An Execution Scheme for Interactive Problem-Solving in Concurrent Constraint Logic Programming Languages

Jimmy H.M. Lee and Ho-fung Leung
Department of Computer Science and Engineering
The Chinese University of Hong Kong
Shatin, N.T., Hong Kong, China

Facsimile: +852 2603 5024
Email: {jlee, lhf}@cse.cuhk.edu.hk

Abstract

Van Emden's *incremental queries* address the inadequacy of current Prolog-style querying mechanism in most logic programming systems for interactive problem-solving. In the context of constraint logic programming, incremental queries involve 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. We use Van Hentenryck and Le Provost's scheme (VHLP-scheme hereafter), a re-execution approach, as a starting point. Re-execution is costly in concurrent languages, in which process creation and inter-process communications are common operations. 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. We modify the conventional trail-unwinding operation in such a way that constraints are used actively to prune the search space. 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. We show the feasibility of our proposal by incorporating the IQ-scheme into IFD-Constraint Pandora, an instance of Constraint Pandora supporting interval and finite domain constraint solving. Our preliminary results agree with that of theoretical analysis.

Keywords: Incremental Execution, Concurrent Constraint Logic Programming

1 Introduction

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 [5]: *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] [29], 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.

In his paper, Van Emden [**?**] points out that incremental queries are useful for nondeterministic goals such as interactively constructing a timetable that satisfies a set of constraints expressed as incremental queries. Van Hentenryck and Le Provost [35] suggest that incremental queries are desirable for adaptation of an existing solution to include new information, modification of a solution to include priorities and preferences, as well as general interactive problem solving.

We are interested in incorporating an efficient incremental querying mechanism into *Constraint Pandora* [16], which defines a family of non-deterministic concurrent constraint logic programming languages. Constraint Pandora is essentially a concurrent version of the CLP scheme [13] based on the execution model of Pandora [1]. Unlike the committed-choice concurrent logic

[1]Also called *incremental execution* and *incremental search* by other authors [3, 4, 35].

languages, such as Parlog [11], GHC [28], and Concurrent Prolog [23], Constraint Pandora supports both don't-know and don't-care non-determinisms, allowing the generation of multiple answers to a query.

In order to avoid the overhead of process creation and inter-process communication, we minimize re-execution and propose to use computation context that is saved along an execution path. Such context can be recovered conveniently by unwinding the trail, a basic operation provided by backtracking. Van Hentenryck and Le Provost [35] criticize backtracking as the source of thrashing behavior [18] due to bad backtrack points and passive use of constraints. We show otherwise: conventional backtracking can be modified in such a way that constraints are used actively to prune search space.

We compare our scheme with that of Van Hentenryck and Le Provost [35] (VHLP-scheme hereafter). 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 safety-guaranteed intermediate execution context, located using oracles. What is described is a scheme for efficient implementation of incremental queries; hence the name IQ. To demonstrate the feasibility of our proposal, we have incorporated IQ in IFD-Constraint Pandora, an instance of Constraint Pandora with finite and interval domain constraints. Preliminary results support our theoretical analysis.

The paper is organized as follows. In section 2, we review related work. Section 3.1 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 3.2. 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 in section 6. In section 7, we introduce IFD-Constraint Pandora and discuss the implementation issues in incorporating IQ into IFD-Constraint Pandora. We end the section with preliminary benchmarking results. Section 8 contains the concluding remarks of the research and sheds light on future work.

2 Related Work

Work in incremental query processing in logic programming can be divided into two camps: functionality and efficient implementation.

2.1 Functionality

The original incremental queries proposal [29], intended as solution to the problem of doing computer graphics interactively and yet without appeal to side effects, considers only addition of constraints. Van Emden *et al.* [31, 30] refine the method in such a way that users can specify either addition or deletion of goals in increments. They also show how incremental queries couple well with a spreadsheet user-interface for logic programs. Ohki *et al.* [22] generalize incremental queries so that users are allowed, in addition to entering queries incrementally, to modify queries and programs, and to act as a part of the inference engine. Users can help the system by giving a solution for a part of the problem, such as assigning a solution to an unbound variable. On backtracking, the system is able to query the user for alternatives of the previous interaction. Cheng *et al.* [5] incorporate incremental queries in the TuplePipes dataflow query model. Again, addition, deletion, and modification of goals are considered. Together with a table (as in relational databases) user-interface, the result is a deductive databases system. Chatalic [4] also allows the user to modify the program during execution of an incremental query. Fages *et al.* [10] present a theoretical analysis of incremental queries in constraint logic programming, which considers addition and deletion of both goals and constraints.

2.2 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 but is adequate for prototyping purpose. 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.

Van Emden *et al.* [30] propose an intelligent backtracking scheme for incremental queries and implement it in a Prolog meta-interpreter. Van Emden and Rosenblueth [31] incorporate the scheme in a WAM-like Prolog machine and implement it at the emulator level. Chatalic [3] uses the meta-interpreter of [30] as a starting point and increases its efficiency by coroutining [21]. It uses the `assert` and `retract` meta-calls to keep track of backtracking information. Chatalic [4] improves upon his own work by extending the meta-interpreter with goal dependency analysis. The aforementioned work is for Prolog.

Van Hentenryck and Le Provost [35] propose the VHLP-scheme 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 Stuckey [20] also use execution paths to guide the re-execution of goals but their approach relies on users to decide where to backtrack.

3 Background

3.1 Constraint Pandora

In this subsection, we describe Constraint Pandora. The CLP scheme [13] is a special case of Constraint Pandora and the results shown in this paper apply equally well to sequential CLP languages. Section 3.1.1 gives the syntax of Constraint Pandora, followed by a presentation of the operational semantics in section 3.1.2. Section 3.1.3 defines Constraint Pandora search tree and its skeleton version, which are important tools in the analysis of our method.

