[16] H. Schmidt, W. Kiessling, U. Guntzer and R. Bayer, "Compiling Exploratory and Goal-Directed Deduction into Sloppy Delta-Iterations", *Proceedings of Symposium on Logic Programming*, San Francisco, pp. 234–243 (1987).

[17] J. Thom and J. Zobel, "NU-Prolog Reference Manual", Melbourne University, Computer Science Technical Report TR–86–10 (1986).

[18] P. Vasey, "Qualified Answers and their Application to Transformation", *Proceedings of the Third International Conference on Logic Programming*, (E. Shapiro Ed.) London, LNCS 225, pp. 425–432 (1986).

[19] M. Wilson and A. Borning, "Extending Heirarchical Constraint Logic Programming: Nonmonotonicity and Inter–Hierarchy Comparison", *Proceedings of North American Conference on Logic Programming*, Cleveland, (1989).

References

[1] A. Borning, R. Duisberg, B. Freeman-Benson, A. Kramer, and M. Woolf, "Constraint Hierarchies", *Proceedings of ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pp. 48–60 (1987).

[2] A. Borning, M. Maher, A. Martindale, and M. Wilson, "Constraint Hierarchies and Logic Programming", *Proceedings of the Sixth International Conference on Logic Programming*, Lisbon, pp. 149–164 (1989). Also Computer Science Technical Report 88–11–10, University of Washington, (1988).

[3] Ph. Chatalic, "IMPRO: An Environment for Incremental Execution in Prolog", submitted for publication, 1989.

[4] M. Cheng, M. van Emden and J. Lee, "Tables as a User Interface for Logic Programs", *Proceedings of the International Conference on Fifth Generation Computer Systems* (ICOT Eds.), Vol 3, pp. 784–791 (1988).

[5] M.H. van Emden, M. Ohki and A. Takeuchi, "Spreadsheets with Incremental Queries as a User Interface for Logic Programming", *New Generation Computing* **4**, pp. 287–304 (1986).

[6] P. van Hentenryck, "Incremental Constraint Satisfaction in Logic Programming", draft paper, 1989.

[7] T. Huynh and C. Lassez, "A CLP($\Re$) Options Trading Analysis System", *Proceedings of the Fifth International Conference and Symposium on Logic Programming* (R. Kowalski, K. Bowen, Eds.), Seattle, pp. 59–69 (1988).

[8] T. Imielinski, "Intelligent Query Answering in Rule Based Systems", *Journal of Logic Programming* **4**, pp. 229–257 (1987).

[9] J. Jaffar, J-L. Lassez, "Constraint Logic Programming", Technical Report 44, Dept. of Computer Science, Monash University (1984), abstract in *Proceedings of ACM Conference on Principles of Programming Languages*, Munich, pp. 111–119 (1987).

[10] J. Jaffar, S. Michaylov, P.J. Stuckey and R. Yap, "The CLP($\Re$) Language and System", IBM Research Report, to appear.

[11] M. Maher, "A Logic Semantics for a Class of Committed Choice Programs", *Proceedings of the Fourth International Conference on Logic Programming* (J–L. Lassez Ed.), Melbourne, pp. 858–876 (1987).

[12] L. Naish, "Negation and Control in PROLOG", LNCS 238, Springer-Verlag (1987).

[13] M. Ohki, A. Takeuchi and K. Furakawa, "A Framework for Interactive Problem Solving based on Interactive Query Revision", *Proceedings of Logic Programming 86*, LNCS 264, pp. 137–146 (1986).

[14] D. Poole, R. Aleliunas, R. Goebel, "Theorist: A Logical Reasoning System for Defaults and Diagnosis", Technical Report CS–86–06, Dept. of Computer Science, University of Waterloo (1986).

[15] V. Saraswat, "Concurrent Constraint Logic Programming", Ph. D. Dissertation, Carnegie-Mellon University (1989).

all the conditions are met, there may be no solutions. In order to increase the chances of a successful search, the user may use some of the conditions to express preferences. For instance

$strong : Delta = 0,\ weak : Theta > 0,$

$strong : abs(Midpoint - Stockprice) \leq 2.5,\ weak : Margin \leq 1000,$

$required : Return > Margin * Interest + Debit + Commission$

where *weak*, *strong* and *required* refer to strengths 2, 1 and 0 respectively. We assume the use of the *unsatisfied_count_better* comparator. Then one answer which has a solution that satisfies constraints 1,2,4,5 is better than an answer with a solution that satisfies constraints 3,4,5. If we are looking for the best answer and we first find the answer satisfying constraints 1,2,4,5 then we can prune the branch that leads to the second answer without computing the entire branch.

The introduction of hierarchical constraints, reduces the problem of underspecifying or overspecifying an options strategy and hence obtaining too many answers or no answers (an almost wasted computation), and provides a natural ranking of the answers obtained.

5 Conclusion

We have presented two classes of commands: those which manipulate an answer to obtain a more specialized answer, and those which generate and search a collection of answers. When answers contain floundered goals it is necessary to distinguish between answers to the original query and answers to a sub-query which is a specialized answer. We do this by entering a sub-shell every time an answer is first specialized by an answer manipulation command, leaving only when we invoke the "Original Answer" command.

