Composition in Hierarchical CLP

Michael Jampel and Sebastian Hunt
{jampel, seb}@cs.city.ac.uk
Department of Computer Science,
City University, Northampton Square,
London EC1V OHB, UK.

Abstract

We propose a variant of the Hierarchical Constraint Logic Programming (HCLP) scheme of Borning, Wilson, and others. We consider compositionality and incrementality in Constraint Logic Programming, introduce HCLP, and present Wilson's proof that it is non-compositional. We define a scheme for composing together solutions to individual hierarchies and show that hierarchy composition can be expressed very simply using filtering functions over bags (multisets); we present some properties of these functions, and define and explain an alternative to bag intersection which is also used in our scheme. We present an example involving three strength levels and show that we can achieve a close approximation to the solutions produced by standard HCLP.

1 Introduction

The HCLP (Hierarchical Constraint Logic Programming) scheme [Borning *et al,* 1989; Wilson and Borning, 1993] greatly extends the expressibility of the general CLP scheme [jaffar and Lassez, 1987]. There is also related work by Satoh [1990]. A semantics has been defined for HCLP [Wilson and Borning, 1989] and some instances of it have been implemented [Menezes *et* a/., 1993]. However, the semantics is not as natural as one might hope, and the implementations are inherently less efficient than those of CLP. We believe that these two issues may be related, and suggest that an equally expressive scheme which gives similar but not identical answers may go some way to overcoming both of them.

Therefore we propose a weakening of the semantics of HCLP, in which the results of composing solutions to individual hierarchies approximate the solutions that would be obtained from combining the programs and starting from scratch. Our semantics are adequate for some of the standard examples in the literature. They are definitely not sufficient to solve some of the other examples exactly, but our approximations are very reasonable.

2 Incrementality and compositionality

An important difference between logic programming languages and, say, theorem provers, is that the former are more efficient; expressibility and completeness are traded-off against efficiency. Standard HCLP is beautifully expressive, but the efficiency of its implementations may be poor, and its semantics lacks certain desirable properties. Our proposal in this paper involves tradirig-off completeness to gain efficiency and more tractable semantics. In constraint logic programming, efficiency is discussed with reference to 'incrementality', and semantics can be discussed with reference to compositionality, which is what we will do in the rest of this section.

Compositionality is a desirable property for a system to have because it shows that the semantics of the system can be modelled by a mathematical system with formal properties, and because it suggests that implementations will be efficient. This is because compositionality implies decomposition ality, i.e. we can solve a complex problem by splitting it into simpler parts, solving them, and then composing the results into a complete solution.

Whereas compositionality is a property of formal systems, incrementality is a (desirable) property of Constraint Logic Programming *implementations.* There is no precise definition, but what it means is that the work required to add an extra constraint to the solution of a large set of constraints and check its satisfiability is proportional to the complexity of the addition, and not related to the size of the initial set. If a system is not incremental, then adding one more constraint to the solution of, say, 20 constraints, involves as much work as solving the system of 21 constraints from scratch.

In fact, even in an 'incremental' system the amount of work required to deal with an additional constraint will probably depend on more than just the constraint itself: the number of variables in the original set may be relevant, as well as other factors.

We wish to suggest that compositionality is weaker than incrementality. a theory which has compositional

semantics may have a non-incremental implementation, but a truly incremental implementation of a non-compositional formalism is not possible. What is more important than the distinction between incrementality and compositionality is the distinction between having both these properties and having neither, which is related to the distinction between 'sufficiently efficient' and 'unusably inefficient'. Both logic programming and constraint logic programming are in principle sufficiently efficient, and they have compositional theories (see next section), and so compositionality is assumed to hold in general, almost without being mentioned. Therefore, the focus in previous CLP work has tended to be on incrementality alone, rather than on its relationship with compositionality.