3.1.1 The Language

Constraint Pandora [16] 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 [19], the constraints in clause guards are *ask*-constraints and those in clause bodies are *tell*-constraints. Intuitively speaking, we check the constraint store for entailment of ask-constraints, and tell-constraints are "assimilated" into the constraint store.

A clause in Constraint Pandora is of the form

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

where H is an atom called the *clause head*, 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`[2]) are seen as system predicates in Pandora, in Constraint Pandora they are seen as

[2]`<=/2` is a system predicate in Parlog (and Pandora). The goal `T <= X` fails if and only if `T` and `X` are not unifiable, and succeeds if and only if `T` and `X` can be unified without binding any variables in `T`. Otherwise, the goal suspends.

constraints in the Herbrand Universe. 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 "C_{ask} and B and C_{tell} implies H."

3.1.2 Operational Semantics

Constraint Pandora belongs to the family of concurrent logic programming languages [6] and is related to the ALPS framework [19]. As in Pandora, "goal" is synonymous with "process" in the constraint version. The execution of Constraint Pandora programs can be abstracted to a sequence of *derivation steps*. A derivation step is a transition from an *execution state* to another. An execution state of Constraint Pandora consists of a *process pool* P, which is a multiset of processes, and a *constraint store* S, which is a set of constraints. 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 the 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.[3] 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 [37], while the don't-care processes use a more "eager" Parlog-like commit law [11].

There are two kinds of derivation steps. When a process executes, it "commits" to a candidate clause (for a don't-know process, *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 derivation steps occurs when every process in the system suspends. The execution state is then said to be a *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 [17]

[3]A clause $H \leftarrow C_{ask} : B, C_{tell}$ is a *candidate clause* of a process A if and only if $C_{ask} \wedge (A = H)$ is entailed (validated) by the constraint store S. This condition can be formally expressed as [19]:

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

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

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 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 it reaches a state in which 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 [16] guarantees that

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

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

3.1.3 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 derivation step. 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
- a leaf with an inconsistent constraint store, and
- a leaf with only suspended don't-care processes in its process pool.

The first kind of leaves corresponds to successful executions. The solution of a successful execution is the constraint store in the leaf. The other two kinds correspond 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 is 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, since we allow only one process to spawn at each derivation step, the execution strategy of Constraint Pandora is similar to the generalized SLD-resolution as defined in [35]. 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.

3.2 Incremental Queries and Preliminaries

In this subsection, we follow the style of [35] 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 [35] 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 an *incremental query*, 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 each Q_i $(2 \leq i \leq n)$.

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. The labels for each of the clauses of the same procedure are unique, and there is a total ordering "<" among them. 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 $L_1' \preceq L_2'$.

Intuitively, $L_1 \preceq L_2$ means that the node with label L_1 is to the "left" of the node with label L_2 in a (skeleton) CPS-tree. 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 throughout this paper.

We are now ready to present some properties of the (skeleton) CPS-tree as adapted from [35]. 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 $L_1 \preceq L_2$. ■

The next property is a corollary to property 3.1.

Property 3.2: Let L be the first success leaf of query $Q = \leftarrow B, C_1$. If $\mathcal{D} \models (\forall)(C_2 \Rightarrow C_1)$, then there is no solution of $Q' = \leftarrow B, C_2$ with label $L' \preceq L$. ■

The next property concerns the 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. The first success leaf to Q_2 is, however, not necessarily the first success leaf to Q_1. ■

4 The VHLP-Scheme Revisited

4.1 The Original VHLP-Scheme

The idea of the VHLP-scheme [35] is based on the following observations. Consider the search tree produced using the Prolog computation rule and selection function. Suppose the *first* successful derivation is found for the current query. Property 3.2 shows that this *first* successful derivation is the "leftmost" one among all others. Now consider the addition of a constraint to the current query. The new successful derivation will either be the same as the current one or occurs to the "right" of the current one (property 3.1). Likewise, after deletion of a constraint from the current query, the new successful derivation will either be the same as the current one or occurs to the "left" of the current one (property 3.3). Here we are abusing the terminologies when we talk about "left" and "right," as the search tree in general changes significantly after the addition or deletion of constraints.

We describe VHLP-scheme 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 the set of constraints in the query, respectively. The original query is $\leftarrow B, C_0$. C_{i+1} is either $C_i \cup \{c\}$ or $C_i \setminus \{c\}$ where c is a single constraint. An oracle $O(C_i)$ that records the execution path is associated with each successful derivation of query $\leftarrow B, C_i$. One representation of $O(C_i)$, which we employ in this paper, is the label of the success leaf.

To add a constraint c to C_i, the execution re-starts 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 re-starts 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. This deletion procedure essentially turns a constraint deletion problem into a constraint addition problem by examining only oracles $O(C_k)$ where $C_k \subseteq C_{i+1}$. The “maximal” oracle O chosen is the one that achieves the best pruning, as determined by lexicographic order of string labels.

4.2 The VHLP-Scheme in Nondeterministic Concurrent Constraint Logic Programming

The VHLP-scheme is designed for sequential constraint logic programming languages, such as CHIP [9] or CLP($\mathcal{R}$) [14]. It shows how to use previous oracles to guide re-execution. This prevents the re-execution from entering the computation paths that are deemed to fail. Hence the overhead of re-execution is reduced.

However, this scheme is in general not efficient 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.* Therefore, the effect of using oracles in Constraint Pandora to reduce the amount of computation in re-execution is not as drastic as in sequential backtracking-based languages.