We have only a partial implementation of the proposed system. Several of the commands require direct access to details of the constraint solver, which is not available to a meta-interpreter. A complete compiled implementation of the proposed query interface requires several modifications to existing implementation techniques: a more flexible constraint solver, path modifications to the WAM, built-in measures including value calculation and storage. However each of these is clearly implementable.

6 Acknowledgements

We would like to thank Alan Borning, Catherine Lassez, Amy Martindale and Molly Wilson for helpful discussions and comments. The OTAS example is due to Catherine Lassez.

where we take the lexicographic ordering of $S_1 \times \cdots \times S_n$. A *solution* is a pre-solution which is maximal under this ordering. (There are many other meaningful orderings on $S_1 \times \cdots \times S_n$, but they don't correspond to the nature of constraint hierarchies as defined in [1].)

By taking the scales to be the set of all sets of constraints ordered by inclusion and $g_i(\theta, C_i) = \{c \in C_i | \mathcal{D} \models c\theta\}$ we obtain the notion of *locally_predicate_better* [1]. If we choose the scales to be $\Re$ and make an appropriate choice of pre-measures we can derive the *globally_better* comparators [2].

For example consider the following pre-measures (on the scale $\Re$):
$h_S(\theta, C) = n$ where n is the number of constraints $c \in C$ such that $\mathcal{D} \models c\theta$, $h_U(\theta, C) = -n$ where n is the number of constraints $c \in C$ such that $\mathcal{D} \not\models c\theta$. The notions of better that arise from taking $g_i = h_S$ (or $g_i = h_U$) for $1 \leq i \leq n$ are known respectively as *satisfied_count_better* [1] and *unsatisfied_count_better* [2]. These two notions are semantically identical for any fixed constraint hierarchy, since they provide identical orderings of pre-solutions.

The above comparators compare different pre-solutions of the same constraint hierarchy, as in [1] and [2]. In our framework it is easy to extend the definition to compare pre-solutions of different hierarchies: α on H is *better* than β on H' if

$$(g_1(\alpha, C_1), \ldots, g_n(\alpha, C_n)) \geq (g_1(\beta, C'_1), \ldots, g_n(\beta, C'_n))$$

However, in order to do pruning of the search tree we need to be able to compare constraint hierarchies and, more generally, goals containing constraint hierarchies. We define the *hierarchical measure* M derived from $g_1, \ldots, g_n$, as the mapping

$$M(G) = lub(g_1(\theta, C_1), \ldots, g_n(\theta, C_n))$$

from goals to $S_1 \times \cdots \times S_n$, where G is a goal containing the constraint hierarchy H and the least upper bound[1] is taken over all pre-solutions θ of H. It is interesting to note that although the comparators based on h_S and h_U are semantically identical on single hierarchies, the hierarchical measure based on h_U is a pruning measure while the measure based on h_S is not.

As an example, consider an OTAS [7] query to generate option strategies for positions which are delta-neutral, theta-positive, within 2.5 points of the current stockprice, require only \$1000 margin and have a maximum return that is profitable. The query contains the following constraints.

$Delta = 0,\ Theta > 0,\ abs(Midpoint - Stockprice) \leq 2.5,$

$Margin \leq 1000,\ Return > Margin * Interest + Debit + Commission$

If the only required condition is that the maximum return is profitable then the current market may provide too many solutions. Yet if we require that

[1] Note that now we must assume that least upper bounds exist for each $S_1, \cdots, S_n$.

each goal. Both the rightmost branches do not need to be further explored because they cannot yield answers better than that already obtained. It is not yet clear in what circumstances the cost of the computation saved by pruning outweighs the overhead of computing the measures.

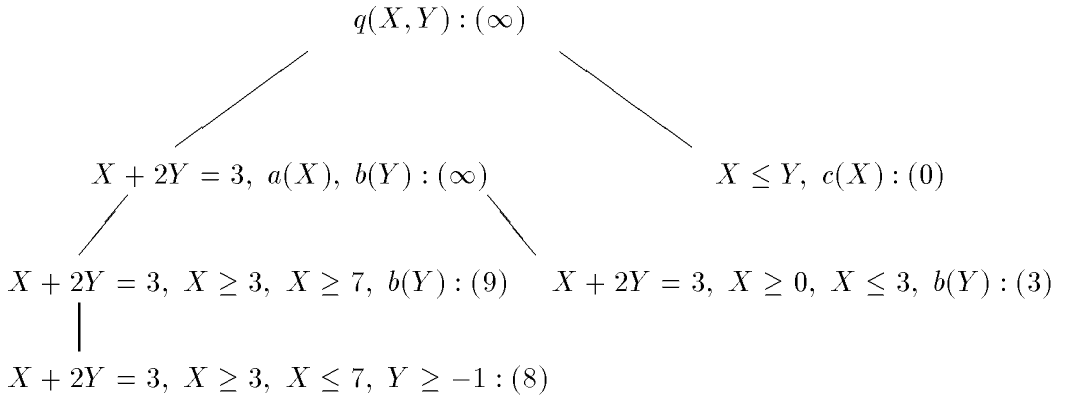

Figure 2: Traversal of search tree for $q(X,Y)$

4.2 Constraint Hierarchies

Constraint hierarchies [1] extend the notion of constraints being strict requirements, to allow constraints to express preferences as well. They were first combined with constraint logic programming (to form the HCLP class of languages) in the context of interactive graphics [2]. There the emphasis was on choosing a preferred solution according to some definition of better, for a fixed constraint hierarchy. We give a different presentation of constraint hierarchies than [1] [2], which enables us to compare solutions from different hierarchies. (This problem is also addressed in [19].) This allows us to develop measures for HCLP programs.

