IQ: Towards Incremental Query Processing in Non-Deterministic Concurrent Constraint Logic Programming

J.H.M. Lee and H.F. Leung
Department of Computer Science
The Chinese University of Hong Kong
Hong Kong
Email: {jlee,lhf}@cs.cuhk.hk
Fax: +852 603 5024

Abstract

Van Emden's *incremental queries* addresses the inadequacy of current Prolog-style querying mechanism in most logic programming systems for interactive problem-solving. There has been little published work on efficient implementation of incremental queries and all work is on sequential systems. In the context of constraint logic programming, incremental queries involves adding new constraints or deleting old ones from a query after a solution is found. This paper presents an implementation scheme IQ of incremental queries in *Constraint Pandora*, which defines a class of non-deterministic concurrent constraint logic programming languages. Constraint Pandora is essentially a concurrent version of the CLP scheme. We use Van Hentenryck and Le Provost's scheme (VHLP-scheme hereafter), which is based on oracles and re-execution, as a starting point. Re-execution is costly in concurrent languages, in which process creation and inter-process communications are common operations. The IQ scheme retains the oracle concept from the VHLP-scheme and is based on backtracking. We modify the conventional backtracking approach in such a way that constraints are used actively to prune the search space. The main idea of IQ is that the basic trail unwinding operation used in backtracking is more efficient than re-execution in reaching an execution context along a recorded execution path. Analysis shows that the IQ scheme is at least as efficient as the VHLP-scheme in sequential systems and is much more efficient in concurrent systems.

1 Motivation and Related Work

Most logic programming systems adopt the Prolog-style querying mechanism, which is unsuitable for interactive problem-solving, such as database and spreadsheet applications. Users have to work in the "planned" or "batch" mode [4]: *make sure you have available in advance all constraints and process them all in one go. If you forgot something, then you will discover this when you have reconstructed the solution from the output, and you can have another try.* Of course, a user can still use Prolog as a pseudo-interactive language by entering a new and improvised, but often redundant, query at each problem-solving step. This is hardly a satisfactory alternative both in terms of human ergonomics and computing usage. First, the users have to manage and re-enter (not necessarily by typing) the complete, and possibly large, set of constraints every time. This is both tedious and error-prone. Second, Prolog starts an independent computation for each query, discarding the possibility of using results of the previous ones.

In interactive problem-solving, users usually do not know how to go about solving a problem. They may have to rely on intermediate results to suggest the next step in the problem-solving process. In some cases, the users do not even have a clear definition of the problem at hand to start with. This happens often in such applications as scheduling and budgeting. Van Emden proposes *incremental queries*[1] [19], a new querying framework for Prolog. According to this framework, users can enter a query by *increments*, which consist of one or more goals. After an increment is entered, Prolog displays an answer, which is the result of, not the increment but, the query accumulated so far. The intermediate answers displayed can assist the user in composing and tuning the next increment of the query. In this framework, variables with the same identifiers in different increments denote the same variable. Thus increments are continuations of the same query. Van Emden *et al* [20] show how incremental queries couples well with a spreadsheet user-interface for logic programs and refine the method so that users can specify either addition or deletion of goals in increments. Ohki *et al* [16] and Cheng *et al* [4] also worked on the topic but their work is more on the functionality of incremental queries than on efficient implementation.

A naive approach to implement of incremental queries is to execute the accumulated query from scratch after each increment is entered. This approach induces much redundant computation. An efficient implementation should make use of the computation context of previous increments as much as possible, but is complicated by the presence of backtracking in the execution mechanism of logic programming languages. For addition of constraints or goals, if a given increment cannot be solved in the current context, the system should propagate backtracking to previous increments. In the case of deletion, the system should be able to recover the previous discarded execution paths in previous increments which are made possible again by the deletion. Former attempts for Prolog use intelligent backtracking [21, 20], the `assert` and `retract` meta-calls [2], and goal dependency analysis [3].

Van Hentenryck and Le Provost [22] proposed an implementation scheme, called the *VHLP-scheme* hereafter, for incremental queries in the context of constraint logic programming. Their scheme allows incremental addition and deletion of, not goals but, constraints only. The lack of backtrack points in the execution of a constraint allows for an efficient scheme based on oracles (or execution paths) and re-execution. Maher and Stucky [15] also use execution paths to guide the re-execution of goals but their approach relies on users to decide where to backtrack.

We are interested in incorporating incremental queries into *Constraint Pandora* [11], which defines a family of non-deterministic concurrent constraint logic programming languages. Constraint Pandora is essentially a concurrent version of the CLP scheme [9] based on the execution model of Pandora [1]. Unlike the committed-choice concurrent logic languages, such as Parlog [8], GHC [18], and Concurrent Prolog [17], Constraint Pandora supports both don't-know and don't-care non-determinisms, allowing the generation of multiple answers to a query.

