $\mathcal{D}$, the domain of the constraints. In addition, HCLP is parameterized by the comparator $\mathcal{C}$. This integration of CLP and constraint hierarchies allows the full programming capabilities and theoretical foundation of logic programming to complement the expressiveness of preferential constraints.

In [2] we described a prototype implementation of HCLP($\mathcal{R}, \mathcal{LPB}$), where $\mathcal{R}$ is the domain of real numbers and $\mathcal{LPB}$ is the *locally-predicate-better* comparator (defined in the next section). Experience with writing programs in HCLP($\mathcal{R}, \mathcal{LPB}$) has provided us with many examples where the $\mathcal{LPB}$ comparator fails to rule out unintuitive solutions. This is a result not simply of using a predicate comparator, but also of restricting the comparator to select among valuations arising from a single constraint hierarchy. This paper extends the notion of comparators to apply to inter-hierarchy comparisons. The earlier definitions of comparators are then shown to be special cases involving a single hierarchy. Given these new definitions, it will be easy to see that all the new inter-hierarchy comparators that we introduce exhibit nonmonotonic behavior. We also note some other interesting properties of constraint hierarchies, and show how using nonprimitive constraints can solve many common problems without the need for inter-hierarchy comparisons. Finally, we discuss some related work.

# 2 HCLP Review

## 2.1 Constraints in Logic Programming

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 $p, q_1, \ldots, q_m$ are predicate symbols, $c_1, \ldots, c_n$ are constraints, and $\mathbf{t}$ denotes a list of terms.

In HCLP, rules are of the form

$$p(\mathbf{t}) \ :- \ q_1(\mathbf{t}), \ldots, q_m(\mathbf{t}),\ s_1 c_1(\mathbf{t}), \ldots, s_n c_n(\mathbf{t}).$$

where $s_i$ indicates the strength of the corresponding constraint $c_i$. Symbolic names are given to the different strengths of constraints. The user defines an arbitrary number of names and their corresponding strengths. One strength, the "required" strength, is special, in that the constraints it labels *must* be satisfied. The other strengths all denote non-required constraints. If the $s_i$ are

all required, then clearly the program is equivalent to the same program in standard CLP with the strengths omitted.

Operationally, goals are executed as in CLP, temporarily ignoring the non-required constraints, except to accumulate them. After a goal has been successfully reduced, there may still be non-ground variables in the solution. The accumulated hierarchy of non-required constraints is solved, using a method appropriate for the domain and the comparator $\mathcal{C}$, thus further refining the values of these variables. Additional answers may be produced by backtracking. As with CLP, constraints can be used multi-directionally, and the scheme can accommodate collections of constraints that cannot be solved by simple forward propagation methods.

### 2.2 HCLP Example

The following is an example from the domain of document formatting. 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 30 lines. Finally there is a default constraint that the white space be 5, that is if it is possible without violating any of the other constraints. Note that if the prefer constraint cannot be satisfied and the table takes up more than a page, then the default constraint will be satisfied.

```
/* set up symbolic names for constraint strengths */
levels([required,strong_prefer,prefer,default]).

table(PageLength, TypeSize,NumRow,WhiteSpace):-
     required (WhiteSpace + TypeSize) * NumRow = PageLength,
     required WhiteSpace > 0,
     strong_prefer WhiteSpace < 10,
     prefer PageLength ≤ 30,
     default WhiteSpace = 5.
```

## 3 More HCLP Examples

The following are examples of HCLP programs that do not behave quite as expected. Although the solutions given by the HCLP interpreter are correct with respect to the semantics of HCLP, many of them are not intuitive. This behavior results from HCLP's inability to discriminate between solutions arising from different rule choices. In the original definitions, comparators select the best solutions for a *single* hierarchy. We call this an *intra-hierarchical* comparison. The following example suggests the need for *inter-hierarchical* comparisons.

```
f(X) :- strong_prefer X > 3, g(X).
g(5).
g(1).
```

Given the goal `f(A)`, HCLP would first return the answer `A = 5`, and on backtracking `A = 1`, since these answers arise from different choices of the rule for `g`, even though the `strong_prefer X > 3` constraint is satisfied for one answer and not the other. In reality, there are two constraint hierarchies produced in attempting to satisfy this goal. One is the hierarchy `required A = 5,`

`strong_prefer A > 3`. The other is `required A = 1, strong_prefer A > 3`. We would like some way of preferring the first answer because it satisfies more of the constraints in its respective hierarchy than does the second answer.