A *constraint hierarchy* H is a multiset of constraints each labeled with a strength. These strengths form a total order, and we will map them onto the integers $0\ldots n$. Let C_i in denote the multiset of constraints in H with strength i, with their labels removed. If $i<j$ then the constraints of C_i are preferred over those of C_j. C_0 is the set of constraints which *must* be satisfied, that is constraints in the usual CLP sense.

A *pre-solution* θ of hierarchy H is a valuation for the variables in H such that $\forall c\in C_0,\ \mathcal{D}\models c\theta$. A *pre-measure* g is a mapping from pre-solutions and sets of constraints to a scale S, e.g. $g(\theta,C)=s\in S$.

Let α and β be pre-solutions of constraint hierarchy H. Let $g_1,\ldots,g_n$ be pre-measures with scales $S_1,\ldots,S_n$. Then α is *better* than β with respect to these parameters if

$$(g_1(\alpha,C_1),\ldots,g_n(\alpha,C_n))\geq(g_1(\beta,C_1),\ldots,g_n(\beta,C_n))$$

An example of a non-representation independent measure is "the number of constraints in G". A measure m is *non-trivial* if $\exists G_1\ G_2,\ m(G_1) < m(G_2)$.

Every non-trivial measure is useful for defining preferred answers, but some measures will require the entire search tree to be traversed in the determination of the best answers. We characterize those measures that can ignore parts of the search tree safely as follows. A *pruning* measure is a measure m such that: if G' can be obtained from G in a single computation step using program P then $m(G') \leq m(G)$. Suppose we wish to find a best answer (according to m) to some query Q. Suppose we already have one answer of value M, and the computation reaches a goal G where $m(G) < M$. If m is a pruning measure then the portion of the search tree rooted at G can safely be ignored, since any answer in this portion will be worst than the current best answer.

A measure m is *monotonic* if

$$\mathcal{D} \models G \rightarrow G' \quad \Rightarrow \quad m(G) \leq m(G').$$

Clearly, any monotonic measure is representation independent. A measure is *constraint-based* if it ignores all the atoms in a goal. It is not difficult to show that any constraint-based monotonic measure is a pruning measure for all programs.

In languages such as CLP($\Re$), where the constraint solver is based upon the Simplex Algorithm, a natural form of preference is the maximal (or minimal) value of an objective function subject to a set of constraints. Given an objective function f on variables $\tilde{y}$ we can define a measure m_f with scale $\Re \cup \{\infty\}$ as follows

$$m_f(C \wedge G) = \max_{\tilde{x} \in sol^n(C)|\tilde{y}} f(\tilde{x})$$

where $sol^n(C)|\tilde{y}$ is the solutions of constraint C restricted to the variables $\tilde{y}$. For the measure to be non-trivial the objective function should be restricted to the variables appearing in the query. In the example in the introduction we might choose $m_f(G) = \min\ Cost$ to find the cheapest flights, and $m_f(G) = \min\ ArrivalTime - DepartureTime$ to find the quickest flights.

The measures m_f are examples of constraint-based monotonic measures, and hence are pruning for all programs. For example, consider the following CLP($\Re$) program

```
q(X,Y) :- X + 2Y = 3, a(X), b(Y).
q(X,Y) :- X <= Y, c(X,Y).
a(X) :- X >= 3, X <= 7.
a(X) :- X >= 0, X <= 3.
b(Y) :- Y >= -1.
c(X,Y) :- 3X - Y = 7, Y >= 0.
```

then the query `q(X, Y)` with pruning measure $m(C \wedge G) = max\ X - Y$ subject to C gives the search tree of Figure 2 with scale values appended to

of the entire computation, since a failed derivation may become successful after the query is decremented. Recent work attacks this problem using dependency-directed backtracking [3] and re-execution [6].

Recently Cheng et al [4] have proposed a user-interface for logic programs based on a view of atoms in a query as operators which transform relations to relations (this view is called TuplePipes). Although the discussion there is very value-oriented, the underlying ideas can be extended to constraint-oriented systems. The TuplePipes concept roughly corresponds, in our framework, to a combination of the simple form of incremental querying presented in [5] and a request for all answers from a query (command $*$, Table 2). It does not have the flexibility (it seems) to cope with intermediate queries which produce an infinite search tree. However it has some novel features which are not addressed in this paper.

4 Finding Best Answers

In situations where many answers are possible, one important technique for limiting the answers presented to the user is to restrict to the N best answers. To find best answers we must have some method to compare two answers. We need to be able to associate a value with each answer, and compare the relative worth of these values. We formalize these concepts in the next subsection.

Hierarchical constraints [1] provide a simple way for the user to specify the relative importance of different constraints in an answer. In the second subsection we extend the current theory to show that constraint hierarchies provide a basis for comparing different answers to a query. In particular, they can express a notion of best answer.