In the VHLP-scheme the oracles are used to guide the re-execution. However, it is not difficult to observe that both $O(C_i)$ and O usually have a common prefix with $O(C_{i+1})$. 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 instead of being re-computed. This is the main idea of the *IQ-scheme.*

5 The IQ Scheme

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, particularly in a nondeterministic concurrent CLP language. The VHLP-scheme simply ignores this useful resources while the unwinding mechanism is inherent in a backtracking-based language. 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 [35], 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, backtrack, and re-execute from there. Unfortunately, conventional backtracking may explore many unrelevant choice points before locating the one that causes the failure. Van Hentenryck and Le Provost [35] regard this drawback as due to bad backtrack points and inactive use of constraints.

The IQ-scheme modifies conventional backtracking as depicted in figure 1. Suppose the length of the execution path of Q_i is l and the content of the constraint store after the k^{th} $(0 \leq k \leq l)$ derivation step 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 $(k+1)^{\text{st}}$ derivation step.

The IQ-scheme uses the added constraint c actively at each derivation

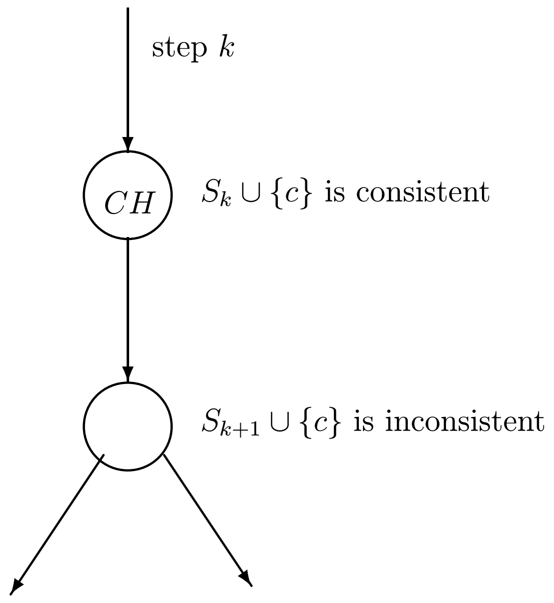
step k
CH
$S_k \cup \{c\}$ is consistent
$S_{k+1} \cup \{c\}$ is inconsistent

Figure 1: The Most Recent Backtrack Point CH

step to determine if it is "worthwhile" to backtrack at that derivation step. 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 [35] with the Constraint Pandora program in figure 2. The numbers at the end of each clause are the clause labels. The 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 (the third choice point) is reached, where failure is detected. Standard backtracking execution is resumed at the fourth choice point. Using the IQ-scheme, we unwind the trail backward through 4, 3, 2, 1, and then 5, where consistency is detected. Standard execution is resumed at the choice point at 5, 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 by property 3.3. 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, 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 picks an oracle prior to $O(C_i)$ that has the best pruning, *i.e.*

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

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 Correctness of the IQ Scheme

In this section we shall prove the correctness of the IQ scheme. Theorem 5.1 establishes the correctness of the algorithm for addition of constraints, while theorem 5.2 establishes the correctness of the algorithm for deletion of constraints.

Theorem 5.1: Let L_1 be the label of the first success leaf of $Q_1 = \leftarrow B, C_1$ and L_2 be the label of the first success leaf of $Q_2 = \leftarrow B, C_1 \cup \{c\}$. Suppose the new constraint c is added after the node labelled L_1 is reached. The node labelled L_2 will eventually be reached.

Proof: Since $C_1 \cup \{c\} \Rightarrow C_1$, by Property 3.1, we have $L_1 \preceq L_2$. Let L be the (possibly empty) common prefix of L_1 and L_2 so that $L_1 = L.n_1.Q_1$ and $L_2 = L.n_2.Q_2$. Obviously, the constraint stores in the nodes labelled by L

and its prefixes are consistent, whether the root of the search tree is Q_1 or Q_2. In the IQ scheme, addition of the new constraint c after the node labelled L_1 is reached requires the resumption of standard execution from the node labelled L after c is added to the constraint store of the node. Therefore, the node labelled L_2 will eventually be reached as $L_1 \preceq L_2$. ■

Theorem 5.2: Let L_1 be the label of the first success leaf of $Q_1 = \leftarrow B, C_1 \cup \{c\}$ and L_2 be the label of the first success leaf of $Q_2 = \leftarrow B, C_1$. Suppose the constraint c is deleted after the node labelled L_1 is reached. The node labelled L_2 will eventually be reached.

Proof: Since $C_1 \cup \{c\} \Rightarrow C_1$, by Property 3.1, we have $L_2 \preceq L_1$. Let L be the (possibly empty) common prefix of L_1 and L_2 so that $L_1 = L.n_1.Q_1$ and $L_2 = L.n_2.Q_2$. Obviously, the constraint stores in the nodes labelled by L and its prefixes are consistent, whether the root of the search tree is Q_1 or Q_2. In the IQ scheme, deletion of the existing constraint c after the node labelled L_1 is reached requires the resumption of standard execution from the node labelled L after c is removed from the constraint store of the node. Therefore, the node labelled L_2 will eventually be reached as $L_2 \preceq L_1$. ■

6 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.

6.1 Addition of Constraints

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

Theorem 6.1: 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 $N_i = \langle P_i, S_i \rangle$, $0 \leq i \leq l$. Let c be a constraint. Then there exists a j, $0 \leq j \leq l+1$, such that the $S_{i_1} \cup \{c\}$ is consistent for all i_1, $0 \leq i_1 < j$, and $S_{i_2} \cup \{c\}$ is inconsistent for all i_2, $j \leq i_2 \leq l$.