We use the VHLP-scheme as a starting point. Re-execution is not suitable for our setting since process creation and inter-process communication incur overheads in the execution of Constraint Pandora programs. Computation context is saved along an execution path and can be reused by unwinding the trail, a basic operation provided by backtracking. Van Hentenryck and Le

[1] Also called *incremental execution* and *incremental search* by other authors [2, 3, 22].

Provost [22] criticize backtracking as the source of thrashing behavior [13] due to bad backtrack points and passive use of constraints. We show how conventional backtracking can be modified in such a way that constraints are used actively to prune search space. The main idea of our method is to re-use execution context (or environment) and to avoid re-execution as much as possible. This is achieved by the trail unwinding operation, which allows us to reach a previous execution context more efficiently than re-execution. For addition of constraints, our scheme arrives at the same "best" backtrack point as the VHLP-scheme. Deletion of constraints is difficult to handle. While the VHLP-scheme starts re-execution from scratch, our scheme starts from a safe-guaranteed intermediate execution context, located using oracles. What is described is a scheme for efficient implementation of incremental queries; hence the name IQ.

The paper is organized as follows. Section 2 gives the syntax and operational semantics of Constraint Pandora. The Constraint Pandora search tree and its skeleton version, which are important analysis tools in latter parts of the paper, are also introduced. An abstract definition of incremental queries and the related terminology are presented in section 3. Before we embark on our method, a brief review of the VHLP-scheme is given in section 4. In section 5, we show how the IQ scheme handles addition and deletion of constraint in an incremental querying session before comparing our results with the VHLP-scheme. Implementation issues are also discussed. Section 6 contains the concluding remarks of the research and sheds light on future work.

2 Constraint Pandora

In this section, we describe Constraint Pandora. The CLP scheme [9] is a special case of Constraint Pandora and the results shown in this paper apply equally well to sequential CLP languages. Section 2.1 gives the syntax of Constraint Pandora, followed by a presentation of the operational semantics in section 2.2. Section 2.3 defines Constraint Pandora search tree, which is an important tool in the analysis of our method.

2.1 The Language

Constraint Pandora [11] defines a family of non-deterministic concurrent CLP languages, obtained by augmenting Pandora [1] with the capacity of constraint-solving. As in the CLP scheme, a different constraint domain yields a different instance of Constraint Pandora language. A Constraint Pandora program syntactically resembles a Pandora program, except that constraints are allowed in clause guards as well as clause bodies. Using the terminology of [14], the constraints in clause guards are *ask*-constraints and those in clause bodies are *tell*-constraints.

A clause in Constraint Pandora is of the form

$$H \leftarrow G, C_{ask} : B, C_{tell}$$

where H is an atom called the *clause head*, G a (possibly empty) conjunction of system predicates called the *guard calls*, C_{ask} a (possibly empty) conjunction of ask-constraints, B a (possibly empty) conjunction of atoms called the *body goals*, and C_{tell} a (possibly empty) conjunction of tell-constraints. While the unification goals (including the matching goals (`<=/2`)) are seen as system predicates in Pandora, in Constraint Pandora they are seen as constraints in the Herbrand

Universe. G and C_{ask} are collectively called the *guard* and B and C_{tell} are collectively called the *body*. Intuitively, a clause denotes the logical relation that "G and C_{ask} and B and C_{tell} implies H."

2.2 Operational Semantics

Constraint Pandora belongs to the family of concurrent logic programming languages [5] and is related to the ALPS framework [14]. As in Pandora, "goal" is synonymous with "process" in the constraint version. The execution of Constraint Pandora programs can be abstracted to the transitions from an *execution state* to another. An execution state of Constraint Pandora consists of a *process pool* P, which is a multiset, and a *constraint store* S, which is a set. Intuitively, the process pool contains all processes spawned during execution, and the constraint store contains the tell-constraints generated during execution.

A Constraint Pandora query is of the form

$$\leftarrow B, C_{tell}$$

where B is a conjunction of body goals, and C_{tell} is a conjunction of tell-constraints. Initially, the process pool and constraint store contain all the goals and all the constraints in the query, respectively. It is assumed that a constraint solver exists, which constantly monitors the constraint store. This constraint solver has two functions. First, it reports *inconsistency* when there is any, and then causes a *failure* in the execution. Second, it solves the constraint variables whenever possible. In the process pool, processes are classified into two kinds. A *don't-know* process executes only if it has at most one candidate clause[2]. If it does not have a candidate clause, a failure occurs. If it has more than one candidate clause, then it suspends. A *don't-care* process executes whenever it finds a candidate clause. In other words, the don't-know processes use a more "prudent" Andorra-like commit law [23], while the don't-care processes use a more "eager" Parlog-like commit law [8].

There are two kinds of transitions of execution states. When a process executes, it "commits" to a candidate clause (for don't-know processes, *the* candidate clause), generates the tell-constraints in the clause body, spawns the processes in the clause body, and then terminates. The newly generated constraints and processes are added to the constraint store and process pool respectively. The old process is removed from the process pool.