4.1 Pruning

Given a query $Q(\tilde{y})$ the computation rule of the underlying CLP system and the program determines a search tree containing all answers to the query. Throughout this subsection we assume this tree remains fixed. In this section a goal is an existentially quantified conjunction of atoms and constraints where only the variables $\tilde{y}$ are free.

A *scale* is a partially ordered set of values. A *measure* is a mapping from goals to a scale. Together a measure and scale define a preference relation between goals. Ideally, a measure is only dependent on the meaning of a goal since, for example, we would not want the relative worth of answers to be altered by the way the answers are represented in the CLP system, or the way they were accumulated. Equivalent answers should have equal measures. Formalizing this property, a measure m is *representation independent* if

$$\mathcal{D} \models G \leftrightarrow G' \quad \Rightarrow \quad m(G) = m(G')$$

during the execution of the original query P, Q and incrementation has a corresponding derivation for the query P, I, Q and vice versa. From this follows the correctness of our approach — the technique does not introduce spurious answers, nor does it omit any answers.

Proposition 2 *The path-based execution of incremental queries is sound and complete.*

On the example above, using the approach of [5] the execution visits every node in the search tree of the original query to the right of the first answer and to the left of the second answer. Placing the query increment before the initial query (as is possible in both our approach and that of [13]) results in the computation avoiding executing any of the `safety_check` goals of the original search tree (except for those in the successful derivation), since the new current information fails most resistor choices. The difference between the approach of [13] and our approach is that while we begin our search from `R1 = 100, R2 = 40` they must re-search all resistor choices, effectively redoing the computation from scratch.

Our current implementation, like those of [5, 13] uses meta-interpretation. A compiler takes a program and produces a new program with each predicate augmented to keep track of path information and to run in a restricted mode with respect to an input path. A query meta-interpreter calls individual atoms in the query, and handles the incremental modifications of the query. The meta-interpreter splits the path information around a query increment and the augmented program searches only new nodes in the search tree of the post-increment. The meta-interpeter also uses each stored answer as an environment to run the query increment, and stores the resulting answers.

In a WAM implementation the path information is already encoded within the choice points. A full implementation of the query modifications we propose could extract this path information while backtracking to the end of the pre-increment, execute the query increment, and then run the post-increment according to the path information. If query increments are only constraints we need not explicitly collect the path information, since it is only used once. We can just use the actual path (as present in the choice points) to determine the post-increment execution. Hence this approach is especially applicable to CLP languages.

$<$**NG**	Add Before	Insert goal **G** before ($\mathbf{N}^{th}$) atom in query
$>$**NG**	Add After	Append goal **G** after ($\mathbf{N}^{th}$) last atom in query

Table 3: Query Modification Commands

An obvious extension to the query handling facilities described above is the deletion of parts of a query. However this may require re-examination

P) versions of the derivation tree of Q. Similarly I_1', I_1'', I_2, I_3 are pruned versions of the derivation tree of I. And $Q_2', \ldots, Q_3''$ are pruned (by the query increment I) versions of Q_2 and Q_3.

The basic technique of our approach is to associate with each derivation a *path* which is the list of clauses chosen at each step in this derivation. In Figure 1 the query is incremented after a successful derivation, and the path for this derivation is shown. Execution of the incremented query proceeds roughly as follows: We use standard backtracking to retrieve the execution state at the end of the pre-increment and then execute the increment. For every successful derivation of the increment which extends the path in P we use the path information to restrict execution of the post-increment. Specifically, the post-increment execution is restricted so that as long as we follow the path of the original derivation (of the post-increment) we can only choose branches at or to the right of that chosen in the original derivation. The path restrictions force the new derivation to traverse only the unexplored part of the search tree. When backtracking reaches the pre-increment the path restrictions are not applicable to subsequent executions of the post-increment. If we have saved answers (c.f. section 3.1) each of these forms an environment for executing the query increment, and is replaced by the answers to these additional executions.