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

1. $S_l \cup \{c\}$ is consistent.
   Since $S_l \supseteq S_{i_1}$, $S_{i_1} \cup \{c\}$ is also consistent, for all i_1 $(0 \leq i_1 \leq l)$. Therefore $j = l + 1$.

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

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

   (b) Since $S_{i_2} \supseteq S_{j'}$ for all i_2 $(j' \leq i_2 \leq l)$, we have $S_{i_2} \cup \{c\} = (S_{j'} \cup \{c\}) \cup (S_{i_2} \setminus S_{j'})$ is inconsistent for all i_2 $(j' \leq i_2 \leq l)$. On the other hand, since $S_{j'-1} \supseteq S_{i_1}$ for all i_1 $(0 \leq i_1 \leq j'-1)$, we have $S_{i_1} \cup \{c\} = (S_{j'-1} \cup \{c\}) \setminus (S_{j'-1} \setminus S_{i_1})$ inconsistent, for all i_1 $(0 \leq i_1 \leq j'-1)$.

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

3. $S_l \cup \{c\}$ is inconsistent and $S_0 \cup \{c\}$ is inconsistent.

   In this case trivially $j = 0$.

■

Figure 3 helps to visualize the proof of theorem 6.1. We now compare the IQ-scheme and the VHLP-scheme based on theorem 6.1. 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 re-starts from N_{j-1}. Therefore these two schemes selects the same "re-start point."

6.2 Deletion of Constraints

Deletion in the IQ-scheme follows the same principles in the VHLP-scheme. In the VHLP-scheme, 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, from the root of the tree. 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 re-start from that choice point. The worst case scenario occurs only when O has no common prefix with $O(C_i)$. In that case, the IQ-scheme also starts execution from the root. Thus the IQ-scheme is at least as efficient as the VHLP-scheme.

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

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

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 "re-start point" as N_{j-1}, 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.

In a concurrent constraint logic programming language such as Constraint Pandora, unwinding is usually much cheaper than re-execution. Bahgat [1] describes a backtracking-based Pandora Abstract Machine (PAM) for multiprocessors designed based on [7]. 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. 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 processes, setting up choice points and environments, as well as assimilating constraints. This argument also applies in sequential constraint logic programming languages such as CHIP and CLP($\mathcal{R}$).

However, it should be noted that whether the IQ-scheme is more efficient than the VHLP-scheme for a particular program and query depends on the position 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.

7 IQ in IFD-Constraint Pandora

Efficiency of the IQ-scheme in a (concurrent) constraint logic programming language depends on efficient implementation of the constraint addition and deletion operations. Constraint addition amounts to testing the satisfiability (and also computing the solved form) of a previously consistent constraint store augmented with a new set of constraints. Since constraint addition is a frequent step in the normal execution, we can basically adopt the existing constraint addition and assimilation mechanism for constraint addition in the IQ-scheme. Constraint deletion concerns the (re)computation of a solved form of a previously consistent constraint store after a set of constraints is deleted from the store. The difficulty of this operation varies with the underlying constraint domain. Huynh and Marriott [27] and Helm *et al.* [12] develop efficient incremental constraint deletion algorithms for linear equalities and inequalities. In this section, we present a realization of the IQ-scheme in an instance of Constraint Pandora with interval and finite domain constraints.

Lee and van Emden [15] show that the domain reduction operator used for interval domain constraint is an instance of LAIR of CHIP [33] and that the lookahead-efficient computation rule is a constraint relaxation algorithm similar to AC-3 [18]. Following this approach, we generalize FD-Constraint Pandora [16] to handle also interval domain constraints. We call the extended language *IFD-Constraint Pandora*, which can be regarded as a concurrent version of ICHIP [15]. IFD-Constraint Pandora features the notion of domains and domain variables [32], abbreviated as *d-variables*. A *domain* is either a non-empty finite set of constants or a non-empty interval of real numbers, whose bounds are floating-point numbers. Each d-variable has the form X^D, where D is the associated domain of X^D. A constraint in IFD-Constraint Pandora contains only ground terms and d-variables. Figure 4 contains a IFD-Constraint Pandora program that solves the 4-queens problem. The "`::/2`" predicate declares d-variables and their associated domains. In this example, we declare the domains of all d-variables as $\{1, 2, 3, 4\}$.

7.1 IFD-Constraint Pandora

Each derivation step in Constraint Pandora consists of two substeps: (1) generation of new processes and constraints and (2) assimilation of the new constraints into the constraint store. Substep (1) is shared by all instances of Constraint Pandora. Each distinguished instance of Constraint Pandora, however, has a unique substep (2). In the following, we use $\vec{D}$ to denote a vector $(D_1, \ldots, D_n)$ of n elements. The constraint assimilation step of IFD-Constraint Pandora is shown in algorithm 5, which is executed once in each derivation step after new constraints are generated. Algorithm 5 is an adaptation of the consistency algorithm AC-3 [18]. The core of the algorithm is the *domain restriction operation* `REVISE()`. Given a constraint p, the domain restriction operation removes inconsistent values from the domains of d-variables appearing in p. The `REVISE()` subprogram differs for different kinds of constraints. The subprogram can be as simple as functions that enforce node- and arc-consistency [18], or it can also be as complex as the algorithms behind the `atmost` and `accumulate` constraints in CHIP [34, 36, 26]. For example,

> Node-consistency: $\texttt{REVISE}(\texttt{even}(X^{\{1,2,3,4,5\}}), (\{1,2,3,4,5\})) = (\{2,4\})$
> Interval Arc-consistency: $\texttt{REVISE}(X^{[1,5]}+Y^{[3,6]} = 10, ([1,5],[3,6])) =$
> $([4,5],[5,6])$.

where the notation "$[x, y]$" denotes a close interval between x and y. We leave the implementation of `REVISE()` unspecified and assume appropriate implementation that satisfies *contractance*, *correctness*, *monotonicity*, and *idempotence* defined in the sense of Benhamou and Older [2]. Algorithm 5 is thus an efficient procedure for coordinating the application of `REVISE()` on constraints in a given set of constraints.