2.1 Query composition in logic programming and CLP

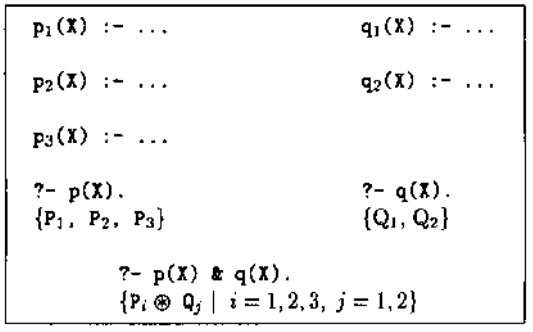

Figure 1: Composing queries in logic programming

Consider the two programs in Figure 1. They may be logic programs or constraint logic programs. The subscripts 1,2,3 are not part of the predicate names; they indicate that the query ?- p(X) will have up to three solutions {P_1, P_2, P3}, one from each of the three clauses of p. The composed query ?- p(X) & q(X) will have up to six solutions, arising from ®, the composition, in some sense, of the elements of the cartesian product of the two sets of solutions. We could treat each of the {P_1, ® Q_j] combinations independently, and make a distinction between this multiplicity and multiple solutions all arising from one branch, but we will just consider the collection of solutions as a whole, ignoring how they arose.

In standard logic programming, the composition of the solutions to two or more queries is the most general unifier (m.g.u.) of those solutions. Calculating the m.g.u. of two solutions takes time related to the size of the solutions, and not related to the size of the original programs. Similarly, in constraint logic programming, solving the conjunction of two output sets of constraints will depend on the size of the output, not on the size or complexity of the input constraints.

Hierarchical CLP is not compositional, and so incremental implementations have to make assumptions which may then need to be retracted [Menezes *et al.*, 1993]. But before we can demonstrate this non-compositionality, it is necessary to provide an overview of HCLP.

3 Hierarchical Constraint Logic Programming

A good introduction to HCLP can be found in Molly Wilson's PhD thesis [1993, chapter 4] or in the early reference [Borning *et al.,* 1989]; here is a brief overview. Just as Logic Programming can be extended to CLP, so CLP can be extended to a Hierarchical CLP scheme including both 'hard' and 'soft' constraints. The HCLP scheme is parameterised not only by the constraint domain *V* but also by the comparator C, which is used to compare and select from the different ways of satisfying the soft constraints.

An HCLP rule has 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 t is a list of terms, $p, q_1, \ldots, q_m$ are atoms and $l_1 c_1, \ldots, l_n c_n$ are labelled constraints. A program is a bag (multiset) of rules, and a query is a bag of atoms. The strengths of the different constraints are indicated by a non-negative integer label. Constraints labelled with a zero are *required* (hard), while constraints labelled *j* for some $j > 0$ are optional (soft), and are preferred over those labelled *k*, where $k > j$. (A program can include a list of symbolic names, such as *required, strongly-preferred,* etc., for the strength labels, which will be mapped to the natural numbers by the interpreter. If the strength label on a constraint is omitted, it is assumed to be *required.)*

Goals are executed as in CLP, except that initially non-required constraints are accumulated but otherwise ignored[1]. If a goal is successful but with a non-unique answer, the accumulated hierarchy of optional constraints is then solved, which refines the valuations in the solution. The method used to solve the non-required constraints will vary from domain to domain, and for different comparators within a given domain.

The constraint store sigma (a set) is partitioned into the set of required constraints *So* and the set of optional ones S_i. The solution set for the whole hierarchy is a subset of the solution set of *So,* such that no other solution could be 'better', i.e. for all levels up to *k,* S_k is completely satisfied, and for level S_{k-1} this solution is better, in terms of some comparator, than all others. Backtracking and incomparable hierarchies give rise to multiple possible solution sets, each a subset of the solution to S_o. (Solutions in standard HCLP are generally described using sets, not bags.)

Certain comparators can be used with any domain. For example, a 'predicate' comparator prefers one solution to another if it satisfies more constraints at some

[1]Menezes et al. use an alternative strategy [1993].

level (and an equal number of constraints at all previous levels). However if the domain has a metric (such as the real numbers) it is possible to ask how far from the preferred answer a solution is, in which case one might prefer fewer constraints to be exactly satisfied if the distance of the answer from a given point can be minimised[2]. Weights can be used within a particular level of the hierarchy in order to influence the solution, but a heavily weighted constraint in a given level is completely dominated by the lightest constraint in any higher level. Wilson calls this property 'respecting the hierarchy' [1993].