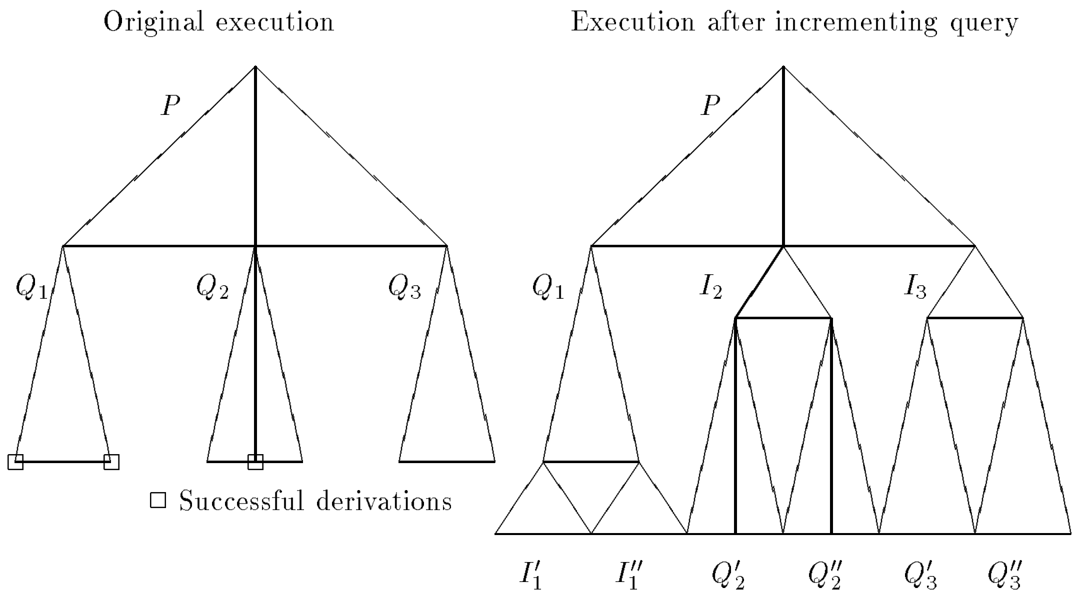

Figure 1: Incremental querying strategy

The execution of the query — before and after incrementation — is shown in Figure 1. The shaded part of each diagram shows the portion of the search tree traversed by the execution. Every derivation generated

```
available_res(40).
available_res(100).
   ...
available_res(2700).
available_res(6000).
available_res(10000).
volt_div(V, I, V2, R1, R2) :-
   V1 = I * R1, V2 = I * R2, V = V1 + V2,
   available_res(R1), available_res(R2).
safety_check(V, I) :- PowerLimit = 100, V * I < PowerLimit.
safety_check(V, I) :- CurrentLimit = 5, V < CurrentLimit.
```

The initial query

```
?- V=110, 30<V2, V2<40, volt_div(V,I,V2,R1,R2), safety_check(V2,I).
```

returns the answer constraint `V = 110, R1 = 100, R2 = 40, V2 = 31.4, I = 0.785`. The user notices that the current is too large for the voltage source in use, and insists on a smaller current by adding the *query increment*

```
??- I <= 0.10.
```

The next answer returned is `V = 110, R1 = 6000, R2 = 2700, V2 = 34.1, I = 0.0126` . The addition of the query increment made the incremented answer *unsatisfiable*. Hence the execution was forced to backtrack and determine a new answer for the initial query. Although there are similarities with the constraint addition discussed in the previous section, in that situation adding the new constraints would cause the answer *unsatisfiable*, but there would be no backtracking.

The first system for incremental querying in logic programming was presented by van Emden et al [5]. In that system, queries are incremented by appending user-specified atoms to the end of the original query. The execution for the query increment continues in the context of the current answer. This strategy is easy to implement and natural with respect to the PROLOG selection rule. The query increment is, in effect, a filter on the answers of the original query.

A second system [13] extended [5] by allowing the user to place the query increment at any position in the original query. It appears to be implemented by backtracking to the position where the query increment is to be added, inserting the increment, and then executing the increment and the remainder of the query in the usual PROLOG fashion. This extension represents a significant improvement, since a well-placed query increment can considerably prune the size of the search tree.

Our approach to implementing incremental querying improves on that of [13]. It produces a smaller tree to be searched by exploiting a more detailed knowledge of the execution before the query was incremented.

In Figure 1, the original query is P, Q and the query increment I is added between P and Q, making P the *pre-increment* part and Q the *post-increment* part of the query. Q_1, Q_2, Q_3 are pruned (by the pre-increment

example, after introducing the constraint `X = helen` the user may request all loans for which Helen is eligible with the command "*".

: **N**	Next Answer	Save current answer and compute next (**N**) answer(s) (where necessary)
; **N**	Next Answer	Ignore current answer and compute next (**N**) answer(s) (where necessary)
b **N**	Go Back	Go back to ($\mathbf{N}^{th}$) last saved answer
*	All Answers	Compute and save all remaining answers
.	Answer Number	Print out current answer number
f **N**	Goto Answer	Goto $\mathbf{N}^{th}$ saved answer
$	Final Answer	Goto final saved answer
/ **C**	Search Forward	Find next saved answer consistent with **C**
? **C**	Search Backward	Find previous saved answer consistent with **C**
h	Halt	Interrupt current execution
n **NM**	Best N Answers	Complete execution retaining the best **N** answers according to measure **M** (see 4.1)
1 **M**	Best Answer so far	Find the best saved answer by measure **M**

Table 2: All-solutions commands

3.2 Incremental Querying

In many situations the Prolog query mechanism forces a user to ask many closely related queries. As observed in [5], users often do not have a clear idea of what question they wish to ask, but their ideas become clearer as they interact with the system. The first few queries made are "range-finding"; they give the user an idea of the variety and magnitude of the database. In the ensuing queries the user converges on the intended query. Sometimes the user simply wishes to explore the contents of the database, without an underlying intended query [16]. And sometimes the user's querying strategy is to start with a too general query and refine it to the intended query.

One problem which all these interaction styles share is that sometimes queries result in too many answers to be comprehensible. We shall present two approaches to the solution of this problem. The first, which we present in this subsection, allows the user to modify a query while the query is still being computed. Such modification makes *incremental* changes to the original query. The second approach allows the user to restrict the answers to "best" answers. This approach is pursued in Section 4.