There are many other examples where an inter-hierarchy comparison results in the removal of undesirable solutions from the set of answers. The following is a program that computes the least number of coins required to make change for a specified amount, subject to the number of available coins of given denominations. Without inter-hierarchy comparison, solutions to the goal `makechange(16,Quarters,Dimes,Nickels,Pennies)` will essentially ignore the preferred constraints because the values for `Quarters, Dimes, Nickels, Pennies` will be bound by the `int` predicate. The desired solution, namely `Quarters=0`, `Dimes=1`, `Nickels=1`, and `Pennies=1`, is returned by the HCLP interpreter. However, returning `Pennies=16` is one solution among many others that is also given.

```
levels([required,prefer]).

makechange(Amount,Q,D,N,P):-
     required Q * 25 + D * 10 + N * 5 + P * 1 = Amount,
     available(AQ,AD,AN,AP),
     required P ≤ AP, required N ≤ AN,
     required D ≤ AD, required Q ≤ AQ,
     int(Q), int(D), int(N), int(P),
     prefer P < 5, prefer N < 2, prefer D < 3.

int(0).
int(X):- required X > 0, int(X - 1).

available(10,2,4,16).
```

In the next section, we first review the definitions given in [2] for solutions to a constraint hierarchy. We then extend this notion by defining the set of solutions to many constraint hierarchies. This lays the theoretical foundation for inter-hierarchy comparators and will allow us to rule out the unintuitive solutions described above.

# 4 Comparators

## 4.1 Definitions

A constraint is of the form $p(t_1, \ldots, t_n)$, where $p$ is an $n$-ary symbol in the constraint predicate symbols $\Pi_{\mathcal{D}}$ of the language, and each $t_i$ is a term. $\Pi_{\mathcal{D}}$ must include $=$. A *labelled constraint* is a constraint labelled with a strength, written $sc$, where $s$ is a strength and $c$ is a constraint.

A *constraint hierarchy* is a multiset of labelled constraints. Given a constraint hierarchy $H$, let $H_0$ denote the required constraints in $H$, with their labels removed. In the same way, we define the sets $H_1, H_2, \ldots$ for levels $1, 2, \ldots$. We also define $H_k = \emptyset$ if level $k$ doesn't exist in hierarchy $H$.

A *solution* to a constraint hierarchy $H$ will consist of a valuation for the free variables in $H$, i.e., a function that maps the free variables in $H$ to elements in the domain $\mathcal{D}$. We wish to define the set $S$ of all solutions to $H$. Each valuation in $S$ must be such that after it is applied all the required constraints hold. In addition, we want each valuation in $S$ to be such that it satisfies the non-required constraints as well as possible, respecting their relative strengths. To formalize this desire, we first define the set $S_0$ of valuations such that all the $H_0$ constraints hold. Then, using $S_0$, we define the desired set $S$ by eliminating all potential valuations that are worse than some other potential valuation using the comparator *better*. (In the definition $c\theta$ denotes the result of applying

the valuation $\theta$ to $c$.)

$$\begin{aligned} S_0 &= \{\theta \mid \forall c \in H_0\ c\theta \text{ holds}\} \\ S &= \{\theta \mid \theta \in S_0 \wedge \forall \sigma \in S_0\ \neg better(\sigma, \theta, H)\} \end{aligned}$$

There are many plausible candidates for comparators. We insist that *better* be irreflexive and transitive. We also 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 $k$:

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

We now define several different comparators. In the definitions we will need an error function $e(c\theta)$ that returns a non-negative real number indicating how nearly constraint $c$ is satisfied for a valuation $\theta$. This function must have the property that $e(c\theta) = 0$ if and only if $c\theta$ holds. 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. For a domain that is a metric space, we can instead use its metric in computing the error: for example, the error for $X = Y$ would be the distance between $X$ and $Y$.