Another kind of transition occurs when every process in the system suspends. The execution state is then said to *deadlock*. If the system deadlocks and there is no don't-know process in the system, a failure occurs. Otherwise, a deadlock handler is invoked to select one of the don't-know processes, based on which a *choice point* is set up. The candidate clauses of the selected don't-know processes are then tried in turn. We assume that the deadlock handler employs a fixed computation rule and clause ordering [12] in the selection of body goals and the trying of rules at each step of the computation. When failure occurs, the execution state will be restored to that of the most

[2] A clause $H \leftarrow G, C_{ask} : B, C_{tell}$ is a *candidate clause* of a process A if and only if $(G \wedge C_{ask} \wedge A = H)$ is entailed (validated) by the constraint store S. This condition can be formally expressed as [14]:

$$\forall x_g[S \models \exists x_l[G \wedge C_{ask} \wedge A = H]]$$

where x_g are the *global variables* in S and x_l the *local variables* in C_{ask} and H.

recent choice point, and an untried candidate clause will be tried. If all candidate clauses at the choice point have been tried, then a new failure occurs.

The execution of a Constraint Pandora query $Q = \leftarrow B, C_{tell}$ *succeeds* if and only if the constraint store is consistent and the process pool is empty. Let P be the Constraint Pandora program for the query and $\mathcal{D}$ the underlying constraint domain. Suppose the constraints in the constraint store at the end of a successful execution is S'. The soundness of Constraint Pandora [11] guarantees that

$$P, \mathcal{D} \models (\forall)(S' \Rightarrow C_{tell} \wedge B).$$

We call S' a *solution* to the query Q.

2.3 Constraint Pandora Search Tree

A *Constraint Pandora search tree* (CPS-tree) is a tree in which each node is an execution state. The root of the tree is the initial execution state corresponding to the query. An execution state is a child of another if and only if the former is obtained from the latter by a transition. Obviously, every node in the tree has at most one child, except that a deadlock state might have more than one child.

There are three possible kinds of leaves. A leaf with an empty process pool and a consistent constraint store corresponds to a successful execution. The solution of this successful execution is the constraint store in the leaf. A leaf with an inconsistent constraint store, or a leaf with only suspended don't-care processes in its process pool corresponds to a failure.

A *skeleton Constraint Pandora search tree* (skeleton CPS-tree) is a "reduced" CPS-tree. Intuitively, a skeleton CPS-tree is obtained from a CPS-tree by concentrating on the root, the deadlock nodes, and the leaves, and ignoring the rest. A skeleton CPS-tree T' of a CPS-tree T can be defined as follows:

- The root of T' is the same as that of T.
- If N is the root or a deadlock node in T', and $(N, N_1, \ldots, N_k, M)$ is a path in T, and $N_1, \ldots, N_k$ $(k \geq 0)$ are not deadlock nodes, and M is a deadlock node or a leaf in T, then M is a child of N in T'.
- There are not any other nodes in T'.

It is important to realize that Constraint Pandora is a generalization of the CLP scheme. A Constraint Pandora clause become an CLP clause when the guard is empty. The queries for both Constraint Pandora and CLP have the same form. Operationally speaking, if we allow spawning only one process at each Constraint Pandora state transition, the execution strategy of Constraint Pandora becomes the generalized SLD-resolution as defined in [22]. Furthermore, the skeleton CPS-tree is the same as the CPS-tree and the generalized SLD-tree. In this regard, our method is also suitable to sequential CLP languages. We make our method as general as possible by neglecting intentionally the structure of any underlying constraint domain.

3 Incremental Queries and Preliminaries

In this section, we follow the style of [22] in defining incremental queries in the context of Constraint Pandora and the relevant terminology. Since skeleton CPS-tree has the same format as a generalized SLD-tree, properties of the generalized SLD-tree [22] hold also for the former. We state the properties without proof.

We observe that succession of queries in an interactive problem-solving session are usually modification of each other. We define an *incremental query sequence* to be a sequence of queries of the form

$$(\leftarrow B, C_1), (\leftarrow B, C_2), \ldots, (\leftarrow B, C_i), (\leftarrow B, C_{i+1}), \ldots$$

where, without loss of generality, C_{i+1} is obtained from C_i by either adding or deleting a constraint. In practice, queries in an incremental query sequence are entered in order. Given a program P and an incremental query sequence $Q_1, Q_2, \ldots, Q_n$, by *incremental queries*, we mean finding the first solutions $\theta_1, \theta_2, \ldots, \theta_n$ to $Q_1, Q_2, \ldots, Q_n$ respectively in such a way that computation results from $Q_1, Q_2, \ldots, Q_{i-1}$ are re-used as much as possible to obtain the solution to Q_i.