For an example of incremental querying, consider the following CLP($\Re$) program for designing voltage dividers from a list of available resistors, subject to some safety conditions, adapted from [10].

```
mortgage(P, I, 1, MP):- P * (1 + I) - MP = 0.
mortgage(P, I, T, MP):- T >= 2, mortgage(P*(1+I)-MP, I, T-1, MP).
.....
?- salary(X,S) when ground(X).
salary(helen, 110000).
salary(bruce, 45000).
?- number_of_dependents(X, N) when ground(X).
number_of_dependents(helen, 3).
.....
```

The query `loan(X, P, MP)` leads to the conditional answer

```
Inc = S - T, A = 2300*N + 5000, salary(X, S),
number_of_dependents(X, N), tax(S, A, T),
loan_type(IntY, TermY, Inc), mortgage(P, IntY/12, TermY*12, MP).
```

If the user adds the constraint `X = helen` then the delayed atoms `salary(X, S)` and `number_of_dependents(X,N)` are woken and execution can proceed determining the final answer `P = 105.006 * MP`. Then presuming that Helen wants a mortgage of $200000 we can add the constraint `P = 200000` obtaining the answer `MP = 1904.646`. If she now wants to consider a mortgage of $180000 then we can undo the last command and then add the constraint `P = 180000`. When Bruce wants to consider a mortgage the user simply returns to the original answer, adds the constraint `X = bruce` and proceeds from there.

3 Modifying Queries

A basic assumption of this section is that the user is simultaneously interested in many answers to a single query. As a consequence we first need to consider a mode of execution in which previous answers may be saved. Basic commands are given in the first subsection. Sometimes there will be too many answers to a query. One solution to this problem is to incrementally add further constraints and atoms to the (already executing) query. We explore techniques for making efficient incremental changes to a query.

3.1 All-Solutions Mode

We first consider a mode of execution in which previous answers may be saved. The mode of execution we describe is only fully sensible when the result of the computation is fully expressed by the answer constraints. That is, the computation does not result in side effects such as input/output or modification of the program.

We require an interface that allows the user to peruse the available answers with ease. Typically a user may want to move non-sequentially through the answers, and compute new answers as they are required. Table 2 provides a summary of some desirable commands for such an interface. In the loan

implication $C \rightarrow \exists\tilde{y}C'$ by comparing the solved form for C with the solved form for $C \wedge C'$. Also, any solver for a language which includes negatives of each constraint can be used to test implication, since $C \rightarrow D$ iff $C \wedge \neg D$ is not satisfiable. Incorporating interfaces to *external solvers* and *graphic displays* does not directly affect the constraint handling facilities of the system.

2.2 Floundered Goals as Answers

By floundered goals we mean goals containing atoms, none of which may be chosen by the computation rule. Such goals may occur in many different contexts: as a result of user-provided control annotations (as in MU-Prolog [12]) or the computation rule of the language [11, 15], as a result of breakpoints created in the process of debugging, as a result of a user interrupt, from a policy of intelligent query answering [8], or intelligent interaction with the user [18], in a system for conducting hypothetical reasoning [14].

In the previous discussion we dealt with answers consisting only of constraints. If delayed atoms appear in the answer we can still apply the manipulations discussed in the previous subsection, although we need to extend them for the more general form of answers. Projection must apply to the entire answer, so no variable which occurs in a delayed atom may be eliminated. Implication becomes a form of subsumption. Deletion requires us to return to the answer state before the deleted constraint was added. Addition (and exemplify) may wake some of the delayed atoms. Furthermore, a user may wake a delayed atom using the *wake* command. In these cases the computation can be allowed to continue. With the definition of answer extended to include floundered goals and the extensions of the commands discussed above, the previous Proposition continues to hold.

Consider the following program for an (imaginary) CLP($\Re$) system supporting "when" declarations [17], which relates salary, eligibility for loans and details of loan repayment.

```
loan(X, P, MP) :- income(X, Inc), loan_type(IntY, TermY, Inc),
    IntM = IntY/12, TermM = TermY*12, mortgage(P,IntM,TermM,MP).
income(X,Inc) :- Inc = S-T, salary(X,S), allowance(X,A), tax(S,A,T).
allowance(X, A) :- number_of_dependents(X, N), A = N * 2300 + 5000.
?- tax(S, A, T) when ground(S) and ground(A).
tax(S,A,T) :- S > A,  taxscale(S-A, T).
tax(S,A,0) :- S <= A.
?- taxscale(I, T) when ground(I).
taxscale(I, T) :- I <= 16000, T = 0.25 * I.
taxscale(I, T) :- 16000 < I, I <= 60000, T = 0.30*(I-16000)+4000.
taxscale(I, T) :- 60000 < I, T = 0.40 * (I - 60000) + 17200.
?- loan_type(Int, Term, Inc) when ground(Inc).
loan_type(11.0/100, 30, I) :- I >= 60000.
loan_type(10.8/100, 20, I) :- I >= 80000.
loan_type(10.5/100, 15, I) :- I >= 120000.
?- mortgage(P, I, T, MP) when ground(I) and ground(T).
```

p **V**	Project	Project answer onto variables **V**
a **C**	Add	Add constraint **C** to answer and simplify
a **G**	Add	Add goal **G** to answer constraint
d **C**	Delete	Delete added constraint **C** from answer
d **G**	Delete	Delete added goal **G** from answer
e	Exemplify	Construct an example ground answer
i **C**	Implication	Test whether answer implies **C**
c **C**	Consequence	Test whether answer is implied by **C**
o	Original Answer	Display original answer
u	Undo	Undo the last command
x **S**	External Solver	Invoke external solver **S** on answer constraint
g **RV**	Graph Answer	Invoke an external routine **R** to graph the answer restricted to 2 (3) variables **V**
w **A**	Wake	Wake atoms **A**
r	Run	Execute awake atoms
z **A**	Sleep	Delay atoms **A**

Table 1: Answer Manipulation Commands

For a CLP($\mathcal{D}$) program P, if the goal G results in answer C then

$$P, \mathcal{D} \models C \rightarrow G$$

The next proposition shows that a similar soundness property is preserved under the manipulations discussed in this section.