The first of the comparators, *locally-better*, considers each constraint in $H$ individually.

**Definition.** A valuation $\theta$ is *locally-better* than another valuation $\sigma$ if, for each of the constraints through some level $k-1$, the error after applying $\theta$ is equal to that after applying $\sigma$, and at level $k$ the error is strictly less for at least one constraint and less than or equal for all the rest.

$$\begin{aligned} &locally\text{-}better(\theta, \sigma, H) \equiv \\ &\quad \exists k > 0 \text{ such that} \\ &\qquad \forall i \in 1 \ldots k-1\ \forall p \in H_i\ \ e(p\theta) = e(p\sigma) \\ &\qquad \wedge\ \exists q \in H_k\ \ e(q\theta) < e(q\sigma) \\ &\qquad \wedge\ \forall r \in H_k\ \ e(r\theta) \leq e(r\sigma) \end{aligned}$$

Next, we define a schema *globally-better* for global comparators. The schema is parameterized by a function $g$ that combines the errors of all the constraints $H_i$ at a given level.

**Definition.** A valuation $\theta$ is *globally-better* than another valuation $\sigma$ if, for each level through some level $k-1$, the combined errors of the constraints after applying $\theta$ is equal to that after applying $\sigma$, and at level $k$ it is strictly less.

$$\begin{aligned} &globally\text{-}better(\theta, \sigma, H, g) \equiv \\ &\quad \exists k > 0 \text{ such that} \\ &\qquad \forall i \in 1 \ldots k-1\ \ g(\theta, H_i) = g(\sigma, H_i) \\ &\qquad \wedge\ g(\theta, H_k) < g(\sigma, H_k) \end{aligned}$$

Using *globally-better*, we now define three global comparators, using different combining functions $g$. The weight for constraint $p$ is denoted by $w_p$. Each weight is a positive real number.

$$
\begin{aligned}
\textit{weighted-sum-better}(\theta, \sigma, H) &\equiv \textit{globally-better}(\theta, \sigma, H, g) \\
\text{where } g(\tau, H_i) &\equiv \sum_{p \in H_i} w_p e(p\tau)
\end{aligned}
$$

$$
\begin{aligned}
\textit{worst-case-better}(\theta, \sigma, H) &\equiv \textit{globally-better}(\theta, \sigma, H, g) \\
\text{where } g(\tau, H_i) &\equiv \max \{w_p e(p\tau) \mid p \in H_i\}
\end{aligned}
$$

$$
\begin{aligned}
\textit{least-squares-better}(\theta, \sigma, H) &\equiv \textit{globally-better}(\theta, \sigma, H, g) \\
\text{where } g(\tau, H_i) &\equiv \sum_{p \in H_i} w_p e(p\tau)^2
\end{aligned}
$$

Finally, we define an important special case of *locally-better*, namely *locally-predicate-better*, that uses the the trivial error function that returns 0 if the constraint is satisfied and 1 if it is not. Similarly, we define a special case of *weighted-sum-better*, namely *unsatisfied-count-better*, that uses the trivial error function and weights of 1 on each constraint.

## 4.2 Extended Definitions

A solution to a *set* of constraint hierarchies $\Delta$ will consist of a valuation for all the free variables in $\Delta$. Normally, the set $\Delta$ will consist of hierarchies that arise from alternate rule choices in a program. We wish to generalize the previous definitions so that the set $S$ contains all solutions to $\Delta$, rather than just to a single hierarchy. In all cases where $\Delta$ consists of a single hierarchy, the following definitions are equivalent to those given above.