In the following, we define a labeling for (skeleton) CPS-tree and an ordering for the labels. We assign to every clause in a program a label. These labels have a total ordering $<$. Let T be a (skeleton) CPS-tree. The nodes of T can be *labeled* as follows:

- The root node is labeled with the empty string ϵ.
- If T has node N with label L and N has m children associated with clauses $i_1, \ldots, i_m$, we assign $L.i_1$, …, $L.i_m$ as labels to the children respectively, where "." is the string concatenation operator.

A string L_1 is *lexicographically smaller than or equal to* string L_2, denoted by $L_1 \preceq L_2$, if and only if one of the following three conditions holds:

- L_1 is ϵ.
- $L_1 = n.L_1'$, $L_2 = m.L_2'$, and $n < m$.
- $L_1 = n.L_1'$, $L_2 = n.L_2'$, and $L_1' \preceq L_2'$.

We see that the first solution plays an important role in the definition of incremental queries. Let P be a program, Q a query, and T the (skeleton) CPS-tree of Q with respect to P. The *first* success leaf in T for Q with respect to P has label L such that for any success leaf with label L', $L \preceq L'$. Since the label of a node in a (skeleton) CPS-tree denotes the computation path from the root to that node, we abuse notation by identifying a node with its label.

We are now ready to present some properties of the (skeleton) CPS-tree adapted from [22]. The following properties concern the addition of constraints.

Property 3.1: Let P be a program. Suppose $Q_1 = \leftarrow B, C_1$ and $Q_2 = \leftarrow B, C_2$ are two queries to P such that

$$\mathcal{D} \models (\forall)(C_2 \Rightarrow C_1).$$

If L_1 and L_2 are the first success leaves of Q_1 and Q_2 respectively, then $L_1 \preceq L_2$. ∎

Property 3.2: Let L be the first success leaf of query $Q = \leftarrow B, C$. If $\mathcal{D} \models (\forall)(C' \Rightarrow C)$, then there is no solution of $Q' = \leftarrow B, C'$ with label $L' \preceq L$. ∎

The next property is useful for deletion of constraints.

Property 3.3: Let P be a program. If $Q_1 = \leftarrow B, C_1$ and $Q_2 = \leftarrow B, C_2$ are two queries to P such that

$$\mathcal{D} \models (\forall)(C_2 \Rightarrow C_1),$$

then any solution to Q_2 is also a solution to Q_1. ∎

Note, however, that the first success leaf to Q_2 is not necessarily the first success leaf to Q_1.

4 The VHLP-Scheme Revisited

4.1 The Original VHLP-Scheme

The VHLP-scheme [22] can be briefly described as follows. Denote the sequence of queries resulted from the previous operations of constraint additions and deletions as $\leftarrow B, C_i$, $i = 0, 1, 2, \ldots$, where B is a multiset of atoms and C_i a set of constraints in the query, respectively. The original query is denoted $\leftarrow B, C_0$. C_{i+1} is either $C_i \cup \{c\}$ or $C_i \setminus \{c\}$ where c is a single constraint. An oracle $O(C_i)$ is associated with each successful derivation of query $\leftarrow B, C_i$. We can use the label of the success leaf as $O(C_i)$.

To add a constraint c to C_i, the execution restarts from the root, with a new query $\leftarrow B, C_i \cup \{c\}$, and follows the oracle $O(C_i)$ (*i.e.* selecting the same clauses) until a failure occurs. Then the execution backtracks to the most recent choice point and proceeds to the next branch to the right, in the same way as Prolog would.

If C_{i+1} is $C_i \setminus \{c\}$, the execution also restarts from the root. However, the oracle it follows will be

$$O = \max_{k=1}^{i} \{O(C_k) \mid C_k \subseteq C_{i+1}\}$$

where max is determined by the lexicographic order of the oracles.

4.2 The VHLP-Scheme in Concurrent Constraint Logic Programming

The VHLP-Scheme is designed for sequential constraint logic programming languages (such as CHIP [7] or CLP($\mathcal{R}$) [10]). Its main contribution is show how to use previous oracles to guide the re-execution. This prevents the re-execution from entering the computation paths that are deemed to fail. Hence the overhead of re-execution is much reduced. Only the goals that need to be re-executed to obtain a new solution will be executed again.

However, this scheme is not suitable for Constraint Pandora. Constraint Pandora is a process-oriented concurrent language. Re-execution from the initial query implies repeating all the overhead of process creation, suspension, scheduling, termination, *etc.*, and this inevitably incurs much overhead that is not found in a sequential constraint logic programming language, in which re-execution is (roughly) equivalent to repeating a sequence of procedure calls.

In the VHLP-Scheme the oracles are used to guide the re-execution. However, it is not difficult to observe that $O(C_{i+1})$ and $O(C_i)$ usually have a common prefix. The choice point corresponding to the point where $O(C_{i+1})$ and $O(C_i)$ start to differ can hence be seen as an intelligent backtrack point. If the execution rolls back to this point (instead of re-executing from scratch up to this point), then a lot of execution context "above" this point can be re-used. This is the main idea of the *IQ scheme*.