Algorithm 5 resembles a classical iterative numerical-approximation technique called "relaxation" [24], which was adopted in a constraint system in [25]. In relaxation, we make an initial guess at the values of the unknowns and then estimate the error of the guess. New guesses are then made using the original guess and the error estimate. This process is repeated until the error is sufficiently small or satisfies a certain termination criterion. In algorithm 5, the initial guess is the input domains. Errors are values in the domains that cannot satisfy the relation associated with the constraint. New domains are computed by removing the inconsistent values. These pruned domains are better approximations of the originals in the sense that they provide more accurate inclusions of the answers. Numerical relaxation may fail to converge or terminate even when the constraints have a solution. Algorithm 5 does not suffer from this problem as shown by the following theorem.

Theorem 7.1: Algorithm 5 always terminates.

Proof: Algorithm 5 halts either when domain restriction of a constraint fails or the set A becomes empty. In the former case, the set of constraints is inconsistent. For the latter case, we observe that the size of set A *decreases* after each iteration of the algorithm unless domains of d-variables are pruned and constraints are moved from P to A. However, the number of possible domains are finite since the precision of a floating-point system. Thus domain restriction cannot occur. Therefore, set A must become empty after a finite number of iterations. ∎

7.2 Effective Points for Constraints

"All constraints are equal, but some constraints are more equal than others."

Operationally, we differentiate between two types of constraints. A constraint *takes effect* if, during execution of algorithm 5, `REVISE()` has used it to prune domains of its associated d-variables, resulting in a substitution which might be propagated to the other constraints in the constraint store. These activities occur in lines 10–18 of algorithm 5. This notion of *effectiveness* of a constraint is important in the deletion of constraint. Suppose the constraint store denotes a conjunction of constraints C. We can compute $C \setminus \{c\}$ by simply removing c from the constraint store if c has not taken effect in the store. Otherwise, removing a constraint that has taken effect involves trailing of bindings and undoing substitutions.

To illustrate the concept of effectiveness, we use the 4-queens program in figure 4, which works by generating the necessary disequality constraints "$\neq$`/2`" to prohibit the queens to be on the same row, positive diagonals, and negative diagonals. The `indomain/1` predicate is then used to enumerate elements in the domains of d-variables and instantiate d-variables to find solutions, by augmenting tree searching with constraint propagation. By virtue of the test-and-generate paradigm in constraint programming, the `indomain/1` predicate is defined to execute only after all other predicates either suspend or terminate. Suppose the initial query is "`?- queens([X1,X2,X3,X4]).`" Let us examine the constraint store right before the first `indomain/1` process starts execution. There are a total of 18 constraints in the store, listed as follows.

```
X1 ≠ X2
X1 ≠ X2 + 1
X1 ≠ X2 - 1
X1 ≠ X3      X2 ≠ X3
X1 ≠ X3 + 2  X2 ≠ X3 + 1
X1 ≠ X3 - 2  X2 ≠ X3 - 1
X1 ≠ X4      X2 ≠ X4      X3 ≠ X4
X1 ≠ X4 + 3  X2 ≠ X4 + 2  X3 ≠ X4 + 1
X1 ≠ X4 - 3  X2 ≠ X4 - 2  X3 ≠ X4 - 1
```

None of these constraints can eliminate values from the domains of the associated d-variables. Thus, no constraints have taken effect yet. Each d-variable has associated domain $\{1, 2, 3, 4\}$. Suppose `indomain(X1)` is now executed and causes `X1` to be bound to `1`. This instantiation triggers algorithm 5 to initiate a sequence of constraint propagation. For example, the value `1` is eliminated from the domain of `X2` by `REVISE()` using the constraint "`X1` $\neq$ `X2`." This is the first time that the constraint "`X1` $\neq$ `X2`" prunes the domains of its d-variables. Thus "`X1` $\neq$ `X2`" takes effect and this effect is propagated to other constraints involving variable `X2`.

We are now ready to define the notion of effectiveness formally. Consider the execution of an IFD-Constraint Pandora program. Let N be a non-leaf node in a CPS-tree and N' be one of its children. If, during the execution of algorithm 5 in the derivation step from N to N', the domain of a d-variable in the constraint store is pruned as a result of application of `REVISE()` using a constraint c in the constraint store of N, then N is said to be the *action* *point* of the constraint c. If N is an action point of c and all of its ancestors in the CPS-tree are not, then N is said to be the *effective point* of c, and is denoted $\mathcal{E}_c$.

7.3 An Approximation to IQ

We assume an IFD-Constraint Pandora implementation scheme similar to those presented in [1, 16] in the following presentation. In constraint addition, a modified trail-unwinding operation is required. At each unwinding step, we extract the constraint store at the choice points, submit the newly added constraint to the constraint store and exercise algorithm 5 for constraint assimilation. This unwinding process starts from the leaf node corresponding to the current success execution, and is repeated until either the root of the CPS-tree is reached or consistency is detected. Thus we can adhere to the IQ-scheme faithfully with addition of constraints.