We first define the set $S_0$ of valuations that satisfy all the required constraints in some hierarchy in $\Delta$. Each valuation $\theta$ in $S_0$ is annotated by the hierarchy $H$ that it satisfies. Using $S_0$, we define the set $S$ as before, only now the comparator *better* is parameterized by a set of constraint hierarchies. Thus we eliminate potential valuations that are worse than some other from any hierarchy in $\Delta$.

$$
\begin{aligned}
S_0 &= \{\theta_H \mid \forall c \in H_0 \; c\theta \text{ holds}\} \\
S &= \{\theta_H \mid \theta_H \in S_0 \wedge \forall \sigma_J \in S_0 \; \neg \textit{better}(\sigma_J, \theta_H, \Delta)\}
\end{aligned}
$$

We refer to the set of solutions $S$ as $S_\Delta$ when there might be ambiguity about the set of hierarchies being solved. $S(\mathcal{C})$ is the set of solutions to a set of constraint hierarchies using the comparator $\mathcal{C}$.

As before, we insist that our comparators be irreflexive and transitive. We also insist that they 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 $k$.

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

Because the *locally-better* comparators consider each constraint in the hierarchy individually to compare how well different valuations satisfy that constraint, we do not redefine those comparators to compare solutions to different hierarchies. In other words, *locally-better* is defined only if $\Delta$ consists of a single hierarchy. Although one can extend the definition to allow inter-hierarchy comparisons, some programs then yield unintuitive solutions, as the example in Section 8 demonstrates.

We do, however, extend *globally-better* to compare valuations arising from different hierarchies. As before, this schema is parameterized by a function $g$ that combines the errors of all the constraints $H_i$ at a given level.

**Definition.** A valuation $\theta_H$ is *globally-better* than another valuation $\sigma_J$ if, for each level through some level $k-1$, the combined errors of the constraints after applying $\theta$ to the constraints in hierarchy $H \in \Delta$ is equal to that after applying $\sigma$ to the constraints in hierarchy $J \in \Delta$, and at level $k$ it is strictly less. ($H$ and $J$ can be either the same or different hierarchies. If they are the same, then the following definition is equivalent to the one for intra-hierarchy comparison.)

$$\begin{aligned} &\textit{globally-better}(\theta_H, \sigma_J, \Delta, g) \equiv \\ &\quad \exists k > 0 \ \text{ such that} \\ &\qquad \forall i \in 1 \ldots k-1 \ \ g(\theta, H_i) = g(\sigma, J_i) \\ &\qquad \wedge \ \ g(\theta, H_k) < g(\sigma, J_k) \end{aligned}$$

Using the combining functions defined above, *globally-better* will produce the global comparators *weighted-sum-better*, *worst-case-better*, *least-squares-better*, and *unsatisfied-count-better*.

### 4.3 Implementation Issues

There is nothing in the definition of the global comparators that prevents the set of hierarchies $\Delta$ from being infinite. In practice, this can occur when rules are recursive, as the following program shows.

```
f(X):- g(X), prefer X < 0.
g(1).
g(X):- g(X - 1).
```

In general, an interpreter using one of the global comparators 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 best solutions. 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 `prefer` constraint in the first rule were altered to `prefer 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.

## 5 Nonmonotonic Aspects of Comparators

Classical logic is monotonic in the sense that adding new axioms to a theory can only enlarge the set of statements that can be inferred. It is never necessary to retract conclusions as new knowledge is gained. The global comparators defined in Subsection 4.2 give rise to nonmonotonic programs, because adding new rules may result in withdrawal of previous solutions. A comparator is *monotonic* if the set of solutions to a set of constraint hierarchies $\Delta$ is a subset of the set of solutions to the set of constraint hierarchies $\Delta \cup \Gamma$, where $\Gamma$ is any set of constraint hierarchies.

**Definition.** Let $\Delta$ and $\Gamma$ be sets of constraint hierarchies. Let $\theta$ 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.** 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 $\Delta = \{\{$`required X = a, default X = b`$\}\}$ and let $\Gamma = \{\{$`required X = b`$\}\}$, where `a` and `b` are two distinct elements in $\mathcal{D}$. (Both $\Delta$ and $\Gamma$ are singleton sets.) Let $\mathcal{C}$ be a comparator that respects the set of hierarchies. We make the following observations. $S_\Delta(\mathcal{C})$ 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.