5 The IQ Scheme for Constraint Pandora

The IQ scheme that we are proposing is based on backtracking. Backtracking is often portrayed as pessimistic and the cause of thrashing behavior in the literature. We are going to show otherwise. One aim of incremental queries is to re-use previous computation context as much as possible. In a backtracking-based language, execution context is saved on the trail at every step along an execution path so that execution can be unwound to a particular point upon backtracking. These various contexts may be expensive to build. The VHLP-scheme simply ignores this useful source of resources while the unwinding mechanism is inherent in a backtracking-based approach. In the following, *the discussion on the IQ scheme is based on the skeleton CPS-tree since backtrack points appear only in deadlock nodes.*

5.1 Addition of Constraints

Suppose we are in an interactive problem-solving session and we have solved up to query $Q_i = \leftarrow B, C_i$ in the incremental query sequence. The new goal $Q_{i+1} = \leftarrow B, C_{i+1}$ is obtained from Q_i by adding constraints. Without loss of generality, we assume that $C_{i+1} = C_i \cup \{c\}$, where c is a single constraint. As suggested in [22], there exists no solution to Q_{i+1} with a computation path lexicographically smaller than the path that leads to the first solution of Q_i by property 3.1. Suppose θ_i is the first solution of Q_i. If $\theta_i \cup \{c\}$ is consistent, we are done and the new solution θ_{i+1} is the union. Otherwise, we need to locate the cause of failure and backtrack from there. Unfortunately, conventional backtracking may explore many unrelevant choice points before locating the one that causes the failure. Van Hentenryck and Le Provost [22] regard this drawback as due to bad backtrack points and inactive use of constraints.

The IQ scheme modifies conventional backtracking as follows. Suppose the length of the execution path of Q_i is l and the content of the constraint store after the k^{th} $(0 \leq k \leq n)$ state transition is S_k. Note that $S_0 = C_i$ and $S_l = \theta_i$. In most implementations, S_k can be obtained from S_{k+1} by unwinding the trail along the execution path of Q_i. We first test the consistency of $S_l \cup \{c\}$. If the union is consistent, then it is the first solution of Q_{i+1}. Otherwise, starting from S_{l-1}, we unwind the trail to the first S_k such that $S_k \cup \{c\}$ is consistent. Standard execution resumes from the most recent backtrack point, say CH, of the $(k+1)^{\text{st}}$ state transition step.

The IQ scheme uses the added constraint c actively at each state transition to determine if it is "worthwhile" to backtrack at that state transition. We shall prove subsequently that the backtrack point CH obtained by IQ is the same as the optimal one obtained by the VHLP-scheme. Let us illustrate the IQ scheme with a small scheduling example exerted from [22] with the Constraint Pandora program in figure 1. The numbers at the end of each clause are the clause labels. The

```
gen(1).                                              ...(1)
gen(2).                                              ...(2)
gen(3).                                              ...(3)
gen(4).                                              ...(4)
gen(5).                                              ...(5)
gen(6).                                              ...(6)
schedule(X1,X2,X3,X4,X5,X6) :-                       ...(1)
    X1 ∈ {1,...,6},..., X6 ∈ {1,...,6},
    X1 ≠ X2,..., X5 ≠ X6,
    X1 > X2,
    X3 < X2,
    X4 < X2,
    X5 < X2,
    X6 < X2,
    gen(X1),...,gen(X6).
```

Figure 1: A Simple Scheduling Example in Constraint Pandora.

variables take their values from 1 to 6, are all different, and obey simple precedence constraints. Additional constraints are entered by users in an incremental querying session. Suppose the initial query is $Q_0 = \leftarrow \texttt{schedule}(\texttt{X1}, \texttt{X2}, \texttt{X3}, \texttt{X4}, \texttt{X5}, \texttt{X6})$. The first solution to Q_0 is $\{\texttt{X1} = 6, \texttt{X2} = 5, \texttt{X3} = 1, \texttt{X4} = 2, \texttt{X5} = 3, \texttt{X6} = 4.\}$ Consider the case when the user adds the constraint $\texttt{X3} \neq 1$. In the conventional backtracking approach, the new constraint is added into the execution context of the first solution. Backtracking will take place for the goals `gen(X6)`, `gen(X5)`, and `gen(X4)` before reaching the backtrack points of `gen(X3)`. The VHLP-scheme will follow the oracle of the Q_0, which is 1.6.5.1.2.3.4, until the prefix 1.6.5.1 is reached, where failure is detected. Standard backtracking execution is resumed at that point. Using the IQ scheme, we shall unwind backward through 4, 3, 2, 1, and then 5, where a consistency is detected. Again, standard execution is resumed at the same point as the VHLP-scheme.