3.1 The disorderly property of HCLP

Wilson discusses a very simple but powerful example in her PhD thesis [1993], which shows the non-monotonic, hence non-compositional, nature of any variant of HCLP which respects the hierarchy.

Wilson defines the 'orderly' property as follows: Let P and Q be constraint hierarchies, and let $S_{\{P\}}(\mathcal{C})$ be the set of solutions to the hierarchy P when comparator $\mathcal{C}$ is used. Then $\mathcal{C}$ is *orderly* if $S_{\{P \cup Q\}}(\mathcal{C}) \subseteq S_{\{P\}}(\mathcal{C})$. A comparator which is not orderly is *disorderly*.

Prop. 1 *Any comparator which respects hierarchies over a non-trivial domain $\mathcal{D}$ is disorderly.*

Proof ([Wilson, 1993]): Let $P = \{\texttt{weak X = a}\}$ and $Q = \{\texttt{strong X = b}\}$ for two distinct elements of $\mathcal{D}$, `a` and `b`. (The existence of distinct elements is what makes $\mathcal{D}$ non-trivial.) $S_{\{P\}}(\mathcal{C})$ will be the valuation which maps `X` to `a`, and if $\mathcal{C}$ respects the hierarchy then $S_{\{P \cup Q\}}(\mathcal{C})$ will be the valuation which maps `X` to `b`. So $S_{\{P \cup Q\}}(\mathcal{C}) \not\subseteq S_{\{P\}}(\mathcal{C})$, so $\mathcal{C}$ is disorderly. □

We can easily adapt this proof to demonstrate non-compositionality. Consider $S_{\{Q\}}(\mathcal{C})$: it will consist of the valuation mapping `X` to `b`. The solution to the composition (conjunction) of the constraints will be $S_{\{P \cup Q\}}(\mathcal{C})$, i.e. a mapping from `X` to `b`. This is not related to, or derivable from, the union, intersection, or any other potentially incremental composition of $S_{\{P\}}(\mathcal{C})$ and $S_{\{Q\}}(\mathcal{C})$. In this paper we define an approximation to HCLP which *is* incremental; our work is based on bags, and so the next section provides an overview of bag theory.

4 Bags and guards

Bags, sometimes called multisets, are like sets except that duplicate elements are allowed i.e. $\{\!|a,a|\!\} \neq \{\!|a|\!\}$ (we use $\{\!|\ |\!\}$ to denote bags). In this section, we define various properties of bags, treating as basic the notion of number of occurrences of an element in a bag [Gries and Schneider, 1994]. A brief overview of some of the properties of bags can be found in Knuth [1969].

[2] For a complete understanding of comparators it is necessary to consider 'error sequences' [Wilson and Borning, 1993], but they are quite complicated and are only used briefly in this paper, so we omit them here for reasons of space.

The intuition is that an element which occurs a times in a bag A, and b times in B, occurs $a+b$ times in the additive-union $A \uplus B$, $\max(a,b)$ times in $A \cup B$, which is the equivalent of set-theoretic union, and $\min(a,b)$ times in the intersection $A \cap B$. We will not use $A \cup B$ in this paper, and so we feel free to refer to $A \uplus B$ as union, rather than the more clumsy 'addition' or 'additive-union'.

Let $e\#B$ denote the natural number n of occurrences of the element e in the bag B. **Examples:** $a\#\{\!|a,a|\!\} = 2$. $b\#\{\!|a,a|\!\} = 0$. A shorthand is to label elements with a superscript indicating the number of occurrences. **Example:** $\{\!|a,a|\!\} = \{\!|a^2|\!\}$. Of course $a\#\{\!|a^k|\!\} \equiv k$. Note that bag intersection is a 'lifted' version of set intersection. In other words, if A and B are both bags with no duplicate elements ('set-like' bags), then so is $A \cap B$.

4.1 Guards

The above is all standard bag theory. Later in the paper we need an alternative to intersection, with certain other properties, and so we now define a new infix binary operator $/\!\!/$ ('guarded by'). If A and B are both bags, the intuition is that $A /\!\!/ B$ contains only those elements of A which are also in B, with the same multiplicity as in A.