To see how the nonmonotonicity property of comparators affects programs, consider the following trivial example.

```
f(X) :- g(X), prefer X > 0.
g(-1).
```

Given the goal `f(A)`, an interpreter for HCLP($\mathcal{R}, \mathcal{C}$), will return the answer `A = -1`. Suppose that now we add the fact `g(1)` to the program. The current implementation of HCLP will return both the answer `A = -1` and `A = 1` since it uses the $\mathcal{LPB}$ comparator, which cannot compare solutions arising from different hierarchies. However, using any global comparator $\mathcal{C}$ that respects the set of hierarchies, an interpreter for HCLP($\mathcal{R}, \mathcal{C}$) would return the single answer `A = 1` to the query `f(A)`. Thus, the set of facts that can be derived from the initial program is not a subset of the facts that can be derived when we add new rules. This behavior is similar to that of nonmonotonic logics (see for example [10, 11]), in which the addition of new axioms to an existing axiom set may result in the retraction of existing theorems.

One of the specific problems addressed by nonmonotonic logics is the frame problem. Consider a system for planning robot actions. We need to reason about the state of the world after the robot has performed some action. We would like to deduce that properties of the world will be the same after the action as they were before, unless there is reason to believe otherwise. However, it isn't possible to express this in standard first-order logic except by a laborious enumeration of specialized facts about each combination of possible robot actions and properties. Nonmonotonic logics do let us state such rules in a general way. So, for example, we might be able to conclude that the light is still on after the robot moves an object from location A to location B. If, however, we receive new information that the robot's arm hit and broke the lamp, then we will have to retract our previous conclusion.

Constraint hierarchies also provide a solution for the frame problem in the particular context of interactive graphics. The role of the frame axioms is filled by default constraints that parts of an object remain fixed for successive states of the object. As with nonmonotonic logics, this can give rise to situations in which the addition of a new rule requires the retraction of previously-computed solutions. For example, suppose that we are moving an object with the mouse, and that we strongly prefer the object remain inside its current window, even if the mouse attempts to move it outside the window boundary. We use a default constraint that the old location of the object be equal to its the new location. Due to the strong prefer constraint, attempts by the mouse to move the object outside the window would have no effect. If we then add a rule that allows the size of the window to be altered to include the mouse position, the default constraints would be overridden, the old solution would need to be retracted, and the object would move.

Although we can't achieve the full power of nonmonotonic logics, as our strengths are allowed only on constraints, we can handle many important cases. In the context of interactive graphics, for example, the use of default constraints obviates the need for an explicit statement of all the factors

that may cause an object to move, thus solving the frame problem for a specific type of application.

# 6 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, in an interactive graphics environment in which we might have a lengthy series of goals relating successive states of an object being moved on the screen, where each goal represents a user's command. Even if some of the commands have not yet been issued, we nevertheless will want to compute the current state of the object so that it can be displayed. In order to achieve this, we need to solve constraint hierarchies incrementally.

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*.

Note that CLP is orderly since adding required constraints to an existing set of required constraints will either narrow or leave the same the set of valuations that satisfy those constraints. This orderliness property is similar to the "stability of rejection" property discussed in [12].

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

**Proof.** Let $H = \{$`default X = a`$\}$, and let $J = \{$`prefer 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. Both nonmonotonicity and disorderliness may lead to revision of solutions, but for different reasons—the former through adding new rules to programs, the latter through adding constraints to existing hierarchies.

The disorderliness result shows that implementing a general, incremental solver for constraint hierarchies presents difficulties. However, a number of solutions are available for cases of practical interest. Consider for example an interactive graphics application. As noted above, a reasonable implementation in logic programming would be to create a series of goals relating successive states of the object. In principle, we might incrementally compute a solution for one of the states given the hierarchy as known so far, only later to have a new constraint applied to this state that invalidates the solution that was found. In practice, however, we would not use HCLP in this way for interactive graphics. All the constraints on a given state would be known before we needed to compute it; further, as-yet-unknown constraints would apply only to future states, not the current one.

Thus, one solution to this problem is to include a variant of the commit operator in the language. Such an operator would act as a cut, but would also solve the current hierarchy, and impose that solution as required constraints on the remaining variables. After this was done, adding more constraints could further refine the answer, but couldn't invalidate it. Further investigation is needed in this area—for example, would it be useful to separate the "commit" and the "solve" aspects of this operation, or to be able to solve for only some of the variables?