With deletion of constraints, we compute the common prefix of O and $O(C_i)$ and unwind the execution to the choice point N that corresponds to the end of this common prefix (note that N is a choice point as we are considering a skeleton CPS-tree). Assume that $C_{i+1} = C_i \setminus \{c\}$ and the constraint store S at N is $C \cup C_i$ for some set of constraints C. Since constraints in S are fully assimilated, we cannot simply remove c from S to obtain $C \cup C_{i+1}$ in general. It is safe to do so only if c has not taken effect at N. We present an *approximation to IQ* for constraint deletion:

- if one of the ancestors of N is an effective point $\mathcal{E}_c$ for constraint c, then
    - if $\mathcal{E}_c$ is not the root node of the CPS-tree, we further unwind the

trail to the first choice point N' at or above $\mathcal{E}_c$, remove c from the constraint store,

  – otherwise, unwind to the root node and remove c from the constraint store,

- otherwise, remove c from the constraint store of N.

After the removal of c from the constraint store, we proceed with oracle O in the same way as for additions in the VHLP-scheme. To support the approximate IQ-scheme for deletion of constraints, we only have to tag the constraints when they take effect during execution.

Figure 6 illustrates the approximate IQ-scheme pictorially. Suppose $Q_i = \leftarrow B, C_i$ and $Q_{i+1} = \leftarrow B, C_{i+1}$, where $C_{i+1} = C_i \setminus \{c\}$. The branch ending with segments 1, 2, and 3 corresponds to the oracle $O(C_i)$. Without loss of generality, we assume that the first solution of Q_{i+1} occurs to the "left" of Q_i, thus ending with segments 1, 2, and 4. Therefore, the node connecting branch segments 2, 3, and 4 is the theoretically best re-start point. As explained, the IQ-scheme may not be able to locate the perfect re-start point. Suppose that the IQ re-start point is at the node labeled N_{IQ}. If the effective point $\mathcal{E}_c$ of constraint c occurs in segment 1, then we have to further unwind to the node N'_{IQ} before it is safe to remove c from the constraint store. If $\mathcal{E}_c$ occurs in either segment 2 or 3, then we can remove c from the constraint store at N_{IQ}.

7.4 Preliminary Results

We present here some results of applying the IQ-scheme to three well-known constraint satisfaction problems to illustrate the performance of the scheme: graph-coloring, n-queens and car-sequencing. The results obtained are summarized in tables 1, 2 and 3 respectively. In these tables, n_{VHLP} is the number of derivation steps needed to re-execute from the root of the tree to the restart point N_{IQ}[4] and n_{IQ} is the number of unwinding steps to reach N_{IQ}. It is shown in these tables that the performance of the IQ-scheme is better than that of the VHLP-scheme.

The graph coloring problem problem is to find an assignment of colors to the nodes in a graph such that the two nodes do not have the same color if they are connected by an edge. The graph consists of 11 nodes and there are 4 different colors to be used. The program first declares for each node a d-variable with an initial domain that contains all four colors. These d-variables are named `X1`, `X2`, ..., `X11` and the colors `1`, `2`, `3` and `4` in the following presentation. Then the program sets up disequality constraints for each edge. After the solution to the initial query is found, we repeatedly perform addition and deletion of constraints to test the performance of the IQ-scheme. The results are summarized in table 1.

We also evaluate the performance of the IQ-scheme using a 10-queens problem, in which 10 queens are to be placed on a 10×10 chessboard such that they do not attack one another. Each of the domain variables `Q1`, `Q2`, ..., `Q10` denotes the column position of the queen placed in row 1, 2, ...,

[4]In these particular programs, $N_{IQ} = N'_{IQ}$.

Query	Increment	Constraint	n_{VHLP}	n_{IQ}
Q_1	-	-	-	-
Q_2	add	`X3=2`	328	9
Q_3	add	`X9=4`	335	2
Q_4	add	`X10`$\neq$`2`	336	1
Q_5	delete	`X9=4`	335	0
Q_6	add	`X2`$\neq$`3`	328	7
Q_7	add	`X9=4`	335	1
Q_8	delete	`X2`$\neq$`3`	328	5
Q_9	delete	`X9=4`	332	0
Q_{10}	add	`X2`$\neq$`3`	328	7
Q_{11}	add	`X6=3`	331	5
Q_{12}	delete	`X2`$\neq$`3`	328	5

Table 1: Benchmark results for Graph Coloring Problem

10, respectively. Disequality constraints are set up such that $\texttt{Q}i \neq \texttt{Q}j$ and $\texttt{Q}i \neq \texttt{Q}j \pm |i-j|$ for all $i, j \in \{1, \ldots, 10\}$. Addition and deletion of constraints are performed after the first solution is found and the results are summarized in table 2.

The car-sequencing problem is the third problem we use to evaluate the performance of the IQ-scheme. The problem instance that we use is an IFD-Constraint Pandora re-implementation of the problem instance presented in

Query	Increment	Constraint	n_{VHLP}	n_{IQ}
Q_1	-	-	-	-
Q_2	add	`Q3`$\neq$`6`	263	4
Q_3	add	`Q1=2`	261	6
Q_4	delete	`Q3`$\neq$`6`	262	4
Q_5	add	`Q6`$\neq$`2`	266	0
Q_6	add	`Q6`$\neq$`1`	266	0
Q_7	delete	`Q1=2`	266	0
Q_8	delete	`Q6`$\neq$`1`	266	0
Q_9	delete	`Q6`$\neq$`2`	266	0

Table 2: Benchmark results for 10-queens Problem

a previous paper [8]. In the problem we are to determine a sequence of 10 cars to be manufactured. These cars belong to six different classes. Each class of car requires 1 to 3 out of 5 available options. The assembly line has a capacity constraint for each of these options, which varies from 1 out of 5 to 2 out of 3. The main domain variables are `C1`, `C2`, ..., `C10` such that `C`i ($1 \leq i \leq 10$) denotes the class (1, 2, ..., or 6) of the ith car in the sequence. Again, we perform some constraint addition and deletion operations after the first solution is found. The results are presented in table 3.