Definition:

$$\begin{aligned} e\#(A /\!\!/ B) &= e\#A, && \text{if } e\#B > 0 \\ &= 0, && \text{if } e\#B = 0 \end{aligned}$$

Examples:

$$\begin{aligned} \{\!|a,a,c|\!\} /\!\!/ \{\!|a,b|\!\} &= \{\!|a,a|\!\} \\ \{\!|a,a,c|\!\} /\!\!/ \{\!|b|\!\} &= \{\!|\ |\!\} \end{aligned}$$

We claim that our variant of HCLP, to be defined below using guards, has a strong mathematical basis. In order to defend this view, we now state some properties of $/\!\!/$. The proofs are straightforward, using the definitions of the symbols involved, and can be found in our technical report [Jampel and Hunt, 1995].

Prop. 2 (Idempotence) $A /\!\!/ A = A$.

Prop. 3 (Empty Bag) $A /\!\!/ \{\!|\ |\!\} = \{\!|\ |\!\} /\!\!/ A = \{\!|\ |\!\}$.

Prop. 4 (Absorption) *If $A \subseteq B$ then* $A /\!\!/ B = A$.

It is clear that $/\!\!/$ is not commutative, i.e. $A /\!\!/ B \neq B /\!\!/ A$, but it *is* associative:

Prop. 5 (Associativity) $A /\!\!/ (B /\!\!/ C) = (A /\!\!/ B) /\!\!/ C$.

Prop. 6 (Union Distributivity)
$(A \uplus B) /\!\!/ C = (A /\!\!/ C) \uplus (B /\!\!/ C)$.

Prop. 7 (Intersection Distributivity)
$A /\!\!/ (B \cap C) = (A /\!\!/ B) \cap (A /\!\!/ C)$.

Prop. 8 (Set Distributivity)
$\mathbf{set}(A /\!\!/ B) = \mathbf{set}(A) /\!\!/ \mathbf{set}(B)$.

4.2 Filter functions

To complete the mathematical tools we use later in this paper, we define a class of filters, functions from bags to bags, which remove some of the elements from the input bag. It is necessary to introduce them here because we

will use them in the next section to define rules for hierarchy composition; particular examples are discussed later, but in general filter functions will be denoted by f. Note that the guard operator // is concerned with the relationship between different strength levels in a hierarchy, whereas filter functions select solutions within a given level. (Similarly, in standard HCLP the comparators compare solutions within a given level.) More details can be found in Section 6, where we examine one particular filter at length.

5 Two-level hierarchy composition

We now come to the main interest of the paper. In this section we define an operation o which combines bags and sets in a certain way to form new bags. Our interpretation of these rules is that o approximates the solutions to a composition of hierarchies.

Let P and Q be two programs, and p and q be queries with respect to each of those programs. We will also use capital letters to refer to the entire hierarchy (i.e. program, query, and solutions). Let $S_0(p)$ be the set of valuations which are solutions to the required constraints in the hierarchy, and let $S_1(p)$ be the set of valuations satisfying both the required and the (one level of) optional constraints. Note that $S_1(p) \subseteq S_0(p)$. Similarly $S_1(q) \subseteq S_0(q)$. Define the solution set for the query p as a tuple containing the solution sets for each level of the hierarchy: $S(p) = (S_0(p), S_1(p))$.

Note that the solutions to a single (branch of a) query are assumed to form a set (or, rather, a set-like bag). The multiplicities will arise when the solutions to separate queries are composed using bag union. (Treating solutions to a single query as bags would give more weight to duplicated solutions from one hierarchy, rather than treating equally all the hierarchies to be composed.)

Later we will use 'p & q' to refer to a standard HCLP query, solved by starting from scratch, and compare it to p o q, which has solutions defined as follows:

Definition (ComposeOl):

$$S_0(p \circ q) =_{def} S_0(p) \cap S_0(q)$$
$$S_1(p \circ q) =_{def} \mathbf{f}[(S_1(p) \uplus S_1(q)) /\!/ S_0(p \circ q)]$$

The reason for guarding the two Si's with So(p 0 q) is to remove solutions for p which are inconsistent with q's required constraints and vice versa, although remember that at this stage we should not really be discussing hierarchies and solutions, because we are still merely manipulating bags. (In fact, as $S_1(p) \subseteq S_0(p)$, it is enough just to guard $S_1(p)$ with $S_0(q)$ and vice versa, as can be shown using absorption and distributivity properties, but this does not generalise neatly to compositions of more than two hierarchies.)