# 7 Nonprimitive Constraints

The current implementation of HCLP allows only *primitive* constraints—that is, constraints of the form $p(t_1, \ldots, t_n)$. However, the formal results in [2] still hold if we allow a larger class of constraints formed by combining primitive constraints using the logical connectives *and*, *or*, and *not*, and universal and existential quantifiers (see [8]). Whereas in CLP the effect of disjunction and conjunction can be achieved indirectly, the same is not true in HCLP as the examples below demonstrate. Extending HCLP to include nonprimitive constraints of this form would add greater expressiveness to programs. Moreover, there are many examples where the use of disjunction eliminates the necessity of inter-hierarchy comparisons, although not all cases can be handled in this fashion.

In CLP, conjunction and disjunction of constraints can be accomplished without explicitly using the disjunction and conjunction connectives. We can achieve the effect of conjunction simply by adding constraints to the body of rules. We can achieve the effect of disjunction, at least semantically, by adding new rules to a program. (Operationally, however, disjunction and choice points in the form of alterate rules can have quite different behavior. Although they will fail on the same queries and succeed on the same ground queries, for non-ground queries they can give a different number of answers and they can give a specific answer a different number of times.)

In HCLP, however, we cannot attain these results so simply. (Note that we are not referring to the conjunction and disjunction of labelled constraints. That is $s(c_1 \vee c_2)$ is allowed but not $s_1c_1 \vee s_2c_2$.) Consider the hierarchy {`required X` $\geq$ `-10, required X` $\leq$ `10, default (X` $\geq$ `5` $\wedge$ `X` $\leq$ `-5)`}. Using the $\mathcal{LPB}$ comparator, the answer to this hierarchy is the set of all valuations for `X` such that `-10` $\leq$ `X` $\leq$ `10.` Suppose we reformulate this hierarchy by removing the conjunction and creating two new default constraints and adding them to the hierarchy. The result will be the hierarchy {`required X` $\geq$ `-10, required X` $\leq$ `10, default X` $\geq$ `5, default X` $\leq$ `-5`}. Using the $\mathcal{LPB}$ comparator, the answer to this hierarchy is the set of all valuations for `X` such that `-10` $\leq$ `X` $\leq$ `-5` and `5` $\leq$ `X` $\leq$ `10.` We cannot achieve the effect of conjunction in HCLP merely by adding constraints to the body of rules as we can in CLP.

Similarly, disjunction cannot be achieved in HCLP simply by adding rules to represent alternate ways of satisfying constraints. Consider the following program:

```
f(X) :- g(X), prefer X > 0.
g(-1).
g(1).
```

As mentioned above, the answer to the goal `f(A)` using *locally-predicate-better* is the two valuations `A = -1` and `A = 1`. Now imagine extending the syntax of HCLP to include not only primitive constraints on the right-hand-side of rules, but disjunctions of primitive constraints, as well. Consider the following program:

```
f(X) :- required (X = -1 ∨ X = 1), prefer X > 0.
```

The answer to the goal `f(A)` is the valuation `A = 1`. This raises two interesting possibilities. The first is that the range of possible HCLP programs could be expanded by extending the class of allowable constraints to include the conjunction and disjunction connectives.

The second is to achieve the effect of inter-hierarchy comparison by allowing the disjunction of primitive constraints, as demonstrated by the previous example. What this accomplishes, in effect, is to merge two hierarchies into a single hierarchy. However, it is not always possible to combine two hierarchies in this fashion. Consider the following simple example.

```
f(X):- required X = 1, prefer X ≤ 0.
f(X):- required X = 2.
```

In satisfying the goal `f(A)`, two hierarchies are created, one of which consists of a single required and a single preferred constraint and one of which consists of a single required constraint. The *locally-predicate-better* comparator will return both the solutions `A = 1` and `A = 2`. A global comparator that respects the hierarchy will return the solution `A = 2`. If we attempted to rewrite this program using disjunction so that the *locally-predicate-better* comparator would return this same solution, we would have to determine a way to combine the required constraints. The only possibility is the following program.

```
f(X):- required (X = 1 ∨ X = 2), prefer X ≤ 0.
```

Unfortunately, the *locally-predicate-better* solution to the goal `f(A)` is the same as for the previous one, namely `A = 1` and `A = 2`. What is worse is that now certain global comparators (for example, *weighted-sum-better*) will return the *wrong* answer `A = 1`. We have related a required constraint from one hierarchy to a preferred constraint of another, thereby losing the intent of the original program.