The main reason for the difference between the performance of these two schemes is largely due to a unique characteristic of the Andorra class of con-

Query	Increment	Constraint	n_{VHLP}	n_{IQ}
Q_1	-	-	-	-
Q_2	add	`C2=3`	127	1
Q_3	add	`C1`$\neq$`1`	126	3
Q_4	add	`C4`$\neq$`1`	130	0
Q_5	delete	`C2=3`	126	4
Q_6	delete	`C1`$\neq$`1`	126	4

Table 3: Benchmark results for Car Sequencing Problem

current logic programming languages: that is, there are usually a large number of deterministic derivation steps between two consecutive choice points. The overhead of unwinding one step backward is clearly much less than replaying these large number of derivation steps. In our graph-coloring program, the execution proceeds for 327 derivation steps before the first choice point is created. Similar phenomenon is observed in the 10-queens problem (261 derivation steps) and the car-sequencing problem (126 derivation steps). The advantages of the IQ-scheme is especially noticeable in concurrent constraint logic programming languages, in which most of the choice points are created for the "labeling" process, while only few choice points are set up during constraint establishment.

8 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 to *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 family of languages. 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 IFD-Constraint Pandora has been built using CHIP to demonstrate the feasibility of our approach. Preliminary results are encouraging. 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] F. Benhamou and W. J. Older. Applying interval arithmetic to real, integer and boolean constraints. *Journal of Logic Programming*, 32(1):1–24, July 1997.

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

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

[5] 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.

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

[7] J. Crammond. The abstract machine and implementation of parallel parlog. *New Gener. Comput.*, 10:385–422, 1992.

[8] M. Dincbas, H. Simonis, and P. Van Hentenryck. Solving the car-sequencing problem in constraint logic programming. In *Proceedings of the European Conference on Artificial Intelligence*, pages 290–295, 1988.

[9] 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.

[10] F. Fages, J. Fowler, and T. Sola. A reactive constraint logic programming scheme. In L. Sterling, editor, *Logic Programming: proceedings of the Twelfth International Conference on Logic Programming*, pages 149–163, Tokyo, Japan, June 1995. MIT Press.

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

[12] R Helm, T. Huynh, K. Marriott, and J. Vlissides. An Object-Oriented Architecture for Constraint-Based Graphical Editing. In C. Laffra, E.H. Blake, V. de Mey, and X. Pintado, editors, *Object-Oriented Programming for Graphics*, Focus on Computer Graphics, Tutorials and Perspectives in Computer Graphics, pages 217–238. Springer-Verlag, 1995.

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

[14] 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.

[15] J.H.M. Lee and M.H. van Emden. Interval computation as deduction in CHIP. *Journal of Logic Programming*, 16(3 & 4):255–276, 1993.

[16] H. F. Leung. *Distributed Constraint Logic Programming*, volume 41 of *World Scientific series in computer science*. World Scientific, Singapore, 1993.

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

[18] A. K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1977.

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

[20] 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.

[21] Lee Naish. An introduction to Mu-Prolog. Technical Report TR-82/2, Department of Computer Science, Melbourne University, 1982. Revised version July 1983.

[22] 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. Springer-Verlag.

[23] E.Y. Shapiro. Concurrent Prolog: a progress report. *IEEE Computer*, 8(4):44–58, 1986.

[24] R.V. Southwell. *Relaxation Methods in Theoretical Physics*. Oxford University Press, 1946.

[25] I.E. Sutherland. *SKETCHPAD: a Man-Machine Graphical Communication System*. PhD thesis, MIT Lincoln Labs, Cambridge, MA, 1963.

[26] The COSYTEC Team. *CHIP V4 User Manuals*, 1993.

[27] T.Huynh and K. Marriott. Incremental constraint deletion in systems of linear constraints. *Information Processing Letters*, 55(2):111–115, July 1995.

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

[29] 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.

[30] 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.

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

[32] P. Van Hentenryck. *Consistency Techniques in Logic Programming.* PhD thesis, University of Namur, Belgium, 1987.

[33] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming.* The MIT Press, 1989.

[34] P. Van Hentenryck and Y. Deville. The cardinality operator: A new logical connective for constraint logic programming. In *Proceedings of the Eighth International Conference on Logic Programming*, pages 745–759, 1991.

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

[36] P. Van Hentenryck, V. Saraswat, and Y. Deville. Design, implementation, and evaluation of the constraint language cc(FD). Technical Report CS-93-02, Computer Science Department, Brown University, Providence, RI, USA, 1993.

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

```
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 2: A Simple Scheduling Example in Constraint Pandora.

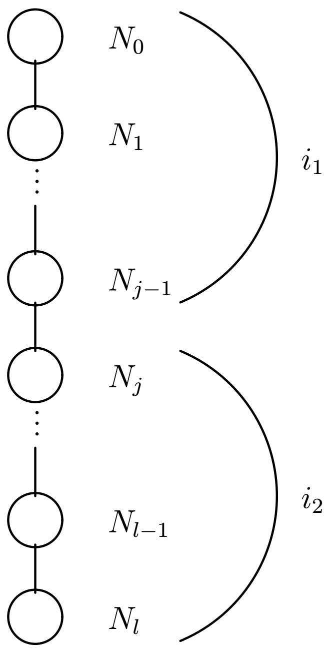

Figure 3: Existence of N_j

```
queens(L) :- [X1,X2,X3,X4] <= L :
    X1 :: 1..4, X2 :: 1..4, X3 :: 1..4, X4 :: 1..4,
    constrain([X1,X2,X3,X4]), indomain(X1),
    indomain(X2), indomain(X3), indomain(X4).