Proposition 1 *Let P be a CLP($\mathcal{D}$) program and G a goal with answer C. Let $y_1, \ldots, y_k$ be all the variables eliminated by projection during a series of manipulations, and let the result of these manipulations applied to C be C'. Then*

$$P, \mathcal{D} \models C' \rightarrow \exists y_1 \ldots y_k G$$

The implementation of these commands need not impose much extra burden on the constraint handling facilities of the basic CLP system. A user *addition* of a new constraint is no different from constraint additions generated in the execution of a query. Most tests for solvability implicitly construct a solution; implementing *exemplify* would only require deriving this solution explicitly. The ability to perform *projections* should already be in the constraint output routine, in order to eliminate auxiliary variables introduced in the computation. The ability to test for *generalized implication* may already be in the CLP system if the system can delay execution of atoms as in [11, 15]. Even if this capability is not so readily available, constraint solvers which are based on a sufficiently stable solved form can test for the

variables. For example in CLP($\Re$) the system might return the answer

$$X + Y + Z \geq 1, \quad -2Y + Z \geq 3, \quad Y - 2Z \geq 2$$

which gives the complete solution set of some query *q(X, Y, Z)*, but it may be more helpful to the user to see explicitly what constraints are placed on the values X can take. In the above example we can conclude $X \geq 6$. This constraint is said to be the *projection* of the above constraint onto X. Y and Z have been *projected out*.

The projection of the solution space of a constraint onto variables $\tilde{x}$ can be represented by the existential quantification of the remaining variables in the constraint. To simplify this representation we need to eliminate the quantified variables. Although full elimination of the quantified variables is possible in some domains — for example real arithmetic, linear arithmetic and Boolean algebra — in most domains it is not possible. Nevertheless, in many domains and for many constraints a partial elimination is possible (e.g. in the domains of finite and infinite trees). We will also refer to this partial elimination as projection.

Given an answer, the user may wish to improve his/her understanding of the solution set by examining the consequences of imposing additional constraints. For example, what happens in the above answer when $X = Y$ (the solution set becomes empty — the constraints are unsatisfiable). Similarly the user may wish to know if the above answer ensures that $2X + 3Y \geq 3$ (which, in fact, it does). To allow these kinds of questions we allow the user to add constraints to the answer and introduce a test for a generalized form of implication (called validation in [11]). This test determines whether

$$\mathcal{D} \models C \rightarrow \exists \tilde{y}\ C'$$

for an answer constraint C and user-supplied constraint C', where $\tilde{y}$ is the list of variables in C' not in the variable environment of C. (The variable environment of C is a set consisting of the variables in the query, the variables in the answer, and any new variables introduced by user-added constraints, except those variables that have been projected out).

One constraint solving methodology of CLP languages is to *delay* the solving of difficult constraints. For these languages the solvability of some answers may be unknown. These so called *maybe* answers can be tested for solvability by invoking a more powerful solver. This approach allows complicated answers to be evaluated by solvers that are too slow to be used in the CLP engine.

Table 1 provides a summary of the answer manipulation commands. (In this table – and others we present – we assume, for simplicity, line-oriented input. The ideas easily generalize to more sophisticated input interfaces). Projection, addition and deletion are the only manipulations that modify the answer. Exemplification can be viewed as a form of repeated addition, while the remaining manipulations are all either tests or particular display techniques.

where B_i is a selected atom, and clause

$$B \leftarrow C',\ D_1,\ \ldots,\ D_m$$

from P, where $\{B = B_i\} \wedge C \wedge C'$ is satisfiable in domain $\mathcal{D}$, and results in the goal

$$\leftarrow \{B = B_i\} \wedge C \wedge C',\ B_1,\ \ldots,\ B_{i-1},\ D_1,\ \ldots,\ D_m,\ B_{i+1},\ \ldots,\ B_n$$

where $\{B = B_i\}$ is a set of constraints equating the arguments of atoms B and B_i. A *computation rule* determines at each step which atom (if any) is selected. A *search tree* for goal G (for some computation rule) is the tree rooted at G where the children of each node G' are the goals obtained from a derivation step applied to G'. A *derivation* of G is a branch in the search tree of G.

Operationally, we can think of a CLP derivation as calling procedures in the usual way, accumulating constraints as they are met, and either verifying that the constraints are solvable or else backtracking if they are not. The derivation terminates with the accumulated set of constraints, called the *answer constraint*, which is presented in a form restricted (as far as possible) to the variables in the original goal by an output routine. For more detail see [9].

There are currently several CLP languages: CAL, CHIP, CIL, CLP($\Re$), PROLOG-III, Trilogy, among others, and pure PROLOG can be thought of as a simple CLP language. Although many of the examples we use throughout the paper are based on CLP($\Re$) [10], the query interface we propose is independent of the domain of the CLP language, and is applicable to all the above languages.