The following program is a more realistic example in which disjunction is not powerful enough to remove the need for inter-hierarchy comparison. This program tries to find a mutually agreeable meeting time for Chris and Judy. Chris is free on Friday from 12 until 4, but she would prefer to meet after 2. She is also free on Saturday from 10 until 1. Judy is free Friday from 1 until 2 and on Saturday from 11 until 12. Even though Chris is available on both days at these times, we would prefer that the query `canmeet(Judy,Chris,T,D,1)` return an answer that does *not* include meeting Friday at 1 because this solution does not satisfy Chris' preference to meet after 2. However, the solution of meeting Saturday at 11 does satisfy all of the constraints in its hierarchy and should be the only solution. (The current version of HCLP will return both of these meeting times since it doesn't compare solutions arising from different rule choices.)

```
levels([required,prefer]).

canmeet(Person1,Person2,Day,Time,Length):-
     free(Person1,Day,StartTime1,EndTime1),
     free(Person2,Day,StartTime2,EndTime2),
     required StartTime1 ≤ Time, required StartTime2 ≤ Time,
     required Time + Length ≤ EndTime1,
     required Time + Length ≤ EndTime2.

free(judy,saturday,11,12).
free(judy,friday,13,14).
free(chris,saturday,10,13).
free(chris,friday,T,16):-
     required T ≥ 12, prefer T ≥ 14.
```

Satisfying the goal `canmeet(Judy,Chris,T,D,1)` involves two distinct constraint hierarchies—one consisting entirely of required constraints and the other consisting of (different) required constraints and the single prefer constraint `StartTime2` $\geq$ `14`. There is no way to reformulate this problem within a single constraint hierarchy using disjunction, because we cannot allow the prefer constraint to eliminate valuations that solve the required constraints in the first hierarchy.

So although extending the syntax of HCLP by introducing nonprimitive constraints enables us to rewrite some programs so that undesirable solutions are eliminated, we cannot always use this technique to dispose of the need for inter-hierarchy comparisons.

# 8 Related Work

Previous work in this area falls into two main areas: constraints and logic programming, and using constraints in applications. The Constraint Logic Programming scheme is described in [6]. A number of CLP languages have been implemented, including including Prolog III [3], CLP($\mathcal{R}$) [5, 7] and CHIP [4]. There has also been considerable work on using constraints in applications such as geometric layout, physical simulations, and user interface design, document formatting, algorithm animation, and design and analysis of mechanical devices and electrical circuits.

Constraint hierarchies were originally developed as an extension to ThingLab [1]. Regarding constraint hierarchies and logic programming, another approach to defining inter-hierarchy comparators has been taken by Michael Maher and Peter Stuckey [9]. (Maher collaborated with us on the original development of HCLP [2].) Rather than defining the set $S_0$ of valuations that solve the required constraints in some member of the set of hierarchies, they define the *pre-solutions* for a single hierarchy. Then they define a *pre-measure* $g$ which is a mapping from pre-solutions and sets of constraints to a scale $S$. Then a pre-solution $\alpha$ is *better* than a pre-solution $\beta$ if

$$(g_1(\alpha, H_1), \ldots, g_n(\alpha, H_n)) \geq (g_1(\beta, H_1), \ldots, g_n(\beta, H_n))$$

using the lexicographic ordering on $S_1 \times \cdots \times S_n$.