5.2 Deletion of Constraints

Deletion of constraints is difficult to handle since the first solution to the new query can be to the "left" of first solution of the current query. Re-execution is unavoidable but it should be minimized. The VHLP-scheme re-executes from the beginning of the new query. The IQ scheme detects and unwinds to a "safe" state transition, where re-execution starts. We follow the VHLP-scheme to build oracles for the evaluation of each increment in an incremental query sequence, i.e. the computation path leading to a solution is stored. We use $O(C_k)$ to denote the oracle of the execution of query Q_k.

For deletion of constraint, we assume $C_{i+1} = C_i \setminus \{c\}$. The VHLP-scheme turns a deletion problem to an addition problem by picking an oracle prior to $O(C_i)$ that has the best pruning, i.e.

$$O = \max_{j=1}^{i-1}\{O(C_j) \mid C_j \subseteq C_{i+1}\},^3$$

where "max" uses the lexicographical ordering on strings. Property 3.2 guarantees that there is no solution to the new goal with a computation path smaller than O. We compute the common prefix of O and $O(C_i)$. Suppose $O(C_i) = L_1.p.L_2$ and $L_1.p$ is the common prefix of O and $O(C_i)$. The IQ scheme unwinds along $O(C_i)$ up to the execution context of p. Suppose that the constraint store at that point is $S = C \cup C_i$. We replace C_i by C_{i+1} in S and then proceed with oracle O in the same way as for additions in the VHLP-scheme. While the VHLP-scheme always starts re-execution from the beginning of a computation, L_1 is usually non-empty and re-execution along L_1 is eliminated.

The IQ scheme combines the best of both re-execution and backtracking by using previous execution oracles and last query's execution context, eliminating the need for re-execution as much as possible.

5.3 Comparison with the VHLP-Scheme

In this section we compare the IQ scheme with the VHLP-scheme. We show that the IQ scheme is more efficient than the VHLP-scheme even in sequential CLP languages, while the latter is not suitable in concurrent CLP languages.

5.3.1 Addition of Constraints

The following theorem shows an important property of the skeleton CPS-tree.

Theorem 5.1: Let $(N_0, N_1, \ldots, N_l)$ be the execution path of the skeleton CPS-tree corresponding to the successful derivation of a query $Q = \leftarrow B, C$, where N_0 is the root of the tree and N_l is a leaf and $N_i = \langle P_i, S_i \rangle$, $0 \le i \le l$. Let c be a constraint. Then there exists a j, $0 \le j \le n+1$, such that the $S_i \cup \{c\}$ is consistent for all i, $0 \le i < j$, and $S_i \cup \{c\}$ is inconsistent for all i, $j \le i \le l$.

Proof: By the definition of skeleton CPS-tree, N_i is reached from N_{i-1} by a sequence of state transitions. Hence $S_i \supseteq S_{i-1}$ for all i $(0 \le i \le l)$. Now consider c. There are three cases:

1. $S_l \cup \{c\}$ is consistent.
   Since $S_l \supseteq S_i$, we have $S_i \cup \{c\}$ is consistent, for all i $(0 \le i \le n)$. Therefore $j = l + 1$.

2. $S_l \cup \{c\}$ is inconsistent but $S_0 \cup \{c\}$ is consistent.

   (a) Obviously, there *exists* a j', $0 < j' \le l$ such that $S_{j'} \cup \{c\}$ is inconsistent and $S_{j'-1} \cup \{c\}$ is consistent.

   (b) Since $S_i \supseteq S_{j'}$ for all i $(j' \le i \le n)$, we have $S_i \cup \{c\} = S_{j'} \cup \{c\} \cup (S_i \setminus S_{j'})$ is inconsistent for all i $(j' \le i \le n)$. On the other hand, since $S_{j'-1} \supseteq S_i$ for all i $(0 \le i \le j'-1)$, we also have $S_i \cup \{c\} = S_{j'-1} \cup \{c\} \setminus (S_{j'-1} \setminus S_i)$, for all i $(0 \le i \le j'-1)$.

[3] Van Hentenrcyk and Le Provost [22] examines oracles up to $O(C_i)$. This is not necessary since $C_{i+1} = C_i \setminus \{c\} \subset C_i$.

(c) Hence we have $j = j'$.

3. $S_l \cup \{c\}$ is inconsistent and $S_0 \cup \{c\}$ is inconsistent.
   In this case trivially $j = 0$.

■

We now compare the IQ scheme and the VHLP-scheme based on theorem 5.1. In the VHLP-scheme, after the addition of a constraint c to the current query $Q = \leftarrow B, C$, the re-execution follows the oracle $O(C)$ until node N_j where the re-execution encounters the first failure. Then the re-execution backtracks to N_{j-1} (if $j > 0$, or it fails) and then proceeds from N_{j-1} without any oracle. The IQ scheme does this in a different way. The execution is unwound from the leaf N_l until the node N_{j-1}. Then c is added to C_{j-1} and the execution restarts from N_{j-1}. Therefore these two schemes virtually selects the same "restart point."