2 Answers

Answers in conventional logic programming are simple and essentially self-explanatory. In CLP answers can be more complex, and in general it is more difficult to fully understand the solution set represented by an answer. The first part of this section presents an approach to overcome these problems. The second part briefly considers a more general situation in which some delayed atoms may be part of an answer. Our approach could be briefly described as providing the ability to manipulate answers. We present and motivate several manipulations, and show that all these manipulations preserve a soundness property. In contrast to sections 3 and 4, the facilities described in this section are useful even when only one answer to a query is of interest.

2.1 Manipulating Answer Constraints

The user may want to restrict his/her attention to a particular subset of the variables in the answers, or to determine relationships between particular

and discuss possible notions of "best" where such pruning is possible. We consider the use of constraint hierarchies [1] to express the notion of "best"

Query answers in conventional logic programming systems (for example X = [a, b | Y], Z = b) are simple and need no further clarification. However in CLP systems answers may be very complex, for example, $Z \geq 4 \times \log(3 + Y - X)$, $X \leq 3 \times e^{Z-Y}$. It is not clear what values X can take, nor is it clear what the relationship is between X and Y. In a conventional querying environment the effect of assuming that $X = Y$ must be calculated by hand. We propose a number of answer manipulations and demonstrate that they satisfy a soundness property.

The Options Trading Analysis System (OTAS) [7] was the initial motivation for the extensions we describe in this paper. OTAS is a decision support tool written in CLP($\Re$) which generates and analyses options-based investment strategies. It typically runs by taking some constraints on the parameters of the investment strategies to be explored and searching to find all the strategies that fit these requirements. The OTAS program already includes some of the features we describe. We are aiming at a principled introduction of these features into CLP systems, to remove the onus of providing such capabilities from application developers.

In the remainder of this section we give a brief review of constraint logic programming. In the second section we present our approach to manipulating answer constraints. The final two sections are devoted to solving the problem of too many answers. Section 3 presents incremental querying, and section 4 the finding of best answers.

1.1 A Brief Review of Constraint Logic Programming

A constraint logic programming [9] system, CLP($\mathcal{D}$), exists in the context of a particular structure $\mathcal{D}$ which determines the meaning of the function and relation symbols. Constraints in the structure are relations upon terms of the structure. An *atomic constraint* takes the form $r(t_1, ..., t_n)$ where r is an n-ary relation symbol defined by $\mathcal{D}$. For example, the following are atomic constraints in the domain of real numbers: $X + Y > 7$, $Z^2 - 3Z + 8 = 0$, $X \times Y \times Z = 3.14$. We assume that the class of constraints contains all atomic constraints and is closed under conjunction and existential quantification. A *solution* of a constraint C is an assignment v of values from $\mathcal{D}$ to variables such that $\mathcal{D} \models v(C)$.

Constraint logic programs differ from logic programs by allowing constraints in bodies of rules and goals. A *constraint logic program* is thus a finite set of rules of the form

$$A \leftarrow C,\ B_1,\ \ldots,\ B_n$$

where A and B$_i$, $1 \leq i \leq n$, are atoms, and C is a constraint.

A *derivation step* using CLP($\mathcal{D}$) program P takes a goal

$$\leftarrow C,\ B_1,\ \ldots,\ B_i,\ \ldots,\ B_n$$

Expanding Query Power in Constraint Logic Programming Languages

Michael J. Maher and Peter J. Stuckey
IBM T.J. Watson Research Center,
P.O. Box 704, Yorktown Heights, NY 10598, U.S.A.

Abstract

We present some extensions to the query interface for Constraint Logic Programming (CLP) languages. One is an improvement of existing incremental querying schemes that always makes use of previous execution, and we verify its correctness. We define a framework for computing best answers to a query, and discuss goal measures which do not require the entire search tree for a query to be traversed. In particular we examine the use of constraint hierarchies in this framework. We also propose a collection of answer manipulation commands for interaction with the executing query.

1 Introduction

The query interface to logic programming systems has remained basically unchanged since the early Prologs. It only allows the user to ask a query, view an answer, or ask for the next answer. As a result the interface is clumsy for queries where the user is interested in several answers. For example, suppose that we are querying an airline booking system. A naive query will simply ask for combinations of flights from (say) New York to Melbourne. However many such combinations exist. After examining a few of the answers, the user will realize that the query was under-specified. It would be desirable to permit the user to constrain this query further by, for example, insisting that there are fewer than 4 connections.

Simply repeating the existing query with the added constraint wastes the previous computational effort. Consequently some form of incremental querying [5, 13] is indicated. Unfortunately the approach of [5] is not well-suited to this problem since it might generate a combination of 6 flights and then reject it, rather than rejecting the combination as soon as the first 4 flights are chosen. The approach of [13] allows the incremented query to have the desired behavior, but in this case gains no benefit from the previous computation, and so repeats the existing query with added constraint. We develop an algorithm for incremental querying which always exploits the previous computation.

A second response a user may take to an under-specified query is to ask for only the "best" answers, for example, the cheapest flight combinations. Again it is preferable that the more expensive flight combinations be eliminated from consideration as early as possible. We formalize the situation