constrain(L) :- [] <= L : true.
constrain(L) :- [X|T] <= L :
    safe(X, T, 1), constrain(T).

safe(X, L, N) :- [] <= L : true.
safe(X, L, N) :- [Y|T] <= L :
    noattack(X, Y, N), N2 is N + 1, safe(X, T, N2).

noattack(X, Y, N) :-
    X ≠ Y, X ≠ Y + N, X ≠ Y - N.

indomain(X) :- X = 1.
indomain(X) :- X = 2.
indomain(X) :- X = 3.
indomain(X) :- X = 4.
```

Figure 4: An IFD-Constraint Pandora program for the 4-queens problem.

Let A denote the *active set* of constraints and P the *passive set* of constraints

```
1.  Initialize $A$ to contain all new constraints
2.  Initialize $P$ to contain all constraints already in the store
3.  while $A$ is not empty
4.      Remove from $A$ a constraint $p$, the domains of its associated variables are $\vec{D}$
5.      $\vec{D'} = \texttt{REVISE}(p, \vec{D})$
6.      if any domain in $\vec{D'}$ is $\emptyset$ then
7.          Exit with failure
8.      else
9.          if $\vec{D} \neq \vec{D'}$ then
10.             foreach d-variable $X^{D_i}$ in $p$ such that $D_i \neq D'_i$
11.                 Generate the substitution $\theta = \{X^{D_i}/Y^{D'_i}\}$
12.                 Apply $\theta$ to $A$ and $P$
13.                 foreach constraint $q$ in $P$
14.                     if $q$ contains $X^{D_i}$ then
15.                         Move $q$ from $P$ to $A$
16.                     endif
17.                 endforeach
18.             endforeach
19.         endif
20.     endif
21.     Add $p$ to $P$
22. endwhile
```

Figure 5: A Relaxation Algorithm for Constraint Assimilation

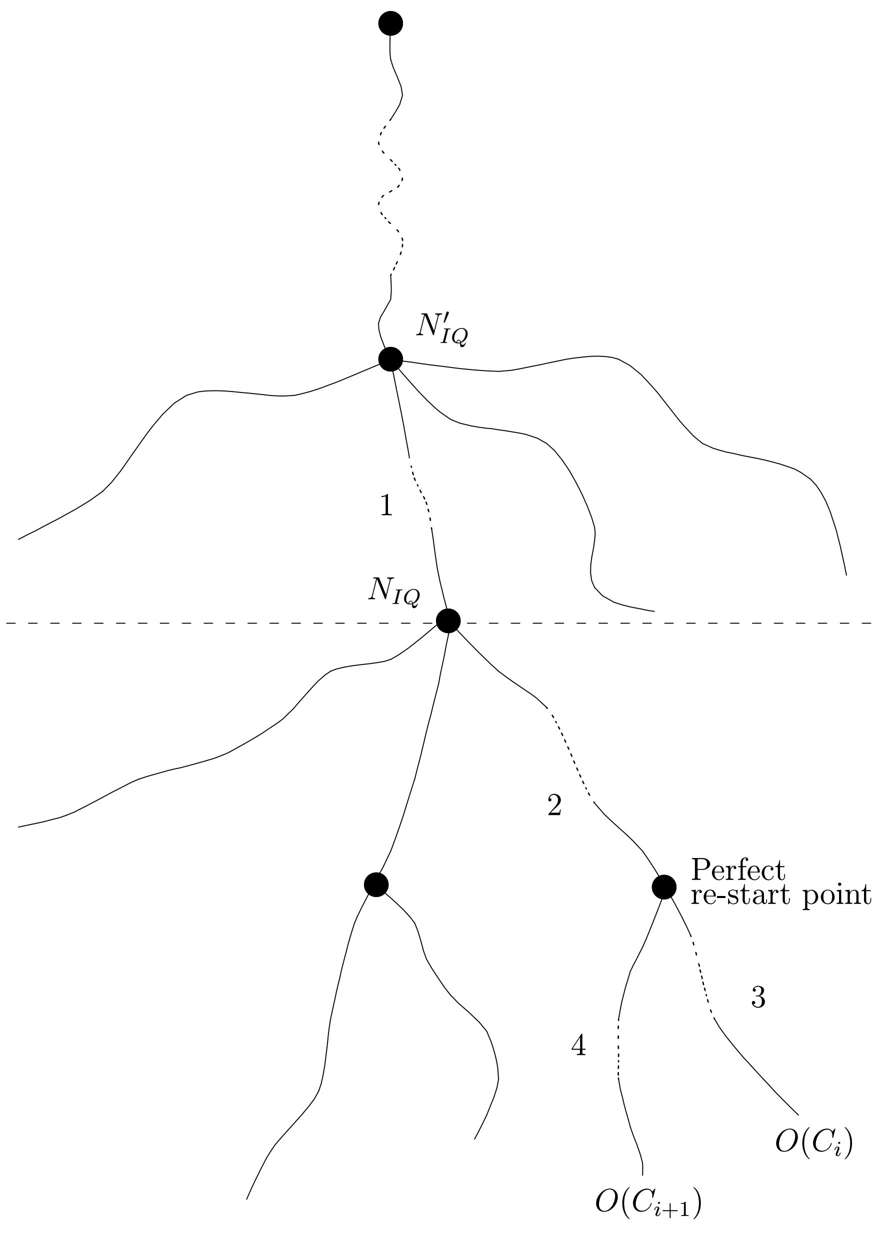

Figure 6: The Approximate IQ-Scheme for Deletion of Constraints