5.3.2 Deletion of Constraints

Deletion in the IQ scheme virtually follows the same principles in the VHLP-scheme. In the VHLP-scheme the re-execution follows the oracle $O = \max_{k=1}^{i}\{O(C_k) \mid C_k \subseteq C_{i+1}\}$, where $C_{i+1} = C_i \setminus \{c\}$ is the new set of constraints. The first portion of O, which is the common prefix of O and $O(C_i)$, exists in the system. In the IQ scheme we unwind the execution to the choice point that corresponds to the end of this common prefix and restart from that choice point with the new set C_{i+1} of constraints. This allows us to re-use the existing execution context as much as possible.

5.3.3 The IQ Scheme and the VHLP-Scheme: an Evaluation

As discussed above, these two schemes select virtually the same "restart point." However, the IQ scheme is more efficient than the VHLP-scheme in a number of aspects, especially in a Constraint Pandora language.

1. It is easy to see that the IQ scheme makes use of not only the previous execution paths (the "oracles"), but also the previous *execution context*. Denote the "restart point" as N_j, the IQ scheme re-uses the nodes $N_0, N_1, \ldots, N_{j-1}$, which already exist in the system, instead of re-creating them from scratch as in the VHLP-scheme.

2. In a concurrent constraint logic programming language such as Constraint Pandora, unwinding is usually much cheaper than re-execution. It is not difficult to imagine that thousands of reduction take place between two choice points in the execution of a Constraint Pandora program, most of which incur *unconditional updates* (*i.e.*, updates that need not be trailed and undone upon backtracking) [1] to the process pool and the constraint store. Unwinding from a choice point to a previous one is definitely much cheaper than re-executing the thousands of reductions, which involve creation and execution of thousands of processes.

3. Even for sequential constraint logic programming languages such as CHIP and CLP($\mathcal{R}$), unwinding one step is usually cheaper than re-executing one step. Unwinding is usually composed of untrailing and discarding work done, while re-executing is a far more complicated process of goal reduction, choice point and environment creation, and constraint assimilation.

4. Whether the IQ scheme is more efficient than the VHLP-scheme for a particular program and query depends on the location of N_j. In general, the VHLP-scheme is more efficient for smaller j, and the IQ scheme is more efficient for larger j. In the average case, say $j = \frac{l}{2}$, the IQ scheme is better than the VHLP-scheme as unwinding $\frac{l}{2}$ steps is cheaper than re-executing $\frac{l}{2}$ steps. This is especially true for Constraint Pandora, in which the cost of re-executing from a choice point to the next one could be several thousand times more expensive than unwinding from a choice point to the previous one.

5.4 Implementation Issues

We are building an implementation of Constraint Pandora based on [6] and [1]. There are a few points to mention in the implementation that are related to the support of incremental queries.

In [1] Bahgat describes a backtracking-based Pandora Abstract Machine (PAM) for multiprocessors designed based on [6]. In the PAM not all updates to the system are trailed. Only those updates that need to be undone in order to restore to the previous choice point are trailed. These are called "conditional updates." Others, which are "unconditional updates," are not trailed nor undone upon backtracking. Intuitively they are due to the deterministic execution of processes and will be automatically undone when the "conditional updates" are undone. Hence unwinding in PAM is in general much cheaper than re-execution. This supports our arguments in section 5.3.3.

In constraint deletion, the IQ scheme explicitly separates the constraint store into two parts: the set C_i of constraints in query Q_i and the other constraints C. When the execution is unwound to the "restart point," we replace C_i with $C_{i+1} = C_i \setminus \{c\}$ and then proceed from there. From an implementation point of view, this requires us to keep two constraint stores during the execution of Q_i, one for C_i and one for C, so that the constraint store C_i can be completely replaced by C_{i+1}. The details of operations depend on the underlying constraint domains and are out of the scope of this paper.

6 Concluding Remarks

User-interface of (constraint) logic programming systems is an aspect that cannot be ignored. Incremental queries, which has applications in spreadsheet, databases, budgeting, scheduling, *etc.*, is an important example. Incremental queries can be used in constraint optimization problems, which involve global search. Most logic programming languages tackle optimization by using higher-order predicates, such as the `minimize/2` predicate in CHIP. With incremental queries, users can post the original constraint *satisfaction* problem and the cost function in the initial query. The result displayed after each increment helps the users to adjust the cost constraints in subsequent increments. The flexibility in incremental queries allows the users *guide* the optimization search by adding stronger constraints or deleting undesirable ones.