Note that we type-coerce sets to be set-like bags whenever we wish to do bag operations on them. Note also that in the definition of S_1(p0 q), we have not included an application of set. We consider it to be the very last action that should be performed on the bags of solutions, perhaps just treated as an aspect of pretty-printing.

Example: Let p and q have the following solution sets: $S_0(p) = \{a, b, c\}$ and $S_0(q) = \{b, c, d\}$. Then $S_0(p \circ q) = \{b, c\}$. Let $S_1(p) = \{b, c\}$ and $S_1(q) = \{b, d\}$. Then

$$\begin{aligned} S_1(p \circ q) &= \mathbf{f}[(\{b, c\} \uplus \{b, d\}) /\!/ \{b, c\}] \\ &= \mathbf{f}(\{b, b, c, d\} /\!/ \{b, c\}) \\ &= \mathbf{f}(\{b, b, c\}) \\ &= \{b, b\} \qquad \text{using } \mathbf{f}_{\max} \end{aligned}$$

Filtering must be left as late as possible within a given level. Specifically, if we are composing more than two hierarchies, the union and guard must be completed before any filtering is done; therefore the general definitions are as follows:

Definition (Multi-Compose):

$$S_0(\circ p_i) =_{def} \bigcap_i S_0(p_i)$$
$$S_{n+1}(\circ p_i) =_{def} \mathbf{f}\left[\left(\biguplus_i S_{n+1}(p_i)\right) /\!/ S_n(\circ p_i)\right]$$

Although o is commutative (i.e. poq — qop), it is not associative, because of the filter functions, i.e. po(qor) = (p o q) o r = p o q o r. Only p o q o r is 'correct' for our purposes, which implies the following: if we have calculated the composition poq but suspect that we may later need to compose with another hierarchy, we cannot discard the solutions to p and q, but must store them to be able to calculate poqor subsequently. Thus we trade space against time compared to standard HCLP.

If there are no constraints at the required level, the solution set is the entire domain (the cartesian product of the domains of all the variables) i.e. $S_0 = \mathcal{U}$. It can be seen that guarding any bag with this universal set using // will not have any effect: $\forall A \cdot A /\!/ \mathcal{U}$ —A. If there are no constraints at one of the optional levels, the solutions for that level will just be the same as for the next higher level i.e. $S_{n+1}(p) = S_n(p)$. Therefore the rules we have defined here will work for hierarchies with arbitrarily many or few constraints at any given level.

In HCLP, the key difference between required and non-required constraints is that the former can cause failure to occur. In other words, the required constraints may have an empty solution set, but no weaker constraints can cause a failure if stronger constraints have been satisfied. In our composition rules, in $A /\!/ B$, A represents the solutions for a weaker level than B, and yet the definition of $/\!/$ allows the possibility of $A /\!/ B$ being empty even if B is not (in the case that $A \cap B = \{\}$). Thus our rules appear to allow failure to arise from optional constraints. In fact, as we define the solution S to be the tuple (So, $S_1, \ldots, S_n$) and not just its final element S_n, it is not a problem if one of the elements of S is empty:

	P	Q	R
required	$X > 0$	$0 < X < 20$	$0 < X < 23$
strong	$X < 10$	$X > 5$	$X > 15$
weak	$X = 4$	X div 7^a	X div 8
S_0	$X > 0$	$0 < X < 20$	$0 < X < 23$
S_1	$0 < X < 10$	$5 < X < 20$	$15 < X < 23$
S_2	$X = 4$	$X = 7, X = 14$	$X = 16$

	$p\&q^b$	$\mathbf{f}_{max}(p \circ q)^c$	$p\&q\&r^d$	$\mathbf{f}_{max}(p \circ q \circ r)$
S_0	$0 < X < 20$	$0 < X < 20$	$0 < X < 20$	$0 < X < 20$
S_1	$5 < X < 10$	$5 < X < 10$	$5 < X < 10$, $15 < X < 23$	$(5 < X < 10)^2$, $(15 < X < 20)^2$
S_2	$X = 7$	$X = 7$	$X = 7$, $X = 8$, $X = 16$	$X = 7, X = 16$