Various comparators can then be realized by the use of different scales and pre-measures. For example, if the scales are the set of all sets of constraints ordered by inclusion and the pre-measure is the set of constraints at a given level satisfied by a valuation, then we obtain the *locally-predicate-better* comparator. Taking the scales to be the set of real numbers and the pre-measure to be $0 - k$, where $k$ is the number of constraints at a given level unsatisfied by the valuation, we will generate the *unsatisfied-count-better* comparator.

It is also possible to obtain the *locally-error-better* comparator, by letting the scales be the set of all sets of possible $\langle$constraint, real number$\rangle$ pairs. A set $s$ is greater than or equal to another set $s'$ if the set of constraints in $s$ is a superset of those in $s'$ and the values of the corresponding numbers in $s$ are less than or equal to those in $s'$. The pre-measure is the mapping $g(\theta, H_i) = \{\langle c, v\rangle \mid c \in H_i \wedge v = e(c\theta)\}$.

Given these definitions, it it simple to extend this notion to allow inter-hierarchy comparisons by applying the above ordering to pre-solutions arising from different hierarchies. Note that in this scheme local comparators can be defined for inter-hierarchy comparison, unlike our definitions which limit them to intra-hierarchy comparison. Doing this can give some unintuitive results, however, as the following example demonstrates.

```
f(X):- g(X), prefer X ≥ 5.
g(X):- required X ≥ 0.
g(X):- prefer X = 6.
```

$\mathcal{LPB}$ with inter-hierarchy comparison (as defined above) gives the answer `A = 6` to the query `f(A)` even though the answer `A` $\geq$ `5` satisfies all of the constraints in the hierarchy `required A` $\geq$ `0, prefer A` $\geq$ `5`. This anomalous result supports the intuition that local comparators should only be defined for single hierarchies.

Our global comparators can be represented in the framework defined above by letting the scales be the set of real numbers and the pre-measures be $0 - g(\theta, H_i)$ where $g$ is one of the combining functions defined in Subsection 4.1.

# 9 Conclusion

Extending the definition of comparators to allow inter-hierarchy comparisons enables us to eliminate undesirable solutions in many types of problems. In addition, this extension leads to nonmonotonic

behavior for all comparators that respect the set of hierarchies. Another property of comparators that respect the set of hierarchies is disorderliness which, like nonmonotonicity, can lead to retraction of solutions. In particular, this quality makes it necessary to employ extra-logical means (e.g. an augmented commit operator) to perform incremental satisfaction of intermediate goals such as might arise in interactive graphics.

We have also shown that introducing nonprimitive constraints in Hierarchical Constraint Logic Programming can increase the expressiveness and power of the constraints. In certain common cases, it can remove the need for inter-hierarchy comparisons.

Future implementations of HCLP languages will benefit from the ability to execute inter-hierarchy comparisons. Despite introducing complexity, these revised comparators will allow greater applicability and usefulness of HCLP programs.

## Acknowledgements

Thanks for many useful discussions and comments on drafts of this paper to Michael Maher, Amy Martindale, and Dan Weld. We would also like to thank an anonymous referee for pointing out a flaw in our draft proofs. This research was sponsored in part by the National Science Foundation under Grant No. IRI–8803294, by a fellowship from Apple Computer for Molly Wilson, and by the Washington Technology Center.

## References

[1] Alan Borning, Robert Duisberg, Bjorn Freeman-Benson, Axel Kramer, and Michael Woolf. Constraint Hierarchies. In *Proceedings of the 1987 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 48–60. ACM, October 1987.

[2] 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.

[3] Alain Colmerauer. An Introduction to Prolog III. Draft, Groupe Intelligence Artificielle, Universite Aix-Marseille II, November 1987.

[4] 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.

[5] Nevin Heintze, Joxan Jaffar, Spiro Michaylov, P. Stuckey, and R. Yap. The CLP($\mathcal{R}$) Programmer's Manual. Technical report, Computer Science Dept, Monash University, 1987.

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

[7] 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.

[8] 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.

[9] 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.

[10] John McCarthy. Circumscription—A Form of Non-Monotonic Reasoning. *Artificial Intelligence*, 13(1,2):27–39, April 1980.

[11] Raymond Reiter. A Logic for Default Reasoning. *Artificial Intelligence*, 13(1,2):81–132, April 1980.

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