In this paper, we have presented an efficient execution scheme IQ to implement incremental queries for CLP languages, especially the concurrent ones. The IQ scheme makes use of oracles as well as previous execution context to minimize re-execution, resulting in a higher efficiency than the VHLP-scheme in both the sequential and concurrent cases. There are three major differences between the IQ scheme and the VHLP-scheme. First, the IQ scheme is designed for concurrent constraint logic programming languages based on the skeleton CPS-tree. The VHLP-scheme is designed for sequential constraint logic programming languages based on an SLD-tree. Second, the main technique used in the IQ scheme is unwinding while that in the VHLP-scheme is re-execution. Third, the IQ scheme makes use of both previous execution paths *and* execution context, while the VHLP-scheme does not make use of the latter.

A prototype of the IQ scheme for Constraint Pandora with finite domain constraint-solving capability has been built using Prolog to demonstrate the feasibility of our approach. Future work includes exploring efficient implementation techniques for the IQ scheme on different Constraint Pandora instances. We have intentionally avoided tying the IQ scheme to a particular instance of Constraint Pandora. It is expected, however, that the structure of constraint domain can be exploited in the implementation of the IQ scheme.

References

[1] R. Bahgat. *Non-Deterministic Concurrent Logic Programming in Pandora.* World Scientific Publishing, 1992.

[2] P. Chatalic. Incremental techniques and Prolog. Technical Report TR-LP-23, European Computer-Industry Research Centre, June 1987.

[3] P. Chatalic. IMPRO: An environment for incremental execution in Prolog. Technical Report TR-LP-42, European Computer-Industry Research Centre, May 1989.

[4] M.H.M. Cheng, M.H. van Emden, and J.H.M. Lee. Tables as a user interface for logic programs. In *Proceedings of the International Conference on Fifth Generation Computer Systems 1988*, pages 784–791, Tokyo, Japan, November–December 1988. Ohmsha, Ltd. Also as Technical Report DCS-97-IR (LP-7), University of Victoria, Victoria, Canada, 1988.

[5] K.L. Clark. Parallel logic programming. *The Computer Journal*, 33(6):482–493, 1990.

[6] J. Crammond. The abstract machine and implementation of parallel parlog. *New Generation Computing*, 10:385–422, 1992.

[7] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The constraint logic programming language CHIP. In *Proceedings of the International Conference on Fifth Generation Computer Systems (FGCS'88)*, pages 693–702, Tokyo, Japan, December 1988.

[8] S. Gregory. *Parallel Logic Programming in PARLOG: The Language and Its Implementation.* Addison-Wesley, 1987.

[9] J. Jaffar and J-L. Lassez. Constraint logic programming. In *Proceedings of the 14th ACM POPL Conference*, pages 111–119, Munich, January 1987.

[10] J. Jaffar, S. Michaylov, P.J. Stuckey, and R.H.C. Yap. The CLP($\mathcal{R}$) language and system. *ACM Transactions on Programming Languages and Systems*, 14(3):339–395, July 1992.

[11] H.F. Leung. *Distributed Constraint Logic Programming.* World Scientific Publishing, 1993. (in print).

[12] J.W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, second edition, 1987.

[13] A.K. Mackworth. Consistency in networks of relations. *AI Journal*, 8(1):99–118, 1977.

[14] M.J. Maher. Logic semantics for a class of committed-choice programs. In J.-L. Lassez, editor, *Proceedings of the Fourth International Conference on Logic Programming*, pages 858–876. The MIT Press, 1987.

[15] M.J. Maher and P.J. Stuckey. Expanding query power in contraint logic programming languages. In *Proceedings of the North American Conference on Logic Programming 1989*, pages 20–36, Cleveland, Ohio, U.S.A., 1989.

[16] M. Ohki, A. Takeuchi, and K. Furukawa. A framework for interactive problem solving based on interactive query revision. In E. Wada, editor, *Proceedings of the Fifth Conference on Logic Programming '86*, pages 137–146, Tokyo, Japan, June 1986. Spriner-Verlag.

[17] E.Y. Shapiro. Concurrent Prolog: a progress report. *IEEE Computer*, 19(8):44–58, 1986.

[18] K. Ueda. *Guarded Horn Clause.* PhD thesis, University of Tokyo, Tokyo, Japan, 1986.

[19] M.H. van Emden. Logic as an interaction language. In *Proceedings of the Fifth Conference on Canadian Society for Computational Studies in Intelligence*, pages 126–128, 1984.

[20] M.H. van Emden, M. Ohki, and A. Takeuchi. Spreadsheets with incremental queries as a user interface for logic programming. *New Generation Computing, OHMSHA, LTD. and Springer-Verlag*, 4:287–304, 1986.

[21] M.H. van Emden and D.A. Rosenblueth. A spreadsheet interface for Prolog. IBM SUR Grant Project No. 058CT-35, final report, August 1986.

[22] P. Van Hentenryck and T. Le Provost. Incremental search in constraint logic programming. *New Generation Computing*, 9:257–275, 1991.

[23] D.H.D. Warren. The Andorra model. Presented at Gigalips Project Workshop, University of Manchester, March 1992.