[a] i.e. X is divisible by 7, shorthand for the disjunctive constraint $X = 0, 7, 14, \ldots$
[b] $p\&q$ calculated using standard HCLP
[c] $p \circ q$ calculated using $\mathbf{f}_{max}$ and Compose012
[d] Calculated using standard HCLP, taking into account error sequences [Wilson, 1993]

Figure 2: Compositions using standard HCLP and using $\mathbf{f}_{max}$

the solution that is offered to the user is no longer the final element of the tuple, but the final *non-empty* element. Thus this aspect of HCLP is not present in our logic, but is left until the interpretation.

6 The filter f_{max}

The rules defined in the previous section are parameterised by filter functions. We now define one particular filter function f_{max}, which is the most interesting when compared to HCLP's 'locally-predicate-better' comparator [Wilson and Borning, 1993]. f_{max} removes those elements of a bag which do not occur a maximal number of times. In other words, if some elements occur once in a given bag, and some elements occur twice, f_{max} defines the bag containing only those elements occurring twice.

Definition:

$$e \# \mathbf{f}_{max}(A) = e\#A, \text{ if } e\#A = \max\{i\#A \cdot i \in A\}$$
$$= 0, \quad \text{otherwise}$$

Examples:

$$\mathbf{f}_{max}\{a, a, b\} = \{a, a\}$$
$$\mathbf{f}_{max}\{a, b\} = \{a, b\}$$

6.1 $\mathbf{f}_{max}$ is incremental

Consider some hierarchy (program, query, and solutions) formed from twenty constraints. Consider the amount of work needed to augment it with a hierarchy resulting from one extra constraint: the two S_0 levels must be intersected, the S_1 levels must be unioned, filtered, then guarded by the combined S_0. Calculating an intersection requires work bounded by the size of the smaller of the two sets or bags, as does set-union. Creating a bag-union could take constant time, or at worst time related to the size of the smaller of the two bags. Filtering and guarding will, admittedly, take time bounded by the size of the union of the two bags. But all these operations (intersection, union, filtering, etc.) are computationally very cheap compared to constraint solving. They may be considered unit time operations when measured on the scale of constraint solving. Therefore, when compared to the cost of composing the two programs and queries and starting from scratch, it is reasonable to say that our scheme is incremental.

7 Example and comparison with HCLP

The Multi-Compose definitions from Section 5 can be unfolded as follows:

Definition (Compose012):

$$S_0(p \circ q) = S_0(p) \cap S_0(q)$$
$$S_1(p \circ q) = \mathbf{f}[(S_1(p) \uplus S_1(q)) /\!/ S_0(p \circ q)]$$
$$S_2(p \circ q) = \mathbf{f}[(S_2(p) \uplus S_2(q)) /\!/ S_1(p \circ q))]$$

We will now use these rules in an extended example, which includes a comparison between our solutions and those of standard HCLP.

Consider the example in Figure 2; part of it (P) is taken from Wilson's thesis [1993, Section 2.2]. The first point to note is that the sub-example of composing P and Q provides motivation for the choice of $\mathbf{f}_{max}$ over simpler filter functions: if we used $\mathbf{f}_{id}$ then $S_0(p \circ q) = S_1(p \circ q) = \{(0 < X < 20)\}$ and $S_2(p \circ q) = \{(X = 4), (X = 7), (X = 14)\}$, i.e. there is not enough discrimination to capture, or even approximate, the behaviour of standard HCLP. (Note that we have been us-

ing {0 < X < 10} as short-hand for the uncountably infinite set-like bag *{(X = 0), (X = 1), (X = 2),..., (X = 0.1), (X — 0.01),...}*, but it is obvious within this notation that the bag union of, say, {0 < X < 10} and {5 < X < 15} has 0 < X < 5 and 10 < X < 15 with multiplicity 1, and 5 < X < 10 with multiplicity 2.)

Secondly, if we initially apply Compose012 to *p* and *q* and then apply it again *to poq* and r, S_2((po *q)* o r) will be the same as S_2 (p o g o r) (with different muliplicities at the S_1 level). But if instead we compose *p* with *qor,* then S_2(po(qor)) = {(X = 4),(X = 16)5, which differs from S_2 ((p o *q)* o r) and is a worse approximation to the result given by standard HCLP. This non-associativity gives rise to multiple possible composition orders and hence solutions, which is why we define the only correct answer to be the one based on composing all three sub-solutions simultaneously.

Thirdly, and most importantly, if we apply Compose012 all at once, we get an answer which differs from the standard HCLP solution, because it omits the valuation which maps X to 8. Any system which calculates exactly the same answers as HCLP will be vulnerable to Wilson's proof of non-compositionality (discussed earlier in Section 3.1), so the question is whether Compose012 gives a reasonable approximation. We believe that it does; in this example, the only valuation omitted is one which was not present in the original sub-solution containing the constraint which gave rise to it. In *R,* the X = 8 possibility for X div 8 was dominated by the strong constraint X > 15. In standard HCLP, this domination is weakened by other strong constraints from completely separate hierarchies. In other words, the only solution not calculated by Compose012 is one which we probably do not want anyway!

8 Conclusions and Further Work

We have seen that by storing the intermediate solutions to a hierarchy in a tuple (S_0,S1 ,.. ., S_n), we can find an approximate solution to its composition with other hierarchies. We use simple filtering functions and bag operations, with clear mathematical characterisations, which avoids the need to invoke the constraint solver and start from scratch calculating a solution to the combination of the constraints. The non-associativity of filter functions means that the method is not truly compositional, but it does avoid the need to invoke the constraint solver. In general, constraint satisfaction is of exponential complexity, compared to which filtering and set operations are very cheap indeed. Therefore we feel justified in calling this scheme 'incremental'.

Thus we have developed a variant of Hierarchical CLP in which the solution to a composed problem is defined in terms of the solutions to its sub-problems. The operations involved are simple to understand and efficient and closely approximate the answers we would obtain from standard HCLP, while avoiding its computational expense and complex semantics.

In the future we wish to extend this scheme to take account of error sequences, and hence bring different comparators within the scope of our filtering scheme. We would like to investigate constraint deletion and dynamic constraint satisfaction, which are difficult in most formalisms but should be made easier by the modular nature of our tuple representation of solutions.

More generally, we are investigating inconsistencies arising from the composition of CLP programs i.e. programs without hierarchical strength labels. We wish to explore the parallels between composition of these unlabeled systems and composition of the labelled systems discussed in this paper.

Acknowledgements

Thanks to Rob Scott, David Gilbert, Steven Eker and Bernie Cohen for helpful discussions on CLP and HCLP.

References

[Borning *et a*/., 1989] A. Borning, M. Maher, A. Martindale, and M. Wilson. Constraint hierarchies and logic programming. *ICLP'89.*

[Gries and Schneider, 1994] D. Gries and F. Schneider. *A Logical Approach to Discrete Math.* Springer.

[Jaffar and Lassez, 1987] J. Jaffar and J.-L. Lassez. Constraint Logic Programming. *POPL'87.*

[Jampel and Hunt, 1995] M. B. Jampel and L. S. Hunt. A compositional theory of constraint hierarchies. Technical Report TCU/CS/1995/5, Department of Computer Science, City University, London.

[Knuth, 1969] D. Knuth. *The Art of Computer Programming.* Addison-Wesley. Volume 2: Seminumerical Algorithms.

[Menezes *et a*/., 1993] F. Menezes, P. Barahona, and P. Codognet. An incremental hierarchical constraint solver. *PPCP'93.*

[Satoh, 1990] K. Satoh. Formalizing soft constraints by interpretation ordering. *ECAI'90.*

[Wilson and Borning, 1989] M. Wilson and A. Borning. Extending Hierarchical Constraint Logic Programming: Nonmonotonicity and inter-hierarchy comparison. *NACLP'89.*

[Wilson and Borning, 1993] M. Wilson and A. Borning. Hierarchical Constraint Logic Programming. *Journal of Logic Programming,* 16(3) :277—318.

[Wilson, 1993] M. Wilson. Hierarchical Constraint Logic Programming. Technical Report 93-05-01, University of Washington, Seattle. (PhD Dissertation).